

®

inmos®

XGA SOFTWARE PROGRAMMER'S GUIDE

First Edition 1991

The information contained in this 1st edition of the *XGA Software Programmer's Guide* (document revision 00) is preliminary information. A 2nd edition is in preparation.

ST **SGS-THOMSON**
MICROELECTRONICS

INMOS is a member of the SGS-THOMSON Microelectronics Group


Other XGA documents

IMS G190 XGA serializer palette DAC, Preliminary information,
INMOS document number 42 1526 01

IMS G200 XGA display controller, Preliminary information,
INMOS document number 42 1525 01

Copyright © INMOS Limited 1991

INMOS reserves the right to make changes in specifications at any time and without notice. The information furnished by INMOS in this publication is believed to be accurate; however, no responsibility is assumed for its use, nor for any infringement of patents or other rights of third parties resulting from its use. No licence is granted under any patents, trademarks or other rights of INMOS.

 **inmos**, IMS and occam are trademarks of INMOS Limited.



is a registered trademark of SGS-THOMSON Microelectronics Group.

IBM, PS/2, and Micro Channel are registered trademarks of IBM.

XGA is a trademark of IBM licensed to SGS-THOMSON.

INMOS is a member of the SGS-THOMSON Microelectronics Group.

INMOS document number: 72 OEK 258 00

ORDER CODE: PMXGASOFT/1

Printed in Italy

Contents overview

Contents	v
Preface	xiii
XGA function	1
1 XGA Overview	3
2 VGA	8
3 132 Column Text	9
4 Extended Graphics	10
5 XGA System Interface	83
XGA programming considerations	93
6 Adapter Co-existence	95
7 Locating the XGA Subsystem	96
8 VGA Primary Adapter Considerations	100
9 General Systems Considerations	102
10 Extended Graphics Modes Selection	103
11 Mode Setting the XGA Subsystem	105
12 Upwards Compatibility	115
13 Programming the XGA Subsystem in Extended Graphics Mode	116
14 Other Programming Considerations	131
15 Sample Code	134

Contents

Preface	xiii
XGA function	1
1 XGA Overview	3
1.1 Major Components	4
1.1.1 System Bus Interface	5
1.1.2 Memory and CRT Controller	5
1.2 Coprocessor	5
1.2.1 Video Memory	5
1.3 Attribute Controller	5
1.4 Sprite Controller	6
1.5 The Serializer, Palette and DAC	6
1.6 A/N Font and Sprite Buffer	6
1.7 Modes Of Operation	6
1.8 Compatibility	6
1.8.1 8514/A	6
1.8.2 LIM EMS Drivers	7
2 VGA	8
3 132 Column Text	9
4 Extended Graphics	10
4.1 Display Controller Description	10
4.1.1 Video Memory Format	10
4.1.2 Pixel Color Mapping	11
4.1.3 Border Color Mapping	11
4.1.4 Direct Access to the Video Memory	11
System Apertures Into Video Memory	11
4.1.5 CRT Controller	12
CRTC Register Interpretations	12
Scrolling	13
4.1.6 Sprite	14
Sprite Color Mapping	14
Sprite Buffer Accesses	14
Sprite Positioning	15
4.1.7 Palette	16
Palette Accesses	16
4.2 Direct Color Mode	17
Coprocessor Functions	18
4.3 Display Controller Registers	18
4.3.1 Register Usage Guidelines	19
4.3.2 Direct Access I/O Registers	19
Operating Mode Register (Address: 21x0)	19
Aperture Control Register (Address: 21x1)	20
Interrupt Enable Register (Address: 21x4)	21
Interrupt Status Register (Address: 21x5)	21

	Virtual Memory Control Register (Address: 21x6)	21
	Virtual Memory Interrupt Status Register (Address: 21x7)	21
	Aperture Index Register (Address: 21x8)	22
	Memory Access Mode Register (Address: 21x9)	22
	Index Register (Address: 21xA)	23
	Data Registers (Addresses: 21xB to 21xF)	24
4.3.3	Indexed Access I/O Registers	24
	Auto-Configuration Register (Index: 04)	24
	Coprocessor Save/Restore Data Registers (Index: 0C & 0D)	24
	Horizontal Total Registers (Index: 10 & 11)	24
	Horizontal Display End Registers (Index: 12 & 13)	25
	Horizontal Blanking Start Registers (Index: 14 & 15)	25
	Horizontal Blanking End Registers (Index: 16 & 17)	26
	Horizontal Sync Pulse Start Registers (Index: 18 & 19)	26
	Horizontal Sync Pulse End Registers (Index: 1A & 1B)	27
	Horizontal Sync Pulse Position Registers (Index: 1C & 1E)	27
	Vertical Total Registers (Index: 20 & 21)	28
	Vertical Display End Registers (Index: 22 & 23)	28
	Vertical Blanking Start Registers (Index: 24 & 25)	29
	Vertical Blanking End Registers (Index: 26 & 27)	29
	Vertical Sync Pulse Start Registers (Index: 28 & 29)	30
	Vertical Sync Pulse End Register (Index: 2A)	30
	Vertical Line Compare Registers (Index: 2C & 2D)	31
	Sprite Horizontal Start Registers (Index: 30 & 31)	31
	Sprite Horizontal Preset (Index: 32)	32
	Sprite Vertical Start Registers (Index: 33 & 34)	32
	Sprite Vertical Preset (Index: 35)	33
	Sprite Control Register (Index: 36)	33
	Sprite Color Registers (Index: 38 - 3D)	33
	Display Pixel Map Offset Registers (Index: 40 - 42)	34
	Display Pixel Map Width Registers (Index: 43 & 44)	34
	Display Control 1 Register (Index: 50)	35
	Display Control 2 Register (Index: 51)	36
	Display ID and Comparator (Index: 52)	37
	Clock Frequency Select Register (Index: 54)	37
	Border Color Register (Index: 55)	37
	Sprite/Palette Index Registers (Index: 60 & 61)	38
	Sprite/Palette Index Registers with Prefetch (Index: 62 & 63)	38
	Palette Mask Register (Index: 64)	38
	Palette Data Register (Index: 65)	39
	Palette Sequence Register (Bits 2:0 only) (Index: 66)	39
	Palette Red Prefetch Register (Index: 67)	39
	Palette Green Prefetch Register (Index: 68)	40
	Palette Blue Prefetch Register (Index: 69)	40
	Sprite Data Register (Index: 6A)	40
	Sprite Prefetch Register (Index: 6B)	40
	External Clock Select Register (Index: 70)	41
4.4	Coprocessor Description	42
4.5	Programmer's View	43
4.6	Pixel Formats	44
4.6.1	Pixel Data	44
	Fixed And Variable Data	44
	XGA Function	44

4.6.2	The Coprocessor View of Memory	44
4.6.3	XGA Pixel Maps	44
	Pixel Maps A, B, And C (General Maps)	44
	Pixel Map M (Mask Map)	45
	Map Origin	45
	X and Y Pointers	46
	Scissoring With The Mask Map1	48
4.6.4	Drawing Operations	51
	Draw and Step:	51
	Line Draw:	53
	Pixel Block Transfer (PxBlt):	56
	Area Fill:	58
4.6.5	Logical And Arithmetic Functions	60
	Mixes:	60
	Breaking the ALU Carry Chain:3	61
	Generating The Pattern From The Source:	62
	Color Expansion:	62
	Pixel Bit Masking:4	62
	Color Compare:5	62
4.6.6	Controlling Coprocessor Operations	63
	Starting a Coprocessor Operation:	63
	Suspending a Coprocessor Operation:	63
	Terminating a Coprocessor Operation:	63
4.6.7	Coprocessor Operation Completion	63
	Accesses To The Coprocessor During An Operation:3	64
4.6.8	Coprocessor State Save/Restore	64
	Suspending Coprocessor Operations:4	64
4.6.9	Save/Restore Mechanism	64
4.7	Coprocessor Registers	65
4.7.1	Register Usage Guidelines	68
4.7.2	Virtual Memory Registers	68
	Page Directory Base Address Register (Coprocessor Registers, Offset:0)	68
	Current Virtual Address Register (Coprocessor Registers, Offset: 4)	68
4.7.3	State Save/Restore Registers	69
	Coprocessor Control Register (Offset: 11)	69
	State Length Registers (Offset: C & D)	69
	Save/Restore Data Ports (I/O Index: C & D)	69
4.7.4	Pixel Interface Registers	70
	Pixel Map Index Register (Offset: 12)	70
	Pixel Map n Base Pointer (Offset: 14)	71
	Pixel Map n Width (Offset: 18)	71
	Pixel Map n Height (Offset: 1A)	71
	Pixel Map n Format (Offset: 1C)	72
	Pixel Maps A, B and C	72
	Mask Map	72
	Bresenham Error Term E (Offset: 20)	73
	Bresenham Constant K1 (Offset: 24)	73
	Bresenham Constant K2 (Offset: 28)	73
	Direction Steps Register (Offset: 2C)	74
	Foreground Mix Register (Offset: 48)	74
	Background Mix Register (Offset: 49)	74
	Destination Color Compare Condition (Offset: 4A)	75
	Destination Color Compare Value (Offset: 4C)	75

	Pixel Bit Mask (Plane Mask) (Offset: 50)	75
	Carry Chain Mask (Offset: 54)	76
	Foreground Color Register (Offset: 58)	76
	Background Color Register (Offset: 5C)	76
	Operation Dimension 1 (Offset: 60)	77
	Operation Dimension 2 (Offset: 62)	77
	Mask Map Origin X Offset (Offset: 6C)	77
	Mask Map Origin Y Offset (Offset: 6E)	77
	Source X Address (Offset: 70)	78
	Source Y Address (Offset: 72)	78
	Pattern X Address (Offset: 74)	78
	Pattern Y Address (Offset: 76)	78
	Destination X Address (Offset: 78)	79
	Destination Y Address (Offset: 7A)	79
	Pixel Operations Register (Offset: 7C)	79
5	XGA System Interface	83
5.1	Multiple Instances	83
5.1.1	Multiple XGA Subsystems in VGA Mode	83
5.1.2	Multiple XGA Subsystems in 132 Column Text Mode	83
5.1.3	Multiple XGA Subsystems in Extended Graphics Mode	83
5.2	XGA POS Registers	83
5.2.1	Register Usage Guidelines	83
5.2.2	Subsystem Identification Low Byte (Base + 0)	84
5.2.3	Subsystem Identification High Byte (Base + 1)	84
5.2.4	POS Register 2 (Base + 2)	84
	XGA Enable (EN, Bit 0)	84
	I/O Device Address (IODA, Bits 1–3)	84
	ROM Address (ROM Addr, Bits 4–7)	84
5.2.5	POS Register 4 (Base + 4)	85
	Video Memory Base Address (Bits 7–1)	85
	Video Memory Enable (VE, Bit 0)	85
5.3	POS register 5 (Base + 5)	86
	1 Mbyte Aperture Base Address (1 Mbyte Base, Bits 3–0)	86
5.4	Virtual Memory Description	86
5.4.1	Address Translation	86
	Page Directory and Page Table Entries	87
5.4.2	The XGA Implementation of Virtual Memory	88
	The TLB	88
	TLB Misses	88
	System Coherency	89
	VM Page Not Present Interrupts	89
	VM Protection Violation Interrupts	90
	The XGA in Segmented Systems	90
5.5	Virtual Memory Registers	90
5.5.1	Page Directory Base Address Register (Coprocessor Registers, Offset:0)	90
5.5.2	Current Virtual Address Register (Coprocessor Registers, Offset: 4)	91
5.5.3	Virtual Memory Control Register (I/O Address: 21x6)	91
5.5.4	Virtual Memory Interrupt Status Register (I/O Address: 21x7)	92

XGA programming considerations	93
6 Adapter Co-existence	95
6.1 Co-existence with VGA	95
6.2 Co-existence with Other XGA Subsystems	95
7 Locating the XGA Subsystem	96
7.1 Reading POS Data	96
7.2 Address Calculations	97
7.2.1 ROM address	97
7.2.2 Coprocessor Registers	97
7.2.3 I/O Registers	97
7.2.4 The Video Memory Base Address	97
4 Mbyte System Video Memory Aperture	98
Video Memory Location in Coprocessor Address Space	98
7.2.5 1 Mbyte Aperture Base Address	99
7.3 Display Type and Video Memory Size	99
8 VGA Primary Adapter Considerations	100
8.1 Chaining the Int 10h Video BIOS Handler	100
8.2 Int 24h, Critical Error Handler	100
8.3 Int 23h Ctrl-Break Exit Address	101
8.4 Int 21h Function 4Ch Program Terminate function	101
9 General Systems Considerations	102
9.1 Co-existing with LIM Expanded Memory Managers	102
9.2 Screen Switch Notification, Int 2Fh	102
10 Extended Graphics Modes Selection	103
10.1 Modes Available	103
11 Mode Setting the XGA Subsystem	105
11.1 Individual Mode Setting Procedures	105
11.1.1 Extended Graphics Mode	105
11.1.2 VGA Mode	107
11.1.3 132 Column Text Mode	107
11.2 System Video Memory Apertures	109
11.2.1 64K System Video Memory Aperture	109
11.2.2 1 Mbyte System Video Memory Aperture	109
11.2.3 4 Mbyte System Video Memory Aperture	109
11.3 Physical Addressability to System Memory	110
11.3.1 Real Mode DOS Environments	110
Extended Memory	110
LIM EMS Managers	110
11.3.2 32 bit DOS Extended Environments	111
11.3.3 Multiple Virtual DOS Machine Environments	111

11.3.4	Protect Mode 16 Bit Segmented Environments	112
	64K Segment Limit	112
	Segment Motion	112
	System Overheads	112
	Access to XGA Registers and System Memory Apertures	112
	Suggested Design Model	112
11.3.5	Paged Virtual Memory (VM) Environments	113
	4K Discontiguous Pages	113
	Page Table Coherency	113
	System Overheads	113
	Access to XGA Registers and System Memory Apertures	113
	Suggested Design Model	113
11.3.6	Video Memory Addressability in VM Mode	113
11.3.7	System Memory Access Limitation	114
12 Upwards Compatibility		115
12.1	XGA Subsystem POS ID Allocations	115
12.1.1	General Register Usage	115
12.1.2	Video BIOS Mode 14h	115
12.1.3	PS/2 Video Memory Apertures	115
13 Programming the XGA Subsystem in Extended Graphics Mode		116
13.1	XGA Coprocessor Pixel Interface Registers	116
13.1.1	Pixel Map Index Register (OFFSET 12h)	116
13.1.2	Pixel Map Base Address Register (OFFSET 14h)	116
13.1.3	Pixel Map Width Register (OFFSET 18h)	116
13.1.4	Pixel Map Height Register (OFFSET 20h)	117
13.1.5	Pixel Map Format Register (OFFSET 1Ch)	117
13.1.6	Other Registers	117
13.2	Using the Coprocessor to Perform a Pixel Blit (PxBit)	118
13.2.1	Mixes and Colors	118
	Foreground and Background Mix Registers	119
	Foreground & Background Color Registers	119
13.2.2	PxBit Dimensions	119
13.2.3	Pixel Map, Source & Destination	119
	Source Map X and Y Registers	119
	Destination Map X and Y Registers	119
	Pattern Map X and Y Registers	119
	Mask Map Origin X and Y Offset Registers	120
13.2.4	Pixel Operations Register	120
	Background Source	120
	Foreground Source	121
	Step Function	121
	Source Pixel Map	121
	Destination Pixel Map	121
	Pattern Pixel Map	122
	Mask Pixel Map	122
	Drawing Mode	122
	Direction Octant	122
	Conclusion	123
13.3	Using the Coprocessor to Perform a Bresenham Line Draw	123
13.3.1	Mixes and Colors	124

	Foreground and Background Mix Registers	124
	Foreground and Background Color Registers	124
13.3.2	Bresenham Line Draw	125
	Bresenham Error Term Register	125
	Bresenham Constant K1 Register	125
	Bresenham Constant K2 Register	126
	Operation Dimension Registers	126
13.3.3	Pixel Map, Source and Destination	126
	Source Map X and Y Registers	126
	Destination Map X and Y Registers	126
	Pattern Map X and Y Registers	126
	Mask Map Origin X and Y Offset Registers	126
13.3.4	Pixel Operations Register	127
	Background Source	127
	Foreground Source	127
	Step Function	128
	Source Pixel Map	128
	Destination Pixel Map	128
	Pattern Pixel Map	128
	Mask Pixel Map	129
	Drawing Mode	129
	Direction Octant	129
	Conclusion	130
13.4	Memory Access Modes (Reg. 21x9)	130
13.5	Motorola/Intel Format	130
13.5.1	System Processor Access	130
13.5.2	XGA Coprocessor Accesses	130
13.5.3	Exploitation	130
14	Other Programming Considerations	131
14.1	Overlapping BitBlits	131
14.1.1	Pixel Block Transfer (PxBlt)	131
14.1.2	Inverting PxBlt	131
14.2	Sprite Handling	131
14.2.1	Sprite Loading	131
14.2.2	Sprite Positioning	131
14.3	Waiting for Hardware Not Busy	131
14.4	Destination Bitmap Width Restriction	132
14.5	Line Length Restriction	133
14.6	System Register Usage	133
14.7	Direct Color Mode	133
14.7.1	Palette Loading	133
14.7.2	Coprocessor Support	133
15	Sample Code	134
15.1	Putting the XGA Subsystem into Extended Graphics Mode	134
15.1.1	Pseudo Code	134
15.1.2	Code Example	136
	Main C Program	136
	Assembler Subroutines	149

15.2	Putting the XGA Subsystem into 132 Column Text Mode	151
15.2.1	Pseudo Code	151
15.2.2	Code Example	152
	Main C Program	152
	Assembler Subroutines	160

Preface

The *XGA Software Programmer's Guide* is intended to provide information for programming the XGA subsystem which is implemented in the IMS G190 XGA serializer palette DAC and the IMS G200 XGA display controller.

This guide contains an overview of the XGA architecture, a description of the XGA subsystem function, and information on programming XGA device registers with programming examples. It should be used in conjunction with the following documents:

IMS G190 XGA serializer palette DAC, Preliminary information,
INMOS document number 42 1526 01

IMS G200 XGA display controller, Preliminary information,
INMOS document number 42 1525 01

This 1st edition of the *XGA Software Programmer's Guide* (document revision 00) will be superseded by the 2nd edition which is in preparation.



XGA function

1 XGA Overview

The following features summarize the capabilities of the XGA subsystem.

VGA: When in VGA mode, the XGA subsystem is VGA register compatible as defined in the 'VGA Function' chapter of the 'Video Subsystem' section in the 'PS/2 Hardware Interface Technical Reference'.

132 Column Text: In this mode, text is displayed in 132 vertical columns using 200, 350 or 400 scan lines. Each character is 8 Pixels wide

Extended Graphics: The extended graphics mode provides the following software and hardware support:

8514/A Adapter Interface Compatibility Compatibility is provided through the XGA Adapter Interface, which is a device driver supplied with the subsystem as programming support for applications operating in the DOS environment.

High Resolution Support Depending on the display attached and the amount of memory installed, the image on a screen can be defined using 1024 Pixels and 768 scan lines with 256 colors.

Direct Color (16 bit True Color) Mode In this mode, each 16-bit Pixel in video memory directly specifies the color of the Pixel, rather than using the palette.

Packed Pixel Format In the packed-format, reads and writes to the video memory can access all of the data that defines a pixel (or pixels) in a single operation

Hardware Sprite The sprite is a 64 by 64 pixel image. When enabled, it overlays the picture that is being displayed. It can be positioned anywhere on the display without affecting the contents of video memory

Display Identification Signals driven by the display identify characteristics of the attached display. Applications can use the IDs to determine the maximum resolution and whether the display is color or mono.

Coprocessor A Coprocessor provides hardware drawing-assist functions throughout real or virtual memory. These functions can be used with the XGA Adapter Interface.

- Pixel-block and bit-block transfers (PxBlt)
- Line drawing
- Area filling
- Logical and arithmetic mixing
- Map masking
- Scissoring
- X,Y axis addressing.

1.1 Major Components

The subsystem components providing extended graphics function are:

- System-bus interface
- Memory and CRT controller
- Coprocessor
- Video memory
- Attribute controller
- Sprite controller
- A/N font and sprite buffer
- Serializer
- Palette
- Digital-to-analog convertor (DAC).

Note: Major subsystem components are implemented in a 2-chip set: the IMS G200 XGA display controller; and the IMS G190 XGA serializer palette DAC. Their boundaries are shown in Figure 1.1.

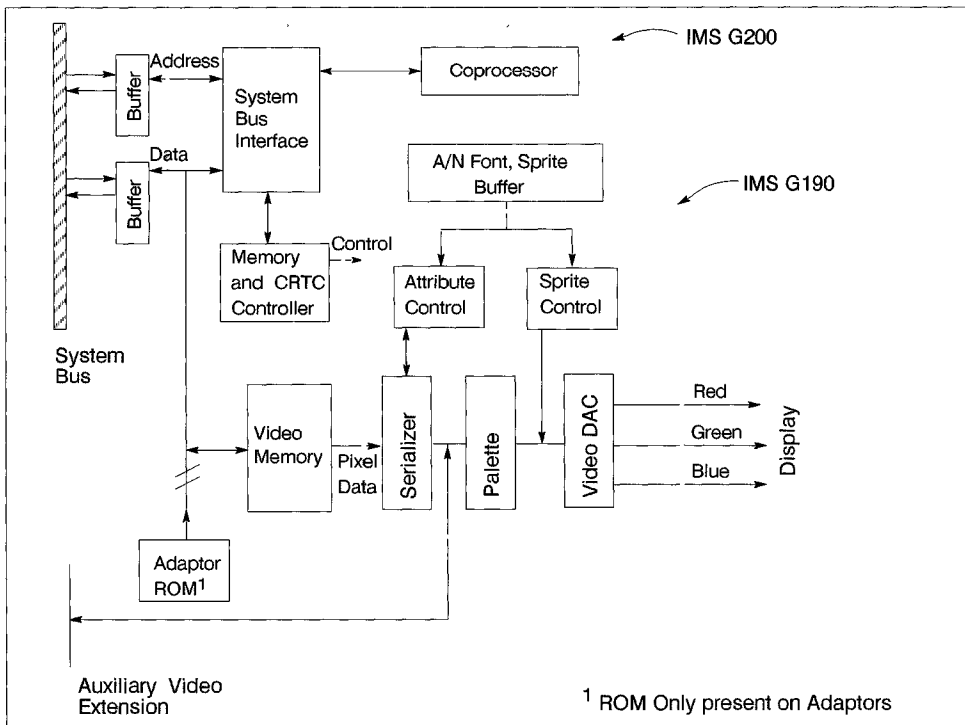


Figure 1.1 XGA Function Block Diagram

1.1.1 System Bus Interface

This component provides control of the interface between the video subsystem and the system microprocessor. It decodes the addresses for VGA and Extended Graphics I/O registers and the memory addresses for the coprocessor memory-mapped registers and video memory.

