CHAPTER 4

GEOMETRY SUBSYSTEM

The Geometry Subsystem is responsible for performing the various transformations and lighting computations on world coordinate data received from the host. It then performs any necessary calculations to reduce the subsequent vertex data into points, lines or spans to pass to the Raster Subsystem. It is also responsible for controlling the interface between the host and the other MGR Subsystems.

The Geometry Subsystem is the heart of the MGR adapter and is responsible for performing the geometry calculations and is also responsible for controlling the operation of the Raster Engine in the Raster Subsystem. The Geometry Engine is a microcode controlled floating point processor that is used to perform the geometry calculations as well as the lighting calculations. The two main components of the Geometry Engine are the microcode sequence controller chip called the HQ chip and the Weitek 3132 Floating Point Data Path chip which does the floating point multiplies and adds. These components along with the other components will be discussed in greater detail in later paragraphs.

The following paragraphs will describe the external interfaces, the major components, the registers, the basic operation and the programming'considerations of the Geometry Subsystem.

External Interfaces

The following paragraphs describe the external Interfaces that the Geometry Subsystem has with the Host Interface Subsystem, the Raster Subsystem and the Display Subsystem.

Host Interface Subsystem Interface

The Geometry Subsystem has an interface with the Host Systemvia the Host Interface Subsystem. The SGI Private bus connects these two subsystems and is used to transfer all data to and from the host system. The Geometry Subsystem uses the 10 Mhz multi-phase clocks provided by the Host Interface Subsystem for its timing.

Raster Subsystem Interface

The Geometry Subsystem has a register pointer and a data bus which are used to access the Raster Engine in the Raster Subsystem. The Geometry Subsystem also controls the addressing for host accesses of the Cursor Chips in the Raster Subsystem. The Geometry Subsystem controls DMA transfers between the Raster Subsystem, the Geometry Subsystem and the Host System.

Display Subsystem Interface

The Geometry Subsystem controls the addressing and data transfers between the Host System and the registers in the various components of the Display Subsystem.

Major Components

The major components of the Geometry Subsystem are shown in the block diagram of Figure 4.1. The **HQ1** chip controls the address decoding for the host and acts as a microcode sequencer. The Weitek 3132 chip performs the floating point calculations and is controlled by the **HQ1.** The microcode **RAM** holds the microcode, while the microcode data RAM holds constants, data variables and data buffers. These four components form the Geometry Engine **5 (GE5)** which performs the geometry calculations, lighting calculations and various other functions.

To execute graphics commands the host sends **graphics** command tokens and data to the Geometry Engine. The Tag FIFO and Data FIFO provide a buffer between the host and Geometry Engine speeds. The host can continue to send graphics tokens while the Geometry Engine is still executing previously sent tokens. The Tag FIFO receives the command tokens from the host and passes the command tokens to the **HQ1** chip which uses the token as an index into a microcode branch table. This causes the appropriate microcode routine to be executed for each token. These tokens instruct the microcode to perform the desired operations. The Data FIFO receives the data parameters associated with the various tokens. The microcode executes instructions which read the dataFIFO and transfers the data into either the Weitek 3132 chip or into the GE5 data RAM.

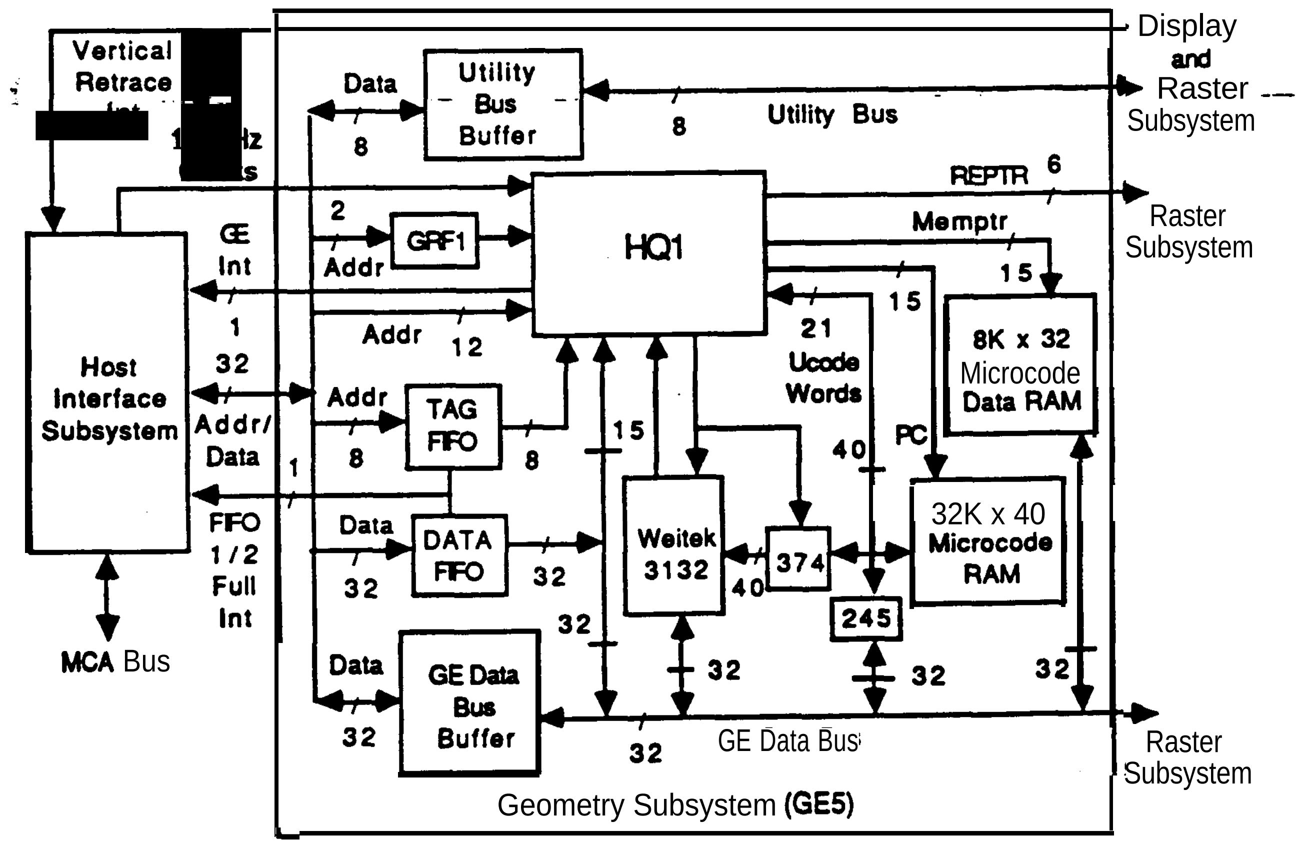

Figure 4.1 Geometry Subsystem Block Diagram

The **GRF1** chip is used to add some additional capabilities to the **HQ1** chip. The GE Data Bus **buffers** and the Utility Bus buffer provide the necessary buffering required for these two data busses. **The** following paragraphs describe the Geometry Subsystem components.

HQ1 Chip

The Head Quarters **(HQ1)** chip is a proprietary Silicon Graphics design and is the main control element in the Geometry Subsystem. The **HQ1** is responsible for controlling the geometry engine and the various data transfers among the **MGR** hardware components. The **HQ1** chip has six functional units as shown in Figure 4.2.

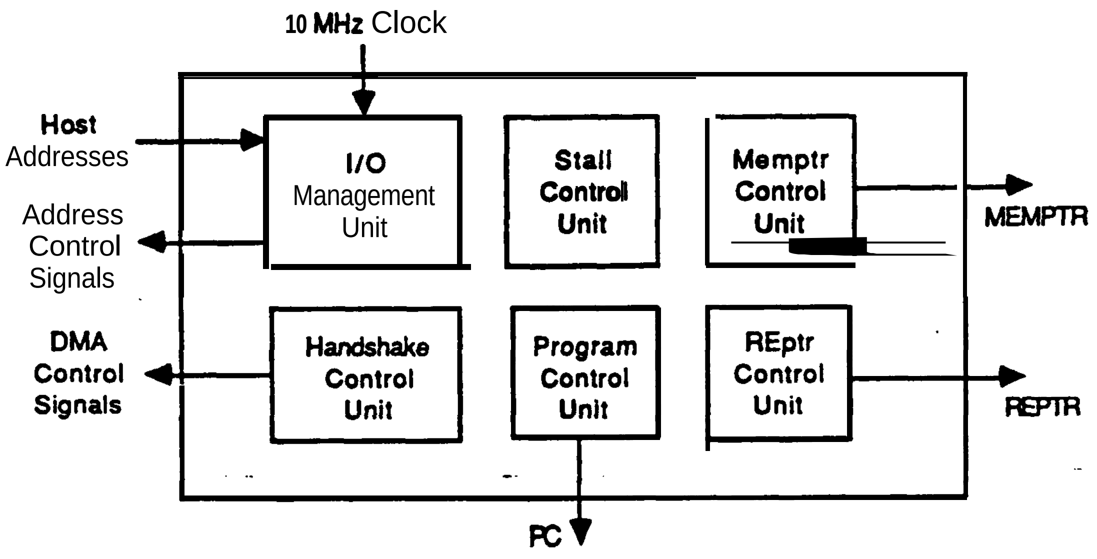

Figure 4.2 **HQ1** Functional Block Diagram

The following paragraphs describe these six functional units.

I/O Management Unit

The **I/O** management unit is responsible for providing the address decoding and bus control for hardware accesses by the Host Interface Subsystem and the Geometry Engine. The host address decoding is shown in Table 4.1. The host can **access** the following hardware components:

- **HQ1** registers
- Tag and Data FIFO
- **Microcode** Code **RAM**
- Microcode Data RAM
- 2 Finish Flags
- 2 Cursor Chips
- 5 **XMAP2** and **5** Color Maps or 5 **XPC1** Chips
- 3 RAMDAC Chips or 1 RGB RAMDAC Chip
- Display Registers

Table 4.1 Host Interface Address Decoding

ADDRESS RANGE	HQMMSB	SRC/DEST	OPERATION	DATA
0x0000 - 0x03FC	0	Ucode RAM low 32 bits	RW	LS 32 bits
0x0000 - 0x03FC	1	Ucode RAM high 8 bits	RW	MS 8 bits
0x0400 - 0x041C	1	XMAP2 channel 0	RW	8 bits
0x0420 - 0x043C	1	XMAP2 channel 1	RW	8 bits
0x0440 - 0x045C	1	XMAP2 channel 2	RW	8 bits
0x0460 - 0x047C	1	XMAP2 channel 3	RW	8 bits
0x0480 - 0x049C	1	XMAP2 channel 4	RW	8 bits
0x04A0 - 0x04BC	1	XMAP2 broadcast	W	8 bits
0x04C0	1	Display Reg 0	RW	8 bits
0x04E0	1	Display Reg 1	RW	8 bits
0x0500 - 0x050C	1	Red DAC	RW	8 bits
0x0510 - 0x0514	1	XPC1 Channel 0	RW	8 bits
0x0520 - 0x052C	1	Green DAC	RW	8 bits
0x0530 - 0x0534	1	XPC1 Channel 1	RW	8 bits
0x0540 - 0x054C	1	Blue DAC	RW	8 bits
0x0550 - 0x0554	1	XPC1 Channel 2	RW	8 bits
0x0560 - 0x056C	1	Cursor chip 0	RW	8 bits
0x0570 - 0x0574	1	XPC1 Channel 3	RW	8 bits
0x0580 - 0x058C	1	Cursor chip 1	RW	8 bits
0x0590 - 0x0594	1	XPC1 Channel 4	RW	8 bits
0x05A0	1	Display Reg 4	RW	8 bits
0x05B0 - 0x05B4	1	XPC1 Broadcast	RW	8 bits
0x05C0	1	Display Reg 3	RW	8 bits
0x05D0 - 0x05DC	1	RGB RAMDAC	RW	8 bits
0x05E0	1	Display Reg 2	RW	8 bits
0x0640 - 0x780	1	HQ1 Command	RW	15 bits
0x0800 - 0x0BFC	X	FIFO	RW	32 bits
0x0C00 - 0x0DFC	X	HQ MAR	W	none
0x0E00 - 0xE04	X	HQ MAR MSB	W	none
0x1400 - 0x17FC	0	Ucode Data RAM	RW	32 bits
0x2000 - 0x2004	X	Finish flags	RW	32 bits

The I/O management unit uses two registers to help control the MGR addressing. The 1 bit HQ Middle Address Register Most Significant Bit (HQMMSB) Register is used to control the address decoding. It is set to either zero or one as shown in Table 4.1. This register's name derives from an earlier implementation when it was bit 7 of the HQ middle address register (MAR). This is no longer the **case,** and the HQMMSB register is addressed independently from the HQ MAR. It can be thought of as the Address Selection Control Register. This register should be set to zero when addressing the low 32 bits of the code RAM or when accessing the data RAM. When accessing the FIFO, the Finish Flags, the HQ MAR register or this register itself the value in the HQMMSB register does not matter. For all other hardware components shown in Table 4.1 the HQMMSB must be set to one before the components can be properly accessed.

The **HQ1** uses the HQ MAR register as a page selection register when accessing the microcode code and data **RAMs.** The MAR register is used to provide the upper 7 bits of the microcode code or data RAM address. The bwer 8 bits of the microcode code or data RAM address provide the word offset into the page selected by the MAR register. The word offset is obtained from bits 2-9 of the host address. These bits are combined with the page address in the MAR register to form the microcode **code** or data RAM address. The 15 bits **allow** 32K words of microcode code or data RAM to be addressed by the host. The **MGR** adapter has 32K words of code RAM so all 15 bits of the address are valid when addressing the code RAM. The **MGR** adapter has **only** 8K words of data RAM so only 13 bits are valid when addressing the data RAM. This means that only 5 bits in the MAR register are **valid** when addressing the data RAM.

The **I/O** management unit decodes the host addresses shown in Table 4.2 and generates the necessary control signals to access the specified Geometry Subsystem components. The host can access the microcode **code** RAM, the microcode data RAM, the HQ Middle Address **Register** (MAR), the HQ MAR Most Significant Bit (HQMMSB) Register and the data and tag FIFO. The host can also issue commands to the **HQ1** to clear the stall condition and to clear the GE interrupt. The host can read the microcode Program Counter (PC) register.

Table 4.2 **HQ1** Address Map

ADDRESS RANGE	HQMMSB	SRC/DEST	OPERATION	R/W	DATA
0x0000 - 0x03FC	0	Ucode RAM low 32	RW Ucode RAM	RW	32 bits
0x0000 - 0x03FC	1	UcodeRAMhigh8	RW Ucode RAM	RW	8 bits
0x0840	1	HQ1	clear stall	W	none
0x0740	1	PC in HO1	Read HQ PC	R	15 bits
0x0780	1	HQ1	clear GE int	w	none
0x0800 - 0x0BFC	X	FIFO	FIFO RW	RW	32 bits
0x0C00 - 0x0DFC	X	HQ MAR	Write HQ MAR	W	none
0x0E00 - 0x0E04	X	HQ MAR MSB	Write HQMMSB	W	none
0x1400 - 0x17FC	0	Ucode Data RAM	RW Data RAM	RW	32 bits
0x2000	X	Finish Flag 0	Read Finish 0	RW	32 bits
0x2004	X	Finish Flag 1	Read Finish 1	RW	32 bits

Two finish flags are provided to synchronize host accesses to the microcode data RAM with the microcode execution and to synchronize the start of **DMA** operations to or from the adapter. The host **can** read and write the finish flags while the microcode can only write to the finish flags.

The host can only access the GE5 data RAM or code RAM while the microcode is in a stalled state. Finish flag 0 is provided to synchronize the host accesses to the microcode data RAM with the microcode. The host systems sends a **GE_FINISH0** token down the FIFO to the microcode to **command** it to set the finish flag when **it** finishes executing the tokens already in the **FIFO. The host then** continuously reads address 0x2000 for **finish** flag 0 until finish flag 0 is set by the microcode. When the microcode finishes its current operations and the FIFO **is** empty it sets finish flag **0 and** does a fetch from the FIFO which causes it to stall until the host sends another token. The host system can then read or write the microcode data RAM without Interfering with the microcode.

Finish Flag 1 is provided to synchronize the start of DMA operations between the host and the MGR adapter. When the host sends a token to the microcode that will require **DMA** operations the microcode will setup for the DMA and then set Finish F&g **1.** The host would poll address **0x2004 until** it is set by the microcode and then the host would start the DMA operation. The use of the **finish** flags will be discussed in greater detail later.

Handshake Control Unit

The handshake control unit is used to provide the necessary hardware handshake signals between the Host Interface Subsystem, the Raster Engine (RE) in the Raster Subsystem, the microcode code and data RAM and the XMAP2 or **XPC1** chips in the Display Subsystem. It stalls the **GE5** when it is appropriate and passes other handshakes through for accesses such as host to XMAP or host to RE.

The handshake control unit allows two types of transfers between the Host Interface Subsystem and the other MGR Subsystems. These two types of transfers are single word transfers and **DMA transfers.**

The host can access the following components using single word transfers:

- **HQ1** PC and **HQ1** commands
- tag/data FIFO
- microcode code RAM
- microcode data RAM
- five **XMAP2s** or five **XPC1s**
- an RGB RAMDAC or three **RAMDACs**
- five Color Maps
- **two** cursor chips
- five display registers

The following DMA transfers can occur:

- between the host and the microcode data RAM
- between the host and the Raster Engine
- between the Raster Engine and the microcode data RAM

These data transfers are described in later paragraphs in the appropriate chapters.

Stall Control Unit

The stall control unit is responsible for stalling the **GE5** when a condition arises that requires it to be stalled. The full stall keeps the **I/O** Management unit in its current state and also keeps the PC, the MEMPTR and the REPTR from changing. The **clock** to the Weitek 3132 is fully suppressed and literally the entire GE5 is stalled. The host can still perform single word transfers as described above.

The following conditions result in a full stall:

Reset - Whenever the **HQ1** is reset it goes into a full stall. After the **HQ1** has been reset it needs to be initialized. The host downloads the microcode code and data and then writes to the clear stall address to start the GE5 running. The host software must be aware of this type of stall since **it** has to issue the clear stall **HQ1** command.

Burst DMA transfer - Whenever the **HQ1** detects a host delay, an RE delay or a host burst disable condition it will stall the GE5 until the appropriate handshake conditions exists to continue the DMA transfer. At that time the **HQ1** will clear the full stall and the transfer will continue. This type of stall is transparent to the host software and would only be somewhat evident if the DMA never completed.

Stall Microinstruction - The **GE5 microcode** may execute a stall instruction which will put the **HQ1** into a full stall until the host writes to the clear stall address. This can be used to synchronize the microcode with the host for passing data. The host software must be aware of the instances where the microcode would issue a stall microinstruction since it would have to issue the clear **stall HQ1** command.

Read from Empty FIFO - When the **GE5** microcode is executing a fetch instruction or a read from data FIFO instruction, the **HQ1** will go into a full stall in the middle of the read or fetch microinstruction if the FIFO is empty. The stall condition will continue until data actually becomes available. This is a stall condition that is frequently encountered and can be used by the host software to access the GE5 data RAM. The **GE_FINISH0** token is used by the host to guarantee that the microcode has processed all of the data in the FIFO and is stalled at a fetch instruction waiting for the next token to **be** sent by the host software. The host software can then safely **access** the GE5 data RAM **before** sending the next command token.

Raster Engine not **ready** - The **HQ1** can load the Raster Engine (RE) registers when the appropriate load enable **signals** are active. If the GE5 microcode attempts to write to an RE register when the load enable signals are not active the **HQ1** will go into a full stall until the load enable signals becomes active. This is a temporary handshake stall between the GE5 and the RE2 and is totally transparent *to* the host software.

PC Control Unit

The Program Counter (PC) control unit acts as a microcode sequencer for the **GE5.** The control unit controls the microcode instruction execution sequencing through up to 32K words of microcode. The PC control unit allows branching and conditional jumps. It uses a 15 bit wide PC to access the microcode code RAM. Microcode **instructions** are read from the address pointed to by the PC and are executed by the **HQ1** and the Weitek 3132 chip.

When the host needs to access the **microcode** code RAM it must **access** the bw 32 bits and the upper 8 bits of the 40 bit word separately. The upper 7 bits of the microcode word address are loaded into the MAR register. The tow 32 bits of the microcode code word are accessed by setting the HQMMSB register to 0. A read or write is then performed and bits 9-2 of the host address are placed in the PC bw byte and the MAR register contents are transferred into the PCs upper 7 bits. The 15 bit PC address is then used to access the bw 32 bits of the microcode code RAM. The same procedure is used to access the upper 8 bits except that the HQMMSB register must be set to 1.

MEMPTR Control Unit

The MEMPTR is used as a data pointer for accessing words in the microcode data RAM. The MEMPTR is 14 bits wide allowing a maximum microcode data RAM size of 16K words. The MGR has only 8K words of microcode data RAM and each microcode data RAM word is 32 bits wide. The MEMPTR is loaded from the microcode instructions to perform microcode data RAM accesses by the microcode.

When the host wants to access the microcode data RAM it sets the HQMMSB to 0. It then loads the MAR register with the upper 6 bits of the microcode data RAM address and does the read or write to the appropriate microcode data RAM address. Bits 9-2 of the host address are placed in the MEMPTR tow byte and the MAR register contents are transferred into the MEMPTRs upper 6 bits. The 14 bit MEMPTR address is then used to access the microcode data RAM.

REPTR Control Unit

The REPTR is used as an address pointer to the RE registers. It is used by the GE5 microcode to load the RE registers._The REPTR is not accessible by the host which means that the host cannot directly access the RE registers. A microcode token GE_LOADRE allows the host to load the RE registers with an assist from the microcode. The host generally does not access the RE register directly and the exceptions will be discussed later.

This concludes our journey inside the HQ1 chip. We will now continue looking at the other hardware components of the Geometry Subsystem as previously shown in Figure 4.1.

FIFO

The FIFO is used to even the fbw of commands from the host and the subsequent command execution by the GE5. The FIFO allows the host to continue to send graphics commands and data while the GE5 is still executing a previously received token. The FIFO has space for 512 entries of 40 bits each. The host writes a 32 bit value to a range of FIFO addresses. The 32 bits of data are stored in the Data FIFO and 8 bits from the host FIFO address are stored in the Tag FIFO. Bits 2 through 9 of the host address are placed in the Tag FIFO. The FIFO therefore occupies a 1K byte range of addresses. The value placed in the Tag FIFO represents a command token to the microcode. When the FIFO becomes half full an interrupt is generated so that the host can avoid overflowing the FIFO.

Tag FIFO

The tag FIFO gets host interface address bits 2 through 9 placed in it when the host writes to the FIFO address. The tag FIFO contains the microcode command tokens which are used as an index into the microcode command branch table. Sixteen of the 256 possible tokens are used to switch to a new graphics context. The other 240 possible tokens are used in conjunction with the FETCH instruction of the microcode to execute microcode commands.

Data FIFO

The Data FIFO contains the data parameters passed to the microcode for the various command tokens. It can contain 512 words of 32 bits each. When the data FIFO is read, the 32 bit word is placed on the GE5 data bus where it is accessed by the GE5.

Weitek 3132

The Weitek 3132 is a floating point data path chip. It provides pipelined floating point multiply and add capability. It also provides 32 working registers to use during the floating point operations. The 3132 is the heart of the GE5 and is used to perform all geometry and lighting transformations.

Microcode Code RAM

The microcode code RAM contains up to 32 K words of microcode. Each word is 40 bits wide and is accessed using the PC in the HQ1.

Microcode Data Ram

The microcode data RAM contains 8K words of data constants and variables. Each word is 32 bits wide and is accessed using the MEMPTR in the HQ1.

Utility Bus Buffer

This is an eight bit transceiver which allows data to pass between the host bus and the utility bus. Only the low byte of the local bus are passed on to the utility bus. The address decoding and control signal generation are handled by the HQ1.

GE Data Bus Buffer

This is a 32 bit buffer which allows data to be passed between the GE5 bus and the host interface. Single word transfers use this data path to transfer data between the host and the microcode code RAM, the microcode data RAM and the HQ1 PC register. DMA transfers between the host and the microcode data RAM or between the host and the Raster Engine also us8 this data path.

GRF1 Gate Array

This chip is used to provide the two finish flags and to correct some other minor design flaws in the HQ1 chip.

Registers

The Geometry Subsystem contains the following registers in the **HQ1** chip and the Weitek 3132 chip:

- **HQ1** Registers
  - Middle Address (HQMAR) register
  - Middle Address Most Significant Bit (HQMMSB) register
  - Program Counter (PC) register
  - Previous PC register
  - Memptr Register
  - Temp Memptr Register
  - **REptr** Register
  - DMA Count Register
- Weitek 3132 Registers
  - 32 Data Registers
  - 3 Temporary Result Registers

These registers are described in the following paragraphs.

HQ Middle Address (MAR) Register

The HQMAR register is the page address register used by the host software to access the 256 word pages of GE5 code and data RAM. The register is 7 bits wide and provides the upper 7 bits of the microcode code and data RAM address. The full address of the word in the code or data RAM is formed by taking the lower 8 bits from bits 2-9 of the host address bus and adding the upper seven bits of the HQMAR register to form the 15 bit address. The us8 of this register wilt be described more completely in the PC and Memptr register diissions later. The HQMAR register can only be written by the host and cannot be read by the host. This means that whenever the host software changes the value in the HQMAR register it must us8 the GE_HQMSAV token to save the value In the GE5 data RAM location HQMSAV. When a context switch is done this saved HQMAR value is part of the saved context. When the context is later restored the saved value of the HQMAR register is used by the host software to restore the HQMAR register. Figure 4.3 shows the format of the HQMAR register.

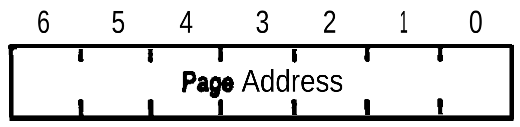

Figure 4.3 HQ Middle Address Register

Bits 6-O : Page Address (Write Only) - Contains 7 bit page address value used for accessing microcode code or data RAM.

