

6 Programmer's Reference



6.1 Overview

This section of the Technical Reference Manual provides information of particular interest to programmers of the SM3110 graphics chip.

A key component of programming for the SM3110 is the ability to access all of the registers, FIFO and data areas of the graphics hardware and memory regions. This section begins with by detailing the addressing spaces and access modes of the SM3110.

Following the address space introduction are the subsections describing the setup and use of the graphics chip itself. The topics presented are arranged from initialization by the BIOS and driver, through the 2D and 3D functions. Then the remaining capabilities of the device are described. Finally, some equates and macros are defined that make the coding easier.

6.2 Address Mapping

Programming for the SM3110 requires access to four(4) address spaces:

- (1) VGA addresses for direct I/O port access at 3CxH, 3BxH, 3DxH
- (2) VGA addresses for VGA frame buffer access at A0000H (128KB);
(larger memory for Super VGA frame buffer data is made accessible via an I/O addressed bank/offset register)
- (3) PCI configuration space (256B)
- (4) SM3110 "Local Memory" addresses, encompassing all the embedded memory and memory-mapped registers (128MB)

In addition to having their standard addresses, address spaces (1) and (3) have memory-mapped addresses within the SM3110 Local Memory range.

6.2.1 Local Memory Address Space

The term “local memory” refers to the 32MB linear address space within which the SM3110 maps all of its available embedded memory, as well as its internal registers and memory-mappings of VGA and PCI registers. In order to support all variations of little-endian/big-endian and word/double word byte-swapping architectures, the SM3110 requires a total address space of 128MB (out of the 4GB available) address space. The four(4) individual 32MB address spaces are multiply-mapped to the one 32MB local memory area required. This is shown in Figure 6-1.

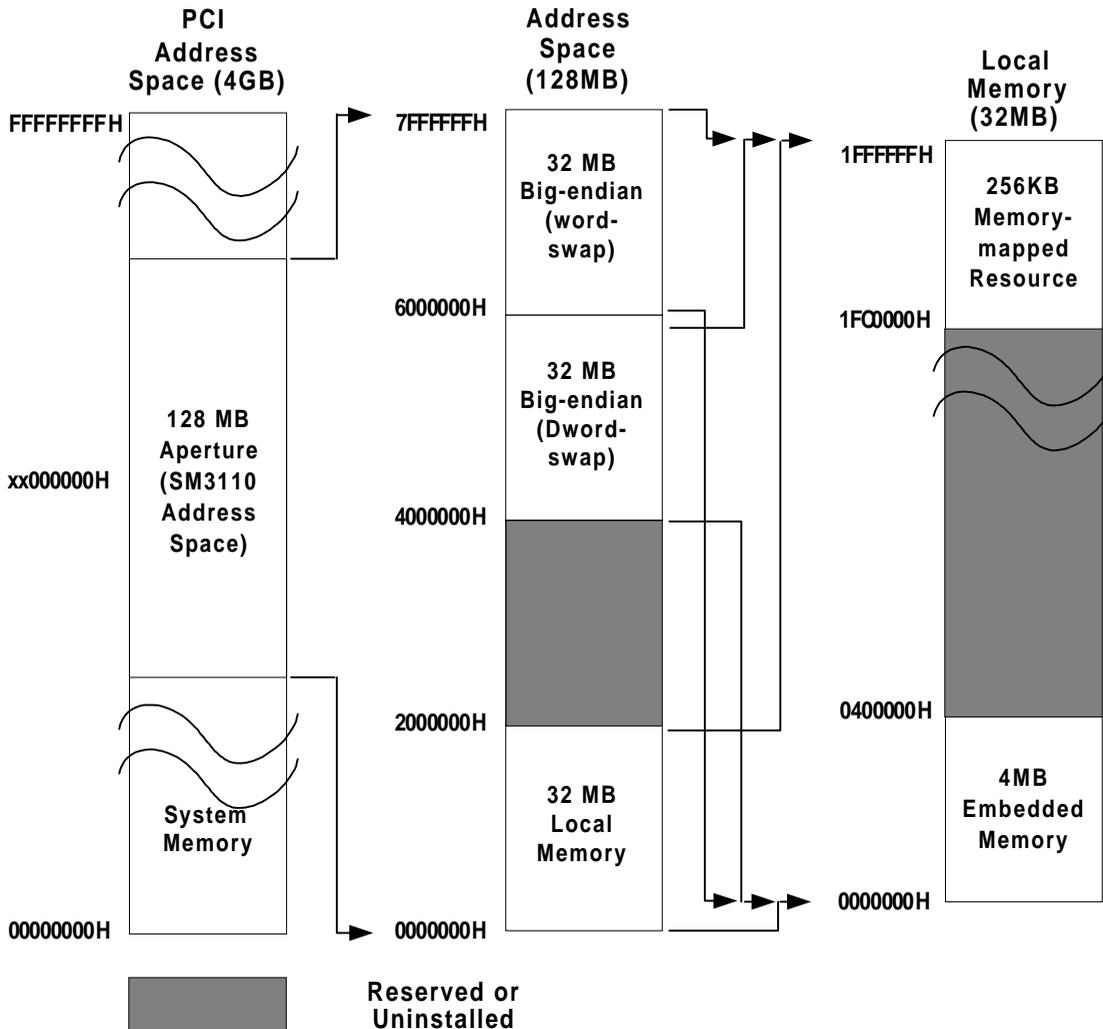


Figure 6-1. SM3110 Address Space Allocation

Range	Size	Little Endian Base	Little Endian (Reserved) Base	Big Endian (Dword Swap) Base	Big Endian (word Swap) Base
Display Memory	31.75MB	00000000H	02000000H	04000000H	06000000H
Memory-mapped Resources	0.25MB	01FC0000H	03FC0000H	05FC0000H	07FC0000H
Total	32.00MB				

Table 6-1. Local Memory Space Allocation

Also shown in Figure 6-1 is a further division of the 32MB local memory addresses into two areas:

- (1) 256KB at the top of the address range, for memory-mapping SM3110 and other system resources
- (2) 31.75MB at the bottom of the address range, for addressing the SM3110 display memory (of which of 4MB is used for embedded memory on the SM3110)

The embedded 4MB display memory is dynamically partitioned by the software to provide data buffers, frame buffers, FIFO's, etc. These divisions and areas are discussed later in this section.

6.2.2 Memory-Mapped Resources Address Space

The 256KB region (at the top of local memory) for memory-mapped resources is further divided into four areas:

- (1) Control addresses (64KB)
- (2) Peripheral I/O addresses (64KB)
- (3) Reserved (64KB)
- (4) Embedded memory configuration registers

as shown in Figure 6-2.

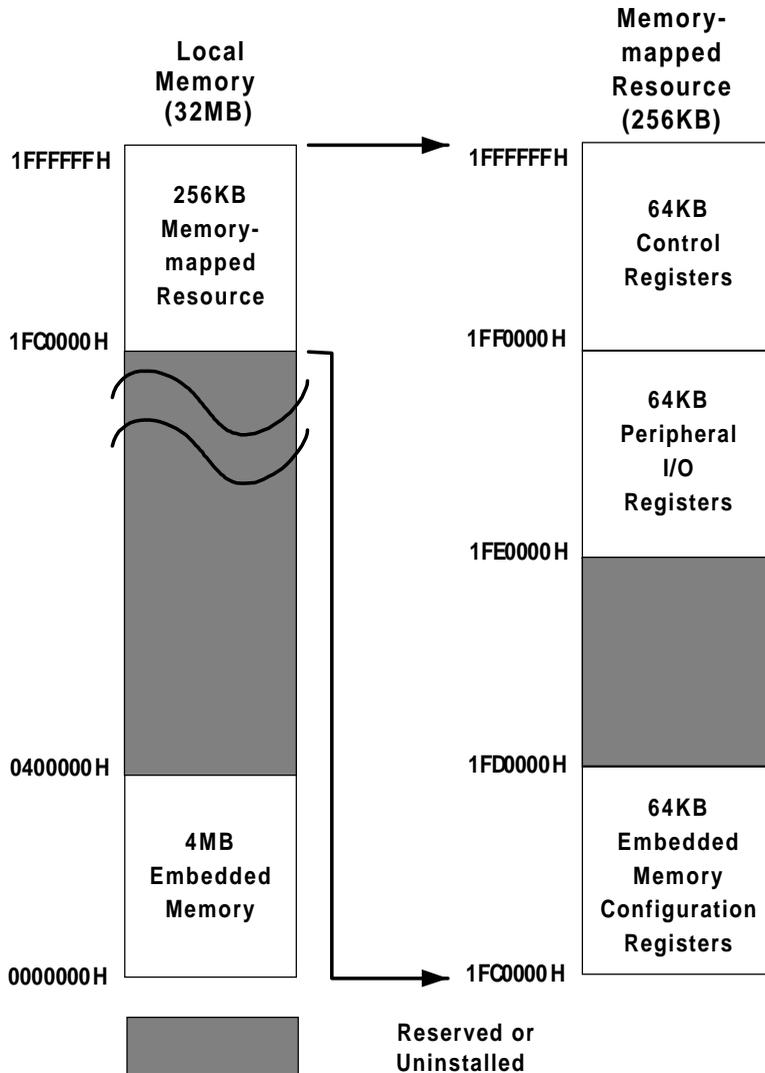


Figure 6-2. Memory-mapped Resources

6.2.3 Control Address Space Memory Mapping

The 64KB Control Address space is divided into five regions as shown in Figure 6-3. Each of these regions is discussed briefly in the following sections. The registers which access these regions are defined in Section 4, Register Summary.

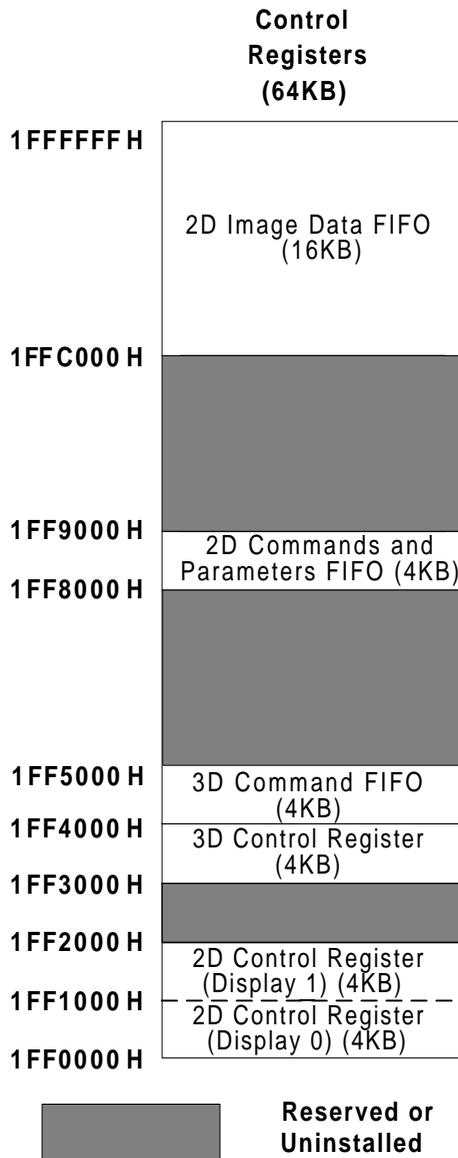


Figure 6-3. Control Address Space

6.2.3.1 2D Control Registers (Display 0/Display 1)

The 2D control registers occupy two 4KB (8KB total) memory-mapped address spaces. Registers that are common to both Display 0 and Display 1 are at the same offset within their respective 4KB address spaces. The layout of this region is shown in Figure 6-4 and described in detail in Section 4, Register Summary.

+F00H	PCI Config		
+E00H	LCD Timing	CRT Timing	
+D00H	LCD Control	CRT Control	
+C00H	Buffer Descriptors		
+B00H	Channel Descriptors		
+A00H	Surface Descriptors		
+900H	Ext Mem Control		
+800H	2D Engine		
+500H	LCD Panel Control		
+400H	Peripheral I/O		
+200H	Host Bus Master		
+000H	Status/Control		Status/Control
	+0xxxH		+1xxxH

Figure 6-4. 2D Control Register Space

6.2.3.2 3D Control Registers

The 3D control registers occupy a 4KB memory-mapped address space. The layout of this region is shown in Figure 6-5 and described in detail in Section 4, Register Summary.

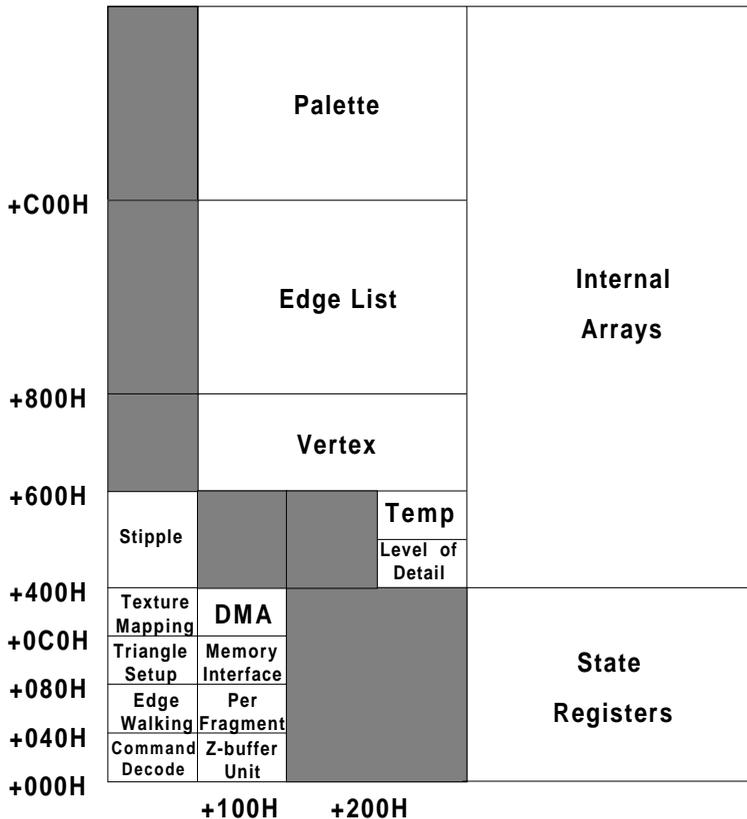


Figure 6-5. 3D Control Register Space

Access to 3D state registers is provided through a 4KB memory-mapped address space. Each register is directly accessible (via PCI) with byte granularity. State registers loaded via the command/parameter stream are, however, only writable with a granularity of 16 bits, i.e., all bits within the 16-bit word are modified. For this reason, control fields are grouped so that fields that are typically modified together are located in the same or nearby words. For PCI access, data must be correctly byte-aligned. These registers can be read and written directly for diagnostic and debug purposes.

6.2.3.3 3D Command/Parameter FIFO

A 4KB input (write-only) FIFO port is reserved for command and parameter transfers from the host to the 3D rendering engine. All accesses to anywhere in the window will be written to the location pointed to by the FIFO write pointer. The actual memory region reserved for the FIFO is configurable in 16KB increments up to 1MB on 16KB boundaries. It can be located at any addressable memory region within the 32MB local address space (see 3D Command FIFO)

6.2.3.4 2D Command/Parameter FIFO

A 4KB input (write-only) FIFO port is reserved for command and parameter transfers from the host to the 2D rendering engine. All accesses to anywhere in the window will be written to the location pointed to by the FIFO write pointer. The actual memory region reserved for the FIFO can be configurable to 1KB, 8KB, 16KB, or 32KB in size. It can be located at any addressable memory region addressable within the 32MB local address space.

6.2.3.5 2D Image Data FIFO

A 16KB input (write-only) FIFO port is reserved for 2D image data transfers from the host. All accesses to anywhere in the window will be written to the location pointed to by the FIFO write pointer. The actual memory region reserved for the FIFO can be configurable to 1KB, 8KB, 16KB, or 32KB in size. It can be located at any addressable memory region within the 32MB local address space.

6.2.4 Display Memory Arrays

The following arrays are located in display (internal/embedded DRAM) memory:

6.2.4.1 Surfaces

All 2D operations operate on surfaces defined by surface descriptor registers that specify the base, stride and pixel format of the specific array.

6.2.4.2 Texture Memory

Texture memory consists of multiple bitmaps containing different views of textures to be applied to objects. A base and end register in increments of 4KB define the region that may be used for texture memory.

6.2.4.3 Z-Buffers

The Z-buffer is typically a 16-bit bitmap the same size as the front and back buffers. It is initialized to a constant value representing the largest (farthest) position from the viewer. It is accessed sequentially in short horizontal spans just prior to rendering the span. If the Z value of the pixel being rendered is less than the stored Z value, the pixel is in front of the previously rendered pixel and it is written to the back buffer. The Z-buffer is then updated with the new Z value. The most frequent Z-buffer accesses are short sequential reads followed by short sequential writes (masked by the byte enable).

6.2.4.4 Memory Buffer Allocation

Memory for textures, draw and Z-buffer are stored within the same logical address space. The entire address space, however, need not be contiguous, i.e., it may be composed of several different regions.

The following restrictions apply to the allocation of buffers:

- The command/parameter/data FIFO must be in display memory. This buffer's size is programmable up to one megabyte.
- The Z-buffer must be contiguous.
- Each of the front and back buffers must be contiguous within a region.
- Textures must be in display memory.

6.2.5 Byte Ordering

As previously discussed, the SM3110 supports both little- and big-endian accesses to the entire linear address space, determined by the most significant address bit ([26]). In big-endian mode, two different byte orders are supported, selected by the 2nd most significant bit ([25]) of the address. Because both data registers and memory-mapped enhanced control registers are accessible in this linear address space, it is possible to access both memory and registers in either byte ordering by using different addresses, even on an individual doubleword basis.

A[26:25]	Endian	Swapping	Processor
00	little endian	Bytes within DWORDs	x86, Alpha
01	<i>Reserved</i>		
10	big endian	Bytes within DWORDs	PowerPC: 8, 32bpp
11	big endian	Bytes within WORDs	PowerPC: 16bpp

Table 6-2. Byte Order Selection

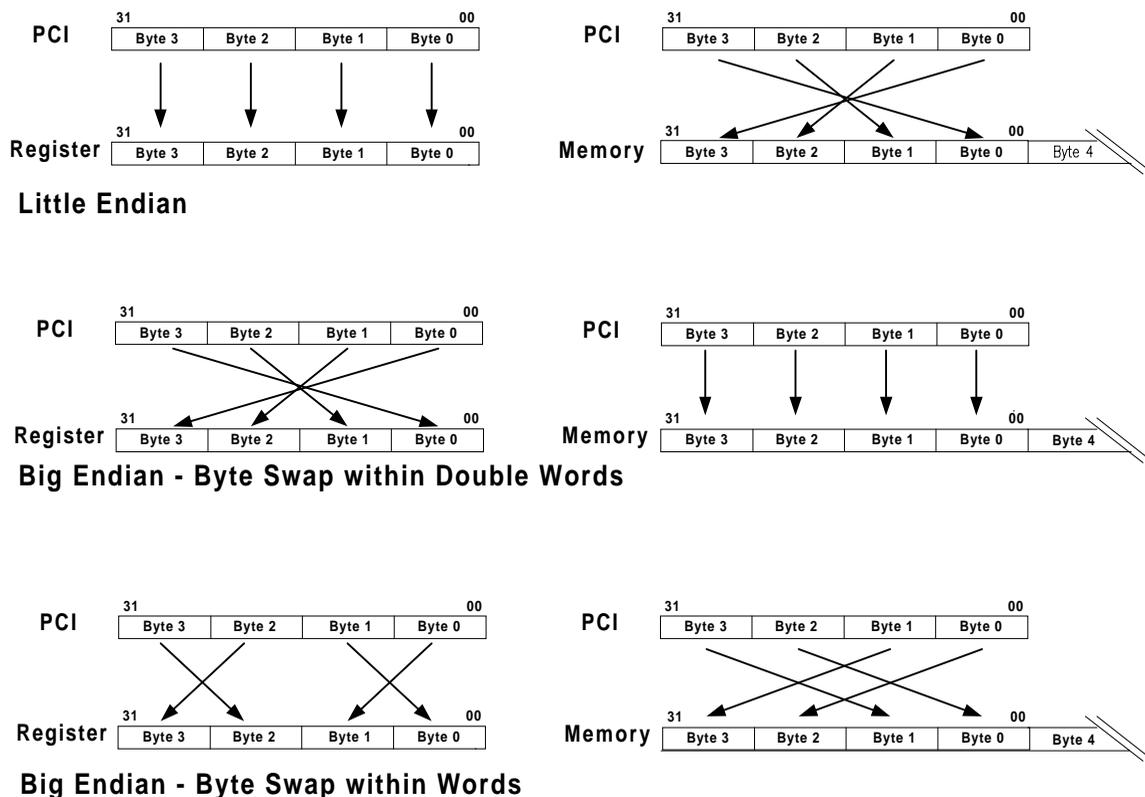


Figure 6-5. Byte Ordering

6.3 BIOS Functions

VGA BIOS (Basic Input and Output Service) provides many low level services to higher level programs. The interface is defined by IBM and VESA committee. Section 5, BIOS Specification, of this document describes in detail all the supported functions. It also defines all the modes supported by the SM3110. This section describes how the BIOS programs the SM3110 in order to achieve those tasks defined in the BIOS Specification.

Among all the services provided by the BIOS, the two most important are initialization and mode setting. The following sections describe the exact sequence the BIOS performs to achieve these two tasks.

6.3.1 BIOS Initialization: Power-On Self Test (POST)

Prior to using any of the functions of the SM3110, several initialization steps are required. These actions are performed as part of the Power-On Self Test (POST) of the BIOS.

POST will be executed during cold boot or warm boot. During cold boot, a hardware-reset signal connected to the chip will be toggled, followed by a call into POST. For warm boot (when the user hits Ctrl-Alt-Del), POST is the only function that gets called. Therefore, POST has the responsibility to initialize the chip back to its reset state. The pseudocode below shows how the BIOS initializes the hardware during the POST function.

```

0x3C3 ? 1                // Turn on video sub-system

0x3CE _ 0x80            // Max locking register
0x3CF _ 0x53            // 'S' - first unlock code
0x3CF _ 0x4D            // 'M' - second unlock code, unlock
                        // extended registers

Program 0x80 to 0x82    // Memory clock

// Initialize following registers
0xF1 _ 0                // Disable all interrupts
0x2F1 _ 0                // Disable host DMA interrupt
0x224 _ 0                // Clear host DMA count
0x8F1 _ 0x80            // Reset 2D engine
0x910 _ 0x37            // Internal refresh rate divide ratio
0xDF5 _ 0                // Disable double scan for display 0
0x1DF5 _ 0              // Disable double scan for display 1
0xDF7 _ 0                // Set DAC 0 to normal
0x1DF7 _ 0              // Set DAC 1 to normal
0xDF4 _ 0                // Disable ECRT 0 - set to VGA
0x1DF4 _ 0              // Disable ECRT 1
0xD00 _ 0                // Disable cursor channel 0
0x1D00 _ 0              // Disable cursor channel 1
0xD10 _ 0                // Disable icon channel
0xD20 _ 0                // Disable scaler channel 0
0x1D20 _ 0              // Disable scaler channel 1
0xD30 _ 0                // Disable display channel 0
0x1D30 _ 0              // Disable display channel 1
    
```

```
0xCF0_0 // Disable deferred control 0
0xCF4_0 // Disable deferred control 1
0xF04_7 // PCI command register

// Initialize LCD registers
Disable stretching - 0x504
Power management Control - 0x508
LCD type and misc. - 0x 510
Dithering and frame rate modulation - 0x514
DSTN starting address - 520

Call set mode routine to set to mode 3

Detect CRT // Check whether monitor is attached

Turn on 0x3C4 index 1 bit 5 // Turn off screen

Memory test // Also detect memory size

Test DAC

Exit
```

6.3.2 Set Mode

After POST, setting mode becomes the most essential portion of BIOS. Its responsibility is to set the hardware to a known working state. The pseudocode below shows how the BIOS handles this function.

```

0xDF4 _ 0          // Turn off ECRTC, enable VGA CRTIC
0xDF7 _ 0          // Set DAC to normal
3C4.1 bit 5 _ 1    // Turn off screen
If(Standard VGA Mode)
{
    Set standard VGA registers - 3C4/5, 3D4/5, 3C0/1, 3CE/F, etc. according to
                                the mode
}
Program 3C8, 3C9    // Initialize palette if necessary

// Reset following registers
0xD00 _ 0          // Disable cursor channel 0
0xD10 _ 0          // Disable icon channel
0xD20 _ 0          // Disable scaler channel 0
0xD30 _ 0          // Disable display channel 0
0x8F1 _ 0x80       // Reset 2D engine
If(Set to enhanced mode)
{
    // Program following registers
    0xB34 _ 4        // Set display 0 channel to surface 4
    0xB30 _ 0        // Surface offset for display 0 channel
    0xE30 _ 0        // Display 0 view port start
    0xA40 _ 0        // Surface 4 base address
    0xD80 _ 1        // scaler and display priority
    0xD90 _ 0        // Disable color key
    0xDF8 _ 0        // Clear display FIFO status
    0xCF0 _ 0        // Disable deferred 0
    0xCF4 _ 0        // Disable deferred 1
    0xEE0 _ 0        // Display 0 sync selector
Program E80 - E94    // Timing parameters for ECRTC 0
Program 0xE34        // View port end
Program 0xA44        // Surface 4 stride and format
Program 0xD30        // Display 0 format
Program 0xDF5        // Double scan control
0xDF4 _ 5          // Turn on ECRTC
}
Program 0x90, 0x91, 0x92 // Dot clock

Clear screen if needed

0x3C4.1 bit 5 _ 0    // Turn on screen

Exit

```

6.4 Driver Functions

The lowest-level driver for the SM3110 is responsible for these functions:

- device detection
- configuration
- memory-mapped control addressing
- memory address allocation
- physical buffers for FIFO's
- linear/segmented mode
- cursor control
- icon programming

Each of these topics is presented in this section.

6.4.1 Device detection

This PCI interrupt routine is called to detect the SM3110 as a PCI/AGP device:

Sample code : Sense presence of SM3110

```

SenseHardware    proc    near
    mov    si, 0                ; Search from 0
SH_FindPCIDevice:
    mov    cx, DEVICEID        ; =4831h, Device ID for SM3110 board
    mov    dx, VENDORID        ; =8888h, Vendor ID for Silicon Magic
    mov    ax, 0B102h          ; Find device command
    int    1Ah                 ; PCI interrupt
    jnc    SH_Found            ; Found

    cmp    ah, 86h             ; Error code in ah - not found
    je     SH_NotSiMagic       ; Search all slots and not found

SH_FindNextSlot:
    inc    si                   ; Search next slot
    cmp    si, 256
    jb     SH_FindPCIDevice

SH_NotSiMagic    ; Search all slots and not found
    xor    ax, ax               ; means fail
    ret

SH_Found:
    mov    wPCIBusAddress, bx   ; bx=bus/device number, save in global
    mov    ax, 1                ; means successful
    ret
SenseHardware    endp
  
```

6.4.2 Device configuration

Once the SM3110 device is detected, the following routine is used to read the PCI/AGP configuration registers.

Sample code: Read PCI/AGP Configuration register

```
ReadPCIDword    proc    near
;;Entry:        ax=index
;;Return:       eax=double word be read
    mov    bx, wPCIBusAddress    ; get from SenseHardware sub-routine
    mov    di, ax                ; index
    mov    ax, 0B10Ah           ; AH:PCI function ID,AL:Read a dword
    int    1Ah                  ; PCI interrupt, return in ECX
    mov    eax,ecx              ; return in EAX
    ret
ReadPCIDword    endp
```

6.4.3 Memory-mapped control address

The last 64KB of the local memory address space is reserved for memory-mapped 2D/3D control registers.

6.4.3.1 Enable/Disable access

To prevent inadvertent access to the enhanced memory-mapped control registers, a specific sequence must be written through the VGA Graphics Controller index/data port to enable access.

Sample code: Enable/Disable memory-mapped control register access

```
EnableIOMap    proc    near
    mov     dx,3ceh
    mov     ax,5380h    ;write S into reg 80h
    out    dx,ax
    mov     ax,4D80h    ;write M into reg 80h
    out    dx,ax
    ret
EnableIOMap    endp

DisableIOMap   proc    near
    mov     dx,3ceh
    mov     ax,5280h    ;write L into reg 80h
    out    dx,ax
    ret
DisableIOMap   endp
```

Since PCI devices will be allocated with different memory addresses, the enable code is programmed just once for insurance, and the disable code should never be used.

6.4.3.2 2D Command Port selector

Use the same way as above to set a variable to the base address of the memory-mapped control register region. This is also the base address of the 2D control region.

Sample code: Get the 2D command port selector

```

GetPointer_MemoryMapCtl proc          near
    call    GetLocalMemoryBase
    mov     eax, PhyVRAMAddress
    add     eax, 1FF0000                ;32M-64K
    mov     PhyCmdPortAddress,eax      ;save as global
    mov     edx, 10000h                ;size=64K
    call    GetSelector
    mov     CmdPortSelector, ax        ;save as global variable
    ret
GetPointer_MemoryMapCtl endp
    
```

After calling `GetPointer_MemoryMapCtl` routine during initialization, the following macros can be used to read/write memory-mapped control registers at any offset.

```

ReadMemoryMapCtl Macro offset,data
    mov     es, CmdPortSelector
    mov     di,offset
    mov     data,es:[di]
Endm
    
```

```

WriteMemoryMapCtl Macro offset,data
    mov     es, CmdPortSelector
    mov     di,offset
    mov     es:[di], data
Endm
    
```

6.4.3.3 2D Image Data Port selector

The 2D Image Data FIFO is located at C000H from the memory mapped control address. Instead of using the 2D Command port selector with offset C000H, it is easier to understand allocating another global variable (selector) to address the area directly.

Sample code: Get the 2D image data FIFO port selector

```
GetPointer_ImagePort    proc    near
    mov     eax, PhyCmdPortAddress
    add     eax, 0C000h        ;last 16K of command port
    mov     edx, 4000h        ;size=16K

    call    GetSelector
    mov     ImagePortSelector, ax    ;save as global variable
    ret
GetPointer_ImagePort    endp
```

After calling `GetPointer_ImagePort` routine at initialization period, the following macros can be used to write data to 2D image data FIFO.

```
WriteImagePort    Macro    offset,data
    mov     es, ImagePortSelector
    mov     di,offset
    mov     es:[di],data
endm
```

6.4.3.4 VGA I/O Ports

Program the VGA I/O port (3Cxh, 3Bxh or 3Dxh) by writing index/data pairs.

Sample Code: Program VGA I/O port

```
VGAIOport    proc    near
    mov     dx, port        ; 3Cxh, 3Bxh or 3Dxh
    mov     al, index
    out     dx, al
    inc     dx
    mov     al, data
    out     dx, al
    ret
VGAIOport    endp
```

6.4.3.5 *A0000H-based 128K VGA memory*

For 16-bit real mode, use A0000H as the segment. A0000H:0 points to screen at (0,0) and A0000H:C30H points to pixel at (48,3) for 1024x768x256colors mode.

For 16-bit protected mode, allocate a selector, set its base to A0000H and set its limit to dwVRAM-Size. Then use the selector:offset pair to access video memory (see also the following section, Local Memory Address Space).

6.4.3.6 *PCI configuration space*

In addition to calling subroutine `ReadPCIDword`, the PCI configuration register can also be accessed from the offset 0F00H to the memory-mapped control selector. For example,

```
mov    ax,30h
call   ReadPCIDword           ;return in eax
```

has the same effect as

```
mov    fs, CmdPortSelector
mov    eax, fs:[0F30h]
```

6.4.4 Memory Address Allocation

Section 6.2 described the address space of the SM3110 and all of its subdivisions. This section describes how each of these is accessed by software.

6.4.4.1 Local memory address space

The SM3110 Address Space occupies four contiguous 32MB address ranges (of the total 4GB PCI address space). The Local Memory Address Space is the lower 32MB of the 128MB range. The PCI address of the local memory address space is located in the BASE 0 index of the PCI configuration register. Use the following routine to get this physical address.