It also provides the bus-master function and determines whether the system data bus is 16- or 32-bits wide.

1.1.2 Memory and CRT Controller

This component controls accesses to video memory by the system microprocessor, displays the contents of video memory on the display, and provides support for the VGA and 132-column text modes.

1.2 Coprocessor

The coprocessor provides hardware drawing-assist functions. These functions can be performed on graphics data in both video memory and system memory.

The coprocessor updates memory independent of the system microprocessor. The instructions are written to a set of memory-mapped registers; the coprocessor then executes the drawing function.

The coprocessor functions are:

Pixel-Block or Bit-Block Transfers This function transfers an entire bit map, or part of a bit map, from one location to another. This transfer can be:

- Within video memory
- Within system memory
- Between system and video memory.

Line Drawing This function draws lines, with a programmable style and thickness, into a bit map in video memory or system memory.

Area Fill This function fills an outlined area with a programmable pattern. This function can be performed on an area outline in video or system memory.

Logical and Arithmetic Mixing These functions provide logical and arithmetic operators that can be used against data in video and system memory.

Map Masking This function provides control over updates to each Pixel for all drawing functions.

Scissoring This function provides a rectangular-mask function, which can be used instead of the mask map.

X,Y Axis Addressing This function allows a Pixel to be specified by its X and Y coordinates within a pixel map, instead of its linear address in memory.

1.2.1 Video Memory

The video subsystem uses a dual-port video memory to store on-screen data. Because this memory is dual port, video memory can be read serially to display its contents at the same time the data is being updated.

1.3 Attribute Controller

The attribute controller works together with the memory and CRT controller to control the color selection and character generation in the 132-column text mode and VGA text modes.

1.4 Sprite Controller

This component is used to display and control the position and image of the sprite, which is used as the cursor. The sprite is not available in 132-column text mode or VGA modes.

1.5 The Serializer, Palette and DAC

The serializer takes data from the serial port of video memory in 16- or 32-bit widths (depending on the amount of video memory installed) and converts it to a serial stream of Pixel data. The Pixel data is used to address a palette location, which contains the color value. The color value is then passed to the DAC, which converts the digital information into analog red, green, and blue signals to the display

1.6 A/N Font and Sprite Buffer

This buffer holds the character fonts while in 132-column text mode and VGA modes. It also stores the sprite image while in Extended Graphics modes.

1.7 Modes Of Operation

The 132 Column Text Mode and all VGA modes are available on the XGA subsystem regardless of the amount of video memory installed.

However, when in Extended Graphics Mode, the amount of Video Memory installed determines the screen resolutions and number of colors that are supported. The following table summarises this relationship:

Video Memory Installed	Resolution	Maximum Colors
512 Kbytes	640 × 480	256
	1024 × 768	16
1 Mbyte	640 × 480	65, 536
	1024 × 768	256

1.8 Compatibility

1.8.1 8514/A

The Extended Graphics Function is *not* hardware register compatible with the 8514/A adapter. Applications written directly to the register level interface of the 8514/A will not run.

The Extended Graphics Function is 8514/A Adapter Interface (AI) compatible in the DOS environment through a DOS AI driver supplied with the the video subsystem.

Applications written to the 8514/A DOS AI should continue to run unchanged with the XGA AI. The following differences should be noted:

OS/2 Protect Mode AI. An XGA AI driver is not available for OS/2 Protect Mode.

640x480 4 + 4 Mode with 512k Display Buffer. This is not an Extended Graphics Mode. However applications which use this mode and which are written to the rules in the 8514/A Technical Reference will run

Dual Display Buffer Applications. 8514/A applications that use VGA and Advanced Function modes on a single display configuration, and rely upon two separate video display buffers, will not run. However such applications should run correctly with two video subsystems (one of which is an XGA) each with a display attached.

Non-Display Memory. The XGA and 8514/A non-display (or offscreen) memory are mapped differently. Applications which use areas of the off screen memory for their own storage may not run.

Adapter Interface Code Size. The XGA AI code size is larger than that for the 8514/A. This will reduce the amount of system memory available to applications.

Adapter Interface Enhancements. The XGA AI is a superset of that provided with the 8514/A. 8514/A applications which use invalid specification of parameter blocks could trigger some of the additional function provided by the XGA AI.

Use of LIM EMS drivers. Applications written to the 8514/A AI which locate resources such as bitmaps or font definitions in LIM memory, and pass addresses of such a resource, located in LIM memory, to the AI will need a LIM driver which has implemented the Physical Address Services Interface for DMA busmasters.

Time dependent applications. Certain XGA and 8514/A functions run at different speeds. Applications which rely on a fixed performance may be affected by these differences.

XGA AI module name and directory. The module name and directory of the XGA AI (`\XGAPCDOS\XGAAIDOS.SYS`) is different from that of the 8514/A (`\HDIPCDOS\HDILOAD.EXE`). Applications written to rely on the existence of either the specific 8514/A module name or directory will not run on the XGA AI.

8514/A and XGA AI code type. The XGA AI has been implemented as a SYS device driver, whereas the 8514/A AI was a TSR ('Terminate and Stay Resident' executable program). Applications written to rely on the AI as a TSR will not run on the XGA AI.

1.8.2 LIM EMS Drivers.

The XGA coprocessor memory mapped registers are located in system memory address space. They reside in the top 1 Kbyte of an 8 Kbyte block of memory assigned to the XGA subsystem. The lower 7 Kbyte of this block is used to address ROM on an XGA subsystem implemented on an adapter card. Despite the fact that an XGA subsystem integrated on the system board does not have a subsystem ROM, an 8 Kbyte block of memory is still allocated to it in order to support the coprocessor memory mapped registers. In this case the first 7 Kbytes of this block does not contain any memory. However the memory mapped registers are still accessed in the top 1 Kbyte.

Applications or drivers [e.g. LIM EMS (Lotus Intel Microsoft Expanded Memory Services Managers) drivers] that scan memory addresses looking for RAM or ROM signatures can incorrectly assume that the entire 8 Kbytes or memory space is available for use.

The location of the 8 Kbyte block of memory assigned to the XGA subsystem can be determined using the System Unit Reference Diskette. The LIM driver installation instructions should be consulted for details on how to avoid address conflicts.

2 VGA

The XGA subsystem is register compatible with the VGA as defined in the VGA Function chapter of the Video Subsystem, in the PS/2 Hardware Interface Technical Reference. Section 11 should be consulted for switching between the different XGA modes.

3 132 Column Text

In this mode the XGA is capable of displaying 132, 8 pixel wide alphanumeric characters on the display. It is register compatible with the VGA except for certain VGA CRTIC registers detailed below.

The following VGA CRTIC register meanings are altered:

Horizontal Total Register: VGA requires that this register holds a value that is five less than the number of characters on a scan line. In 132 column text mode this register requires a value that is one less than the number of characters on a scan line.

The End Horizontal Retrace Register: In 132 Column Text Mode the VGA End Horizontal Retrace Register (bits 0 to 4) have no effect. The Extended Graphics Mode Horizontal Sync Pulse End Register (Index: 1A) is used instead. This allows a larger horizontal count.

The Sync Pulse Delay Bits (bits 5 and 6) and the End Horizontal Blanking Bit 5 (bit 7) continue to be effective. However the Sync Pulse Delay bits are now defined as follows:

bit 6	bit 5	No. of pixels Delay
0	0	0 Pixels Delay.
0	1	2 Pixels Delay.
1	0	4 Pixels Delay.
1	1	6 Pixels Delay.

See Section 11 for details on invoking 132 Column Text mode.

4 Extended Graphics

The Extended Graphics Modes provide applications with high resolution, a large color range, and high performance. The XGA coprocessor provides hardware assistance in drawing and moving data in video memory and in system memory. The Extended Graphics Modes are controlled using a bank of 16 I/O registers and the coprocessor is controlled by a bank of 128 memory mapped registers. Section 7.2 gives details of register address calculations.

4.1 Display Controller Description

4.1.1 Video Memory Format

The XGA Video Memory appears to the system as a byte addressable, packed array of pixels. The pixels may be 1,2,4,8, or 16 bits long. The first pixel in memory is displayed at the top left hand corner of the screen, the next pixel is immediately to its right and so on. Addressing is not necessarily contiguous going from one horizontal line to the next. This depends on the values in the Display Pixel Map Width registers as discussed in 4.1.5.

There are two orders of pixels supported, Intel order and Motorola order.

The Memory Access Mode Register (for Display Controller accesses), and the Pixel Map n Format Register (for coprocessor accesses), should be used to make the pixels appear in the required order to the system.

These two formats are described by the following tables:

	Byte n+2	Byte n+1	Byte n+0
	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Pixel size = 1bpp			
Pixel number	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
Bit significance	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
Pixel size = 2bpp			
Pixel number	11 11 10 10 9 9 8 8	7 7 6 6 5 5 4 4	3 3 2 2 1 1 0 0
Bit significance	1 0 1 0 1 0 1 0	1 0 1 0 1 0 1 0	1 0 1 0 1 0 1 0
Pixel size = 4bpp			
Pixel number	5 5 5 5 4 4 4 4	3 3 3 3 2 2 2 2	1 1 1 1 0 0 0 0
Bit significance	3 2 1 0 3 2 1 0	3 2 1 0 3 2 1 0	3 2 1 0 3 2 1 0
Pixel size = 8bpp			
Pixel number	2 2 2 2 2 2 2 2	1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0
Bit significance	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0	7 6 5 4 3 2 1 0
Pixel size = 16bpp			
Pixel number	1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0
Bit significance	7 6 5 4 3 2 1 0	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
The tables represent the first three bytes of the memory map in Intel Order, and shows the layout of the pixels within them for all pixel sizes (bpp = bits-per-pixel).			

Table 4.1 Memory map — Intel Order

	Byte n+0								Byte n+1								Byte n+2							
	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Pixel size = 1bpp																								
Pixel number	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Bit significance	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Pixel size = 2bpp																								
Pixel number	0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11	11
Bit significance	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
Pixel size = 4bpp																								
Pixel number	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3	4	4	4	4	5	5	5	5
Bit significance	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0	3	2	1	0
Pixel size = 8bpp																								
Pixel number	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
Bit significance	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Pixel size = 16bpp																								
Pixel number	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
Bit significance	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8
The tables represent the first three bytes of the memory map in Motorola Order, and shows the layout of the pixels within them for all pixel sizes (bpp = bits-per-pixel)																								

Table 4.2 Memory map – Motorola Order

4.1.2 Pixel Color Mapping

In 1,2,4, or 8 bits-per-pixel modes the Palette address is the numerical value of the pixel.

In 16 bits-per-pixel (Direct Color) mode, the color mapping is 5 bits red, 6 bits green, 5 bits blue. See Section 4.2

4.1.3 Border Color Mapping

In the border area of the display, the palette is addressed by the Border Color Register (Index: 55). The Border Area is defined in 4.1.5.

4.1.4 Direct Access to the Video Memory

An application can use normal memory accesses to read or write pixels in the Video Memory. All the bits of one or more pixels can be accessed in a single memory cycle.

System Apertures Into Video Memory

The XGA subsystem Video Memory is accessed in system memory address space through three possible 'apertures'. These are:

The 4 Mbyte Aperture: This aperture allows all of Video Memory to be addressed consecutively. If an access is made at an offset higher than the amount of memory installed, no memory is written and undefined values are returned when read.

The 1 Mbyte Aperture: This aperture allows up to 1 Mbyte of Video Memory to be addressed consecutively. If an access is made at an offset higher than the amount of memory installed, no memory is written and undefined values are returned when read.

Note: To use the 1 Mbyte window, the Aperture Index Register (Address: 21x8) **must** be set to zero.

The 64 Kbyte Aperture: This aperture allows up to 64 Kbytes of Video Memory to be addressed consecutively.

This aperture can be located at any 64 Kbyte section of the Video Memory using the Aperture Index Register (Address: 21x8).

See Section 11.2 for details on locating and using these apertures.

4.1.5 CRT Controller

This controller generates all the timing signals required to drive the serializer and the display. It consists of two counters, one for horizontal parameters, and one for vertical parameters, and a series of registers. The counters run continuously, and when the count value reaches that specified in one of the associated registers, the event controlled by that register occurs.

See Section 11 for mode tables including CRTC register values.

CRTC Register Interpretations

A pictorial representation of what function each of the CRTC registers can be seen in Figure 4.1.

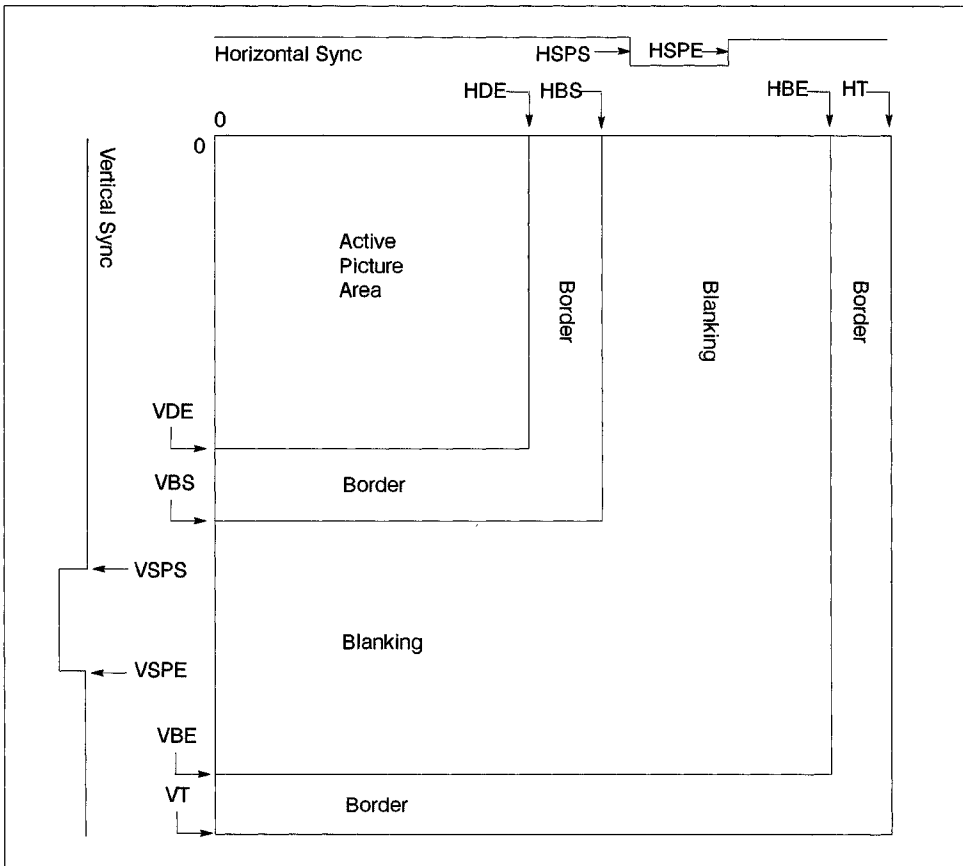


Figure 4.1 CRTC Register Definitions

The registers which control a horizontal scan of the display are:

HT:	Horizontal Total Register.
HDE:	Horizontal Display End Register.
HBS:	Horizontal Blanking Start Register.
HBE:	Horizontal Blanking End Register.
HSPS:	Horizontal Sync Pulse Start Register.
HSPE:	Horizontal Sync Pulse End Register.

The registers which control a vertical scan of the display are:

VT:	Vertical Total Register.
VDE:	Vertical Display End Register.
VBS:	Vertical Blanking Start Register.
VBE:	Vertical Blanking End Register.
VSPS:	Vertical Sync Pulse Start Register
VSPE:	Vertical Sync Pulse End Register.

The XGA can be programmed to to inform the host processor of the start and the end of the Active Picture Area using a system interrupt. An enable and a status bit exist for each interrupt. See 'Interrupt Enable Register (Address: 21x4)' and 'Interrupt Status Register (Address: 21x5)'.

Scrolling

Some, or all, of the displayed picture can be made to scroll. The first pixel displayed on the screen is controlled by the Display Pixel Map Offset registers. These can be altered to a granularity of 8 bytes giving coarse horizontal scrolling. Vertical scrolling is achieved by altering the Display Pixel Map Offset registers in units of 1 line length. The line length is stored in the Display Pixel Map Width registers. The value stored in the width registers is the amount of **memory** allocated to each line, not necessarily the physical length of the line being displayed. See Figure 4.2.

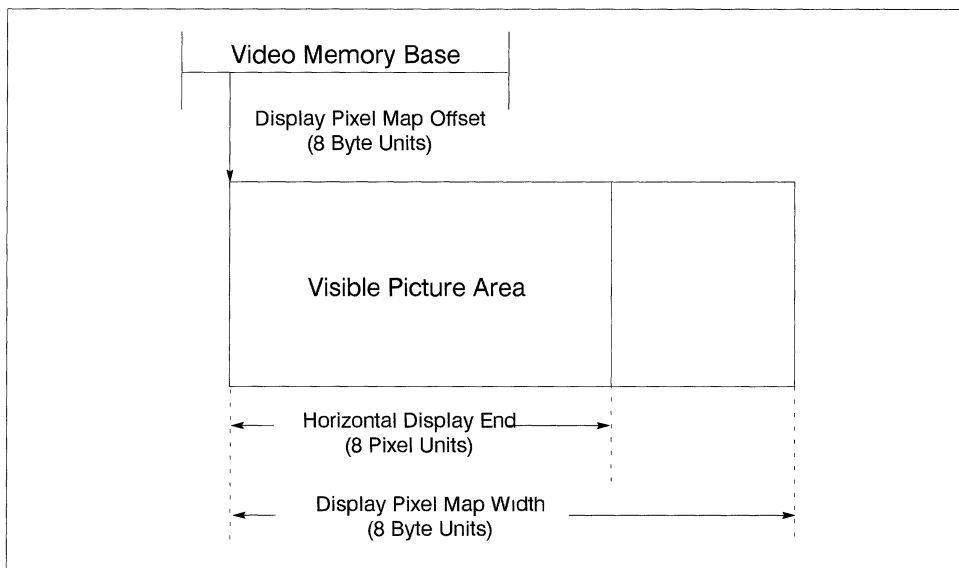


Figure 4.2 Display Pixel Map Offset & Width Definitions

The Display Pixel Map Width Registers should be loaded with a value greater than or equal to the length of line being displayed. The most efficient use of Video Memory is achieved when the width value is made equal to the length of the line being displayed. However, it is often more convenient to load a width value that specifies the start of each line to be on a suitable address boundary.

An area at the bottom of the display can be prevented from scrolling using the Vertical Line Compare Registers (Index: 2C and 2D).

4.1.6 Sprite

The sprite is a 64 × 64 pixel image stored in the XGA Alpha/Sprite buffer. When active, it overlays the picture that is being displayed. Each pixel in the Sprite can take on four values, that can be used to achieve the effect of a colored marker of arbitrary shape.

Sprite Color Mapping

The Sprite is stored as 2 bit packed pixels, using Intel format, in the Sprite Buffer. Address zero is at the top left corner of the Sprite.

These 2 bit pixels determine the sprite appearance as shown in the table below:

Bits(1:0)	Sprite Effect
00	Sprite Color 0
01	Sprite Color 1
10	Transparent
11	Complement
<p>Sprite Colors 0 and 1: These colors are set by writing to the Sprite Color Registers (Index: 38 – 3D).</p> <p>Transparent: The underlying pixel color is displayed.</p> <p>Complement: The ones complement of the underlying pixel color is displayed.</p>	

Sprite Buffer Accesses

The sprite buffer is written to by loading a number into the non-prefetch Sprite Index Hi and Sprite/Palette Index Lo registers which indicates the location of the first group of four sprite pixels to be updated (2 bits-per-pixel implies 4 pixels per byte) Then the first four pixels are written to the Sprite Data register. This stores the sprite pixels in the sprite buffer and automatically increments the Index registers. A second write to the Sprite Data register then loads the next four Sprite pixels and so on.

When reading from the Sprite buffer, the prefetch function is used. The index or address of the first sprite buffer location to be read is loaded into the Index registers. Note however that writing to either the Sprite Index Hi, or the Sprite/Palette Index Lo register with prefetch will increment both registers as a single value. As a result, the first byte of the index should be written to a non-prefetch Index register, and the second byte to the other Index register with prefetch. For example: Sprite Index Hi (no prefetch) then Sprite/Palette Index Lo (with prefetch).

The action of writing to an Index register with prefetch, causes the Sprite data stored at the location specified in the index registers to be stored in the a holding register and subsequently increments the index registers as a single value. The action of reading the Sprite Data Register returns the four Sprite pixels which were prefetched, and causes the holding register to be loaded with the next four Sprite Pixels. Another read from the Sprite Data register then returns the next 4 sprite pixels, and so on.

The sprite and the palette are written and read using the same hardware registers, so any task updating either of these on an interrupt thread must save and restore the following registers:

- Sprite/Palette Index Lo Register (Index: 60)
- Sprite Index Hi Register (Index: 61)
- Palette Sequence Register (Index: 66)
- Palette Red Prefetch Register (Index: 67)
- Palette Green Prefetch Register (Index: 68)
- Palette Blue Prefetch Register (Index: 69)
- Sprite Prefetch Register (Index: 6B)

Note: The Sprite/Palette Index Lo With Prefetch (Index: 62) and Sprite Index Hi With Prefetch (Index: 63) should not be saved and restored.