HQ Middle Address Most Significant Bit (HQMMSB) Register

The HQMMSB register is a 1 bit wide register that is used by the HQ1 chip as an address control bit to control the addressing of the hardware components shown in Table 4.1. The use of this register is shown in Tables 4.1 and 4.2. This register must be set to 0 when accessing the low 32 bits of the GE5 code RAM or the GE5 data RAM. The bit must be set to one when accessing the upper 8 bits of the GE5 code RAM or when issuing an HQ1 command. It must also be set to one when accessing the the hardware components in the Display Subsystem and the Cursor chips in the Raster Subsystem. When accessing the HQMAR register, the HQMMSB register, the FIFO and the Finish Flags the value in the HQMMSB register can be either a zero or a one. The host can oniy write this register so the host software must make sure it manages the value in this register as needed for each hardware component access. The host software shouid always clear the HQMMSB register when it is finished accessing a hardware component. The value of this register is determined by the value of the host address bit 2. This means that this register is cleared by writing to address 0xE00 and is set by writing to address 0xE04. The value which is written to these addresses do not affect the clearing or setting of the HQMMSB register.

The name of this register is from an earlier implementation when it was bit 7 of the HQ MAR register. It is now accessed independently of the HQ MAR register. The format of the register is shown in Figure 4.4.

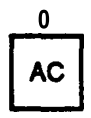

Figure 4.4 HQ MAR Most Significant Bit Register

Bit 0 : Address Control (AC) Bit (Write Only). Contains a bit used by the HQ1 in decoding host addresses.

HQ Program Counter (PC) Register

The PC register is a 15 bit wide register which is used by the **HQ1** to access the GE5 microcode code RAM as it sequences through the microinstructions. When the **HQ1** chip is reset as part of the MGR adapter reset the value in the PC register is undefined. The host software &es not have direct write **access** to the PC register but it can load a value into the PC register indirectly by loading a page address into the HQMAR register **and** then reading or writing a word in the code RAM. The tower 8 **bits** of the PC are **loaded** from the host address bits 2-9 and the upper 7 bits of the PC register are **loaded** from the HQMAR register. The HQMMSB register controls whether the bwer 32 bits or the upper 8 bits of the code RAM are accessed. This technique is used by the host software to download the microcode to the adapter.

Once the microcode is downloaded the host bads the HQMAR register with zero and reads code word zero to cause the PC value to be set to zero. The clear stall command can then be issued to cause the microcode to begin executing the microcode hard initialization code. At this point the **HQ1** controls the value in the PC register as it executes the microcode instructions. The host software can read the PC register by issuing the **HQ1** read PC command. The format of the register is shown in Figure 4.5.

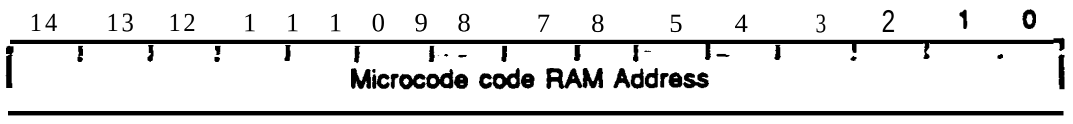

Figure 4.5 HQ Program Counter Register

Bits 14-O **:** Microcode code RAM address. Contains the address of the microcode word which will be accessed by the **HQ1** or by the host system.

HQ Previous PC Register

The Previous PC register is a 15 bit wide register which is used by the **HQ1** to save the value int the HO PC register. Normally the use of the previous PC register Is totally transparent to the host software. During the execution of the GE_CTXO or **GE_CTX1** tokens the **HQ1** saves the current PC **register** value. The context switch microcode saves the value In the previous PC register in the GE5 data RAM location PCSAVE. This value is part of the saved context data that is saved **in** the host data RAM. When the host software later restores the saved context **it** uses the value In the PCSAVE **location** while doing the context restore operation. The use of the PCSAVE value will be discussed in the discussion of the **GE_CTX1** to&en. The format of the register is shown In Pigure 4.6.

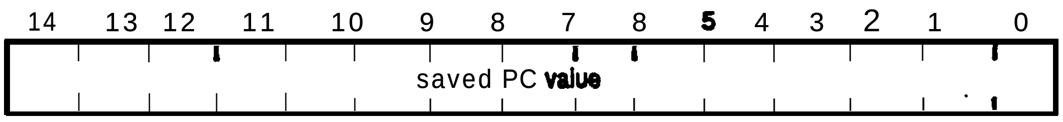

Figure 4.6 HQ Previous PC Register

Bits 14-O **:** Saved PC Value. Contains the saved value from the PC register.

HQ Memptr Register

The Memptr register is a 14 bit wide register which is used by the HQ1 to access the GE5 microcode data RAM. When the HQ1 chip is reset as part of the MGR adapter reset the value in the Memptr register is undefined. The host software does not have the ability to read the Memptr register. The host software also cannot directly write to the Memptr register but it can load a value into the Memptr register indirectly by loading a page address into the HQMAR register and then reading or writing a word in the data RAM. The lower 8 bits of the Memptr are loaded from the host address bits 2-9 and the upper 6 bits of the Memptr register are loaded from the HQMAR register. The HQMMSB register must be set to zero while the data RAM is being accessed. Normally the value in the Memptr register is loaded by the microcode as It executes. The format of the register is shown in Figure 4.7.

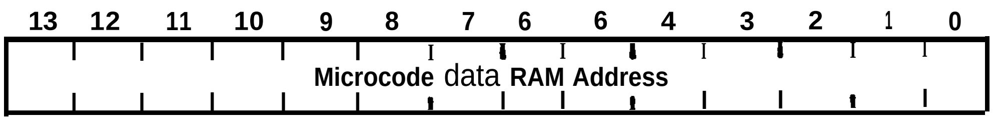

Figure 4.7 HQ Memptr Register

Bits 13-O : Microcode data RAM address. Contains the address of the data word which will be accessed by the HQ1 or by the host system.

HQ Temp Memptr Register

The Temp Memptr register is a 14 bit wide register which is used by the HQ1 to temporarily save the Memptr register for a single microcode instruction. The host software does not have the ability to read or write the Temp Memptr register and it is only described here for completeness. The format of the register is shown In Figure 4.8.

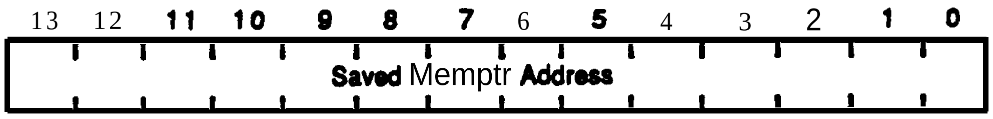

Figure 4.8 HQ Temp Memptr Register

Bits 13-O: Saved Memptr Value. Contains the saved value from the memptr register.

HQ REptr Register

The **Reptr** register is a 6 bit wide register which is used by the **HQ1** to address the registers in the Raster Engine. The host software does not have the ability to read or write the REptr register and it is only described here for completeness. The format of the register is shown in **Figure** 4.9.

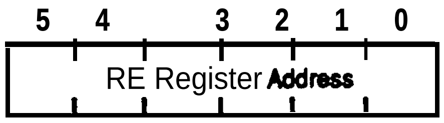

Figure 4.9 HQ REptr Register

Bits 5-O **: Raster Engine Register** Address. Contains the address of an RE2 **register** to be accessed.

HQ DMA Count Register

The DMA Count register is a 11 bit wide register which is used by the **HQ1** to count the number of words **DMAed** by the **HQ1.** The microcode bads the DMA count into the register and executes a repeatgez instruction which causes the dma transfers to continue while the DMA count register is greater than zero. For each word transfered the **HQ1** decrements **count** in the DMA Count register. The host software does not have the ability to read or write the DMA Count register and it is only described here for completeness. The format of the register is shown **in Figure** 4.10.

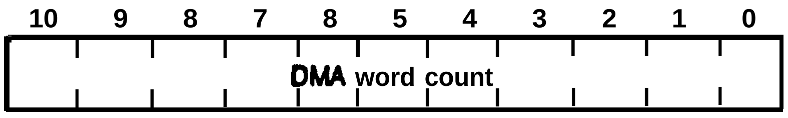

Figure 4.10 HQ DMA Count Register

Bits 1 O-O **:** DMA Word Count. Contains the number of words to be DMAed by the **HQ1.**

Weitek Registers

The Weitek chip contains 32 register which are used to store the source data and the results of the floating point multiplies and adds. The chip also contains 3 temporary registers used to hold the results of multiply and add operations. Each of the registers is 32 bits wide and can hold an IEEE format single precision floating point value or a 24 bit sign extended **2's** complement integer value. The host software does not have the ability to read or write the Weitek registers. The format of the register is shown in FIgure 4.11.

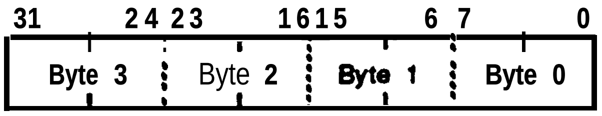

Figure 4.11 Weitek Registers

Bits 31-O : Floating Point or Integer Data. Contains the source data and result data of floating point operations by the Weitek.

Interrupts

The Geometry Subsystem generates two interrupts which can generate an interrupt to the host system. These are the GE interrupt and the FIFO Half-Full interrupt. These interrupts go to the Host interface Subsystem which contains the interrupt mask register and the interrupt status register. Refer to the Host interface Subsystem chapter for a description of these registers and for a description of how a host interrupt is generated.

FIFO Half-Full Interrupt

The FIFO Half-Full interrupt is generated by the FIFO hardware when it becomes half full. The FIFO can hold up to 512 entries and so if 256 entries are in the FIFO it generates an interrupt when the next tag/data entry is placed In it. The half full level was chosen to provide adequate time for the host operating system to respond to the interrupt without the FIFO becoming completely full.

GE Interrupt

The GE interrupt is generated by the HQ1 chip in response to a microcode instruction. The microcode causes a GE interrupt to be generated after it completes it's startup initialization after the microcode is downloaded. It also generates interrupts during picking and feedback mode each time the pick/feedback buffer becomes full. if the GE_ENDPICK or GE_ENDFEEDBACK tokens are sent tbe microcode causes a GE interrupt to be generated to tell the host when it can read the final pick or feedback data from the buffer.

The HQ1 latches the GE interrupt and the host must issue a clear GE interrupt HO command to clear the interrupt. This must be done before the interrupt status register GE interrupt bit is cleared to prevent a spurious interrupt to be generated by the Eddy Chip.

GE5 Basic Operations

The Geometry Subsystem contains a single microcoded processor capable of 20 million floating-point operations per second (MFLOPS). This processor is referred to as the Geometry Engine (GE) and the MGR adapter has the fifth generation version which is referred to as the GE5. The basic operation of the GE5 can be viewed at two different levels. The first level is the low level microcode operational level. The second level is the 3D graphics functional level. The following sections describe these two levels.

Microcode Operational Level

The GE5 consists of the HQ1, the Weitek 3132, the microcode code RAM, the microcode data RAM and the FIFO. The HQ1 acts the microcode sequencer and uses the PC register to read microcode words from the code RAM. The Weitek 3132 is used to perform the floating point arithmetic needed to perform the 3D graphics operations. The data RAM is holds various constants, variables and data buffers needed by the microcode. The FIFO is used by the microcode *to receive* host command tokens and data parameters. The tokens instruct the microcode to perform the necessary graphics operations.

The microcode instructions are either one or two 40 bit words. The 40 bit microinstruction fields provide control for the Weitek 3132 chip, the PC control unit, the REPTR, the MEMPTR and the various GE5 data paths. Most instructions use only a single 40 bit word format. However, some instructions such as conditional and unconditional branches require a second 40 bit word as part of the microinstruction. This second 40 bit field is used for constants, for target addresses for branches, for values to load in the MEMPTR, for values used to set the interrupt bits and for control values for the DMA channels.

Since the microcode is executed from RAM it must be downloaded by the host into the microcode code RAM. The host reads a file called **ge5_re2.bin** which contains the microcode code and data constants. Before doing the download the host issues a reset command to the MGR adapter which stalls the GE5 and resets the other hardware components. The host then writes the microcode code into the code RAM and the data constants into the data RAM.

The microcode contains a branch instruction at location zero to an initialization routine. The host reads location zero of the code RAM which causes the PC register to be loaded with zero. The host then clears the GE5 stall condition. The microcode executes the initialization routine. When the initialization is completed the microcode issues a GE interrupt to the host and does a **stall** instruction. Meanwhile the host polls the interrupt status register waiting for the GE interrupt to occur. Once the GE interrupt occurs the host issues a clear GE interrupt command to the HQ1 and then issues a clear stall command to cause the microcode to start executing.

The microcode expects the host to send down three parameters at this point. The first parameter informs the microcode if the extended bitplanes are installed. A zero means that the extended bitplanes are not installed and a one means that they are installed. The second parameter indicates if the Z buffer bitplanes are installed. A zero means that they are not installed and a one means that **they** are installed. The third parameter indicates if the VRAM is 256K or 1 Meg chips. A zero indicates 256K and a ***one*** indicates 1 Meg chips. Refer to the Display Subsystem chapter for information on how to read the display registers to determine this infonation. At this point the microcode loads these parameters into the appropriate Raster Engine registers and does a token FETCH instruction from the tag FIFO.

The microcode is organized as a branch table in low memory and the corresponding routines above the table in the microcode code RAM. The microcode does FETCH instructions to get command tokens from the tag FIFO. Each FETCH causes the HQ1 to read an eight bit quantity from the tag FIFO. The HQ1 shifts the token left by one bit to form an index into the branch table and toads the index into the PC. The left shift is necessary because each branch instruction in the table takes two 40 bit microcode word locations.

The microcode branch instruct&n is executed by the HQ1 which causes the corresponding routine higher up in the microcode RAM to be executed. The HQ1 executes the microcode instructions contained in the routine to perform the desired function. The last instruction in the routine will be a FETCH instruction. This causes the HQ1 to get the next token from the tag FIFO. This sequence is repeated as long as the host continues to send tokens down the FIFO.

If the command token requires data parameters the appropriate microinstructions are performed to read the 32 bit words from the data FIFO. The microcode can access the microcode data RAM as needed for access to constants, variables and data buffers. The microcode also accesses the various Raster Engine registers as needed to perform drawing operations.

The microcode can also execute jump to subroutine and return from subroutine instructions to allow the microcode to contain subroutines of commonly used code. The return address is saved on an 8 level stack in the HQ1.

Any further discussion of the operation of the microcode is beyond the scope of this document. The following section describes the 3D functions implemented by the microcode.

3D Graphics Operations

The MGR adapter provides support for the drawing of a wide range of 3D geometric objects. The objects can be drawn with a variety of attributes and can be solid filled using either flat shading or Gouraud shading. Support is also provided for drawing objects with multiple lighting **sources** with various lighting parameters specified for the surface characteristics of the objects being drawn. The GE5 microcode performs the necessary coordinate transformations from world coordinates to device coordinates and controls how the Raster Engine performs the pixel rendering operations using either RGB or color index pixel formats. The microcode supports a single graphics context and provides the necessary support for switching between multiple graphics contexts.

The GE5 processes a stream of command tokens and data **which** control the operation of the adapter. As each token is executed the microcode performs the various coordinate transformations and pixel rendering operations specified by the tokens and data. The various graphics context parameters In the GE5 data RAM are updated as necessary as the **various** tokens are executed. The appropriate Raster Engine registers are loaded to specify the drawing parameters and then the instruction register is loaded to cause the Raster Engine to perform a drawing operation.

The following functional operations are supported by the MGR adapter:

- Window Management Support
- Graphics Context Support
- Coordinate Transformations
- Drawing of World Coordinate Geometric Objects
- Drawing of Screen Coordinate Objects
- Lighting Support
- Feedback Support

  Picking and Selecting Support
- Pixel Rendering Support

These functional operations are described'in the following sections.

Graphics Resource Management

The host operating system must provide the necessary operating system software to manage the graphical resources provided by the MGR adapter. The graphical resources which need to be managed include:

- On screen windows
- Current graphics context state information
- 2 DMA channels
- 2 Cursor Chips
- XPC1 or XMAP2 mode registers
- Color map tables
- DAC color palettes

The management of the DMA channels is discussed in the Host Interface Subsystem chapter. The management of the mode registers, the color maps and the DAC color-palettes are discussed in the Display Subsystem chapter. The management of the cursor chips is discussed in the Raster Subsystem chapter. The following paragraphs describe the management of the on screen windows and the graphics context.

Window Management

If the host software wants to provide multiple graphics applications with the ability to simultaneously share the MGR adapter it must provide a window manager which controls multiple on screen windows. The window manager must manage the window creation and manipulation requests of the various graphics processes as they execute. The window manager provides multiple on screen windows by using the Window ID bitplanes and the hardware screen mask to control which bitplanes the graphics applications can write to. The window manager must also provide the user interface for controlling the appearance and location of the on screen windows.

The MGR adapter provides 2 WID bitplanes on the base adapter and 4 WID bitplanes for the enhanced adapter. This means that the base adapter can have up to 4 on screen windows and the enhanced adapter can have up to 16 on screen windows. If the fast Z clear mode is enabled on the enhanced adapter then the number of on screen windows is limited to 8 since two window IDs are used for each window in this mode.

The Window Manager can use the tokens shown in Table 4.3 to control the WID bitplanes and the hardware screen mask. The WID bitplanes allow a bit by bit control of which pixels can be written into. This allows very complex windows to be managed by the window manager. The hardware screen mask is used to clip all pixel writes outside of the specified rectangular area.

Table 4.3 Window Management Tokens

Token	Description
GE_CURRENTWID	Specifies the current WID to use during window ID checking
GE_ENABLWID	Used to enable the WID checking of lines
GE_ENABWID	Used to enable WID checking for other drawing except lines
GE_FLATMODE	Used to specify the span drawing mode and RE2 DX parameters
GE_SCRMASK	Used to specify the hardware screen mask rectangular coordinates
GE_SETPIECES	Used to set a clipping piece list used by GE_SCREENCLEAR token

The MGR adapter provides Window ID bitplanes which can be used to control pixel writes by the Raster Engine. The Window Manager writes the window ID for each on screen window into the WID bitplanes at the appropriate locations. The GE_CURRENTWID token is then used to specify which Window ID is currently active. if WID checking has been enabled then the current WID will be wmpared with the data in the WID bitplanes and only those pixels where the current WID is the same as the WID data will be written. The GE_ENABWID token controls the WID checking for the shaded span and write buffer instructions of the Raster Engine. The GE_ENABLWID token controls the WID checking for the Draw tines instructions of the Raster Engine. These instructions are described in greater detail in the Raster Subsystem chapter.

The GE_SCRMASK token specifies a rectangle which is loaded into the Raster Engine screen mask registers. The screen mask is used to clip all pixel writes for pixels whose location is outside Of the screen mask rectangle. The screen mask clipping is always enabled for all of the Raster Engine instruction which write pixels. If no screen mask clipping is desired then the screen mask rectangle should be set to the full screen dimensions.

The WID checking operations cause the Raster Engine instructions to draw tokens slower than when WID checking is not enabled. The use of the screen mask or WID checking to perform pixel writ8 clipping is dependent on the shape of the currently active window. If the current window is unobscured (a single 1 piece window) then the screen mask can be used to clip pixel writes which are outside of the window. If the window is partially obscured by other windows then the WID checking must be enabled to clip pixel writes.

Since the WID checking performance is most noticeable for screen clear operations a special optimization mechanism is provided by the GE5 microcode. This mechanism consists of the piece list which is set using the GE_SETPIECES token. This token allows the window manager to specify up to four rectangular clipping masks which are used by the GE SCREENCLEAR and the GE CZCLEAR tokens. If the current window is unobscured or obscured with only 2 to 4 rectangular &es then the piece list is used to control pixel clipping so that only the pieces of the current window are cleared. If the window is divided into more than 4 rectangular pieces or is a non-rectangular shape then WID checking must be used to control the pixel clipping. The SIMPLE flag in the GE5 data RAM controls whether WID checking or the piece list control the screen clear pixel clipping. If the current window contains from one to four rectangular pieces then the SIMPLE flag is set to 1. If the window is divided into more than 4 rectangular pieces or the window is non-rectangular then the SIMPLE flag is set to -1. The discussion below describes how the SIMPLE flag is written into the current context or a saved context.

The GE_FLATMODE token is used to specify if the flat span instruction or the shaded span instruction is used to draw filled polygons. The flat span instruction is faster than the shaded span instruction but it does not allow window checking operations to be performed. The flat span instruction can not draw shaded spans. The. shaded span instruction can be used to draw either shaded or flat spans which can be WID checked. The GE_ENABWID token can be used to enable WID checking all the time for shaded spans and the GE_FLATMODE token can be used to specify whether the shaded or flat span token is used to fill polygons. If the GE_SHADEMODEL token specifies shaded fill mode then GE_FLATMODE token will be ignored and the shaded span instruction will be used. If the GE_SHADEMODEL specifies a flat mode then the GE_FLATMODE token will control whether the flat span or the shaded span will be used to fill the polygon. If the window is obscured and WID checking is needed then the FLATMODE parameter of GE_FIATMODE will be set to 1 to indicate that the shaded span instruction should be used. In this case the FLATDX parameter will be Set to 0x4000 for the RE2 DX register. If the window is unobscured then the FLATMODE parameter is set to 2 to indicate that the flat span instruction should be used to fill the polygon. The FLATDX parameter would be set to 0 for this case.

The window manager can use the tokens described above only for the current graphics context. However for the other windows whose shapes have changed the window manager can not use the tokens described above to change the window management parameters. These tokens can only be used for the current context if the window manager can guarantee that the graphics application is not in the process of sending a data parameter stream for a token. A better approach would be to update the parameters in the saved context in memory and then have these changes take affect when the context is restored. For the current context this could be done by having the window manager switch out the current context, change the parameters and then switch the context back into the adapter. The parameters which are changed in the saved context are shown in Table 4.4.

The ENABLWID parameter Is used to enable or disable line WID checking. The shaded span WID checking would always be enabled as described above. The FLATMODE and FLATDX parameters are used the same as described above. The NEWORG parameter is a flag which instructs the context restore microcode to recalculate the screen mask and viewport parameters. If NEWORG is set to 1 the recalculation is performed and if NEWORG is -1 then the recalculations are not performed. The XORG and YORG parameters are used to specify the lower left comer of the window. The SIMPLE flag is used as described above. Finally the last 17 parameters are the piece list count and the specifications for the 4 piece rectangles. The 24 parameters are stored contiguously in the saved context and can be written from an array into the context data area.

Table 4.4 Window Control Parameters in the Context

Address	Description
ENABLWID	Controls whether line WID checking is enabled or disabled
FLATMODE	Controls polygon fill shade mode (flat fill or shaded fill)
FLATDX	Specifies the RE2 Span parameter specifying the DX value
NEWORG	Indicates if the viewport or screen mask parameters have changed
XORG	Specifies the lower left x screen coordinate of the current window
YORG	Specifies the lower left y screen coordinate of the current window
SIMPLE	Indicates if the piece list or WID checking is used by screen clear tokens
NUMPIECES	Indicates the number of pieces into which the window is divided
YLEN1	Specifies the y direction length of the first piece
LLY1	Specifies the lower left y screen coordinate of the first piece
XLEN1	Specifies the x direction length of the first piece
LLX1	Specifies the lower left x screen coordinate of the first piece
YLEN2	Specifies the y direction length of the second piece
LLY2	Specifies the lower left y screen coordinate of the second piece
XLEN2	Specifies the x direction length of the second piece
LLX2	Specifies the lower left x screen coordinate of the second piece
YLEN3	Specifies the y direction length of the third piece
LLY3	Specifies the lower left y screen coordinate of the third piece
XLEN3	Specifies the x direction length of the third piece
LLX3	Specifies the lower left x screen coordinate of the third piece
YLEN4	Specifies the y direction length of the fourth piece
LLY4	Specifies the lower left y screen coordinate of the fourth piece
xLEN4	Specifies the x direction length of the fourth piece
LLX4	Specifies the lower left x screen coordinate of the fourth piece

The programming considerations for the WID bitplanes are described in the Raster Subsystem chapter.

Grap hics Context Management

The MGR adapter provides hardware and microcode support for only a single graphics context **so** if the host operating system supports **multitasking** operations then it must perform a graphics context switch when the current graphics process is made **inactive** and another process is activated. When switching contexts the current graphics context must be saved in the host data RAM and a different context must be loaded into the adapter and made active. The host operating system maintains the appropriate data structures to **allow** the current state informatbn to be saved and a new context switched to. The exact nature of these data structures is beyond the scope of this document and will only be described in terms of the data which must be maintained for the proper operatbn ***of*** the adapter.

The graphics context contains various state information which represents the current state of the MGR adapter. The graphics context state information **is** maintained in both the host data RAM and **in** the MGR adapter. The host data RAM contains the state information which describes the window manager state for the various windows be displayed on the adapter and the state of the ***Host*** Interface and Display Subsystems. The adapter maintains the current state information for the Geometry Engine and the Raster Engine.

The graphics context consists of the following state information:

- Host maintained state-.
  - Current Context
    - graphics state maintained by graphics application (such as the SGI **gl)**
    - Window Manager state information
    - Host interface Subsystem state
    - Display Subsystem state
  - Saved context state for each inactive process
- Adapter maintained state for current context
  - **GE5** operational state
  - current contents of the Weitek registers
  - current contents of the Raster Engine registers shadowed in **GE5** data RAM

The GE5 data RAM layout is shown **in** Fiiure 4.12 and the symbolic names of the address are defined in the file **ge5_glob.h.** This header file also defines the **individual** address offsets **in** the variable data area and the other areas. The DIVMOD table is used by the microcode to format the X screen coordinates into the format required by the Raster Engine. The Constants table is used to hold constants information required by the **GE5** microcode as it executes. The DIVMOD and Constants tables are not part of the graphics context since they can be considered as read only constants.