Sample code: Get the physical address of the base of local memory.

```
GetLocalMemoryBase    proc    near
    mov    ax, 10h                      ;Base 0 physical address index
    call   ReadPCIDword
    and    eax,0FF000000h                ;align
    mov    PhyVRAMAddress,eax           ;save in global,will be used later
    ret
GetLocalMemoryBase    endp
```

6.4.4.2 Display (video) memory address

The display (video) memory starts from the beginning of the local memory address, with a physical memory size which can be set in a global variable by the following routine.

Sample code: Get display memory size

```
GetVideoMemSize  proc  near
    xor    eax,eax
    mov    dx,3ceh
    mov    al,8fh                ;BIOS save bank size during POST
    out   dx,al
    inc   dx
    in    al,dx                ;al has size in 64k units
    shl   eax,16              ;size in eax
    mov   dwVRAMSize,eax      ;save in global, will be used later
    ret
GetVideoMemSize  endp
```

To access the local memory, the physical address needs to be converted to a linear/logical address:

Sample code: Convert Physical to Linear Address

```
ConvertPhysicalToLinearAddress proc                                near
;;Entry:  eax=physical address, edx=size
;;Return: eax=linear address
    shld  ebx,eax,16
    mov   cx,ax                ;BX:CX has base physical address
    mov   edi,edx
    dec   edi
    shld  esi,edi,16           ;SI:DI has limit
    mov   ax,0800h             ;convert physical to linear address.
    Int   31h                  ;DOS Protected Mode Interface (DPMI)
                                call, return BX:CX = linear address

    shrd  eax,ebx,16
    mov   ax,cx                ;eax= linear address
    ret
ConvertPhysicalToLinearAddress endp
```

For 16-bit programs, data is pointed to by segment:offset pair (in real mode) or by selector:offset pair (in protected mode).

6.4.4.3 16-bit Real Mode Addressing

In 16-bit real mode (i.e., DOS environment), data is pointed to by segment:offset pair. For example, to access memory at B8004h, program as follows:

Sample code: Read/Write data to segment:offset in 16-bit real mode

```

mov     ax,0B000h           ;segment or B800h
mov     es, ax
mov     di, 8004h          ;offset or 4h

```

Then use es:[di] to read/write data.

6.4.4.4 16-bit Protected Mode Addressing

In 16-bit protected mode (i.e., Microsoft Windows environment), data is pointed to by selector:offset pair. The segment needs to be converted to a selector by DOS Protected Mode Interface (DPMI) calls.

Sample code: Get a selector in 16-bit protected mode

```

GetSelector     proc    near
;; Entry:      eax= dwPhysAddr,  edx=dwSize
;; Return:     ax=selector assigned
    mov     dwSize, edx           ;save in stack or global variable
    mov     dwPhysAddr, eax       ;save in stack or global variable

    xor     ax, ax
    mov     cx, 1                 ;count
    int     31h                  ;DPMI, allocate an LDT selector in ax
    mov     wSelector, ax        ;save selector in stack or global variable

    mov     eax, dwPhysAddr
    mov     edx, dwSize
    call    ConvertPhysicalToLinearAddress    ;return eax=linear addr

    shld   ecx, eax, 16
    mov    dx, ax                 ;CX:DX = linear address
    mov    bx, wSelector         ;BX=selector
    mov    ax, 7
    int    31h                  ;DPMI, set selector base.

    mov    edx, dwSize
    dec   edx
    shld   ecx, edx, 16         ;CX:DX = limit
    mov    ax, 8
    int    31h                  ;DPMI, set selector limit

    mov    ax, wSelector        ;return value in ax
    ret

GetSelector     endp

```

6.4.4.5 32-bit Mode Addressing

For 32-bit code, since all memory is within the 4GB logical space, the linear address points to the proper location of the memory.

By combining with 16-bit protected code, use the following routine to get display (video) memory pointer:

Sample code: Get the Video Memory pointer

```
GetPointer_VideoMemory proc near
    call GetLocalMemoryBase
    call GetVideoMemSize
    mov  eax, PhyVRAMAddress
    mov  edx, dwVRAMSize
ifdef USE_32BIT
    call ConvertPhysicalToLinearAddress
    mov  dwVRAMLinearAddress,eax           ;save as global variable
else ;;USE_16BIT protected mode
    call GetSelector
    mov  wVRAMSelector,ax                ;save as global variable
endif
    ret
GetPointer_VideoMemory endp
```

After calling the `GetPointer_VideoMemory` routine at initialization time, the following macros can be used to read/write display (video) memory:

Sample code: Read/Write Video Memory

```

#ifdef USE_32BIT

ReadVideoMemory      Macro      offset,data
    mov     edi, dwLinearAddress_VideoMemory
    add     edi,offset
    mov     data,[edi]
Endm

WriteVideoMemory     Macro      offset,data
    mov     edi, dwLinearAddress_VideoMemory
    add     edi,offset
    mov     [edi],data
Endm

Else                  ;;USE_16BIT protected mode

ReadVideoMemory      Macro      offset,data
    mov     es, wVRAMSelector
    mov     di,offset
    mov     data,es:[di]
Endm

WriteVideoMemory     Macro      offset,data
    mov     es, wVRAMSelector
    mov     di,offset
    mov     es:[di],data
Endm

#endif
```

Note: For simplicity, only 16-bit protected mode code will be used for the remainder of this document.

6.4.5 Physical Buffers for FIFOs

When programming or addressing data in the 3D command, 2D command and 2D image data FIFO areas, the SM3110 chip hardware will first decode the address and transfer data to the physically allocated buffer and then execute the command or retrieve data in a FIFO order. These buffers must be physically allocated somewhere in the 4MB display memory. By convention: use the last 128KB for these buffers

- 3D command FIFO buffer: 64KB
- 2D image data FIFO buffer: 32KB
- 2D command FIFO buffer: 4KB
- Cursor Deferred control buffer: 1KB
- Display Deferred control buffer: 1KB
- Cursor buffer: 2KB
- Icon: 2KB
- others

For example, consider the 2D command FIFO. The 2D command FIFO can only be accessed within a 4KB range: offsets +8000H to +9000H within the Control Address Space. However, the SM3110 can actually access up to 32KB by using addresses in this range repeatedly. The following code shows how this is done (in this example only 16KB is accessed).

Sample code: Set buffer locations for FIFOs

```
GetFIFOBufferLocation proc near
    call GetLocalMemoryBase
    call GetVideoMemSize
    mov     eax, PhyVRAMAddress
    add     eax, dwVRAMSize
    sub     eax, 20000h ;reserved last 128K for FIFO buffer
    mov     dwPrivatePhyAddr, eax;save as global
    ret
GetFIFOBufferLocation endp
```

Please refer to “Equates” section. Here is code to set buffer addresses:

```
SetBufferBase_2DCmdFIFO proc near
    call GetFIFOBufferLocation
    add     eax, CMDFIFO_OFFSET
    or      eax, ECMDFIFOSIZE
    mov     fs, CmdPortSelector
    mov     fs:[CMDBufferBaseAddress],eax
    ret
SetBufferBase_2DCmdFIFO endp

SetBufferBase_2DImageFIFO proc near
    call GetFIFOBufferLocation
    add     eax, IMGFIFO_OFFSET
    or      eax, EIMGFIFOSIZE
    mov     fs, CmdPortSelector
    mov     fs:[IMGBufferBaseAddress],eax
    ret
SetBufferBase_2DImageFIFO endp
```

6.4.6 Linear/Segmented mode

Display memory can be configured as a large contiguous address space (linear mode) up to 32MB, or as multiple 64KB segments (segmented mode). For linear mode, it is easy to access the memory by 32-bit linear pointer or selector:[32-bit offset]. Usually in 16-bit programs with 16-bit offsets such as BX,SI or DI, the maximum value is 64KB. To access memory beyond 64KB requires setting the memory bank.

Sample Code: Program VGA Memory bank

```

GetVGAMemoryBank proc near
    mov     dx, 3CEh
    mov     al, 84h           ;bank offset register index
    out     dx, al
    inc     dx
    in      al, dx           ;bit 6:1=bank in 64K unit
    and     ax,7Fh          ;bit 7=1 for packed pixel mode banking mode
    shr     ax,1
    ret                                           ;return ax=bank #
GetVGAMemoryBank endp

SetVGAMemoryBank proc near ;ax=bank #
    add     al, al           ;bit 6:1=bank in 64K unit
    or      al, 80h         ;bit 7=1 for packed pixel mode banking mode
    mov     dx, 3CEh
    mov     ah, 84h         ;bank offset register index
    out     dx, ax
    ret
SetVGAMemoryBank endp

```

Thus, after programming

```

mov     ax,1
call    SetVGAMemoryBank

```

wVRAMSelector:[0] points to the pixel at (0,64) instead of (0,0) in 1024x768x256color mode. Obviously, accessing memory in linear mode is easier, simpler and recommended.

6.4.7 Cursor control

The SM3110 supports both monochrome and color cursors. The cursors can be turned on and off. The turning on and the positioning of the cursor can be synchronized with VBLANK.

6.4.7.1 Cursor Format

The monochrome cursor can be up to 256 x 256 pixels in size. It is defined as two 1-bit-per-pixel (bpp) masks: the AND mask and the XOR mask. The cursor display logic is described below.

AND Mask	XOR Mask	Result
0	0	Background Color (BLACK)
0	1	Foreground Color (WHITE)
1	0	Screen Pixel
1	1	NOT Screen Pixel

The color cursor can be up to 32 x 32 pixels. It is defined as a one-bit-per-pixel AND mask and a 16-bit-per-pixel color bitmap which is the XOR bitmap. The cursor display logic is described below.

AND Mask	XOR Bitmap	Result
0	Cursor Pixel	Cursor Pixel
1	Cursor Pixel	Screen Pixel XOR Cursor Pixel

6.4.7.2 *Cursor data buffer and surface*

The cursor pattern is stored in the offscreen memory. Cursor buffer allocation is mentioned in Section 6.4.5, Physical Buffers for FIFOs. A surface descriptor has the physical base address of the cursor in the offscreen memory. This example shows how to prepare buffer before setting cursor.

Sample code: Set buffer location for cursor and deferred control

```
;fs= CmdPortSelector: selector points to the register base address
SetBuffer_Cursor proc near
    mov     eax, dwPrivatePhyAddr      ;get from GetFIFOBufferLocation
    addeax, CURSOR_OFFSET
    sub     eax, PhyVRAMAddress       ;relative to local memory base
    mov     dwCursorBufferOffset,eax  offscreen to load the cursor pattern,
                                     save as global
    mov     fs:[CURSOR0SURFACE],eax   ;surface descriptor base
    mov     fs:[ChCurOffsets],0      ;offsets into cursor surface, maybe !=0
                                     if clipped
    mov     fs:[ChCurSIndex],CURSOR0SURFACEINDEX ;set cursor surface
                                     descriptor index
    ret
SetBuffer_Cursor endp
```

6.4.7.3 Set Monochrome Cursor data

For monochrome cursor, the cursor source (i.e., from Windows) is specified as two 1 BPP bitmaps, the AND and XOR masks. SM3110 defines the cursor surface as 1 BPP format, and sets the stride to 2 times the cursor width. These two masks are written to the cursor surface buffer as interleaved words with the AND mask followed by the XOR mask.

Following example sets a monochrome cursor with size of 256 x 256 (the maximum possible size). Note that the cursor surface stride is fixed at 256 pixels (2 BPP).

Sample code: Set Monochrome cursor

```

;fs= CmdPortSelector: selector points to the register base address
;ds:[esi] points to the cursor pattern, 512 bytes of AND mask followed by 512
                        bytes of XOR.

SetMonochromeCursor    proc    near
    mov     es, wVRAMSelector
    mov     edi, dwCursorBufferOffset        ;es:[edi] points to the offscreen
                                                to load the cursor pattern.

    mov     fs:[CursorFormat],CURSOROFF     ;turn off cursor
    mov     gdwCursorFormat, CURSOR1       ;global variable to turn on cur-
                                                sor later

    mov     eax,(256*2)                     ;stride*2 - only for mono cursor
    or     eax,BPP1 shl 20;surface is 1 BPP
    mov     fs:[CURSOR0SURFACE+4],eax      ;set stride and format

    mov     ecx,64                          ;count of rows to handle
loop_nextrow:
    mov     bx,4
loop_inlrow:
    mov     ax,ds:[esi+512];load XOR mask
    shl     eax,16                          ;transfer to high bits
    loadw                                ;load AND mask ls 16 bits
    stosd                                ;save AND & XOR masks in off
                                                screen memory

    dec     bx
    jnz     loop_inlrow
    add     edi,256*2/8 - 16                 ;offset into next row
    loop   loop_nextrow
    ret
SetMonochromeCursor    endp

```

Note: The cursor is now set and needs to be turned on and positioned. The monochrome cursor can be a maximum of 256 x 256 pixels. If the given cursor size is smaller, the cursor pattern is loaded into the top left corner of the cursor surface and the viewport adjusted to display the cursor. The CursorPositionStarts and CursorPositionEnds are set based on the cursor width and height; the desired portion of the cursor surface is displayed.

6.4.7.4 Set Color Cursor data

For color cursor, the cursor source (i.e., from Windows) is specified as a 1 BPP AND mask and a color bitmap which can be 8, 16 or 24 BPP. SM3110 defines the cursor surface as a 16 BPP format with the desired stride in pixel units. The color pixels and the AND mask are combined into 16-bit pixels with the most significant bit containing the AND mask and the remaining 15 bits containing RGB value with a 5-5-5 format. The resulting 16-bit pixels are written into the cursor surface.

The cursor size is 32x32 for this example, the maximum possible size for color cursors. Assume the current mode is 16 BPP, RGB 5:6:5 mode. The given cursor definition has 128 bytes of monochrome bitmap (the AND mask) and 2KB of color bitmap (the XOR component). The color cursor surface stride has to be 32 pixels (16 BPP).

Sample code: Set Color cursor

```

;fs= CmdPortSelector: selector points to the register base address
;ds:[esi] points to the cursor pattern,128 bytes of AND mask followed by 2048
                        bytes of color bitmap.

SetColorCursor  proc          near
    mov     es, wVRAMSelector
    mov     edi, dwCursorBufferOffset ;es:[edi] points to the offscreen to
                                        load the cursor pattern.

    mov     fs:[CursorFormat],CURSOROFF ;turn off cursor
    mov     gdwCursorFormat, CURSOR16 ;global variable to turn On cursor later
    mov     eax,32                      ;stride
    or      eax,BPP16 shl 20            ;surface is 16 BPP
    mov     fs:[ CURSOR0SURFACE+4],eax  ;set stride and format

    mov     ecx,32*2                    ;count of words to handle
    mov     ebx,128                    ;init. offset to color bitmap
OLOOP:
    mov     ax,ds:[esi]                 ;load AND mask
    rol     eax,16                      ;and mask in MS 16 bits
    mov     dx,16                      ;inner loop count
@@:
    mov     ax,ds:[esi+ebx]             ;load color component
    rcr     ax,6                        ;convert 565 to 555, discard LS Green
                                        bit
    rol     ax,5                        ;prepare to load AND mask in MS bit
    rcl     eax,1                       ;get AND mask bit into carry
    rcr     ax,1                        ;rotate into MS bit with 555 color
    add     ebx,2                       ;offset to next pixel of color bitmap
    stosw                                ;store AND mask and color component into
                                        offscreen
    dec     dx                          ;process 16 pixels
    jnz     @B
    add     esi,2                       ;point to next word of AND mask
    loop   OLOOP
    ret
SetColorCursor  endp

```

Note: The cursor is now set and needs to be turned on and positioned. The color cursor size can be a maximum of 32 x 32 pixels. If the given cursor size is smaller, the cursor pattern is loaded into the top left corner of the cursor bitmap and the viewport adjusted to display the cursor. The CursorPositionStarts and CursorPositionEnds are set based on the cursor width and height; the desired portion of the cursor surface is displayed. The 24 BPP, RGB 888 bitmaps are converted to a 16-bit, 555 format by packing the most significant 5 bits of the red, green and blue components. If the color depth is 8 BPP, the RGB values for each color index (pixel) are converted to an RGB 555 word. No conversion is necessary for 16-bit RGB 555 bitmaps.

6.4.7.5 *Cursor Deferred control*

To avoid the tearing or jumping of the cursor image, all writes to the cursor registers are recorded into the Deferred Cursor Buffer and rewritten during VBLANK. The immediate register address, with encoded byte enables followed by register data, is recorded. The immediate registers are updated with the recorded data during VBLANK. Macros LDAB, LDAH and LDAD are used to load the deferred offsets with encoded byte enables. These macros are defined in the “Equates” section.

Sample code: Set buffer location for cursor and deferred control

```

;fs= CmdPortSelector: selector points to the register base address

SetBuffer_CursorDeferredCtl    proc    near
    mov     eax, dwPrivatePhyAddr    ;get from GetFIFOBufferLocation
    add     eax, CURDEFERRED_OFFSET
    sub     eax, PhyVRAMAddress      ;relative to local memory base
    mov     dwCurDeferBufferOffset,eax ;offscreen to load the cursor deferred
                                         (addr,data) pairs

    ret
SetBuffer_CursorDeferredCtl    endp

```

Usually SM3110 uses the “Deferred control pointer 0” as the cursor deferred control pointers. To use the deferred control, first check if the deferred control status is replaying. When idle, reset the read pointer to the buffer allocated above, write the (command address, value) pairs to the buffer, then update the write pointer to the address of the last pair. Finally, enable the replay bit in the status register. During the next VBLANK period, the SM3110 will get the data from the deferred read pointer and execute the ‘command’ with the following ‘value’. The read pointer will be automatically updated by the hardware after the data was read. It will not stop until the read pointer matches the write pointer. An example is shown in the next section of “Set Cursor Position”.

6.4.7.6 Set Cursor Position and Turn Cursor On

The cursor has been previously turned off and a new cursor shape has been set. The cursor display and position parameters have to be set during VBLANK to avoid tearing or jumping of the cursor. This is accomplished by writing to the Deferred buffer and enabling replay; and the real register updates occur during VBLANK. The cursor position can be set independently if the cursor was already on.

The cursor position and display size can be controlled, along with offsets which enable partial cursors to be displayed at the left and top edges of the screen. The cursor surface descriptor is set up for the base address, format and stride. The (x,y) offsets are set along with the (x,y) position of the top left corner of the cursor. The (x,y) offsets are set to (0,0) if the cursor is fully visible. Note that the cursor position (x,y) can only be positive numbers. To enable partial cursor to be visible at the left or top edges of the screen, the cursor position x or y is set to 0. The corresponding x or y offset, the portion of the cursor not visible in pixel units, is set in the offset register. The cursor format has to be set for the display portion along with the extents.

Sample code: Set Cursor Position and Turn Cursor On

```

;fs= CmdPortSelector: selector points to the register base address
;Cursor height and width are assumed 32x32
;Deferred Control buffer in the offscreen memory, selector es points to the
                        screen base .
;variables gCursorX/Y is cursor location, gCursorHotX/Y is hot spot inside
                        cursor,
;variables ScreenWidth/Height is current screen dimension.

MoveCursor      proc      near
;read Deferred control status - disarms Deferred control replay
@@:
    mov     al,fs:[CursorReplayStsReg]    ;wait while replay active
    test   al,REPLAYACTIVE;
    jnz    @B                             ;skip to load cursor if inactive

    mov     edi, dwCurDeferBufferOffset  ;physical buffer reserved for
                                        deferred control
    mov     fs:[CursorReplayPtr],edi     ;Reset replay pointer
    mov     es, wVRAMSelector            ;es:[edi]=offscreen for deferred control

    mov     eax, gdwCursorFormat; CURSOR1/16
    LDAD   ebx,CursorFormat;
    mov     es:[edi],ebx                 ;record reg address and data into
                                        deferred buffer
    mov     es:[edi+4],eax                ;turn on mono or color cursor
    add     edi,8

;clip cursor at Y axis
    xor     bx,bx                         ;assume cursor yoffset = 0
    mov     cx,32                          ;assume extent = 32
    mov     ax,gCursorY                    ;ax = screen coordinate of cursor
    sub     ax,gCursorHotY                 ;correct for hot spot
    jge     clipcursorY_bottom             ;Is adjusted coordinate negative?
    add     cx,ax                           ;yes, shorten extent
    sub     bx,ax                           ;advance yoffset
    xor     ax,ax                           ;set screen coordinate to zero.
    jmp     clipcursorY_done

```

```

clipcursorY_bottom:
    mov     dx,ax                ;dx = top of cursor
    add     dx,cx                ;dx = bottom+1 of cursor
    sub     dx,ScreenHeight     ;dx = # pixels cursor is clipped on bottom
    jle     clipcursorY_done    ;If 0 or neg, cursor is completely visible
    sub     cx,dx                ;adjust extent
clipcursorY_done:

    shl     eax,16               ;accumulate y position
    shl     ebx,16               ;offset
    shl     ecx,16               ;and y end in upper 16 bits
;clip cursor at X axis
    xor     bx,bx                ;assume cursor xoffset = 0
    mov     cx,32                ;assume extent = 32
    mov     ax,gCursorX         ;ax = screen coordinate of cursor
    sub     ax,gCursorHotX      ;correct for hot spot
    jge     clipcursorX_right   ;Is adjusted coordinate negative?
    add     cx,ax                ;yes, shorten extent
    sub     bx,ax                ;advance xoffset
    xor     ax,ax                ;set screen coordinate to zero.
    jmp     clipcursorX_done
clipcursorX_right:
    mov     dx,ax                ;dx = lhs of cursor
    add     dx,cx                ;dx = rhs+1 of cursor
    sub     dx,ScreenWidth      ;dx = # pixels cursor is clipped on right side
    jle     clipcursorX_done    ;If 0 or neg, cursor is completely visible
    sub     cx,dx                ;adjust extent
clipcursorX_done:
    add     ecx,eax;cx = (right,bottom)
;
    mov     es:[edi],edx         ;record reg address and data into deferred
                                ;buffer
    mov     es:[edi+4],eax       ; set cursor position top left x,y
    add     edi,8

    LDAD   edx,ChCurOffsets;
    mov     es:[edi],edx         ;record reg address and data into deferred
                                ;buffer
    mov     es:[edi+4], ebx      ;set cursor offsets
    add     edi,8

    LDAD   edx,CursorPositionEnds;
    mov     es:[edi],edx         ;record reg address and data into deferred
                                ;buffer
    mov     es:[edi+4], ecx      ;set bottom right position
    add     edi,8

    sub     edi,4                ;go back to the last valid address
    mov     fs:[CursorRecordPtr],edi ;set cursor write base pointer
    mov     fs:[CursorReplayCtlReg],ENABLEREPLAY ;Enable replay at next VBLANK
    ret
MoveCursor     endp

```

The cursor will be displayed at the next VBLANK.

6.4.8 Icon

The SM3110 supports a hardware icon.

Sample code: Set icon buffer

```
;fs= CmdPortSelector: selector points to the register base address
SetBuffer_Icon  proc  near
    mov     eax, dwPrivatePhyAddr      ;get from GetFIFOBufferLocation
    addeax, ICON_OFFSET
    sub     eax, PhyVRAMAddress       ;relative to local memory base
    mov     dwIconBufferOffset,eax    ;offscreen to load the icon pattern,
                                        save as global
    mov     fs:[ICONSURFACE],eax      ;surface descriptor base
    mov     fs:[ChIconOffsets],0      ;offsets into icon surface
    mov     fs:[ChIconSIndex],ICONSURFACEINDEX ;set icon surface descrip-
                                        tor index
    ret
SetBuffer_Icon  endp
```

Programming for the icon is similar to the cursor functions in the previous section. Follow those routines to set Monochrome/Color Icon to the allocated icon buffer, set icon position and turn on icon, except use “IconChFormat/IconChPositionStarts/ChIconOffsets/IconChPositionEnds” instead of “CursorFormat /CursorPositionStarts/ChCurOffsets/CursorPositionEnds” respectively.

6.5 2D Functions

6.5.1 Display Operations

The SM3110 controller provides a very powerful and flexible mechanism for defining surfaces and displaying multiple surfaces via channels. Multiple surfaces can be displayed with alpha blending and color keying.

The display memory can be logically viewed as surfaces. The surface descriptors define surfaces. Each descriptor has the base address of the surface (in physical memory), the stride (in pixels) and the pixel size (in bytes). Surface stride has to be a multiple of 16 pixels, and the base address has to start on a 4-byte boundary. The SM3110 allows 16 descriptors, 00H through 0FH. The 2D engine or the CPU can render into any surface. It is possible to perform inter-surface BitBLTs. Surfaces are displayed by assigning them to one of seven(7) channels: two cursors, one icon, two displays and two scalars. Surface descriptors 00 and 0Fh are reserved for cursor channels 0 and 1, descriptor 01h is reserved for the icon channel and descriptors 04h and 0Eh are reserved for display channels 0 and 1 by BIOS and the display driver.

There are two sets of cursor/scaler/display: for output channel 0 and output channel 1. The cursor/scaler/display set for output channel 0 is associated with the LCD output, and the cursor/scaler/display set for output channel 1 is associated with the CRT. A surface can be attached to a display channel. The attached surface is displayed on the screen through a viewport located at a specified position. The screen resolution is based on the mode currently set. The pixel format and size have to be specified for each channel. The viewport location and its extents (size) can be specified. It is thus possible to display any portion of a surface on the viewport, clipped at the left and top edges. The X,Y offsets become the origin into the surface bitmap that is displayed in the viewport. With the X,Y offsets set to 0,0, the bitmap is displayed with the top left corner coincident with the top left corner of the viewport. Figure 6-6 below shows an example of a surface being displayed by a display channel.

The icon channel is used to display ICON bitmap when LCD is powered down. The cursor channel is provided for hardware cursor support which is described in the previous section. The surface associated with the scaler channel can be of type RGB or YUV, and can be stretched or shrunk. Only RGB surfaces can be attached to display channels. The scaler channel can be used as a blend or as key control for the display channels. Certain modes can display only two channels due to bandwidth and/or memory limitations.

By convention, when an enhanced mode is set, surface 4 is allocated by the BIOS to describe the bitmap for that mode. This surface, the primary surface, is attached to a display channel. The surface 0 is reserved for the cursor channel. The base addresses for the primary surface and the cursor surface can be obtained from making extended BIOS calls. During mode setting, the cursor and scaler channels are turned off by programming the channel format to 0. The channels can be turned on later by programming with a legal format.

The surfaces attached to scaler and display channels can be displayed on the screen simultaneously. The overlay priority control register can specify the display order of the surfaces. The scaler channel can be assigned an order as foreground for display, or as a background surface which can be blended or colorkeyed with the foreground display channel. The blending can be constant defined by the Mix control register. The keying can be on the source (foreground channel) or the destination (background channel). If source keying is selected, foreground channel pixels having the key color display pixels from the background channel; the foreground channel

pixels not having the key color are displayed everywhere else. If destination keying is selected, foreground channel pixels corresponding to key color pixels in the background channel are displayed. The background channel pixels are displayed everywhere else in the overlapping area. If a control channel is specified for keying, it takes precedence over the key control registers. Figure 6-7 shows the overlay and mix control. In certain modes, it is possible to have only two channels displayed due to memory and or bandwidth limitations.

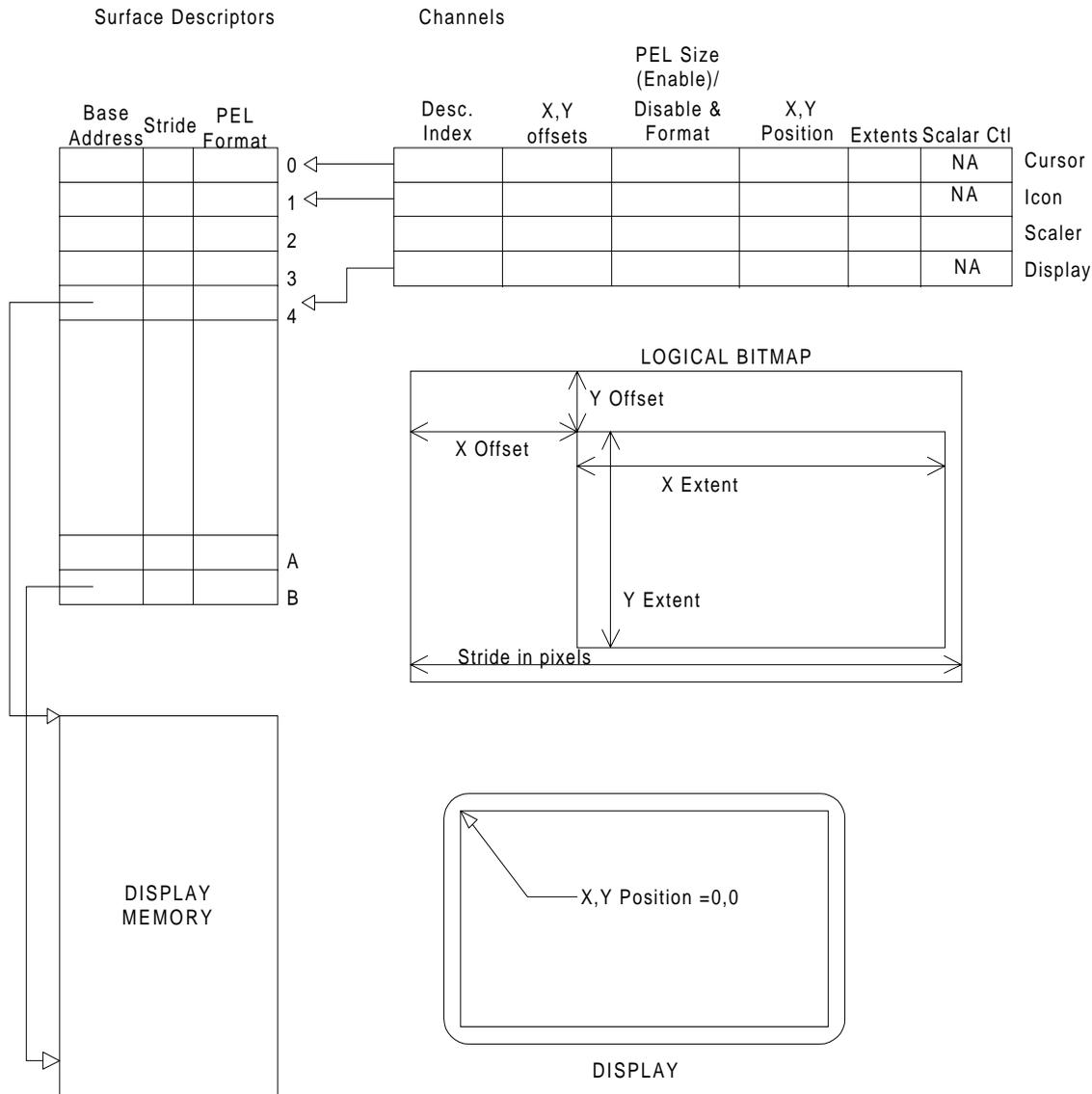


Figure 6-6. Surface Definition

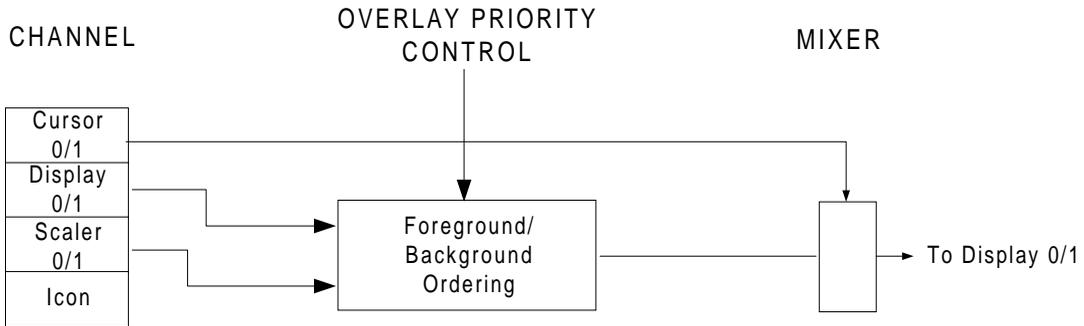


Figure 6-7. Overlay Priority and Mixing

This section provides examples to perform common display operations. These are explained as specific examples for clarity, but could be combined in real applications.

- Define surfaces.
- Channel controls - assign surface
- Channel controls - set location, size, scale factors etc.
- Enable/disable a channel.
- Set Overlay priority.
- Set Back/Fore Mix control (blend and overlay).
- Query the scanline being displayed.