Sprite Positioning

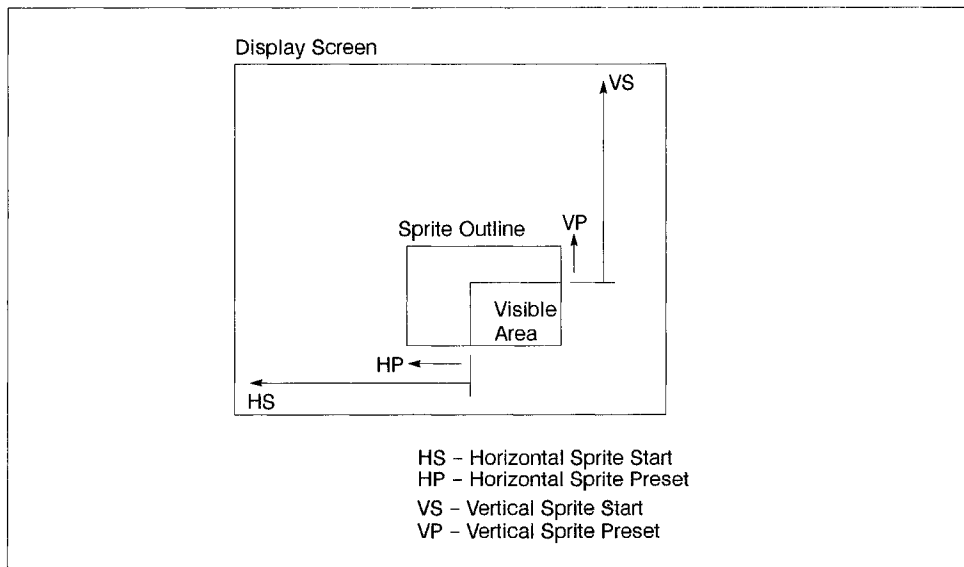


Figure 4.3 Sprite Positioning

The sprite position is controlled by two types of registers: Start and Preset. The start registers control where the first displayed sprite pixel appears on the screen, and the preset registers control which sprite pixel is first displayed within the 64 × 64 Sprite definition. Using these registers, the sprite can be made to appear at any point in the picture area. If the sprite overlaps any edge, the part of the sprite outside the picture area is not visible (does not wrap). See Section 14.2

The XGA can be programmed to inform the host processor when the last line of the sprite has been displayed on each frame using a system interrupt. This interrupt is called Sprite Display Complete. An enable and a status bit exist for this interrupt. See 'Interrupt Enable Register (Address: 21x4)' and 'Interrupt Status Register (Address: 21x5).

4.1.7 Palette

The palette has 256 locations. Each location contains 3 fields, one each for red, green and blue. It is used to translate the pixel value to a displayed color.

Before the pixel value is used to address the palette, it is masked by the palette mask register, thus all bits in the pixel corresponding to zeros in the palette mask register are forced to zero before reaching the palette.

Palette Accesses

The Palette Data register is one byte wide. However each palette location is made up of three fields (Red, Green and Blue). As a result three writes to the Palette data register are required for each palette location. The data written is held in a three field holding register, the contents of which is loaded into the palette RAM when all three fields have been filled. The Palette Sequence register controls which of the three holding register fields (Red, Green or Blue) is selected for access with each write to the Palette Data register.

There are two update sequences possible:

- Red,Green,Blue
- Red,Blue,Green,No access

The Palette is written to by loading a number into the non-prefetch Sprite/Palette Index Lo Register which indicates the index, or address, of the first group of three palette color locations to be updated (Red, Green and Blue). As the palette has only 256 locations, the Sprite Index Hi Register is not used. The first color byte can then be written to the Palette Data Register. This stores the color byte in the holding register field indicated by the Palette Sequence Register. The sequence register then increments to point to the next field as determined by the update order. A second write to the Palette Data Register loads the next holding register field, and the sequence register increments again. A third write to the Palette Data Register loads the remaining holding register field. If update sequence 1 is selected the palette location is then loaded from the holding register and the sequence register increments again, returning to its starting value. However, if the update sequence 2 is selected, a fourth write to the Palette Data Register is necessary before the palette location is loaded; the No-access data is ignored.

When reading from the Palette, the prefetch function is used. The index or address of the first Palette location to be read is loaded into the Sprite/Palette Index Lo Register (with prefetch).

The action of writing to the Index register with prefetch causes the Palette holding register to be loaded with the three color fields from Palette location pointed to by the value in the Index register and causes the index to increment. A subsequent read from the Palette Data Register returns the data from the holding register color field pointed to by the Palette Sequence Register and causes the sequence register to be incremented to point to the next color field. When the last color field, indicated by the Palette Sequence Register has been read, the holding register is loaded with the next palette location data and the index is incremented as before.

The sprite and the palette are written and read using the same hardware registers, so any task updating either of these on an interrupt thread must save and restore the following registers:

- Sprite/Palette Index Lo Register (Index: 60)
- Sprite Index Hi Register (Index: 61)
- Palette Sequence Register (Index: 66)
- Palette Red Prefetch Register (Index: 67)
- Palette Green Prefetch Register (Index: 68)
- Palette Blue Prefetch Register (Index: 69)
- Sprite Prefetch Register (Index: 6B)

Note: The Sprite/Palette Index Lo With Prefetch (Index: 62) and Sprite Index Hi With Prefetch (Index: 63) should not be saved and restored.

Note: All of the palette Red and Blue locations must be loaded with '0' if the subsystem has a monochrome monitor attached.

4.2 Direct Color Mode

Direct Color is a mode whereby the pixel values in the video memory directly specify the displayed color.

The XGA subsystem can display direct color as a 16 bit pixel where the color fields are shown below. These fields provide the most significant bits of the inputs to the DACs with the color value. Any missing lower order bits are always specified to be '0'

The bits in the 16-bit direct color data word are allocated to the DAC bits as follows.

Word bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0						
(5R, 6G, 5B)	msb		RED				lsb		msb		GREEN				lsb		msb		BLUE		lsb	

When selecting this mode, the palette **must** be loaded with data shown in Figure 4.4. Only half of the palette should be loaded. Bit 7 of the Border Color Register specifies which half to load. If the Border Color Register bit 7 = 0, load the upper half of the palette (locations '80' hex to 'FF' hex). If the Border Color Register bit 7 = 1, load the lower half (locations 0 to '7F' hex).

Location (hex) B.C =	Red (hex)	Green (hex)	Blue (hex)
0 : 1			
80 : 0	0	0	0
81 : 1	0	0	2
82 : 2	0	0	4
83 : 3	0	0	6
.	.	.	.
.	.	.	.
9E : 1E	0	0	3C
9F : 1F	0	0	3E
A0 : 20	0	0	0
A1 : 21	0	0	2
.	.	.	.
.	.	.	.
BE : 3E	0	0	3C
BF : 3F	0	0	3E
C0 : 40	0	0	0
C1 : 41	0	0	2
.	.	.	.
.	.	.	.
DE : 5E	0	0	3C
DF : 5F	0	0	3E
E0 : 60	0	0	0
E1 : 61	0	0	2
.	.	.	.
.	.	.	.
FE : 7E	0	0	3C
FF : 7F	0	0	3E

Figure 4.4 XGA Direct Color Palette Load

The values in the table above should be written as a byte to the Palette Data Register and have been chosen to ensure future compatibility.

See Sections 11.1.1 and 14.7 for details on this mode.

Coprocessor Functions

The XGA subsystem coprocessor functions do not function in 16 bits-per-pixel mode. However the coprocessor can function in 8 bits-per-pixel mode while data is being displayed in 16 bits per pixel. As a result the coprocessor can be used to move data (PxBit) from one area of memory to another.

Care should be taken however when attempting to use any of the logical or arithmetic functions as each operation will be performed on only one byte of data at a time and not the full 16 bit pixel.

If using the coprocessor to move data into the Video Display Buffer in 8 bits-per-pixel format, while displaying in 16 bits per pixel mode the width of the destination map should be doubled. See Section 14.7

4.3 Display Controller Registers

The Display Controller Registers occupy sixteen I/O addresses, and they are referred to subsequently in the text as (Base + 0) to (Base + F). Chapter 7 provides details of locating and using these registers.

An indexed addressing scheme is used in which an index number selecting a register is written to at address (Base + A) and then the register can be read or written at addresses (Base + B) to (Base + F). The multiple addresses for the data port mean that writes to a single register can be achieved in a single 16 bit instruction, the low byte containing the address, and the high byte the data, while registers which need to be accessed repeatedly (that is, the Sprite Data, the Palette Data, and the Coprocessor Save/Restore Data) can be accessed by setting the index correctly, and then executing REP INS or REP OUTS instructions, either 2 or 4 bytes at a time, to minimize the amount of PS/2 bus bandwidth used. Certain registers which are used often can be read or written directly.

The sixteen I/O addresses are assigned as follows:

- Base + 0 Operating Mode Register
- Base + 1 Aperture Control Register
- Base + 2 Reserved
- Base + 3 Reserved
- Base + 4 Interrupt Enable Register
- Base + 5 Interrupt Status Register
- Base + 6 Virtual Memory Control Register
- Base + 7 Virtual Memory Interrupt Status Register
- Base + 8 Aperture Index Register
- Base + 9 Memory Access Mode
- Base + A Index
- Base + B Data
- Base + C Data
- Base + D Data
- Base + E Data
- Base + F Data

4.3.1 Register Usage Guidelines

Unless stated otherwise:

All registers are eight bits long.

May be both read and written at the same address or index.

When read they return the data last written for all implemented bits.

Registers are NOT initialized by reset.

Reserved Register Bits —

- Register Bits marked with a ‘-’ are reserved and must be masked out if a test is to be performed on the register contents. If non reserved bits of the same register are being updated, these bits must be written to with ‘0’.
- Register Bits marked with a ‘#’ are reserved and must be masked out if a test is to be performed on the register contents. If non reserved bits of the same register are being updated, these bits must be preserved. Therefore a Read-Modify-Write operation is recommended.

Reserved Registers. Unspecified Registers, or registers marked as Reserved, in the XGA I/O address space are reserved. They must not be written to or read from.

Write Only Registers. On a read, the values returned from these registers are Reserved and Unspecified.

Read Only Registers. The contents of these registers must not be modified.

Counters should not be relied upon to wrap from the high value to the low value.

Register fields defined with valid ranges must not be loaded with a value outside the specified range.

Register field values defined as reserved must not be written

The function that all Extended Graphics Mode registers imply is only operative in Extended Graphics Mode even though the registers themselves are still readable and writable in VGA modes.

Writing to the Extended Graphics Mode registers when in VGA mode may cause VGA registers to be corrupted.

4.3.2 Direct Access I/O Registers

Operating Mode Register (Address: 21x0)

7	6	5	4	3	2	1	0
-	-	-	-	RF	DM		

This register can be both written and read.

The fields are defined as follows:

Coprocessor Register Interface Format		
RF	0	Intel Layout
	1	Motorola Layout
Display Mode		
DM	000	VGA Mode (Address Decode Disabled)
	001	VGA Mode (Address Decode Enabled)
	010	132 Column Text Mode (Address Decode Disabled)
	011	132 Column Text Mode (Address Decode Enabled)
	100	Extended Graphics Modes
	101	Reserved
	110	Reserved
	111	Reserved

Coprocessor Register Interface Format (RF): This bit selects whether the coprocessor registers are arranged in Intel or Motorola order. See Section 4.7.

Display Mode (DM): These bits are used to select between the display modes available. Both VGA and 132 Column Text modes respond to VGA I/O and memory addresses. When the XGA subsystem is in either of these modes addressing of the I/O registers and the video memory can be inhibited.

Aperture Control Register (Address: 21x1)

7	6	5	4	3	2	1	0
-	-	-	-	-	-	ASL	

This register can be both written and read.

It controls a 64 Kbyte Aperture through which the XGA memory can be accessed in system address space. This window gives real mode applications and operating systems a means of accessing the XGA video memory. The 64 Kbyte area of the XGA memory accessed by this window is selected using the Aperture Index Register. By varying the value of the index register, the 64 Kbyte aperture can be used to access the entire memory contents of the subsystem.

The aperture is controlled as described in the following table:

Aperture Size and Location		
ASL	00	No 64 Kbyte Aperture
	01	64 Kbytes at address '000A0000'x
	10	64 Kbytes at address '000B0000'x
	11	Reserved

This 64 Kbyte Aperture and a 1 Mbyte Aperture are both paged using the Aperture Index Register. As a result these two apertures cannot be used together. See **System Apertures Into Video Memory** on page 11 .

Interrupt Enable Register (Address: 21x4)

7	6	5	4	3	2	1	0
CC	CR	-	-	-	SC	P	B

This register can be written and read.

It contains bits to enable and disable individually the interrupt conditions that can be generated by the subsystem. When a bit is '1', the corresponding interrupt is enabled. When it is '0', the interrupt is disabled. The bits of this register have no effect on the interrupt status bits as defined in the Interrupt Status register below, but prevent the corresponding interrupt condition from causing a system interrupt. The bit definitions are detailed in the Interrupt Status register following.

Interrupt Status Register (Address: 21x5)

7	6	5	4	3	2	1	0
CC	CR	-	-	-	SC	P	B

This register can be written and read.

It indicates the interrupt status bits that can be generated by the subsystem and is used to reset the corresponding interrupt. On a read a '1' indicates that the corresponding interrupt condition has occurred, and a '0' that it has not. Writing a '1' to any defined bit clears the corresponding interrupt condition, while writing a '0' has no effect. The bits are assigned and defined as follows:

Bit name	Interrupt Assignment
CC	Coprocessor Operation Complete (4.5)
CR	Coprocessor Access Rejected (page 64)
SC	Sprite Display Complete (1.4)
P	Picture (Figure 4.1.5) (End of blanking)
B	End of Picture (Figure 4.1.5) (Start of blanking)

Virtual Memory Control Register (Address: 21x6)

Full details of this register are given in Section 5.5.3.

Virtual Memory Interrupt Status Register (Address: 21x7)

Full details of this register are given in Section 5.5.4.

Aperture Index Register (Address: 21x8)

7	6	5	4	3	2	1	0
-	-	Aperture Index					

This register can be written and read.

It is used to provide address bits to the video memory when the aperture in system address space being used is smaller than the amount of video memory installed. It is used to move both the 64 Kbyte aperture and the 1 Mbyte aperture. All six bits are used to move the 64 Kbyte aperture in the video memory, with a granularity of 64 Kbytes. When moving the 1 Mbyte aperture the granularity is restricted to 1 Mbyte and only bits 5 and 4 are used. The lower order bits should be written to '0' in this case.

See Section 4.1.4 for details on the use of video memory apertures. The bits used are described in the following table:

Aperture Size	Index bits used
64 Kbytes	5:0
1 Mbyte	5:4

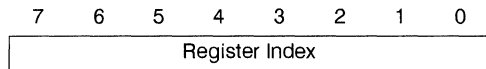
Memory Access Mode Register (Address: 21x9)

7	6	5	4	3	2	1	0
-	-	-	-	PO	PS		

This register can be written and read.

It controls pixel ordering when the video memory is being accessed by the system (not the coprocessor). Intel or Motorola order can be selected. The pixel size must also be declared, as this register is controlling a 'pixel swapper' which converts from the external format specified, to the internal format used by the adapter when the pixels are written, and converts back when they are read. Note that it is important always to set this register correctly when accessing video memory with the system processor. Values are assigned as follows.

Pixel Order		
PO	0	Intel Order
	1	Motorola Order
Pixel Size		
PS	000	1 bit
	001	2 bits
	010	4 bits
	011	8 bits
	100	16 bits
	101	Reserved
	110	Reserved
	111	Reserved

Index Register (Address: 21xA)

This register can be written and read.

It selects which indexed Extended Graphics Mode register is accessed when any address (Base + B) to (Base + F) is read or written. Index values are assigned as follows:

Index	Register	Index	Register
04	Auto-Configuration Register	30	Sprite Horizontal Start Lo
0C	Coprocessor Save/Restore Data A	31	Sprite Horizontal Start Hi
0D	Coprocessor Save/Restore Data B	32	Sprite Horizontal Preset
		33	Sprite Vertical Start Lo
		34	Sprite Vertical Start Hi
10	Horizontal Total Lo	35	Sprite Vertical Preset
11	Horizontal Total Hi	36	Sprite Control
12	Horizontal Display End Lo	38	Sprite Color 0 Red
13	Horizontal Display End Hi	39	Sprite Color 0 Green
14	Horizontal Blanking Start Lo	3A	Sprite Color 0 Blue
15	Horizontal Blanking Start Hi	3B	Sprite Color 1 Red
16	Horizontal Blanking End Lo	3C	Sprite Color 1 Green
17	Horizontal Blanking End Hi	3D	Sprite Color 1 Blue
18	Horizontal Sync Pulse Start Lo	40	Display Pixel Map Offset Lo
19	Horizontal Sync Pulse Start Hi	41	Display Pixel Map Offset Mi
1A	Horizontal Sync Pulse End Lo	42	Display Pixel Map Offset Hi
1B	Horizontal Sync Pulse End Hi	43	Display Pixel Map Width Lo
1C	Horizontal Sync Position	44	Display Pixel Map Width Hi
1E	Horizontal Sync Position	50	Display Control 1
		51	Display Control 2
		52	Display Id and Comparator
20	Vertical Total Lo	54	Clock Frequency Select
21	Vertical Total Hi	55	Border Color
22	Vertical Display End Enable Lo	60	Sprite/Palette Index Lo
23	Vertical Display End Enable Hi	61	Sprite Index Hi
24	Vertical Blanking Start Lo	62	Sprite/Palette Index Lo with Prefetch
25	Vertical Blanking Start Hi	63	Sprite Index Hi with Prefetch
26	Vertical Blanking End Lo	64	Palette Mask
27	Vertical Blanking End Hi	65	Palette Data
28	Vertical Sync Pulse Start Lo	66	Palette Sequence
29	Vertical Sync Pulse Start Hi	67	Palette Red Prefetch Register
2A	Vertical Sync Pulse End	68	Palette Green Prefetch Register
2C	Vertical Line Compare Lo	69	Palette Blue Prefetch Register
2D	Vertical Line Compare Hi	6A	Sprite Data
		6B	Sprite Prefetch Register
		70	External Clock Select Register

Note: Undefined Index Values are Reserved.

Figure 4.5 XGA Index Register Assignments

Data Registers (Addresses: 21xB to 21xF)

These data registers are used when reading and writing to the register indexed by the Index Register (21xA). The read/write operation can be of byte, word, or double-word size using these data registers.

To perform a byte write to an indexed register, a single 16 bit cycle to address 21xA can be used with the index in the lower byte and the data to be written in the upper byte. For indexed registers which require successive writes, the index can be loaded using a byte write to address 21xA, followed by either a word or a double-word access to address 21xC. Only the byte wide register selected by the index is updated. Word or double-word accesses result in two or four byte wide accesses to the same indexed register

4.3.3 Indexed Access I/O Registers

See 'Index Register (Address: 21xA)' for a table of the indexed registers.

Auto-Configuration Register (Index: 04)

7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	BS

This is a read only register.

Bus Size (BS, Bit 0): This bit indicates whether the subsystem is interfaced to a 16 or a 32 bit system interface. When set to '0' the the system interface is 16 bit, and when set to '1' the system interface is 32 bit.

Coprocessor Save/Restore Data Registers (Index: 0C & 0D)

These registers are an image of a port in the Coprocessor. See 4.6.8 for a description of their use.

Horizontal Total Registers (Index: 10 & 11)

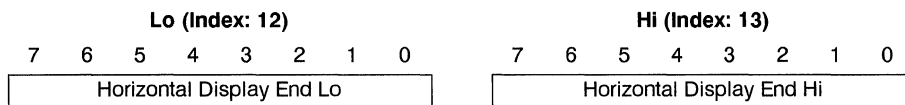
Lo (Index: 10)							
7	6	5	4	3	2	1	0
Horizontal Total Lo							

Hi (Index: 11)							
7	6	5	4	3	2	1	0
Horizontal Total Hi							

These registers can be written and read.

They define the total length of a scan line in units of eight pixels. They **must** be loaded as a 16 bit value in the range 0000 to 00FF hex. Values are assigned as follows:

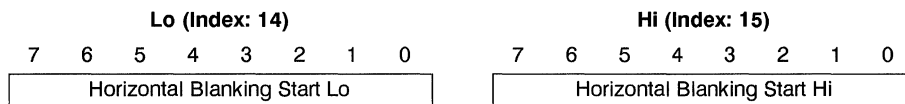
Value (hex)	Horiz Total (pixels)
0000	8
0001	16
0002	24
and so on until	
00FF	2048

Horizontal Display End Registers (Index: 12 & 13)

These registers can be written and read.

They define the position of the end of the active picture area relative to (after) the start of the active picture area in units of eight pixels. They **must** be loaded as a 16 bit value in the range 0000 to 00FF hex. Values are assigned as follows:

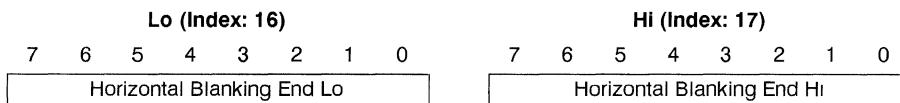
Value (hex)	Display End (pixels)
0000	8
0001	16
0002	24
and so on until	
00FF	2048

Horizontal Blanking Start Registers (Index: 14 & 15)

These registers can be written and read.

They define the position of the end of the picture border area relative to (after) the start of the active picture area in units of eight pixels. They **must** be loaded as a 16 bit value in the range 0000 to 00FF hex. Values are assigned as follows:

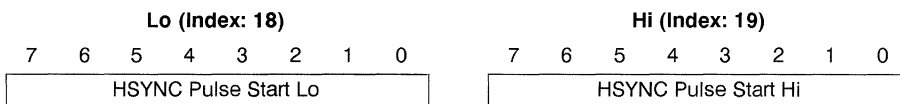
Value (hex)	Blanking Start (pixels)
0000	8
0001	16
0002	24
and so on until	
00FF	2048

Horizontal Blanking End Registers (Index: 16 & 17)

These registers can be written or read.