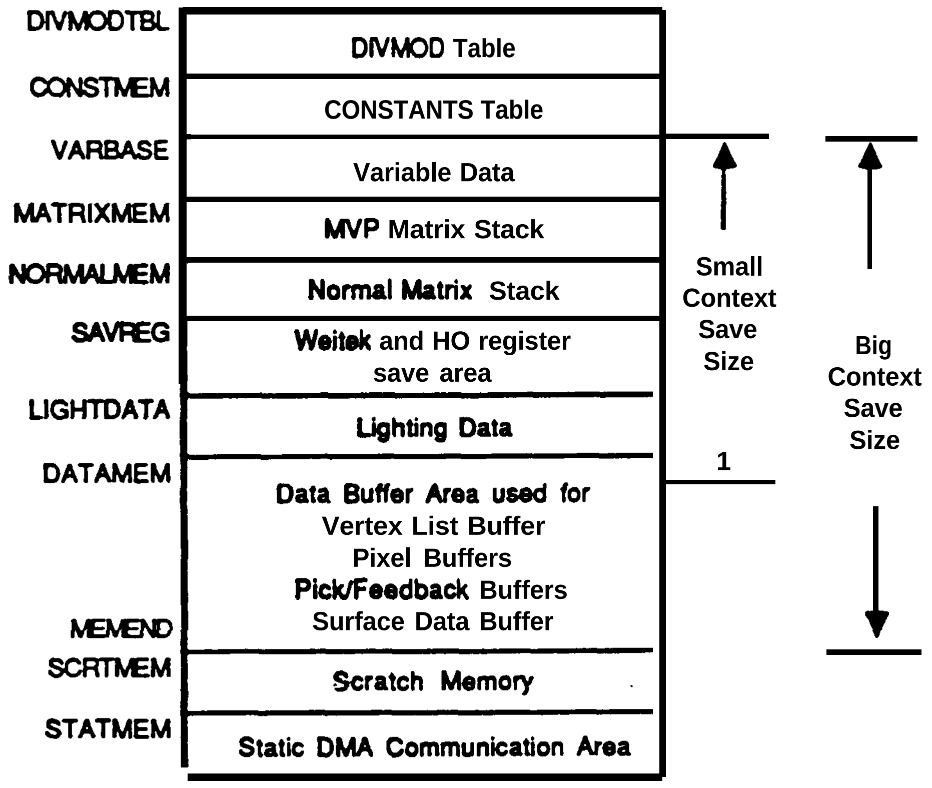

Figure 4.12 GE5 Data RAM Layout

The graphics context area includes the following data:

- the variable data area
- the Modeling/Viewing/Projection matrix stack
- the Normal matrix stack
- the' register save area
- lighting data area
- data buffer area

The **variable** data area contains the state information of the GE5 microcode as it executes the various tokens that it receives. This area contains the following types of data:

- current color data
- current graphics position
- current character position
- current **viewport** data

- current screen mask data
- shadow RE2 register data
- window manager data (as defined in the previous section)
- various branch and mode flags

The window manager data described in the previous section is one area of the saved context that is manipulated by the host software. The additional data items shown in Table 4.5 are also accessed by the host software. The window manager sets the current **WID** in the saved context by writing to the **CURWID** data location in the context. The other data items are used by the host software which does the context switch. The use of these data locations are described in the programming considerations for graphics contexts in the Programming Considerations section of this chapter.

Table 4.5 Context Data Accessed by the Host

Address	Description
CURWID	Contains the current WID data
HQMSAV	Contains the HQM MAR register value to be restored
MEMPTR	Contains the GE5 memory pointer when the context was saved
PCSAVE	Contains the saved microcode Program Counter
REPTR	Contains the RE2 register pointer when the context was saved

The host software should not change any other data in the saved context besides those described above. The host software can access other areas of the GE5 data RAM to get return data from the microcode as it executes tokens. These data locations are described for the individual tokens and are summarized at the end of the Token Definitions chapter.

Since the MGR adapter can only contain a single context it is necessary for the host software to maintain the appropriate data structures to hold saved contexts for graphics processes which have been switched out of the adapter and saved in the host memory. After the host switches out a context it restores a previously saved context or initializes a new context. The tokensshown in Table 4.6 are used to perform the context switching and context initialization.

Table 4.6 Context Switch Tokens

Address	Description
GE_CTX0	Initiates a Context Switch
GE_CTX1	Initiates a Context Switch (in case first token is not recognized)
GE_INIT	Initializes a new context

The **GE_CTX0** and the GE **CTX1 tokens** are used to initiate a context switching operation. The use of these tokens is described% the Programming considerations section of this chapter. The **GE_INIT** token is used to initialize a new context. This token causes the microcode to set all of its mode flags to their default conditions and to initialize the various data structures. The **GE_INIT** token definition defines the default settings for the various microcode data variables.

Coordinate Transformations

The MGR adapter provides support for three types of coordinate systems which include the world coordinate system, the normalized coordinate system and the device coordinate system. The world coordinates are specified as (x, y, z) triplets with values which are appropriate to the objects being drawn. The world coordinates can be specified using either floating point or integer values but the integer values will be converted to floating point by the microcode. The transformations performed on the world coordinates are always done in floating point. The world coordinate system uses a right handed system in which the positive x axis is to the right, the positive y axis is up and the positive z axis toward the viewer.

The GE5 microcode provides support for the necessary matrix transformations to transform 3D world coordinates into device coordinates for use by the Raster Subsystem during the pixel rendering operations. **The microcode manages a 16** level 4 x 4 matrix stack which contains the concatenated modeling, viewing and projection matrices which are used to transform the 3D world coordinate data to the 3D normalized coordinates. The normalized coordinates are clipped against a bounding cube which has the dimenslons of -1 to +1 along each of the ‘x, y and z axis. Once the normalized coordinates have been clipped against the normalized viewing volume they have the perspective division applied to them. The perspective division consists of dividing the transformed x, y and z coordinate values by the transformed w coordinate value. Finally the coordinates are multiplied by the device scaling factors specified by the viewport specification. The viewport translation offset factors are added tn the resulting coordinates- to get the resulting window relative device coordinates.

The MGR adapter also provides a corresponding 16 level 3 x 3 matrix stack of normals for lighting surface normal transformations. The use of the normal matrix stack is described in the later section on lighting support.

The tokens shown in Table 4.7 are provided for the coordinate transformations. The **GE_LOADMATRIX** token Is used to load the initial projection matrix on the Modeling/Viewing/Projection matrix stack. The projection matrix can be either a perspective or orthographic projection matrix. The GE MULTMATRIX token is used to concatenate the viewing and modeling matrices with the project&n matrix on the top of the MVP matrix stack. The GE_PUSHMATRIX token is used to push **the matrices** on the MVP stack down one level. The current top of the matrix stack is left unchanged. The GE_POPMATRIX token is used to pop the matrices on the MVP stack up one level. The previous current top of the stack matrix is overwritten.

Table 4.7 Coordinate Transformation Tokens

Token	Description
GE_LOADMATRIX	Loads the initial coordinate transformation matrix
GE_LOADVIEWP	Loads the viewport device coordinate mapping values
GE_MMODE	Used to control the transformation matrix stack mode
GE_MULTMATRIX	Multiplies the current transformation matrix with another matrix
GE_POPMATRIX	Used to do a pop operation on the matrix stack
GE_PUSHMATRIX	Used to do a push operation on the matrix stack

The **GE_MMODE** token is provided to support lighting calculations. When it is set to single mode the MVP stack is not connected in any way to the normal matrix stack. When the matrix mode is set to viewing or projection mode then the push and pop operations cause both the MVP and normal matrix stacks to be pushed or popped. The use of the **GE_MMODE** token is described in greater **detail** in the section on lighting support.

The **GE_LOADVIEWP** token is used to specify the scaling and translation factors applied to the normalized coordinates to transform them to device coordinates. The translation factors are added to the origin of the current window so that the resulting device coordinates are window relative.

The tokens shown in Table 4.8 are provided by the GE3 microcode to assist in the development of a PHIGS graphics package. These tokens are not used by the Graphics Library supplied by Siiiin Graphics. The tokens use the bottom six levels of the MVP matrix stack for their operations. These tokens are defined in the Token Definitions chapter.

Table 4.8 PHIGS **Matrix** Support Tokens

Token	Description
GE_COMPOSEMATRIX	**Used to concatenate two matrices** on **PHIGS** matrix stack
GE_COPYMATRIX	Used to copy a matrix from one location to another on PHIGS stack
GE_LOADTOPMATRIX	Used to copy **a matrix to the** top of the PHIGS matrix stack
GE_SETMATRIX	Used to load a matrix on the PHIGS matrix stack

Drawing World Coordinate 3D Geometric Objects

The GE5 microcode provides support for drawing geometric objects whose coordinates are specified either in world or screen coordinates. The microcode provide support for drawing the following **world** coordinate geometric objects:

- points
- lines and poiyiines
- closed polyiines
- filled polygons
- triangle meshes
- **curves**
- NURBS
- screen aligned boxes
- text characters

The following paragraphs describe the world coordinate geometric drawing support provided by the microcode. The screen coordinate drawing support is described in a later section.

Current Graphics Position

The current graphics position **specifies the starting position** where the next line drawing operation begins. After the line drawing operation is **completed** the current graphics position is updated to the **end** of the line position. The point drawing operations also update the current graphics **position to** the specified point position.

Point Drawing

The GE5 microcode supports the drawing of points whose location is specified in world coordinates. The point is one pixel wide and after the point is drawn the current graphics position is updated to the location specified for the point. The tokens shown in Table 4.9 are provided for drawing points.

Table 4.9 Point Drawing Tokens

Token	Description
GE_PNT2	Draw point at absolute floating point 2D coordinate
GE_PNT2I	Draw point at absolute integer 2D coordinate
GE_PNT3	Draw point at absolute floating point 3D coordinate
GE_PNT3I	Draw point at absolute integer 3D coordinate
GE_PNT4	Draw point at absolute homogeneous coordinate
GE_PNT4I	Draw point at absolute homogeneous coordinate
GE_SMOOTHPOINT	Used to enable or disable the drawing of antialiased points

The location of the points- can be specified in either floating point or integer coordinates. The integer coordinates are converted to floating point values by the microcode. The coordinate location is an absolute coordinate which can be specified in 2D, 3D or homogeneous coordinates. The 2D coordinate specifies a row vector [x y 0 1] in which the z element is 0 and the w element is 1. The 3D coordinate specifies a row vector [x y z 1] in which the w element is 1. The homogeneous coordinate specifies a row vector [x y z w] in which the w element is specified by the host software.

The **GE_SMOOTHPOINT** token is used to enable or disable the drawing of antialiased lines.

Line and Polyline Drawing

The GE5 microcode provides support for drawing lines and polylines. The line drawing begins by specifying a current graphics position and then issuing one or more line drawing tokens. The tokens shown in Table 4.10 are provided for setting the current graphics position.

Table 4.10 Set Current Graphics Position Tokens

Token	Description
GE_MOVE2	Set current graphics position using floating point absolute 2D coordinates
GE_MOVE2I	Set current graphics position using integer absolute 2D coordinates
GE_MOVE3	Set current graphics position using floating point absolute 3D coordinates
GE_MOVE3I	Set current graphics position using integer absolute 3D coordinates
GE_MOVE4	Set current graphics position using floating point homogeneous coordinates
GE_MOVE4I	Set current graphics position using integer homogeneous coordinates
GE_RMOVE2	Set current graphics position using floating point relative 2D coordinates
GE_RMOVE2I	Set current graphics position using integer relative 2D coordinates
GE_RMOVE3	Set current graphics position using floating point relative 3D coordinates
GE_RMOVE3I	Set current graphics position using integer relative 3D coordinates

The new current graphics position can be specified in either floating point or integer coordinates. The Integer coordinates are converted to floating point values by the microcode. The new graphics position can be specified as either an absolute coordinate or a relative coordinate. The absolute coordinate replaces the current graphics position with the new coordinate position. The relative coordinate is added to the current graphics position to create the new current graphics position. The coordinates can be specified in 20.30 or homogeneous coordinates. The 2D coordinate specifies a row vector [x y 0 1] in which the z element is 0 and the w element is 1. The 30 coordinate specifies a row vector [x y z 1] in which the w element is 1. The homogeneous coordinate can only be specified in the absolute format and specifies a row vector [x y z w] in which the w element is specified by the host software.

The line drawing tokens are shown in Table 4.11. A single line is drawn by sending one of the draw tokens which causes a line to be drawn from the current graphics position to the coordinate position specified with the draw tokens. After the line is drawn the current graphics position is updated to the end of the line coordinate. To draw polylines it is only necessary to set a current graphics position with the move tokens shown above and then send multiple draw tokens. The current graphics position is updated to the end of the last line drawn.

The draw tokens can be specified in either absolute or relative forms using either floating point Or integer coordinates. The coordinate specifying where the line is to be drawn to can be specified in 2D, 3D or homogeneous coordinates.

The lines can be drawn with many different attributes as shown in Figure 4.12. The GE ANTIALIASE token allows the enabling or disabling of the antialiased line drawing mode. The antialiased lines have a smoother appearance but take longer to draw. The Z compare hardware in the RE2 can be used to improve the appearance of intersecting antialiased lines. Refer to the Z Buffer support section later in this chapter for additional details.

Table 4.11 Line Drawing Tokens

Token	Description
GE_DRAW2	Draw line from current GPOS to absolute floating point2D coordinates
GE_DRAW2I	Draw line from current GPOS to absolute integer 2D coordinates
GE_DRAW3	Draw line from current GPOS to absolute floating point 3D coordinates
GE_DRAW3I	Draw line from current GPOS to absolute integer 3D coordinates
GE_DRAW4	Draw line from current GPOS to absolute homogeneous coordinates
GE_DRAW4I	Draw line from current GPOS to absolute homogeneous coordinates
GE_RDRAW2	Draw line from current GPOS to relative floating point 2D coordinates
GE_RDRAW2I	Draw line from current GPOS to relative integer 2D coordinates
GE_RDRAW3	Draw line from current GPOS to relative floating point 3D coordinates
GE_RDRAW3I	Draw line from current GPOS to relative integer 3D coordinates

The GE_DEPTHCUE token is used to enable or disable depth cued lines. The GE_RGBSHADERANGE token is used to set the depth cued shade range for RGB pixels while the **GE_SHADERANGE** token is used to set the **depth** cued shade range for Color Index pixels.

Table 4.12 Line Attribute Tokens

Token	Description
GE_ANTIALIASE	Used to enable or **disable antialiase** line drawing mode
GE_DEPTHCUE	Used to enable or disable depth cued line drawing
GE_LINESTYLE	Used to set the line stipple pattern
GE_LINEWIDTH	Used to set the line width in pixels
GE_LSREPEAT	Used to set the repeat factor for the bits in the line stipple
GE_RESETLS	Used to cause the line stipple pattern to be reset
GE_RGBSHADERANGE	Used to **set** the **depth cue** color shade range for RGB pixels
GE_SHADERANGE	Used to set the depth cue color shade range for Color Index pixels
GE_SUBPIXEL	Used to enable the calculation of sub pixels during antialiase mode

The **GE_LINEWIDTH** token is used to specify a wide line which covers multiple adjacent pixels. The lines can also be drawn with a stipple pattern applied to them. The **GE_LINESTYLE** token is used to specify a line stipple pattern and the GE_LSREPEAT token is used to **specify** a count of **how many** times each pixel in the stipple pattern will be used as the line Is drawn.The **GE_RESETLS** token is a flag which causes the stipple pattern **to be reset for each** line drawn. Refer to the Raster **Subsystem** chapter for additional details on the use of stipple patterns in line drawing.

The **polylines** tokens can be used to draw several higher level unfilled **geometric** objects such **as:**

- circles

- arcs
- ellipses

The circles and arcs can be drawn by **dividing** the circle or arc into a large number of small line segments and then issuing the move and draw tokens. **This** technique can also be used to draw elliptical shapes. The dosed **polyline** tokens **described** in the *next* paragraph automatically save the first coordinate and draw a closing line segment from the end of the last line segment to the beginning of the first lIne segment. The arc drawing would need to use the open **polylines** since the end line segment does not connect back to the beginning line segment. The drde and ellipse drawing could perhaps be better done **with** the dosed **polylines** but are shown here since this is how the Graphics **Library** draws them.

Closed PolyLine Drawing

The GE5 microcode provides a special form of the line drawing which causes a closed polyline to be drawn. The tokens which **support closed polylines** are shown in Table 4.13. The polylines are drawn as described above but they are enclosed by the two tokens shown. The **GE_CLOSEDLINE** token sets a flag in the microcode which causes the microcode to **save the new** current graphics position which is set by the first move token. The draw to&ens then cause the polylines to be drawn and after the fast line segment is drawn the GE_ENDCLOSEDUNE token causes the microcode to draw one last line segment **to the** saved graphics **position of the beginning of the first line segment.**

Table 4.13 Closed PolyLine Drawing Tokens

Token	Description
GE_CLOSEDLINE	**Starts Pdy Line Mode**
GE_ENDCLOSEDLINE	**Ends Poly Line Mode**

The line segments which form the closed polyline are drawn with the same attributes as described above for line segments. The closed polylines are used to draw unfilled geometric objects which are formed by a closed polyline boundary. The closed polyline tokens can be used to draw the following higher level unfilled geometric objects:

- rectangles
- polygons

The next section describes the mechanism for drawing filled geometric objects.

Filled Polygon Drawing

The tokens described in this section are provided to draw closed polylines whose interiors are filled with a specified pattern. The objects being drawn are specified as a sequence of vertices which form a vertex list stored in the GE5 data RAM. The tokens shown in Table 4.14 are used to specify the vertices which are stored in the vertex list which can hold a maximum of 256 vertices.

Table 4.14 Vertex Tokens

Token	Description
GE_RVERTEX2	Set vertex using floating point relative 2D coordinates
GE_RVERTEX2I	Set vertex using integer relative 2D coordinates
GE_RVERTEX3	Set vertex using floating point relative 3D coordinates
GE_RVERTEX3I	Set vertex using integer relative 3D coordinates
GE_VERTEX2	Set vertex using floating point absolute 2D coordinates
GE_VERTEX2I	Set vertex using integer absolute 2D coordinates
GE_VERTEX3	Set vertex using floating point absolute 3D coordinates
GE_VERTEX3I	Set vertex using integer absolute 3D coordinates
GE_VERTEX4	Set vertex using floating point homogeneous coordinates
GE_VERTEX4I	Set vertex using integer homogeneous coordinates

The vertices can be specified in either absolute or relative formats. The relative format causes the specified vertice coordinates to be added to the current graphics position to get the new current graphics position. The vertices **are** specified in world coordinates using either floating point or integer coordinates. The vertices can also be specified in **2D,** 3D or homogeneous coordinates.

To draw a filled polygon the tokens shown in Table 4.15 are used to specify the drawing style. Two **styles** of polygon drawing are supported by the microcode. An older style causes an outlined polygon to be drawn while the newer style causes a point sampled polygon to be drawn.

The old style polygons have a line segment drawn between the vertices**and** are filled inside the line segment using point sampling. This style is slower than the point sampled style but gives a more even edge appearance. The old style polygons are drawn by sanding the GE-POLYGON token followed by the vertice list and then ended with the GE_ENDOLDPOLYGON **token.** When the GE_ENDOLDPOLYGON token is received by the microcode the **vertice** list is traversed and the filled polygon is drawn. The GE_FATPOLY token is used to enable or disable the fattening of the outlined edges when drawing old style polygons. If enabled the edge lines will be adjusted so that they appear slightly wider than if this mode is disabled. This mode is provided for compatibility with older SGI products.

The newer style of polygon drawing does the point sampled fill but does not drawn the outline around the polygon edges. This method is faster than the older method. The newer style polygons are drawn by sending the GE-POLYGON token followed by the vertice list and then ended with the GE_ENDPOLYGON token. When the GE_ENDPOLYGON token is received the vertice list is traversed and the point sampled polygon is drawn.

Table 4.15 Polygon Drawing Tokens

Token	Description
GE_BACKFACE	Used to enable or dii bacface polygon removal mode
GE_CONCAVE	Used to enable or disable concave polygon drawing mode
GE_ENDOLDPOLYGON	Used to draw an old style outlined polygon
GE_ENDPOLYGON	Used to draw a new style **point sampled polygon**
GE_FATPOLY	Used to enable or dii the **wide** lines mode for outlined polygons
GE_POLYGON	Used to specify the beginning of a **vertice** list for a polygon

The default polygon type is convex polygons which have the characteristic that if any polygon edge line were extended forever it would never cross any other edge line except at the two vertices where it connects to its adjoining two edges. The other type of polygon which can be drawn is a concave polygon in which the above attribute does not hold true. Some edges of the polygon do go into the polygon and so therefore the edge lines if extended would cross other**edges** of the polygon at locations other than the adjoining vertices. The GE-CONCAVE token is used to enable or disable the drawing of concave polygons. When the concave mode is disabled the convex polygon mode is in affect. The microcode requires the host to set the concave mode before sending the concave polygon vertices to let the microcode know that a concave-polygon is being drawn.

The GE_BACFACE token is a form of hidden line removal **supported** for polygons when a Z buffer is not **available** for removing hidden lines. The **backface** polygon removal requires that all polygon **vertice** lists **be** specified in a **counterclockwise** order. This means that if an object had been rotated and the polygon was being drawn in a clock wise **order** it must now **be** a back facing polygon and should **be removed.** The GE_BACKFACE token is used to enable or disable back facing polygon removal. This mechanism does not work completely for concave polygons and should not be enabled if a **Z** buffer card is installed.

The tokens shown in Table 4.16 control the fill pattern and shading attributes used during the drawing of a filled polygon. The **GE_SETPATTERN** token is **used** to specify a 32 x 32 bit pattern **which** is used to fill the polygon. The GE-PATTERN token is used to enable or **disable** the use of the pattern. The GE_SHADEMODEL is used to specify whether **the** fill ***colors*** are flat or GOURAUD **shaded.**

Table 4.16 Polygon Fill Attribute Tokens

Token	Description
GE_PATTERN	Used to enable or disable the use of the pattern for filled polygons
GE_SETPATTERN	Used to specify a 32 x 32 bit pattern and enable pattern usage
GE_SHADEMODEL	**Specifies** whether **GOURAUD** or Flat shading is used for filled polygons

The old style polygon microcode is**used** to draw the following filled geometric objects:

- old style outlined polygons
- rectangles

- arcs
- circles

The new style of point sampled polygons are used only for drawing new style polygons.

Triangle Mesh Drawing

The microcode provides support for drawing a series of triangles which form a triangle mesh. The triangle vertices are specified using the vertex tokens shown in table 4.14. The tokens shown in Table 4.17 are used to draw the triangle meshes.

Table 4.17 Triangle Mesh Drawing Tokens

Token	Description
GE_BEGINMESH	Starts Triangle Mesh Mode
GE_ENDMESH	Ends Triangle Mesh Mode
GE SWAPMESH	Swaps the order of the saved triangle vertices

The GE_BEGINMESH token causes the microcode to start using the following vertices for use in drawing triangles. The first three vertices are used to draw the first triangle. The microcode saves two vertices and a pointer to one of the two vertices. Initially the pointer will point to the first vertex save location. When the first vertex arrives the vertex is stored in the vertex save location pointed to by the pointer. After the first vertex is saved the pointer is updated to point to the second vertex save location. After the second vertex is received it is stored in the second save location and the pointer is updated to point to the first save location. When the-third vertex arrives the triangle is drawn, the third vertice replaces the first vertex in the save location being pointed at by the pointer and the pointer is updated to point to the second vertex save location. When the fourth vertex is received by the microcode a second triangle is drawn using the second, third and fourth vertice. The fourth vertfce replaces the vertex which is being pointed to by the pointer and then the pointer updated to point to opposite save location. This process continues until the GE_ENDMESH token is received by the microcode at which time the triangle mesh drawing mode is ended.

The GE_SWAPMESH token is provided to allow the pointer to be swapped from the vertex it is currently pointing at to the other vertex save location. The next vertex received will replace the vertex being pointed after the new triangle is drawn and the pointer will be updated to point to the other vertex save location. The ability to swap the pointer between the saved vertices allows the host software to control the order in which the vertices are replaced by the newly received vertices. This allows the host software to draw triangles which require a particular vertice to be used in two different triangles where the normal sequence of vertices would have caused the needed vertice to have been replaced by a new vertice if the swap pointer token had not been used to switch the pointer to the opposite vertice so that the needed vertice is not replaced by the next received vertice. Refer to SGI Graphics Library users guide for examples of the use of the triangle mesh tokens.

Curve Drawing

The microcode provides support for drawing curves which include Berier cubic curves, cardinal spline curves and B-spline curves. The GE_LOADMATRIX and the GE_MULTMATRIX are used to load the appropriate forward difference matrix onto the top of the matrix stack and the GE_CURVEIT token shown in Table 4.18 is used to draw the curves using the forward difference matrix. An iteration count specifies the number of line segments which are drawn to form the curved line.

Table 4.18 Curve Drawing Tokens

Token	Description
GE_CURVEIT	iterates the matrix stack and draws lines

The same tokens can also be used to draw surface patches which is a wireframe of curve segments. The parametric bicubii surface is the mathmatical basis for the surface patches.

NURBS Drawing

The microcode provides the tokens shown in Table 4.19 for drawing the non-uniform rational B-spline curves.

Table 4.19 NURBS Drawing Tokens

Token	Description
GE_PUSHV	Used to move setv coefficients to a different location in the data RAM
GE_SETSURFSCALE	Used to reparameterize each patch in a surface to [0 1]
GE_SETV	Used to compute the second parametric value
GE_SETVHI	Used to compute the DeCastelau coefficient
GE_STRIP	Used to compute the object space vertices along a strip of a patch
GE_SURFMODE	Used to indicate if a nurbs context exists
GE_SURFNTURF	Used to indicate the polynomial order of the two parametric values
GE_SURFP1	Used to compute the first parametric value
GE_1LOAD1	Used to load 1 NURBS data parameter
GE_1LOAD3	Used to load 3 NURBS data parameters
GE_1LOAD4	Used to load 4 NURBS data parameters

Screen Aligned Box Drawing

The microcode provides support for drawing a very fast filled 2D rectangle. The rectangle must be aligned to the x and y coordinates so the transformation matrix used to transform the world coordinate used to specify the lower left and upper right comers of the rectangle cannot have any rotation angles other than zero applied to them. The tokens shown in Table 4.20 are used to draw the filled screen aligned rectangles. As shown in the table the coordinates can be specified in either floating point or integer world coordinate values.

Table 4.20 Screen Aligned Box Drawing Tokens

Token	Description
GE_SBOXF	Used to draw a screen aligned filled box specified with floating point
GE SBOXFI	Used lo draw a screen aligned filled box specified with integer

The filled screen aligned boxes are drawn using the current line style parameter specified with the GE_LINESTYLE and GE_LSREPEAT tokens. When drawing sboxes the lighting, depthcueing, z buffering and Gouraud shading cannot be used.

Character Drawing

The microcode provides support for drawing text characters. The characters are drawn at the current character position which is set with the tokens shown in Table 4.21. The character position is specified in either 20, 3D or homogeneous coordinates using floating point or integer coordinate values. The world coordinates are transformed into screen coordinates which are saved for the GE_DRAWCHAR token shown in Table 4.22 to use to draw the character. The **GE_GETCPOS** token returns the current transformed screen coordinate character position. The old style pixel read and write tokens described later also use and affect the transformed screen coordinate current character position.