In all the examples, the writes to the display control registers are deferred and written during VBLANK. Writes to these registers are recorded into the Deferred Control Buffer and replayed during VBLANK. The immediate register address with encoded byte-enables, followed by register data, are recorded. Macros LDAB, LDAW and LDAD are used to load the deferred offsets with encoded byte-enables (these macros are defined in Section 6.14.2, Macros). This eliminates the undesirable tearing effects. A surface buffer attached to a channel can be swapped with a newly rendered surface without visual artifacts. It is appropriate to modify the controls via the immediate registers if the channel is disabled for display.

6.5.1.1 Define a Surface

This routine defines a surface given the base address, the bits per pixel and the stride in pixel units. The base address should be on a doubleword boundary, and the stride should be a multiple of 16-pixel units. The surface is uninitialized. Make sure that the surface entry being defined is not already assigned to any surface.

Sample code: Define a Surface

```

;fs selector points to the register base address
;the variable bitapixel has the BPP with the stride in surfacestride. The sur-
    face number is in surfacenum (0..F) and
    surface
;base address is in the variable surfacebase.

DefineSurface    proc    near
    mov    ax,NEXTSURFACE        ;calculate the surface
    mul    ax,surfacenum        ;register address
    mov    bx,ax                ;
    mov    eax,surfacebase      ;load surface base address
    mov    fs:[bx+SurfaceBaseAddress],eax    ;set base address for given
    surface
    mov    ax,bitapixel         ;load bits per pixel
    shl    eax,20               ;move to upper bits
    mov    ax,surfacestride     ;load surface stride
    mov    fs:[bx+SurfaceStrideFormat],eax    ;set stride and format
    ret
DefineSurface    endp

```

6.5.1.2 Channel Controls - assign surface

The given surface is assigned to the channel 0,1,2 or the control channel. In a typical operation, one surface assigned to a channel is being displayed, and a second surface is being rendered into or composed. The surfaces are then flipped to display the newly rendered surface. The surface flip needs to occur during VBLANK to avoid undesirable screen artifacts. This is accomplished by recording the writes to the Display Control registers into the Deferred Control Buffer and replaying the register writes into the registers during VBLANK.

Sample code: Display Deferred control

```

;fs selector points to the register base address
;the variable surfacenum has the Surface descriptor index of surface to be
;attached. The variable channelnum has the channel
;number, one of (CHANNELCTL, CHANNEL0,CHANNEL1, CHANNEL2).
;Variable PhyCmdPortAddress has the immediate register physical base
;address, add offsets to yield register addresses
;displayreplaybase has the base address offset of Display Deferred
;Control;buffer in the offscreen memory.
;gs selector points to the physical screen base.

```

```

DisplayDeferredControl   proc   near
    mov     edi,fs:[DisplayRecordPtr]           ;load edi display write pointer
;read Display control status - disarms Deferred Display replay
    mov     al,fs:[DisplayReplayStsReg]        ;read status, disarm replay
    and     al,REPLAYPENDING+REPLAYACTIVE      ;
    cmp     al,REPLAYPENDING                   ;skip to add to Control buffer
    jz      RECORDLOADS                       ;if replay pending

```

```

@@:
    mov     al,fs:[DisplayReplayStsReg]        ;else wait while replay active
    test    al,REPLAYACTIVE                    ;
    jnz     @B                                 ;skip to record writes if inactive
    mov     edi,displayreplaybase              ;set read and write pointers
    mov     fs:[DisplayRecordPtr],edi          ;set display write base pointer
    mov     fs:[DisplayReplayPtr],edi          ;set display read base pointer

```

RECORDLOADS:

```

    sub     edi,PhyCmdPortAddress               ;edi has offset into buffer
    mov     ax,NEXTCHANNEL                      ;calculate
    mul     ax,channelnum                       ;register address offset for
                                                ;channel
    movzx   ebx,ax                              ;in ebx
    LDAB   edx,ChannelIndex                    ;
    add     edx,ebx                             ;add to base
    or      edx,PhyCmdPortAddress              ;record reg address and data into
                                                ;deferred buffer
    mov     gs:[edi],edx                        ;
    mov     al,surfacenum                       ;load surface base number, dis-
                                                ;played after next VBLANK
    mov     gs:[edi+4],al                       ;set surface number
    add     edi,(8-4)                           ;compute write pointer
    add     edi,PhyCmdPortAddress               ;
    mov     fs:[DisplayRecordPtr],edi          ;set display write pointer
    mov     fs:[DisplayReplayCtlReg],ENABLEREPLAY;Enable replay at next VBLANK
    ret
DisplayDeferredControl   endp

```

6.5.1.3 Channel Controls - Offsets & viewports location

The offsets, viewport location and the extents are set for a *display* channel. It is assumed that a surface has been previously assigned to this channel and is being displayed. Setting the offsets establishes the new origin into the surface being displayed. The viewport X,Y sets the location of the top left corner of the display viewport, and the extents sets the width and height of the display viewport. If the channel is being displayed, these parameters need to be set during VBLANK to avoid undesirable screen artifacts. This is accomplished by recording the writes to the Display Control registers into the Deferred Control Buffer and replaying the register writes during VBLANK. A *scaler* channel can be set in a similar manner along with the location and extents. This controls the stretch factor in the x and y directions.

Sample code: Channel control

```

;fs selector points to the register base address
;The variable channelnum has the channel number, one of (CHANNELCTL,
CHANNEL0,CHANNEL1, CHANNEL2).
;Variable PhyCmdPortAddress has the immediate register physical base address,
add offsets to yield register addresses
;displayreplaybase has the base address offset of Display Deferred Control
buffer in the offscreen memory.
;gs selector points to the physical screen base.
;Variables offsetx, offsety, viewx, viewy, and viewendx, viewendy contain the
offsets, viewport location starts and ends.
;The view location parameters are clipped to screen coordinates. The variable
sstride contains the stride of the surface
;sheight has the surface height.

```

```

ChannelControl  proc  near
    mov     edi,fs:[DisplayRecordPtr]      ;load edi display write pointer
;read Display control status - disarms Deferred Display replay
    mov     al,fs:[DisplayReplayStsReg]   ;read status, disarm replay
    and     al,REPLAYPENDING+REPLAYACTIVE ;
    cmp     al,REPLAYPENDING              ;skip to add to Control buffer
    jz      RECORDLOADS                   ;if replay pending

@@:
    mov     al,fs:[DisplayReplayStsReg]   ;else wait while replay active
    test    al,REPLAYACTIVE                ;
    jnz     @B                             ;skip to record writes if inactive

    mov     edi,displayreplaybase         ;set read and write pointers
    mov     fs:[DisplayRecordPtr],edi     ;set display write base pointer
    mov     fs:[DisplayReplayPtr],edi     ;set display read base pointer

```

```

RECORDLOADS:
    sub     edi,PhyCmdPortAddress          ; edi has offset into buffer
    mov     ax,NEXTCHANNEL                 ;calculate
    mul     ax,channelnum                   ;register address offset for channel
    movzx   ebx,ax                          ;
    LDAD    edx,ChViewPositionStarts      ;load Channel View base

```

```

add    edx,ebx                ;offset into desired channel
or     edx,PhyCmdPortAddress ;record reg address and data into
                                deferred buffer

mov    gs:[edi],edx          ;
mov    ax,viewy              ;load viewport x,y
shl   eax,16                 ;
mov    ax,viewx              ;
mov    gs:[edi+4],eax        ;set viewport x,y

LDAD   edx,ChViewPositionEnds ;load Channel Extent Base
add    edx,ebx                ;offset into desired channel
or     edx,PhyCmdPortAddress ;record reg address and data into
                                deferred buffer

mov    gs:[edi+8],edx        ;
mov    ax,viewyendy          ;load x,y ends
shl   eax,16                 ;
mov    ax,viewendx          ;
mov    gs:[edi+12],eax       ;set x,y ends

LDAD   edx,ChannelOffsets    ;load Channel offset Base
add    edx,ebx                ;offset into desired channel
or     edx,PhyCmdPortAddress ;record reg address and data into
                                deferred buffer

mov    gs:[edi+16],edx       ;
mov    ax,offsety            ;load x,y offsets into the surface
                                from top left corner

shl   eax,16                 ;
mov    ax,offsetx           ;
mov    gs:[edi+20],eax       ;set x,y offsets

if (Surface type YUV) {
LDAD   edx,Ch0U0ffsets       ;load Channel U offsets
or     edx,PhyCmdPortAddress ;record reg address and data into
                                deferred buffer

mov    gs:[edi+24],edx       ;
mov    ax,offsety            ;load U offsety = offsety/2+sheight
shr   ax,1                   ;
add   ax,sheight            ;
shl   eax,16                 ;
mov    ax,offsetx           ;U offsetx = offsetx/2
shr   ax,1                   ;
mov    gs:[edi+28],eax       ;set U offsets

LDAD   edx,Ch0V0ffsets       ;load Channel V offsets
or     edx,PhyCmdPortAddress ;record reg address and data into
                                deferred buffer

mov    gs:[edi+32],edx       ;
mov    ax,offsety            ;load V offsety = offsety/2+sheight
shr   ax,1                   ;
add   ax,sheight            ;
shl   eax,16                 ;

```

```
mov    ax,offsetx                ;V offsetx = (offsetx+stride)/2
add    ax,sstride                ;
shr    ax,1                      ;
mov    gs:[edi+36],eax           ;set V offsets
add    edi,(40-4)                ;compute write pointer
} else {
add    edi,(24-4)                ;compute write pointer
}
add    edi,PhyCmdPortAddress     ;
mov    fs:[DisplayRecordPtr],edi ;set display write pointer
mov    fs:[DisplayReplayCtlReg],ENABLEREPLAY ;Enable replay at next
                                           VBLANK
ret
ChannelControl    endp
```

6.5.1.4 *Enable/Disable Channel Display*

When a display channel is either enabled or disabled, the attached surface is either displayed or turned off. The disabling can be effected immediately by writing to the immediate registers. The enable needs to occur during VBLANK to avoid undesirable screen artifacts. This is accomplished by recording the writes to the Display Control registers into the Deferred Control Buffer and replaying the register writes during VBLANK. On enabling/disabling channels, the display FIFO controls have to be set.

Sample code: Enable/Disable Channel Display

```

;fs selector points to the register base address
;Variable physicalbase has the immediate register physical base address, add
                                offsets to yield register addresses
;displayreplaybase has the base address offset of Display Deferred Control
                                buffer in the offscreen memory.
;gs selector points to the physical screen base.
;The variable channelnum has the channel number, one of (CHANNELCTL,
                                CHANNEL0,CHANNEL1, CHANNEL2).
;displayformat contains the format and bitdepth of the surface being dis-
                                played, e.g. 16 BPP YUV 4.2.2.
;If the display channel is to be disabled, displayformat contains 0

```

```

EnableChannelDisplay    proc    near
    mov     bx,NEXTCHANNEL        ;calculate
    mul    ax,channelnum          ;register address offset for channel
    movzx  ebx,ax ;
    mov    cl,displayformat;
    test   cl,0ffh                ;turn on display
    jnz    @F                     ;yes: skip to turn on display
    mov    fs:[bx+ChannelFormat],cl ;no turn off channel display
    jmp    DONE
@@:
    mov    edi,fs:[DisplayRecordPtr] ;load edi display write pointer
                                ;read Display control status - disarms
                                Deferred Display replay
    mov    al,fs:[DisplayReplayStsReg];read status, disarm replay
    and    al,REPLAYPENDING+REPLAYACTIVE;
    cmp    al,REPLAYPENDING        ;skip to add to Control buffer
    jz     RECORDLOADS            ;if replay pending
@@:
    mov    al,fs:[DisplayReplayStsReg];else wait while replay active
    test   al,REPLAYACTIVE;
    jnz    @B                     ;skip to record writes if inactive

    mov    edi,displayreplaybase    ;set read and write pointers
    mov    fs:[DisplayRecordPtr],edi ;set display write base pointer
    mov    fs:[DisplayReplayPtr],edi ;set display read base pointer

```

```

RECORDLOADS:
    sub    edi,physicalbase        ; edi has offset into buffer
    LDAB  edx,ChannelFormat       ;load Channel format Base
    add   edx,ebx                 ;offset into desired channel
    or    edx,physicalbase       ;record reg address and data into
                                deferred buffer

    mov   gs:[edi],edx           ;
    mov   gs:[edi+4],cl         ;enable channel

    add   edi,(8-4)              ;compute write pointer
    add   edi,physicalbase      ;
    mov   fs:[DisplayRecordPtr],edi ;set display write pointer
    mov   fs:[DisplayReplayCtlReg],ENABLEREPLAY ;Enable replay at next
                                VBLANK DONE:

    ret

EnableChannelDisplay    endp

```

6.5.1.5 Set Display FIFO Controls

The display FIFO has 10 entries and needs to be programmed for entry allocation and thresholds for the control channel as well as display channels 1 and 2. The FIFO control register has to be reprogrammed on enabling or disabling display channels.

```

;fs selector points to the register base address
;Variable physicalbase has the immediate register physical base address, add
                                offsets to yield register addresses
;displayreplaybase has the base address offset of Display Deferred Control
                                buffer in the offscreen memory.
;gs selector points to the physical screen base.
;The variable channelnum has the channel number, one of (CHANNELCTL,
                                CHANNEL0,CHANNEL1, CHANNEL2).
;displayformat contains the format and bitdepth of the surface being dis-
                                played, e.g. 16 BPP YUV 4.2.2.
;If the display channel is to be disabled, displayformat contains 0

```

6.5.1.6 Set Channel 0 X,Y Scale Factors

Sets up X and Y scale factor registers for for channel 0. The scale factors are based on the source and destination image sizes. The scaling interpolation can be bilinear or nearest-neighbor. It is not possible to scale down below an X or a Y factor of 1/16; the scaling upper bound is 8 for YUV and 16 for RGB. The MPEG uses a bilinear interpolation scheme if the X scale factor is greater than 0.25; nearest-neighbor is used otherwise. For non-MPEG applications, bilinear interpolation is used with pre-filtering if the X scale factor is less than or equal to 0.5.

```

/* The following variables are given:
/*
Given
int srcXstride, srcYstride
int dstXstride, dstYstride, dstXoffset, dstYoffset, dstWidth, dstHeight, dstX-
left, dstYtop
(Refer to Figure 6-6 for definition of above arguments.)
Boolean MPEG (1 if source is from SM3110's MPEG decoder, otherwise 0)
Boolean YUV422(1 if source is in interleave 4:2:2 format, 0 if planar format
4:2:0)
Boolean YUV (1 if source is in YUV format, 0 if RGB)
*/

#define IncFracBits          11
#define INTG(x)              ((x)>>IncFracBits)
#define FRAC(x)              ((x)&((1<<IncFracBits)-1))

/* Clamp scaling factors to supported range */

xFactor = (double)dstXstride/srcXstride;

/* clamp scale Factor in x down to 1/16 or up to 8 for YUV or 16 for RGB */
if (xFactor < 0.0625) {
    xFactor = 0.0625;
    dstXstride = srcXstride * xFactor + 0.5;
} else {
    if ((RGB && (xFactor > 16.0)) {
        xFactor = 16.0;
        dstXstride = srcXstride * xFactor + 0.5;
    } else {
        if (YUV && (xFactor > 8.0)){
            xFactor = 16.0;
            dstXstride = srcXstride * xFactor + 0.5;
        }
    }
}

yFactor = (double)dstYstride/srcYstride;

/* clamp scale Factor in y down to 1/16 or up to 8 for YUV or 16 for RGB */
if (yFactor < 0.0625) {

```

```

yFactor = 0.0625;
dstYstride = srcYstride * yFactor + 0.5;
} else {
if ((RGB && (yFactor > 16.0)) {
    xFactor = 16.0;
    dstYstride = srcYstride * yFactor + 0.5;
} else {
if (YUV && (yFactor > 8.0)){
    yFactor = 16.0;
    dstYstride = srcYstride * yFactor + 0.5;
}
}
}

/* Adjust the source x stride if pre-scaling by hardware is to be performed.
*/

prescale = 1;
if (!MPEG && (xfactor<=.5)) {
while (xfactor<=.5) {
    xfactor *= 2.;
    prescale <<= 1;
}
}

/* We need to pre-filter in X if prefilter > 1. Create a new source bitmap,
average every 2, 4 or 8 pixels in the x direction based on prescale of 2,
4 or 8 respectively. If p0,p1,p2,p3,p4,p5,p6,p7,p8..pn are pixels in a
row. If variable prescale == 2, then the new source pixels will be pn0,
pn1, pn2... where pn0=(p0+p1)>>1, pn1=(p2+p3)>>1... If variable prefilter
== 4, then the new source pixels will be pn1, pn2, pn3... where
pn0=(p0+p1+p2+p3)>>2, pn1=(p4+p5+p6+p7)>>2. Note: If srcWidth, the width
of the bitmap is not an even multiple of prefilter, the last pixel in the
row pixel is the average of the trailing remainder of pixels (less than
prescale) . */

if (prefilter > 1) {
    srcXstride = dstXstride/xfactor+.5;
    prefilter the Bitmap
}

/* Determine interpolation mode */
if (MPEG && (xFactor<=0.25))
    interpolationMode = XNEAREST+YNEAREST;
else
    interpolationMode = XBILINEAR+YBILINEAR;

/* Compute x and y increments for Y component */
xIncY = (int)(((double)(srcXstride-1)/(dstXstride-1))*(1<<IncFracBits)+.5);
yIncY = (int)(((double)(srcYstride-1)/(dstYstride-1))*(1<<IncFracBits)+.5);

```

```

/* Find (xOffsetY, yOffsetY) from (dstXoffset, dstYoffset) */
if (dstXleft < 0 && dstXoffset < abs(dstXleft)) {
    dstXoffset = -dstXleft;
    dstWidth = dstWidth + dstXleft - dstXoffset;
}
srcXstart = dstXoffset*xIncY;          /* srcXstart in 0.21.11 format */
xOffsetY = INTG(srcXstart);           /* xOffsetY = x offset for Y component
*/
initialXfracUV = FRAC(srcXstart>>1)>>(IncFracBits-4);          /* 0.0.4 for-
mat */
..... (Compute initialYfracUV in a similar way.)

/* Compute x and y increments for UV components */
xOffsetUV = xOffsetY>>1;
srcXstrideUV = srcXstride>>1;
if (!YUV420) {
    srcYstrideUV = srcYstride;
    yOffsetUV = yOffsetY;
} else {
    srcYstrideUV = srcYstride>>1;
    yOffsetUV = yOffsetY>>1;
}
xIncUV = (int)(((double)(srcXstrideUV-xOffsetUV-1-initialXfracUV*.0625) /
    (dstXstride-dstXoffset-1))*(1<<IncFracBits)+.5);
yIncUV = (int)(((double)(srcYstrideUV-yOffsetUV-1-initialYfracUV*.0625) /
    (dstYstride-dstYoffset-1))*(1<<IncFracBits)+.5);

viewX = dstXleft + dstXoffset;
viewY = dstYtop + dstYoffset;
interpolationMode = (initialYfracUV << 16) + initialXfracUV;

```

Sample code: Set Scaler Factor

```

;fs selector points to the register base address
;Variable physicalbase has the immediate register physical base address, add
    offsets to yield register addresses
;displayreplaybase has the base address offset of Display Deferred Control
    buffer in the offscreen memory.
;gs selector points to the physical screen base.

```

```

Set Scaler Factor      proc    near
    mov     edi,fs:[DisplayRecordPtr]          ;load edi display write pointer
;read Display control status - disarms Deferred Display replay
    mov     al,fs:[DisplayReplayStsReg]       ;read status, disarm replay
    and     al,REPLAYPENDING+REPLAYACTIVE    ;
    cmp     al,REPLAYPENDING                  ;skip to add to Control buffer
    jz     RECORDLOADS                        ;if replay pending

```

```

@@:
mov     al,fs:[DisplayReplayStsReg]      ;else wait while replay active
test   al,REPLAYACTIVE                  ;
jnz    @B                                ;skip to record writes if inactive

mov     edi,displayreplaybase           ;set read and write pointers
mov     fs:[DisplayRecordPtr],edi       ;set display write base pointer
mov     fs:[DisplayReplayPtr],edi      ;set display read base pointer

RECORDLOADS:
sub     edi,physicalbase                 ; edi has offset into buffer
LDAD   edx,Ch0Controls                  ;load Scaler controls, interpolation
mode with initial fractions
or     edx,physicalbase                 ;record reg address and data into
deferred buffer

mov     gs:[edi],edx                    ;
mov     eax,interpolationMode           ;
mov     gs:[edi+4],eax                  ;

LDAD   edx,Ch0YRControls                ;load Scaler controls, Y/R increaments
or     edx,physicalbase                 ;record reg address and data into
deferred buffer

mov     gs:[edi+8],edx                  ;
mov     ax,yIncY                        ;
shl    eax,16                           ;
mov     ax,xIncY                        ;
mov     gs:[edi+12],eax                 ;

LDAD   edx,Ch0UVGBControls              ;load Scaler controls, UV/GB increaments
or     edx,physicalbase                 ;record reg address and data into
deferred buffer

mov     gs:[edi+16],edx                 ;
mov     ax,yIncUV                       ;
shl    eax,16                           ;
mov     ax,xIncUV                       ;
mov     gs:[edi+20],eax                 ;

LDAD   edx,Ch0Offsets                   ;load Scaler controls, X,Y offsets
or     edx,physicalbase                 ;record reg address and data into
deferred buffer

mov     gs:[edi+24],edx                 ;
mov     ax, yOffsetY                     ;
shl    eax,16 `                          ;
mov     ax, xOffsetY ;                   ;
mov     gs:[edi+28],eax                 ;

LDAD   edx,Ch0UOffsets                  ;load Scaler controls, X,Y offsets for U
plane
or     edx,physicalbase                 ;record reg address and data into
deferred buffer

mov     gs:[edi+32],edx                 ;

```

```

mov     ax, yOffsetY      ;
shr     ax,1              ;
add     ax,srcYstride ;
shl     eax,16            ;
mov     ax, xOffsetY     ;
shr     ax,1              ;
mov     gs:[edi+36],eax   ;

LDAD    edx,Ch0VOffsets   ;load Scaler controls, X,Y offsets for V plane
or      edx,physicalbase ;record reg address and data into deferred
                               buffer

mov     gs:[edi+40],edx   ;
mov     ax, yOffsetY     ;
shr     ax,1              ;
add     ax,srcYstride    ;
shl     eax,16            ;
mov     ax,xOffsetY     ;
add     ax,srcXstride    ;
shr     ax,1              ;
mov     gs:[edi+44],eax   ;

LDAD    edx,ChViewPositionStarts ;load Channel View base
add     edx,ebx           ;offset into desired channel
or      edx,physicalbase  ;record reg address and data into
                               deferred buffer

mov     gs:[edi+48],edx   ;
mov     ax,viewY          ;load viewport x,y
shl     eax,16            ;
mov     ax,viewX          ;
mov     gs:[edi+52],eax   ;set viewport x,y

LDAD    edx,ChViewPositionEnds ;load Channel Extent Base
add     edx,ebx           ;offset into desired channel
or      edx,physicalbase  ;record reg address and data into
                               deferred buffer

mov     gs:[edi+56],edx   ;
mov     ax,viewY          ;load x,y ends
add     ax,dstHeight      ;
shl     eax,16            ;
mov     ax,viewX          ;
add     ax,dstWidth       ;
mov     gs:[edi+60],eax   ;set x,y ends

add     edi,(64-4)        ;compute write pointer
add     edi,physicalbase  ;
mov     fs:[DisplayRecordPtr],edi ;set display write pointer
mov     fs:[DisplayReplayCtlReg],ENABLEREPLAY ;Enable replay at next
                               VBLANK

ret

Set Scaler Factor      endp

```

6.5.1.7 Set Overlay Priority

Sets overlay priority for the channel displays. The channel displays are logically designated as background and foreground for composing the resulting display. When no mix or overlays are set, the foreground appears on top of the background image. If only one channel is enabled, it is assigned as the background; the foreground is assigned a channel number that is disabled.

Sample code: Set Overlay Priority

```

;fs selector points to the register base address
;Variable physicalbase has the immediate register physical base address, add
                                offsets to yield register addresses
;displayreplaybase has the base address offset of Display Deferred Control
                                buffer in the offscreen memory.
;gs selector points to the physical screen base.
;variables forechannel and backchannel have channel numbers , one
;of ( CHANNEL0,CHANNEL1,CHANNEL2 ).

```

```

SetOverlayPriority proc near
    mov     edi,fs:[DisplayRecordPtr];load edi display write pointer
                                ;read Display control status - disarms
                                Deferred Display replay
    mov     al,fs:[DisplayReplayStsReg]      ;read status, disarm replay
    and     al,REPLAYPENDING+REPLAYACTIVE;
    cmp     al,REPLAYPENDING                ;skip to add to Control buffer
    jz     RECORDLOADS;if replay pending

```

```

@@:
    mov     al,fs:[DisplayReplayStsReg];else wait while replay active
    test    al,REPLAYACTIVE                 ;
    jnz    @B                               ;skip to record writes if inactive

    mov     edi,displayreplaybase          ;set read and write pointers
    mov     fs:[DisplayRecordPtr],edi      ;set display write base pointer
    mov     fs:[DisplayReplayPtr],edi     ;set display read base pointer

```

```

RECORDLOADS:
    sub     edi,physicalbase                ;edi has offset into buffer
    LDAD   edx,OverlayPriority              ;load Overlay priority
    or     edx,physicalbase                ;record reg address and data into
                                deferred buffer

    mov     gs:[edi],edx                    ;
    mov     al,forechannel                  ;load foreground
    sub     al,2                            ;normalize
    shl     eax,8                          ;
    mov     al,backchannel                  ;load background channel
    sub     al,2                            ;normalize
    mov     gs:[edi+4],eax                  ;enable channel

    add     edi,(8-4)                       ;compute write pointer
    add     edi,physicalbase                ;
    mov     fs:[DisplayRecordPtr],edi      ;set display write pointer

```

```
mov    fs:[DisplayReplayCtlReg],ENABLEREPLAY
                                ;Enable replay at next VBLANK
ret
SetOverlayPriority endp
```

6.5.1.8 Set Mix Controls - Blend

With the priority controls set, the surfaces attached to the background can be blended with the foreground channel in certain modes. The blending can be constant or controlled on a per-pixel basis by defining a control surface and has to be enabled. For constant blend, the blend factor is set in the Mix Control register. This determines the percentage of source and destination used for the blend. With blend enabled, if a control surface (4BPP) is defined and enabled, the blend factor is defined by the control surface bitmap on a per-pixel basis. The constant blend is used for areas that do not overlap the control surface.

Sample code: Set Mix Controls - Blend

```
;fs selector points to the register base address
;Variable physicalbase has the immediate register physical base address, add
offsets to yield register addresses
;displayreplaybase has the base address offset of Display Deferred Control
buffer in the offscreen memory.
;gs selector points to the physical screen base.
;Assume that surfaces have been previously defined and attached to channel 0
and channel 1, and priority has been set.
;This example sets a constant blend for back and fore channels and enables
blending. Blend constant is specified by the
;variable blendfactor (0 < blendfactor < 0fh). If a 4BPP blend surface has
been previously defined and attached to
;the control surface, enables blend on a per pixel basis on the overlapping
areas.
```

```
SetBlending      proc    near
    mov     edi,fs:[DisplayRecordPtr] ;load edi display write pointer
                                           ;read Display control status - disarms
                                           Deferred Display replay
    mov     al,fs:[DisplayReplayStsReg];read status, disarm replay
    and     al,REPLAYPENDING+REPLAYACTIVE;
    cmp     al,REPLAYPENDING           ;skip to add to Control buffer
    jz     RECORDLOADS                ;if replay pending
```

```
@@:
mov     al,fs:[DisplayReplayStsReg];else wait while replay active
test    al,REPLAYACTIVE           ;
jnz     @B                        ;skip to record writes if inactive

mov     edi,displayreplaybase     ;set read and write pointers
mov     fs:[DisplayRecordPtr],edi ;set display write base pointer
mov     fs:[DisplayReplayPtr],edi ;set display read base pointer

RECORDLOADS:
sub     edi,physicalbase          ;edi has offset into buffer
LDAW    edx,Dsp0MixCtl            ;load Mix control register
or      edx,physicalbase         ;record reg address and data into
                                deferred buffer

mov     gs:[edi],edx              ;
mov     ax,blendfactor            ;load blend constant
or      ax,BLENDENABLE           ;enable blend
mov     gs:[edi+4],ax             ;load back/fore mix data

add     edi,(8-4)                 ;compute write pointer
add     edi,physicalbase          ;
mov     fs:[DisplayRecordPtr],edi ;set display write pointer
mov     fs:[DisplayReplayCtlReg],ENBLEREPLAY
                                ;Enable replay at next VBLANK

ret
SetBlending     endp
```

6.5.1.9 Set Mix Controls - Color Keying

Color keying allows pixels with a specified color (the key color) on an overlapping surface to become transparent and display the pixels from the underlying surface. The key color is set and the keying enabled. If the surface being compared is a YUV surface, the chroma compare color (Ch0ChromaCmp) is set. The keying operation can be performed on background (RGB) with the foreground channel. If keying is enabled and a 1 BPP control surface is defined and enabled, the control surface specifies the keying. Pixels corresponding to a 1 bit on the control surface allow the foreground to be displayed, while a 0 bit displays pixels from the background surface.

Sample code: Set Mix Controls - Color Keying

```

;fs selector points to the register base address
;Variable physicalbase has the immediate register physical base address, add
offsets to yield register addresses
;displayreplaybase has the base address offset of Display Deferred Control
buffer in the offscreen memory.
;gs selector points to the physical screen base.
;Assume that surfaces have been previously defined and attached to channel 0
and channel 1, and priority has been set.
;This example sets key color given in variable keycolor and enables keying for
the back channels.

```

```

SetColorKeying    proc            near
    mov     edi,fs:[DisplayRecordPtr]    ;load edi display write pointer
                                                ;read Display control status -
                                                disarms Deferred Display replay

    mov     al,fs:[DisplayReplayStsReg]  ;read status, disarm replay
    and     al,REPLAYPENDING+REPLAYACTIVE
    cmp     al,REPLAYPENDING            ;skip to add to Control buffer
    jz     RECORDLOADS                 ;if replay pending

@@:
    mov     al,fs:[DisplayReplayStsReg]  ;else wait while replay active
    test    al,REPLAYACTIVE
    jnz    @B                          ;skip to record writes if inactive

    mov     edi,displayreplaybase        ;set read and write pointers
    mov     fs:[DisplayRecordPtr],edi    ;set display write base pointer
    mov     fs:[DisplayReplayPtr],edi    ;set display read base pointer

```

```

RECORDLOADS:
    sub     edi,physicalbase           ; edi has offset into buffer
    LDAD   edx,Disp0ColCmp           ;load Display0 back/fore key compare
                                           color register
    or     edx,physicalbase         ;record reg address and data into
                                           deferred buffer

    mov    gs:[edi],edx              ;
    mov    eax,keycolor              ;set key color
    mov    gs:[edi+4],eax            ;load Key compare color

    LDAW   edx,Dsp0MixCtl            ;load Mix control register
    or     edx,physicalbase         ;record reg address and data into
                                           deferred buffer

    mov    gs:[edi+8],edx            ;
    mov    ax,ENABLECOLKEY+KEYNORMAL+KEYSRCFORE
    mov    gs:[edi+12],ax            ;load back/fore mix data

    add    edi,(16-4)                ;compute write pointer
    add    edi,physicalbase          ;
    mov    fs:[DisplayRecordPtr],edi ;set display write pointer
    mov    fs:[DisplayReplayCtlReg],ENABLEREPLAY ;Enable replay at next
                                           VBLANK

    ret
SetColorKeying   endp

```

6.5.1.10 *Get Displayed Scanline*

Returns the current scanline being displayed. The first line of active display is 0 increasing in Y, resets to 0 at the end of vertical blanking.

```

;fs selector points to the register base address.

mov ax,fs:[VerticalCounter]          ;read scanline currently being displayed

```

6.5.2 2D Command and Data FIFOs

All commands, parameters and operand data for the rendering engine are communicated by the host processor through two FIFOs that are fetched, parsed and executed by the rendering engine command processor. This allows concurrence between the host processor and the rendering engine that significantly improves drawing performance.