They define the position of the start of the picture border area relative to (after) the start of the active picture area in units of eight pixels. They **must** be loaded as a 16 bit value in the range 0000 to 00FF hex. Values are assigned as follows:

Value (hex)	Blanking End (pixels)
0000	8
0001	16
0002	24
	and so on until
00FF	2048

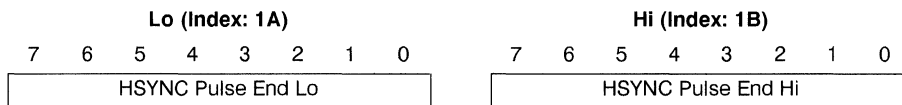
Horizontal Sync Pulse Start Registers (Index: 18 & 19)

These registers can be written and read.

They define the position of the start of horizontal sync pulse relative to (after) the start of the active picture area in units of eight pixels. They **must** be loaded as a 16 bit value in the range 0000 to 00FF hex. Values are assigned as follows:

Value (hex)	HSYNC Pulse Start (pixels)
0000	8
0001	16
0002	24
	and so on until
00FF	2048

Horizontal Sync Pulse End Registers (Index: 1A & 1B)



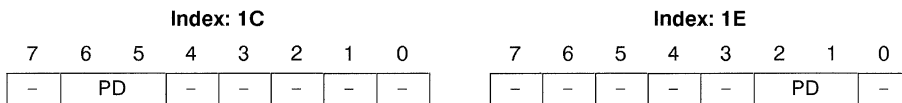
These registers can be written and read

They define the position of the end of horizontal sync pulse relative to (after) the start of the active picture area in units of eight pixels. They **must** be loaded as a 16 bit value in the range 0000 to 00FF hex.

This Extended Graphics Mode register is also used in 132 Column Text Mode in place of the VGA "End Horizontal Retrace" register. In that mode each eight pixel unit is equivalent to one eight pixel character. Values are assigned as follows:

Value (hex)	HSYNC Pulse End (pixels)
0000	8
0001	16
0002	24
	and so on until
00FF	2048

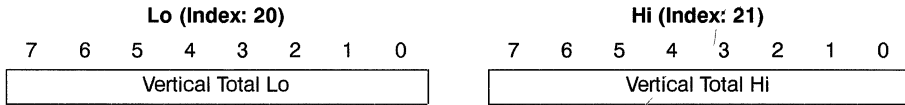
Horizontal Sync Pulse Position Registers (Index: 1C & 1E)



These registers are **WRITE ONLY**

They allow the HSYNC signal to be delayed by up to 6 pixels. The required value **must** be written to both registers.

PD	Sync Pulse Delay	
	00	0 pixels delay
01	2 pixels delay	
10	4 pixels delay	
11	6 pixels delay	

Vertical Total Registers (Index: 20 & 21)

These registers can be written and read.

They define the total length of a **frame** in units of one scan line. They **must** be written as a 16 bit value in the range 0000 to 07FF hex. Values are assigned as follows:

Value (hex)	Total Length (Scan Lines)
0000	1
0001	2
0002	3
and so on until	
07FF	2048

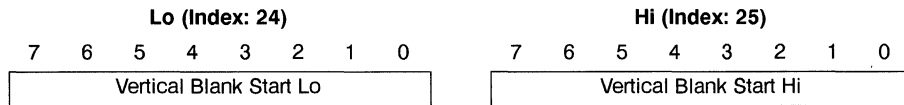
Vertical Display End Registers (Index: 22 & 23)

These registers can be written and read.

They define the position of the end of the active picture area relative to (after) the start of the active picture area in one scan line units. They **must** be written as a 16 bit value in the range 0000 to 07FF hex. Values are assigned as follows:

Value (hex)	Display End (Scan Lines)
0000	1
0001	2
0002	3
and so on until	
07FF	2048

Vertical Blanking Start Registers (Index: 24 & 25)

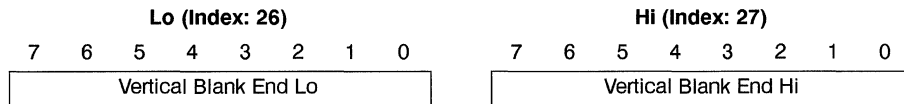


These registers can be written and read.

They define the position of the end of the picture border area relative to (after) the start of the active picture area in units of one scan line. They **must** be loaded as a 16 bit value in the range 0000 to 07FF hex. Values are assigned as follows:

Value (hex)	Border End (Scan Lines) (Blanking Start)
0000	1
0001	2
0002	3
	and so on until
07FF	2048

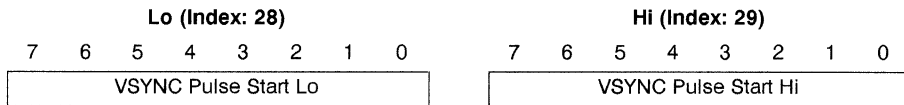
Vertical Blanking End Registers (Index: 26 & 27)



These registers can be written and read.

They define the position of the start of the picture border area relative to (after) the start of the active picture area in units of one scan line. They **must** be loaded as a 16 bit value in the range 0000 to 07FF hex. Values are assigned as follows:

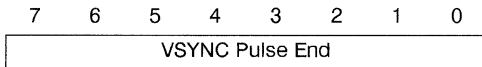
Value (hex)	Border Start (Scan Lines) (Blanking End)
0000	1
0001	2
0002	3
	and so on until
07FF	2048

Vertical Sync Pulse Start Registers (Index: 28 & 29)

These registers can be written and read

They define the position of the start of the vertical sync pulse relative to (after) the start of the active picture area in units of one scan line. They **must** be loaded as a 16 bit value in the range 0000 to 07FF hex. Values are assigned as follows:

Value (hex)	Sync Pulse Start (Scan Lines)
0000	1
0001	2
0002	3
	and so on until
07FF	2048

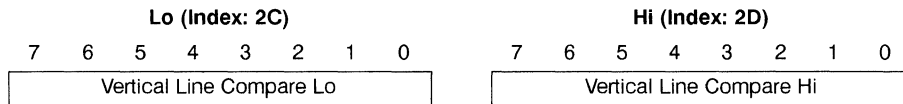
Vertical Sync Pulse End Register (Index: 2A)

This register can be written and read

It defines the position of the end of vertical sync pulse. The value loaded is the Least Significant (LS) byte of a 16 bit value which defines the end of the vertical sync pulse relative to (after) the start of the active picture area in units of one scan line. The vertical sync end position **must** be within 31 scan lines of the vertical sync start position.

Note: Before setting the Operating Mode Register (Address: 21x0) into VGA or 132 Column Text Mode, bit 5 of **this** register must be set to '1'.

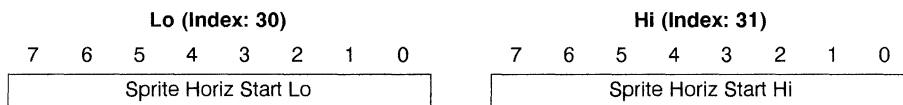
This register may not return the value written. However the returned value **is** valid for Save/Restore operations.

Vertical Line Compare Registers (Index: 2C & 2D)

These registers can be written and read.

They define the position of the end of the scrollable picture area relative to (after) the start of the active picture area in units of one scan line. They **must** be loaded as a 16 bit value in the range 0000 to 07FF hex. Values are assigned as follows:

Value (hex)	Scrollable End (Scan Lines)
0000	1
0001	2
0002	3
and so on until	
07FF	2048

Sprite Horizontal Start Registers (Index: 30 & 31)

These registers can be written and read.

They define the position of the start of the Sprite relative to (after) the start of the active picture area in pixels. They **must** be loaded with a 16 bit value in the range 0000 to 07FF hex. Values are assigned as follows:

Value (hex)	Sprite Start (pixels)
0000	0
0001	1
0002	2
and so on until	
07FF	2047

Sprite Horizontal Preset (Index: 32)

7	6	5	4	3	2	1	0
-	-	Sprite H Preset					

This register can be written and read.

It defines the horizontal position within the 64 by 64 sprite area at which the sprite starts. The sprite always ends at position 63 (that is, it does not wrap). Values are assigned as follows:

Value (hex)	Sprite Start (pixels)
00	0
01	1
02	2
	and so on until
3F	63

Sprite Vertical Start Registers (Index: 33 & 34)

Lo (Index: 33)								Hi (Index: 34)							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Sprite Vert Start Lo								Sprite Vert Start Hi							

These registers can be written and read.

They define the position of the start of the Sprite relative to (after) the start of the active picture area in units of one scan line. They **must** be loaded with a 16 bit value in the range 0000 to 07FF hex. Values are assigned as follows:

Value (hex)	Sprite Start (Scan Lines)
0000	0
0001	1
0002	2
	and so on until
07FF	2047

Sprite Vertical Preset (Index: 35)

7	6	5	4	3	2	1	0
-	-	Sprite V Preset					

This register can be written and read.

It defines the vertical position within the 64 by 64 sprite area at which the Sprite starts. The sprite always ends at position 63 (that is, it does not wrap). Values are assigned as follows:

Value (hex)	Sprite Start (Scan Lines)
00	0
01	1
02	2
	and so on until
3F	63

Sprite Control Register (Index: 36)

7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	SC

This register can be written and read.

It controls whether the Sprite is visible or invisible. When set to '1', the Sprite appears on the screen at the location controlled by the Sprite position registers. When set to '0', no Sprite is displayed. This bit should be set to '0' before any attempt is made to access the Sprite image in the Sprite Buffer otherwise the Sprite buffer contents will be corrupted.

Sprite Color Registers (Index: 38 - 3D)**Sprite Color 0****Red (Index: 38)**

7	6	5	4	3	2	1	0
Sprite Color 0 Red							

Green (Index: 39)

7	6	5	4	3	2	1	0
Sprite Color 0 Green							

Blue (Index: 3A)

7	6	5	4	3	2	1	0
Sprite Color 0 Blue							

Sprite Color 1**Red (Index: 3B)**

7	6	5	4	3	2	1	0
Sprite Color 1 Red							

Green (Index: 3C)

7	6	5	4	3	2	1	0
Sprite Color 1 Green							

Blue (Index: 3D)

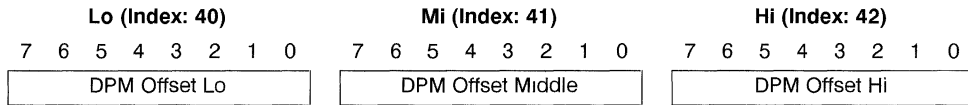
7	6	5	4	3	2	1	0
Sprite Color 1 Blue							

These registers can be written and read.

They define the red, green and blue components of the pixels displayed when the sprite data for those pixels selects Color 0 or Color 1. These colors are passed directly to the DACs and not through the palette, so should be programmed to give the actual color required.

Note: Only the 6 **most** significant bits of these registers are used.

Display Pixel Map Offset Registers (Index: 40 – 42)

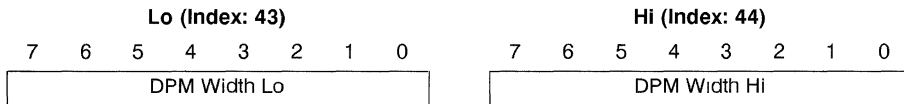


These registers can be written and read

They define the address of the start of the visible portion of the video buffer in units of 8 **BYTES**. They **must** be loaded as a single value in the range 00000 to 1FFFF hex. Values are assigned as follows:

Value (hex)	DPM Offset (bytes)
00000	0
00001	8
00002	16
and so on until	
1FFFF	1048658

Display Pixel Map Width Registers (Index: 43 & 44)



These registers can be written and read.

They define the width of the Display Pixel Map in units of 8 bytes. They **must** be loaded as a single value in the range 000 to 3FF hex. Values are assigned as follows:

Value (hex)	DPM Width (bytes)
000	0
001	8
002	16
and so on until	
7FF	16376

Display Control 1 Register (Index: 50)

7	6	5	4	3	2	1	0
SP	#	VE	SO	*	DB		

This register can be written and read.

Note: Bit 2 above (marked with a *) must be masked out if a test is being performed on the contents of this register. Also it must only be written to a '1'.

Bit 5 above (marked with a #) is Reserved. The rules specified in 4.3.1 for such bits must be observed.

Values are assigned as follows:

	Sync Polarity			
		Vertical	Horizontal	(No. Lines)
SP	00	+	+	(768 Lines)
	01	+	-	(400 Lines)
	10	-	+	(350 Lines)
	11	-	-	(480 Lines)
VE	Video Extension			
	0	Video Extension Disabled		
	1	Video Extension Enabled		
SO	Display Scan Order			
	0	Non Interlaced		
	1	Interlaced		
DB	Display Blanking			
	00	Display Blanked, CRTC reset		
	01	Display Blanked, Prepare for reset		
	10	Reserved		
	11	Normal operation		

The Video Extension Bit (VE) should be set to '1' when the subsystem is in VGA Mode. Note that it must be set to '0' (Disabled) if the subsystem is in Extended Graphics or 132 Column Text Mode.

When resetting the CRTC, the DB bits above should be set to '01' (prepare for reset) first, followed by '00' (CRTC reset)

Display Control 2 Register (Index: 51)

7	6	5	4	3	2	1	0
VSF	HSF	-	PS				

This register can be written and read.

It contains fields defining the pixel size (for the serializer, palette and DAC) and the display scale factors (horizontal and vertical). The horizontal scale factor controls how many times each pixel is replicated horizontally, and the vertical scale factor controls how many times each line is replicated.

Values are assigned as follows:

Vertical Scale Factor		
VSF	00	×1
	01	×2
	10	×4
	11	Reserved
Horizontal Scale Factor		
HSF	00	×1
	01	×2
	10	×4
	11	Reserved
Pixel Size		
PS	000	1 bit
	001	2 bits
	010	4 bits
	011	8 bits
	100	16 bits (Direct Color Mode)
	101	Reserved
	110	Reserved
	111	Reserved

Display ID and Comparator (Index: 52)

7	6	5	4	3	2	1	0
BD	GD	RD	#	DT			

This register is a **READ ONLY** register.

It contains fields indicating the type of display attached and the state of three diagnostic status bits associated with the DAC. Values are assigned as follows:

Blue DAC Comparator Status		
BD	0	Blue DAC output high
	1	Blue DAC output low
Green DAC Comparator Status		
GD	0	Green DAC output high
	1	Green DAC output low
Red DAC Comparator Status		
RD	0	Red DAC output high
	1	Red DAC output low
Display Type		
DT	0000 to 1111	As defined by displays. See table 10.1.

Clock Frequency Select Register (Index: 54)

7	6	5	4	3	2	1	0
-	-	-	-	CS	VCS		

This register can be written and read.

This register must be used in conjunction with the "External Clock Select" register. It is therefore defined under 'External Clock Select Register (Index:70)'

Border Color Register (Index: 55)

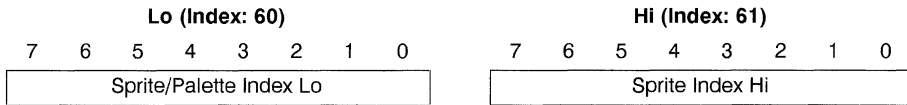
7	6	5	4	3	2	1	0
Border Colour							

This register can be written and read.

This register holds the Border Color palette index. That is, the index of the palette location selected to be displayed in the picture border area of the display.

The inverse of bit 7 is used for palette address bit 7 when in Direct Color mode. See Section 4.2.

Sprite/Palette Index Registers (Index: 60 & 61)



These registers can be written and read

They are used for specifying the index when reading from the Sprite or the Palette. See Sections 4.1.6 and 4.1.7 for details on using these registers.

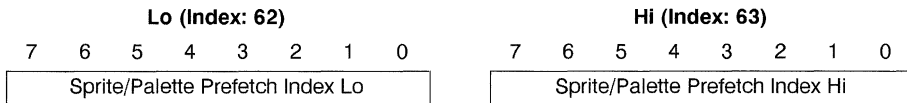
The Palette has 256 locations available and so only (Index:60) is used for the palette. It can be loaded with any palette index value in the range 0 to FF hex.

The Sprite has more than 256 locations available and so both (Index:60) and (Index:61) are used. They can be loaded with any Sprite Index value in the range 0 to 3FFF hex.

Accessing these registers does not cause any action other than loading or returning the value of the index to occur.

They **must** be saved and subsequently restored by any interrupting task that uses the palette or sprite registers.

Sprite/Palette Index Registers with Prefetch (Index: 62 & 63)



These registers can be written and read.

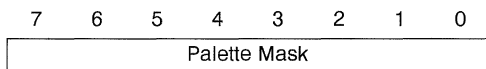
They are used for specifying the index when reading from the Sprite or the Palette. See Sections 4.1.6 and 4.1.7 for details on using these registers.

When reading from the Palette, only (Index:62) should be used. In addition to loading the register, writing (Index:62) also causes the Palette prefetch registers to be loaded, and the Index value to be incremented.

When reading from the Sprite, either (Index:62) or (Index:63) can be used. Writing to either register also causes the Sprite prefetch registers to be loaded, and the Index value to be incremented as a single value.

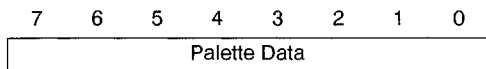
These registers should **NOT** be saved and subsequently restored in hardware task switches.

Palette Mask Register (Index: 64)



This register can be written and read.

The contents are ANDed with each Display Memory Pixel Value and the result is used to index the palette

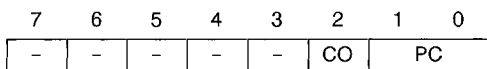
Palette Data Register (Index: 65)

This register can be written and read.

It is an image of the currently selected Palette RAM location. The data returned on read may not be that last written because of the selection mechanism described in Section 4.1.7.

For Mono Displays, all of the Palette Red and Blue locations must be loaded with '0'.

Only the 6 most significant bits of this register are used.

Palette Sequence Register (Bits 2:0 only) (Index: 66)

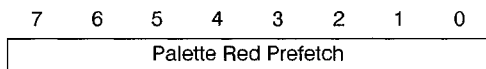
This register can be written and read.

It contains two fields, one defining which of the R, G or B elements of the currently selected palette location is the current one for the Palette Data Register, the other defining the sequence to be followed for selecting the R, G and B elements for successive Palette Data Register accesses.

See Section 4.1.7.

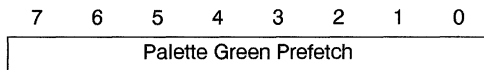
Color Order		
CO	0	R, G, B, R, G, B,
	1	R, G, B, x, R, G, B, x,
Palette Color		
PC	00	R
	01	G
	10	B
	11	x

Note: x = discarded data

Palette Red Prefetch Register (Index: 67)

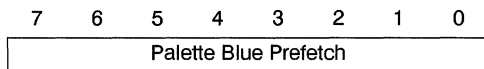
This register can be written and read.

It is not used for any normal function but **must** be saved and subsequently restored by any interrupting code that uses the sprite or palette registers

Palette Green Prefetch Register (Index: 68)

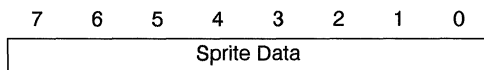
This register can be written and read.

It is not used for any normal function but **must** be saved and subsequently restored by any interrupting code that uses the sprite or palette registers.

Palette Blue Prefetch Register (Index: 69)

This register can be written and read.

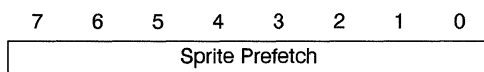
It is not used for any normal function but **must** be saved and subsequently restored by any interrupting code that uses the sprite or palette registers.

Sprite Data Register (Index: 6A)

This register can be written and read.

It is an image of the currently selected Sprite buffer location. The data returned on read may not be that last written because of the selection mechanism described in Section 4.1.6.

When used for writing sprite data, the sprite pixels are Intel format packed pixels.

Sprite Prefetch Register (Index: 6B)

This register can be written and read.

It is not used for any normal function but **must** be saved and subsequently restored by any interrupting code that uses the sprite or palette registers.

External Clock Select Register (Index: 70)

7	6	5	4	3	2	1	0
C	-	-	-	-	-	-	-

This register can be written and read.

It must be used in conjunction with the Clock Frequency Select Register (Index 54) shown below:

7	6	5	4	3	2	1	0
-	-	-	-	CS		VCS	

The combined function of these register fields is detailed in the table below:

	Clock Selected		
	C	CS	Selected Clock
C and CS	0	00	VGA 8 pixel Character Mode and 640×480 Graphics Mode Clock.
	X	01	VGA 9 pixel Character Modes clock.
	X	10	Clock sourced from Video Extension Interface.
	X	11	1024×768 Graphics Mode Clock.
	1	00	132 Column Mode Clock.
VCS	Video Clock Scale Factor		Mode
	00	×1	VGA and 640×480 Graphics Modes.
	01	×2	
	10	Reserved	
11	Reserved		

Clock Selection– The 'C' and 'CS' fields of index 70 and 54 must be used together. Setting each field to the values shown in the table above will select the specified clock. Note: If the 132 Column Text Mode Clock is selected, then 'C' **must** be returned to '0' before any VGA or 640×480 graphics is selected. See Section 11 for details on mode switching.

The video clock scale factor controls the divide ratio of the selected video clock before it is used by the CRTIC. The operation of the video clock scale factor is invisible to the programmer, but it must be set as shown to allow correct operation of the hardware.

4.4 Coprocessor Description

The XGA coprocessor provides autonomous drawing functions for the video sub-system. “Autonomous drawing functions” means that the coprocessor draws into memory (either video memory or system memory) independently of the host system processor, while the host processor is performing some other operation.

The coprocessor supports 1,2,4 or 8 bits bits per pixel. See Section 14 7 for details of using the coprocessor when displaying in 16 bits-per-pixel (Direct Color) mode.

Typically, the execution of an operation using the coprocessor involves the following steps:

- The host system processor sets up the coprocessor registers to perform a particular operation
- The host system processor writes to the Pixel Operations register to start the coprocessor operation
- The coprocessor performs the drawing operation The host system processor can be performing some other function at this time.
- The coprocessor completes the drawing operation, informs the host system processor, and becomes idle
- The process repeats . . .

The coprocessor operates on pixels within pixel maps. A pixel map is an area of memory at a given address with a defined height, width, and pixel format; see Section 4.6.3

Pixels from a Source are combined with pixels from a Destination under the control of a Pattern and Mask, and the result is written back to the Destination.