Table 4.21 Set Current Character Position Tokens

Token	Description
GE_CMOVE2	Set current character position using floating point absolute 2D coordinates
GE_CMOVE2I	Set current character position using integer absolute 2D coordinates
GE_CMOVE3	Set current character position using floating point absolute 3D coordinates
GE_CMOVE3I	Set current character position using integer absolute 3D coordinates
GE_CMOVE4	Set currentcharacter position using floating point homogeneous coordinates
GE_CMOVE4I	Set current character position using integer homogeneous coordinates
GE_GETCPOS	Get current character position

The characters which are drawn are not affected by the scaling and rotation specified in the current transformation matrix. The characters are always drawn along the **horizontal** axis. The characters could be drawn vertically by modifying the font definitions for the characters and by manipulating the current graphics after each character is drawn. The current character position is updated to the pixel to the right of the last pixel in the current character being drawn. The current character position does not affect the current graphics **position** in any way.

Table 4.22 Character Drawing Token

Token	Description
GE_DRAWCHAR	Used to draw a text character with the **specified** font metrics

The characters being drawn are clipped to the **viewport** and to the screen mask. If the current character position is outside the **viewport** then the current character position Is invalid and the characters are not drawn. The screen mask will clip the bits of the character which are outside of the screen mask.

This completes the discussion of the world coordinate drawing support provided by the microcode. The next section will discuss some routines whose coordinates are specified in screen coordinates.

Drawing Screen Goordinate 2D Objects

The GE5 microcode supports a limited number of tokens that draw 2D screen coordinate objects. Screen coordinate direct pixel reads from the bitplanes are also supported. These object which are drawn using screen coordinates do not have the world coordinate to. screen coordinate transformations performed so the objects cannot be rotated, scaled or translated. The screen coordinate support includes:

- Fast 2D lines
- Pixel Reads
- Pixel Writes

The following paragraphs describe these screen coordinate objects.

Fast 2D Line Drawing

The microcode provides support for drawing fast 2D lines as shown in Table 4.23. The GE_FMOVE token is used to specify a fast line drawing position and the GE_FDRAW token is used to draw the fast 2D line from the current fast line drawing position to the coordinates specified in the GE_FDRAW token. The GE_FLINE token specifies the starting and ending coordinates of the line whii is drawn.

Table 4.23 Fast ‘2D Line Tokens

Token	Description
GE_FDRAW	Draw fast 2D line from current fast line graphics position.
GE_FLINE	Draw fast 2D line between specified starting and ending coordinates
GE_FMOVE	Set the current fast 2D line current drawing position

The Graphics Library does not currently fully support the us8 of these fast line drawing tokens.

Pixel Reads and Writes

The MGR adapter provides hardware and microcode support for reading, writing and copying pixels directly to and from the bitplanes. The various microcode command tokens which are provided are summarized in Table 4.24. The pixel support can be divided into two basic types which are the old style single scan line type and the newer multiple scan line type.

Table 4.24 Pixel Read and Write Token Comparisons

Token	R/W	Max Number of Scan Lines	Max Number of Pixels	Starting Pixel Location	Data Transfer Method	Pixel Zoom support	Pixel Packing Suppor
GE_READPIXELS	R	1	1280	Current Character Position	RE2 to GE5 DRAM DMA then Host DRAM Reads	No	No
GE_READPIXDMA	R	1	1280	Current Character Position	RE2 to Host DMA	No	No
GE_RECTREAD	R	1024	7280	X, Y Data Parameters	RE2 to GE5 DRAM DMA then Host DRAM Reads	No	No
GE_READBLOCK	R	1024	1280 X 1024	X, Y Data Parameters	RE2 to Host DMA	No	No
GE_WRITEPIXELS	w	1	1280	Current Character Position	Host FIFO Writes then GE5 DRAM to RE2 DMA	No	Yes
GE_RECTWRITE	w	1 024	1280	X, Y Data Parameters	Host FIFO Writes then GE5 DRAM to RE2 DMAs	No	Yes
GE_WRITEBLOCK	w	1024	1280 X 1024	X Y Data Parameters	Host to GE5 DRAM DMAs then GE5 DRAM to RE2 DMAs	Yes	Yes
					Host to RE2 DMAs	No	No
GE_RECTCOPY	R/W	1024	1280 X 1024	Source and Dest x, Y Data Parameters	RE2 to GE5 DRAM DMA then GE5 DRAM to RE2 DMA for each line	No	Yes

For the old type of pixel reads and writes the host software simply specifies the number of pixels to be read or written and the transformed screen coordinate copy of the current character position is used as the starting address for the pixel reads or writes. The pixels can only be read or written from the current scan line. The maximum number of pixels that can be read or written is limited to 1280. These limitations meant that the host user software would have to provide it's own support for rectangle reads and writes by setting the current character position and doing the necessary pixel reads or writes. After the pixels have been read or written the current character position is updated to be to the right of the last pixel accessed. The **GE_READPIXELS**, GE_READPIXDMA and **GE_WRITEPIXELS** tokens are the old style of tokens.

For the newer style of pixel support the host specifies the starting x and y location of a rectangle and the x length and the y length of the sides of the rectangle. This **allows** multiple Scan lines to be read or written in one execution of the token. This means that an entire rectangle can be read or written at once. These tokens do not affect the current character position. The GE_RECTREAD, **GE_READBLOCK, GE_RECTWRITE, GE_WRITEBLOCK** and the GE_RECTCOPY tokens are the new **style.**

Refer to the various token definitions for additional details on each token. For the tokens which Perform host DMA transfers refer to the pixel DMA support paragraphs in the programming considerations section of this chapter. For additional details on the RE2 DMA support refer to the Raster Subsystem chapter.

Reading Pixels

The pixel reads are performed from the bitplanes which have been selected with the GE_READSOURCE token. The pixels are read using either RE2 to Host DMA or the combination of RE2 to GE5 Data RAM DMA followed by the host reading the individual words from the GE5 Data RAM. If the selected bitplanes are the Frame Buffer bitplanes then the buffer which is read is-specified by the GE_READBUF token.

Writing Pixels

The pixel writes are performed to the bitplanes **which** have been selected with the GE_RWMODE token. The pixels are written using either Host to RE2 DMA or the combination of Host FIFO writes to the GE5 data RAM followed by GE5 Data RAM to RE2 DMA. The GE_WRITEBLOCK and the GE_RECTCOPY tokens **allow** the pixels to be zoomed in the x and y dlrectbns. The x and y room factors are specified with the GE_ZOOMFACTOR token. The three pixel writing tokens also support the use of packed pixels in the host words sent to the adapter. When pixel zooming or pixel packing is used the raster operation specified with the GE_RASTEROP function must be set to 3 **(SRC_COPY).** When pixel packing is **not** used then the raster operation can be any legal value.

This completes the discussion of the drawing support provided by the microcode. The following **section** discusses the lighting support provided by the microcode as it executes the drawing tokens.

Lighting Modes

The GE5 microcode provides support for performing lighting calculations needed to greatly enhance the realism of the displayed geometry. The lighting support tokens provided by the microcode are shown in table 4.25. The microcode allows the host software to specify the following parameters:

- light source properties
    - color
    - location
    - direction
- surface material properties
    - ambient light property
    - diffuse light property
    - emission property
    - shininess property
    - specular light property
- lighting model properties
    - viewer position
    - view direction
    - lighting attenuation factor

The microcode supports up to eight local or infinite point light sources. If a lighting mode is enabled then the color the GE5 microcode applies to each transformed coordinate is a function of the coordinate position, the normal direction, the lighting model, the light sources and the characteristics of the surface material. The result of the lighting calculations is a color value which is different than the RGB color or Cobr Index color values specified by the host. The modified color values reflect the effects of the lighting properties described above.

When performing lighting mode calculations it is necessary to separate the Modeling, Viewing and Projection matrix into two separate matrices. The Modeling and Viewing matrix is separated from the Projection matrix. This is so that the normal vectors can be multiplied by the inverse transpose of the Modeling and Viewing matrix. The host software must provide the support for the two transformation matrix stacks. The microcode provides support for a single transformation matrix stack and a normal matrix stack. The GE_MMODE token allows the host software to specify whether the projection matrix is a single matrix or if it is a Viewing/Modeling matrix or a Projection matrix. When it is a single matrix the push and pop matrix tokens only affect the transformation stack and do not affect the normal matrix stack. When the transformation stack is a Viewing or Projection matrix the push and pop matrix tokens affect both the transformation matrix stack and the normal matrix stack. The host software *must use* the GE_LOADMATRIX and the GE_MULTMATRIX to manage the matrix stack. The host software handles the separation of the

Modeling and Viewing matrix from the Projection matrix since their is not a separate matrix stack on the adapter for these separated matrices.

Table 4.25 Lighting Support Tokens

Token	Description
GE_ABNORMAL	Flag used to cause the normal matrix to be recalculated
GE_LIGHTATTR1	Used to load 1 lighting vector parameter
GE_LIGHTATTR2	Used to load 2 lighting vector parameters
GE_LIGHTATTR3	Used to load 3 liahtina vector parameters
GE_LIGHTDATA4	Used to load lighting data parameters
GE_LIGHTDIRECTION	Used to specify the light source direction
GE_LIGHTMEMPTR	Used to set the address pointer to the lighting data in GE5 data RAM
GE LIGHTMOVDATA	Used to move liahtina data uD or down in the GE5 data RAM
GE_LIGHTPOSITION	Used to load the lighting source position
GE_LMCOLOR	Used to set the lighting mode color parameters
GE_LOADAMBIENT	Used to load the ambient lighting parameters
GE_LOADASUM	Used to load the ambient sum lighting parameters
GE_LOADDIFFUSE	Used to load the diffuse reflections lighting parameters
GE_LOADEMISSION	Used to load material emission lighting parameters
GE_LOADLCOLOR	Used to load the lighting source color
GE_LOADNORMAL	Used to load the top of the normal matrix stack
GE_LOADSPECULAR	Used to load the specular reflectance lighting data
GE_MMODE	Used to specify the transformation matrix mode
GE_MULTNORMAL	Used to multiply a normal matrix to the current normal matrix
GE_NORMAL	Used to update the normal vector used in lighting calculations

The normal matrix stack is used to calculate the lighting normals. The Modeling and Viewing matrix is used to transform the world coordinates into eye coordinates. The inverse transpose of the Modeling and Viewing matrix is used to transform the normal vector from world coordinates to eye coordinates. The lighting calculations are then performed using the eye coordinate data. These calculations produce the color data which is used to as the pixel color data. The Projection matrix is used to transform the eye coordinates into normalized coordinates and the viewport specification is used to transform the normalized coordinates into screen coordinates. The screen coordinates are used lo select the pixel location which will be rendered with the calculated color values.

The next sections will describe the support provided for bounding boxes, feedback and picking.

Bounding Box Support

The microcode provides support for culling operations. The culling operation determines which parts of a drawing are below a minimum s&e and thus are too small to draw. The objects which are smaller than the minimum size are not drawn. The tokens which support the culling operation are shown in Table 4.26.

Table 4.26 Bounding Box Tokens

Token	Description
GE_BEGINBBOX	Used to begin bounding box mode
GE_ENDBBOX	Used to end bounding box mode

The GE_BEGINBBOX token is used to specify the xmin and ymin values used to check for minimum feature size and places the microcode into bounding box mode.The GE_ENDBBOX token is used to end the bounding box mode.

Feedback Support

The microcode provides support for feedback operations which are used to allow the host software to get the transformed and clipped data from the Geometry Pipeline. The tokens shown in Table 4.27 are provided for feedback support. The GE-FEEDBACK token is used to cause the microcode to enter the feedback mode of operation. While in feedback mode the output of the Geometry Pipeline are placed in the feedback buffer. When the buffer becomes full a GE interrupt is generated to inform the host that the feedback data needs to be read from the buffer. The GE_PASSTHROUGH token is used to place known marker values in the feedback buffer to help the host software determine whether some of the data sent to the Geometry Pipeline has been clipped out and is not in the pipeline. The GE_ENDFEEDBACK token is used to take the microcode out of the feedback mode. When this token is received by the microcode it generates a final GE interrupt to allow the host to get the partially full feedback buffer.

Table 4.27 Feedback Tokens

Token	Description
GE_ENDFEEDBACK	Used to end feedback mode
GE_FEEDBACK	Used to begin feedback mode
GE_PASSTHROUGH	Used to place markers in the feedback data

All the graphical objects are transformed, clipped and scaled by the viewport specifications. The lighting calculations are performed and the resulting data is placed in the feedback buffer. Because of clipping, more or fewer vertices may appear in the feedback buffer than were sent to the geometry engine.

After a feedback session, the feedback buffer can contain any or all of the following data:

- points	**FB_POINT(95.0)**
- moves	**FB_MOVE(103.0)**
- draws	**FB_DRAW(113.0)**
- polygons	**FB_POLYGON(72.0)**
- character moves	**FB_CMOV(163.0)**
- passthrough markers	**FB_PASSTHROUGH(176.0)**
- Z buffer data	**FB_ZBUFFER(46.0)**
- linestyle patterns	**FB_LINESTYLE(76.0)**
- setpattern values	**FB_SETPATTERN(133.0)**
- linewidth values	**FB_LINEWIDTH(77.0)**
- lsrepeat values	**FB_LSREPEAT(90.0)**

Each of the above data types **Is** assigned a floating point coda value **which** is deflned in the file **feed.h.** The values from feed.h are shown above for the data types shown. Obviously if any differences exist between the values shown above and **feed.h** then feed.h is correct.

The vertex data for the points, moves and draws always appear in groups of 6 floating point values which follow the floating point code number. The following data format would be in the feedback buffer for a point:

```
95.0
x, y , z, r, g, b
```

The x, y and **z** values are In screen coordinates but not window coordinates. The **r,** g and b values are the color values that would be written Into the frame buffer at the vertex. The **r** value will be in the range of 0 to 255 in RGB mode and in the range 0 to 4095 in Color index mode. The g and b values are always in the range of 0 to **255 In** RGB mode and are undefined in Color Index mode.

For polygons the data in the feedback buffer **is** the coda value for polygon followed by a count value which **specifies** how manv vertex data values are present for this polygon entry In the feedback buffer. **The** count will always be a multiple of six since the vertex **data** consists of six values as **shown** above. The foliowing example shows the **polygon** format:

```
72.0  16.0
x1, y1 , zl, rl, gl, b1
x2, y2, z2, r2, g2, b2
x3, y3, z3, r3, g3, b3
```

The character move type consists 'of the **code** value followed by the x, y and z screen coordinate values as shown below:

```
163.0
x, y, z
```

The rest of the commands **(FB_PASSTHROUGH, FB_ZBUFFER, FB_LINESTYLE, FB_SETPATTERN, FB_LINEWIDTH, FB_LSREPEAT)** have only their code values and a single value placed in the feedback buffer. The following example shows the **FB_PASSTHROUGH** format:

```
176.0
value
```

While in feedback mode the host can sand any of the tokens to the microcode but only those shown above **will** be **placed** in the feedback buffer. The others will *cause* their actions to be taken but **will** not produce any entries in the **feedbsck** buffer.

Picking and Selecting Support

The microcode provides support for picking and selecting objects which are drawn near the cursor location. The to&ens which are provided for picking and selecting support are shown in Table 4.28. The **GE_PICKMODE** token is used to place the microcode in picking or selecting mode. While in picking or selecting mode nothing is actually displayed on the screen.

The host software specifies the picking region by using the **GE_LOADMATRIX** token to toad a special pick projection matrix which causes the pick region to fill the entire viewport. The normal projection matrix and the viewing and modeling matrices would then be concatenated to the special picking matrix using the **GE_MULTMATRIX** token. Any world coordinate which is transformed and is not clipped is therefore within the pickIng region.

The host software specifies the selecting region by concatenating a special selecting viewing matrix to the already specified projection matrix using the GE_MULTMATRIX token. Any world coordinate which is transformed and is not clipped is therefore within the selecting region. The selecting mechanism is a more general form of picking. Other than the differences in how the transformation matrix is modified the picking and selecting modes operate the same.

Table 4.28 Picking and Selecting Tokens

Token	Description
GE_ENDPICKMODE	Used to end pick mode
GE INITNAMES	Used to initialize the name stack
GE LOADNAME	Loads a name on the top of the name stack
GE_PICKMODE	Used to begin pick mode
GE_PICKTYPE	Used to specify the pick mode type
GE_POPNAME	Used to pop the names on the name stack
GE PUSHNAME	Used to push the names on the name stack

The key to the picking and selecting support is the name stack which is used to indicate which items are within the picking or selecting region. The name stack comes into play for objects which are picked or selected. The name stack is used to store marker names which are copied to the pick buffer when an object is picked or selected. The **GE_INITNAMES** token is used to initialize the name stack and leave it empty. The GE_LOADNAME token is used to place a name marker on the top of the name stack. The name markers are 32 bit integer values. The GE_PUSHNAME token is used to push the names on the name stack down a level and to add a new name to the stack. The GE_POPNAME token is used to pop all the names on the name stack up one level. When a pick or select occurs the contents of the name stack are copied to the pick buffer.

The pick buffer is an area of the GE5 data RAM whii is also used as the feedback buffer. This means that picking and feedback modes cannot be used at the same time. The **GE_PICKTYPE** token is used to specify when the data on the name stack is copied to the pick buffer. and where in the buffer the data is placed. When the pick type is set for pick none the microcode is in pick mode but nothing is ever recorded in the pick buffer. When the pick type is set for pick first the first time a pick occurs the name stack is copied to the pick buffer and all other picks are ignored. When the pick type isset for pick last each pick causes the name stack to be written to the beginning of the pick buffer. When the pick type is pick all each pick causes the name stack to be concatenated to the current data in the name stack. When the pick buffer becomes full the microcode generates a GE interrupt to inform the host to read the pick buffer. The GE_ENDPICKMODE token is used to take the microcode out of pick mode. The microcode generates a final interrupt to inform the host to get the last partial pick buffer when It receives the GE_ENDPICKMODE token. This is how the host gets the pick data for the pick first and pick last types. The pick none type still generates the interrupt but no data is present in the pick buffer. The default pick type is pick all.

GE5 Pixel Rendering Support

The GE5 does the first stage of scan conversion. The Raster Subsystem can only draw lines and spans. The GE must pass the appropriate sbpe information for fines. Scan conversion of polygons requires the GE5 to break the polygon into trapezoids, to calculate the edge slopes of each trapezoid and to iterate depth and color components along the slope. This data is then passed to the Raster Subsystem for pixel rendering.

The GE5 performs scan conversions using floating-point precision to maintain coordinate integrity during iterations and slop8 calculations. It also computes the x coordinates of polygon edges to a fractional pixel tolerance. The GE5 first corrects all depth and color component iterations to the nearest pixel center and then iterates them in full pixel steps. As a result, iterated color and depth values remain planer across polygonal surfaces and subsequent Zbuffer calculations result in clean intersections.

The calculated x and y screen coordinates are used by the. Raster Engine in addressing the various bitplanes. The z screen coordinates are used by the Raster Engine to perform Z buffer calculations. The x screen coordinate values are converted to a special DIVMOD format in which they are divided by 5 and the quotient is used as an offset into the VRAM chips and the remainder is used to select which of the five pixel pipelines the pixel is written into. Actually the divmod formatting is done by doing a table lookup in the constants portion of the GE5 data RAM. The concepts are described in greater detail in the Raster Subsystems chapter.

The next paragraph describes some of the tokens which are provided to control the pixel formats and attributes.

Pixel Format and Attribute Support

The tokens shown in Table 4.29 are provided to control the pixel format and attributes. The GE_PIXTYPE token is provided to select the format of the pixels written into the Frame Buffer bitplanes. Four types of pixels are supported by the Raster Engine and include the 24 bit RGB pixels, 12 bit RGB double buffered pixels, 8 or 12 bit Color Index double buffered pixels and 4 bit Color Index double buffered pixels. The GE_RGBCOLOR token is used to specify the color components for the RGB pixel types and the GE-COLOR token or theGE_COLORF token is used to specify the color index value for the Color Index pixels.

Table 4.29 Pixel Format and Attribute Tokens

To&en	Description
GE_AUXWRITEMASK	Used to write the AUXMASK value
GE_COLOR	Used to specify an integer color index
GE_COLORF	Used to specify a floating point color index
GE_DRAWMODE	Used to specify the drawing mode
GE_ENABDITH	Used to enable or disable dithering
GE_PIXTYPE	Used to specify the pixel format
GE_PIXWRITEMASK	Used to write the PIXMASK value
GE_RASTEROP	Used to specify a logical bitwise operation
GE_RGBCOLOR	Used to specify an RGB color
GE SCREENCLEAR	Used to clear the bitplanes other than the Z buffer

The GE_DRAWMODE token specifies which bitplanes are written with the calculated pixel values. The bitplanes which can be written include the Frame Buffer bitplanes, The Pop Up (PUP) bitplanes, the User Auxiliary (UAUX) bitplanes and the Window ID (WID) bitplanes. The Frame Buffer bitplanes are used to display the image on the screen. The PUP bitplanes are used by the Window Manager software to draw overlays on the screen such as menus. The UAUX bitplanes allow a user graphics application to draw overlays on the screen. The Window ID bitplanes are used to control pixel clipping for obscured windows. They also are used by the Display Subsystem to control the pixel display formatting.

The GE_SCREENCLEAR token is used to clear the bitplanes selected with the GE_DRAWMODE token to the color specified with the GE_RGBCOLOR, GE-COLOR or GE_COLORF tokens. If the selected bitplanes are the Frame Buffer then the color token used depends on the GE_PIXTYPE. For the PUP, UAUX and WID bitplanes the GE-COLOR or GE_COLOLRF token specifies the color since these bitplanes represent color Index type of pixels.

The actual bits if each pixel which are written depend on the value in the pixel write mask and the aux write mask. The GE_PIXWRITEMASK is used to set the write mask which controls which bits of the pixels are written into the Frame Buffer bitplanes. The GE_AUXWRITEMASK isused to set the aux mask which controls which bits are written into the PUP, UAUX, WID and Z Buffer bitplanes. The bits which have a corresponding 1 bit in the pixel mask or aux mask are written and the bits whose corresponding mask bits are zero are not written. These two write masks are applied on a bit by bit basis for each pixel write operation by the RE2. The exception is for the Z Buffer bitplanes where if the single Z mask bit is a one the 24 bit Z value may be written and if the Z mask bit is a zero then the 24 bit Z value is not written.

The GE_ENABDITH token is used to enable or disable dithering operation for the 12 bit RGB pixels and the Color Index pixels written into the Frame Buffer. The dithering operation is used to smooth the color transitions when a limited number of colors are available. The dithering operation is not valid for the 24 bit RGB pixels. During a screen dear the dithering should be disabled.

The GE_RASTEROP token is provided to specify a logical bitwise operation which can be performed between the new pixel value and the current pixel value already in the bitplanes at the current pixel location. The raster operation can be applied to all of the bitplanes except the Z buffer bitplanes. The default raster op is 3 which is a copy of the new pixel into the bitplanes without doing the logical bitwise operation. When any of the other 15 raster ops is specified the performance of the RE2 in drawing pixels is approximately nines times slower than when the raster op is 3.

These concepts are discussed in much greater detail in the Raster Subsystem and Display Subsystem chapters. The Z buffer checking support is described in the next section.

Z Buffer Support

The MGR adapter provides hardware and microcode support for writing Z values into a Z buffer and for conditioning pixel writes by comparing new Z values with the Z values already in the Z Buffer. For the Z Buffer to be used the optional Z Buffer card must be installed. This card contains 24 bitplanes in a 1280 x 1024 configuration. The tokens which are used by the host software to control the Z Buffer are shown in Table 4.30.

Table 4.30 Z Buffer Support Tokens

Token	Description
GE_AUXWRITEMASK	Used to set the Write Mask for the Z Buffer and other bitplanes
GE-FAR	Clears the Z Buffer and the other bitplanes to the specified values
GE_DEPTHFN	Used to temporarily make Z compares pass while clearing Z Buffer
GE_ZBUFFER	Used to enable or disable Z Buffer mode
GE_ZFUNCTION	Used to set the Z compare function used during Z Buffer mode
GE_ZSOURCE	Used to select the Z compare source (Z Buffer or Frame Buffer)

As the GE5 executes the various drawing tokens it transforms the 3D world coordinates to device coordinates. The modeling, viewing and projection matrix is used to transform the world coordinates to normalized device coordinates and the viewport specification is used to map the normalized device coordinates to actual device coordinates for rendering into the bitplanes by the microcode and the RE2. Refer to the section on coordinate transformations described previously in this chapter for additional details. The x and y device coordinates are used to select which pixel location is to be written into. The z device coordinate can be used to perform depth comparisons to control whether the pixel is written at the current pixel location. The depth comparison is performed by the RE2 using the Z comparator hardware. The GE_ZBUFFER token is used to enable or disable the Z comparisons. If Z Buffer mode is enabled the Z comparisons will be used to condition the pixel writes and if the Z Buffer mode is disabled the Z comparator is set so that the Z comparisons always pass and therefore do not prevent any pixel writes.

The z values in the Z Buffer are 24 bits wide and are treated by the Z comparator hardware as signed integers in 2's complement format. The GE_VIEWPORT token which is used to specify the device coordinate mapping can therefore specify a device mapping for the z axis with a range of 0x800000 to 0x7FFFFF. The z axis positive direction depends on the projection matrix which is loaded into the MVP matrix. The Graphics Library sets the projection matrix for a left hand screen coordinate system so the z axis direction is into the screen and away from the viewer. This means that the z values closer to the viewer are less than those that are further from the viewer. The GE_ZFUNCTION token is used to specify the relational comparison function used by the Z comparator hardware. The default setting when Z Buffer mode is enabled is to set the compare function to less than or equal which allows only the pixels closer to the viewer to be drawn. Those pixels which are further from the viewer than previously drawn pixels would not be drawn. If the projection matrix specifies a right handed coordinate system then the positive z axis direction is out of the screen and toward the viewer so the relational comparison would beset to greater than or equal for pixels closer to the viewer to be drawn.