The two FIFOs are implemented in a portion of local memory allocated by the display driver. This has several advantages:

1. The size of these buffers may be (relatively) much larger than can be implemented using conventional SRAM of latch array FIFOs. This reduces the amount of overhead that the driver must spend managing a command/data FIFO (to prevent overflows).
2. It allows much larger data images (particularly BLTs and text) to be written into the buffer. This allows a longer sequence of commands to be posted by the driver so that it can return earlier or perform another task.
3. It allows the state of a command and its operand data to be retained in local memory if some task switch occurs with minimal state-saving overhead.

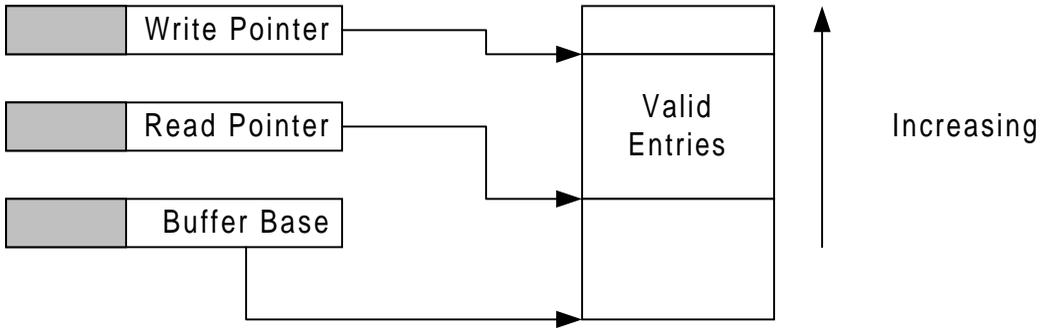
The control/parameter and image/data FIFO are implemented as circular buffers. The base addresses, write pointers and read pointers are stored memory controller registers and must be initialized by the display driver. The base address and size of each circular buffer are configurable and should be set up by writing directly to the registers before the engine is used the first time. They should not have to be changed again unless the buffer needs to be moved or resized.

The write pointer is normally updated by hardware when control information is written to the rendering engine aperture. The read pointer is incremented when this information is read by the rendering engine command processor. If the read pointer reaches the write pointer, the circular buffer is empty and fetching of control information stops pending further writes. If the write pointer reaches the read pointer, the circular buffer is full and further data will not be accepted from the host. The host bus interface may abort (if possible) to prevent locking out other independent transfers.

The driver should make every attempt to avoid having a transfer stall due to a filled condition by checking the status of the buffer if a large transfer is to be performed.

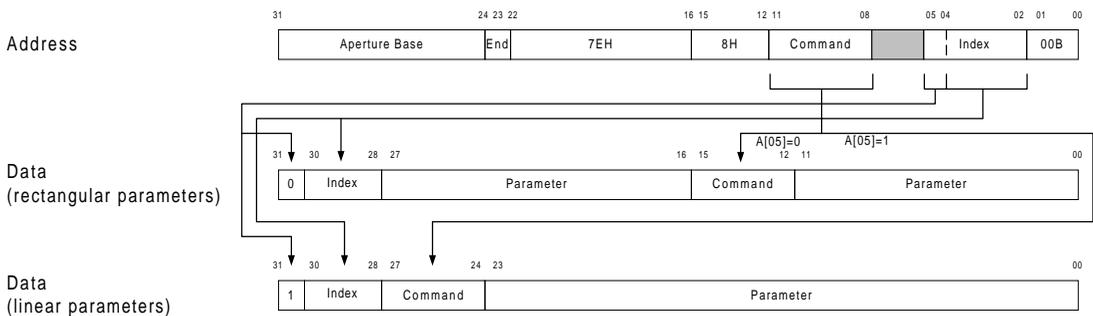
6.5.2.1 Image/Data Control Port

The 2D image data is also passed through buffers in local memory. Transfer of data images must be broken up into blocks of 64KB or less, constrained by the size of the buffer allocated. The buffer base address and buffer size must be initialized prior to use. The read and write pointers must also be initialized to zero, indicating an empty buffer. These should not have to be accessed again during normal operation.



6.5.2.2 2D Command Port

The command port is a 64KB port through which control and parameter information can be written in local memory. Additional control information is encoded in the address to reduce the number of data transfers required across the host bus. The least significant 16 bits of address are extracted, parsed and inserted into bits [31:28] and [15:12] of each parameter doubleword before it is written into the rendering buffer list. The actual address used bears no relationship to the address to which the parameter is written to local memory. If the command is other than a NOP, a doubleword entry is automatically stored in the buffer.



6.5.2.3 FIFO Status

The FIFO pointers will be updated whenever data is written by the host through the command/parameter or image data ports or read by the engine, which must have been started.

The read and write pointers may be directly read by the host to determine how full or empty a FIFO is. Note that if the host is writing to the respective FIFO or the engine is operating, the value read may be inaccurate for one of several reasons:

1. The exact value of each register may be off due to the latency in performing the read operation.
2. The accuracy of the value read may be wrong due to the latency of reading more than one byte, which may take several clock cycles. If one of the counters is updated while a read occurs, the value read from each byte may occur at slightly different times, yielding an inconsistent sample of counter contents. For example, the read from the first byte samples the contents of the entire counter as 03F0H but returns only the least significant byte as F0H; then the read of the second byte samples the contents of the entire counter as 0400H but returns the most significant byte as 04H. The bus interface will assemble these two bytes and return a value of 04F0H, which is incorrect. To eliminate this problem, it is recommended that only the most significant byte be read unless the engine is stopped. This will guarantee a maximum inaccuracy of 100H bytes, because the content of the least significant byte is not known.
3. A single bit returns the status of each FIFO indicating whether less than 1/8th of the FIFO remains unfilled. The driver may check this single bit to determine whether the exact status of the FIFO should be read to determine whether additional command should be written. Note that in normal circumstances the large size of the command/parameter FIFO means it should never get completely filled.

An additional flag byte is provided to indicate that the FIFOs are completely empty. This is useful for the image data FIFO, since this indicates the maximum amount of data that can be written.

6.5.3 2D Rendering Engine Commands

The commands and parameters are programmed via the 64KB memory-mapped control port, and any image data (for operations involving a source bitmap or pattern) is copied to a 32KB image port within 16KB address range (C000h to FFFFh). The parameter registers are at specific addresses, and alternate parameter addresses have additional control information encoded. Parameters written to these specific addresses trigger execution of commands. This mechanism eliminates the need to explicitly program a command.

The parameter addresses and the commands are defined in Section 6.14.1, Equates. The 2D programming examples use Intel assembly language style constructs. All examples assume:

```
fs = CmdPortSelector: selector points to the register base address
es = ImagePortSelector: selector points to the image data buffer base address
```

The 2D commands include:

- Wait Engine Idle, Check FIFO Size, Set Clip Rectangle
- Load Mono Pattern, Load Mono Color, Load General Pattern
- Block fill, Transparent_Text and Opaque Text
- BitBLT with 256 ROPs
- Transparent BitBLT with 256 ROPs
- Line segments

It is assumed that the controller has been set to the desired mode, i.e., the resolution and the color depth have been selected. Note that these commands are operational only in 8BPP, 16BPP or 24BPP modes. The programming is consistent for all color depths. The color information is significant in the lower order bits.

6.5.3.1 Wait Engine Idle

To ensure data integrity, direct read/write to display memory may need to wait for all rendering engine operations to finish.

Sample code: Wait Engine Idle

```
WaitEngineIdle          macro                wSelector, reg
Local      _loop
_loop:
    Mov     reg, wSelector:[Status2D]
    test   reg, BUSY2D
    jnz    _loop
endm
```

6.5.3.2 Check FIFO size

The 2D engine commands written into the Control Port addresses are stored in a command FIFO. The image data written into the Image Port addresses are stored in the image FIFO. The engine executes these commands and consumes the image data. The application can query the FIFO entries available in eighths of the total allocated size. Both the command FIFO and image FIFO size can go up to 32KB maximum. In Section 6.4.5, Physical Buffer for FIFOs, command FIFO is allocated 16KB as an example. One eighth of allocated buffer is the minimum size to be checked.

Sample code: Check FIFO size

```

WaitCMDFIFO      macro      wSelector, Eighths
Local      _loop
_loop:
    cmp     byte ptr wSelector&:[CmdFIFOStatus], Eighths
    jl     _loop
endm

WaitIMGFIFO      macro      wSelector, Eighths
Local _loop
_loop:
    cmp     byte ptr wSelector&:[ImgDataFIFOStatus], Eighths
    jl     _loop
endm
    
```

Thus, to wait for 2D command FIFO 4KB (1/4 of 16KB) empty, use this code.

```

mov     fs, CmdPortSelector
WaitCMDFIFO fs, FIFO1_4TH
    
```

To wait for 2D image FIFO 4KB (1/8 of 32KB) empty, use this code.

```

WaitIMGFIFO     fs, FIFO1_8TH
    
```

6.5.3.3 Clip Rectangle

SM3110 supports hardware clipping to eliminate the software checking whether the rendering destination's dimension is out of the defined boundary. Note that all bounding coordinates are inclusive.

Sample code: Set Clip rectangle

```

;fs= CmdPortSelector: selector points to the register base address
; lpClipRect = clip rectangle

ClipRectangle    proc    near
    mov     gClipFlag,0                ;global flag, default 0
    cmp     word ptr lpClipRect+2, 0   ;see if clip rect exist?
    je      CR_done                    ;no, exit
    lds     si,lpClipRect              ;ds:si-->clip rect

    WaitCMDFIFO  fs, FIFO1_8TH
    mov     eax,[si]                   ;left and top are inclusive
    mov     edx,[si][4]                ;right and bottom are exclusive
    sub     edx,10001h                 ;make them inclusive
    mov     fs:[ClipULXY], eax         ;Load h/w clip registers
    mov     fs:[ClipLRXY], edx
    mov     gClipFlag,F_CLIP          ;save for later use
CR_done:
    ret
Cliprectangle    endp

```

After calling this routine, remember to combine the global clipping flag 'gClipFlag' when setting the parameter "FLAGS".

6.5.3.4 Load Mono Pattern

Loads an 8x8 mono pattern into the internal pattern register. The pattern is stored internally as a color pattern with a 1 in the mono pattern replaced by the color from the Foreground color register and a 0 replaced by the color from the Background color register. This pattern will be used in subsequent Pattern Copy or BitBLT commands that involve a mono pattern.

Sample code: Load Mono Pattern

```

;fs= CmdPortSelector: selector points to the register base address
;es=ImagePortSelector: selector points to the image data buffer base address.
;ds:[esi] points to a 8 byte buffer that contains the 8x8 mono pattern
;eax has the foreground color COLOR0 and ebx has the background color COLOR1

LoadMonoPattern proc      near
    WaitCMDFIFO    fs, FIFO1_8TH
    mov     fs:[BGColor],ebx           ;load background color and load
    mov     fs:[FGColor+EXEC_LOAD_MPATTERN],eax       ;foreground color and exe-
                                                cute command
    WaitIMGFIFO    fs, FIFO1_8TH
    xor     edi,edi                   ;index into image buffer == 0
    movsd                    ;load 8 bytes of mono pattern
    movsd
    ret
LoadMonoPattern endp

```

6.5.3.5 Load Color Pattern

Loads an 8x8 color pattern into the internal pattern register. This pattern will be used in subsequent Pattern Copy or BitBLT commands that involve a color pattern.

Sample code: Load Color Pattern

```

;fs= CmdPortSelector: selector points to the register base address
;es=ImagePortSelector: selector points to the image data buffer base address.
;ds:[esi] points to a 8x8 color bitmap buffer that contains the mono pattern

LoadColorPattern proc  near
    WaitCMDFIFO    fs, FIFO1_8TH
    mov     fs:[EXEC_LOAD_CPATTERN+NOLOAD],0         ;execute command **
    mov     ecx,dwBpp                                ;8/16/24 bpp
    add     ecx,ecx                                  ;8*8*bpp/8 /4 for DWORD count=bpp*2
    WaitIMGFIFO    fs, FIFO1_8TH
    xor     edi,edi                                  ;index into image buffer == 0
    rep     movsd                                    ;load the color pattern
    ret
LoadColorPattern endp

```

NOTE: The destination X,Y for the succeeding Pattern Copy or BitBLT can be set while executing the load color pattern command.

6.5.3.6 Load General pattern

For the general case, a Load Pattern routine can be as follows:

Sample code: Load Pattern

```
;lpPattern points to a 8x8 color/mono pattern

LoadPattern      proc      near
    cmp     word ptr lpPattern+2, 0      ;see if pattern exist?
    je     LP_done                       ;no, exit
    lds    si, lpPattern                 ;ds:si-->clip rect
    cmp    ds:[si].BitsPerPixel,1      ;check if color or mono
    je     LP_mono
LP_color:
    lea    esi,ds:[si].Color            ;points to start of color pattern
    call   LoadColorPattern
    jmp    LP_done
LP_mono:
    lea    esi,ds:[si].Mono             ;points to start of mono pattern
    call   LoadMonoPattern
LP_done:
    ret
LoadPattern      endp
```

6.5.3.7 Block fill (Opaque Rectangle)

Draws a solid rectangle filled with COLOR1. The rectangle top left corner is located at X1,Y1 with height of HEIGHT1 and width WIDTH1. This command can also be used to do the BitBLT operations with ROP set to WHITENESS (255) or BLACKNESS (0).

Sample code: Block fill (Opaque Rectangle)

```

;fs= CmdPortSelector: selector points to the register base address

OpaqueRectangle proc      near
    WaitCMDFIFO fs, FIFO1_8TH
    Set2WordsToReg32WIDTH1,HEIGHT1,ax           ;eax=(W1,H1), W1 in low
                                                word
    Set2WordsToReg32X1,Y1,bx                   ;ebx=(X1,Y1), X1 in low
                                                word
    mov     edx,COLOR1                          ;the color for the fill
                                                rectangle
    mov     fs:[XYExtents],eax                  ;set height and width
    mov     fs:[DestXY],ebx                     ;load top left x,y
    mov     fs:[BGColor+EXEC_OPAQUE_RECT],edx  ;load color and execute
                                                command
    ret
OpaqueRectangle endp

```

Note: Throughout this document, the following macro is used to move the source and destination rectangle dimensions into registers.

Sample code: Set 2 WORDs to a 32-bit register

```

Set2WordsToReg32 macro Lo_word, Hi_Word, reg16
    mov     reg16,Hi_Word
    shl     e&reg,16
    mov     reg16,Lo_Word                       ;upper-left X,Y coordinate of the rect-
                                                angle(X in low word)
endm

```

6.5.3.8 Transparent Text

The monochrome bitmap that defines the text character is rendered. The image is color-expanded with pixels corresponding to a 1 colored with COLOR1, set in the foreground color register. Pixels corresponding to a 0 are unmodified. The character is rendered with the top left corner at X1,Y1. The character width and height are WIDTH1 and HEIGHT1, respectively. Each new row of the bitmap defining the text begins on the next bit. The last double word written to the image port may have to be padded .

Sample code: Transparent Text

```

;fs= CmdPortSelector: selector points to the register base address
;es=ImagePortSelector: selector points to the image data buffer base address.
;ds:[esi] points to the monochrome character data
;ecx has count double words in the monochrome image = (( (width +7)/8) *
                                     height) + 3) / 4 dwords

TransparentText  proc  near
    WaitCMDFIFO  fs, FIFO1_8TH
    Set2WordsToReg32WIDTH1,HEIGHT1,ax      ;eax=(W1,H1), W1 in low word
    Set2WordsToReg32X1,Y1,bx              ;ebx=(X1,Y1), X1 in low word
    mov  edx,COLOR1                       ;the foreground color
    mov  fs:[FGColor],edx                  ;set foreground color
    mov  fs:[XYExtents],eax                ;set height and width
    mov  fs:[DestXY+ EXEC_TRANS_TEXT],ebx ;load top left x,y & execute cmd
    WaitIMGFIFO  fs, FIFO1_8TH
    xor  edi,edi                            ;index into image buffer == 0
    rep  movsd                               ;copy mono bitmap
    ret
TransparentText  endp

```

6.5.3.9 Opaque Text

The monochrome bitmap that defines the text character is rendered. The image is color-expanded with pixels corresponding to a 1 colored with COLOR1, the foreground color. Pixels corresponding to a 0 are colored with COLOR0, the background color.

Sample code: Opaque Text

```

;fs= CmdPortSelector: selector points to the register base address
;es=ImagePortSelector: selector points to the image data buffer base address.
;ds:[esi] points to the monochrome character data
;ecx has count double words in the monochrome image = (( (width +7)/8) *
                                     height) + 3) / 4 dwords

OpaqueText proc    near
    WaitCMDFIFO    fs, FIFO1_8TH
    Set2WordsToReg32WIDTH1,HEIGHT1,ax          ;eax=(W1,H1), W1 in low word
    Set2WordsToReg32X1,Y1,bx                  ;ebx=(X1,Y1), X1 in low word
    mov     edx,COLOR1                        ;the foreground color
    mov     edi,COLOR0                        ;the background color
    mov     fs:[BGColor],edi                 ;set background color
    mov     fs:[FGColor],edx                 ;set foreground color
    mov     fs:[XYExtents],eax               ;set height and width
    mov     fs:[DestXY+ EXEC_OPAQUE_TEXT],ebx ;load top left x,y & execute command

    WaitIMGFIFO    fs, FIFO1_8TH
    xor     edi,edi                            ;index into image buffer == 0
    rep     movsd                               ;copy mono bitmap
    ret

OpaqueText endp

```

6.5.3.10 BitBLT

The BitBLT operation involves movement of rectangular blocks of data from the source to the destination on the screen. The source can be on the screen or in a system memory bitmap. If the source is from the system memory, the bitmap can be monochrome or color. The monochrome bitmaps are color-converted based on the foreground and the background colors. If the source bitmap bit is a 1, the foreground color is used to render, and if the bitmap bit is a 0, the background color is used. An 8x8 pattern can be involved in the BitBLT operation. The three operands pattern, source and destination can be combined based on the ternary raster operation specified (ROP). The ternary ROP specifies a combination of AND, OR, XOR and NOT operations between the three operands. The resulting data is rendered into the destination.

BitBLT with ROPs that do not contain source, pattern or destination operands can be rendered using the BitBLT command. The BitBLT operations with only the destination operand are similar to the BitBLT operation with pattern operand described below. All BitBLT operations that contain a pattern operand must first load the monochrome or the color pattern. The load mono pattern or the load color pattern command is used for this operation. BitBLT operations with source operand in local memory (screen) must set up drawing directions in X and Y, if the source and destination rectangles overlap. Default drawing directions for all other BitBLTs are increasing in the X and Y directions.

6.5.3.11 Pattern BLT Without Source Operand

Performs a BitBLT to the specified destination rectangle. The ROP defined for this operation contains only the pattern operand or a pattern and destination combination. The top left of the rectangle is at X1,Y1 and the rectangle has a height HEIGHT1 and a width WIDTH1. The monochrome or color pattern is previously loaded with a pattern load command. The rendered pattern is aligned with the origin 0,0. If only a destination operand is present in the ROP, no pattern is loaded.

Sample code: Pattern BLT

```

;fs= CmdPortSelector: selector points to the register base address
;variable RopValue = ROP function, contains only the pattern, or pattern and
                        destination operands

PatternBlt      proc   near
    call  ClipRectangle      ;set clip rectangle if necessary
    call  LoadPattern       ;Load the pattern to internal buffer
    WaitCMDFIFO  fs, FIFO1_8TH
    Set2WordsToReg32WIDTH1,HEIGHT1,ax      ;eax=(W1,H1), W1 in low word
    Set2WordsToReg32X1,Y1,bx              ;ebx=(X1,Y1), X1 in low word
    mov   fs:[XYExtents],eax              ;set height and width
    mov   fs:[DestXY],ebx                 ;load top left x,y & execute command
    mov   ecx,F_XP+F_YP+CMD_BITBLT      ;draw in X left to right, Y top to bot-
                                        ;tom, blt command

    or    ecx,gClipflag                  ;set clip flag
    mov   cl,RopValue                    ;move ROP into low byte
    mov   fs:[ FLAGS + EXEC_CMD],ecx     ;set flags register & execute
                                        ;command

    ret
PatternBlt      endp

```

For the pure **pattern copy** (ROP=0F0h) case without clipping rectangle, the last 5 command lines can be combined as following:

```

    mov   fs:[ DestXY + EXEC_PAT_COPY],ebx      ;load top left x,y & exe-
                                        ;cute command

```

6.5.3.12 Display Memory Source BLT with/without Pattern

Performs a BitBLT to the specified destination rectangle. The ROP defined for this operation contains a source operand or a source and destination combination. There can be an associated pattern operand in each case. The top left of the destination rectangle is at X1,Y1 and the rectangle has a height HEIGHT1 and a width WIDTH1. The source is in local display memory and the top left of the source rectangle is at X0,Y0. If a pattern is present, the monochrome or a color pattern is previously loaded with a pattern load command. The rendered pattern is aligned with the origin at 0,0. If source and destination rectangles overlap, then BLT directions need to be determined in parameter 'FLAGS'.

Sample code: Screen To Screen BLT

```

;fs= CmdPortSelector: selector points to the register base address
;variable RopValue = ROP function, contains only source, or source and destination operands

ScreenToScreenBlt    proc    near
    call    ClipRectangle;set clip rectangle if necessary
    call    LoadPattern;Load the pattern to internal buffer

    WaitCMDFIFO    fs, FIFO1_8TH
    mov     ecx,F_XP+F_YP+CMD_BITBLT+F_COLOR            ;draw X left to right, Y
                                                    top to bottom, blt
                                                    cmd,color src

;;set blt directions
    mov     ax,Y1
    cmp     ax,Y0
    jlsbd_done                    ;Y1<Y0, positive direction
    jg     sbd_set_neg            ;Y1>Y0, negative direction
    mov     ax,X1
    cmp     ax,X0
    jlesbd_done                  ;Y1=Y0, X1<=X0, positive direction
sbd_set_neg:                ;set negative direction
    orecx,F_XN+ F_YN            ;decreasing x,y
    mov     ax,HEIGHT1
    mov     dx,WIDTH1
    dec     ax                    ;HEIGHT1-1
    dec     dx                    ;WIDTH1- 1
    add     Y1,ax                ;Y1 = Y1+HEIGHT1-1
    add     Y0,ax                ;Y0 = Y0+HEIGHT1-1
    add     X1,dx                ;X1 = X1+WIDTH1- 1
    add     X0,dx                ;X0 = X0+WIDTH1- 1
sbd_done:
    or     ecx,gClipflag        ;set clip flag
    mov     cl,RopValue          ;move ROP into low byte
    mov     fs:[FLAGS],ecx      ;set flags register

    Set2WordsToReg    32WIDTH1,HEIGHT1,ax            ;eax=(W1,H1), W1 in low word
    Set2WordsToReg    32X1,Y1,bx                    ;ebx=(X1,Y1), X1 in low word
    Set2WordsToReg    32X0,Y0,dx                    ;edx=(X0,Y0), X0 in low word
    mov     fs:[XYExtents],eax                ;set height and width
    mov     fs:[SrcXY],edx                    ;load source x,y and destination
    mov     fs:[DestXY+EXEC_CMD],ebx          ;x,y & execute command
    ret

ScreenToScreenBlt    endp

```

Note:

A screen-to-screen BLT does not necessarily mean from visible area to visible area. The source and/or destination may be from offscreen memory. There are two ways to perform this type of BLT. The first is to specify a Surface Descriptor for the offscreen surface. If the surface to be BLT'ed is not the current one, set the base address, (Ax0h), pixel size (Ax4h) and surface stride (Ax4h), then set the surface descriptor (824h) to that surface index. Do a normal screen-to-screen BLT and the operation will use the specified surface.

In the second BLT method, set the source stride register (804h) to the source's stride in offscreen memory, then set the linear source address register (838h) to the source's offset in offscreen memory. Set the destination using surface descriptors as described above and the BLT using the LinBLT operation (820h[11:08]=6H) instead of the standard BitBLT

6.5.3.13 System Memory Source BLT with/without Pattern

Performs a BitBLT to the specified destination rectangle. The source is image data in system (CPU) memory. The ROP defined for this operation contains a source operand or a source and destination combination. There can be an associated pattern operand in each case. If a pattern is present, the monochrome or color pattern is previously loaded with a pattern load command. The rendered pattern is aligned with the origin at 0,0. The top left of the destination rectangle is at X1,Y1 and the rectangle has a height HEIGHT1 and a width WIDTH1. The source data can be monochrome or color. The image data is moved in one of three(3) ways: as an opaque BLT, as a monochrome transparent BLT, or as a color transparent BLT. Each of these is shown below.

Opaque BitBLT

Usually opaque operation is used with ternary ROP. With a monochrome source, the 1BPP image data is color expanded. Pixels corresponding to a 1 in the source bitmap are colored with the foreground color, and pixels corresponding to a 0 are colored with the background color.

Sample code: Image Data Source BLT

```

;fs= CmdPortSelector: selector points to the register base address
;es=ImagePortSelector: selector points to the image data buffer base address.
;variable RopValue = ROP function, contains only source, or source and destination operands

ImageDataSrcBlt proc near
    call ClipRectangle           ;set clip rectangle if necessary
    call LoadPattern             ;Load the pattern to internal buffer
    WaitCMDFIFO fs, FIFO1_8TH

    ;;set command flags portion - begin
    mov ecx,F_IMGDATA+ F_DWORD+CMD_BITBLT ;CPU to screen Blt, mono source
    lds esi,lpSrcDevice          ;point to source device
    cmp ds:[esi].BitsPerPixel,1 ;check if color or mono?
    Jne SB_src_color

SB_src_mono:
    Mov eax,COLOR1              ;load foreground color
    
```

```

Mov     ebx,COLOR0                ;load background color
mov     fs:[FGColor],eax
mov     fs:[BGColor],ebx
jmp     SB_check_src_done
SB_src_color:
or      ecx,F_COLOR+ F_TCOPAQUE   ;source is color bitmap, no transparency
SB_check_src_done:
or      ecx,gClipflag             ;set clip flag
mov     cl,RopValue               ;move ROP into low byte
mov     fs:[FLAGS],ecx           ;set flags register
;;set command flags portion - en

Set2WordsToReg32WIDTH1,HEIGHT1,ax ;eax=(W1,H1), W1 in low word
Set2WordsToReg32    X1,Y1,bx      ;ebx=(X1,Y1), X1 in low word
mov     fs:[XYExtents],eax        ;set height and width and destination
mov     fs:[DestXY+EXEC_CMD],ebx ;top left x,y & execute command

;calculate the number of dwords to be transferred= ((width*bpp + 31) / 32) *
; height dwords
movzx  eax,ds:[esi].BitsPerPixel
mul    WIDTH1
add    eax,31
shr    eax,5
;note that each row is padded so new rows begin on a fresh dword.  If the image
;is color then F_COLOR bit of the FLAGS has to be set
;In F_DWORD mode, new rows use the next dword of source data, rows ends may
;have to be padded
;if the image is larger then 64K, the transfer is performed in increments of
;the image buffer size

mov     edx,ds:[esi].WidthBytes   ;src rect's bytes per scan line
lea    esi,ds:[esi].BmpData      ;pointer to image data in CPU
mov     ebx,HEIGHT1
SB_loop:
mov     ecx,eax                  ;transfer dword count
Push   esi
WaitIMGFIFO  fs, FIF01_8TH
xor     edi,edi
rep    movsd                     ;transfer the image
pop    esi
add    esi,edx                   ;advance to next line of src data
dec    ebx
jnz    SB_loop
ret
ImageDataSrcBlt  endp

```

Monochrome Transparent BitBLT

To inhibit rendering of pixels corresponding to a 0 bit or a 1 bit in a monochrome source bitmap, set the Mono Source BLT Transparency bits in the FLAGS register to the desired configuration. The following example transfers a mono rectangular image to screen. The monochrome image is color expanded. Pixels corresponding to a 1 in the source bitmap are colored with the foreground color. Pixels corresponding to a 0 are unmodified.

In the “set command flags portion” of sample code above, modify to the following

```
mov     ecx,F_IMGDATA+ F_MBT+F_DWORD+CMD_BITBLT
                                           ;CPU to screen Blt, mono source,
                                           Transparent background

mov     eax,COLOR1
mov     fs:[FGColor],eax                 ;load foreground color
or      ecx,gClipflag                    ;set clip flag
mov     cl,RopValue                      ;move ROP into low byte
mov     fs:[FLAGS],ecx                   ;set flags register
```

Note: By setting the F_MFT bits in the FLAGS register instead of the F_MBT bits, background color is rendered for pixels corresponding to a 0 bit in the source bitmap. The pixels corresponding to the 1 bit in the source bitmap are unmodified. Selecting the F_MNORMAL mode in the FLAGS register performs a straight color expansion, with 0's and 1's colored with background and foreground respectively. The F_MINVERT performs an inverted color expansion, with 1's expanded to background color and 0's expanded to foreground color.

Color Transparent BitBLT

Transparent BitBLT commands are similar to normal BitBLT commands. With a color bitmap as a source, it is possible to inhibit rendering pixels of a given source color. With this configuration selected in the flags register, if source pixels are the same as the compare color set in the background color register, corresponding pixels in the destination are not rendered. The flags can also be set so updates occur only when the color matches. The following example performs a color Transparent BitBLT. The source color bitmap is in the CPU system memory. The compare color is COLOR_COMPARE and has to be loaded into the background color register. The destination is not updated for pixels when source pixel color is equal to COLOR_COMPARE. The destination rectangle is at X1,Y1 with a height HEIGHT1 and width WIDTH1. The ROP is SOURCECOPY.

In the “set command flags portion” of sample code above, modify to the following

```
mov     eax,F_IMGDATA+ F_COLOR+F_DWORD+F_TCSRC+CMD_BITBLT
                                               ;CPU to screen transparent Blt, color
                                               source,Transparent color source

mov     eax,COLOR_COMPARE
mov     fs:[TRANSColor],eax                ;load compare color
or      ecx,gClipflag                      ;set clip flag
mov     cl,0CCh                            ;move SOURCECOPY ROP into low byte
mov     fs:[FLAGS],ecx                    ;set flags register
```

Note: The transparency function can be applied to the destination by setting F_TCDEST in the flags register instead of the F_TCSRC bits. Destination pixels are unmodified if the destination pixel color equals COLOR_COMPARE the compare color as set in the background color register.

Transparent pattern BLTs can also be performed for BitBLTs involving patterns by selecting F_TCPAT bits in the flags register. The pattern has to be previously loaded. Destination pixels are not updated if the pattern pixel color equals COLOR_COMPARE, the compare color. In summary, color transparent flags can set to F_TCSRC, F_TCDEST or F_TCPAT in the FLAGS register for Source, Destination or Pattern transparency, respectively. Additionally, the F_TBINVERT flag can be set so the sense of the transparency can be inverted. The BitBLT destination is updated only when a color match occurs. Transparent BLTs can also be performed with local display memory (screen to screen) BitBLTs.

6.5.3.14 Line Draw

The SM3110 controller does not have an explicit line draw command. It does provide commands to draw horizontal or vertical intercepts which can be used to draw lines between arbitrary X,Y coordinate pairs. The line drawing can be classified into 8 direction types (eight octants) based on the drawing direction. A single command draws a horizontal line intercept and can position the destination X,Y to the next pixel in the previous or next row, if drawing in X major ($|x_2-x_1| > |y_2-y_1|$). A series of these commands can be used to draw lines which are X major, with X increasing or X decreasing. If the lines are Y major ($|y_2-y_1| > |x_2-x_1|$), the line can be constructed by drawing a series of vertical line intercepts. The destination X,Y after drawing of each vertical intercept can be positioned to the next row (up or down), and to the previous or next column by setting appropriate FLAGS. For efficiency, treat horizontal and vertical lines as special cases.

Draw X Major line

The example below describes a line drawn in the first octant. The start point is X1,Y1 and the end point is X2,Y2 with $|X_2-X_1| > |Y_2-Y_1|$ and $X_2 > X_1$ and $Y_2 < Y_1$.