After each access the Source, Destination, Pattern and Mask addresses are updated according to the function being performed, and the operation is repeated until a programmed limit is encountered.

The drawing operation can be one of Pixel Block Transfer (PxBlt), Bresenham Line Draw, or Draw and Step.

The function performed to combine the Source and Destination data can be a logical or arithmetic operation. One of two possible operations is selected for each pixel by the value of the corresponding Pattern pixel. In addition, a Mask pixel for each pixel allows the Destination to be protected from update.

Pattern data can be generated automatically from Source data This is done by detecting pixels with a value of “0”.

A color compare function is provided. This allows the modifying of the destination pixel to be dependent on the value of the destination pixel compared to a programmable value.

Three general purpose pixel maps can be defined in memory Each map has a defined start address, pixel width and height, and number of bits-per-pixel. Source, Pattern and Destination data can reside in any combination of these maps. There is also a mask map that has its own defined start address, width, height and format. Mask data is always taken from this map.

Source, Pattern and Destination data are each addressed by unique X,Y pointers. Mask data is addressed by the Destination X,Y pointers; see Figure 4.7. Should the Source or Pattern X,Y pointers move outside the defined extremities of their pixel maps, they are automatically reset so as to wrap round to the opposite side of the pixel map. If the Destination X,Y pointers move outside the extremities of the Destination map, update of the Destination map is inhibited until the X,Y pointers move back inside the map.

Figure 4.6 shows a representation of the coprocessor graphics data flow. The diagram indicates the passage of **one pixel** through the data flow. In reality, multiple pixels are processed in one cycle.

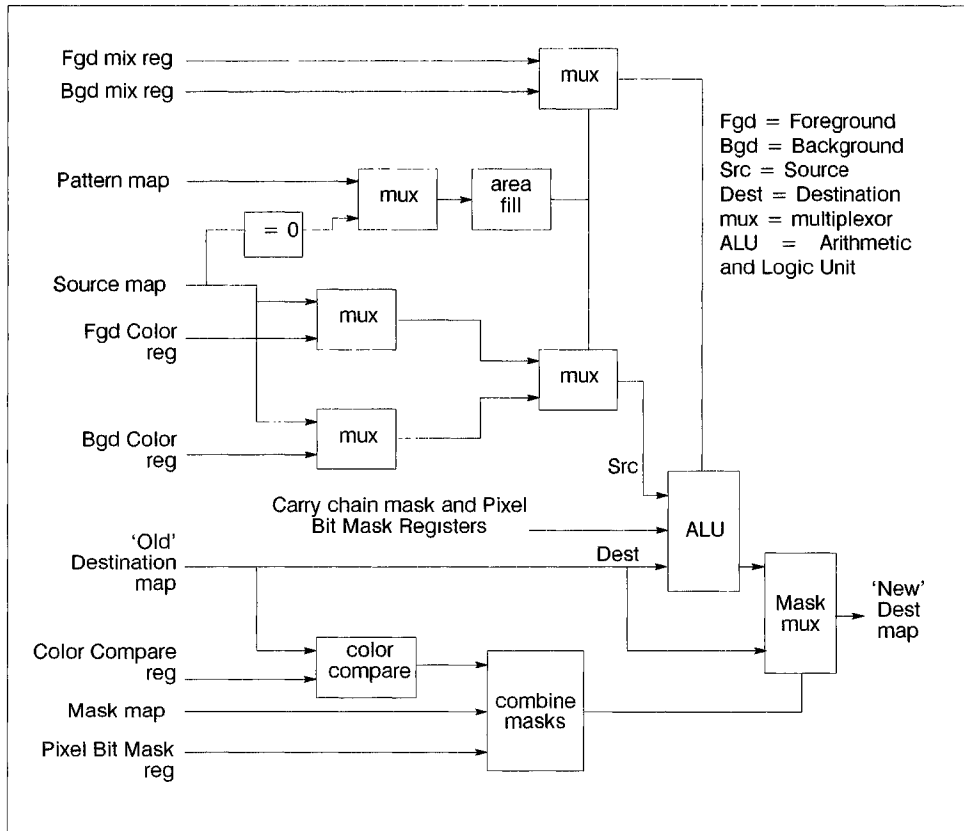


Figure 4.6 Coprocessor data flow.

4.5 Programmer's View

An 'operation' is defined as the execution of a single PxBit, Line Draw or Draw and Step function

An operation is set up by first loading the coprocessor's registers with appropriate data, such as X,Y coordinates, function mixes, dimensions, and so on. The operation is then initiated by writing to the Pixel Operation Register. This defines the flow of data in the operation and also starts the operation. The coprocessor then executes and completes the operation when some programmed limit has been reached.

There is one exception to the above sequence of initiating operations. This is the Draw and Step function, described in the section 'Draw and Step' on page 51.

The XGA can be programmed to inform the host processor of the completion of an operation using a system interrupt. This interrupt is called the Coprocessor Operation Complete Interrupt. An enable bit and status bit exist for this interrupt in the 'Interrupt Enable Register (Address: 21x4)' on page 21 and 'Interrupt Status Register (Address: 21x5)' on page 21

A mechanism is provided to allow the host processor to either suspend or terminate an operation before it has completed. The suspension of operations is required to allow task switches, while termination of operations can be used to recover from errors.

4.6 Pixel Formats

The Coprocessor can manipulate images with 1, 2, 4 or 8 bits per pixel. It manipulates packed-pixel data, so each data double-word (32 bits) contains 32, 16, 8 or 4 pixels respectively.

The pixels can be in one of two different formats, 'Motorola' or 'Intel'. See Intel Order on page 10 and Motorola Order on page 11.

Each pixel map manipulated by the Coprocessor can be defined as either Motorola or Intel format. If the Destination Map has a different format to that of the Source, Pattern or Mask Maps, the Coprocessor automatically translates between the two formats.

Motorola or Intel format is controlled by a bit in the Pixel Map Format Register.

4.6.1 Pixel Data

Fixed And Variable Data

In the course of executing an operation, the Coprocessor reads in Source, Pattern and Mask data, and reads and writes Destination data. The Source, Pattern and Mask data can either be fixed throughout the operation, or vary from pixel to pixel.

XGA Function

If fixed data is to be used, the data is written to the relevant fixed data register in the Coprocessor before the operation is started (Foreground and Background Color Registers).

If variable data is required, the data is read **from memory** by the Coprocessor during the course of the operation. The Coprocessor only allows variable data to be provided from memory, and does not allow the system unit host processor to supply variable data.

4.6.2 The Coprocessor View of Memory

To the programmer, the Coprocessor treats video memory and system memory in the same manner. Thus data can be moved between system memory and video memory by defining pixel maps at the appropriate addresses.

Accesses to the XGA video memory are faster than accesses to system memory.

The Coprocessor can address all the video memory.

The Video Memory Base Address Registers hold a value that indicates the base address at which the Video Memory appears in system address space. This base address is on a 4 Mbyte address boundary. The Coprocessor assumes that the whole 4 Mbytes of address space above this boundary is reserved for its own video memory. All addresses outside this 4 Mbyte block are treated as system memory.

Section 4.1.4 further describes video memory addressing.

4.6.3 XGA Pixel Maps

Pixel Maps A, B, And C (General Maps)

The Coprocessor defines three general purpose pixel maps in memory, called Pixel Maps A, B and C. Each map is defined by four registers:

Pixel Map Base Pointer This specifies the linear start address of the map in memory.

Pixel Map Width This specifies the width of the map in pixels. The value programmed should be 1 less than the required width.

Pixel Map Height This specifies the height of the map in pixels. The value programmed should be 1 less than the required height.

Pixel Map Format This specifies the number of bits-per-pixel of the map, and whether the pixels are stored in Motorola or Intel format.

The Source, Pattern and Destination data can each reside in any of Pixel Maps A, B or C, determined by the contents of the Pixel Operations Register.

These maps may be defined to be any arbitrary size up to 4096 by 4096 pixels. Individual pixels within the maps are addressed using X,Y pointers. See 'X and Y Pointers' on page 46.

Pixel Maps can be located in video memory and in system memory.

There are two restrictions on map usage: the Source and Destination maps must have the same number of bits-per-pixel, and the Pattern map must be 1 bit-per-pixel.

Pixel Map M (Mask Map)

In addition to the three general purpose maps, the Coprocessor also defines a Mask Map. This map is closely related to the Destination map. It allows the Destination to be protected from update on a pixel-by-pixel basis, and can be used to provide a scissoring-type function on any arbitrary shaped area. See 'Scissoring with the Mask Map' on page 48.

The Mask Map is described by a similar set of registers to the general purpose pixel maps A, B and C, but it is fixed at 1 bit-per-pixel.

The Mask Map differs from the Source, Pattern and Destination maps in that:

- The Mask Map uses the Destination X and Y pointers.
- The position of the Mask Map origin relative to the Destination is defined by the Mask Map Origin X and Y Offsets.

See 'X and Y Pointers' on page 46.

Map Origin

The origin of a pixel map is the point where $X=0$ and $Y=0$.

The Coprocessor defines the origin of all its pixel maps as being at the top left corner of the map. The direction of increasing X is to the right; the direction of increasing Y is downwards. Figure 4.7 illustrates the X Y addressing of an XGA map.

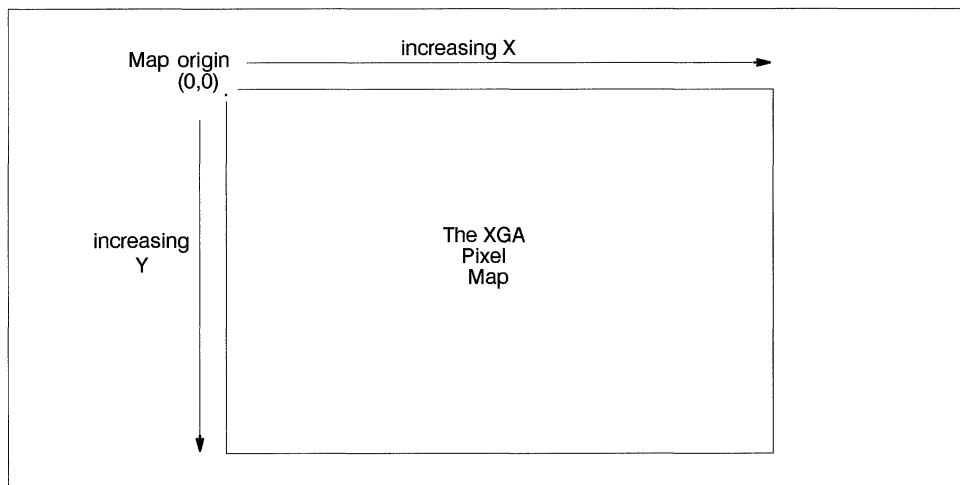


Figure 4.7 The XGA Pixel Map Origin

In storage, pixels to the right of and **below** the origin are stored in ascending, contiguous memory locations.

X and Y Pointers

Source And Pattern Maps: These maps each have X and Y pointers that determine the pixel to be accessed for that map. The two sets of pointers are completely independent and are modified as the operation proceeds.

If, in the course of an operation, the Source or Pattern pointers are moved beyond the extremities of the Pixel Map containing the Source or Pattern data, they are reset to the opposite edge of the pixel map. Source and Pattern maps can thus be regarded as continuous in that they wrap round at their extremities. This allows a small pattern to be repeated or 'tiled' over a large area in the destination map, in a single operation; see Figure 4.8.

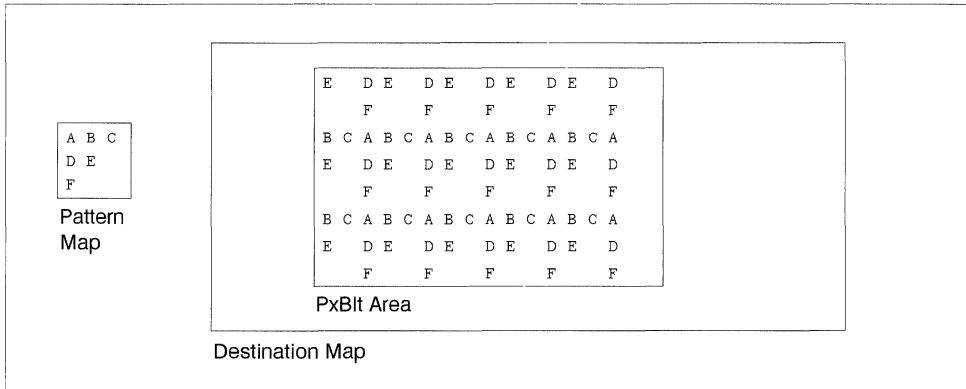


Figure 4.8 Repeating Pattern ('Tiling')

Destination Map: If a Destination X or Y pointer is moved beyond the extremity of the Pixel Map containing the Destination, the pointers are not wrapped, but updates to the Destination are disabled until the pointers are moved to within the defined Pixel Map. This mechanism is effectively a fixed scissor window around the Destination pixel map.

A “guardband” exists around the Destination Map that ensures that the Destination X and Y pointers do not wrap when they move outside the limits of the map. The guardband is 2048 pixels deep on all sides of the largest definable Destination Map. The guardband is illustrated in Figure 4.9.

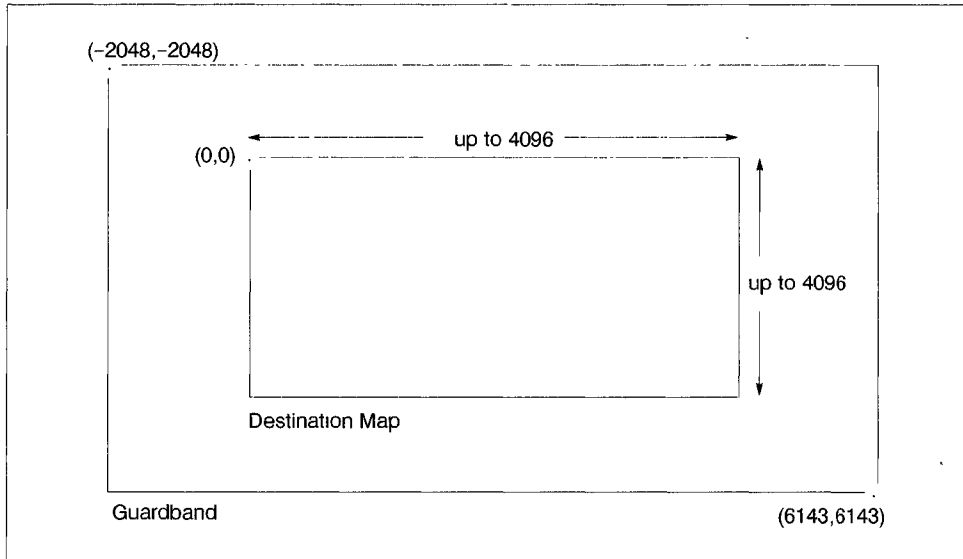


Figure 4.9 Destination Map Guardband

The Guardband allows the Destination X and Y addresses to range from -2048 to $+6143$. All pixels within the Destination Map can be updated, but updates to pixels that lie within the guardband are inhibited. The size of the Destination Map is determined by the Map Width and Height, so pixels that lie within the range $(0, 0)$ to $(\text{width}-1, \text{height}-1)$ can be updated. The Guardband occupies pixel X addresses -2048 to -1 and 'Width' to 6143 , and Y addresses -2048 to -1 and 'Height' to 6143 .

In order to correctly address the Destination Map and take advantage of the Coprocessor's Destination Boundary Scissor capability, programmers can calculate Destination X and Y addresses using 16-bit two's-complement numbers. They should ensure that all X and Y addresses generated by the operation they program lie within the range -2048 to 6143 , and bear in mind that all pixels that they want drawn should lie inside the bounds of the Destination Map. Any X and Y addresses generated that lie outside the range -2048 to 6143 cause the XGA X and Y pointers to wrap and can produce erroneous results.

Mask Map: The Mask Map width and height can be any size less than or equal to the dimensions of the Destination map. The Mask Map can therefore be smaller in size than the Destination Map. If this is the case, the hardware needs to know where the Mask Map is positioned relative to the Destination Map. Two pointers, the Mask Map Origin X Offset and Mask Map Origin Y Offset specify the X,Y position in the Destination at which the Mask Map origin is located. Figure 4.10 illustrates the use of these pointers.

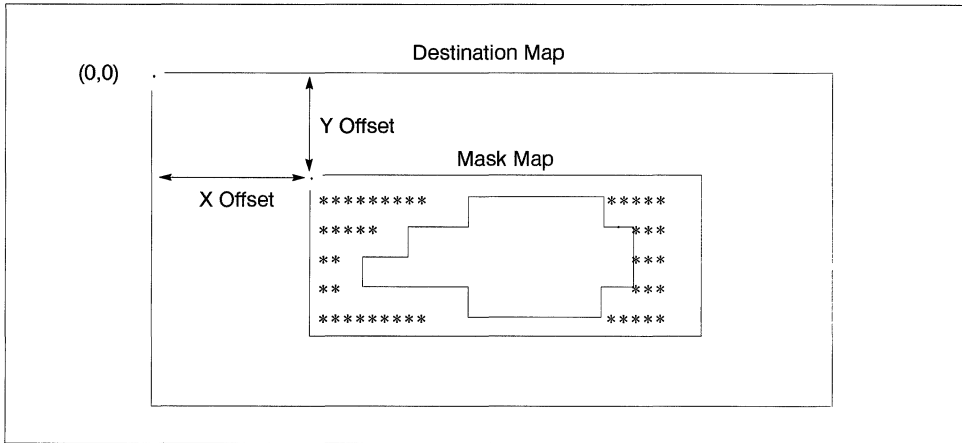


Figure 4.10 Mask Map Origin X and Y Offsets

The Mask Map takes its X and Y pointers from the Destination X and Y pointers. For every pixel in the Destination, the corresponding pixel in the Mask Map is read and update of the Destination enabled or disabled depending on the value of the Mask pixel.

Scissoring With The Mask Map

Hardware scissoring is provided in the Coprocessor using the Mask Map. There are three ways that the Mask Map can be used for any operation, as follows:

Disabled The Mask Map contents and boundary position are ignored.

Boundary Enabled The contents of the Mask Map are disabled, but the boundary of the Mask Map acts as a rectangular scissor window on the Destination map. No memory is required to store the map contents in this mode.

Enabled The contents of the Mask Map can be used to provide a possibly non-rectangular scissor window. The boundary of the Mask Map also provides a rectangular scissor window at the extremities of the Mask Map.

The Mask Map mode is controlled by a field in the Pixel Operation Register. The modes are described in detail below. Throughout the description below, pixels that are located **on** a scissor boundary are treated as if they are **inside** it.

Mask Map Disabled: When the Mask Map is disabled, any updates to the Destination are always performed regardless of the position or contents of the Mask Map. No memory need be reserved for the Mask Map, and the contents of the Mask Map Base, Width, Height, Format and Origin Offset registers are ignored.

However, should the operation being performed attempt to draw **outside** the boundary of the Destination Map, the update is automatically inhibited. The Destination X and Y pointers are incremented as normal, but update of the Destination is not enabled until the pointers move back inside the bounds of the Destination Map. Thus a fixed hardware scissor window exists around the boundary of the Destination Map. This Destination Boundary Scissor is always enabled regardless of the Mask Map Mode.

Figure 4.11 illustrates the Destination Boundary Scissor operation when the Mask Map is disabled.

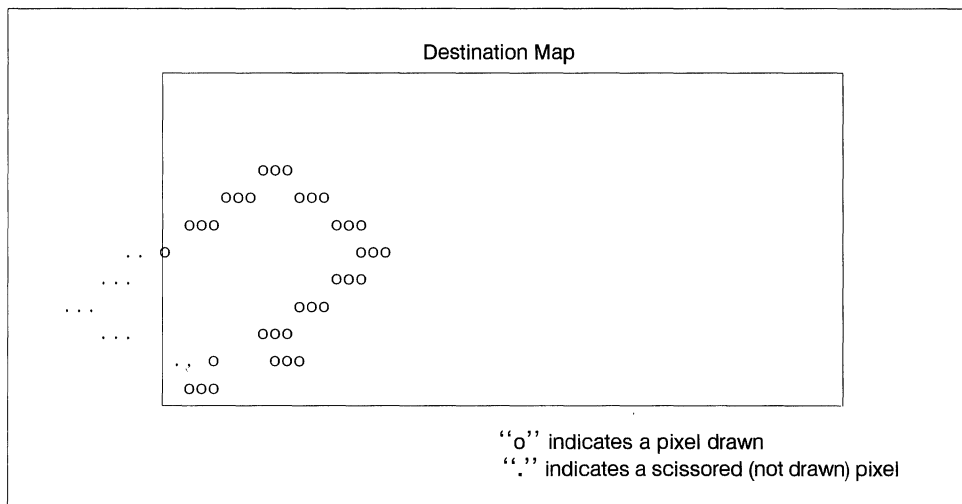


Figure 4.11 Destination Boundary Scissor.

Mask Map Boundary Enabled: Mask Map Boundary Enabled mode provides a single rectangular scissor window within the Destination map. The **contents** of the Mask Map are ignored, and thus no memory need be reserved for the Mask Map in this mode.

In Boundary Enabled mode, the size and position of the Mask Map must be specified. Thus the Mask Map Width, Height, and Origin Offset registers must be defined. These four registers together define a rectangular boundary within the Destination Map. Updates to the Destination Map that lie **inside** this boundary take place as normal. Updates **outside** this boundary are inhibited.

Figure 4.12 illustrates a Mask Map Boundary Enabled scissoring operation.