The Z comparator hardware has two modes of operation in which it can do comparisons between different sources as specified by the GE_ZSOURCE token. The normal mode of operation is to compare new Z values with the Z values in the Z Buffer. A second mode is provided to support the drawing of antialiased lines. This mode compares new color values with the color values in the Frame Buffer bitplanes. This mode is only valid when the GE_ANTIALIASE token has been used to select an antialiase drawing mode. For this mode the z compare function set with the GE_ZFUNCTION token would be to allow pixels with a color value greater than or equal to the current color to be drawn. This makes intersecting antialiased lines to have an improved appearance by allowing the brighter pixels at the intersections to be drawn.

The GE_AUXWRITEMASK token is used to specify the write mask for the PUP, UAUX, WID and Z Buffer bitplanes. Bit 8 of the AUXMASK controls the writes to the Z Buffer. When this bit is 1 all 24 bits are written to the Z Buffer and when this bit is 0 the 24 bits cannot be written into the Z Buffer. Unlike the other bitplanes there is no bit by bit writer mask for the Z Buffer. The GE_DEPTHFN token provides a mechanism for specifying the Z comparator function. It has the same format as the GE_ZFUNCTION token but it can be used when the Z Buffer *mode is* disabled.

Normally the drawing operations of the GE5 cause the z values to be written into the Z buffer. The host software would not normally write directly to the Z Buffer other than to clear the z values to a known state. However if the host needs to read or write to the Z Buffer directly it can use the pixel read and write tokens discussed previously in this chapter.

Two methods are provided for clearing the Z Buffer and include the fast Z clear method and the GE_CZCLEAR token. After the Z Buffer is cleared the first Z compare at each pixel location will pass. After the first compare the remaining compares will depend on the relational Z compare function and the current values in the Z Buffer. The fast Z clear method is used on the enhanced adapter and is invalid on the base adapter. The GE_CZCLEAR token can be used to clear both the Frame Buffer bitplanes and the Z Buffer bitplanes to the specified color and Z value respectively. This method is always used on the base adapter since the fast Z clear method is invalid. The GE_CZCLEAR token could be used on the enhanced adapter if the host software desired to clear the Z Buffer to a specific value. The two methods of clearing the Z Buffer are discussed in greater detail in the Raster Subsystem chapter.

Miscellaneous Support

The tokens shown in Table 4.31 are provided to implement functions that do not functionally match any of the functional groups discussed previously.

The **GE_FINISH0** token is provided to allow the host software to put the microcode in a state which will allow the host software to access the **GE5** data RAM. The microoode must be in a stalled state when the host attempts to access the data RAM. The host software clears Finish flag 0, sends the **GE_FINISH0** token to the microcode and then polls finish **flag** 0 until it is set or a time out occurs. After the microcode has finished processing all of the other tokens in the FIFO it finally executes the **GE_FINISH0** token. This token causes the microoode to set finish flag 0 and to execute a fetch token. Since the host has not sent any additional tokens the FIFO **is** empty and so the fetch will cause the microcode to stall. The host can then access the microcode data **RAM** without affecting the microcode. If a timeout occurs it is an indication that the microcode may already be stalled which would indicate that some activity the host software engaged in has caused a problem for the microcode which caused it to stall. It could also indicate a hardware problem.

Table 4.31 Miscellaneous **Tokens**

Token	Description
GE_FINISH0	**Used to cause the microcode to set Finish flag 0**
GE_HQMSAV	**Used to cause the the** microcode to save **the** MAR register
GE_LOADGE	**Loads the specified data** at the **specified address in data RAM**
GE_LOADRE	**Loads the specified data in the specified RE register**
GE_VERSION	**Returns the microcode version string**

The GE_HQMSAV token is provided to allow the host software to keep a copy of the page *address written* into the HQMAR register in the GE8 data RAM so that it **will** be saved as part of the graphics **context.** This is necessary because the host software cannot read the HQMAR register. This means that if the **current** context were switched out of the adapter and another context changed the HQMAR register then when the original context it would have no way of restoring the proper value in the HQMAR register if it were not saved in the current context. The GE_HQMSAV token has a data parameter which is the page address that is going to be written into the HQMAR register following the GE_HQMSAV token being sent to the microcode. The **microcode** reads the page address from the data FIFO and places it in the GE8 data RAM **location** HQMSAV.

The GE_LOADGE token allows the host to have the **microcode** write a value into a data RAM location without having to go through the procedure of using the **GE_FINISH0** to cause the microcode to go **into** a fetch stall, changing the HQMAR page address and then writing the value into the data RAM.

The GE_LOADRE token allows the host software to bad a value into an RE2 register. Generally the host software should never change the **RE2** registers. The exception to this are the **TOPSCAN** and ENABRGB registers. The **TOPSCAN** register specifies the starting row and column used to display the data in the VRAM bitplanes. The ENABRGB register is used to enable the 8 bit RGB mode for the base adapter. These registers are described in greater detail in the Raster Subsystem chapter.

The GE-VERSION token is used to read the microcode version number from the microcode data ram. This token causes the version number to be placed in the pick buffer pointed to by the address in the data ram variable PICKPTR. Since the version number would overwrite the first word of the pick data this token should not be sent while picking or feedback mode are active. The version number **consists** of a 32 bit value.

Programming Considerations

The following paragraphs describe the programming considerations for the Geometry Subsystem. The programming considerations for the Geometry Subsystem are divided into the following areas:

- Hardware Component Programming
- Inltialitation Programming
- 3D Graphics Programming
- Win&w and Context Management Programming

The hardware components programming section describes the macros that Silicon Graphics has developed for accessing the various components. These macros are defined in the file mgr.h which should be considered as the correct version should any differences exist between the examples shown in this document and the contents of the **mgr.h** file.

Each section also shows example code fragments showing the usage of the macros. Examples are also shown for how to program various functional requirements such as how to download the microcode. The examples are taken from various Silicon Graphics source code and represent code fragments **which** are not **necessarily** intended to represent the exact **real world** usage but are rather a guideline as to the usage ***of*** the macros and of how ***to*** accomplish various functional tasks.

Another important point to remember is that this document is only intended to offer technical information about the **MGR** adapter. The programming examples do not try to define in what type of host software they would reside such as a kernel device driver, window manager or Graphics Library. The necessary host software data structures and other programming requirements have intentionally been abstracted as much as possible. This document is only intended to show the interface requirements of the **MGR** adapter.

Hardware Component Programming

The programming considerations for accessing the hardware components of the Geometry Subsystem include the following **topics:**

- **MGR** Base Address Programming
- **HQ1** Register Programming
- **HQ1** Command Programming
- Finish Flag Programming
- Microcode Code RAM Programming
- Microcode Data RAM Programming
- GE and FIFO Half-Full Interrupt Programming
- FIFO Programming

The **followinq** paragraphs describe the programming considerations for accessing the various "Geometry Subsystem hardware components.

MGR Base Address Programming

To access any of the hardware components on the MGR adapter the base memory address of the card must be determined. This is done during the initialization of the card as described in the Host Interface Subsystem chapter. The physical memory address of the card is programmed into the POS subaddress registers. The MGR adapter occupies a 32K byte range of addresses beginning at the base address programmed into the POS subaddress registers. If the operating system of the host system uses physical addresses only such as MSDOS then the **physical** addresses can be used to access the hardware components on the adapter. In this case the base address is simply placed in the variable GRP. The address for a particular component is formed by adding the address offset of the hardware component to the value in GRP. This address is then used to read or **write** the desired component.

If the operating system provides a virtual to physical address translation mechanism such as UNIX or OS/2 then the physical address programmed into the POS subaddress registers and the associated 32K byte range of addresses must be mapped to a 32K virtual address range. The virtual address is a kernel virtual address if the software is part of the kernel such as a kernel device driver. The virtual address is a user virtual address if the software is part of a user application program such as the Graphics Library. The virtual to physical address mapping technique is host operating system dependent and is beyond the scope of this document. In any case the virtual address is placed into the variable GRP. The address for a particular component is formed by adding the address offset of the hardware component to the value in GRP. This address is then used **to** read or write the desired component.

The variable GRP would be defined as an unsigned tong and would have the physical address assigned to it.

```
unsigned long       GRP;
```

The macro **GRPsetup** is defined in the file mgr.h and k used to dedare GRP as an extem.

```
#define GRPsetup      extem unsigned long GRP
```

The user mode programs would assign the **user** virtual address to GRP and the kernel device driier would perform the necessary user virtual to physical address mapping. The user virtual address shown here is 0x1000 but it could be any virtual address which has the least significant 12 bits **equal** to zero.

```
#define   GR1_GFXPG_VADDR  0x1000

#define   GRP   GR1_GFXPG_VADDR
```

The use of the GRP variable is shown in later paragraphs when the various macros are defined.

HQ1 Register Programming

The **HQ1** has two host programmable registers **which** can be written by the host but cannot be read by the host. These are the HQ Mile Address Register (MAR) and the HQ MAR Most **Significant Bit** (HQMMSB) Register. The Registers section of this chapter defines how these two registers are used. The include file **mgr.h** contains macros which are used to write these two registers.

The HQMMSB is a 1 bit register which is used as an address control bit to select which **MGR** adapter component is addressed. The HQMMSB register is set and cleared by writing to two different addresses. The data value is unused and should be set to zero. The macro **HQMMSB_WR** allows the host to set or clear the HQMMSB register. The macro takes the value of x and shifts it left twice and then does an or of that value with the MGR adapter base address GRP and the offset address of the HQMMSB register. The value of bit 2 in the host address is placed in the register. If x is zero then the HQMMSB register is cleared and if x is a one then the register is set.

```
#define HQMMSB_OFF        0xE00 /* HQMMSB base address */

#define   HQMMSB_WR(x) \
    *(volatile long *)(GRP | HQMMSB_OFF | ((x) << 2)) = 0
```

The HQ MAR is an address page register which is used to provide the upper 7 bits of the microcode code RAM address when it is accessed by the host. It Is also used to provide the upper 5 bits of the microcode data RAM address when it is accessed by the host. When a write is done by the host to the HQ MAR register address, bits 8-2 of the host address are placed in the HQ MAR register.

The macro **HQM_WR** is provided to write the new page address into the HQMAR register. The macro take the desired address value and shift bits 15-8 right 8 bits. The bits are then shifted left 2 bits to put it into bits 8-2 as needed by the adapter. This value Is then **ored** with the HQ MAR base address to form the complete host address. When this address is written to the adapter **the page** address in bits 8-2 is written into the HQ MAR register. This macro is used In device drivers or in code that cannot have a context switch occur while it is being executed. The HQMSAV location in the GE5 data RAM can be read to get the value to restore Into the HQ MAR register.

```
#define HQM_OFF           0xC00  /* HQ MAR base address */

#define   HQM_WR(addr) \
    '(volatile long ●  )(GRP | HQM_OFF |  ((addr)  >> 8) << 2)) = 0

/* The HQ MAR can address a page size of 256 words In the data or code RAM before it must be
changed to address the next page. ●  /

#define HQM_PG_SIZE        256
```

The macro **GL_HQM_WR** sends the token **GE_HQMSAV** and the new page address value down the FIFO to the microcode so that the microcode will **save** the page address in the microcode data RAM in address HQMSAV. The new page address is then written into the HQ MAR register by the host using the HQM_WR macro. This macro is used when setting the HQ MAR from user mode **code** such ***as the* graphics library.** This **is** done so that if a context switch occurs after the HQ MAR is set the context manager code can restore the HQ MAR register when the context is restored. This is necessary since the HQ MAR register cannot be read by the host.

```
#define  GL_HQM_WR(addr) { \
   '(volatile long *)(GRP | FIFO-OFF | ((GE_HQMSAV) << 2)) = 0; \
    '(volatile long *)(GRP | FIFO-OFF | ((GE-DATA) << 2)) = addr; \
    '(volatile bng *)(GRP | HQM_OFF | ((addr) << 2)) = 0: }
```

The HQ1 also contains the PC register which can be read directly by the host using the HQ1 HQRDPC command described in the next section. The PC can also be written indirectly by reading from the microcode code RAM.

Example Usage:

```
#include "mgr.h"

GRPsetup;           /* declare GRP extern  • /

HQMMSB_WR(1);           /* set HQMMSB register •  /

HQMMSB_WR(0);           /* clear HQMMSB register */

/* set HQ MAR to PICKPTR address in microcode dataRAM*/

HQM_WR(PICKPTR);

/* send GE_HQMSAV token and CPOSX address to microcode to save new HQ MAR address in
 microcode data RAM. Then sat HQ MAR to CPOSX address.'

GL_HQM_WR(CPOSX);
```

HQ1 Command Programming

The HQ1 accepts three commands from the host. These commands are clear stall, clear GE interrupt and read HQ1 PC register. These commands are issued by doing a read or write to the appropriate address as shown in Table 4.2. Three macros are provided in mgr.h to issue the commands. The HQCLRSTL macro is used to clear a microcode stall condition. The HQRDPC macro is used to read the HQ1 PC **register.** The PC is 15 bits wide so the 16 bits of data which are read are masked to 15 bits. The HQCLRINT macro is used to clear the GE interrupt which was generated by the microcode.

```
#define HQCLRSTL_OFF 0x640        /* dear stall cmd address'/

#define HQRDPC_OFF  0x740         /* read HQ PC cmd address */

#define HQCLRINT_OFF  0x780       /* clear GE int cmd address */

#define HQCLRSTL() \
    *(volatile long  *  )(GRP | HQCLRSTL_OFF) = 0

#define HQRDPC(x) \
   x = '(volatile short *   )(GRP | HQRDPC_OFF) & 0x7FFF

#define HQCLRINT() \
    *(volatile long *)(GRP | HQCLRINT_OFF) = 0
```

Example Usage:

```
#include "mgr.h"

int        addr;

GRPsetup;

HQCLRSTL();            /* clear the stall */

HQRDPC(addr);          /* read the PC into addr */

HQCLRINT();            /* clear the GE interrupt */
```

Finish Flag Programming

The Geometry Subsystem contains two finish flags. These flags are used to synchronize host and microcode execution. The host software clears the appropriate finish flag and then polls the finish flag address waiting for it to be set by the microcode. The flag being set indicates that the microcode is in a known state and the host can proceed with a desired operation.

finish flag 0 is used to indicate to the host that the microcode is in a stalled state so that the microcode can read the GE5 data RAM. To use finish flag 0 the host clears finish flag 0 by writing a zero to its address and then sends the **GE_FINISH0** token to the FIFO. The host then polls finish flag zero until it is set by the microcode. The microcode continues executing tokens which are in the FIFO until it gets the **GE_FINISH0** token. It then sets finish flag 0 and does a fetch to get the next token. Since the host has not sent any additional tokens the FIFO will be empty and the microcode will stall. At this time the host can safely access the microcode data RAM.

Finish flag 1 is used to indicate to the host that the microcode is ready for a DMA operation to begin or to indicate that a DMA operation has completed. In the case where the host wishes to do an output DMA operation to the MGR adapter, it clears finish flag 1 and then writes the appropriate token (such as **GE_WRITEBLOCK)** to the FIFO to tell the microcode of the output DMA operation. It then polls finish flag 1 until it has been set by the microcode. After the microcode sets Finish flag 1 the host then starts the DMA. In the input DMA case, the host clears finish flag 1 and then writes the appropriate token (such as GE_READBLOCK) to the FIFO. It then polls finish flag 1 until it is set by the microcode which indicates that the-host can start the input DMA operation. For those tokens which cause the microcode to do a DMA from the RE2 to the pixel buffer in the GE5 data RAM Finish flag 1 is used to indicate to the host that the DMA has been completed and the host can read the pixel buffer.

Three macros are provided in mgr.h to access the finish flags. the **FINISH_RD** macro is used to read either finish flag 0 or 1. The FINISH_WR macro is used to write a value to either finish flag 0 or 1. The **FINISH_POLL** macro is used to return the current boolean state of either finish flag 0 or 1.

```
#define   FINISH0_OFF   0x2000   /* Finish flag 0 address */

#define   FINISH1_OFF   0x2004   /* Finish flag 1 address */

#define   FINISH_RD(addr, x) \
    x= '(volatile long *)(GRP| (addr)) & 0x1

#define   FINISH_WR(addr, x) \
    '(volatile long *)(GRP| (addr)) = x

#define   FINISH_POLL(addr)\
    ('(volatile long *)(GRP| (addr)) & 0x1)
```

Example Usage:

```
#include "gecmds.h"
#include "mgr.h"

int       flag1;

GRPsetup;
```

```
FINISH_WR(FINISH0_OFF, 0); /* clear finish flag 0 */

GE[GE_FINISH0].i = 0; /* Tell microcode to set finish 0 */

while (1) {
    if (FINISH_POLL(FINISH0_OFF)
            break;
}

FINISH_RD(FINISH1_OFF, flag1); /* read finish flag 1 */
```

Microcode Code RAM Programming

The microcode code RAM is 40 bits wide and k accessed in two parts. The bwet 32 bits are programmed with the HQMMSB register set to zero. The upper 8 bits are programmed with the HQMMSB register set to one. The **code** RAM k **divided** into 256 word pages and the HQ MAR register must programmed to contain the page address **which** represents the upper 7 bits of the code RAM address. Bits 9-2 of **the** host address represent the word address in the current page and form the bw 8 bits of the microcode code RAM address. The code is then read or written with the HQMMSB register set appropriately for the upper or bwer **part** of the microcode word. As the code RAM is accessed the address must be checked for 256 word page boundaries and the HQ MAR register updated as the page boundaries are crossed.

Two macros are provided to read and write the microcode code RAM. The **URAM_RD** macro k used to read either the lower 32 bits or the upper 8 bits of the microcode word. The **URAM_WR** macro **is** used to write either the **lower** 32 bits or the upper 8 bits of the microcode word.

```
#define URAM_OFF 0x000 /* microcode code RAM base */

#define   URAM_RD(addr, x) \
   x = *(volatile long *)(GRP|URAM_OFF| (((addr) & 0xFF) << 2))

#define   URAM_WR(addr, x) \
   *(volatile long *)(GRP|URAM_OFF| (((addr) & 0xFF) << 2)) = x
```

Example Usage:

```
#include "mgr.h"

long        addr, data:

GRPsetup;

/* write to both the low 32 bits and the upper 8 bits */

HQMMSB_WR(0);               /* access low 32 bits */

URAM_WR(addr, data):        /* write lower 32 bits */

HQMMSB_WR(1);               /* access upper 8 bits */

URAM_WR(addr, data); /* write upper 8 bits */

/* read both the low 32 bits and the upper 8 bits */

HQMMSB_WR(0);               /* access low 32 bits */

URAM_RD(addr, data):        /* read lower 32 bits */

HQMMSB_WR(1);               /* access upper 8 bits */

URAM_RD(addr, data);        /* read upper 8 bits */
```

Microcode Data Ram Programming

The **microcode** data RAM contains **constants, variables** and data buffers used by the microcode during its execution. The host software reads the **microcode** data RAM to get return data from the microcode after some tokens have been executed. The various symbolic names and address constants are **defined** in the include file g8Sgtob.h. The microcode data RAM address locations **used** by the host to **access** microcode return values are defined in the token definition section.

The microcode data RAM is 32 bits wide and Is accessed with the **HQMMSB** register set to zero. The data RAM is **divided** into 266 word pages and the HQ MAR register must programmed to contain the page address which represents the upper **5** bits of **the code** RAM address. Bits 9-2 of the host address form the bw 8 bits of the microcode data RAM address.

Two macros are provided to read and write the microcode data RAM. The **DRAM_RD** macro is used to read the microcode data RAM. The **DRAM_WR** macro is **used** to write the microcode data RAM.

```
#define    DRAM-OFF    0x1400 /* microcode data RAM base ● I

#define    DRAM_RD(addr, x) '((long *)&x) = \
    '(volatile long *)(GRP + DRAM-OFF | (((addr) & 0xFF) << 2))

#define    DRAM_WR(addr, x) \
    '(volatile long *)(GRP + DRAM-OFF | (((addr) & 0xFF) << 2)) = ((long *)&x)
```

Example Usage:

```
#include "mgr.h"

long        addr, data;

GRPsetup;

HQMMSB_WR(0);              /* access data RAM 32 bits */

DRAM_WR(addr, data);       /* write 32 bits */

DRAM_RD(addr, data);       /* read 32 bits */
```

GE and FIFO Half-Full Interrupt Programming

The Geometry Subsystem can generate two hardware interrupts which go to the Host Interface Subsystem. These interrupts are the GE interrupt and the FIFO Half-Full interrupt. The Host Interface Subsystem contains the interrupt mask register and the interrupt status register which the host system uses to enable interrupts and to determine which interrupt caused the host system interrupt. Refer to the Host Interface Subsystem chapter for programming considerations for these registers.

The GE interrupt is latched in the HQ1 and must be cleared by the host software. This is done by using the HQCLRINT wmmand as described above. The host should clear the GE interrupt in the HQ1 before it clears the GE interrupt bit in the interrupt status register. This prevents the possibility of a false GE interrupt from occurring.

The FIFO Half-Full interrupt follows the half-full state of the FIFO hardware. When the FIFO goes above half-full the interrupt is generated as an edge triggered event. The host software clears the FIFO half-full bit in the interrupt status register and does not have to issue any commands to the Geometry Subsystem hardware. The Host Interface Subsystem provides two status bits which can be polled to determine when the FIFO has gone back below the half-full state.

If the host software does not use the FIFO Half-full interrupt to notify it of potential FIFO overflows it must poll the two status bits to check for FIFO half-full conditions. The GEWAIT macro is provided to handle this case. For FIFO writes from device drivers in kernel mode the polling does not have to be done as long as the number of FIFO writes would not cause a FIFO overflow. In the Silicon Graphics graphics library this macro is placed before all token writes to the FIFO to check the FIFO status. If either status bit is set then the macro delays until they both become cleared indicating that the FIFO is no longer half-full. If the interrupt method is used then GEWAIT should be defined to be nothing. If the polling method is used then the GEWAIT macro is left defined as shown below.

```
#ifdef    NO_FIFO_HF_INT
#define   GEWAIT \
    while (('(volatile char •    )GFX_CTRLO_ADDR & GFX_FIFOSTAT));
#else
#define   GEWAIT
#endif
```

Example Usage:

```
#include "mgr.h"

int       color = 3;

/* GEWAIT will poll the FIFO half-full status if the FIFO Half-full interrupt is not used. */

GEWAIT;
GE[GE_COLOR].i = 0;
GE[GE_DATA].i = color:
```

FIFO Programming

The FIFO is used as a buffer between the host and the microcode to allow the host execution to proceed in parallel with the microcode execution of tokens. The host sends tokens and data to the microcode by writing to a 1K byte range of addresses reserved for the FIFO. For host addresses within the FIFO address range the hardware takes bits 9-2 of the host address and places those 8 bits in the Tag FIFO. The hardware also places the 32 bits on the data bus in the Data FIFO.

When a command token is written the data value should be zero. The exception is when the GE-DATA token is written, then the data should contain the desired data parameter. Each byte in the Tag FIFO is used by the HQ1 as an index into the microcode branch table. The branch instruction located at the address pointed to by the token index points to the microcode function which will perform the necessary operations for the token.

The host can write tokens and data parameters using several different methods. Which method is used is dependent on the type of host operating system and whether the token is being written from a kernel mode device driver or from a user mode application program. The MGR card is memory mapped so the tokens and data can be written to the FIFO using a pointer to reference the FIFO. The following examples demonstrate the different methods as used by Silicon Graphics in kernel mode device drivers and in user mode graphics library code.

Kernel Mode Device Driver

Kernel mode device drivers can access the MGR adapter using kernel virtual addresses. The include file mgr.h contains various macros which are provided for writing tokens and data to the FIFO.

The macro **FIFO_WR** is. used *to* write a token or data to the FIFO from kernel device drivers. The token cmd is left shifted twice and **ored** to the FIFO address. This places the token in the address in bits 9-2 as described above. The data is then written to the desired FIFO address. The following code fragment shows these macros.

```
#define    FIFO-OFF    0x800        /* FIFO base address */

#define    FIFO_WR(cmd, x) \
    '(volatile long *)(GRP| FIFO-OFF | ((cmd) << 2)) = '((long') &(x))
```

The GRP variable will be assigned to the kernel virtual address of the adapter as discussed in the Base Memory Address Programming section of this chapter.

If the kernel driver needs to send a floating point constant it should declare a data variable which is assigned the floating point constant so that the compiler will calculate the floating point value and store **it's** hex equivalent in the data segment of the kernel driver. The **FIFO_WR** macro gets the address of the data variable and casts the address into a long pointer and then gets the data variables hex value and writes it to the address formed using the cmd. This technique allows the kernel driver to send either integer variables or floating point constants to the FIFO.

Example Usage:

```
#include "mgr.h"

GRPsetup;

int        one = 1:
    FIFO_WR(GE_DATA, one):          /* such as one of the 3 download parameters */
```

User Mode Program using Virtual Memory

The Silicon Graphics graphics library uses a user virtual address pointer to access the FIFO. The kernel graphics output device driver will perform the necessary user virtual address to physical address mapping to map the user virtual address to the physical address of the MGR graphics adapter.

The include files **mgr.h** and **imsetup.h** contain the following macros which are used to access the FIFO from user mode. The following typedef is used to define the union of tong and floating point values. This is necessary to allow the user mode software to send both floating point and integer data to the FIFO.

```
    typedef union gedata {
                long      i;
                float     f;
    } gedata_t;
```

The **GR1_GFXPG_VADDR** macro is used to define the virtual address assigned to the base **address** of the MGR adapter. The address of 0x1000 is an arbitrary address and can be set to any **virtual** address. The only **limitation** is that the low 12 bits of the address should be. zero, The kernel driver will map the user virtual address to the MGR adapter physical address.

```
    #define GR1_GFXPG_VADDR 0x1000

    #define FIFO_OFF      0x800        /* offset of FIFO from base address of adapter .
```

The FIFO-MAP macro is used to define the FIFO virtual base address. The **im_GEsetup** macro is used to define a pointer to the FIFO virtual base address.

```
    #define FIFO_MAP \
        (volatile union gedata *   )(GRl_GFXPG_VADDR | FIFO-OFF))

    #define    im_GEsetup volatile union gedata *GE = FIFO-MAP:
```

The FIFO virtual address is assigned to the pointer GE which is the user virtual address **ored** with the FIFO-OFF value. The graphics library uses the C language array syntax to **write** to the range of FIFO addresses. A C union is used to allow both floating point and Integer data to be sent to the FIFO. The GE pointer is used in the form **GE[token].i =** 0 to send command tokens to the FIFO. The GE pointer is used in the form **GE[GE_DATA].i =** integer-data or **GE[GE_DATA].f = float_data** to send data parameters to the FIFO. The GE **variable** is a pointer to a unbn of **longs** and floats. The bng and floating point data are assumed to be 32 bit quantities. The compiler will perform the normal array arithmetic to multiply the token value by 4 **(sizeof** long or float) and adds the scaled value to the GE pointer.