Algorithm:

```
x = X1;
dx = X2 -X1; dy = Y1 - Y2;
error = 2 dy - dx;
intercept = 0x10000; /* preset y extent = 1, indicates X major */

while (x < X2)
{
    intercept += 1;
    if (error <= 0)
    {
        error += 2 dy;
    } else
    {
        error += 2 * (dy - dx);
        draw_pixels(intercept);
        intercept = 0x10000; /* preset y extent = 1, indicates X major */
    }
    x += 1;
}
if (intercept != 0x10000)
    draw_pixels(intercept);
```

Sample code: Draw X Major line

```

;fs= CmdPortSelector: selector points to the register base address
DwarXMajorLine  proc  near
    call  ClipRectangle                ;set clip rectangle if necessary
    WaitCMDFIFO  fs, FIFO1_8TH
    mov     ecx,F_XP+F_YN+F_CLIP+CMD_LINE    ;Clip on, x positive,
                                           ;y negative, horizontal line

    or     ecx,gClipflag                ;set clip flag
    mov     cl,RopValue                  ;move ROP into low byte
    mov     fs:[FLAGS],ecx                ;set flags register
    mov     eax, COLOR1                  ;the draw color
    Set2WordsToReg32X1,Y1,bx             ;ebx=(X1,Y1), X1 in low word
    mov     fs:[FGColor],eax              ;load color
    mov     fs:[DestXY],ebx               ;set start x,y
    mov     eax,10000h                    ;intercept, preset y extent = 1, indi-
                                           ;cates X major

    mov     si,X1                          ;x = X1
    mov     dx,X2
    sub     dx,si                          ;dx=X2-X1
    mov     bx,Y1
    sub     bx,Y2                          ;dy=Y1-Y2
    add     bx,bx
    mov     di,bx                          ;2*dy
    sub     bx,dx
    mov     cx,bx                          ;error = 2*dy-dx
    sub     bx,dx                          ;2 * (dy-dx)

DXL_loop:
    cmp     si,X2                          ;while (x < X2)
    jge     DXL_do_remaining                ;{
    inc     eax                               ; intercept += 1;
    cmp     cx,0                               ; if (error <= 0)
    jg     DXL_greter                          ; {
    add     cx,di                               ; error += 2 dy;
    jmp     DXL_compare_done                    ; } else
DXL_greter:
    add     cx,bx ; error += 2 * (dy - dx);
    mov     fs:[XYExtents+EXEC_CMD],eax        ; draw_pixels(intercept);
    mov     eax,10000h                          ; intercept=0x10000
    ; //reset intercept

DXL_compare_done:
    inc     si                               ; x += 1;
    jmp     DXL_loop                            ;}

DXL_do_remaining:
    cmp     eax,10000h
    je     DXL_done                            ; if (intercept != 0x10000)
    mov     fs:[XYExtents+EXEC_CMD],eax        ; draw_pixels(intercept);
DXL_done:
    ret
DwarXMajorLine  endp

```

Note: The last pixel is never drawn. Each new line has to load a start X,Y.

Draw Y Major line

The example below describes a line drawn in the second octant. The start point is X1,Y1 and the end point is X2,Y2 with $|X2-X1| < |Y2-Y1|$ and $X2 > X1$ and $Y2 < Y1$.

Algorithm:

```
y = Y1;
dx = X2 -X1; dy = Y1 - Y2;
error = 2 dx - dy;
intercept = 1;          /* preset x extent = 1, indicates Y major */

while (y > Y2)
{
    intercept += 0x10000;
    if (error <= 0)
    {
        error += 2 dx;
    } else
    {
        error += 2 * (dx - dy);
        draw_pixels(intercept);
        intercept = 1; /* preset x extent = 1, indicates Y major */
    }
    y -= 1;
}
if (intercept <> 1)
    draw_pixels(intercept);
```

Sample code: Draw Y Major line

```

;fs= CmdPortSelector: selector points to the register base address

DwarYMajorLine  proc  near
    call  ClipRectangle                ;set clip rectangle if necessary
    WaitCMDFIFO fs, FIFO1_8TH
    mov   ecx,F_XP+F_YN+ CMD_LINE     ;Clip on, x positive, y negative
    or    ecx,gClipflag                ;set clip flag
    mov   cl,RopValue                  ;move ROP into low byte
    mov   fs:[FLAGS],ecx               ;set flags register
    mov   eax, COLOR1                  ;the draw color
    Set2WordsToReg32X1,Y1,bx          ;ebx=(X1,Y1), X1 in low word
    mov   fs:[FGColor],eax             ;load color
    mov   fs:[DestXY],ebx              ;set start x,y
    mov   eax,1                        ;intercept, preset x extent = 1, indicates Y major
    mov   si,Y1                        ;y = Y1
    mov   dx,si                         ;
    sub   dx,Y2                        ;dy=Y1-Y2
    mov   bx,X2                         ;
    sub   bx,X1                         ;dx=X2-X1
    add   bx,bx                         ;
    mov   di,bx                        ;2*dx
    sub   bx,dx                         ;
    mov   cx,bx                        ;error = 2*dx-dy
    sub   bx,dx                        ;2 * (dx-dy)
DYL_loop:
    cmp   si,Y2                        ;while (y > Y2)
    jle   DYL_do_remaining              ;{
    inc   eax                           ;    intercept += 1;
    cmp   cx,0                          ;    if (error <= 0)
    jg    DYL_greter                    ;    {
    add   cx,di                          ;    error += 2 dx;
    jmp   DYL_compare_done              ;    } else
DYL_greter:                             ;    {
    add   cx,bx                          ;                error += 2 * (dx - dy);
    mov   fs:[XYExtents+EXEC_CMD],eax    ;                draw_pixels(intercept);
    mov   eax,1                          ;
intercept=1                             ;//reset intercept
DYL_compare_done:                       ;    }
    dec   si                             ;    y -= 1;
    jmp   DYL_loop                       ;}
DYL_do_remaining:
    cmp   eax,1
    je    DYL_done                       ; if (intercept != 1)
    mov   fs:[XYExtents+EXEC_CMD],eax    ;
draw_pixels(intercept);
DYL_done:
    ret
DwarYMajorLine      endp

```

Note: The last pixel is never drawn. Each new line has to load start X,Y.

6.6 3D Functions

6.6.1 Overview

A 3D graphics pipeline contains two partitions, the geometry pipeline and the rasterization pipeline (see Figure 6-8, below). The determination of feature set and the trade-off between performance and design cost are made based on the requirements for the target market, and the computational and I/O bandwidth requirement analysis. The analysis assures the load-balancing between a given CPU and the graphic processor, and the performance-balancing between the rendering engine and I/O devices including bus and memory. Based on this analysis, the host CPU performs the geometry portion and the SM3110's 3D engine accelerates the rasterization pipeline as shown in Figure 6-8, below.

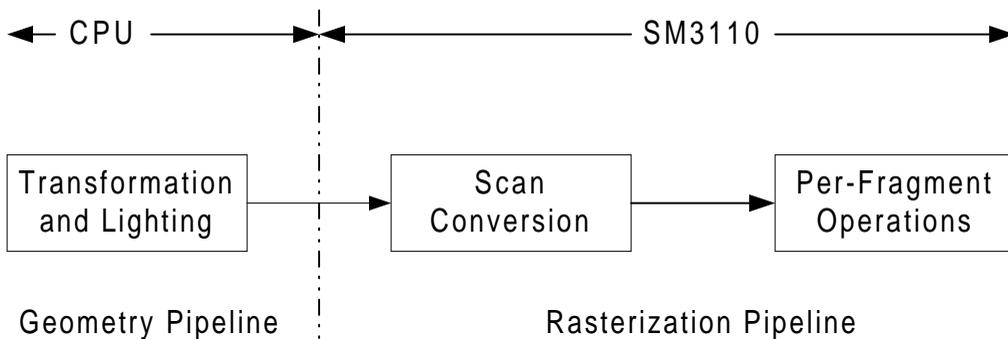


Figure 6-8. 3D Functions Partition

6.6.1.1 Feature Set

The SM3110 3D rendering engine provides high-performance rendering of 3D primitives with a complete feature set of rendering functions:

- Primitives
- Triangle (including setup and sub-pixel displacement)
- Point
- Texturing
- On-chip texture memory
- Perspective correction
- Pixel-level mipmapping
- Filtering: nearest-neighbor, bilinear, trilinear
- Texture transparency
- Texture blending
- Flat and smooth (Gouraud) shading
- Fogging
- Z buffering
- Scissoring
- Stippling
- Alpha blending
- Dithering
- Logic operations
- Double buffering

6.6.1.2 3D Rendering Engine

The 3D driver interfaces with the 3D engine by way of a 3D command FIFO in the frame buffer memory, which allows decoupling of driver and rendering functions. Figure 6-9 shows the block diagram of the 3D engine.

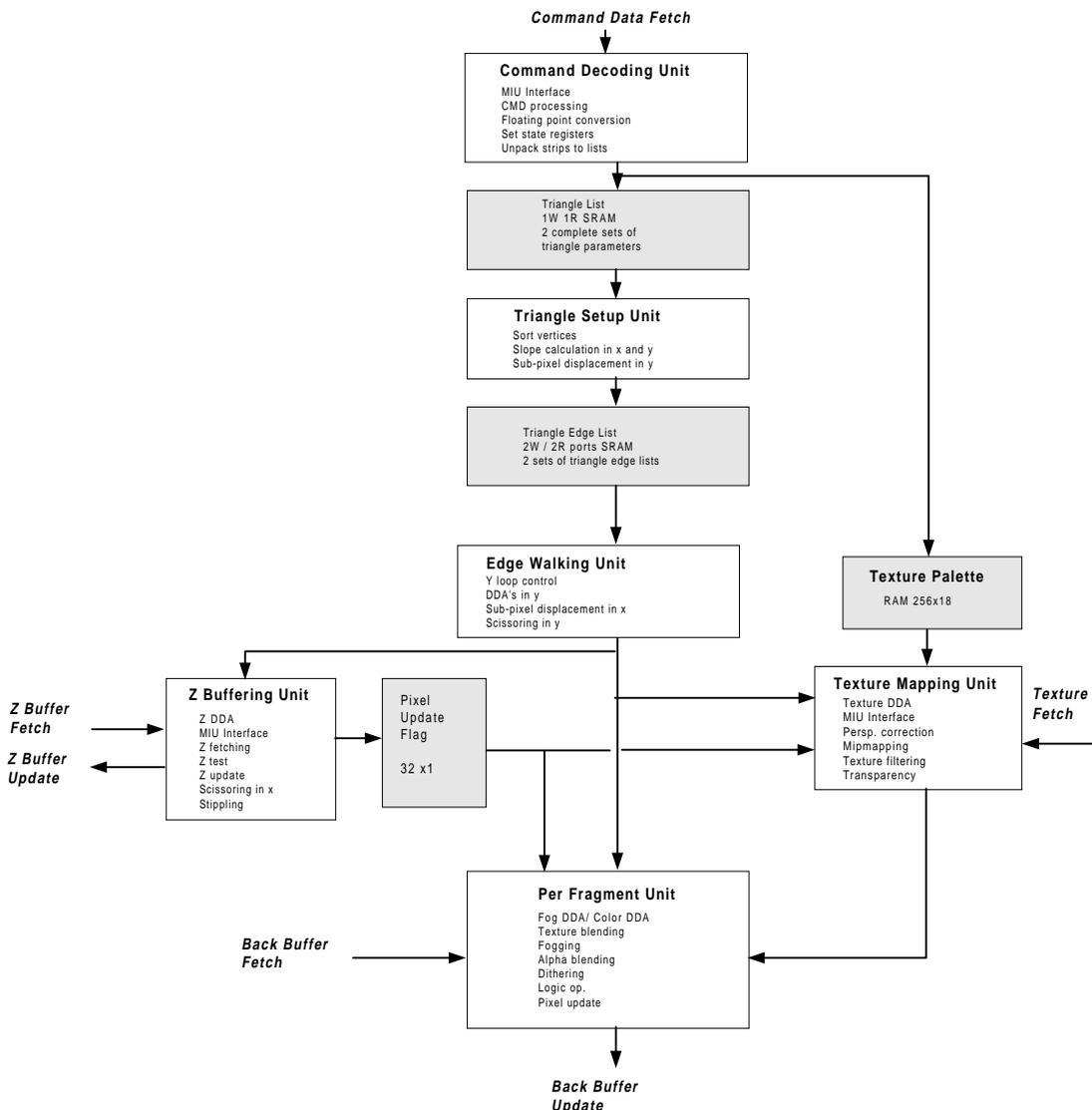


Figure 6-9. 3D Engine block Diagram

6.6.1.3 3D Rasterization Pipeline

The figure below shows functions within the 3D rasterization pipeline.

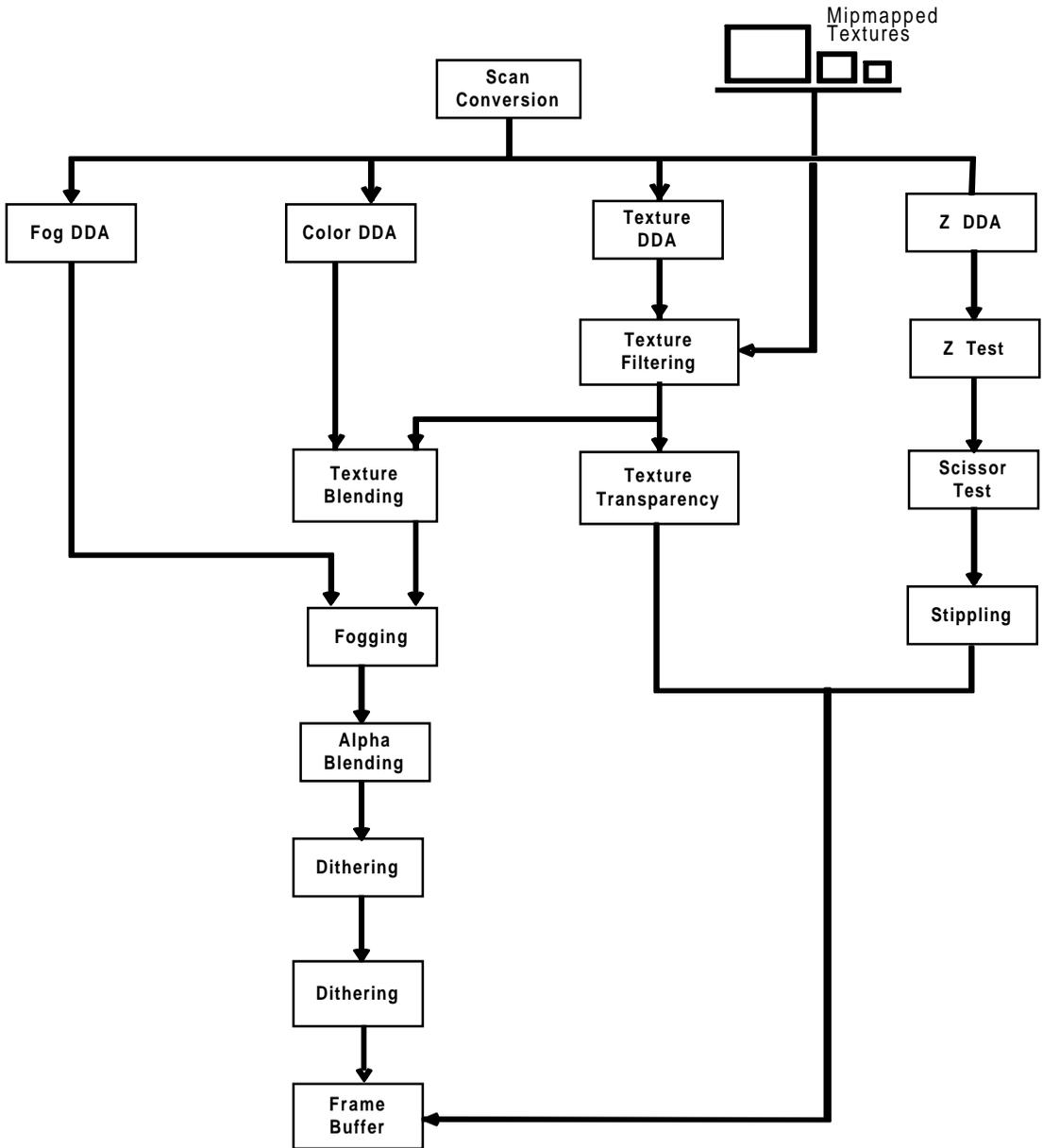


Figure 6-10. 3D Rasterization Pipeline

The order of rendering functions is critical to rendering accuracy and the diagram below represents the 3D pipeline.

<i>scisy</i>	ztst	scisx	stip	<u>tex</u>	<u>tfil</u>	<u>ckey</u>	<u>zwrt</u>	tmod	spec	fog	clmp	blnd	<u>dith</u>	<u>rop</u>
--------------	------	-------	------	------------	-------------	-------------	-------------	------	------	-----	------	------	-------------	------------

where:

italics – means pixel may be rejected at this step

underline – means only do this step if previous tests pass

stip – stipple test

ztst – z test

tex – texture mapping

tfil – texture filtering

tmod – texture modulation with diffuse color

spec – specular color

fog – apply fog

clmp – clamp results

blnd – blend source with destination

dith – apply dithering pattern

rop – apply raster operation

zwrt – write z value to z-buffer (if enabled)

scisy, scisx – scissor in y and x

ckey – color key

(Refer to the rasterization pipeline diagram above.)

1. Scissoring in Y

Failed test = reject pixel (updating interpolants).

2. Perform z-buffer test (only set pass/fail).

3. Scissoring in X (only set pass/fail).

4. Perform stipple test (only set pass/fail).

5. If previous pixel tests pass, do texture mapping, repeat for as many samples as is required by step six.
 - Look up RGB texture value from q, u, v corresponding to x, y.
 - If the texture has an alpha channel, read the A value, otherwise set A to 1.0
6. If previous pixel tests pass, perform bi- or tri-linear filtering operation on RGBA values.
7. If previous pixel tests pass, do alpha related visibility tests.
 - If chroma key transparency is enabled and the RGBA texture color lies inclusively within the chroma key range, set pass/fail for this pixel.
 - If 'alpha test' enabled, set pass/fail for pixel, based on alpha value non-zero/0
8. If previous pixel tests pass, write z value to z-buffer if z-write is enabled.
9. Modulate (see blending in a following step) texture color with diffuse color to get srcColor. If MODULATE_ALPHA is set, do the same for the alpha channel as well. Update color interpolants after this step.
10. Add specular color to srcColor. The alpha channel is not affected by this stage. This stage may produce results greater than 1.0. Update specular color interpolants.
11. Apply RGB fog. This stage may produce results greater than 1.0
12. Clamp colors to range [0.0,1.0).
13. Alpha blend srcColor with destColor to get finalColor.
14. Convert finalColor to destination pixel format, applying dither pattern.
15. Apply raster operation.
newDest = src Rop dest.
16. Write final color to destination at x, y.

6.6.2 3D Engine Operation

6.6.2.1 Initialization

All of the initialization of the SM3110 3D engine is done by the driver. The reason for this is partly because the BIOS doesn't have access to the 3D register space and partly because the driver is more flexible for handling different initialization conditions.

Prior to activation of the 3D driver, the 2D driver has already set up 32-bit linear pointers to the 3D register space, 3D command FIFO port and frame buffer memory. Refer to the 2D section for more information on this procedure. 3D registers are loaded by either using the memory-mapped byte address (minimum 16-bit reads/writes) or via the register load command in the 3D command FIFO (see that section for more details).

On power up, most registers in the SM3110 have a predefined initial state. Refer to the SM3110 register section for more information. The driver can always return the hardware to this initial state by toggling the reset bit in the Command Decode Unit (CDU) control register (0x0038).

Initialization consists of setting up certain of the state registers prior to do 3D rendering. Note that as part of context creation, the driver will initialize virtually all the registers, but the registers below are set up as part of driver initialization, prior to any 3D contexts being created. (Register addresses below are offsets within the 3D control register space of the SM3110 local address space. See the Section 4.2.2, 3D Control, for more details.)

The driver initializes the following registers:

CDU control	+038H	set to stop while initializing other regs, then start
Cmd FIFO base	+198H	internal memory start of command FIFO, 16KB aligned
Cmd FIFO size	+1C8H	size of command FIFO in bytes, 16KB aligned
Cmd FIFO write ptr	+1C0H	set to 0
Cmd FIFO read ptr	+1C4H	set to 0
Texel base	+0E6H	set to 0 (see texture loading for how texture addresses set up)
Palette index offset	+0E4H	set to 0
Texture loading base	+1E8H	set to 0
Texture loading end	+1ECH	set to 0
DRAM control	+1B9H	set according to DRAM parameters (but note that texture cache invalidate bit is used during rendering)
Internal memory timing	+1BAH	set according to DRAM parameters
Address mapping	+1A0H 1A7H	allows memory (in 2MB increments) to be marked as internal/external (n/a for SM3110) or relocated. driver currently direct maps memory.

For context creation, that is, prior to doing 3D rendering, it will be necessary to set up all the registers associated with the desired rendering mode. See the SM3110 register section for more information on the meaning and syntax of these registers. These consist of the following:

Registers applicable to 3D rendering context (offset within 3D Control region)

Modes		
Vertex Format	+000H	defines primitives to be rendered
Control	+0B8H	enables for scissors, texture, perspective correction, mipmapping, subpixel displacement, specular, fog and flat shading
Texture		
Texture Format	+0E0H	format of texture
Texture Map Size	+0E2H	\log_2 of texture height and width
Texture Wrap	+0F4H	texture wrap mode (clamp, repeat, mirror)
Texture Interpolation	+0F8H	alpha test enable, interpolation in uv space (nearest neighbor, bilinear) and between mipmap levels (nearest neighbor, linear)
Texture Transparency Color	+0C0H +0C4H	range of colors for texture transparency
Texture Transparency Enable	+134H	enables texture transparency
Texture LOD Table	+580H - +5A8H	see Texture Loading section for details
Texture Blend	+150H	enable, alpha and color blend modes
Texture Palette	+C00H- +FFCH	256 entry texture palette
Visibility		
Viewport Limits	+080H +082H +100H +102H	viewport top, bottom, left and right edges
Z Buffering	+138H	enable, write enable and z test function
Stipple	+120H	enable and x,y pattern offsets
Stipple Mask	+400H - +47CH	32x32 bit stipple mask
Buffers		
Z Buffer Stride	+130H	stride of z buffer
Back Buffer Stride	+170H	write enable and stride
Pixel Format	+174H	format of back buffer
Back Buffer Base Address	+180H	back buffer address relative to frame buffer start
Z-Buffer Base Address	+188H	Z-buffer address relative to frame buffer start
Pixel Characteristics		
Fog Color	+158H	RGB fog color
Alpha Blend	+15CH	enable, source and destination blend controls
Dither Enable	+160H	enables dither
Logic Op	+164H	enable and logic op

6.6.2.2 3D Command FIFO

The primary interface of the driver with the 3D hardware is through the 3D command FIFO. This includes loading of state registers, primitive commands, synchronization and DMA. Using the command FIFO serializes processing of commands and allows for decoupling of driver and rendering processing. The actual command FIFO is allocated in frame buffer memory (see Initialization), but is written to via a port, which is a fixed offset relative to the Control Address Space of the SM3110. Any commands written to the 4KB 3D command FIFO port are added to the next available location in the 3D command FIFO. Thus, all the FIFO management is handled by the hardware. A diagram of the operation of the 3D command FIFO is shown in the figure below.

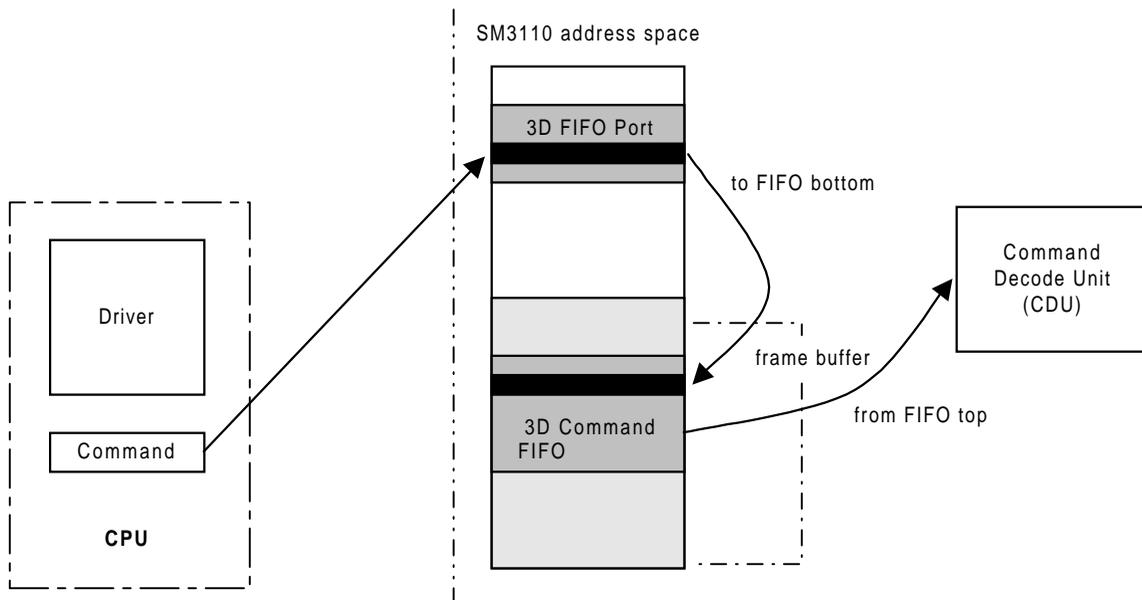


Figure 6-11. 3D Command FIFO

The driver only needs to ensure that sufficient FIFO entries are available prior to writing. This is done as shown in the following code sample:

```
// Read fast status register
regVal = ReadRegister(0x1F0);
// now compute remaining DWORDS in FIFO
remDWORDS = (FIFOSIZE - regVal.fifoStatus + 255)/sizeof(DWORD)
```

where:

FIFOSIZE – total size of 3D command FIFO in bytes

255 – is added because the read is only accurate to 256 byte units

It is recommended that the reads to the FIFO status be minimized by keeping track of the available DWORDS in a local variable and only reading again when this value is depleted.

When writing to the 3D command FIFO port, normal considerations regarding optimal PCI burst lengths should be kept in mind. That is, it may be more efficient to buffer a series of commands in CPU memory and then write them all at once.

It should also be noted that the SM3110 hardware reads the FIFO on 128-bit (4 DWORD) boundaries, which means that alignment must be kept in mind to ensure that the FIFO is able to be emptied, for example, on frame boundaries. In order to do this, 4 NOP command DWORDs should be written to the FIFO at end of frame. This will ensure that no non-NOP commands are left in the FIFO (it doesn't matter if NOP commands are left there).

6.6.3 3D Command Format

The command/parameter format consists of a sequence of 32-bit doublewords:

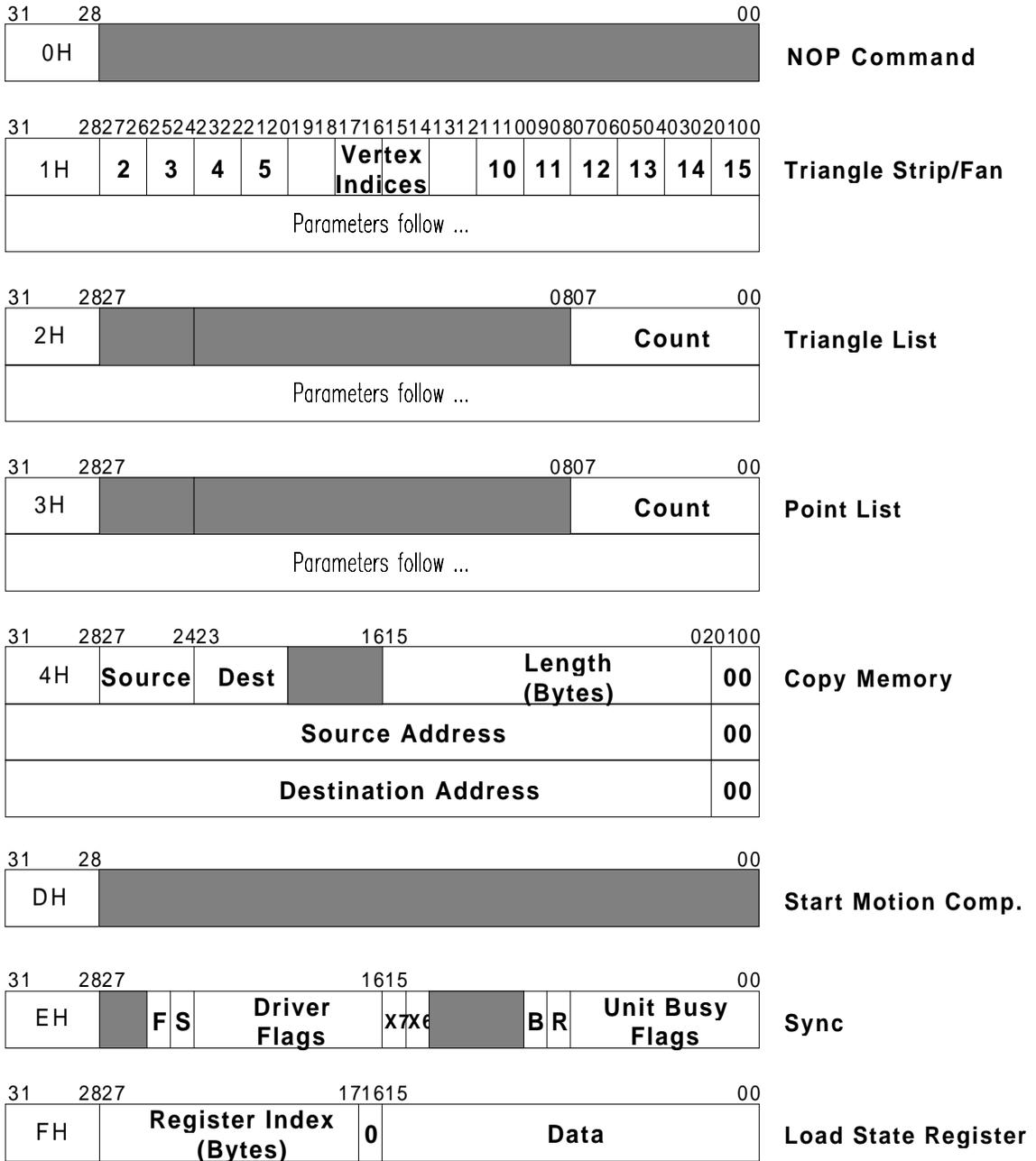


Figure 6-12. 3D Command Formats

31:28 Command	
Value	Semantics
0H	NOP.
1H	Triangle Strip/Fan.
2H	Triangle List.
3H	Point List.
4H	Copy Memory.
5-CH	<i>Reserved.</i>
DH	Start Motion Compensation Command
EH	Control/Sync/control field in immediate data field.
FH	Load State Register.

6.6.4 3D Primitive Commands

The SM3110 handles 3 types of primitives: triangle lists, triangle strips/fans and point lists. Lines are not directly supported but may be constructed in software using triangles. One significant consideration is that the SM3110 requires the extent of primitives to be less than 127 in x and y. Primitives which exceed this limit, must be subdivided in software. Also, backface culling and clipping must be done in software, as this is not done in the SM3110. Vertex order is not important.

All parameters/coordinates are represented in floating point using IEEE single precision format (1.8.23). An optional fixed point (11.21) format is also provided. All color/pixel data is represented as integer 8.8.8.8.

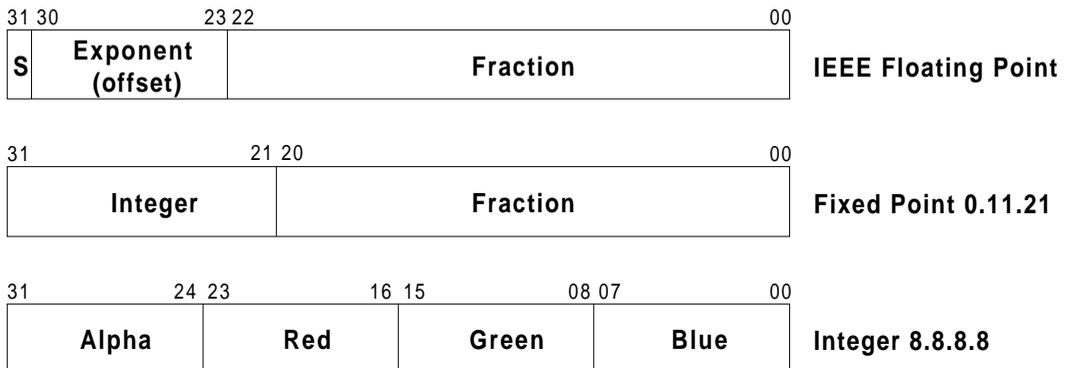


Figure 6-13. Parameter Formats (in Memory)

6.6.4.1 Triangle Strip/Fan

Up to 15 triangles may be specified in one triangle strip/fan list. The last valid vertex is indicated by specifying 0 in the successive vertex number field.