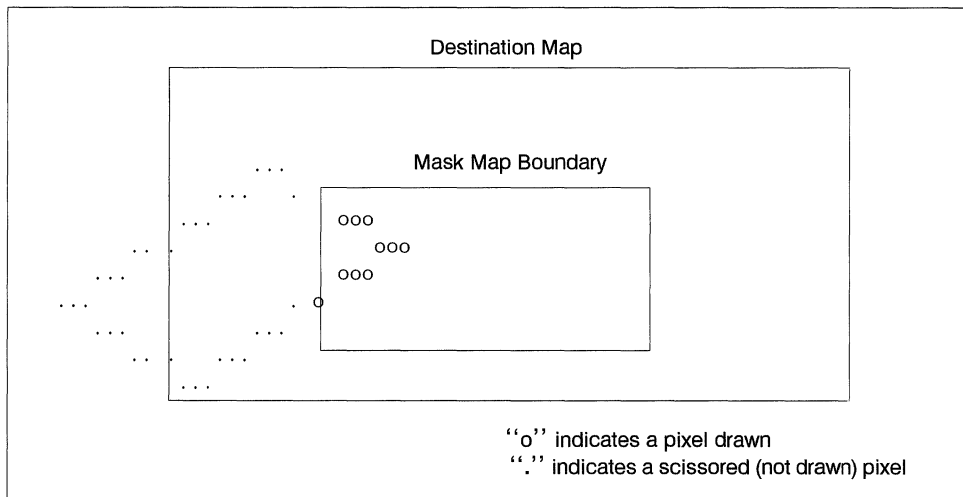


Figure 4.12 Mask Map Boundary Scissor.

Mask Map Enabled: When the Mask Map mode is enabled, both the Mask Map boundary and contents provide scissoring action. Memory must be reserved to hold the Mask Map pixels and the

Mask Map Base, Width, Height, Format, and Origin Offset registers must be set up to point to the mask data and describe its size and position relative to the Destination Map.

For any pixel in the Destination that is about to be updated, the corresponding Mask Map pixel is examined.

If the Mask pixel is inactive (its value is '0'), the Destination pixel update are inhibited. If the Mask pixel is active (its value is '1'), the Destination pixel is updated as normal.

This mode allows the user to draw non-rectangular scissor windows in the Mask Map prior to an operation, and then, in a single execution of an operation, to apply a non-rectangular scissor window to that operation.

Memory must be reserved to hold the Mask Map contents in this mode. The Mask data is fixed at 1 bit-per-pixel, so for a full screen Mask Map for a 1024 x 768 screen, 96 Kbytes are required. If, however the operation to be scissored does not cover the whole Destination Map, a Mask Map smaller than the Destination Map can be used in order to save memory. For applications with no memory available for the Mask Map contents, the Mask Map Boundary Enabled mode should be used.

Figure 4.13 illustrates a Mask Map Enabled scissor operation.

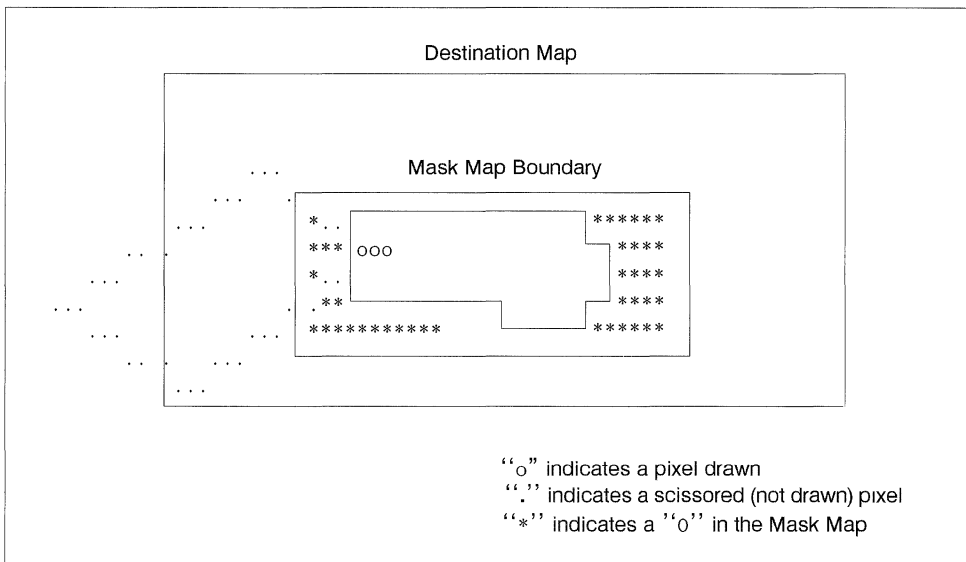


Figure 4.13 Mask Map Enabled Scissor

Before performing an operation that requires a non-rectangular scissor, the user must first draw the non-rectangular mask into the Mask Map. Typically, windowing systems only permit rectangular windows, so the mask can be drawn using a sequence of PxBit operations that have fixed source data. For more complex shapes, the Line Draw and Draw Step functions can be used to draw area outlines that can then be filled.

Typically, a large number of operations can be performed, all using the same mask, so the overhead per operation in setting up the mask is small. Overall, the use of the mask to perform non-rectangular scissors greatly improves the performance of a given drawing operation over that when a single rectangular scissor is provided by the hardware.

4.6.4 Drawing Operations

There are four drawing operations provided by the Coprocessor. They are:

- Draw and Step
- Line Draw
- Pixel Block Transfer (PxBlt)
- Area Fill

These operations are described in detail in the following sections

The operations can be either one-dimensional or two-dimensional. Draw and Step and Line Draw are one-dimensional while the PxBlts are two-dimensional. Draw and Step and Line Draw are collectively described as 'draw' operations in the following text

Either of the draw operations can be either 'read' or 'write'. These qualifiers to the operation are described in 'Line Draw' on page 53.

Draw and Step:

The Draw and Step operation draws a pixel at the Destination and then updates the X Y pointers to one of the pixel's 8 neighbors according to a 3-bit code.

A run of up to 15 address steps can be specified in a fixed direction by each Draw and Step code. An 8-bit code describes the vector, as shown in Figure 4.14

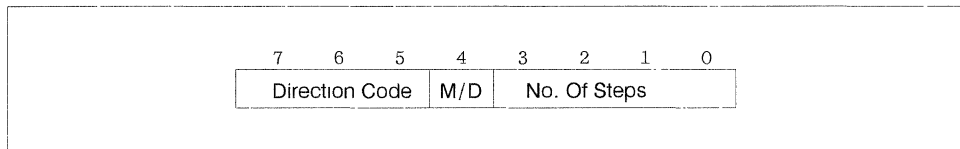


Figure 4.14 Draw and Step code

No. Of Steps: This field indicates how many steps should be taken, from 0 to 15. The X Y pointers are updated **after** the pixel is drawn, so a Draw and Step function always attempts to draw at least one pixel

The number of steps to be taken in the Draw and Step operation is one less than the number of pixels that the hardware attempts to draw. Thus when the number of steps is programmed to 5, 6 pixels are drawn; when 0 steps are specified, 1 pixel is drawn. After the Draw and Step operation, the X and Y pointers point to the last pixel that the operation attempted to draw (this pixel may not actually be drawn if the 'Last Pel Null' Drawing Mode is active).

For example, a Draw and Step code of hex '35' moves X,Y pointers starting at coordinates (17,10) to coordinates (22,5), as shown in Figure 4.15.

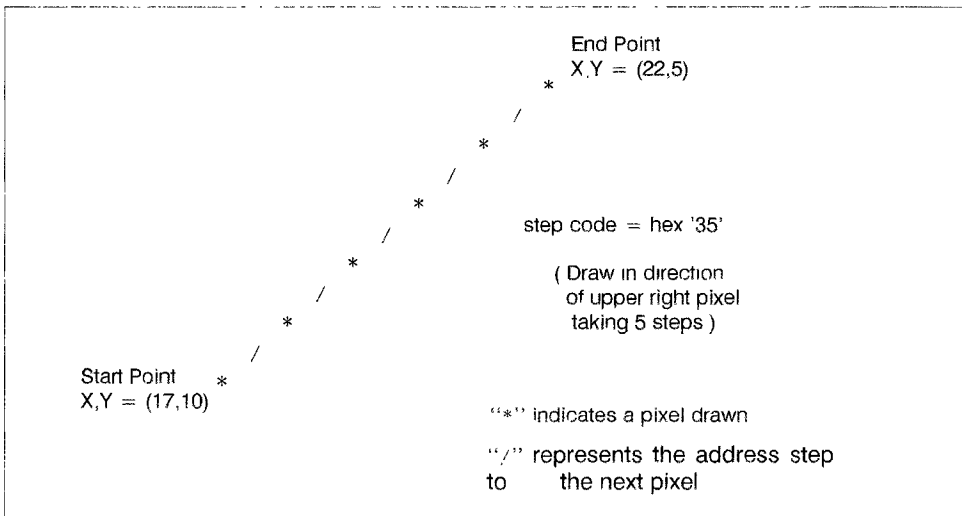


Figure 4.15 Example of Draw and Step

M/D: This field specifies if the operation is a move operation or draw a draw operation. When set to a '1' this bit indicates that pixels should be drawn. When cleared to '0', it indicates that X and Y pointers should be modified as normal, but no pixels should be drawn.

Direction Code: This field indicates the direction of drawing relative to the current pixel, as shown in Figure 4.16.

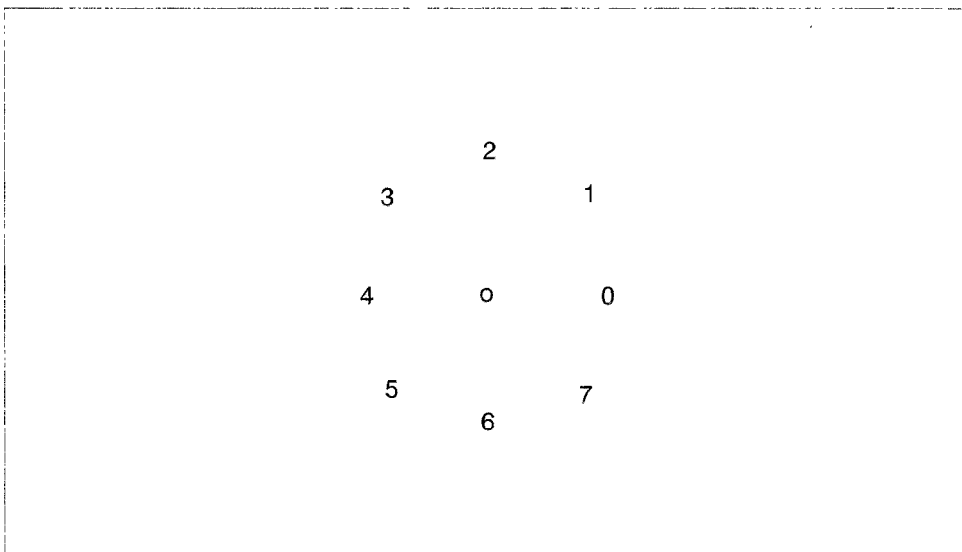


Figure 4.16 Draw and Step direction codes

Draw and Step codes should be written to the Direction Step Register. Each write to the register can load up to four Draw and Step codes in one access. The Draw and Step codes are executed starting with that in the least significant byte. Each group of up to four codes written to the Direction Step register is treated

as being ONE operation in that all codes are executed before the coprocessor indicates that the operation has completed. However, for the purposes of first pixel null and last pixel null drawing (described below), each code describes a distinct line.

The Draw and Step operation differs from other operations in that it is not initiated through the Pixel Operation Register. The action of writing a Draw and Step code to the **most significant** byte of the Direction Step Register initiates the Draw and Step operation.

The Pixel Operation Register still needs to be loaded in order to specify the particular Draw and Step function and the data flow for the operation. This must be done **before** any data is written to the Direction Step Register. Writing the Pixel Operation register with a function of Draw and Step does NOT initiate a Draw and Step operation, but sets up the parameters for the operation. It is the action of writing steps to the Direction Step register that initiates the Draw and Step operation. If the Pixel Operation register specifies a function other than Draw and Step when the Direction Step register is written, no operation takes place.

A Draw and Step code of '00' is treated by XGA as a 'Stop' code. If a Stop code is encountered as one of the (up to) four codes in the Direction Step register, the Draw and Step operation completes after that code has been executed. The completion of the operation is indicated in the normal way through the Coprocessor Control register, and, in addition, a flag bit in the Coprocessor Control register indicates that the operation completed because a Stop code was encountered. This mechanism allows software to load sequences of Draw and Step codes to the Coprocessor without needing to keep track of the number of codes that make up the figure being drawn.

If it is required to program less than four codes to the Direction Step register, two possible approaches can be taken. Either the first unwanted step code can be set to '00' (Stop), and all 32 bits of the register written, or only the required number of codes can be written to the Direction Step register. However, in the latter case, the codes must be written to the most significant bytes of the register. The two methods are shown in Figure 4.17.

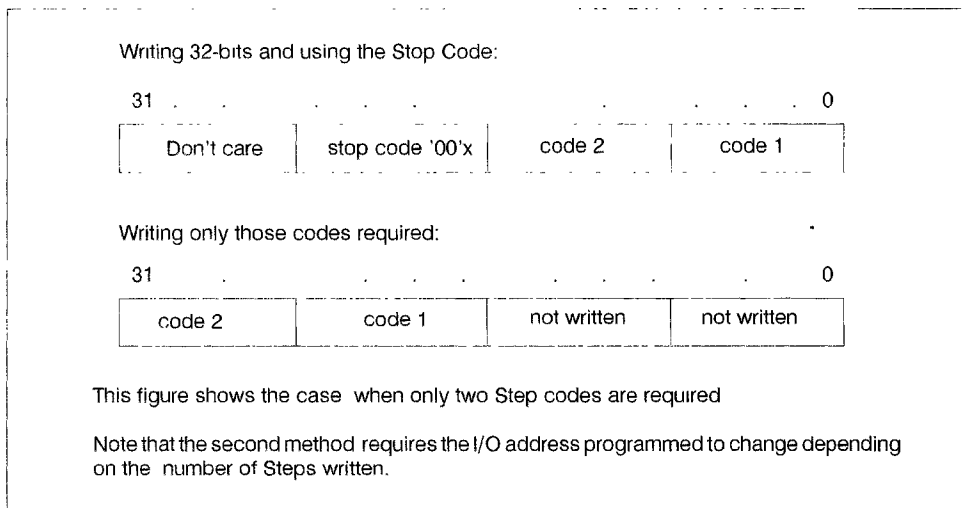


Figure 4.17 Programming less than four Step codes

Line Draw:

The Line Draw function uses the Bresenham line drawing algorithm to draw a line of pixels into the destination. The Bresenham line drawing algorithm operates with all parameters normalized to the first octant (octant 0). The octant code for the actual octant in which the line lies must be specified in the octant field of the Pixel Operation Register. This contains a 3 bit code that is made up of three 1 bit flags called DX, DY and DZ.

DX is 1 for negative X direction, 0 for positive X

DY is 1 for negative Y direction, 0 for positive Y

DZ is 1 for $|X| < |Y|$, 0 for $|X| > |Y|$ ('|X|' is the magnitude of X, the value ignoring the sign)

The octant value is formed by concatenating DX, DY and DZ.

Figure 4.18 shows the encoding of Octants.

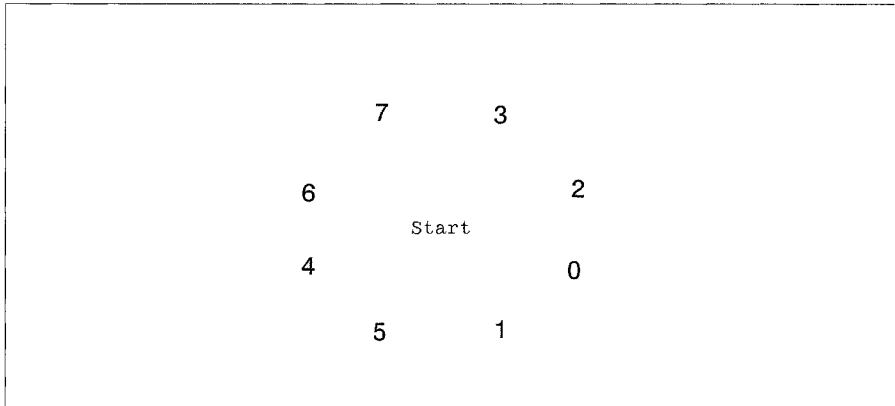


Figure 4 18 Bresenham Line Draw Octant encoding

The length of the line (delta X when normalized) must be specified in the Dimension 1 Register

The Coprocessor provides the following registers to control the Draw Line address stepping:

- Bresenham Error Term, $ET = 2 * \text{deltaY} - \text{deltaX}$
- Bresenham Constant, $K1 = 2 * \text{deltaY}$
- Bresenham Constant, $K2 = 2 * (\text{deltaY} - \text{deltaX})$

On completion of the drawing operation, X and Y pointers point at the last pixel of the line.

The Coprocessor draw operations that take Source data from a Pixel Map apply the specified address update to only one of either the Source or Destination Map. The XY address in the other Map is always **incremented in X only**. There are two possibilities, called Read Draw and Write Draw.

Write Draw After every pixel drawn, the Source XY pointers are incremented in X only. The Destination XY pointers are updated according to the current function specified (either Bresenham Line Draw, or Draw and Step).

Read Draw After every pixel drawn, the Source XY pointers are updated according to the current function specified (either Bresenham Line Draw, or Draw and Step). The Destination XY pointers are incremented in X only.

The 'read' and 'write' in the terms Read Draw and Write Draw refer to the direction of data transfer of the Map that is having its addresses updated by the specified function. So, during a Read Line Draw, the Map from which data is read (the Source) has its addresses updated by the Bresenham Line Draw function. During a Write Draw and Step, the map to which data is written (the Destination) has its addresses updated by the Draw and Step function.

Note, in particular, that to draw a fixed color line (by taking the source from the Foreground and/or Background Registers) a **write** draw function should be used.

Figure 4 19 illustrates the stepping of X and Y pointers during a Read Line Draw and Write Line Draw

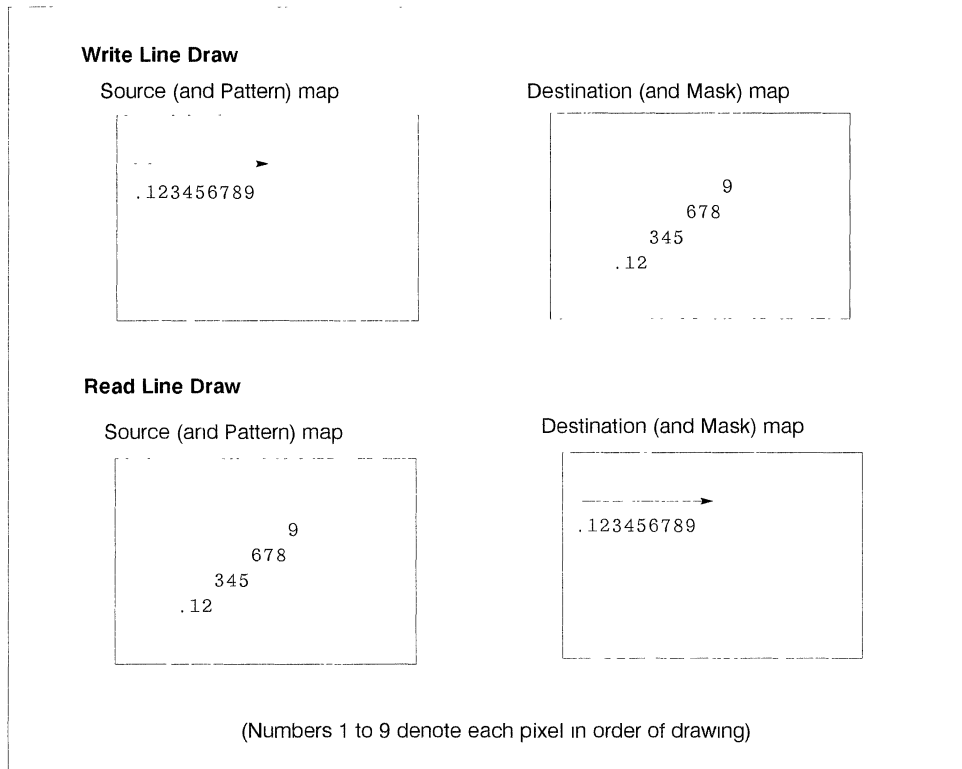


Figure 4.19 Memory to memory Line Draw address stepping.

Note that in the map that is not having the current addressing function applied, the X pointer is always **incremented** regardless of the direction of X in the current addressing function. The Y pointer for the same map is not updated at all during the operation.

The above description refers only to the Source and Destination maps. The Pattern map X and Y pointers are updated in the same manner as the Source pointers, and the Mask map X and Y pointers (that are not directly accessible by the user) are updated in the same manner as the Destination pointers.

If an attempt is made to move any of the map pointers outside the bounds of their current map then the rules set out in 'X and Y Pointers' on page 46 apply as normal; the Source and Pattern pointers wrap, and the Mask and Destination scissor. Thus if it is required to draw a line with a repeating color scheme and pattern, the Source Map Width and Pattern Map Width should be set to the required run length of the repeating colors and pattern respectively. The coprocessor automatically draws the repeating run of colors and pattern. Conversely, if a line with a long non-repeating color scheme or pattern is required, the Source and Pattern Map Widths must be set to equal or exceed the line length, or wrapping occurs.

Drawing with Null Endpoint Pixels: It is common when drawing lines to draw a series of lines one after the other with the endpoint of one line being the starting point of the next line. Such composite lines are called 'polylines'. A problem can arise in that the common endpoint of the two abutting lines is drawn twice, once as the last pixel of the first line, and once as the first pixel of the second line. If, say, a mix of XOR is active, then the common pixel is first drawn and then removed, and similar problems arise with different mixes.

In order to avoid drawing the endpoints of polylines twice, the Coprocessor provides functions that inhibit the drawing of the end pixel of lines. Depending on the function selected, either the first pixel or the last pixel of individual lines is not drawn (drawn 'Null'). The choice of whether to draw first or last pixel null is

arbitrary as long as one or the other is used for the whole figure being drawn. It is usually a convention of the graphics application as to whether first or last pixel null is used.

First and Last pixel null drawing functions are provided for both the Bresenham Line Draw function, and the Draw and Step function. In all cases the programming of parameters is the same as for normal Line Draw and Draw and Step. Only the contents of the Drawing Mode field in the Pixel Operations register are different.

Area Boundary Drawing: The outline of an object is drawn using either Bresenham Line Draw or Draw and Step functions, or a combination of both. The outline is created by observing a number of rules that are detailed below.