Example Usage:

```
    #include "mgr.h"
    #include "imsetup.h"
    #include "gecmds.h"

    im_GEsetup;
```

```
int         int_data = 1:
float       float-data = 3.0;

GEWAIT;
GE[GE_TOKEN].i = 0;     I' an example token ●  /
GE[GE_DATA].i = int_data;
GE[GE_DATA].f = float-data;
```

Some processors have a problem sending floating point data to the MGR adapter. In these cases the floating point data can be sent out by using the C languages ability to cast pointers to different types as described earlier for the **FIFO_WR** macro. The address of the floating point variable is cast to an integer pointer and this pointer is used to read the hex value which the compiler has generated for the floating point value. This hex value is then written to the FIFO as If it were an integer. For other languages this type of work around may not be allowed and some other solution will have to be developed. The following is an example of the C language work around.

```
#include "mgr.h"
#include "imsetup.h"
#include "gecmds.h"

im_GEsetup;

int         int_data = 1;
float       float-data = 3.0;

GEWAIT;
GE[GE_TOKEN].i = 0; /* an example token that has an integer and floating point parameter */
GE[GE_DATA].i = int data;
GE[GE_DATA).i =  ●  (int *)&float_data;
```

User Mode Program Using Physical Memory

The only difference between this mode of operation and the virtual memory mode is that the **GR1_GFXPG_VADDR** macro can be assigned to the GRP variable instead ***of the*** virtual address.

```
#define GR1_GFXPG_VADDR GRP
```

In this case the GRP variable would have the physical base address of the MGR adapter. This allows the user mode program to write directly to'the physical FIFO address **still** using the same GE pointer as **described** above.

Now that we know how to program the hardware components **in** the Geometry Subsystem we can move on to initializing the subsystem and the graphics context.

Initialization Programming

Before the MGR adapter can be used to draw **3D** graphics objects it must be initialized. The following steps must be performed to initialize the adapter from a power on condition:

- search all adapter slots to find the MGR adapter **ID** in POS registers 0 and 1
- program the POS registers not already programmed during system startup
  - determine the physical address of MGR adapter and program it in subaddress regs
- reset the adapter
- read the hardware configuration bits **in** the display registers
- download the microcode
  - write the three hardware configuration parameters to the FIFO
- initialize the Display Subsystem
  - **initialize** the Display Registers
  - initialize the 5 **XPC1** or **XMAP2** chips
  - initialize the 5 Color Map chips if enhanced adapter
  - initialize the RGB RAMDAC chip or the 3 RAMDAC chips
- initialize the Raster Subsystem
  - toad the **TOPSCAN** register value
  - bad the ENABRGB register value
  - clear the bitplanes
  - initialize the 2 Cursor chips
- initial&e the graphics context
- initialize the transformation matrix
- initialize the **viewport** and the screen mask
- initialize the piece list

The programming considerations for programming the POS registers and **for** resetting the adapter are described **in the Host Interface Subsystem.** The Raster Subsystem initialization steps are described in the Raster Subsystem chapter. The Display Subsystem initialization is described **in** the Display Subsystem chapter.

The following paragraphs describe the programming considerations required to perform the microcode download and to initialize a new graphics context.

Microcode Download Programming

The microcode must be downloaded by the host since it is resident in RAM on the **MGR** adapter. The host reads a file called **ge5_re2.bin** which contains the microcode code and data constants. Before doing the download the host issues a reset command to the MGR adapter which stalls the GE5 and resets the other hardware components. The host then writes the microcode code into the code RAM and the data constants into the data RAM. The following code fragment shows the procedure for reading the **ge5_re2.bin** file and then downloading the microcode and data constants.

```
#include "mc.out.h"
#Include "ge5_glob.h"
#include "mgr.h"
#include "gr1_ucode.h"

#define          DEFAULTBINFILE      "/etc/gl/ucode/ge5_re2.bin"

mcout_t head;
uc_hdr_t uc;
int       ucfd;
char      • bufp:
register  unsigned long    *lp;
register  long        count:
register  long        addr;
tong        zero = 0, one = 1;
long        tmp;

/* Load the microcode from ge5_re2.bin into the structure pointed to by the variable uc. The
following code fragments are from the file ge5_load.c  • /

path = DEFAULTBINFILE;

/* open microcode file, read header and check for the correct magic number stored in the
header. */

if ((ucfd = open(path, O_RDONLY, 0)) == -1) {
    /* . handle error condition */
}

if (read(ucfd, &head, sizeof(head)) == -1) {
    /* handle error condition */
}

if (head.f_magic != GE5_MAGIC) {
    /* handle error condition */
}

/* transfer version and magic numbers to header */

uc.magic = head.f_magic;
uc.version = head.f_version;

/* allocate a descriptor buffer, seek to it and read it from the ge5.bin file into uc */
```

```
bufp = malloc(head.f_desclen+1);

if (bufp == NULL) {
    /* handle the error condition */
}

if (lseek(ucfd, head.f_descoff, 0) == -1) {
    /* handle the error condition */
}

if (read(ucfd, bufp, head.f_desclen) == -1) {
    /* handle the error condition */
}

/* Null terminate descriptor and save the pointer in uc*/

bufp[head.f_desclen] = '\0';
uc.descriptor = bufp;

/* transfer code length to header, obtain buffer, seek to microcode and read it from file */

uc.codelen = head.f_codelen;
bufp = calloc( (head.f_codelen+3)>>2, sizeof(long) );

if (bufp == NULL) {
    /* handle the error condition */
}

if (lseek(ucfd, head.f_codeoff, 0) == -1) {
    /* handle the error condition */
}

if (read(ucfd, bufp, head.f_codelen, 0) == -1) {
    /* handle the error condition */
}

uc.code = (unsigned long * ) bufp;      /* save buffer pointer */

/* transfer data constants length to header, obtain buffer, seek to constants and read from file
* /

uc.constlen = head.f_constlen;
bufp = calloc( (head.f_constlen+3)>>2, sizeof(long) );

if (bufp == NULL) {
    /* handle the error condition */
}

if (lseek(ucfd, head.f_constoff, 0) == -1) {
    /* handle the error condition */
}

if (read(ucfd, bufp, head.f_constlen, 0) == -1) {
    /* handle the error condition */
}

uc.constants = (unsigned long *) bufp; /* save pointer */

/* This completes the code fragment for loading the contents of the ge5.bin file into the uc
structure in memory. ● /

/* Download the microcode into code RAM from uc-> code. The code fragments are from the file
grl_downid.c */

addr = 0;                  /* microcode starts at address = 0 ● /
lp = uc->code;             /* get pointer to code words */

for (count = 0: count < uc->codelen; count +=2*sizeof(long)){
    if (addr % HQM_PG_SiZE -- 0)   f address on a page boundary? */
        HQM_WR(addr);              /* yes, set HQ MAR page address */
      HQMMSB_WR(0);                f low 32 bits of code RAM */
        URAM_WR(addr, *lp++);      /* write lower 32 bits */
      HQMMSB_WR( 1);               f upper 8 bits of code RAM */
      URAM_WR(addr, ● ip++);       f write upper 8 bits */
      addr++;                      f next address */
}

f Download constants into microcode data RAM */

addr = DIVMODTBL;          f start address of constants */

HQMMSB_WR(0);              f 32 bit data RAM word */

HQM_WR(addr);              f set HQ MAR address ● /

ip = uc->constants;        f get pointer to constants */

for (count = 0 ; count < uc->constlen; count += sizeof(long)) {
    if (addr % HQM_PG_SIZE == 0)        f address on a page boundary? */
        HQM_WR(addr);                   /* yes, set HQ MAR page address */
     DRAM_WR(addr, *lp++);              f write the data word */
    addr++;                             f next address */
}

/* read address zero of code RAM to set PC = 0 ● /

HQMMSB_WR(1);              /* set for following accesses● /

HQM_WR(0);                 f set HQ MAR to zero */

URAM_RD(0, tmp);           f read URAM so PC = 0 */

f Unstaii the microcode so that it executes the initialization code which begins at address zero
* /

HQCLRSTLQ:
```

```
/* Poll the GE interrupt which the microcode will set when it's done. If the GE interrupt is not
set after a millisecond then indicate an error.  If the GE interrupt is set then the microcode is
initialized and is stalled waiting for the host to restart it.● /

_delay (1000);          /* wait for a millisecond */

if ( !(GFXINTR_TEST(GFX_INT_GE)) ) {
    printf ("GR1_DOWNLD: GE5 not responding\n");
     return (-1);       /* assumes code fragment is function */
}

HQCLRINT();                  /* clear the GE Interrupt in HQ1 */

GFXINTR_CLR(GFX_INT_GE);     /* clear GE int bit In int stat reg ● /

HQCLRSTL();          /* clear the stall to restart microcode */

/* this ends the code fragment from grl_downld.c for downloading the code and data constants */
```

The microcode is now waiting for three data parameters to be sent down the FIFO to indicate the various installed options. The display registers must be read to determine If the extra bitplanes and Zbuffer are installed. Refer to the Display Subsystem chapter for examples of how to read the display registers and the setting of the hwconfig variable. The hwconfig variable is set during the Display Subsystem initialization to indicate which options are installed.

The first parameter indicates if the extra bitplanes option is installed. If the extra bitplanes daughterboard is installed then a one is sent else a zero is sent.

The second parameter indicates if the Zbuffer option is installed. If it is installed then a one is Sent else a zero is sent.

The third parameter Indicates if 256K or 1 Meg **VRAMs** are installed. The MGR adapter uses only 1 **Meg VRAMs** so a one should be sent

The following code fragment, from **gr1.c**, shows how the appropriate parameters are sent down.

```
if (hwconfig & DREG_BITPLANES)
    FIFO_WR(GE_DATA, zero):      /* no extra bitplanes */
else
    FIFO_WR(GE_DATA, one);       /* extra bltplanes */

if (hwconfig & DREG_ZBUF)
    FIFO_WR(GE_DATA, zero):     r no Zbuffer installed */
else
    FIFO_WR(GE_DATA, one):      /* Zbuffer installed ● /

FIFO_WR(GE_DATA, one);             /* MGR uses 1 MEG VRAM */

/* the microcode is now ready to accept tokens */
```

The initialization performed by the microcode after receiving the three parameters includes the following initialization steps:

- Perform hard initialization after reset operation
    - Save the three parameters in data variables in the GE5 data RAM
    - Load the hardware configuration into the appropriate RE2 registers
- Perform the soft initialization which is performed by the GE_INIT token.
  See the next paragraph for the GE_INIT initialization steps

Refer to the registers section of the Raster Subsystem chapter for a description Of the RE2 register Settings after the hard initialization has completed. The microcode is all set to receive tokens SO now we should initialize the graphics context and get on with drawing some 3D graphics objects.

New Context Initialization Programming

The graphics context is initialized by sending the **GE_INIT** token to the microcode. This should only be done once for each new graphics process as it does it's Initialization steps. The following initialization steps are performed by the microcode when itreceives the **GE_INIT** token:

Window Manager Variables

SIMPLE	Set screen clear flag to indicate fast clear mode of 1 to 4 piece win&w
FLATMODE	Set flat mode **= 2,** for fast block write fill mode without **WID** checking
RE2DX	Set RE2 **DX** value to 0 to match **flatmode** = 2
NEWORG	Set new window origin flag to false
XORG	Set window x origin **to 0**
YORG	Set window y origin to 0
ENABLWID	Set for no line **WID** checking

Disable **WID** checking
Set Screen Mask for full screen rectangle

Other Graphics Context Variables

CURWID	Set Current Window ID to 0
HQMSAV	Set saved **HQMAR** register value io 0

Coordinate Transformation Support

Initialize the Normal matrix stack linked list pointers
Initialize the MVP matrix stack linked llst pointers
Initialize the top matrix on the MVP matrix stack to the identity matrix

Point Drawing

Turn off antialiased point drawing mode

Line Drawing

Initialize the current graphics position to 0
Turn off antialiased line drawing mode
Set antialiase weighting to 0 **(ASELECT = 0)**
Enable the use of stipple patterns
Set the stipple pattern to **0xFFFF**
Set the stipple bit repeat count to 1
Turn off depthcued line drawing mode
Set the line width for 1 pixel
Turn off wide line drawing mode (line width of 1 pixel is default)

Closed Polyline Drawing

Turn off closed polyline drawing mode

Filled Polygon Drawing

Turn off concave polygon mode (convexpolygons are the default)

Turn off backfacing polygon removal mode
Turn off pattern masking during filled polygons
Enable the old style outlined polygon drawing mode
Disable subpixel starting vertices mode
Set shade model for flat shading of filled polygons

Triangle Mesh Drawing

Turn off Triangle Mesh drawing mode

NURBS Drawing

Turn off NURBS drawing mode

Character Drawing

Make the current character position invalid
Set pattern alignment to drawing relative mode for characters

Pixel Reads/Writes

Set pixel buffer flag to indicate pixel buffer is empty
Set RWMODE to 0 (Frame Buffer port destination of pixel write)
Set-Read Buffer to 0 (Front Buffer for Frame Buffer pixel reads)
Set the zoom x factor to 1
Set the *zoom y* factor to 1
Turn off pixel packing mode

Lighting Support

Initialize the MVP matrix free list pointer
Initialize the Normal matrix free list pointer
Initialize the top matrix on the Normal matrix stack to the identity matrix
Set the matrix mode to SINGLE for a combined MVP matrix
Set the abnormal flag off to indicate no recalculation of the normal matrix is required
Turn off Lighting mode

Bounding Box Support

Turn off the bounding box mode

Feedback Support

Turn off Feedback mode

Pick Support

Turn off Picking mode

Pixel Format and Attributes

Set Pixel Type to 2 (12 bit Color Index)
Set Pixel write mask to 0xFFF (front buffer)
Set Aux write mask to 0

Set WID data value to 0
Set PUP data value to 0
Set UAUX data value to 0
Set Raster Operation to 3 (Src Copy)
Turn off dithering mode

Z Buffer Support

Disable **Z** Buffer checking
Set Depth Function to 7 for always pass (Disable **Z Checking)**
Set for **Z** Buffer compare with new **Z** value

Miscellaneous

Clear Finish flag 0
Clear Finish flag 1

Refer to the registers section of Raster Subsystem chapter for a description of the RE2 register settings after the **GE_INIT** initialization is completed. The graphics context is now initialized but the transformation matrix contains only an identity matrix so we need to load a projection matrix before we can draw any 3D graphics objects.

3D Graphics Programming

This section discusses the programming considerations for drawing 3D graphics objects. The following topics are discussed in this section:

- Coordinate Transformation Programming
- Viewport Programming
- Pixel Format and Attribute Programming
- Z Buffer Programming
- Pixel DMA Programming

The following paragraphs describe the programming considerations for drawing 3D graphics objects.

Coordinate Transformation Programming

As described in the basic operations section the GE5 microcode provides support for a coordinate transformation matrix stack. When lighting mode is not enabled the current top of the stack matrix contains the concatenated modeling, viewing and projection matrix. This matrix is used to transform the world coordinate data into normalized coordinates. The **viewport** specification is then used to transform the normalized coordinates into screen coordinates. The transformation matrix is initialized to the identity matrix by the **GE_INIT** token so the host software must toad a projection matrix which will map the desired range of **world** coordinates to a normalized coordinates with a range of -1 to **+1** in the x, y and **z** directions.

The following coordinate transformation matrices and **viewport** transformations are described in the following sections:

- Projection Matrix Transformations
- Viewing Matrix Transformations
- Modeling Matrix Transformations
- **Viewport** Transformations

Projection Matrix Programming

While many different types of projections could be used the SGI Graphics Library supports orthographic and perspective projections. The Graphics Library specifies the world coordinates in a right handed coordinate system and the screen coordinates In a left handed coordinate system. The right handed coordinate system has the positive x to the right, the positive y up and the positive z direction toward the viewer. The left handed coordinate system reverses the positive z direction to be away form the viewer. The right handed to left handed coordinate system transformation is accomplished simply by multiplying the z scale factor portion of the projection matrix by -1.

The 2D and 3D orthographic projection matrices are shown in Figure 4.13. The left and right values represent the minimum x and maximum x values that the world coordinates will have with the positive x direction to the right. The bottom and top values represent the minimum and maximum y world coordinate values with the positive y direction up. The far and near values represent the minimum and maximum z world coordinate values with the near value being closer to the viewer.

2D Orthographic Projection Matrix

$$\begin{vmatrix} \frac{2.0}{right - left} & 0 & 0 & 0 \\ 0 & \frac{2.0}{top - bottom} & 0 & 0 \\ 0 & 0 & -1.0 & 0 \\ \frac{-(right + left)}{right - left} & \frac{-(top + bottom)}{top - bottom} & 0 & 1 \end{vmatrix}$$

3D Orthographic Projection Matrix

$$\begin{vmatrix} \frac{2.0}{right - left} & 0 & 0 & 0 \\ 0 & \frac{2.0}{top - bottom} & 0 & 0 \\ 0 & 0 & \frac{-2.0}{far - near} & 0 \\ \frac{-(right + left)}{right - left} & \frac{-(top + bottom)}{top - bottom} & \frac{-(far + near)}{far - near} & 1 \end{vmatrix}$$

Figure 4.13 2D and 3D Orthographic Projection Matrices

The perspective projection transformation matrices are shown in Figure 4.14. The perspective projections account for the far shortening of distant objects and gives a more realistic projection of near and far objects. The perspective projection is used to project all points in a world coordinate space to a single view point. This forms an infinite four sided pyramid with the apex at the viewers eye. The near and far clipping planes modify the pyramid into a rectangular viewing frustum. The apex is at the origin and the line of sight is down the negative z axis. The projection plane is perpendicular to the line of sight.

FOV Perspective Projection **Matrix**

$$\begin{vmatrix} \dfrac{\cot\left[\frac{fov}{2.0}\right]}{aspect} & 0 & 0 & 0 \\ 0 & \cot\left[\frac{fov}{2.0}\right] & 0 & 0 \\ 0 & 0 & \dfrac{-(far + near)}{far - near} & -1 \\ 0 & 0 & \dfrac{-2.0 \times far \times near}{far - near} & 0 \end{vmatrix}$$

Window Perspective Projection Matrix

$$\begin{vmatrix} \dfrac{2.0 \times near}{right - left} & 0 & 0 & 0 \\ 0 & \dfrac{2.0 \times near}{top - bottom} & 0 & 0 \\ \dfrac{right + left}{right - left} & \dfrac{top + bottom}{top - bottom} & \dfrac{-(far + near)}{far - near} & -1 \\ 0 & 0 & \dfrac{-2.0 \times far \times near}{far - near} & 0 \end{vmatrix}$$

Figure 4.14 Perspective Projection Matrices

The matrix labeled FOV Perspective lets the host software specify a field of view in the y direction and an aspect ratio of the x size to the y size. The near and far z clipping planes are also specified. The matrix labeled Window **Perspective** specifies the near and far clipping plane distances. It also specifies the left, right, bottom and top values of the near clipping plane rectangle of the viewing frustum. The window perspective matrix should not be confused with the window manager screen windows since they refer to different things.

The GE_LOADMATRIX is used to bad the projection matrix onto the matrix at the top of the matrix stack. **The** Initial projection matrix that the Graphics **Library** loads onto the MVP matrix is the projection matrix for a 2D orthographic projection which specifies the world coordinates to be the same as the screen coordinates. The **GE_LOADMATRIX** token definition shows an example of this projection matrix being **loaded.**

-Viewing Matrix Programming

The projection matrix loaded into the MVP matrix on the top of the matrix stack has a viewing direction which is looking down the z axis and the projection plane is perpendicular to the z axis. The host software can change the viewing position and direction by concatenating the appropriate rotation and translation matrices to the projection matrix already loaded into the transformation matrix. The host software can concatenate a translation along the z axis and a rotation along the x, y and z axis to change the viewing position and direct&n. If the angles of rotation and the translation distance are specified for a right handed coordinate system they all need to be multiplied by a -1 since the transformation matrix converts the right handed world coordinate system to a left handed screen coordinate system.

Example Code :

```
#include "mgr.h"
#include "imsetup.h"
#include "gecmds.h"

im_GEsetup;

float      tx = 0, ty = 0, tz = -5.0;
float      angle = -50;
float      sinz = sin(angle), cosz = cos(angle);
float      m[4][4];
int        row, col;

for (row = 0 ; row c 4 ; row++)
    for (col = 0 ;col< 4 ;)
            m[row][col++] = 0 ;

/* Multiply current matrix by translation matrix */

m[0][0] = 1.0;
m[1][1] = 1.0;
m[2][2] = 1.0;
m[3][0] = tx;
m[3][1] = ty;
m[3][2] = tz;
m[3][3] = 1.0;

GEWAIT;
GE[GE_MULTMATRIX].i = 0;

for (row = 0 ; row < 4 ; row++)
    for (col = 0 ; col < 4 ;)
            GE[GE_DATA].f = m[row][col++];

/* Multiply current matrix by rotation around z axis matrix */

m[0][0] = cosz;
m[0][1] = sinz;
m[1][0] = -sinz;
m[1][1] = cosz;
```

```
m[2][2] = 1.0;
m[3][3] = 1.0;

GEWAIT;
GE[GE_MULTMATRIX].i = 0;

for (row = 0 ; row < 4 ; row++)
    for (col = 0 ; col < 4 ;)
            GE[GE_DATA].f = m[row][col++];
```

Modeling Matrix Programming

The translation and rotation matrices concatenated to the projection matrix for the viewing position and direction transformations affect all of objects drawn on the screen. The modeling transformations allow the host software to apply scaling, rotation and translation transformation to the individual objects It draws.

The translation matrix is shown in Figure 4.15 and is concatenated to the current matrix on the top of the matrix stack using the **GE_MULTMATRIX** token. An example of concatenating a translation matrix to the current matrix is shown in the example code later in this section.

$$\begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ Tx & Ty & Tz & 1 \end{vmatrix}$$

Figure 4.15 Translation Matrix

The scaling matrix Is shown in Figure 4.16 and is concatenated to the current matrix on the top of the matrix stack **using** the **GE_MULTMATRIX** token. An example of concatenating **a** scaling matrix ***to*** the current matrix Is shown In the example code later in this section.

$$\begin{vmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Figure 4.16 Scaling Matrix

The rotation matrices are shown in Figure 4.16 and are concatenated to the current matrix ***on*** the top of the matrix stack using the **GE_MULTMATRIX** token. An example of concatenating a rotation matrix to the current matrix is shown in the example code later in this section.

Rotate around X axis

$$\left|\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) & 0 \\ 0 & -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{matrix}\right|$$

Rotate around Y axis

$$\left|\begin{matrix} \cos(\theta) & 0 & -\sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{matrix}\right|$$

Rotate around Z axis

$$\left|\begin{matrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(+) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}\right|$$

Figure 4.17 Rotation Matrices

This is accomplished using the **GE_MULTMATRIX** token to concatenate the various modeling transformations to the current viewing and projection transformation matrix. The GE_PUSHMATRIX and GE_POPMATRIX tokens can be used to control which modeling transformations **are applied** to the various **objects** which are drawn. The GE_PUSHMATRIX token can be used to save a particular MVP matrix and the **GE_POPMATRIX** token can be used to restore the saved **MVP** matrix. The transformation matrix at the top of the matrix stack is always used to perform the world coordinate to normalized coordinate transformation. The following example code shows how the scaling, rotation and translation matrices are concatenated to the current matrix on the top of the matrix stack.

Example Code :

```
#include "mgr.h"
#include "imsetup.h"
#include "gecmds.h"

im_GEsetup;

float       tx = 10.0, ty = 20.0, tz = 5.0;
float       sx = 2.3, sy = 4.0, sz = 1.1;
float       angle = 30;
float       sinx = sin(angle), cosx = cos(angle);
float       m[4][4];
int         row, col;

for (row = 0 ; row < 4 ;row++)
    for (cot = 0 ;col< 4 ;)
            m[row][col++] =  0 ;

/* Multiply current matrix by translation matrix */

m[0][0] =  1.0;
m[1][1]  =  1.0;
m[2][2]  =  1.0;
m[3][0] =  tx;
m[3][1]  =  ty;
m[3][2]  =  tz;
m[3][3] =  1.0;

GEWAIT;
GE[GE_MULTMATRIX].i = 0;

for (row = 0; row < 4 ;  row++)
    for (col = 0 ; col < 4 ;)
            GE[GE_DATA].f  =  m[row][col++];

for (row = 0 ; row < 4 ;row++)
    for (col = 0 ; ccl < 4 ;)
            m[row][col++] =  0:

/* Multiply current matrix by scaling matrix */

m[O][O] = sx:
m[1][1]  =  sy;
m[2][2]  =  sz;
m[3][3] =  1.0;

GEWAIT;
GE[GE_MULTMATRIX].i = 0;
```

```
for (row = 0 ; row < 4 ; row++)
    for (col = 0 ; col < 4 ;)
        GE[GE_DATA].f = m[row][col++];

/* Multiply current matrix by rotation around x axis matrix */

m[0][0] = 1.0;
m[1][1] = cosx;
m[1][2] = sinx;
m[2][1] = -sinx;
m[2][2] = cosx;
m[3][3] = 1.0;

GEWAIT;
GE[GE_MULTMATRIX].i = 0;

for (row = 0 ; row < 4 ; row++)
    for (col = 0 ; col < 4 ;)
        GE[GE_DATA].f = m[row][col++];

/* the rotation matrix for the y and z axis are similar to the x axis shown above */
```

Viewport Programming

The host software must specify a **viewport** which defines the normalized coordinate to screen coordinate transformation. The **viewport** specification includes the x, y and **z** scale factors which are used to scale the normalized device coordinates to the screen coordinates. The **viewport** specification also specifies the non-window relative translation values which are added to the window origin x and y values. The window relative translation values are added to the scaled screen coordinate values to get the window relative screen coordinates. The window relative screen coordinates are then used to perform the various drawing operations.

The window relative left screen coordinate of the **viewport** cannot be less than **-XMAXSCREEN.** The window relative right screen coordinate of the **viewport** cannot be greater than **2*XMAXSCREEN.** The window relative bottom screen coordinate cannot be less than **-YMAXSCREEN.** The window relative top screen coordinate cannot be greater than **2*YMAXSCREEN.** The right - left size of the **viewport** cannot be greater than 2048 and the top - bottom size of the **viewport** cannot be greater than 2048.