Field	Function
31:28	Command. 1H . The first triangle is implied and specified by the first three vertices.
	Vertex Index. Specifies which vertex should be replaced in a triangle strip or fan for the
27:26	2nd
25:24	3rd
23:22	4th
21:20	5th
19:18	6th
17:16	7th
15:14	8th
13:12	9th
11:10	10th
09:08	11th
07:06	12th
05:04	13th
03:02	14th
01:00	15th triangle .

Value	Semantics
1-3	Vertex Number.
0	End.

+4	1st Vertex of 1st triangle.
	2nd Vertex of 1st triangle.
	3rd Vertex of 1st triangle.
	Successive vertices.

The offset of the first parameter of each successive vertex is

$$(3+n- 2) \times \text{Size}_{\text{VERTEX_PARAMETER_LIST}} + 4 ,$$

where n is the triangle number (starting from 1).

6.6.4.2 Triangle List

Field	Function
31:28	Command. 2H .
27:08	Reserved.
07:00	Number of Triangles.

6.6.4.3 Point List

Field	Function
31:28	Command. 3H .
27:08	Reserved.
07:00	Number of Points.

Vertices can contain different numbers of elements based on the setting of the bits in the vertex format register (see 4.2.2.1, Vertex Format). The table below shows valid combinations:

	texture type	fog	specular	coordinate	color
x				xy,xyz	
y				xy,xyz	
z				xyz	
color					rgb,rgba
specular		fog	specular rgb		
u	uv,uvw,uvw,uvw,mip				
v	uv,uvw,uvw,uvw,mip				
u/w	uvw,uvw,uvw,mip				
v/w	uvw,uvw,uvw,mip				
rhw	uvw,uvw,uvw,mip				

Some special considerations should be made with regard to some of the primitive characteristics. These are noted below:

flat shading – SM3110 implements this by assuming that the color and alpha are the same in all 3 vertices and then not interpolating between them.

fog – the fog values in the vertices are treated as though a value of 0x00 is no fog and a value of 0xFF is fog color only

z scaling – the range of z values expected by the hardware is from 0 to 65535.99999.

alpha test – the SM3110 doesn't support a full alpha test, but it does support the alpha != zero case by setting the bit in the texture interpolation control register (+0F8H).

texture uv and wrap – the SM3110 has 10 integer bits for u,v addressing. This includes the texture width/height (which must be a power of 2, but not necessarily square) and any wrap bits. Thus, a 64-textel wide texture can have a wrap of 16 in the u direction (and similarly for v). Note that u,v coordinates supplied with primitives must be pre-multiplied by the texture width/height.

6.6.5 Copy Memory and DMA Command

The SM3110 3D engine has DMA capability in the form of the copy memory command (0x4) for the 3D command FIFO. This consists of a source and destination code and addresses, and a length in doublewords. Some useful ways in which this command can be used include:

- load local memory from PCI or AGP space (e.g., for texture DMA)
- pattern fill local memory from register (does 128-bit operations)
- set driver value in PCI or AGP space (by loading value in fill register, then transferring to local memory and from local memory to PCI or AGP space) for flagging when command FIFO operations are complete

The table below specifies the source and destination addresses and transfer length for the internal DMA. All internal addresses are 23 bits relative to the PCI base address, all external (AGP/PCI) addresses are linear 32 bits.

	<i>field</i>	<i>function</i>
+0	31:28	Command. 4H .
	27:24	Source Flags.
	23:20	Destination Flags.
	<i>value</i>	<i>semantics</i>
	0H	Internal non-aligned. Within SM3110 internal embedded DRAM.
	1H	Internal, aligned to natural 16B boundary.
	2H	State Registers except for Palette. Destination only .
	3H	Palette. Used to load palette RAM, 18b per clock. Destination only .
	4H	AGP/PCI Linear.
	5H	<i>Reserved</i> .
	6H	32b Fill Register. Source only . Used to fill memory.
	7H	Internal. Translate (texture) addresses. Destination only .
	8-FH	<i>Reserved</i> .
	<i>bits</i>	<i>field</i>
	19	Transfer Request. Must Be One for transfer to occur.
	18:00	Length in doublewords-1
+4	31:00	Starting Source Address
+8	31:00	Starting Destination Address.

When the palette is the destination, bits [09:02] of the destination address specify the palette index. Bits [01:00] specify the bytes within the palette and should be set to 0, byte 0 is Blue, byte 1 is Green and byte 2 is Red, byte 3 is not used. The source (internal or external) and destination palette addresses must be doubleword aligned. The matrix of source and destination address spaces is shown below:

Destination	Source			
	<i>0-Internal Non-aligned</i>	<i>1-Internal Aligned</i>	<i>4-External AGP/PCI</i>	<i>6-Fill Register</i>
0 - Internal Non-aligned	Yes	No	Yes	Yes
1 - Internal Aligned	No	Yes	No	Yes
2 - State Registers	Yes	No	No	No
3 - Palette RAM	Yes	No	No	No
4 - AGP/PCI	No	No	No	No
7 - Internal Texture	Yes	No	Yes	No

Table 6-3. DMA Transfer Source/Destination Matrix

Restrictions:

Transfers performed between Internal Aligned source and destination will be done at a 16 bytes per transfer.

Transfers between the 32-bit Fill Register and Internal Aligned destination will be done at 16 bytes per transfer.

The palette RAM cannot be the destination addressed through the State Register space; it must be accessed only through its own dedicated space.

All other transfers are performed at 4 bytes per transfer.

When setting up DMA operations in the 3D command FIFO, care should be taken with regard to other commands in the FIFO. Since there is only one DMA engine, each subsequent DMA command will wait until the previous DMA command is complete; that's not true for other, non-DMA commands. It may be necessary to put a sync command in the FIFO after the DMA command to wait until the DMA engine is idle before processing subsequent commands in the FIFO. For example, this would be the case if a DMA was being used to load a texture and the next command in the FIFO was a primitive which used the texture. See the synchronization command section for information on how this is handled.

Note that when DMAing in PCI space, physical addresses will be needed in the address field. If a transfer larger than a physical page (4Kbyte) is desired, simply use multiple DMA commands. If it is necessary to wait for this chain of DMAs to be done, it is only necessary to put a sync command after the last DMA. This could be used for texture loading, as described in the section on texture loading.

6.6.6 Synchronization Command

Synchronization is performed by the appropriate use of the Sync command. A conditional wait is performed on specific conditions set by internal and external hardware or driver software. When the conditions are met the wait terminates and command interpretation and execution resumes. The driver specifies a set of conditional flags that may be tested. A mask function performs the following operation:

$$\text{RESULT} = \text{OR}_{i=23 \text{ to } 00} [\text{Mask}_i \text{ AND Flag}_i]$$

When the condition becomes “true” the wait terminates.

The wait condition is defined as follows

```
condition = DrvFlg3 & cmdreg[19] |
           DrvFlg2 & cmdreg[18] |
           Sigl2D & cmdreg[17] |
           Sync2D & cmdreg[16] |
           (fc_rbusy | fc_wbusy) & cmdreg[11] |
           mcmem_busy & cmdreg[10] |
           busy & cmdreg[9] |
           bd_pipe_busy_m & cmdreg[7] |
           fd_pipe_busy_m & cmdreg[5] |
           zd_pipe_busy_m & cmdreg[4] |
           (td_ag_pipe_busy_m | td_tp_pipe_busy_m) & cmdreg[3] |
           ed_pipe_busy_m & cmdreg[2] |
           sd_pipe_busy_m & cmdreg[1] |
           cdu_busy & cmdreg[0];
```

$$\text{CDU stall} = \text{cmdreg}[25] \& (\text{cmdreg}[24] \& \sim\text{condition} \text{ or } \sim\text{cmdreg}[24] \& \text{condition})$$

Some of the useful events are covered in the sections on DMA, 2D/3D synchronization and performance issues. In addition to these, it is possible to sync on driver flags which are set in registers 0x024 and 0x026, or on external event pins.

Synchronization Command EH

<i>Field</i>	<i>Function</i>
+0	31:28 Command. EH .
	27:26 <i>Reserved</i> .
	25 Function.
	<i>Value</i> <i>Semantics</i>
	0 NOP. continue command interpretation.
	1 Sync: wait until condition, then continue command interpretation.
24	Condition Sense.
	<i>Value</i> <i>Semantics</i>
	0 Not Result.
	1 Result.
<i>Field</i>	<i>Function</i>
23:20	<i>Reserved</i> .
19:18	DrvFlags. Correspond to the Driver Flags in CDU Register bytes 027H - 024H
	<i>Value</i> <i>Semantics</i>
	0 Driver Flag Byte Bit 0 is '0'.
	1 Driver Flag Byte Bit 0 is '1'.
<i>Field</i>	<i>Function</i>
17	2D Signal
	<i>Value</i> <i>Semantics</i>
	0 Signal to 2D is low
	1 Signal to 2D is high
<i>Field</i>	<i>Function</i>
16	2D Synchronization
	<i>Value</i> <i>Semantics</i>
	0 2D has not signalled 3D
	1 2D has signalled 3D

Synchronization Command EH continued

<i>Field</i>	<i>Function</i>
15:12	<i>Reserved.</i>
11	Format Conversion.
10	Motion Compensation.
09:00	Hardware Status Mask.
09	Busy.
08	Running.
07	Direct Memory Access.
06	<i>Reserved.</i>
05	Per Fragment Unit.
04	Z-Buffering Unit.
03	Texture Mapping Unit.
02	Edge Walking Unit.
01	Triangle Setup Unit.
00	Command Decode Unit.

<i>Value</i>	<i>Semantics</i>
0	Not Busy.
1	Busy.

6.6.7 Load State Register Command

Register loading is supported by the load state register command. Using the 3D command FIFO to load registers is primarily useful because it allows serialization of state with primitives. The format is shown below.

This command is used to load state and control registers, and internal arrays.

	<i>Field</i>	<i>Function</i>
+0	31:28	Command. FH .
	27:16	Register Index. Bit [16] must be 0.
	15:00	Data

6.6.8 Texture Loading

The SM3110 requires textures to be stored in an optimized format in frame buffer memory. A number of texture formats are supported (see 6.6.10, Buffer Issues) and mipmaps are supported, with the restriction that the mipmaps (as well as the start of non-mipmapped textures) begin on 2KB byte boundaries (although sharing of lower level mipmaps within a single 2KB block is possible, as described below).

The basic mechanism for loading textures involves setting up certain registers and then transferring the texture data, either by memory-mapped write to frame buffer memory, or via DMA (see DMA section). This is done using the Load State Register command. The following registers must be set up for texture loading:

texel size	0x1E0	texture format
texture base	0x1E8	base address for texture translation (s/b 0)
texture end	0x1EC	end address for texture translation (s/b 4MB)
tex param (shift)	0x1E4	$\max(0, \log_2 u - bsxe)$ where: $\log_2 u$ is \log_2 of texture width $bsxe$ is based on texture format bit width (see Table 6-5)
tex param (row V shift)	0x1E4	$\max(0, (\log_2 u - \log_2 uoffset)) + rowVShiftOffset$ where: $\log_2 u$ is \log_2 of texture width $\log_2 uoffset$ is based on texture format bit width (see table) $rowVShiftOffset$ is based on texture format bit width (see Table 6-5)
tex param (U page offset)	0x1E4	offset from start of page (normally 0)
tex param (V page offset)	0x1E4	offset from start of page (normally 0)
tex param (page address)	0x1E4	destination address in 2K byte units

Table 6-4. Texture Loading Registers

	1 bit	2 bit	4 bit	8 bit	16 bit
bsxe	5	4	3	2	1
$\log_2 uoffset$	7	7	6	6	5
rowVShiftOffset	0	1	1	2	2

Table 6-5. Texture Loading Register Values

When loading textures, note that the actual destination address goes in the texture parameters' page address, but the destination for memory-mapped write or DMA purposes should always be the start of frame buffer memory. This allows access to the entire 4MB internal memory space in the SM3110. The address translation done by the SM3110 loading will then translate and correctly load anything sent between its base and end addresses (which is why they should be 0 and 4MB, respectively). Normally, textures (or mipmap levels of textures) will start at the beginning of the 2KB page, but in some cases, the driver will share the 2KB page of the lowest level mipmap which doesn't take an entire page. In this case, the u and v page offsets can be used to specify the starting point.

When primitives are rendered using textures, similar information is required to be loaded into the state registers, including texel format (0x0e0) and a similar value to the texture parameter register above, but with an entry for each mipmap level (texture LOD table, 0x580-0x5a8). In addition, a texture size register is required (0xe2), with the \log_2 values of the texture width (u) and height (v).

It is also necessary to be aware of the texture cache invalidate bit (0x1b9), which will need to be toggled if the texture cache contents are no longer valid. This could happen if the contents of an existing texture are reloaded.

Texture palettes are loaded using the load state register 3D command FIFO command. For SM3110, texture palette colors are 6 bits each (R,G,B), so an 18-bit data field is used for this register load command (vs. other commands, which have a 16-bit data field).

6.6.9 2D/3D synchronization

Because the SM3110 has separate FIFOs for 2D and 3D, a mechanism is provided to ensure synchronization between the two. The capability of adding a synchronization command to either FIFO, which waits for a signal from the other is available. The basic mechanism is as follows:

```
// To have 2D wait on 3D, add the following to the 2D command FIFO:
2D command 0xd

// To have 2D signal 3D, add the following to the 2D command FIFO:
2D command 0xc

// To have 3D wait on 2D, add the following to the 3D command FIFO:
3D command 0xe, sync2D = 1, function = 1, sense = 1
load state register 0x20 with 0 // this is to clear 2D signal after rcvd

// To have 3D signal 2D, add the following to the 3D command FIFO:
load state register 0x22 with 1
load state register 0x22 with 0 // this is to clear 3D signal after sent
```

These synchronization commands are useful for events which are anticipated, for example on a once-per-frame basis. If 2D commands were used to clear the z-buffer (actually there are more efficient 3D commands for this, see DMA section), the 2D blt command could be followed by a signal3D command in the 2D command FIFO and a waiton2D in the 3D command FIFO. That way, no 3D rendering commands would be processed until the z-buffer BLT was complete.

More often, it is necessary to ensure that one or the other engine is inactive before proceeding. In this case, it is necessary to wait for the corresponding FIFO to be empty, as shown below:

```
// To wait for 3D engine to be idle and FIFO empty:
flush 3D FIFO // add 4 null commands to FIFO, see 3D command FIFO desc.
poll 3D engine busy bit of register 0x1F0 until clear

// To wait for 2D engine to be idle and FIFO empty:
poll 2D engine busy bit of register 0x8F0 until clear
```

6.6.10 Buffer Issues

6.6.10.1 Buffer Alignment

The following table shows required alignments for 3D buffers on SM3110:

Buffer	Alignment	Pitch Alignment
front buffer/back buffer	4K bytes	16 pixels
z-buffer	4K bytes	16 pixels
3D command FIFO	16K bytes	n/a
textures	2K bytes	n/a

6.6.10.2 Buffer Formats

The following table shows available formats for buffers on SM3110:

Buffer	Formats
front buffer/back buffer	RGB0555, RGB565, ARGB4444, ARGB1555, RGB888, ARGB8888
z-buffer	16 bit
textures	CLUT1, CLUT2, CLUT4, CLUT8, RGB0555, RGB565, ARGB4444, ARGB1555

6.6.11 Performance Issues

A number of the performance issues in the SM3110 3D engine are related to general 'good practice' for 3D applications. This includes minimizing state changes and minimizing texture loading. For state changes, the 3D engine only has one set of state registers, so it is necessary to make sure the 3D pipeline is idle before modifying the state registers. This is done by adding the following command to the 3D command FIFO:

```
// To wait for 3D pipeline idle
control/sync command with sync bit and bits for Per Fragment Unit, Z-Buffering
Unit, Texture Mapping Unit, Edge Walking Unit, Triangle Setup Unit and Command
Decode Unit set
```

Another bottleneck could come from the fact that the 3D engine will stall if it detects that the bounding boxes for 2 triangles overlap, in order to prevent read-after-write consistency errors. This function can be enabled under software control (register 0x004). If it is known that 2 triangles will not overlap (for example in a strip/fan) this function should be turned off.

6.7 LCD Panel Programming

The SM3110 supports an LCD flat panel display. The supported panel sizes are 640x480, 800x600, 1024x768, and 1280x1024. Both DSTN and TFT panels with different color bits are also supported. The following block diagram shows the display subsystem of SM3110.

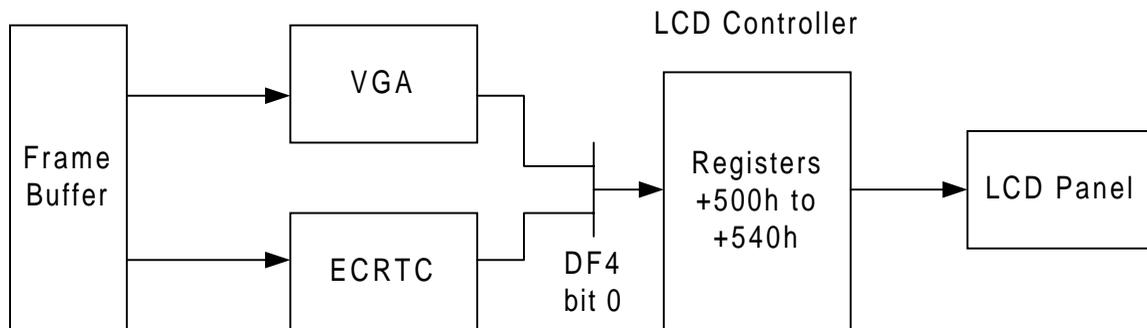


Figure 6-14. LCD Display Subsystem

6.7.1 LCD Flat Panel Registers

The following registers are LCD flat panel related. These are offsets within the 2D Control Address (see Section 4.2.1, 2D Control).

LCD Type and Miscellaneous - +510h

Bits 2-0 is the panel stretch ratio used for standard VGA modes. Values must be found using trial and error. When stretching is not needed, 0 is programmed.

Bits 7-4 define the panel type. The SM3110 family currently supports 7 types of panels.

Other bits need to be programmed according to the data sheet of the particular LCD panel.

LCD Image Compensation – +504h

Bit 7 and bit 5 enable or disable stretching for vertical and horizontal, respectively. This register will take effect only in standard VGA modes.

Dithering and Frame rate Modulation - +514h

Bit 7 and 6 define the number of frames used for frame rate modulation. More frames tend to result in better quality. Bits 5-3 control the dithering and are used if the color depth of the current mode is greater than that of the panel. Again, different values should be tried out to determine the best display quality for each particular panel.

DSTN Buffer Starting Address – +520h

When a DSTN panel is selected, some additional memory in the frame buffer is required to store a half-frame of the panel display data. This register specifies the starting address from the beginning of the frame buffer. The formula for the size required is as follows.

$$\text{Size} = \text{Panel Width} * \text{Panel Height} * 3^1 / 8^2 / 2^3$$

1. DSTN is 3 bits per pixel.
2. There are 8 bits per byte.
3. Half frame.

LCD CRTC Registers – +530h ... +53Ch

These registers contain the same values from VGA CRTC or ECRTC except for register 534 bits 8- 0 and register 53C bits 10-0, which are panel width and height, respectively. Note that the horizontal parameters are counted by characters, that is, they need to be divided by 8.

LCD Adjustment Control – +540h

This register is used to fine-tune the display image size and centering. Each panel has different characteristics and needs to be tuned to get the best display image size and positioning.

ECRTC Stretch Register – +EE4h

This register is used to stretch the display image under enhanced mode. Each value represents a different stretching ratio. There is a small, fixed set of stretch ratios defined for vertical and horizontal stretching. The closest value is picked to achieve the maximum stretch ratio, i.e., the largest possible image for the given panel resolution.

6.7.1.1 *Initialization*

During power up, only register 510 (LCD Type and Misc. Register) needs to be set according to the panel. If the panel is DSTN, register 520 (Base Address Register) needs to be programmed, too. Sufficient frame buffer memory needs to be reserved for DSTN panel to display properly.

6.7.1.2 *Set to Desired Mode*

Setting to a desired mode requires programming LCD-related registers as well as VGA CRTC or ECRTC registers. DCLK (dot clock – register 90, 91, and 92) needs to be programmed accordingly, too.

6.7.1.3 *Standard VGA Modes*

To set to standard VGA modes, the standard VGA CRTC registers are programmed as usual. Registers +530h to +53Ch need to be programmed according to the panel size. Use register 504 to enable or disable stretching. If stretching is disabled, the display image will be centered by default. Also use registers 530 to 540 to adjust the display image to desired position.

6.7.1.4 *Enhanced Modes*

No matter which enhanced mode, the timing parameters of the panel resolution have to be programmed into ECRTC registers as well as registers 530 to 53C. Refresh rate has to be 60Hz. For example, a panel with size of 1024x768 needs to use the parameters from the 1024x768 60Hz portion of the table. Any mode with resolutions higher than the panel size can't be set.

To center the display image, view port registers (E30 to E34) need to be programmed by the following formula:

$$\text{View port start} = (\text{panel size} - \text{real resolution}) / 2$$

$$\text{View port end} = \text{view port start} + \text{real resolution}$$

To stretch the display image, register 504 has no effect. Instead, register EE4 is used. The following tables list all the possible stretch ratios to be programmed.

Table 6-6. Horizontal Stretch Ratio

Mode Width	Panel Width	Ratio	Result Width	Program Value
640	800	4 -> 5	800	3
640	1024	5 -> 8	1024	4
720	1024	3 -> 4	960	2
800	1024	25 -> 32	1024	7
640	1280	1 -> 2	1280	1
720	1280	75 -> 128	1230	8
800	1280	5 -> 8	1280	4
1024	1280	4 -> 5	1280	3

Table 6-7. Vertical Stretch Ratio

Mode Height	Panel Height	Ratio	Result Height	Program Value
400	600	3 -> 4	533	2
480	600	4 -> 5	600	3
400	768	75 -> 128	682	8
480	768	5 -> 8	768	4
600	768	25 -> 32	768	4
400	1024	15 -> 32	853	5
480	1024	15 -> 32	1024	5
600	1024	75 -> 128	1024	8
768	1024	3 -> 4	1024	2

For some resolutions, it is not possible to stretch to the exact panel size. In this case, the closest ratio is picked. It is still necessary to program viewport registers to center the image after stretching. Again, registers 530 to 540 can be used to fine-tune the display image to the desired position.

6.7.1.5 Set Mode Sequence

The following describes the sequence of set mode:

```
Turn off screen through sequence register - 3C4 index 1 bit 5
If(Standard VGA Mode)
{
  Set standard VGA registers - 3C4/5, 3D4/5, 3C0/1, 3CE/F, etc. according to
  the mode
  Initialize palette if necessary - 3C8, 3C9
  Turn on or off stretching - 504
}
else// Enhanced mode
{
  Disable ECRTC0 - program 0 to DF4
  Define a surface for display - A00...AF0, A04...AF4
  Program channel surface index register - B34
  Program display format - D30
  Program enhanced dot clock - 090, 091, 092
  Put correct timing parameters into ECRTC registers - E80... E94
  Program display view port registers - E30...E34
  Initialize palette if necessary - 3C8, 3C9
  Program stretch register according to Table 6-6 and Table 6-7 - EE4
  Enable ECRTC0 - Program 5 to DF4
}
Program LCD dithering register - 514
Program LCD timing registers - 530...53C
Use LCD Adjustment Control Register to center the display image - 540
```

6.8 Dual View

6.8.1 Overview

The SM3110 supports two display subsystems. They can be set up to display the same content from the same frame buffer location. This is called simultaneous view. They can also be set up to display the same or different portions of frame buffer with different refresh rates and color depths. This is called dual view. (Support for dual view is provided partly by the SM3110 hardware and partly by the host operating system; if the operating system does not allow a single graphics chip to drive two displays, this feature is not available.) The following diagram shows the relationship of these two display subsystems.

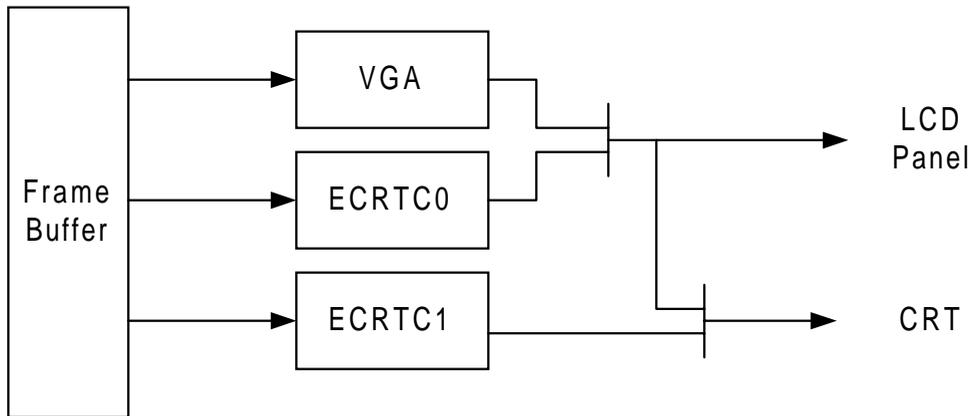


Figure 6-15. Dual View Mechanism

6.8.2 Enabling Second Display

Enabling the second display involves the following registers:

Second Dot Clock – 1090, 1091, 1092

Correct clock rate needs to be set for different modes and refresh rates.

Surface/Channel/Format registers – A00...AF0, B84, 1D30

A different surface memory has to be selected and programmed. Channel index register B84 has to be programmed to the selected surface index. Also register 1D30 (display channel format) needs to be programmed to the correct format according to the mode.

Display 0 and 1 control registers – DF4 and 1DF4

Set bit 4 of DF4 to 1 to hook the second display output to ECRTC1. Also set 1DF4 to 5 to enable ECRTC1.

Double Scan and Sync Polarity Registers – 1DF5, 1DF6

These two registers control both horizontal and vertical double scan and polarity according to each mode.

ECRTC1 View Port and Timing registers – 1E00...1E94

ECRTC1 supports only CRT, so no LCD consideration is involved. These registers can be set to different resolution and refresh rates.

6.8.2.1 *Enable Simultaneous View*

Bit 4 of Display 0 Control Register (DF4) enables dual view. Set it to 0 will enable simultaneous view. Everything displaying on the first display will show up on the second display, too.

6.8.2.2 *Enable Dual View*

The following sequence will enable dual view:

```
Set to simultaneous view - turn off DF4 bit 4
Disable ECRTC1 - program 0 to 1DF4
Define a surface - A00...AF0, A04...AF4
Program channel surface index register - B84
Program second display format - 1D30
Program second dot clock - 1090, 1091, 1092
Put correct timing parameters into ECRTC registers - 1E80... 1E94
Program second display view port registers - 1E30...1E34
Enable ECRTC1 - Program 5 to 1DF4
Enable dual view - turn on DF4 bit 4
```

6.9 Motion Compensation

The SM3110 supports Motion Compensation (MC) hardware acceleration when displaying MPEG-2 video data. While the graphics chip is performing motion compensation, the rest of video decoding (variable length decoding, inverse scan, inverse quantization and Inverse Discrete Cosine Transform (IDCT)) is done by a software client (e.g., SoftDVD®).

6.9.1 Overview

MPEG-2 uses three kinds of picture storage methods. Intra (*I frames*), Predicted (*P frames*) and Bi-directional (*B frames*). Intra frames are frames coded as standalone images, very much like JPEG pictures. They allow random access points within a video stream and create a reference frame from which other frames are built. In general, an I frame will occur around twice a second. Predicted frames contain motion vectors describing the difference from the closest previous I frame or P frame. This forward prediction allows for greater compression than with I frames. Bi-directional frames are like P frames in that they use motion vectors, but with B frames, motion vectors are generated by looking both forward and backward. This forward and backward referencing allow B frames to have the greatest compression ratio. Figure 6-16 shows the data flow during this process.

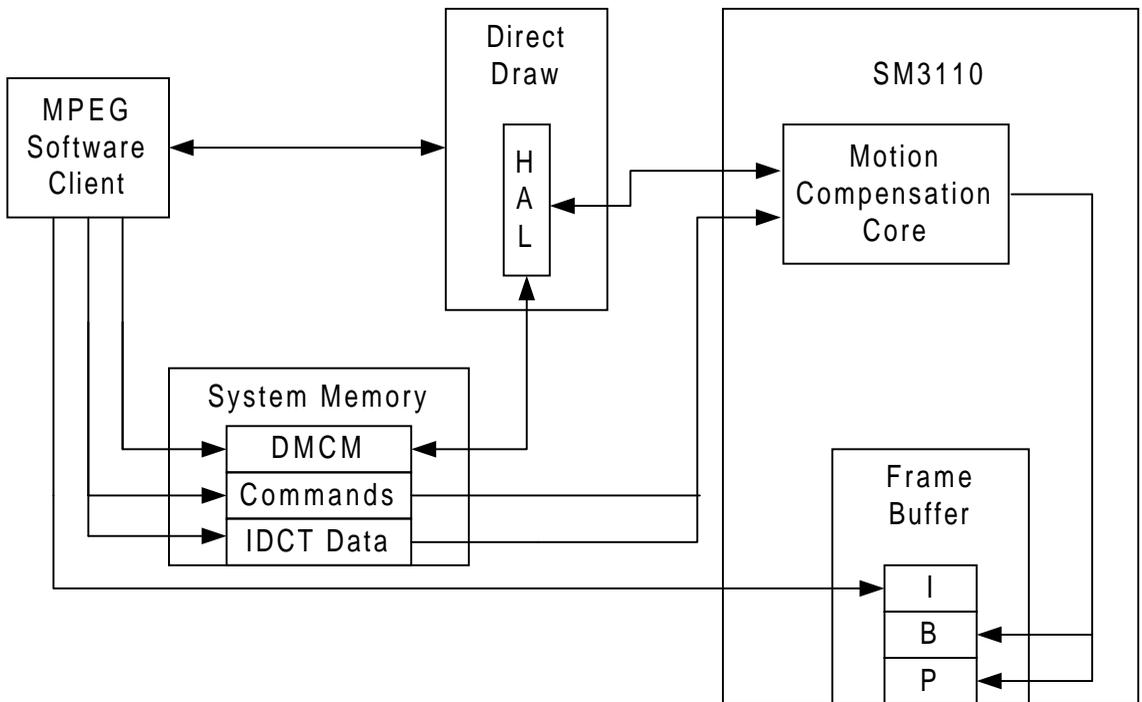


Figure 6-16. Motion Compensation Dataflow

Microsoft's DirectDraw is used as the interface between the MPEG software client and the motion compensation core. The block diagram shown above illustrates the data flow. The client and the driver communicate following the steps shown below.

<u>Client</u>	<u>Driver</u>
1 Direct Show - Tell driver to create surface.	Create Surface
2 Direct Show - Will lock/unlock the surface	Fill data structure
3 Decoder - Tell Driver what slot for decode	
4 Decoder - Lock, Tells what surface to display	Display Slot
5 Decoder - Goto 3	

6.9.2 Block Diagram

To use the integrated MC core, the MC driver first ensures that the MC subsystem is idle by programming the status registers. The driver writes the MC commands and immediate data into the 3D command/parameter FIFO. The MC subsystem is returned to normal processing mode by issuing an End of Stream MC command. These steps are explained in detail in the following sections. Figure 6-17 shows the MC block diagram of the SM3110.