The rules for area-boundary line drawing are:

- If a line is drawn from screen top-to-bottom, then draw with Last Pixel Null and draw only the Last Pixel in every horizontal run of pixels.
- If a line is drawn from screen bottom-to-top, then draw with First Pixel Null and draw only the First Pixel in every horizontal run of pixels.
- If a line is Horizontal, then draw none of the pixels.
- Always draw with a mix of XOR.

The Coprocessor implements the above drawing rules in hardware. In order to draw a shape as an area outline, it should be drawn as for a normal Line Draw or Draw and Step operation, but with the 'draw area boundary' Drawing Mode selected in the Pixel Operation Register and a mix of XOR.

Area Outline Scissoring: It is important during area outline drawing to ensure that the correct outline is drawn when the outline intersects the scissor boundary. In particular, when the outline is scissored by a vertical boundary at the left of a map (an XMIN boundary), a pixel is drawn in the outline to activate filling at that boundary.

Using the XGA's combination of Mask Map and fixed Destination Boundary scissoring, area outlines are incorrectly scissored by the Mask Map, but correctly scissored by Destination Map boundary scissoring. The correct area can be filled by ensuring that the Mask Map scissoring is **disabled** when the outline is drawn and **enabled** or **boundary enabled** when the scan/fill part of the area fill is drawn. This results in the correct, scissored figure being drawn. See 'Scissoring with the Mask Map' on page 48.

Pixel Block Transfer (PxBlt):

The PxBlt function transfers a rectangular block of pixels from the Source to the Destination. The width and height of the rectangle are specified in the Dimension 1 and Dimension 2 registers. The transfer can be programmed to start at any of the four corners of the rectangle and proceeds towards the diagonally opposite corner. The address is stepped in the X direction until the edge of the rectangle is encountered whereupon X is reset and the Y direction is stepped. This process is repeated until the entire rectangle has been transferred.

PxBlt's can be implemented in normal WRITE mode or in READ/MODIFY/WRITE mode. This is dependent on the number of bits per pixel and the mix being used.

If the PxBlt is being implemented in READ/MODIFY/WRITE mode (that is, 1,2 or 4 bits per pixel with ANY mix or 8 bits per pixel with a READ/MODIFY/WRITE mix) then either:

- Ensure that the destination map has a base address that is on a double-word (four byte) address boundary, and is an exact number of double-words wide.
- If the destination map is not double-word aligned, ensure that the destination map boundary is not crossed during the PxBlt operation.

PxBlt Direction. The PxBlt direction indicates the direction in which the X, Y address is stepped across the rectangle. It also defines the starting corner of the transfer. This is significant if the destination rectangle

overlaps the source rectangle, and care must be taken to ensure that the PxBlt direction is correctly programmed in such cases to achieve the desired result

This field, when concerned with PxBIts determine the direction that the PxBlt is drawn in

The encoding is as follows

'000'b or '001'b	Start at Top LH corner of Area increasing right and down.
'100'b or '001'b	Start at Top RH corner of Area increasing left and down
'010'b or '011'b	Start at Bottom LH corner of Area increasing right and up.
'110'b or '111'b	Start at Bottom RH corner of Area increasing left and up.

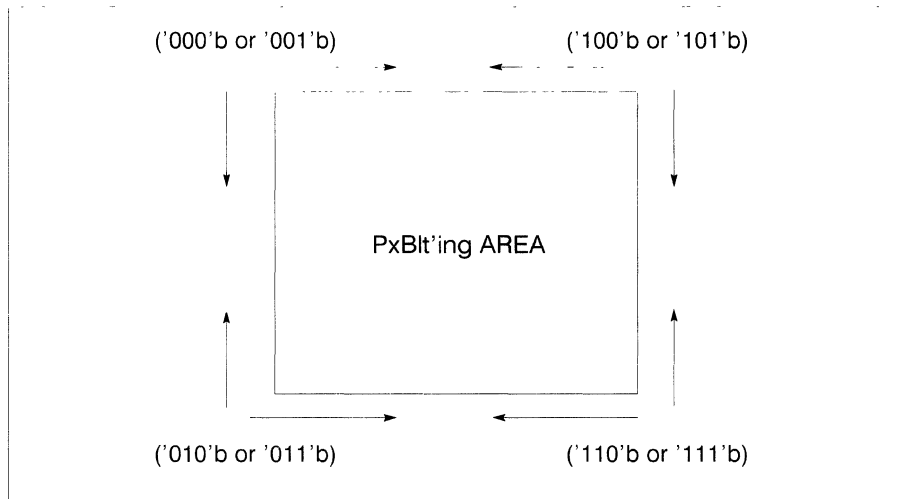


Figure 4.20 PxBlt Direction Codes

After a PxBlt operation has completed, the X and Y pointers are set so that the X pointer contains its original value at the start of the PxBlt, and the Y pointer points to its value on the last line of the PxBlt plus or minus 1, depending on the Y direction that the PxBlt was programmed.

See Section 14.1 for details on PxBIts where the Source and Destinations overlap

Inverting PxBlt: As detailed in 'Map Origin' on page 45, the Coprocessor assumes that the origin of a pixel map is at the top left corner of the map, with Y increasing downwards. Applications which use an origin at the bottom left of the map (Y increasing upwards) use either of the following:

- Modify all Y coordinates by subtracting the map height from them before passing the modified coordinates to the display hardware.
- Use the coprocessor Inverting PxBlt operation.

The Inverting PxBlt use requires the application to draw into an off-screen pixel map without any Y coordinate modification, and then use the Inverting PxBlt operation to move the data to the destination map

Figure 4.21 illustrates the X,Y addressing of the Inverting PxBlt operation, and shows how the result of the Inverting PxBlt appears the same as the original when displayed as an inverted pixel map (i.e. with the origin at the bottom left).

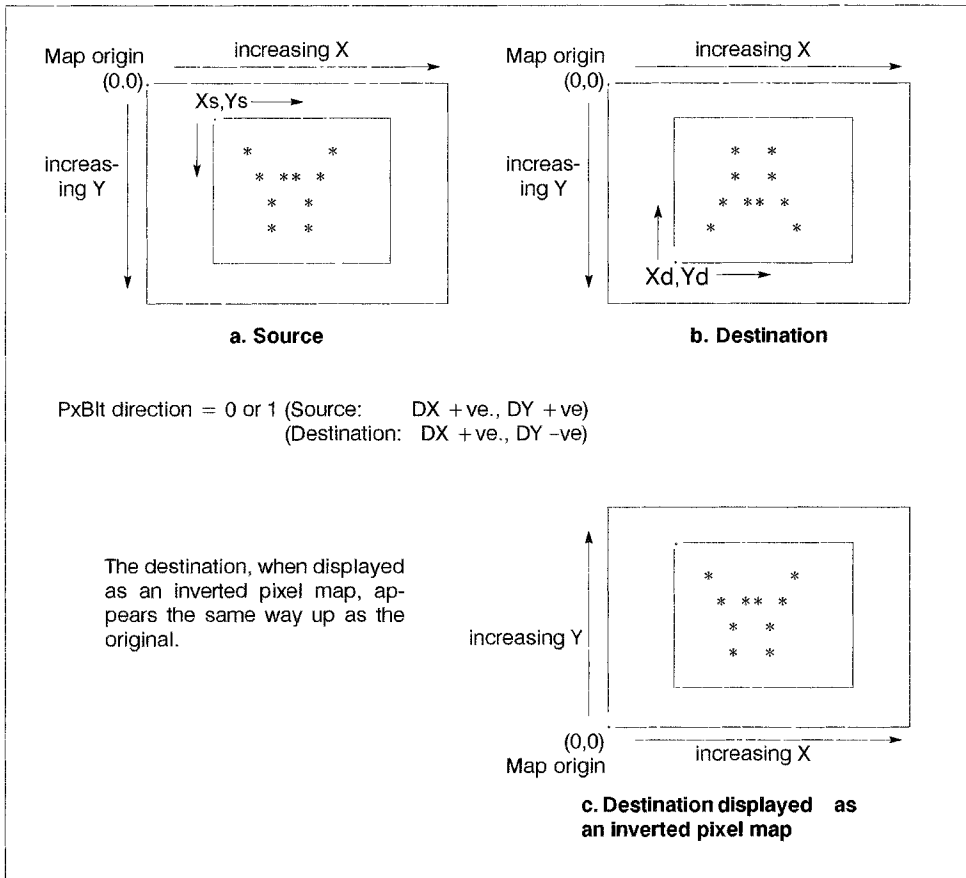


Figure 4.21 Inverting PxBlt

An Inverting PxBlt is set up in the same manner as a standard PxBlt with the following notes:

- The PxBlt direction set by the user applies to the updating of the Source X and Y addresses.
- The Destination Y pointer should be programmed to the opposite (in Y) corner of the Destination rectangle.
- The function field in the Pixel Operation register should be set to Inverting PxBlt as opposed to PxBlt.

See Section 14.1 for details on PxBltS where the Source and Destinations overlap.

Area Fill:

The following steps are required to perform an Area Fill operation without a 'user' pattern:

- 1 Draw the closed outline of the area to be filled using the Area Boundary Drawing Mode. Typically a unique, off-screen pixel map would be defined into which the area boundary would be drawn. This pixel map should be initialized to contain '0' valued pixels before the boundary is drawn. This pixel map must be in a 1 bit-per-pixel format.
- 2 Designate the pixel map in which the area boundary was drawn to be the Pattern Map.

- 3 Specify the desired Destination.
- 4 Select the desired Foreground Mix and Source.
- 5 Specify the Background Mix to be 'Destination (code 5)'.
- 6 Specify the operation direction to be any direction with X increasing (Codes 0 or 1, 2 or 3). This is because the pattern data is scanned from left to right. Selection of a negative X direction code for Area Fill operations results in fill errors.
- 7 Initiate the Area Fill operation.

During the Area Fill operation, the Coprocessor applies a 'filling' function to the Pattern pixels before they are used to select Background and Foreground Sources and Mixes in the usual way. The filling function modifies the Pattern pixels horizontally line by line. It scans the Pattern from left to right, and on encountering the first Foreground (1) pixel, sets all subsequent pixels to Foreground (1) until the next Foreground pixel is encountered. This process is illustrated in Figure 4.22

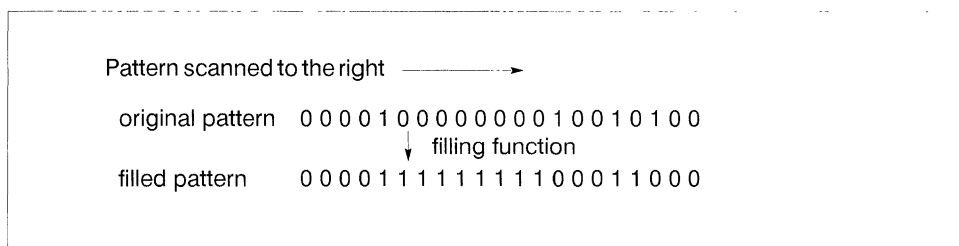


Figure 4.22 Pattern Filling

The 'filled' Pattern is generated internally to the coprocessor. It is then used exactly as the Pattern in any normal operation, with Foreground (1) pixels selecting Foreground Source and Mix, and Background (0) pixels selecting Background Source and Mix. During Area Fill operations, it is desired to fill the specified area and leave all other pixels unchanged. This is why the Background Mix was specified to be 'Destination' in the list above. Figure 4.6 shows the position of the Pattern-filling circuitry in the coprocessor data flow.

Area fill operations that **do** require a pattern fill must be performed in two stages. This is because Area Fill PxBlt operations use the Pattern Map to perform the area fill function and thus cannot include a 'user' pattern in a single operation. However, by first combining the contents of the Mask Map with a mask of the filled area, a full pattern PxBlt of an area can be achieved as follows:

- 1 Both the Pattern Map and the Destination Map should be defined as the map containing the previously-drawn area boundary. The Source map should be defined as the map that would normally supply the Mask map for the operation. The Mask Map facility itself should be disabled. An Area Fill PxBlt should be performed with the following conditions:
 - Foreground Source = Source pixel map
 - Foreground Mix = source (code 3)
 - Background Source = Background Color
 - Background Color = 0
 - Background Mix = source (code 3)

Note that all the maps in this operation should be 1 bit-per-pixel.

This operation combines the Mask data for the pattern Area Fill with a mask of the filled area.

- 2 A second non area fill PxBlt should be performed with the combined mask generated in stage 1 defined as the Mask Map. All other maps can be utilized as normal with no restrictions

4.6.5 Logical And Arithmetic Functions

During an operation in the Coprocessor, Source data is combined with Destination data, under the control of Pattern data, and the result is written back to the Destination. In addition Mask data can be included in the operation to selectively inhibit updating of Destination data

“Source data” can be either Foreground Source or Background Source on a pixel-by-pixel basis. The Foreground Source is combined with the Destination using the Foreground Mix, the Background Source is combined with the Destination using the Background Mix. It is the Pattern that determines whether the Source and Mix are Foreground or Background for a particular pixel. If the Pattern pixel is “1”, Source and Mix are Foreground, if it is “0”, they are Background.

The Foreground and Background Sources can each be either a fixed color over the whole operation, or pixel data taken from the Source Pixel Map. Two fields in the Pixel Operation Register determine whether fixed colors or Source Pixel Map data is used in an operation. The fixed color that can be used as the Foreground Source is called the Foreground Color and is stored in the Foreground Color Register. The fixed color that can be used as the Background Source is called the Background Color and is stored in the Background Color Register.

The possible combinations of Source, Destination and Pattern are shown below:

- Pattern Pixel = 1 (Foreground Source)
 - New Destination Pixel = Old Destination Pixel **Fgd OP** Foreground color
 - New Destination Pixel = Old Destination Pixel **Fgd OP** Pixel Map Source
- Pattern Pixel = 0 (Background Source)
 - New Destination Pixel = Old Destination Pixel **Bgd OP** Background color
 - New Destination Pixel = Old Destination Pixel **Bgd OP** Pixel Map Source

In the above, **Fgd OP** means the logical or arithmetic function specified in the Foreground Mix Register. **Bgd OP** means the logical or arithmetic function specified in the Background Mix Register.

The above operations can be overridden by the contents of the Mask map. If the Mask pixel is 0, the Destination pixel is not modified. If the Mask pixel is 1, the selected operation is applied to the Destination pixel.

Mixes:

The Foreground and Background mixes provided by the XGA are independent. The XGA provides all logical mixes of two operands and six arithmetic mixes. The arithmetic mixes are the ones available through the Adapter Interface. The mixes provided are shown in Table 4.3

Code(hex)	Function			
00		zeros		
01		source	AND	dest
02		source	AND	NOT dest
03		source		
04	NOT	source	AND	dest
05		dest		
06		source	XOR	dest
07		source	OR	dest
08	NOT	source	AND	NOT dest
09		source	XOR	NOT dest
0A	NOT	dest		
0B		source	OR	NOT dest
0C	NOT	source		
0D	NOT	source	OR	dest
0E	NOT	source	OR	NOT dest
0F		ones		
10		maximum		
11		minimum		
12		add with saturate		
13		subtract (dest-source) with saturate		
14		subtract (source-dest) with saturate		
15		average		

Table 4.3 Mixes provided by XGA

Mix Codes 16 hex to FF hex are Reserved.

The term 'saturate' implies that if the result of an arithmetic operation is greater than all '1's, the final result under saturate remains all '1's. Similarly if the result of an arithmetic operation is less than '0', the final result under saturate remains at '0'.

Breaking the ALU Carry Chain:

It may be required to limit the operation of the ALU to certain bits in a pixel (for example to perform an operation on both the upper and lower four bits of an 8 bit pixel independently). In this case the user would not want arithmetic operations to propagate a carry from one group of bits in the pixel to the next. One solution is to use the XGA Pixel Bit Mask to ensure that only one component of the pixel is processed at a time. The disadvantage of this technique is that the operation must be repeated once for each component in the pixel.

The Carry Chain Mask: The XGA provides an alternative mechanism that allows pixels with component fields to be correctly processed in one pass. The user can specify a mask that determines how carry bits are propagated in the ALU (Arithmetic and Logic Unit). By loading the appropriate mask in the Carry Chain Mask Register before performing an operation involving either or both an arithmetic operation or color compare, the user effectively divides the pixel into independent fields. The mask prevents the ALU carry being propagated across the field boundaries.

Each bit in the mask enables or disables the propagation of the carry from the corresponding bit in the ALU to its more significant neighbor. The mask is N-1 bits wide for a pixel N bits wide as the carry from the most significant bit of the ALU is not propagated.

An example Carry Chain Mask for an 8-bit pixel with two 4 bit fields could be:

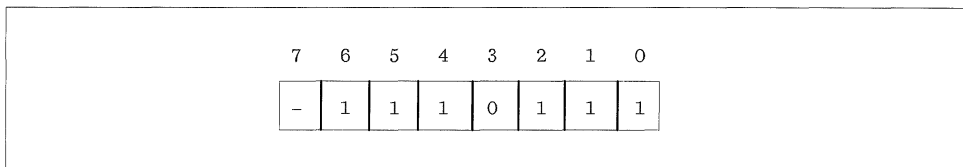


Figure 4 23 Example Carry Chain Mask for an 8 bit pixel

Bits outside the required mask size for a given pixel size need not be written in the register.

Generating The Pattern From The Source:

Pattern data for an operation can be supplied by any one of Pixel Maps A, B or C, or can be fixed to '1' (Foreground Source) throughout the operation. In addition to these four possibilities, Pattern data can also be internally generated by the coprocessor from Source Pixel Map data. A comparison operation is performed on each Source pixel and the Pattern data is generated depending on the result.

The comparison operation compares the Source pixel to '0'. For any Source pixel with a value of '0', a '0' (Background) Pattern pixel is generated. For any non-zero Source pixel, a '1' (Foreground) Pattern pixel is generated. The internally-generated Pattern is then used to select between Foreground and Background Sources and Mixes in the usual way. When the Pattern is internally generated, the coprocessor ignores the Pattern Pixel map contents.

This capability allows the Background Source data and Mix to be forced for all '0' value pixels in the Source. In particular it allows a 'transparency' function to be provided, whereby, for example, a multi-bit character can be drawn onto a Destination with the Destination data 'showing through' any '0' (black) pixel in the Source character definition.

Color Expansion:

If the Source pixels for an operation have fewer bits-per-pixel than the Destination pixels, the Source pixels must be 'expanded' to be the same size as those in the Destination before they are combined. The process is referred to as 'color expansion'.

The major use of Color Expansion is to draw 1 bit-per-pixel character sets on 'n' bit-per-pixel Destinations. The Coprocessor performs this function in hardware, but does not have a Color Expansion Look-Up-Table (LUT). Instead, the 1 bit-per-pixel character map should be defined as the Pattern map. The Pixel Operations Register should be programmed to use Foreground and Background Color Registers and not the Source Map. The Foreground and Background Color registers act as a two-entry Color Expansion LUT in this case, and the character map is correctly expanded to the number of bits-per-pixel in the Destination.

Pixel Bit Masking:

The Pixel Bit Mask allows any combination of bits in a pixel in the Destination to be protected from update (being written). A mask value should be loaded to the Pixel Bit Mask register to selectively enable or disable updating of Pixel bits as required.

This mask is entirely analogous to the Plane Mask in subsystems which are plane as opposed to packed-pixel oriented.

When the Destination bits-per-pixel is less than 8 bits, only the low order bits of the Pixel Bit Mask register are significant.

A bit that is not write enabled is prevented from affecting either arithmetic operations or the underpaint comparison. In effect, masked bits are completely excluded from the operation or comparison.

Color Compare:

The value that the Destination pixels are compared with is stored in the Color Compare Value Register. The Color Compare Condition Register indicates the condition under which the Destination update is inhibited. The possible conditions are shown in Table 4.4.

Condition Code	Condition
0	always true(disable update)
1	Destination data > Color compare value
2	Destination data = Color compare value
3	Destination data < Color compare value
4	always false (enable update)
5	Destination data > = Color compare value
6	Destination data < > Color compare value
7	Destination data < = Color compare value

A comparison result of 'true' prevents update to the Destination.

Table 4.4 Color compare conditions.

4.6.6 Controlling Coprocessor Operations

Starting a Coprocessor Operation:

Coprocessor operations are started by writing the **most significant byte** of the Pixel Operations register. An exception to this is the Draw and Step function, for details see 'Draw and Step' on page 51.

Suspending a Coprocessor Operation:

Coprocessor operations can be suspended before they have completed. The state of the Coprocessor, including internal register contents, can then be rapidly read to allow task state saving. A previous task can be restored through the same data port, and the restored operation restarted.

A field in the Coprocessor Control register is used to suspend and restart Coprocessor operations

Terminating a Coprocessor Operation:

Operations can be terminated before they have completed. The state of the Coprocessor registers that are updated as the operation proceeds is undefined after the operation is terminated, and their contents should thus not be relied upon. Section 4.7 details which registers are updated as an operation proceeds.

A field in the Coprocessor Control register is used to terminate operations

4.6.7 Coprocessor Operation Completion

There are two methods by which the host can detect the completion of a Coprocessor operation:

- 1 Receive an Operation Complete interrupt from the XGA
- 2 Poll the Coprocessor Busy bit in the Coprocessor Control Register

These methods are described below:

Coprocessor Operation Complete Interrupt: The coprocessor provides an Operation Complete Interrupt that can interrupt the system on completion of an operation. The interrupt is enabled by a bit in the 'Interrupt Enable Register (Address: 21x4)' on page 21 and its status is indicated by a bit in the 'Interrupt Status Register (Address: 21x5)' on page 21.

Regardless of the state of the Operation Complete interrupt enable bit, the status bit is always set to '1' on completion of an operation. The application should ensure that this bit is reset before starting an operation. This is done by writing a '1' back to the status bit.

If the interrupt enable bit is '1' then the completion of an operation not only sets the interrupt status bit, but also causes an interrupt to be raised. Again, the host processor should reset the interrupt by writing a '1' back to the status bit after servicing the interrupt.

Coprocessor Busy Bit: The Coprocessor Busy bit in the Coprocessor Control register indicates if the coprocessor is executing an operation. It is set to '1' by the hardware when the Coprocessor is executing an operation and reset to '0' when the operation completes. Applications can poll this bit to determine if the Coprocessor is busy. See Section 14.3

Accesses To The Coprocessor During An Operation:

When the coprocessor is executing an operation, the system processor can only perform read accesses to the coprocessor registers. Write accesses can corrupt operation data and are therefore not permitted.