The GE_LOADVIEWP token is used to load the **viewport** specification. The definition of the GE_LOADVIEWP token shows an example code fragment for loading the **viewport** specification. After the **viewport** is loaded the host software should load the hardware screen mask. Programming the hardware screen mask is covered in the Window and Context Management Programming section later in this chapter.

Pixel Format and Attribute Programming

The MGR adapter has a base and enhanced hardware configuration. The base adapter supports 8 Frame Buffer bitplanes; 2 PUP bitplanes, 2 WID bitplanes and optionally 24 **Z** Buffer bitplanes. The enhanced adapter supports 24 Frame Buffer bitplanes, 2 PUP bitplanes, 2 UAUX bitplanes, 4 WID bitplanes and optionally 24 **Z** Buffer bitplanes. The GE_DRAWMODE token controls the access to the Frame Buffer, PUP, UAUX and WID bitplanes. The GE_ZBUFFER and the GE_ZFUNCTION token control the access to the Z Buffer bitplanes. The normal mode of operation is that the GE_DRAWMODE token is used to select one of the drawing modes and the pixel and aux write masks are set to allow only the selected bitplanes to be written into. The Z buffer is normally enabled to perform hidden line removal but this is independent of the drawing modes for the other bitplanes.

The following paragraphs discuss the programming considerations for the Frame Buffer, **PUP,** UAUX and WID bitplanes and the section on Z Buffer Programming covers the Z Buffer **bitplane** programming considerations.

Frame Buffer Bitplane Programming

The MGR adapter supports two types of pixels which can be written into the Frame Buffer bitplanes. The two types of pixels are the RGB pixels and the Color Index pixels and the GE_PIXTYPE token is **used** to select the pixel type. The GE_RGBCOLOR token **is** used to specify the red, green and blue **color** components for the RGB pixel types. The GE-COLOR and GE_COLORF tokens are used to specify the color index value for the Color Index pixels. The adapter also supports two buffers into which the Frame Buffer pixels can be written. The **GE_PIXWRITEMASK** is used to control which buffer is written into. The double buffered mode is provided for smooth animation effects in which one buffer is displayed while the other buffer is being displayed. The mode registers in the XPC or XMAP chips in the Display Subsystem must be set appropriately for the pixel type and the buffer select bit in the mode register determines which of the two buffers are displayed.

The enhanced adapter configuration supports 24 bit single buffer pixels and 12 bit RGB double buffered pixels. The enhanced adapter also supports 12 bit Color Index double buffered pixels and 4 bit Color index double buffered pixels. The 4 bit Color Index pixels are generally not used on the enhanced adapter. The base adapter supports 8 bit RGB single buffer pixels which are a special representation of the 24 bit RGB pixels. Refer to the Raster Subsystem chapter for details on the 8 bit RGB pixel formats. The base adapter also supports 8 bit Color Index single buffer pixels and 4 bit Color index double buffered pixels. The following code examples show how to write pixels into the Frame Buffer bitplanes. These examples assume an enhanced adapter configuration is being used.

Example Code :

```
#include "gecmds.h"
#include "imsetup.h"

im_GEsetup;

int         red, green, blue;
int         color;
Int         z_mask = 0x11;       /* enable fast z clear mode and Z buffer updates */

/* set draw mode for Frame Buffer */

GEWAIT;
GE[GE_DRAWMODE].i = 0;
GE[GE_DATA].i = 1;
GE[GE_DATA].i = 1;
GE[GE_DATA].i = 1;

/* set the aux write mask so we don't affect the PUP, UAUX and WID bitplanes */

GE[GE_AUXWRITEMASK].i = 0;
GE[GE_DATA].i = 0 | (z-mask << 4);

/* Clearing the Frame Buffer example ● /

GE[GE_PIXTYPE].i = 0;
GE[GE_DATA].i = 0;       /* set pixel type for 24 bii RGB pixels */

/* set RGB color for a black color */

red = 0;
green = 0;
blue = 0;

GE[GE_RGBCOLOR].i = 0;
GE[GE_DATA].i = red:
GE[GE_DATA].i = green:
GE[GE_DATA].i = blue;

/* set the pixel write mask so ail 24 bits are cleared */

GE[GE_PIXWRITEMASK].i = 0;
```

```
GE[GE_DATA].i = OxFFFFFF;

/* Clear the Frame Buffer bitplanes ● /

GE[GE_SCREENCLEAR].i = 0;

/* 24 bit RGB pixel example ● /

GE[GE_PIXTYPE].i = 0;
GE[GE_DATA].i = 0;        /* set pixel type for 24 bit RGB pixels ● I

/* set RGB color for a full red color ● /

red = 0xFF;
green = 0;
blue = 0;

GE[GE_RGBCOLOR].i = 0;
GE[GE_DATA].i = red;
GE[GE_DATA].i = green;
GE[GE_DATA].i = blue;

/* set the pixel, write mask so all 24 bits are written */

GE[GE_PIXWRITEMASK].i = 0;
GE[GE_DATA].i = OxFFFFFF;

/* Now we can do any type of 3D drawing we would like and the pixels will be written into the
Frame Buffer bitplanes and theZ values will be updated● /

GE[GE_MOVE3I].i = 0;
GE[GE_DATA].i = 100;
GE[GE_DATA].i = 200;
GE[GE_DATA].i = 10;       /* set the current graphic8 position to 100, 200, 10 */

GE[GE_DRAW3I].i = 0;
GE[GE_DATA].i = 600;
GE[GE_DATA].i = 400;
GE[GE_DATA].i = 10;       /* draw a line from current graphics position to 600, 400, 10 */

/* 12 bit RGB pixel example ● /

GE[GE_PIXTYPE].i = 0;
GE[GE_DATA].i = 1;        /* set pixel type for 12 bit RGB pixels */

/* set RGB color for a white color ● I

red = 0xFF;
green = 0xFF;
blue = 0xFF;

GE[GE_RGBCOLOR].i = 0;
GE[GE_DATA].i = red;
GE[GE_DATA].i = green;
```

```
GE[GE_DATA].i = blue;

/* set the pixel write mask so the back buffer is written */

GE[GE_PIXWRITEMASK].i = 0;
GE[GE_DATA].i = 0xF0F0F0;

/* Draw a closed polyline to draw an unfilled rectangle */

GE[GE_CLOSEDLINE].i = 0;        /* set closed polyline mode */

GE[GE_MOVE2I].i = 0;
GE[GE_DATA].i = 100;
GE[GE_DATA].i = 200;            /* set the current graphics position to 100, 200 */

GE[GE_DRAW2I].i = 0;
GE[GE_DATA].i = 100;
GE[GE_DATA].i = 400;            /* draw a line from current graphics position to 100, 400 */

GE[GE_DRAW2I].i = 0;
GE[GE_DATA].i = 400;
GE[GE_DATA].i = 400;            /* draw a line from current graphics position to 400, 400 */

GE[GE_DRAW2I].i = 0;
GE[GE_DATA].i = 400;
GE[GE_DATA].i = 200;            /* draw a line from current graphics position to 400, 200 */

GE[GE_ENDCLOSEDLINE].i = 0; /* end closed polyline mode and draw last line to 100, 200*/

/* 12 bit Color Index pixel example */

GE[GE_PIXTYPE].i = 0;
GE[GE_DATA].i = 2;        /* set pixel type for 12 bit Color Index pixels */

/* set color index to index of 4 which is programmed to be blue color */

color = 4;

GE[GE_COLOR].i = 0;
GE[GE_DATA].i = color;

/* set the pixel write mask so the front buffer Is written */

GE[GE_PIXWRITEMASK].i = 0;
GE[GE_DATA].i = 0x000FFF;

/* Draw a filled polygon */

GE[GE_POLYGON].i = 0; /* set filled polygon mode */

GE[GE_VERTEX2I].i = 0;
GE[GE_DATA].i = 100;
GE[GE_DATA].i = 200;            /* set first vertex to 100, 200 */
```

```
GE[GE_VERTEX2I].i = 0;
GE[GE_DATA].i = 200;
GE[GE_DATA].i = 400;          /* set second vertex to 100, 400 */

GE[GE_VERTEX2I].i = 0;
GE[GE_DATA].i = 400;
GE[GE_DATA].i = 400;          /* set third vertex to 400, 400 */

GE[GE_ENDPOLYGON].i = 0;      /* draw point sampled polygon */

/* 4 bit Color Index pixel example */

GE[GE_PIXTYPE].i = 0;
GE[GE_DATA].i = 3;       /* set pixel type for 4 bit Color index pixels */

/* set color index to index of 6 which is programmed to be cyan color */

color = 6;

GE[GE_COLOR].i = 0;
GE[GE_DATA].i = color;

/* set the pixel write mask so the back buffer is written */

GE[GE_PIXWRITEMASK].i = 0;
GE[GE_DATA].i = 0x0000F0;

/* Draw a point */

GE[GE_PNT2I].i = 0;
GE[GE_DATA].i = 100;
GE[GE_DATA].i = 200;          /* draw point at 100, 200 */
```

PUP Bitplane Programming

The PUP bitplanes are used by the window manager to display pop up overlay menus and window outline rectangles and whatever else it may wish to place in an overlay. The 2 bits in the PUP bitpianes are used as an index into the auxiliary color map in the XMAP2 chip on the enhanced adapter. On the base adapter the 2 PUP bitplanes are used as an index into the auxiliary color palette in the RGB RAMDAC. Because the PUP bits represent a color index the GE-COLOR or GE_COLORF token is used to specify the color index value which is written into the PUP bitplanes. Bits 0 and 1 of the AUX write mask controls the pixel writes into the PUP bitplanes. The pixel write mask should be set to 0 when writing to the PUP bitplanes so that the Frame Buffer bitplanes are not changed.

The 0 color index is a transparent overlay so it is not a valid color to program into the PUP bitplanes except when clearing the bitplanes. The normal usage of the Z Buffer is to remove hidden lines while drawing into the Frame Buffer. This means that the Z buffer checking would not normally be needed when writing to the PUP bitplanes. Since the base adapter has the PUP bitplanes as part of the Z Buffer port the Z Buffer should be set to pass all z compares and the aux write mask should be set so that the Z buffer values will be written. This makes sure that the PUP bitplanes will be written. Refer to the Raster Subsystem chapter for additional details on the Z

Buffer port and the PUP **bitplane** writes. The following example code shows how to write into the PUP bitplanes.

Example Code :

```
#Include "gecmds.h"
#include "imsetup.h"

im_GEsetup;

int         color;
int         z_mask = 0x11;      /* enable fast z clear mode and Z buffer updates */

/* set draw mode for PUP bitplanes */

GEWAIT;
GE[GE_DRAWMODE].i = 0;
GE[GE_DATA].i = -1;
GE[GE_DATA].i = -1;
GE[GE_DATA].i = 1;

/* set the pixel write mask to 0 so the pixel bitplanes are not affected */

GE[GE_PIXWRITEMASK].i = 0;
GE[GE_DATA].i = 0;

/* set the aux write mask to allow the PUP bitplanes to be written and the UAUX and WID
bitplanes to not be written. The Z mask values allows the fast z clear mode to be active and the Z
values to be written */

GE[GE_AUXWRITEMASK].i = 0;
GE[GE_DATA].i = 0x03 | (z-mask << 4):

/* Clearing the PUP bitplanes example */

/* set color index to index of 0 which is transparent overlay */

color = 0:

GE[GE_COLOR].i = 0;
GE[GE_DATA].i = color:

/* Clear the PUP bitplanes */

GE[GE_SCREENCLEAR].i = 0;

/* PUP bitplane drawing example */

/* set color index to index of 1 which is programmed as red color */

color = 1;

GE[GE_COLOR].i = 0;
GE[GE_DATA].i = color;
```

```
/* Draw a closed polyline to draw an unfilled rectangle */

GE[GE_CLOSEDLINE].i = 0;        /* set closed polyline mode */

GE[GE_MOVE2I].i = 0;
GE[GE_DATA].i = 100:
GE[GE_DATA].i = 200;            /* set the current graphics position to 100, 200 */

GE[GE_DRAW2I].i = 0;
GE[GE_DATA].i = 100;
GE[GE_DATA].i = 400;            /* draw a line from current graphics position to 100, 400 */

GE[GE_DRAW2I].i = 0:
GE[GE_DATA].i = 400;
GE[GE_DATA].i = 400;            /* draw a line from current graphics position to 400, 400 */

GE[GE_DRAW2I].i = 0;
GE[GE_DATA].i = 400;
GE[GE_DATA].i = 200;            /* draw a line from current graphics position to 400, 200 */

GE[GE_ENDCLOSEDLINE].i = 0; /* end closed polyline mode and draw last line to 100, 200*/
```

UAUX Bitplane Programmirig

The UAUX bitplanes are used by the various graphics applications to display user overlays. The 2 bits in the UAUX bitplanes are used as an index into the auxiliary color map in the **XMAP2** chip on the enhanced adapter. On the base adapter there are no UAUX bitplanes. Because the UAUX bits represent a color index the GE-COLOR or GE_COLORF token is used to specify the color index value which is written into the UAUX bitplanes. **Bits** 2 and 3 of the AUX write mask controls the pixel writes into the UAUX bitplanes. The pixel write mask should be set to 0 when writing to the **UAUX** bitplanes so that the Frame Buffer bitplanes are not changed.

The 0 color index is a transparent overlay so it is not a valid color to program into the UAUX bitplanes except when clearing the bitplanes. The normal usage of the Z Buffer is to remove hidden lines while drawing into the Frame Buffer. This means that the Z buffer checking would not normally be needed when writing to the UAUX bitplanes. The Z Buffer should be set to pass all **z** compares and the aux write mask should be set so that the Z buffer values will be written. This makes sure that the UAUX bitplanes will be written.

The 2 PUP bitplanes and the 2 UAUX biilanes can be combined to act as 4 UAUX bitplanes by using the **GE_LOADGE** token to set the **UAUX_4BIT** ftag in the **GE5** data RAM. When this flag is set to **1** the 4 bitplanes are combined to form 4 UAUX bitplanes. When the flag is set to -1 the 4 bitplanes are separated into 2 PUP and 2 UAUX bitplanes. On the base adapter the flag would be used to control whether there are 2 PUP bitplanes or 2 UAUX bitplanes. The following example**code** shows how to write into the UAUX bitplanes.

Example Code :

```
#include "gecmds.h"
#include "imsetup.h"

im_GEsetup;
```

```
int         color:
int         z_mask = 0x11;       /* enable fast z clear mode and Z buffer updates */

/* set draw mode for UAUX bitplanes */

GEWAIT;
GE[GE_DRAWMODE].i = 0;
GE[GE_DATA].i = -1;
GE[GE_DATA].i = -1:
GE[GE_DATA].i = -1;

/* set the pixel write mask to 0 so the pixel bitplanes are not affected */

GE[GE_PIXWRITEMASK].i = 0;
GE[GE_DATA].i = 0;

/* set the aux write mask to allow the UAUX bitplanes to be written and the PUP and WID
bitplanes to not be written. The Z mask values allows the fast z clear mode to be active and the Z
values to be written */

GE[GE_AUXWRITEMASK].i = 0;
GE[GE_DATA].i = 0x0C | (z_mask << 4);

/* Clearing the UAUX bitplanes example ● /

/* set color index to index of 0 which is transparent overlay */

color = 0;

GE[GE_COLOR].i = 0;
GE[GE_DATA].i = color;

/* Clear the UAUX bitplanes */

GE[GE_SCREENCLEAR].i = 0;

/* UAUX bitplane' drawing example● /

/* set color index to index of 1 which is programmed as red color */

color = 1;

GE[GE_COLOR].i = 0;
GE[GE_DATA].i = color:

/* Draw a closed polyline to draw an unfilled rectangle */

GE[GE_CLOSEDLINE].i = 0;       /* set closed polyline mode */

GE[GE_MOVE2I].i = 0;
GE[GE_DATA].i = 100;
GE[GE_DATA].i = 200;           /* set the current graphics position to 100, 200 */
```

```
.. GE[GE_DRAW2I].i = 0;
   GE[GE_DATA].i = 100;
   GE[GE_DATA].i = 400;          /* draw a line from current graphics position to 100, 400 */

   GE[GE_DRAW2I].i = 0;
   GE[GE_DATA].i = 400;
   GE[GE_DATA].i = 400;          /* draw a line from current graphics position to 400, 400 */

   GE[GE_DRAW2I].i = 0;
   GE[GE_DATA].i = 400;
   GE[GE_DATA].i = 200;          /* draw a tine from current graphics position to 400, 200 */

   GE[GE_ENDCLOSEDLINE].i = 0; /* end closed polyline mode and draw last line to 100, 200 */

   /* 4 UAUX bitplane example */

   #include "ge5_glob.h"

   GE[GE_LOADGE].i = 0;
   GE[GE_DATA].i = UAUX_4BIT;
   GE[GE_DATA].i = 1;                 /* set flag for 4 UAUX bitplanes */
```

WID Bitplane Programming

The 2 or 4 bits of WID bitplanes are used to control pixel writes when WID checking is enabled and they are the index into the mode registers in the XPC or XMAP chips in the Display Subsystem. The mode registers select which of the two buffers will be displayed and also control the pixel display formatting. The mode registers also provide the overlay and underlay enable bits for controlling the overlay displays from the PUP and UAUX bitplanes. Refer to the Display Subsystem chapter for additional details on the mode registers. Because the WID bits represent a color index the GE_COLOR or GE_COLORF token is used to specify the color index value which is written into the WID bitpianes. Bits 4 through 7 of the AUX write mask controls the pixel writes into the WID bitplanes. The pixel write mask should be set to 0 when writing to the WID bitplanes so that the Frame Buffer bitplanes are not changed.

The normal usage of the Z Buffer is to remove hidden lines while drawing into the Frame Buffer. This means that the Z buffer checking would not normally be needed when writing to the WID bitplanes. The Z Buffer should be set to pass all z compares and the aux write mask should be Set so that the Z buffer values will be written. This makes sure that the WID bitplanes will be written. The following example code shows how to write into the WID bitplanes.

Example Code :

```
#include "gecmds.h"
#include "imsetup.h"

im_GEsetup;

int      color;
int      xl = 100, yl = 200, x2 = 400, y2 = 800;
int      z_mask = 0x11;      /* enable fast z clear mode and Z buffer updates */

/* set draw mode for WID bitplanes */
```

```
GEWAIT;
GE[GE_DRAWMODE].i = 0;
GE[GE_DATA].i = -1;
GE[GE_DATA].i = 1;
GE[GE_DATA].i = 1;

/* set the pixel writ8 mask to 0 so the pixel bitplanes are not affected */

GE[GE_PIXWRITEMASK].i = 0;
GE[GE_DATA].i = 0;

/* set the aux write mask to allow the WID bitplanes to be written and the PUP and UAUX
bitplanes to not be written. The Z mask values allows the fast z clear mode to be active and the Z
values to be written */

GE[GE_AUXWRITEMASK].i = 0;
GE[GE_DATA].i = 0xF0 | (z-mask << 4);

/* Clearing the WID bftplanes example */

/* Set color index to index of 0 */

color = 0;

GE[GE_COLOR].i = 0;
GE[GE_DATA].i = color;

/* Clear the WID bitplanes */

GE[GE_SCREENCLEAR].i = 0;

/* WID bitplane drawing example */

/* set color index to index of 1 */

color = 1;

GE[GE_COLOR].i = 0;
GE[GE_DATA].i = color;

/* Draw a screen aligned rectangle to fill a window piece */

GEWAIT;
GE[GE_SBOXFI].i = 0;
GE[GE_DATA].i = xl ;
GE[GE_DATA].i = yl;
GE[GE_DATA].i = x2;
GE[GE_DATA].i = y2;
```

Z Buffer Programming

The normal usage of the Z buffer is to perform hidden line removal by comparing new z values with the current z values for each pixel as it is drawn. As world coordinates are transformed into screen coordinates an x, y and z screen coordinate value Is produced. The x and y values are used to select a pixel location and the z value is used by the Z compare hardware to condition the pixel writes. Since the screen coordinate system Is a left handed coordinate system with the positlve x axis into the screen the Z compare should be set to pass if the new z value Is less than or equal to the current z value at the pixel location specified by the x and y values. If the Z compare gasses then the new pixel is written and the z value is updated. If the new z is greater than the current z value It means that the new pixel is behind the current pixel so it is not written.

The compare hardware can also be used to compare color values so that when antlaliased lines Intersect the brightest pixels are drawn at the point of the intersection. In this case the compare logic should be set so that a compare passes if the new color Is greater than or equal to the current color.The GE_ZFUNCTION token is used to specify the relational comparison that the Z compare hardware will perform. The GE_ZSOURCE token is used to specify whether the comparison is between z values or between color values, The GE_ZBUFFER token is used to specify whether the comparison hardware is enabled or not. When the comparison is enabled the value specified by the GE_ZFUNCTION token is used as the comparison operation. When the comparison is disabled the comparison relational operation is set so that all compares pass thus allowing all pixel writes to be performed without any conditioning by the comparison hardware. The following code examples show how to program the Z buffer hardware.

Example Code :

```
#Include "gecmds.h
#include "imsetup.h"

im_GEsetup;

int         rfunction, zsource;

/* Aux mask would be set as shown in the previous four sections to enable fast z clear mode and
z updating. */

/* enable Z buffer mode */

GEWAIT;
GE[GE_ZBUFFER].i = 0;
GE[GE_DATA].i = 0;

/* set Z source to z compare and set Z function to <= • /

zsource = 1;            /* select Z buffer and new z value for comparison• /

GEWAIT;
GE[GE_ZSOURCE].i = 0;
GE[GE_DATA].i = zsource;

rfunction = 3;          rZ compare passes if <= */

GEWAIT;
```

```
GE[GE_ZFUNCTION].i = 0;
GE[GE_DATA].i = zfunction;

/* set Z source to color compare and set Z function to >= */

zsource = -1;            /* select Frame buffer and new color value for comparison */

GEWAIT;
GE[GE_ZSOURCE].i = 0;
GE[GE_DATA].i = zsource;

zfunction = 6;           /* Z compare passes if >= */

GEWAIT;
GE[GE_ZFUNCTION].i = 0;
GE[GE_DATA].i = zfunction;

/* disable Z buffer mode */

GEWAIT;
GE[GE_ZBUFFER].i = 0;
GE[GE_DATA].i = -1;
```

Pixel DMA Programming

The MGR adapter provides support for performing both read and write pixel DMA transfers. For the read transfers the DMA can handle a single line or a multiple line block. The GE_READPIXDMA token uses the SINGLE mode of operation to read a single scan line of pixels. The GE_READBLOCK token uses the multiple line block mode of operation to read pixels from multiple scan lines. For the write transfers the DMA can be performed in a multiple line block mode or in a LNBYLN mode to transfer multiple lines on a line by line basis. The GE_WRITEBLOCK token uses the block mode if it is writing a rectangle from a host buffer which is not packed and the x and y zoom factors are both 1.0. The GE_WRITEBLOCK token uses the LNBYLN mode if the host buffer is packed and/or the x or y zoom factor is greater than 1 .O. The following paragraphs describe the various Operating modes for the DMA transfers. The code fragment which follows the operating mode descriptions assumes that if flags is zero then it is doing a write DMA in block mode with no GE interrupt.

Single Mode

The single mode of operation is a read DMA mode and uses a Raster Engine to Host buffer DMA transfer to read pixels for a single scan line from the selected bitplanes. The source bitplanes to be read are specified with the GE_READSOURCE token. The starting x and y location for the pixel read are specified by the current character position. The single mode is enabled In the following code fragment by passing a flags word with the SINGLE bit set. The READ bit should also be set to indicate that a single read DMA transfer is to be performed.

To read the pixels in single mode the host must perform the following steps:

- clear finish flag 1
- send the GE_READPIXDMA token and the following data parameters down the FIFO
    - the x direction pixel length
- setup the next DMA transfer
    - x length is the word count to be transfered
    - buffer pointer address is the start address for the transfer
    - transfer direction is graphics adapter to host
- during the first time through the loop
    - poll finish flag 1 until it is set by the microcode Inditing that either the DMA can be started or that an error has occurred and the DMA has been canceled
    - read the DMA_FLAG word in the microcode data RAM to determine if an error has occurred. If an error has occurred abort the token execution
- start the read DMA transfer
- poll the EDDY DMA status register or process the DMA complete interrupt to determine when the DMA transfer has completed or if an error occurred

Block Mode

The block mode of operation uses Host buffer to Raster Engine DMA transfers to write the rectangle into the selected bitplanes. It also uses the Raster Engine to Host buffer DMA transfer to read the rectangles from the selected bitplanes. The microcode allows the host to read or write the rectangle as one large block or as multiple smaller blocks. For the multiple blocks method the rectangle is divided along the y axis. The number of rows in each block can be different but the number of pixels in the x direction must be the same in each block. Each block can also be in different host buffers since the Eddy DMA registers would be set up differently for each block. As the DMA transfers are performed the rectangle is read or written beginning at the lower left comer specified by the x, y parameters and progresses to the upper right corner. For a write the different blocks are concatenated into one contiguous rectangle. As each block is transfered the host software needs to determine when the current block transfer has completed so the next transfer can begin. The host can tell the microcode to generate a GE interrupt after it has completed the current block or it can poll the Eddy chip's DMA status registers or use a DMA complete interrupt to determine when the last line of the block has finished being DMAed. The DMA complete or the GE interrupt method may be more efficient in multitasking environments. The interrupt or poll method is specified to the microcode with the dma_flag parameter. The block mode is enabled in the following code fragment by passing a flags word with the SINGLE bit and the LNBYLN bit not set. The INTR bit set indicates that the GE interrupt should be generated at the end of each block. If the READ bit is set then a block read transfer is done else a block write transfer is done.