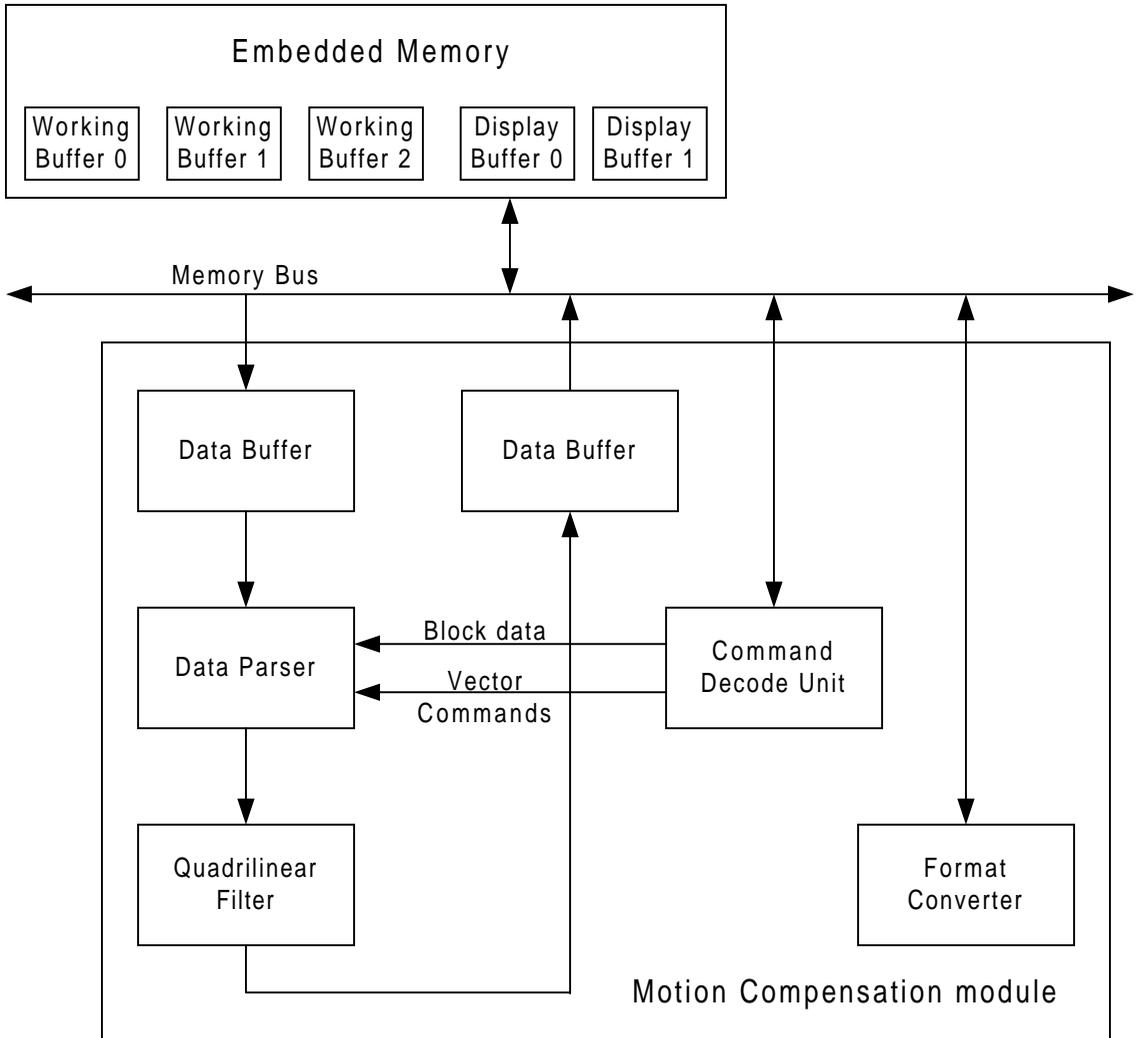


Figure 6-17. Motion Compensation Block Diagram

6.9.3 Motion Compensation Client Commands

Motion Compensation is a “back-end” process which is part of the overall MPEG-2 decoding model. The “front-end” process includes parsing the bitstream and reconstructing IDCT data. The interface between the front-end and the MC back-end will include the command stream. Data can be interleaved in such a way that the command comes first, followed by any IDCT data required by it, followed by the next command, etc.

6.9.3.1 DMCM Commands

DMCM (dwCommand field in the DMCM structure) is a 4CC command code which is set by the client (SoftDVD). It determines which action the DDraw driver will take on Unlock. There are two commands available in this implementation.

```
//----- COMMANDS -----  
#define DMCM_CMD_NONE 0  
#define DMCM_CMD_DISPLAY 2
```

The default command is DMCM_CMD_NONE - do nothing. It is used when client locks surface just to check driver's capabilities, version number or current state.

The other command is: DMCM_CMD_DISPLAY - new frame has to be shown.

These commands are sent to the MC core through the Command Decode Unit (CDU) from the 3D command FIFO.

6.9.3.2 DMCM surface allocation

The DMCM DDraw surface is an interface for hardware-assisted MPEG motion compensation using Silicon Magic's SM3110 graphics chips. The DirectShow client (SoftDVD) passes to the DDraw driver motion vectors, coefficient data (the output of IDCT) and control data on a frame by frame basis.

Memory is allocated for the DirectDraw motion compensation surface (DMCM) structure. The definition for this interface structure is given below.

```
//----- DMCM COMMANDS -----
// Commands for DDraw
#ifdef dmcm_h
#define dmcm_h
#define DMCM_CMD_NONE          0
#define DMCM_CMD_DISPLAY      2

typedef struct MCSiMagicBeginData_TAG
{
    BYTE    PictureStructure;
    BYTE    PictureCodingType;
    BYTE    TopFieldFirst;
    int    ForwardRefSlot;
    int    BackwardRefSlot;
    int    DestinationSlot;
} MCSiMagicBeginData;

typedef struct MCSiMagicMacroBlk_TAG
{
    int    hOffset;           // upper left macroblock coordinates in pels
    int    vOffset;           // upper left macroblock coordinates in scanlines
    BYTE    DCTType;           // Field DCT or Frame DCT
    BYTE    codedBlockPattern; // Coded Block Pattern
    BYTE    overflowCodedBlockPattern; // Overflow CodedBlock Pattern
    BYTE    motionType;
    int    PMV[2][2][2];      // contains motion vectors in half pel units
    int    motionFieldSelect[2][2]; // Motion
    int    macroblockType;
    LPBYTE *lpIDCTData;
    LPBYTE *lpOverflowData;
} MCSiMagicMacroBlk;
```

```

typedef struct
{
    struct
    {
        PBYTE      pIDCTData;    // pointer to IDCT data buffers
        PBYTE      pYBuffer;     // pointer to Y plane for I-frame
        PBYTE      pUBuffer;     // pointer to U plane for I-frame
        PBYTE      pVBuffer;     // pointer to V plane for I-frame
        UINT       uPitch;       // pitch for Y in frame buffer; half of this for UV
        UINT       uWaitTime;    // time to sleep in ms.
        LPVOID     lpfStartDecodeDriverFunction; // Called at start of decode
        LPVOID     lpfEndDecodeDriverFunction;   // Called at end of decode
        LPVOID     lpfProcessMBDriverFunction;   // Process Macroblock Function
        DWORD      dwIntstanceData; //value to be passed back in parameter of
                                   the *lpfStartNewDecodeDriverFunction
    }
    DriverData;                    // initialized by DDraw upon surface LOCK

    struct
    {
        DWORD      dwCmd;         // DMCM command (display)
        DWORD      dwTaskType;    // DMCM command (display)
        DWORD      dwDisplayTaskType; // DMCM command (display)
        int        dwDecodeBuffIndex;
        DWORD      dwDecodeTaskCount;
        DWORD      dwDataSize;    // Size of IDCT data to transfer
        DWORD      dwFrameType;   // Type of Frame (1:I, 2:P, 3:B)
        DWORD      dwSlotNumber;  // Logic slot number
        DWORD      dwDisplayBuffIndex;
        DWORD      dwBobMode;     // 0:off, 1:Top 1'st, 2:Btm 1'st, 3:Only
                                   Top, 4:Only Btm
    }
    ClientData;                    // filled by the client before the surface
                                   UNLOCK

} SURFMCSiMagic, *LPSURFMCSiMagic;

typedef LPSURFMCSiMagic (__stdcall *LPSTARTDECODEFUNC_MG)(MCSiMagicBegin-
    Data *);
typedef LPSURFMCSiMagic (__stdcall *LPENDECODEFUNC_MG)();
typedef LPSURFMCSiMagic (__stdcall *LPPROCESSMBFUNC_MG)(MCSiMagicMacroBlk
    *);

#endif

extern LPSURFMCSiMagic __stdcall StartDecodeMotionComp( MCSiMagicBeginData
    *);
extern LPSURFMCSiMagic __stdcall EndDecodeMotionComp();
extern LPSURFMCSiMagic __stdcall ProcessMacroBlock(MCSiMagicMacroBlk *);

```

6.9.4 Registers

6.9.4.1 Start and End Decode Functions

Two functions are defined in the driver that start and end the MC decode process. The driver passes pointers to these two functions to SoftDVD. The StartDecode function checks the status register to ensure that the 3D/MC subsystem is not busy and then writes the MC command to the command FIFO. SoftDVD passes the size of the data written to the driver during in the EndDecode function using which the driver writes the IDCT data to the command FIFO.

6.9.4.2 CDU Start Motion Compensation Command

The CDU interprets **0xD (13)** as the Motion Compensation command. This command places the CDU into MC mode.

Field	Function
31:28	Command. DH
27:00	Reserved.

6.9.4.3 End of Stream Instruction

In MC mode, the Command Decode Unit interprets 3D command FIFO data as MC core instructions. To leave MC mode, an End of Stream instruction is sent to the command FIFO. This instruction is **20000000H**.

Field	Function
31:00	MC End of Stream. 20000000H

6.9.4.4 Format Conversion Instruction

The MPEG-2 decoded bitstream is converted from the 4:2:0 YCrCb format to the SM3110 supported 4:2:2 format. The Command Decode Unit interprets a command with a leading value of **0xC (12)** in the four msb bits as an MC format conversion command. These commands are not valid between the MC command **0xD (13)** and the End of Stream command **20000000H**.

Field	Function
31:28	Command CH.
27:26	Slot Number (0,1,2,3). Selects which slot should be converted.
25:24	Horizontal Compression Ratio

<i>Value</i>	<i>Semantics</i>
0	1-1 compression (no compression)
1	2-1 compression
2	4-1 compression
3	8-1 compression
23:00	Destination buffer address.

In MC mode, the registers for slot base addresses and plane offsets need to be set by the driver. These are as follows:

Slot Base Addresses:

Offset	Field	Function
+340H	31:00	Slot 0 Base Address
+360H	31:00	Slot 1 Base Address
+380H	31:00	Slot 2 Base Address
+3A0H	31:00	Slot 3 Base Address

Plane 1/ Plane 3 Offsets:

Offset	Field	Function
+344H	31:00	Slot 0 Plane 1/Plane 3 Offset
+364H	31:00	Slot 1 Plane 1/Plane 3 Offset
+384H	31:00	Slot 2 Plane 1/Plane 3 Offset
+3A4H	31:00	Slot 3 Plane 1/Plane 3 Offset

Plane 2 Offsets:

Offset	Field	Function
+348H	31:00	Slot 0 Plane 2 Offset
+368H	31:00	Slot 1 Plane 2 Offset
+388H	31:00	Slot 2 Plane 2 Offset
+3A8H	31:00	Slot 3 Plane 2 Offset

6.9.5 Display Memory Requirements

6.9.5.1 Frame and Format Buffers

MPEG-2 uses a 4:2:0 video format which implies that for each video frame, the number of samples of each chrominance component (Cr and Cb) is one-half of the number of samples of luminance (Y), both horizontally and vertically.

With the 4:2:0 format, each pixel will take an average of 12 bits (eight bits for the luma and four for the chroma) which translates to each pixel requiring 1.5 bytes of storage. This tells us that one frame or picture in NTSC format which is 720 x 480 will require 518,400 bytes of memory while one frame in PAL format which is 720 x 576 will require 622,080 bytes of memory. The motion compensation core requires that 3 buffers of equal size need to be available to store the I, P and B frames that will be shown.

Table 6-8. Memory required for the three frame buffers

NTSC:	$720 \times 480 \times 1.5 = 518,400 \text{ bytes} \times 3 = 1,555,200 \text{ bytes}$
PAL:	$720 \times 576 \times 1.5 = 622,080 \text{ bytes} \times 3 = 1,866,240 \text{ bytes}$

The MPEG-2 decoded bitstream is in the 4:2:0 YCrCb pixel format as mentioned earlier. SM3110 can directly display 4:2:2 YCrCb progressive video. This requires a conversion to the 4:2:2 video format. Two format conversion buffers are allocated for this purpose. The memory requirements are shown below.

Table 6-9. Memory required for the two format conversion buffers

NTSC:	$720 \times 480 \times 2 = 691,200 \text{ bytes} \times 2 = 1,382,400 \text{ bytes}$
PAL:	$720 \times 576 \times 2 = 829,440 \text{ bytes} \times 2 = 1,658,880 \text{ bytes}$

The total memory requirement for NTSC is 1,555,200 bytes + 1,382,400 bytes = 2,937,600 bytes and the total memory required for PAL is 1,866,240 bytes + 1,658,880 bytes = 3,525,120 bytes.

6.9.5.2 Screen Resolutions Supported

Based on the above memory requirements for frame buffers and format conversion buffers, the following screen resolutions are supported.

Table 6-10. Screen Resolutions Supported

Format	Total Memory Available (bytes)	Required for Motion Comp. (bytes)	Memory Available for Display (bytes)	Supported Display Resolution
NTSC	4,194,304	2,937,600	1,256,704	800x600@16bpp (=960,000 bytes) 1024x768@8bpp (=786,432 bytes)
PAL	4,194,304	3,525,120	669,184	800x600@8bpp (=480,000 bytes)

6.9.5.3 System Memory Requirements

Table 6-11 lists the system memory requirement for the DirectDraw surface command buffer. This is the buffer that is used to store the commands and data that are sent to the video card driver (HAL) from SoftDVD

Table 6-11. Command and Data buffer sizes (in bytes) for NTSC and PAL format

ID	Type	Formula	NTSC	PAL
A	Macro Blocks	(pixels)/(16*16)	1,350	1,620
B	# Blocks	6 * A	8,100	9,720
C	Commands	12 * B	97,200	116,640
D	Data	64 * B	518,400	622,080
E	Data with 8 or 16 bit Error	2 * D	1,036,800	1,244,160
	Total Buffer Size	C + E	1,134,000	1,360,600

6.10 Power Management

6.10.1 Overview

The Power Management Function supports Microsoft's OnNow specification and the VESA BIOS Extensions/Power Management (VBE/PM) Standard. The Power Management Function receives commands (function calls) from the Operating System to place the display controller and display monitor(s) into one of several power saving/consuming states. The Power Management Function does not establish any power usage policy; it only supports the policy defined by the operating system.

Power management supports four states: On, Standby, Suspend and Off. These states represent four successive levels of power consumption. The operating system selects the appropriate state based upon user parameters, available system power and keyboard/mouse events.

6.10.2 OnNow Power Management

For Power Management in a Windows operating system environment, Microsoft has specified how power management can be supported "OnNow Power Management and Display Device Class Drivers". This specification describes which functions are expected from a miniVDD.

6.10.3 VESA BIOS Power Management

For BIOS Power Management, the VBE/PM Standard specifies how power management is supported. This standard specifies a set of functions (actually sub-functions) which are invoked by application software (using interrupt 10H) to utilize power saving features of display hardware.

6.10.4 Other Power Management

Other than VESA and OnNow power management, software can perform dynamic power management according to the hardware usage. SM3110 is designed in a way that each individual block can be shut off independently. The remaining issue will be when and what to shut off or turn on at any particular time. Generally speaking, under a normal simultaneous view state, display 1 and scaler 1 can all be turned off, while display 0 should be always on.

6.10.5 Registers for OnNow and VESA DPMS

What follows are details to support power management either in BIOS or a miniVDD using the SM3110. Support for each power consuming/saving state is described as if the state is entered without any dependence upon the previous state.

6.10.5.1 D0 – On

Register	bit(s)	Description
+0DF7H +1DF7H	01:00	Enable Horizontal Sync by setting field to 0.
+0DF7H +1DF7H	03:02	Enable Vertical Sync by setting field to 0.
+0D30H +1D30H	31:21	Restore video output by setting field to a nonzero number based on display format and stride information.
+00D4H	06:00	Enable MCLK by setting all bits to 0.
+00D8H	03:00	Enable DCLK by setting all bits to 0.
+0DF7H +1DF7H	07	Power up DAC by setting bit to 0.

6.10.5.2 D1 – Standby

Register	bit(s)	Description
+0DF7H +1DF7H	01:00	Disable Horizontal Sync by setting field to the value 1 or 2. This depends upon what the display panel expects as a pulse (a voltage increase pulse or a voltage drop pulse). 0 implies voltage is always low, 1 implies voltage is always high.
+0DF7H +1DF7H	03:02	Enable Vertical Sync by setting field to 0.
+0D30H +1D30H	31:21	Make Video go “Blanked” by setting field to 0.
+00D4H	06:00	Enable MCLK by setting all bits to 0.
+00D8H	03:00	Disable DCLK by setting all bits to 1.
+0DF7H +1DF7H	07	Power down DAC by setting bit field to 1.

6.10.5.3 D2 – Suspend

Register	bit(s)	Description
+0DF7H +1DF7H	01:00	Enable Horizontal Sync by setting field to 0.
+0DF7H +1DF7H	03:02	Disable Vertical Sync by setting field to 1 or 2. Depends upon what display panel expects as a pulse (a voltage increase pulse or a voltage drop pulse). 0 implies voltage is always low, 1 implies voltage is always high.
+0D30H +1D30H	31:21	Make Video go “Blanked” by setting field to 0.
+00D4H	06:00	Enable MCLK by setting all bits to 0.
+00D8H	03:00	Disable DCLK by setting all bits to 1.
+0DF7H +1DF7H	07	Power down DAC by setting bit field to 1.

6.10.5.4 D3 – Off

Register	bit(s)	Description
+0DF7H +1DF7H	01:00	Disable Horizontal Sync by setting field to the value 1 or 2. This depends upon what the display panel expects as a pulse (a voltage increase pulse or a voltage drop pulse). 0 implies voltage is always low, 1 implies voltage is always high.
+0DF7H +1DF7H	03:02	Disable Vertical Sync by setting field to 1 or 2. Depends upon what display panel expects as a pulse (a voltage increase pulse or a voltage drop pulse). 0 implies voltage is always low, 1 implies voltage is always high.
+0D30H +1D30H	31:21	Make Video go “Blanked” by setting field to 0.
+00D4H	06:00	Disable MCLK by setting all bits to 1.
+00D8H	03:00	Disable DCLK by setting all bits to 1.
+0DF7H +1DF7H	07	Power down DAC by setting bit field to 1.

6.10.6 Registers for Dynamic Power Management

The registers to control individual blocks are 0xD4 and 0xD8. The following table is a breakdown of their definition.

Block	Registers	Description
Display 0	+00D8H bit 0	Should be always on
Display 1	+00D8H bit 1, +00D4H bit 3	Shut off when using simultaneous view
Scaler 0	+00D8H bit 3, +00D4H bit 1	Turn on during CreateSurface() in DirectDraw Turn off during DestroySurface() in DirectDraw
Scaler 1	+00D8H bit 2, +00D4H bit 2	When dual view is enabled, same condition as Scaler 0
2D Engine	+00D4H bit 5	Turn on during 2D driver enable Turn off during 2D driver disable
3D Engine	+00D4H bit 6	Turn on during context creation in Direct3D Turn off during context destroy in Direct3D
MIU	+00D4H bit 0	Should be always on

6.11 Video Capture

6.11.1 Overview

The SM3110 controller supports YUV4:2:2 and CCIR 656 video input and interrupt mechanisms for video capture. A I²C video decoder must be used to convert the incoming NTSC/PAL/SECAM video signal to digital streams. A configurable input/output data port is provided to access up to 8 bits of data that can be used for I²C device control.

6.11.2 I²C device interface

Pins 2 and 3 of the configurable data port are reserved to control the video decoder I²C device. Pin 2 is for clock (SCL) and pin 3 is for data (SDA).

Sample code: Control CIO port (configurable input/output data port)

```

SCL_BIT    equ    4                ;bit 2 for Clock
SDA_BIT    equ    8                ;bit 3 for Data

InitCIOport    proc    near
    mov     fs,CmdPortSelector      ;mapio selector
    mov     al,0F3h                ;bit 3:2=00 to enable the mask
    mov     fs:[412h],al           ;CIO control mask
    mov     holding_register,0FFh  ;pull high init.
    ret
InitCIOport    endp

WriteBit    macro    bMask,fHighLow ;set clock/data pin to Low(0) or High(non-0)
    mov     al,fHighLow
    neg     al                      ;entry:0,    non-0
    sbb     al,al                   ;    0,    FF
    and     al,bMask                ;    0,    SCL_BIT/SDA_BIT
    and     holding_register, NOT bMask
    or      holding_register, al
    mov     al, holding_register
    mov     byte ptr fs:[411h],al   ;CIO output
endm

ReadBit    macro    bMask                ;read clock or data pin disable mask
    mov     ah,fs:[412h]             ;CIO control mask
    or      ah,bMask                ;set 1 to disable
    mov     fs:[412h],ah           ;CIO control mask
;Read in holding_register
    mov     al,fs:[410h]            ;CIO input port
    mov     holding_register,al
;enable mask
    and     ah,NOT bMask            ;set 0 to enable

```

```

    mov    fs:[412h],ah                ;CIO control mask
;return in al=0 or non-0
    and    al, bMask                  ;data in mask bit
    endm

Delay macro
    push   cx
    mov    cx, delay_time             ;delay time adjust to fit I2C device
                                           duration spec

delay_loop:
    loop   delay_loop
    pop    cx
    endm

```

With macros for reading and writing individual bits to the I²C device, this is an example of how to write one full byte of data to the I²C device.

```

SendByteToI2cDevice proc near ;entry: bData
    mov    bl,bData
    mov    cx,8                       ;8 bits
    WriteBit    SCL_BIT,0             ;write clock pin to low
sbi2c_loop:
    shl    bl,1
    sbb    bh,bh
    WriteBit    SDA_BIT,bh           ;write data pin

    WriteBit    SCL_BIT,1             ;write clock pin to high
    Delay
    WriteBit    SCL_BIT,0             ;write clock pin to low to create clock pulse
    Delay
    loop   sbi2c_loop                ;loop 8 times for a BYTE

    Delay

    WriteBit    SDA_BIT,1             ;write data pin with HIGH to check for ACK
    Delay
    Delay
    Delay
    WriteBit    SCL_BIT,1             ;write clock pin to HIGH for a clock pulse
    Delay

    ReadBit     SDA_BIT                ;read bit in AL to see if it is pulled low by
                                           I2C device

    Delay

    push   ax                          ;save
    WriteBit    SCL_BIT,0             ;write clock pin to LOW for a clock pulse
    Delay
    pop     ax                          ;restore
    ret                                     ;return in AL with 0 (ACK) or non-0 (no ACK)

```

```
SendByteToI2cDevice  endp

ReadByteFromI2cDevice proc near
    mov     cx,8                ;8 bits
    WriteBit SCL_BIT,0        ;write clock pin to low
rbi2c_loop:
    WriteBit SCL_BIT,1        ;write clock pin to high
    Delay

    ReadBit  SDA_BIT          ;write data pin in AL=0 or non-0
    neg     al
    adc     bl,bl              ;carry bit shift to BL

    WriteBit SCL_BIT,0        ;write clock pin to low to create a clock pulse
    Delay
    loop   rbi2c_loop         ;loop 8 times for a BYTE

    Delay

    WriteBit SDA_BIT,0        ;write data pin to send ACK
    Delay
    WriteBit SCL_BIT,1        ;write clock pin to HIGH for a clock pulse
    Delay
    WriteBit SCL_BIT,0        ;write clock pin to LOW for a clock pulse
    Delay

    WriteBit SDA_BIT,1        ;write data pin device
    Delay

    mov     ax,bx              ; return read BYTE data in AL
    ret
ReadByteFromI2cDevice  endp
```

6.11.3 Video port control

As with other graphics data, the SM3110 surface mechanism is used to associate the video input with a buffer. The scaler channels are used to associate the surface to a display. This section provides examples to perform common control operations. These are explained as specific examples for clarity, but could be combined in real applications.

To prepare for video capture function, following steps are required at the initialization period.

1. Define scaler and video capture surfaces.
2. Channel controls - assign surface
3. Set Overlay priority.
4. Set ColorKey

Sample code: Prepare for video capture

```

PrepareVideoCapture      proc   near
    ;;Define a surfaces for video data input
    mov     ax, VIDCAP_SURFACEINDEX    ;surface descriptor used for video cap-
                                         ture

    mov     surfacenum,ax
    mov     eax, VIDCAP_WIDTH
    mov     surfacestride,eax
    mov     edx, VIDCAP_HEIGHT
    mul     edx
    mov     edx,dwPrivatePhyAddr      ;total mem-reserved buffer
    sub     edx,eax                    ;assign surface location at the end of
                                         memory

    mov     surfacebase,edx
    Call    DefineSurface              ;define the surface to accept video data
    mov     fs,CmdPortSelector        ;mapio selector
    mov     ax,VIDCAP_HEIGHT
    mov     fs:[VideoPortMaxHeight],ax ;set video input height here
    ;;assign the defined for scaler and video capture channel
    mov     eax,surfacenum
    mov     fs:[ChScaler0SIndex],eax  ;assign to scaler channel
    mov     fs:[ChVidCapSIndex],eax  ;assign to video capture channel
    xor     eax,eax
    mov     fs:[ChScaler0Offsets],eax ;scaler channel default start at (0,0)
    mov     fs:[ChVidCapOffsets],eax ;video capture channel always start at
                                         (0,0)

    ;;set color key
    mov     eax,color-key
    mov     fs:[Dsp0ColCmp],eax       ;set dest. Color-key color
    mov     eax,200h
    mov     fs:[Dsp0MixCtl],eax       ;enable dest. Color-key

    ;;set overlay control
    mov     eax,10h                    ;BG:scaler channel, FG:display channel
    mov     fs:[Overlay0Priority],eax ;enable Color-key overlay
    ret

PrepareVideoCapture      endp

```

The method used to shrink or stretch video data from a scaler channel to a display channel was described in display controls, Section 6.5.1: 2D Functions, Display Operations.

To enable or disable video, both the video input port and the scaler channel need to be programmed.

Sample code: Enable or Disable Video

```
EnableVideo proc near
;;enable video input port
    mov     fs,CmdPortSelector      ;mapio selector
    mov     al,prescale             ;prescale factor, 0,1,2,3 to shrink
                                        video input by factor 1,1/2,1/4 and 1/8
    shl     al,4                    ;move to bit 4 and 5
    or      al,1                    ;bit 0 to enable/disable
    mov     fs:[VideoPortConfig],al ;set video input height here
;;enable video to display
    mov     eax,58h
    mov     fs:[ChScaler0Format],eax ;set format to YUY2 to enable
    ret
EnableVideo endp

DisableVideo proc near
;;disable video input
    mov     fs,CmdPortSelector      ;mapio selector
    mov     al,prescale             ;prescale factor, 0,1,2,3 to shrink
                                        video input by factor 1,1/2,1/4 and 1/8
    shl     al,4                    ;move to bit 4 and 5
    mov     fs:[VideoPortConfig],al ;set video input height here
;;disable video to display
    xor     eax,eax
    mov     fs:[ChScaler0Format],eax ;0 to disable
    ret
DisableVideo endp
```

6.11.4 Interrupt for video capture

An interrupt line serves as the hardware interrupt scheme for video capture. The following functions show how to enable, disable and clear the interrupt line.

Sample code: Control interrupt line

```

EnableInterrupt proc near
    mov     fs,CmdPortSelector           ;mapio selector
    mov     al,020h                     ; bit 5 for Video capture interrupt
    or      fs:[ InterruptMask],al      ; 0F1h
    mov     al,02h                       ;bit 1 for External Video Input
    or      fs:[DispIntFlag],al         ; 0DF0h
    ret
EnableInterrupt endp

DisableInterrupt proc near
    mov     fs,CmdPortSelector           ;mapio selector
    mov     al,0fch                      ;bit 1 for External Video Input, bit
                                        ;0:reset flag bit
    and     fs:[DispIntFlag],al         ; 0DF0h
    mov     al,0dfh                     ; bit 5 for Video capture interrupt
    and     fs:[ InterruptMask],al      ; 0F1h
    ret
DisableInterrupt endp

ClearInterrupt proc near
    mov     fs,CmdPortSelector           ;mapio selector
    mov     al,02h                       ;bit 1 for External Video Input
    or      fs:[DispIntFlag],al         ; 0DF0h
    ret
ClearInterrupt endp

```

6.12 TV Out

See Chapter 7, Application Notes in the TV encoder section for details.

6.13 Diagnostics

The SM3110 contains some features useful for diagnostic purposes. The primary one of these is the diagnostic port, which allows predefined signals from different subsystems in the chip to be brought out to 8 output pins. To do this, software must set the appropriate 3-bit code in the diagnostic port register for the desired subsystem (other subsystems must be set to 0 to act as a pass-through).

In addition, a performance counter capability allows counting of any of the 8 diagnostic port bits selected by setting the appropriate counter source and enabling the performance counter in register 0x03E of the Command Decode Unit. The resultant performance count appears in register 0x010.

Also, some of the SM3110 internal SRAM values for the Triangle Setup Unit and Command Decode Unit are visible via registers 0x5C0-0x5FC, 0x600-0x7FC and 0x800-0xBFC.

6.14 Software Definitions

This section provides software definitions that are useful in understanding and programming the functions of the SM3110.

6.14.1 Equates

```

VENDORID                equ    8888H
DEVICEID                 equ    4831H
Memory/Register Map
; Offsets to register areas, command ports, image ports, cursor area and
; deferred register write buffer

PERIPHERAL_CTL_BASE     equ    01FE0000H
REGISTER_BASE_OFFSET    equ    01FF0000H
IMAGE_BASE_OFFSET       equ    01FFC000H

; PrivateMemSelector equates
PRIVATEAREA_SIZE        equ    20000H    ;we reserve 128K for all fifos
CMD3DFIFO_SIZE          equ    10000H    ;64K 3d cmd fifo
CMDFIFO_SIZE            equ    4000H     ;16K 2d cmd fifo
IMGFIFO_SIZE            equ    8000H     ;32K image fifo
CURDEFERREDBUFFER_SIZE equ    400H      ;1K cursor defer fifo
DSPDEFERREDBUFFER_SIZE equ    400H      ;1K display defer fifo
CURSORBUFFER_SIZE      equ    0800H     ;2K cursor image buffer
ICONBUFFER_SIZE        equ    0800H     ;2K icon image buffer
ECMDFIFOSIZE           equ    2         ;encoded size - 16K
EIMGFIFOSIZE           equ    3         ;encoded size - 32K

CMD3DFIFO_OFFSET       equ    0
IMGFIFO_OFFSET         equ    CMD3DFIFO_OFFSET+CMD3DFIFO_SIZE
CMDFIFO_OFFSET         equ    IMGFIFO_OFFSET+IMGFIFO_SIZE
CURDEFERRED_OFFSET     equ    CMDFIFO_OFFSET+CMDFIFO_SIZE
DSPDEFERRED_OFFSET     equ    CURDEFERRED_OFFSET+CURDEFERREDBUFFER_SIZE
CURSOR_OFFSET         equ    DSPDEFERRED_OFFSET+DSPDEFERREDBUFFER_SIZE
ICON_OFFSET           equ    CURSOR_OFFSET+CURSORBUFFER_SIZE

; Rendering Engine Control/Status registers
RenderEngineStatus     equ    008F0h    ;rendering engine status
RenderEngineControl    equ    008F1h    ;rendering engine control
Status2D               equ    08F0H     ;2D engine status
BUSY2D                 equ    010B      ;2D engine is busy executing
RUNNING2D              equ    001B      ;2D engine is running (not stopped)

Control2D              equ    08F1H     ;2D engine control register
RESET2D                equ    01000000B ;Reset 2D engine
SINGLESTEP2D           equ    000000010B ;Executes next command and stops
START2D               equ    000000001B ;Starts 2D engine
STOP2D                equ    00000000B  ;Stops 2D

```

```

; Interrupts
InterruptStatus      equ    000F0H      ;Interrupt status
InterruptMask        equ    000F1H      ;Global interrupt mask
I_DISPLAY            equ    10000000B   ;Display 0 Interrupt
I_DISPLAY1           equ    01000000B   ;Display 1 interrupt
I_VideoCapture       equ    00100000B   ;Video Capture interrupt
I_PeripheralIO       equ    00010000B   ;Peripheral IO interrupt
I_HostDMA            equ    00000100B   ;Host DMA interrupt

; Power Management
OutPutEnablePM       equ    000E0H      ;Pwr Mgmnt. Output enable
EXTMEMBUS            equ    10000000B   ;Ext. Mem Bus
VAFCPORT             equ    01000000B   ;VAFC port
PERIPHIO             equ    00100000B   ;Peripheral IO port

; Surface Descriptor Registers
NEXTSURFACE          equ    010H        ;multiplier to index into next
                                     surface registers
SurfaceBaseAddress    equ    00A00H      ;Surfaces base address
SurfaceStrideFormat   equ    00A04H      ;Surfaces stride format base
S0BaseAddress         equ    00A00H      ;Surface 0 Base address
S0StrideFormat        equ    00A04H      ;Surface 0 Stride and format
S1BaseAddress         equ    00A10H      ;Surface 1 Base address
S1StrideFormat        equ    00A14H      ;Surface 1 Stride and format
S2BaseAddress         equ    00A20H      ;Surface 2 Base address
S2StrideFormat        equ    00A24H      ;Surface 2 Stride and format
S3BaseAddress         equ    00A30H      ;Surface 3 Base address
S3StrideFormat        equ    00A34H      ;Surface 3 Stride and format
S4BaseAddress         equ    00A40H      ;Surface 4 Base address
S4StrideFormat        equ    00A44H      ;Surface 4 Stride and format
S5BaseAddress         equ    00A50H      ;Surface 5 Base address
S5StrideFormat        equ    00A54H      ;Surface 5 Stride and format
S6BaseAddress         equ    00A60H      ;Surface 6 Base address
S6StrideFormat        equ    00A64H      ;Surface 6 Stride and format
S7BaseAddress         equ    00A70H      ;Surface 7 Base address
S7StrideFormat        equ    00A74H      ;Surface 7 Stride and format
S8BaseAddress         equ    00A80H      ;Surface 8 Base address
S8StrideFormat        equ    00A84H      ;Surface 8 Stride and format
S9BaseAddress         equ    00A90H      ;Surface 9 Base address
S9StrideFormat        equ    00A94H      ;Surface 9 Stride and format
SABaseAddress         equ    00AA0H      ;Surface A Base address
SAStrideFormat        equ    00AA4H      ;Surface A Stride and format
SBBaseAddress         equ    00AB0H      ;Surface B Base address
SBStrideFormat        equ    00AB4H      ;Surface B Stride and format
SCBaseAddress         equ    00AC0H      ;Surface C Base address
SCStrideFormat        equ    00AC4H      ;Surface C Stride and format
SDBaseAddress         equ    00AD0H      ;Surface D Base address
SDStrideFormat        equ    00AD4H      ;Surface D Stride and format
SEBaseAddress         equ    00AE0H      ;Surface E Base address
SEStrideFormat        equ    00AE4H      ;Surface E Stride and format
SFBaseAddress         equ    00AF0H      ;Surface F Base address

```

```

SFStrideFormat          equ    00AF4H          ;Surface F Stride and format

CURSOR0SURFACEINDEX    equ    0
CURSOR1SURFACEINDEX    equ    1
SCALER0SURFACEINDEX    equ    2
SCALER1SURFACEINDEX    equ    3
DISPLAYSURFACEINDEX    equ    4
FIRSTSURFACEINDEX      equ    5
LASTSURFACEINDEX       equ    0Eh
DISPLAY1SURFACEINDEX   equ    0Eh
ICONSURFACEINDEX       equ    0Fh

CURSOR0SURFACE equ SurfaceBaseAddress +(CURSOR0SURFACEINDEX* NEXTSURFACE)
CURSOR1SURFACE equ SurfaceBaseAddress +(CURSOR1SURFACEINDEX* NEXTSURFACE)
SCALER0SURFACE equ SurfaceBaseAddress +(SCALER0SURFACEINDEX* NEXTSURFACE)
SCALER1SURFACE equ SurfaceBaseAddress +(SCALER1SURFACEINDEX* NEXTSURFACE)
DISPLAYSURFACE equ SurfaceBaseAddress +(DISPLAYSURFACEINDEX* NEXTSURFACE)
FIRSTSURFACE   equ SurfaceBaseAddress +(FIRSTSURFACEINDEX* NEXTSURFACE)
LASTSURFACE    equ SurfaceBaseAddress +(LASTSURFACEINDEX* NEXTSURFACE)
DISPLAY1SURFACE equ SurfaceBaseAddress +(DISPLAY1SURFACEINDEX* NEXTSURFACE)
ICONSURFACE    equ SurfaceBaseAddress +(ICONSURFACEINDEX* NEXTSURFACE)

BPP1          equ    1
BPP4          equ    3
BPP8          equ    4
BPP16         equ    5
BPP15         equ    5
BPP24         equ    6

CHANNELCUR    equ    0
CHANNELICON   equ    1
CHANNEL0      equ    2
CHANNEL1      equ    3
CHANNEL2      equ    4

; Display Channel parameters
ChannelOffsets equ    00B00H          ;Offsets register base
ChannelIndex   equ    00B04H          ;Index register base
ChCurOffsets  equ    00B00H          ;Cursor Channel display X,Y offsets
ChCurSIndex   equ    00B04H          ;Cursor Channel attached surface
index

ChIconOffsets  equ    00B10H          ;Control Channel display X,Y offsets
ChIconSIndex   equ    00B14H          ;Control Channel attached surface
index

Ch0Offsets     equ    00B20H          ;Channel 0 display X,Y offsets
Ch0SIndex      equ    00B24H          ;Channel 0 attached surface index
Ch1Offsets     equ    00B30H          ;Channel 1 display X,Y offsets
Ch1SIndex      equ    00B34H          ;Channel 1 attached surface index
Ch2Offsets     equ    00B40H          ;Channel 2 display X,Y offsets
Ch2SIndex      equ    00B44H          ;Channel 3 attached surface index

```

```

Ch0UOffsets      equ    00BB0H      ;Channel 0 U offsets
Ch0VOffsets      equ    00BC0H      ;Channel 0 V offsets
ChAudSIndex      equ    00BA4H      ;Audio Channel Playback Surface Index
ChVidCapSIndex   equ    00BE4H      ;Video Capture Channel Surface Index

```

; Buffer Descriptor Registers

```

CMDBufferBaseAddress  equ    00C00H      ;Command Buffer Base/Size
CMDBufferWritePtr     equ    00C04H      ;Command buffer Write pointer
CMDBufferReadPtr      equ    00C08H      ;Command buffer Read pointer
IMGBufferBaseAddress  equ    00C10H      ;Image data Buffer Base/Size
IMGBufferWritePtr     equ    00C14H      ;Image buffer Write pointer
IMGBufferReadPtr      equ    00C18H      ;Image buffer Read pointer
CmdFIFOStatus         equ    00C0CH      ;Command Parameter FIFO status
ImgDataFIFOStatus     equ    00C1CH      ;Image Data FIFO status
FIFOMASK              equ    0FH        ;
FIFOFULL              equ    0          ;Fifo full
FIFO1_8TH             equ    1          ;Fifo 1/8 empty
FIFO1_4TH             equ    2          ;Fifo 1/4 empty
FIFO3_8TH             equ    3          ;Fifo 3/8 empty
FIFO1_2TH             equ    4          ;Fifo 1/2 empty
FIFO5_8TH             equ    5          ;Fifo 5/8 empty
FIFO3_4TH             equ    6          ;Fifo 3/4 empty
FIFO7_8TH             equ    7          ;Fifo 7/8 empty
FIFOEMPTY             equ    8          ;Fifo empty
CursorRecordPtr       equ    00C24H      ;Cursor Record pointer - Write
                        pointer
CursorReplayPtr       equ    00C28H      ;Cursor Replay pointer - Read pointer
DisplayRecordPtr      equ    00C34H      ;Display Record pointer - Write
                        pointer
DisplayReplayPtr      equ    00C38H      ;Display Replay pointer - Read
                        pointer

DMALocalMemAddPtr     equ    00C44H      ;Host DMA Local Memory Address
                        Pointer
CursorReplayStsReg    equ    00CF0H      ;Cursor Replay status
DisplayReplayStsReg   equ    00CF4H      ;Display Replay status
REPLAYPENDING         equ    010000000B
REPLAYACTIVE          equ    01B        ;Deferred register update is active
CursorReplayCtlReg    equ    00CF0H      ;Cursor Replay control(R/W)
DisplayReplayCtlReg   equ    00CF4H      ;Display Replay control(R/W)
ENBLEReplay          equ    00000001B    ;Enable replay of deferred regis-
                        ter writes

; Display Control
ChannelFormat         equ    00D00H      ;Channel format
CursorFormat          equ    00D00H      ;Cursor Channel Format
CURSOROFF             equ    0          ;
CURSOR1               equ    00010000B    ;enable mono cursor
CURSOR16              equ    01010000B    ;enable color cursor
IconChFormat          equ    00D10H      ;Icon Control Channel Format
Ch0Format             equ    00D20H      ;Scaler channel(0) format
Ch0YRControls         equ    00D24H      ;Scaler channel(0) controls YR

```

Ch0UVGBControls	equ	00D28H	;Scaler channel(0) controls UV/GB
INC_FRAC_BIT	equ	11	;Fractional component, LS bits
Ch0Controls	equ	00D2CH	;Scaler channel(0), Increment values
XNEAREST	equ	00000H	;X - Nearest neighbor
XBILINEAR	equ	08000H	;X - Bilinear
YNEAREST	equ	00000H	;Y - Nearest neighbor
YBILINEAR	equ	080000000H	;Y - Bilinear
Ch1Format	equ	00D30H	;Channel 1 format
Ch2Format	equ	00D40H	;Channel 2 format
CLUT4	equ	(BPP4 shl 4)+1	;4 BPP color lookup
CLUT8	equ	(BPP8 shl 4)+1	;8 BPP color lookup
RGB555	equ	(BPP16 shl 4)	;RGB 5-5-5
RGB565	equ	(BPP16 shl 4)+1	;RGB 5-6-5
RGB888	equ	(BPP24 shl 4)	;RGB 8-8-8
YUV420	equ	(BPP8 shl 4)+8	;8 Bit YUV 4-2-0 Planar MPEG
YUV9	equ	(BPP8 shl 4)+9	;8 Bit YUV 9 - Indeo
YUV422	equ	(BPP16 shl 4)+8	;16 Bit YUV 4-2-2
YVU422	equ	(BPP16 shl 4)+9	;16 Bit YVU 4-2-2
OverlayPriority	equ	00D80H	;Overlay priority control
Ch0ChromaCmp	equ	00D84H	;Channel 0, chroma comparator
Dsp0ColCmp	equ	00D88H	;Display0 color comparator
Dsp1ColCmp	equ	01D88H	;Display1 color comparator
Dsp0MixCtl	equ	00D90H	;Mix control between Display0
Dsp1MixCtl	equ	01D90H	;Mix control between Display1
ENABLECTL	equ	0001000000000000B	;Enable control surface
DISABLECTL	equ	0000000000000000B	;Enable control surface
ENABLECOLKEY	equ	0000100000000000B	;Enable color keying
DISABLECOLKEY	equ	0000000000000000B	;Enable color keying
KEYINVERT	equ	0000010000000000B	;Key sense inverted selects Dest.(Back) channel
KEYNORMAL	equ	0000000000000000B	;Key sense normal selects Src.(Fore) channel
KEYSRCBACK	equ	0000001000000000B	;Key source Dest. (Back) channel
KEYSRCFORE	equ	0000000000000000B	;Key source . (Fore) channel
BLENDEnable	equ	0000000100000000B	;Enable blending
BLENDDisable	equ	0000000000000000B	;Disable blending
AudioChFormat	equ	00DE0H	;Audio Playback channel format
AUDIOPLAY	equ	0000000B	;Audio Play
AUDIOMUTE	equ	0100000B	;Audio mute
AUDIOPLAYENA	equ	0010000B	;Audio enabled at next display HSYNC
AUDIOPLAYDISA	equ	0000000B	;Audio disabled at next display HSYNC
AUDIOBSIZE256	equ	00000000B	;Audio buffer size 256 bytes
AUDIOBSIZE512	equ	00000001B	;Audio buffer size 256 bytes
AudioChClkControlseq	equ	00DE4H	;Serial Audio port clock controls
DispIntFlag	equ	00DF0H	;Display Interrupt Flag
DISPVBLANKINT	equ	001B	;Display Vblank Interrupt occurred
EXTVIDEOSYNCINT	equ	010B	;External Video Sync interrupt occurred

```

DispIntEnable      equ    00DF1H      ;MAXH only, SM3110 merged to F1H
DisplayStatus     equ    00DF2H      ;
ModeCtlReg        equ    00DF4H      ;Mode control Register
TIMEVGACRTC      equ    00000000B    ;Timing - VGA CRTC
TIMEENHCRTC      equ    00000100B    ;Timing - Enh. CRTC
TIMEENHVGA       equ    00001000B    ;Timing - Enh. CRTC genlock VGA CRTC
TIMEENHEXT       equ    00001100B    ;Timing - Enh. CRTC genlock with Exter-
                                nal Sync
DISPENAVGA       equ    00000000B    ;Display Output - Enable VGA
DISPENAENH       equ    00000001B    ;Display Output - Enable Enhanced
DISPVGAENH       equ    00000010B    ;Display Output - VGA and Enh
DISPDISABLED     equ    00000011B    ;Display Output - disabled
SpDisplayControl  equ    00DF5H      ;Special Display Control
EnhSyncPolarityRegequ 00DF6H      ;Enh Sync Polarity Register
HSYNCPOS         equ    001B         ;
VSYNCPOS         equ    010B         ;
SyncLevelReg     equ    00DF7H      ;Sync levels for power management
SYNCSNORMAL      equ    0           ;
HSYNCSET0        equ    00001B      ;Force Hsync to 0
HSYNCSET1        equ    00010B      ;Force Hsync to 1
VSYNCSET0        equ    00100B      ;Force Vsync to 0
VSYNCSET1        equ    01000B      ;Force Vsync to 1
ScalerFifoSize   equ    00DF8H      ;Display Scaler FIFO Size
ScalerFifoThresholdequ 00DFAH      ;Display Scaler FIFO Threshold
PaletteRWState   equ    00DFCH      ;Palette Read/Write status
;
; Display Timing
ChViewPositionStarts  equ    00E00H    ;Channel View position starts
ChViewPositionEnds    equ    00E04H    ;Channel View position ends
CursorPositionStarts  equ    00E00H    ;Cursor Channel Viewport posi-
                                tion top left
CursorPositionEnds    equ    00E04H    ;Cursor Channel Viewport extents
                                bot. right
IconChPositionStarts  equ    00E10H    ;Icon Control Channel Viewport
                                position top left
IconChPositionEnds    equ    00E14H    ;Icon Control Channel Viewport
                                extents bot right
Ch0ViewPositionStarts  equ    00E20H    ;Channel 0 Viewport position top
                                left
Ch0ViewPositionEnds    equ    00E24H    ;Channel 0 Viewport extents bot
                                right
Ch1ViewPositionStarts  equ    00E30H    ;Channel 1 Viewport position top
                                left
Ch1ViewPositionEnds    equ    00E34H    ;Channel 1 Viewport extents bot
                                right
Ch2ViewPositionStarts  equ    00E40H    ;Channel 2 Viewport position top
                                left
Ch2ViewPositionEnds    equ    00E44H    ;Channel 2 Viewport extents bot
                                right
VerticalCounter      equ    00E9AH      ;Vertical counter - Current scan-
                                line displayed
    
```



```

FLAGS          equ    0000000000000001000000000100000B    ;Flags Register
SURFACEBaseIndex equ    0000000000000001000000000100100B    ;Src. Dest. Base
                                                    indices
FGColor        equ    0000000000000001000000000101000B    ;Foreground
                                                    Color
BGColor        equ    0000000000000001000000000101100B    ;Background
                                                    Color
TRANSColor     equ    0000000000000001000000000110000B    ;Transparent
                                                    Color

; FLAGS Register Definition
;
; 10987654321098765432109876543210
F_BYTE         equ    0000000000000000000000000000000000000B ;Image data byte aligned
F_WORD         equ    00000000010000000000000000000000000000B ;Image data word aligned
F_DWORD        equ    00000000100000000000000000000000000000B ;Image data dword aligned
F_BIT          equ    00000000110000000000000000000000000000B ;Image data bit aligned
                                                    ;- valid with mono data
                                                    only
F_XN           equ    00000000010000000000000000000000000000B ;X decreasing - origin at
                                                    top left
F_XP           equ    00000000000000000000000000000000000000B ;X increasing - left to
                                                    right
F_YN           equ    000000000010000000000000000000000000000B ;Y decreasing
F_YP           equ    0000000000000000000000000000000000000000B ;Y increasing - top to bot-
                                                    tom
                                                    ; If Bit 12 is 0 - Mono
                                                    Source
F_MPT          equ    00000000000001000000000000000000000000B ;Mono Blt - pattern
                                                    transparent
F_MNORMAL      equ    000000000000000000000000000000000000000B ;Mono Blt - color expand
                                                    ;1==Foreground, 0==Back-
                                                    ground
F_MBT          equ    00000000000000010000000000000000000000B ;Mono Blt Transparency
                                                    ; - Background transparent
F_MFT          equ    00000000000000100000000000000000000000B ;Mono Blt Transparency
                                                    ; - Foreground transparent
F_MINVERT      equ    00000000000000110000000000000000000000B ;Mono Blt - color expand
                                                    ;0==Foreground, 1==Back-
                                                    ground
                                                    ; If Bit 12 is 1 - Color
                                                    Source
F_TCINVERT     equ    00000000000001000000000000000000000000B ;Invert color transparency
F_TCOPAQUE     equ    000000000000000000000000000000000000000B ;No color transparency

```

F_TCSRC	equ	00000000000000 <u>01</u> 000000000000000B;Color Source transparency
F_TCDEST	equ	00000000000000 <u>10</u> 000000000000000B;Color Destination transparency
F_TCPAT	equ	00000000000000 <u>11</u> 000000000000000B;Color Pattern transparency
F_CLIP	equ	00000000000000 <u>01</u> 000000000000000B;Enable clipping
F_NOCLIP	equ	00000000000000 <u>00</u> 000000000000000B;Disable clipping
F_USEPATTERN	equ	00000000000000 <u>00</u> 000000000000000B;Use pattern buffer
F_USEBGCOLOR	equ	00000000000000 <u>01</u> 000000000000000B;Use Background color for pattern
F_LOCALMEM	equ	00000000000000 <u>00</u> 000000000000000B;Blt Src Local memory
F_IMGDATA	equ	00000000000000 <u>01</u> 000000000000000B;Blt Src System memory
F_MONO	equ	00000000000000 <u>00</u> 000000000000000B;Source Format Monochrome
F_COLOR	equ	00000000000000 <u>01</u> 000000000000000B;Source Format Color
CMD_BITBLT	equ	00000000000000000000 <u>01</u> 000000000B;Bitblt command
CMD_LINE	equ	00000000000000000000 <u>01</u> 000000000B;Line command
CMD_MEMTEST	equ	00000000000000000000 <u>10</u> 000000000B;Memory test command
F_ROPMASK	equ	00000000000000000000 <u>11111111</u> B;ROP mask

6.14.2 Macros

The following macro is used to generate deferred an address value for its immedate register. The register address must be ORed with the physical screen base address and written to the deferred port data area followed by the register data (both values must be doubleword). These writes are recorded into the deferred buffer and immedate registers are written with this data during VBLANK (replay).

Load Deferred Register Address with DWORD

```
LDAD      MACRO  reg, labl
           mov    reg, PhyCmdPortAddress
           add    reg, labl
           ENDM
```

6.14.3 Clock Synthesizer Programming Guide

There are three Phase-Locked Loop-based programmable clock synthesizers in SM3110 for the two display clocks (DCLK and PCLK) and one memory/2D/3D engine clock (MCLK) from an externally supplied reference frequency. Each of the output frequencies may be programmed to up to 400 MHz. An external, crystal-controlled oscillator generates the reference frequency (typically 14.31818 MHz) that is driven into the SM3110 on pin Y22. Alternatively, an external crystal can be connected between pins Y22 and W23 to generate the reference frequency. All three PLL synthesizers in the SM3110 are the same design.

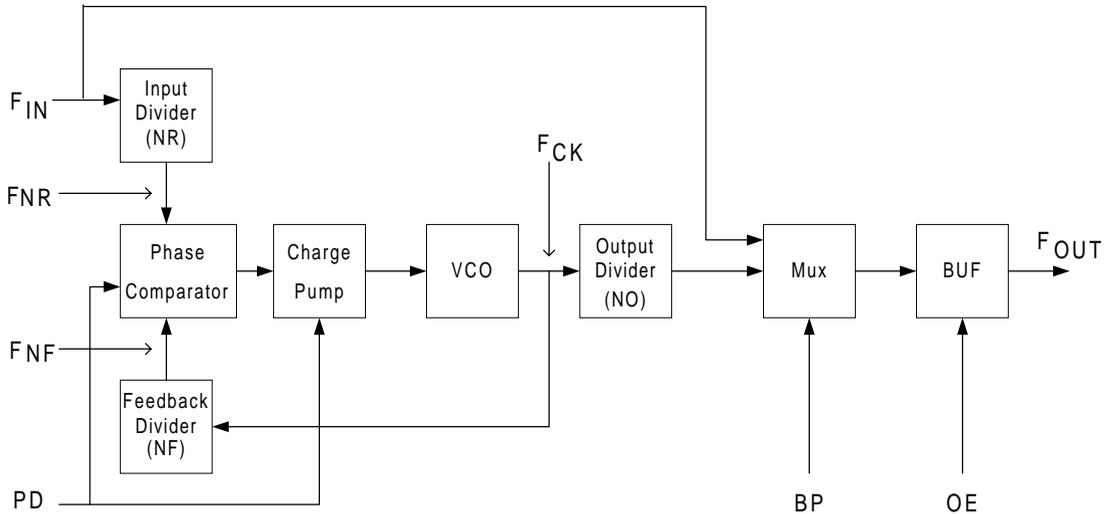


Figure 6-18. PLL Block Diagram

6.14.3.1 Theory of Operation

The Phase Comparator compares the phase difference of the input clocks from Input Divider (or Reference Divider), F_{NR} , and Feedback Divider (or Loop Divider), F_{NF} . If the edge of F_{NR} arrives earlier than the edge of F_{NF} , the output of the comparator adjusts the Charge Pump to provide a higher voltage to VCO, which increases the frequency of F_{CK} and brings the edge of F_{NF} earlier. If F_{NR} is later than F_{NF} , the Charge Pump decreases the voltage supplied to the VCO, which decreases frequency of F_{CK} and delays the edge of F_{NF} . With both edges aligned all the time, their frequencies must be the same, too.

With programmable Input and Feedback Dividers, the F_{NR} and F_{NF} can be at the same frequency while F_{IN} and F_{CK} can be different frequencies. This means the SM3110 can provide a wide range of output frequencies F_{CK} with a fixed input reference frequency F_{IN} .

The Output Divider provides a wider output frequency range for F_{OUT} while it keeps the VCO running at narrower, optimized range. Without this divider, F_{OUT} would have to have the same frequency range as F_{CK} .

These clock synthesizers have built-in power management which is controlled by signal PD. It can turn off the clock circuit to save power when not in use. The output buffer can also be turned off with signal OE.

The BP signal controls a muxing bypass circuit. When the BP signal is set to bypass mode, the output F_{OUT} and input F_{IN} have the same frequency.

6.14.3.2 Registers

Two registers control each clock synthesizer. In the first register, two byte specify the three (reference, feedback loop and output) dividers. In the second register, one byte controls the output and the PLL/VCO power down.

6.14.3.3 Frequency Control for Memory/Dot/Panel Clock

<i>Index</i>	<i>Field</i>	<i>Access</i>	<i>Function</i>
+080H	15:00	R/W	MCLK Loop Control
+0090H	15:00	R/W	DCLK Loop Control
+1090H	15:00	R/W	DCLK Loop Control
	<i>Bits</i>	<i>Field</i>	
	15:14	Output Divider	
		<i>Value</i>	<i>Semantics</i>
		0	NO has no division
		1	NO is divide by 2
		2	NO is divide by 2
		3	NO is divide by 4
	<i>Bits</i>	<i>Field</i>	
	13:09	Reference Divider	
		<i>Value</i>	<i>Semantics</i>
		0	Reserved
		1-31	Divide ratio (NR) is this value plus two
	<i>Bits</i>	<i>Field</i>	
	08:00	Feedback Loop Divider	
		<i>Value</i>	<i>Semantics</i>
		0	Reserved
		1-511	Divide ratio (NF) is this value plus two

The actual divide values for NR and NF are the binary values programmed in this register, bit 13 to 9 for NR and bit 8 to 0 for NF, plus two. For example, if the binary value in bit 13 to 9 is 10011 (decimal 19), the actual dividing value for Input Divider is decimal 21. If the binary value in bit 8 to 0 is 011001010 (decimal 202), the actual dividing value for Feedback Divider is decimal 204. The actual divide value for NO is specified in the register definition.

6.14.3.4 VCO Control for Memory Clock Synthesizer

Index	Field	Access	Function
+082H	07:00	R/W	MCLK VCO Control
	Bits	Field	
	07	PLL Module Internal Bypass Control (BP)	
		Value	Semantics
		0	Disabled. <i>Default</i> , normal operation.
		1	Enabled.
	Bits	Field	
	06	PLL Module Internal power down Control (PD)	
		Value	Semantics
		0	Disabled <i>Default</i> , normal operation.
		1	Enabled. PLL is in power down mode.
	Bits	Field	
	05	Clock Tree Source Select (Set by level of strapping input from ROMA1. Input pulled up to VDD.)	
		Value	Semantics
		0	External Clock
		1	Internal PLL Clock. <i>Default</i> .
	Bits	Field	
	04	Load PLL Counter to Working Register	
		Value	Semantics
		0	Normal Operation. <i>Default</i> .
		1	Load Register. Automatically returns to 0 after loading working register.
	Bits	Field	
	03:02	Reserved	
	01	PLL Module Internal output enable Control (OE)	
		Value	Semantics
		0	Enabled
		1	Disabled
	Bits	Field	
	00	Clock Tree Source Enable	
		Value	Semantics
		0	Disabled. MCLK forced to '0'.
		1	Enabled. MCLK running.

6.14.3.5 VCO Control for Dot/Panel Clock Synthesizer

Index	Field	Access	Function
+0092H	07:00	R/W	DCLK VCO Control (CRT)
+1092H	07:00	R/W	DCLK VCO Control (LCD)
	<i>Bits</i>	<i>Field</i>	
	07		PLL Module Internal Bypass Control (BP)
		<i>Value</i>	<i>Semantics</i>
		0	Disabled. <i>Default</i> , normal operation.
		1	Enabled.
	<i>Bits</i>	<i>Field</i>	
	06		PLL Module Internal power down Control (PD)
		<i>Value</i>	<i>Semantics</i>
		0	Disabled <i>Default</i> , normal operation.
		1	Enabled. PLL is in power down mode.
	<i>Bits</i>	<i>Field</i>	
	05		Clock Tree Source Select (Set by level of strapping input from ROMA1. Input pulled up to VDD.)
		<i>Value</i>	<i>Semantics</i>
		0	External Clock
		1	Internal PLL Clock. <i>Default</i> .
	<i>Bits</i>	<i>Field</i>	
	04		Load PLL Counter to Working Register
		<i>Value</i>	<i>Semantics</i>
		0	Normal Operation. <i>Default</i> .
		1	Load Register. Automatically returns to 0 after loading working register.
	<i>Bits</i>	<i>Field</i>	
	03		PLL Module Internal output enable Control (OE)
		<i>Value</i>	<i>Semantics</i>
		0	Enabled
		1	Disabled

VCO Control for Dot/Panel Clock Synthesizer (continued)

Bits	Field						
02	Reference Internal Oscillator Pad Control (Power Down)						
	<table border="1"> <thead> <tr> <th>Value</th> <th>Semantics</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Enabled. <i>Default.</i></td> </tr> <tr> <td>1</td> <td>Disabled. shut down oscillator to reduce power dissipation in sleep mode.</td> </tr> </tbody> </table>	Value	Semantics	0	Enabled. <i>Default.</i>	1	Disabled. shut down oscillator to reduce power dissipation in sleep mode.
Value	Semantics						
0	Enabled. <i>Default.</i>						
1	Disabled. shut down oscillator to reduce power dissipation in sleep mode.						
01	Divide Ratio Select						
	<table border="1"> <thead> <tr> <th>Value</th> <th>Semantics</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>VGA DCLK Frequencies, VGA Mode. <i>Default.</i> Use divide ratio specified in either 94 95H or 96 97H selected by 3CCH[3:2].</td> </tr> <tr> <td>1</td> <td>Programmable DCLK Frequencies, Enhanced Mode. Use divide ratio specified in F0 F1H.</td> </tr> </tbody> </table>	Value	Semantics	0	VGA DCLK Frequencies, VGA Mode. <i>Default.</i> Use divide ratio specified in either 94 95H or 96 97H selected by 3CCH[3:2].	1	Programmable DCLK Frequencies, Enhanced Mode. Use divide ratio specified in F0 F1H.
Value	Semantics						
0	VGA DCLK Frequencies, VGA Mode. <i>Default.</i> Use divide ratio specified in either 94 95H or 96 97H selected by 3CCH[3:2].						
1	Programmable DCLK Frequencies, Enhanced Mode. Use divide ratio specified in F0 F1H.						
00	Clock Tree Source Enable						
	<table border="1"> <thead> <tr> <th>Value</th> <th>Semantics</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>Disabled. DCLK forced to '0'.</td> </tr> <tr> <td>1</td> <td>Enabled. DCLK running.</td> </tr> </tbody> </table>	Value	Semantics	0	Disabled. DCLK forced to '0'.	1	Enabled. DCLK running.
Value	Semantics						
0	Disabled. DCLK forced to '0'.						
1	Enabled. DCLK running.						

6.14.3.6 Selecting Register Value

The output frequency of VCO, F_{CK} , is determined by following equation:

$$F_{CK} = F_{in} * (NF / NR)$$

Where F_{CK} is the output frequency of VCO, F_{in} is the input reference frequency, NR is the input divider value, and NF is the feedback divider value.

The output frequency is determined by following equation:

$$F_{out} = \frac{F_{in} * NF}{NR * NO}$$

Where F_{out} is the output frequency, F_{in} is the input reference frequency, NR is the input divider value, NF is the feedback divider value and NO is the output divider value.

6.14.3.7 Programming Limitations

The synthesizer is very flexible. It doesn't have any special requirement of the programming order. However, VCO output frequency, F_{CK} , should be kept in its mid frequency range, which is about 200 MHz.

Bit 4 of the 082H, 0092H and 1092H registers is used to load the divider value into each PLL. These registers should be the last ones programmed when changing any divider values.

6.14.3.8 Register Values for Common Frequencies

The table below shows most of the frequencies required by VESA defined modes. This table is shown only as a programming example. There may be other register values not shown which can be used to achieve the same frequency. Please consult the equations in the earlier section for programming values. This table is based on reference input clock FIN at 14.31818 MHz.

Resolution	Refresh Rate	NF Register Value	NR Register Value	NO Register Value	VCO Freq.	Actual Freq.	Pixel Clock Freq.	% Error
640x480	60	216	29	3	100.7	25.17	25.175	-0.02
640x480	75	42	3	3	126	31.5	31.5	0
640x480	56	169	15	3	144.04	36.01	36	0.03
800x600	60	188	15	3	160.04	40.01	40	0.02
800x600	75	247	16	3	198.08	49.52	49.5	0.04
800x600	72	431	29	3	200	50	50	0
800x600	85	438	26	3	225	56.25	56.25	0
1024x768	60	343	17	3	260	65	65	0
1024x768	70	218	19	2	150	75	75	0
1024x768	75	31	1	2	157.5	78.75	78.75	0
1024x768	85	64	3	2	189	94.5	94.5	0
1280x1024	60	345	21	2	216.02	108.01	108	0.01
1280x1024	75	130	5	2	270	135	135	0
1280x1024	85	64	1	2	315	157.5	157.5	0
1600x1200	60	179	6	2	323.96	161.97	162	-0.02
1600x1200	65	96	2	2	350.8	175.4	175.5	-0.06
1600x1200	70	130	3	2	378	189	189	0
1600x1200	75	97	5	0	202.5	202.5	202.5	0
1600x1200	85	318	18	0	229.1	229.09	229.5	-0.18