To read or write the rectangle in block mode the host must perform the following steps:

- clear finish flag 1
- send the GE_WRITEBLOCK or the GE_READBLOCK token and the following data parameters down the FIFO
  - pixel packing mode parameter set to 0
  - lower left X screen location Of the rectangle
  - the x direction pixel length
  - tower left y screen location of the rectangle
  - dma_flag parameters down the FIFO
- loop white any rows remain to be sent performing the following steps
  - if the ylen is > 0 send the current blocks y length down the FIFO and adjust the row counter for block size and if y length <= 0 then send the DMA_DONE flag down the FIFO
  - setup the next DMA transfer
    - x length is the word count to be transfered
    - current buffer pointer address is the start address of the transfer
    - if a read DMA the direction is graphics adapter to host and for a writ8 DMA transfer the direction is host to graphics adapter

- during the first time through the loop
    - poll finish flag 1 until it is set by the microcode indicating that either the first DMA can be started or that an error has occurred and the DMA has been canceled
    - read the **DMA_FLAG** word in the microcode data RAM to determine if an error has occurred. If an error has occurred abort the token execution
- during the remaining times through the bop
    - if the GE interrupt mode of operation is being used to check for the block transfer complete then wait for a semaphore to be set
    - if the GE interrupt is not being used, then poll the Eddy DMA status register or wait for a **DMA** complete interrupt for the appropriate channel to see if the previous **DMA** had completed
- if the row counter is > 0 start the next DMA
- adjust the current buffer pointer for the next line
- decrement the ylen parameter by the block size.

- Continue with the next pass through the loop

Line By Line Mode

The line by line mode of operation uses Host buffer to **GE5** Data RAM DMA transfers and then GE5 Data RAM to Raster Engine DMA transfers to write the rectangle into the selected bitplanes. The host must send the rectangle on a line by line basis since the GE5 Data RAM has pixel buffer which can **only** hold a maximum of 1280 long words. The line by line mode allows the host buffer to be in a packed or unpacked format. The mode also albws no pixel zooming to be in effect or a zoom in the x direction, y direction or both. The x and y zoom factors are specified with the **GE_ZOOMFACTOR** token. If the host buffer is packed then it can contain either 2 or 4 pixels per 32 **bit** word. As the DMA transfers are performed the rectangle is written beginning at the **lower** left corner specified by the x, y parameters and progresses to the upper right corner. The different lines are concatenated into one contiguous rectangle. If pixel packing or pixel zooming in the x direction are being used then the raster operation specified with the **GE_RASTEROP** token must be set to 3 **(SRC_COPY).** If packing and x zooming are not being used then the **raster** operation can be any of the 16 legal values.

As each line is transfered the host software polls the finish 1 flag to determine when the microcode has finished with the current line so that the **GE5** Data RAM is free to receive the next fine. For each line transfered to the GE5 data RAM the line is **DMAed** to the RE2 for writing into the selected bitplanes. Each line will be transferred to the RE2 the number of times specified in the y zoom factor. The RE2 hardware will handle the x zoom of each pixel it receives.

The host buffer **contains** the pixel data in a contiguous array and the buffer can be located on a byte boundary. The Eddy chip requires the DMA transfers to occur on a long word boundary. Also the number of pixels on each line can be a value that does not end on a **long** word boundary even if the buffer did start on a long word boundary. This requires the host software to calculate for each line the necessary starting byte offset and the dma word count taking into account the cases where a line does not start on a long word boundary and does not end on a long word boundary. The host must also manage it's host buffer pointer to take into account the times when a long word contains pixels from two lines and therefore that long word must be sent as the last word of the current line and again as the first word of the next line.

Since there are four bytes per long word the byte offsets and dma counts will repeat after every four lines. This means that the host can calculate the necessary parameters for the first four lines and then use them repeatedly for each group of four lines to **be** sent. In the following discussion and **code** example the first word byte offsets for the four lines are called **haddr[0], haddr[1], haddr[2]** and **haddr[3].** The number of tong words needed by a line if started on a tong word boundary is called iinesizlong. The actual dma word counts for **the** four lines taking into account the starting and ending byte offsets from long word boundaries are called **dma[0], dma[1], dma[2]** and **dma[3].** A flag word must also be calculated which allows the microcode to calculate the actual dma word counts as well. The margin flag parameter specifies the number of bytes between the ending byte offset of a line and the starting byte of the next tong word. If the line starts and ends on tong word boundaries or the ending byte offset is just 1 past the **start** of a tong word then the dma word count will **be** the same for each line. In this case the margin flag is set to -1 to allow the microcode to skip the dma word count calculations. The example code fragment shown later shows how these variables are calculated. The line by line mode is enabled in the following code fragment by passing a flags word with the **LNBYLN** bit set. The READ, SINGLE and **INTR** bits should not be set for a line by line mode transfer since it is a write line by line transfer only.

To write the rectangle in **line by** line mode the host must perform the **following** steps:

- calculate the linesizlong variable which tells the microcode how long each line is in **long** words if it started on a long word boundary
- calculate the margin flag variable which tells the microcode how to calculate the actual dma count for each line if it is does not start on a long word boundary
- calculate **haddr[0], haddr[1], haddr[2]** and **haddr[3]** variables which indicate the byte offset for the first pixel in the first word of the first four lines
- calculate **dma[0], dma[1], dma[2]** and **dma[3]** variables which are used as the actual dma **word count for the (row 8 3)**
- clear finish flag 1
- send the GE_WRITEBLOCK token and the following data parameters down the FIFO
    - pixel packing mode parameter set to 0, 1 or 3 (pixels/long - 1)
    - lower left x screen location of the rectangle
    - the x direction pixel length
    - lower **left y screen location of the rectangle**
    - **the y** length parameter
    - **if** the pixel packing mode is not zero then **send the following pixel packing parameters**
        - **the iinesizlong parameter (indicates the number of words that would have to be transfered** if the line started on **a long word boundary)**

- the margin flag (used to calculate the actual number of words for the current line if does not start on a long word boundary)
- the **haddr[0]** parameter
- the haddr[l] parameter
- the **haddr[2]** parameter
- the **haddr[3]** parameter

- **loop** while any rows remain to be sent performing the following steps
  - **setup the** DMA transfer for **the** next line
    - dma count for (row **&** 3) is the word count to be transfered
    - current buffer pointer address is the start address of the transfer
    - the direction is host to graphics adapter
  - decrement the row counter_
  - during the first time through the loop
    - poll finish flag 1 until it is set by the microcode indicating that either the first DMA can be started or that an **error** has **occurred** and the DMA has **been canceled**
    - **read the DMA_FLAG** word in **the** microcode data RAM to determine if **an** error has occurred. **If** an error has **occurred** abort **the** token **execution**
  - during the remaining times through the loop
    - poll the finish 1 flag to see if the microcode is done with the last line previously **sent**
  - if row counter **>** 0 start the next DMA
  - adjust the current buffer pointer for, the next line
  - dear finish flag 1
- Continue with the next pass through the loop

Refer to the Host Interface chapter for information on the Eddy DMA registers which are used to setup and start the DMA transfers. The DMA **control/status** registers provide the DMA complete and **the** DMA **error** status.

For the selected destination bitplanes the appropriate **bitplane** mask register **will determine** which bits in the bitplanes will actually be written. The **GE_PIXWRITEMASK** will affect which bits in the frame buffer would be written. **The** GE_AUXWRITEMASK token would control bit writes to the PUP, UAUX, **WID** and Z **buffer** bitplanes. The pixel values written to the frame buffer PUP or UAUX bitplanes can have the logical operation **performed on each pixel before it is written. The**

GE_RASTEROP token is used to specify the desired logical operation to be performed. The write masks and the logical operation do not affect read DMA operations.

The current screen mask will clip writes to any bitplanes to the bits on or within the screen mask rectangle. If window ID checking is enabled then the bits to be written will be WID checked and only those that pass the check will be written. The exception to the checking are the Z buffer and WID bitplanes. On the base configuration adapter the PUP bitplanes are also not WID checked. On the enhanced adapter the PUP planes will be WID checked. The screen mask and WID checking do not have any effect on read DMA operations.

For write DMA transfers the host buffer is expected to have data that is appropriately formatted for the destination bltplanes. Refer to the RE2 DMA support paragraphs in the Raster Subsystem chapter for a description of the pixel formats and the pixel packing requirements for both read and write DMA transfers from and to all of the different bitplanes.

Example Code :

```
/* The code fragment below assumes that it is part of a function in the operating system kernel
or in a kernel level device driver.  Normally this function would be called by a higher level
function with the appropriate parameters to perform the necessary transfer. */

#include  "mgr.h"
#include  "ge5_glob.h"

#define   TIMEOUT_1 STOMA   5000000    /* 1st DMA timeout value */
#define   TIMEOUT_GDMA      1000000    /* not 1st DMA timeout value */
#define   DONE              -1         /* Microcode expects this value */
#define   GDMA_NOINTR       -1         /* Microcode expects this value */
#define   GDMA_INTR          1         /* Microcode expects this value */
#define   DMA_CANCEL        -1         /* Microcode may return this value */
#define   MAXXLEN            1280      /* pixels in 1 complete scan line */
#define   MAXYLEN            1024      /* the number of rows in entlre screen */
#define   ERROR             -1         /* arbitrary, system dependent value */
#define   DOLOCK             1         /* arbitrary value */
#define   FIRST-TIME         2         /* arbitrary value */
#define   LNBYLN             4         /* arbitrary value */
#define   INTR               0x40      /* arbitrary value */
#define   BLOCKSIZE          50        /* let's do 50 lines at a time */

/* The do_pixel_dma function handles both read and write DMA operations. For the DMA read
operations the function handles the both the single line case and the multiple line cases. For the
DMA write operations the function handles the multiple line case or the line by line case for
packed or zoomed rectangle writes.

do_pixel_dma (token, pixbuf, pixsize, flags, x, xlen, y, ylen)
    long    token, pixsize, flags, x, xlen, y, ylen;
    char    *pixbuf;
{

tong        upacmode = (4 >> (pixsfze >> 1)) - 1;
long        'bufptrlong;
long        blocksize, row, i;
tong        haddr[4];
long        dma[4];
long        offset, lineoffset, iinesiz, linesiziong, margin;
int         tmp = 0, dma_flag = DONE:
int         tmpl = 0;
int         do_lock;

if ((xien > MAXXLEN) || (yien > MAXYLEN))
    return (ERROR);

/* Do the necessary kernel level things to mark the current context as doing a DMA, set the DMA
bck semaphore and anything else such as determining if the host buffers have been locked
already or if this function needs to do it. If this function will do the bck then flags |= DOLOCK
should be executed here */

/* calculate the byte length of a line and the long word length of a Mock */

iinesiz = xlen . pixsize;                /* get the byte count for a line */

blocksize = xlen . BLOCKSIZE;            /* get blocksize word length for block mode ● 1

/* if we are in a LNBYLN mode then calculate the pixel offsets into each of the lines and the dma
size of the lines. */

if (flags & LNBYLN) {

    /* get byte offset of starting byte in pixbuf from long word boundary */

    offset = (long) pixbuf & 3;

    /* Using the byte count for a line, determine the byte offset of the last long word assuming
    the line starts on a long word boundary'/

    lineoffset = linesiz & 3;

    /* Calculate the starting pixel byte offset into the first word for each of the first four lines
    * /

    haddr[0] = offset;                          /*. first line is the buffer offset */
    haddr[l] = (haddr0 + lineoffset) & 3;       /* add line offset to prevbus pixel offset */
    haddr[2] = (haddrl + lineoffset) & 3;       /* add line offset to previous pixel offset */
    haddr[3] = (haddr2 + lineoffset) & 3;       /* add line offset to prevbus pixel offset */

    /* Calculate the number of bng words required for each line if it were long word aligned */

    linesiziong = (iinesiz + sireof(iong) - 1) >> 2;

    /* Calculate the number of long words to be sent for each line taking into account the
    non word alignment of the byte offsets of the first byte and the last byte of the line. ● /

    for (i = 0 ; i < 4 ; ++i)
            dma[i] = (iinesir + haddr[i] + sizeof(long) - 1) & -3:

    /* Calculate the margin flag to allow the microcode to calculate the number of words
     it will receive for each line. if the lineoffset is 1 or the buffer is long word aligned
    (offset == 0 && lineoffset == 0) then the four dma lengths are equal to the
    finesirlong calculation so set the margin flag to -1 so the microcode will use the
    linesizlong parameter for the dma lengths. If the lineoffset is > 1 or the offset is > 0 then
    the dma lengths for the four lines will vary and must be calculated by the microcode. */

    margin = (sizeof (long) - lineoffset) & 3;

    if (margin == 3 || (offset == 0 && lineoffset == 0))
        margin = -1;
}

FINISH_WR(FINISH1_OFF, 0); /* Clear finish 1 flag since it will be polled later */

if (flags 8 INTR)
    /* Clear interrupt semaphore which will be checked later */

/* Send the token. The remaining tokens to be sent depend on the mode */

FIFO_WR(token, tmp);

/* the data parameters depend on the transfer mode */

if (flags & LNBYLN) {
    FIFO_WR(GE DATA, upacmode);
    FIFO_WR(GE_DATA, x);
    FIFO_WR(GE_DATA, xlen);
    FIFO_WR(GE_DATA, y);
    FIFO_WR(GE_DATA, ylen);          /* Tell ucode the entire y length now */

    /* the remaining parameters are sent only if the host buffer is packed */

    if (upacmode) {                          /* pixel packing in effect? */
        FIFO_WR(GE_DATA, linesizlong); /* the number of long words per aligned line */
        FIFO_WR(GE_DATA, margin):   /* flag to calculate the actual unaligned line sizes● /
        FIFO_WR(GE_DATA, haddr[0]);   /* pixel offset into first word of line 0 */
        FIFO_WR(GE_DATA, haddr[1]);   /* pixel offset into first word of line 1 ● /
        FIFO_WR(GE_DATA, haddr[2]);   /* pixel offset into first word of row 2 */
        FIFO_WR(GE_DATA, haddr[3]);   /* pixel offset into first word of row 3 ● /
    }
} else if (flags & SINGLE) {
        FIFO_WR(GE_DATA, xlen);
} else {     /* block mode ● /

    FIFO_WR(GE_DATA, upacmode);              /* should be 0 for block mode ● /
    FIFO_WR(GE_DATA, x);
    FIFO_WR(GE_DATA, xlen);
    FIFO_WR(GE_DATA, y);
    if (flags & INTR)
        FIFO_WR(GE_DATA, GDMA_INTR); /* send down dma interrupt flag */
    else
        FIFO_WR(GE_DATA, GDMA_NOINTR); /* send down no dma interrupt flag */

  /* Now do the loop to read or write the blocks of data or the separate lines using multiple DMA
  transfers. This example assumes that for block mode each block is of size BLOCKSIZE. If
```

```
    BLOCKSIZE were equal to yien then only one DMA transfer would be performed to read or
    write the entire rectangle. For the single mode oniy one pass through the loop is made. */

flags |= FIRSTTIME;
 bufptrlong = (long) &pixbuf[0];        /* need long ptr to host buffer ● /
row = 0;

while (ylen > 0) {

  /* The buffers must have already been locked by the user process for the LNBYLN mode */

  if (flags & DOLOCK) {
      /* Lock the host buffer if necessary'/
       if (lock_error) {
         FIFO_WR(GE_DATA, done-flag):     /* Tell uccde we quit● /
         /* Do any necessary kernel level cleanup */
         return (ERROR):
     }
  } else if (flags & READ)
            /* flush the data cache or anything else that needs to be done for a read */

  /* Tell the microcode how big the current block is if in block mode */

  if (!(flags & (LNBYLN | SINGLE)))
         if (ylen > 0 )
              if (ylen < BLOCKSIZE)
                  FIFO_WR(GE_DATA, yien);
             else
                  FIFO_WR(GE_DATA, BLOCKSIZE);
         else
                  FIFO_WR(GE_DATA, dma_flag);     /* Tell the microcode were done */

  /* During the first time through the loop set up Eddy DMA channel 0 and for each successive
  pass use the alternate channel.  Set the start address to the current host buffer pointer
  (bufptriong).  ● /

  if (flags & READ)
         /* Set the direction bit for graphics adapter to host. */
  else
         /* Set the direction bit for host to graphics adapter. ●  I

  if (!(flags & (LNBYLN | SINGLE)) {

     /* set the DMA length to the word count of the block to be sent.

  } else if (flags & LNBYLN) {

     /* For the LNBYLN mode we break up the transfer into groups of four lines. Use the
     dma[row & 3] length for the current line. */

  } else {        /* SINGLE ● /

     /* set the DMA length to xien. */
```

```
}

/* The first time through the while loop we must wait for the microcode to set finish flag 1
Indicating that it is ready for us to start the DMA transfers. All other times through the loop
we just wait for the previously started DMA to complete. */

if (flags & FIRSTTIME) {      /* First pass through the while loop */

    /* Poll the finish 1 flag to see if the microcode is ready to have the DMA started. This
    example shows a simple spin loop with a counter. */

        for (tmp = 0 ; 1; tmp++) {
                if (tmp & 0x7)          /* Don't poll every time */
                        continue;
                if (FINISH_POLL(FINISH1_OFF));
                        break;
                if (tmp > TIMEOUT_1STDMA) {
                    /* process error, probably need to reset graphics adapter */
                    /* do any kernel level cleanup */
                    return (ERROR);
                }
        }

    /* Check to see if microcode has decided there is no DMA to do. This would only occur if
    something is wrong with the data parameters. */

        HQM_WR(DMA_FLAG);                  /* Read DMA flag in data RAM */
        DRAM_RD(DMA_FLAG, tmp);

        HQM_WR(HQMSAV);            /* Get any previously saved HQ MAR value */
        DRAM_RD(HQMSAV, tmpl);
        HQM_WR(tmp1);              /* Restore the HQ MAR to previous value */

    /* If DMA_FLAG contained DMA_CANCEL then the microcode did not like the parameters
    and canceled the token execution. Just clean up and exit. */

        if (tmp == DMA_CANCEL) { /* DMA_CANCEL defined in g85glob.h */
                /* do any kernel level clean up */
                return (ERROR);
        }

        flags &= ~FIRSTTIME;       /* clear first time through loop flag */

} else {                  /* not first time through the loop */
        if (!(flags & LNBYW)) {     /* block mode */

            if (flags & INTR) {

                /* GE interrupt signals end of previous block transfer */

                /* wait for GE interrupt service routine to set the semaphore */

            } else {      /* Eddy dma complete indicates the end of the block transfer */
```

```
                f poll Eddy for DMA complete or DMA error on the DMA started in the
                previous time through the loop */

                if (dma_error) {
                    f reset the graphics adapter and do any necessary kernel level cleanup */
                    return (ERROR);
                }
            }
        } else {    /* LNBYLN mode ●  /

              f Poll the finish 1 flag to see if the microcode is ready to have the next DMA
              started. In zoom pixel mode the ucode is not necessarily done after the last byte
              has been transferred. This example shows a simple spin loop with a counter.*/

               for (tmp = 0 ;1; tmp++) {
                 if (tmp & 0x7)         /* Don't poll every time ●  /
                         continue;
                 if (FINISH_POLL(FINISH1_OFF));
                         break:
                 if (tmp > TIMEOUT_GDMA) {
                     f process error, probably need to reset graphics adapter */
                     /* do any kernel level cleanup●  /
                     return (ERROR);
                 }
               }
        }

    FINISH_WR(FINISH1_OFF, 0);       f clear the finish 1 flag for next time thru loop */

    if (ylen > 0)
            /* Set appropriate start bit in the Eddy chip to start the next DMA transfer */

    if (!(flags 8 LNBYLN)
            bufptrlong += blocksize;
    else {
            bufptriong  += dma[row 8 3];        /* add dma word count of line to ptr */
            if ( haddr[(row & 3) + 1])          /* if next line is not word aligned */
                    --bufptrlong; f the last word of this line is 1st word of next line */
    }

    if (flags & SINGLE) { f single only goes through the bop once */

        f poll Eddy for DMA complete or DMA error on the DMA started in the previous time
        through the loop */

    }
    if (!(flags & (LNBYLN | SINGLE))) /* block mode */
            ylen -= BLOCKSIZE;
    } else  {       f LNBYLN mode or single mode */
            --ylen;
            ++row;
    }
} /*  end of while loop */
```

```
    /* do any kernel level final processing */
     return  (0);
  } /* The rectangle should now have been read or written if no errors were encountered */
```

Window and Context Management Programming

The window manager controls the following aspects of the MGR adapter:

- Window origin
- Hardware screen mask
- WID checking enable or disable
- WID **bitplane** programming
- Window piece list
- Graphics Context Switching

The window manager controls these components of the MGR adapter by writing to the saved graphics contexts in the host memory and by instructing the graphics applications to reset some of **the** parameters by issuing a REDRAW input event to the application.

The following paragraphs describe the programming considerations for these **MGR** resources.

Saved Context Programming

The window manager communicates primarily with the graphics context to manage the window origin, WID checking and the piece list. The window manager writes to the saved graphics context to **control these resources.** The **GR1_GETOCX** macro is provided to **allow** the window manager to access the saved data variables in the graphics context. This macro is defined in the include file **ge5_glob.h** and is shown here for reference only. Obviously the defines **in** the ge5glob.h file supersede these definitions shown here if any differences should **be** found at a later time.

```
#define MAXCONS          1450
#define DMABASE1         MAXCONS - 1
#define GR1_GETOCX(addr) ((addr) - (DMABASE1 + 1))
```

The following paragraphs indicate when the **GR1_GETOCX** macro is used to **access** the saved context.

Window Origin Programming

The window manager must write the window origin into the graphics context when the window origin is changed. The **GR1_GETOCX** macro **is** used to update the XORG and **the** YORG addresses in the **saved** context with the x and y window origin **values. The** window manager also sets the NEWORG flag in the context to instruct the context restore microcode to recalculate the **viewport** window **origin offsets using the XORG and the YORG values.**

The window manager also communicates with the graphics applications through a shared memory segment. The window origin values are programmed into the shared memory for use by the application when it receives a REDRAW event or when it programs the **viewport** and screen mask values. The Graphics Library uses the window origin values to do window relative offset calculations. The values are also returned to the application program when it does a getorigin system **call to** enable it to also **do window** relative calculations.

Hardware Screen Mask Programming

The RE2 chip in the Raster Subsystem contains a built-in hardware screen mask. The screen mask is specified as a rectangular region of the the screen which clips all pixel writes which have a screen location outside of the screen mask rectangle. The screen mask is set to the same size as the **viewport** rectangle and the screen mask rectangle lower left and upper right coordinates are added to the window x and y origin to make the screen mask rectangle window relative. If the screen mask is smaller than the window the host software must set the **SM_VP_CLIP** flag in the GE5 data RAM. The **GE_LOADGE** token **is** used to bad the flag value. If the screen mask is smaller than the window the flag should be set to 1 and if the screen mask is not smaller than the window then the flag should be set to -1. The GE_SCRMASK token is used to set the screen mask and it shows an example of how to program the screen mask.

WID Checking Management Programming

The window manager must keep track of whether windows are unobscured or are obscured. Unobscured windows are not covered by any other windows on the screen and are therefore a single rectangular piece. Obscured windows are partially covered by one or more windows and the obscured window consists of multiple rectangular pieces. For unobscured windows the screen mask can be used for pixel clipping so that only the pixels within the window are written. When the window is obscured and consists of two or more rectangular pieces the window manager must use the Window ID bitplanes to control the pixel writes. This is **done by** writing the window ID of the current window in the WID bitplanes for the various rectangular pieces and then enabling window ID checking. The **CURWID** variable in the context is set with the current window ID which is used to perform the WID checking.

When the window is obscured and WID checking is enabled then the **ENABLWID** checking flag is set to cause the line WID checking to be enabled. Normally the shaded span **WID** checking is enabled and the **FLATMODE** flag in the context is set to control whether shaded spans or flat spans are used to draw filled polygons. If the GE_SHADEMODEL token had set the polygon fill mode for shaded fills then the **FLATMODE** variable **is ignored** and shaded spans are drawn which are WID checked. If the shade model is set for flat shading the **FLATMODE** variable is set by the window manager based on whether WID checking is needed or not. If WID checked flat spans are required then the **FLATMODE** variable is set using the **GR1_** GETOCX macro to a value of 1 so that the shaded spans are drawn using a flat shading. If WID **checking** is not needed then the **FLATMODE** variable is set to 2 so that the **fills** are done using the flat span instructions. The FLATDX variable is set to 0x4000 if the**FLATMODE** is set to 1 and is set to 0 if the **FLATMODE** variable is set to 2. Refer to the Raster Subsystem chapter for additional details on the shaded span and flat span instructions.

WID **Bitplane** Management Programming

The window manager loads the rectangular areas which form the obscured window by writing the current Window ID into the rectangular areas. The GE_SBOXFI token can be used to program the rectangular areas of the window. The window manager turns off patterns and sets the shademodel to flat so that the window ID bitplanes are written using the flat span instruction.

Piece List Programming

The microcode provides a piece list mechanism which is used to optimize screen clears for obscured windows. The piece list can contains a count and up to 4 rectangular piece descriptions. When the window is unobscured or is obscured with more than four pieces the piece list count is set to 1 and the single rectangle specification is set to the same size as the screen mask. If the obscured **window** consists of 2 to 4 pieces then the piece count is set to the number of pieces and the rectangle locations and sizes are written into the list. The window manager can write the piece list into the saved context or the current application can write the piece list after it receives a REDRAW event. The window manager would have written the piece list description into the shared memory region shared by the window manager and the graphics applications. The graphics application can use the GE_SETPIECES token to write the piece list.

The **GE_SCREENCLEAR** and GE_CZCLEAR tokens use the piece list to clear the bitplanes. The piece list must be set or these tokens **will** not function properly. No other tokens are affected by the piece list. The window manager uses the SIMPLE flag in the context to control whether **the** two tokens use flat ***spans or*** shaded spans to clear the **bitplanes.** If **the** window is **1** to 4 pieces then the SIMPLE flag is set to 1 so that the non-WID checked flat spans are used to clear the bitplanes. if the window is more than 4 pieces then the SIMPLE flag is set to -1 so that the WID checked shaded span instruction is used to clear the bitplanes.

Context Switch Programming

As the host operating system switches among the various processes running on the system it is necessary for **it** to switch the current **context** out of the adapter and to restore a context into the adapter. The host must also remap the user virtual address to physical mapping required for the graphics application to access the adapter memory mapped addresses. The graphics context switching is very dependent on the host operating system and the host window manager software. The host operating system and window manager must manage the switching of the currently active context out of the adapter and the **switching** of a different context into **the** adapter. **A graphics** context switch can occur before the **microcode** gets ail the parameters for a token. This means that the the host software cannot send any tokens to the adapter after this context has been restored. Before a context is restored the host **software** must use the value of the saved PC in the saved context to access the restart instruction and the following instruction and adjust the **memptr** and **reptr** values appropriately. The **GE_CTX0** and the **GE_CTX1** token descriptions describe the context switching steps involved in saving the current context and restoring a previously saved context.