If the system processor attempts to write data to the coprocessor registers during an operation, the coprocessor allows the access to complete, and the currently executing operation can be corrupted. The coprocessor raises an interrupt to the host system to indicate that a write access occurred during an active operation and that the operation may have been corrupted. This interrupt is called the Coprocessor Access Rejected Interrupt. An enable bit and status bit exist in the the 'Interrupt Enable Register (Address: 21x4)' on page 21 and 'Interrupt Status Register (Address: 21x5)' on page 21 for this interrupt.

There is one exception to this rule. The Coprocessor Control register can be written during an operation. See 'Coprocessor Control Register (Offset: 11)' on page 69.

4.6.8 Coprocessor State Save/Restore

When operating in a multitasking environment it is necessary to save and restore the state of the display hardware when switching tasks

It is possible that a task switch is required when the Coprocessor is in the course of executing an operation. Thus not only the contents of registers visible to the host system but also contents of internal registers (the 'state' of the Coprocessor) must be saved/restored. The Coprocessor has special hardware that allows it to suspend the execution of an operation and efficiently save and restore task states

Suspending Coprocessor Operations:

At any time during the execution of a Coprocessor operation, the operation can be suspended by writing to a bit in the Coprocessor Control Register. Any currently executing memory cycle is completed, after which the Coprocessor suspends the operation. The system can then save and restore the Coprocessor contents as described below, and restart the restored operation by clearing the bit in the Coprocessor Control Register.

4.6.9 Save/Restore Mechanism

The Coprocessor provides two special 32-bit Save/Restore Data Ports through which all the coprocessor state data passes when the state is being saved or restored. The number of double-words that should be read or written is determined by two read only registers: the State Length registers, A and B. The amount of data to be saved/restored is less than 1 Kbyte. State saving software should perform string I/O read instructions, reading data from the two Save/Restore Data Ports in turn into memory. The coprocessor hardware automatically provides successive double-words of data on successive reads. After the state has been saved, the Coprocessor is in a reset state.

Restoring the state of the Coprocessor is performed using a similar process. State data should be moved back into the Coprocessor using string I/O write instructions. The state data should be written back into the Coprocessor in the same order to that in which it was read (that is, first out should be first in).

Note that the exact number of double-words specified in the State Length Registers must be read/written when saving/restoring the Coprocessor state. Failure to do this leaves the Coprocessor in an indeterminate state

4.7 Coprocessor Registers

The XGA Coprocessor supports two register interface formats. The type of interface required (Intel or Motorola) is set when selecting the XGA Extended Graphics Mode in the Operating Mode Register.

The difference between Intel and Motorola formats is that, with two exceptions, the bytes within each four bytes of register space are reversed. Thus byte 0 becomes byte 3. The two exceptions are the Direction Steps register and the Pixel Operation register. The bytes within these registers are not reversed because the byte order is important to the operation being performed.

The majority of the registers are not **directly** readable by the host system. All of those that cannot be read directly can be indirectly read using the Coprocessor State Save/Restore mechanism. See Section 4.6.9. Only the following registers are directly readable by the host system:

- State Save/Restore Data Port
- State Length Registers
- Coprocessor Control register
- Virtual Memory Control register
- Virtual Memory Interrupt Status register
- Current Virtual Address register
- Bresenham Error Term
- Source Map X and Y Pointers
- Pattern Map X and Y Pointers
- Destination Map X and Y Pointers

The contents of most Coprocessor registers are not changed during a Coprocessor operation. Most registers therefore do not need to have their contents reloaded before starting another similar operation. The list below indicates which registers' contents change during an operation.

Bresenham Error Term The Error Term is updated throughout Line Draw operations.

Source Map X and Y The Source map X and Y pointers are updated for any operation that specifies Source Pixel Map as either or both Foreground or Background second operands in the Function 0 or 1 fields in the Pixel Operations register (that is, any operation that uses the Source Pixel Map updates these pointers).

Pattern Map X and Y The Pattern Map X and Y pointers are updated during any operation that uses a Pattern Pixel Map (that is, any operation that does not have the Pattern field in the Pixel Operation register set to 'Foreground').

Destination Map X and Y The Destination Map X and Y pointers are updated during all operations.

The following tables show the Coprocessor register space in both Intel and Motorola formats:

XGA Coprocessor Registers- Intel Register Format				
Coprocessor Address Space				
Byte 3	Byte 2	Byte 1	Byte 0	
Page Directory Base Address				0
Current Virtual Address				4
				8
		State B len	State A len	C
	Pixel Map Index	Coprocessor Control		10
Pixel Map n Base Pointer				14
Pixel Map n Height		Pixel Map n Width		18
		Pixel Map n Format		1C
		Bresenham Error Term		20
		Bresenham K1		24
		Bresenham K2		28
Direction Steps				2C
				30
				34
				38
				3C
				40
				44
	Dest Color Comp Cond.	Bgd Mix	Fgd Mix	48
Destination Color Compare Value				4C
Pixel Bit Mask				50
Carry Chain Mask				54
Foreground Color Register				58
Background Color Register				5C
Operation Dimension 2		Operation Dimension 1		60
				64
				68
Mask Map Origin Y Offset		Mask Map Origin X Offset		6C
Source Map Y Adr		Source Map X Adr		70
Pattern Map Y Adr		Pattern Map X Adr		74
Dest. Map Y Adr		Dest. Map X Adr		78
Pixel Operation				7C

Table 4.5

XGA Coprocessor Registers- Motorola Register Format				
Coprocessor Address Space				
Byte 0	Byte 1	Byte 2	Byte 03	
Page Directory Base Address				0
Current Virtual Address				4
				8
		State B len	State A len	C
Pixel Map Index		Coprocessor Control		10
Pixel Map n Base Pointer				14
Pixel Map n Height		Pixel Map n Width		18
			Pixel Map n Format	1C
Bresenham Error Term				20
Bresenham K1				24
Bresenham K2				28
Direction Steps				2C
				30
				34
				38
				3C
				40
				44
Dest Color Comp. Cond.		Bgd Mix	Fgd Mix	48
Destination Color Compare Value				4C
Pixel Bit Mask				50
Cary Chain Mask				54
Foreground Color Register				58
Background Color Register				5C
Operation Dimension 2		Operation Dimension 1		60
				64
				68
Mask Map Origin Y Offset		Mask Map Origin X Offset		6C
Source Map Y Adr		Source Map X Adr		70
Pattern Map Y Adr		Pattern Map X Adr		74
Dest Map Y Adr		Dest. Map X Adr		78
Pixel Operation				7C

Table 4.6

4.7.1 Register Usage Guidelines

Table 4.5 and Table 4.6 are summaries of the Coprocessor registers for Intel and Motorola format registers respectively.

The following points should be noted when accessing registers detailed in this chapter:

Reserved Register Bits–

- **Register Bits marked with a ‘–’** are reserved and must be masked out if a test is to be performed on the register contents. If non reserved bits of the same register are being updated, these bits must be written to with ‘0’.
- **Register Bits marked with a ‘#’** are reserved and must be masked out if a test is to be performed on the register contents. If non reserved bits of the same register are being updated, these bits must be preserved. Therefore a Read–Modify–Write operation is recommended.

Reserved Registers. Unspecified Registers, or registers marked as Reserved, in the XGA coprocessor address space are reserved. They must not be written to or read from.

Write Only Registers. On a read, the values returned from these registers are Reserved and Unspecified.

Read Only Registers. The contents of these registers must not be modified.

Counters should not be relied upon to wrap from the high value to the low value.

Register fields defined with valid ranges must not be loaded with a value outside the specified range.

Register field values defined as reserved must not be written.

The following sections describe the Coprocessor registers in detail. Unless stated otherwise, the register definitions are in Intel Format.

4.7.2 Virtual Memory Registers

The XGA Coprocessor Virtual Memory Implementation is detailed in Section 5.4.

Page Directory Base Address Register (Coprocessor Registers, Offset: 0)

This register is detailed in Section 5.5.1.

Current Virtual Address Register (Coprocessor Registers, Offset: 4)

This register is detailed in Section 5.5.2.

4.7.3 State Save/Restore Registers

The following registers allow the internal state of the Coprocessor to be efficiently saved and restored. Section 4.6.9 describes this mechanism.

Coprocessor Control Register (Offset: 11)

This register indicates if the coprocessor is currently executing an operation. In addition, the current Coprocessor operation can be terminated or suspended by writing to this register.

The format of this register when writing is as follows:

7	6	5	4	3	2	1	0
-	-	TOp	-	SOp	-	SR	-

The format of this register when reading is as follows:

7	6	5	4	3	2	1	0
BSy	-	TOp	OpS	SOp	-	SR	-

Coprocessor Busy (BSy, bit 7): Reading this bit indicates whether the coprocessor is currently executing an operation. If the Busy bit is '1', the coprocessor is currently executing an operation. If it is '0', the Coprocessor is idle.

Terminate Operation (TOp, bit 5): Coprocessor Operations can be terminated by writing a '1' to the Terminate Operation bit. The application should then ensure that the operation has terminated before proceeding. It can do this either by waiting for the Operation Complete interrupt (if enabled), or it can poll the Coprocessor Busy bit until the coprocessor goes 'Not Busy' (bit = '0').

After the Coprocessor has terminated the operation it automatically clears the Terminate Operation bit to '0'.

The Coprocessor is returned to its initial power-on state, with Coprocessor interrupts masked off, and certain other register bits reset. All registers should be assumed invalid and reprogrammed before another operation is initiated.

Suspend Operation (OpS, SOp, bits 4,3): Coprocessor Operations can be suspended by writing a '1' to the Suspend Operation bit. The Operation Suspended bit is then set to '1' by the Coprocessor when it has suspended the operation. The OpS bit should therefore be polled by the host system processor to ensure that an operation has been suspended before saving/restoring is started.

Writing a '0' to the SOp bit restarts a suspended Coprocessor operation. This should be done to restart a restored operation after a task switch. When the operation restarts, the Coprocessor resets OpS to '0'.

The action of suspending an operation causes the TLB (Translate Look-aside Buffer, Section 5.4.2) to be flushed.

State Save/Restore (SR, bit 1): This bit selects whether to save or restore the Coprocessor state. When set to a '0', a state restore can be performed, when set to a '1', a state save can be performed. The Coprocessor Control Register must be written, with the Suspend Operation bit set and the Save/Restore bit set appropriately before **each** State Save or State Restore.

State Length Registers (Offset: C & D)

These read-only registers return the length, in double-words, of the two parts, A and B, of the Coprocessor State for save and restore.

Save/Restore Data Ports (I/O Index: C & D)

These registers are directly mapped to I/O address space and do not appear in the Coprocessor register summary. However they are Coprocessor registers and are described here.

These registers are used to save and restore the two parts, A and B, of the internal state of the Coprocessor. After a state save/restore is initiated, string I/O reads/writes should be executed from/to these registers. The data can be read/written using any combination of byte, word or dword accesses, provided that the exact number of dwords specified in the State Length registers is read/written. Failure to read/write the correct amount of data leaves the Coprocessor in an indeterminate state

Data should be written back to this port in the same order as it was read (i.e. first out, first in).

4.7.4 Pixel Interface Registers

The following is a detailed description of the Coprocessor PI registers.

Pixel Map Index Register (Offset: 12)

7	6	5	4	3	2	1	0
-	-	-	-	-	-	-	Index

This is a **WRITE ONLY** register

Each Pixel Map used in the XGA is described by four registers, as follows:

- The Pixel Map Base Address register
- The Pixel Map Width register
- The Pixel Map Height register
- The Pixel Map Format register

Each Pixel Map has its own copy of these registers, so there are four copies of these registers in the XGA, one for each of:

- The Mask Map
- Pixel Map A
- Pixel Map B
- Pixel Map C

Only one of these banks of Pixel Map registers is visible to the host system at any time, and the Pixel Map Index register is used to select to which of the Maps the registers apply. The encoding of the 4-bit Pixel Map Index register is shown in Table 4.7.

Pixel Map Index	
00	Mask Map
01	Pixel Map A
10	Pixel Map B
11	Pixel Map C

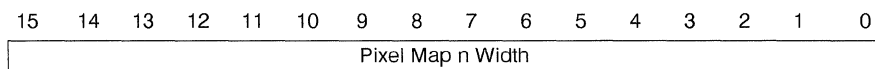
Table 4.7 Pixel Map Index register encoding

Before loading the Pixel Map Base Address, Width, Height, and Format for a particular Map, the programmer must set up the Pixel Map Index register to point to the required Map's registers. For example, to set up Map B's registers, the Pixel Map Index should first be loaded with '10'b

Pixel Map n Base Pointer (Offset: 14)

This is a **WRITE ONLY** register

It specifies the byte address in memory of the start of a Pixel Map. If virtual address mode is enabled, this address is a virtual address, otherwise it is a physical address.

Pixel Map n Width (Offset: 18)

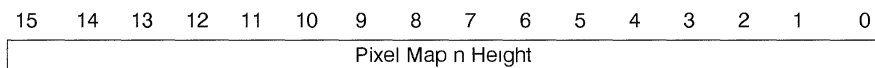
This is a **WRITE ONLY** register and can be loaded with any value in the range 0 to 4095.

It specifies the width of a Pixel Map. The width is measured in pixels, that is, independent of the number of bits/pixel.

Widths are used during address stepping to specify the width of the pixel map. Steps with a Y direction component are achieved by the hardware adding/subtracting the width ± 0 or 1.

The pixel map width is also used for wrapping the Source and Pattern maps, or to implement the fixed scissor boundary around the Destination map.

The value loaded in the width register should be 1 less than the bitmap width. For a bitmap that is 1024 pixels wide, the width register should be loaded with 1023 (hex 03FF).

Pixel Map n Height (Offset: 1A)

This is a **WRITE ONLY** register and can be loaded with any value in the range 0 to 4095.

It specifies the height of a Pixel Map. The height is measured in pixels, that is, independent of the number of bits/pixel.

The pixel map height is used for wrapping the Source and Pattern maps, or to implement the fixed scissor boundary around the Destination map.

The value loaded in the height register should be 1 less than the pixel map height. For a bitmap that is 768 pixels high, the height register should be loaded with 767 (hex 02FF).

Pixel Map n Format (Offset: 1C)

7	6	5	4	3	2	1	0
-	-	-	-	PO	PS		

This register is a **WRITE ONLY** register.

It specifies the format of a Pixel Map as detailed in the table below:

Pixel Order		
PO	0	Intel Order
	1	Motorola Order
Pixel Size		
PS	000	1 bit
	001	2 bits
	010	4 bits
	011	8 bits
	100	Reserved
	101	Reserved
	110	Reserved
	111	Reserved

Motorola/Intel Format (PO, Bit 3): This bit selects the format for the memory-to-screen mapping. When set to '0', the pixel map is Intel-ordered; when set to '1', the pixel map is Motorola-ordered. Section 4.6 describes the difference in formats.

Pixel Size (PS, Bits 2-0): This field specifies the number of bits/pixel in the pixel map. Pixel maps occupied by the Source or Destination map can be 1, 2, 4, or 8 bits-per-pixel. The Pixel map occupied by the Pattern map **must** be 1 bit-per-pixel. Programming the Pattern to be taken from a Pixel Map that does not contain 1-bit pixels produces undefined results.

Pixel Maps A, B and C

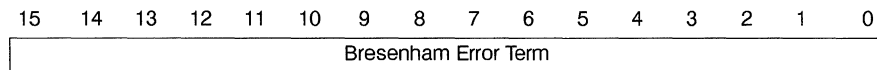
Pixel Maps A, B and C are all described by similar registers. The different maps are merely 3 instances of pixel maps that can have different locations in memory, sizes and formats.

It should be remembered that the Pattern map used by the XGA must be 1 bit-per-pixel. It is the responsibility of the user to ensure that the Pattern map resides in a Pixel Map that is 1 bit-per-pixel. Failure to do this produces undefined results.

Mask Map

The Mask Map has a Base Pointer, Width, and Height that are similar to those of Pixel Maps A, B, and C.

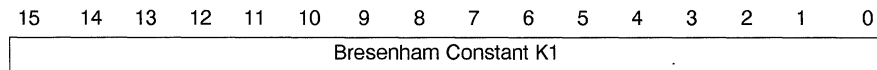
The Mask Map Format register differs from Maps A, B, and C in that only the Motorola/Intel format bit of the Mask Map is programmable by the user. This acts in the same way as the bit for Maps A, B and C. The number of bits-per-pixel is assumed to be 1 bit-per-pixel. However, the bits-per-pixel should always be programmed to 1 bit-per-pixel to ensure future compatibility.

Bresenham Error Term E (Offset: 20)

This register can be written and read.

It specifies the Bresenham Error Term for the Draw Line function. The value is a signed quantity, calculated as $((2 \cdot \Delta Y) - \Delta X)$ after normalization to first octant.

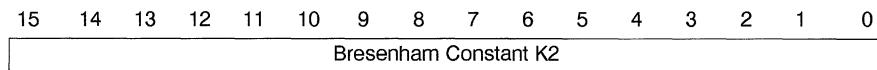
This register **must** be written as a 16-bit sign extended two's complement number in the range -8192 to $+8191$.

Bresenham Constant K1 (Offset: 24)

This register is a **WRITE ONLY** register.

It specifies the Bresenham Constant, K1, for the Draw Line function. The value is a signed quantity, calculated as $2 \cdot \Delta Y$ after normalization to first octant.

This register **must** be written as a 16-bit sign extended two's complement number in the range -8192 to $+8191$.

Bresenham Constant K2 (Offset: 28)

This register is a **WRITE ONLY** register

It specifies the Bresenham Constant, K2, for the Draw Line function. The value is a signed quantity, calculated as $2 \cdot (\Delta Y - \Delta X)$ after normalization to first octant.

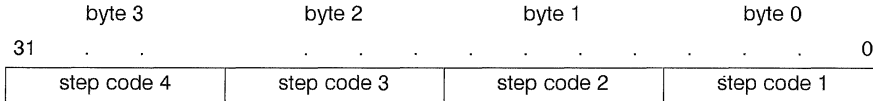
This register **must** be written as a 16-bit sign extended two's complement number in the range -8192 to $+8191$.

Direction Steps Register (Offset: 2C)

This is a **WRITE ONLY** register.

The byte order of this register is independent of whether the Intel or Motorola register interface is enabled. For convenience, this register is also shown as it would appear in a 32 bit register in a Motorola style processor.

Intel View of Register



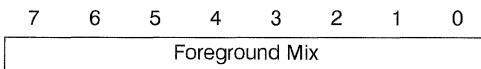
Motorola View of Register



This register is be used to specify up to 4 Draw and Step codes to the coprocessor and to initiate a Draw and Step operation.

The action of writing data to byte 3 of this register initiates a Draw and Step operation. Therefore a Draw and Step operation can be initiated by a single 32 bit access, by two 16 bit accesses where bytes 2 & 3 are written last, or by four byte accesses where byte 3 is written last. If multiple Draw and Step operations are required with the same Draw and Step codes, the operation can be initiated by simply writing to byte 3.

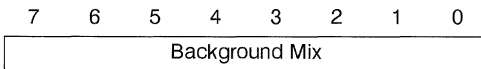
Before initiating a Draw and Step operation, the Pixel Operation Register must be configured to set up the data path and flags for the Draw and Step operation. See Figure 4.14 for full details

Foreground Mix Register (Offset: 48)

This is a **WRITE ONLY** register

It holds the foreground mix value that specifies a logic or arithmetic function to be performed between the Destination and Function 1 second operand pixels during an operation where the Pattern pixel value is 1.

See Section 4.6.5 for details and mix functions available.

Background Mix Register (Offset: 49)

This is a **WRITE ONLY** register.

It holds the background mix value that specifies a logic or arithmetic function to be performed between the Destination and Function 0 second operand pixels during an operation where the Pattern pixel value is 0.

See Section 4.6.5 for details and mix functions available.

Destination Color Compare Condition (Offset: 4A)

7	6	5	4	3	2	1	0
-	-	-	-	-	Condition		

This is a **WRITE ONLY** register.

Condition (Bits 2-0): This three bit field specifies the Destination Color Compare Condition under which Destination update is inhibited. The Condition is encoded as follows

Destination Color Compare Condition	
000	Always true (disable update)
001	Dest > col comp value
010	Dest = col comp value
011	Dest < col comp value
100	Always false (enable update)
101	Dest > = col comp value
110	Dest < > col comp value
111	Dest < = col comp value

Destination Color Compare Value (Offset: 4C)

31	0
Destination Color Compare Value	

This register is a **WRITE ONLY** register.

It contains the comparison value with which the Destination pixels are compared when Color Compare is enabled. Only the corresponding number of bits-per-pixel in the Destination are required in this register (for example, if the Destination is 4 bits-per-pixel, only the 4 low order bits of this register are used). Therefore the bits of this register more significant than the number of bits per pixel need not be written.

See page 62 for details of the Color Compare function.

Pixel Bit Mask (Plane Mask) (Offset: 50)

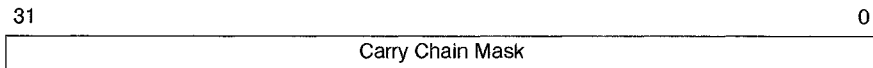
31	0
Pixel Bit Mask	

This register is a **WRITE ONLY** register.

It determines which bits within each pixel are subject to update by the coprocessor. A '1' means the corresponding bit is **enabled** for updates. A '0' means the corresponding bit is not updated.

A bit that is not write enabled is prevented from affecting either arithmetic operations or the Destination Color Compare comparison. In effect, masked bits are completely excluded from the operation or comparison. Only the corresponding number of bits-per-pixel in the Destination are required in this register (for example, if the Destination is 4 bits-per-pixel, only the 4 low order bits of this register are used). Therefore the bits of this register more significant than the number of bits per pixel need not be written.

See page 62 for details of the Pixel Bit Mask Function.

Carry Chain Mask (Offset: 54)

This register is a **WRITE ONLY** register

It contains a mask up to 31 bits wide. The mask used to specify how the carry chain of the ALU is propagated when performing arithmetic update mixes and color compare operations

'0' in the mask means that the carry out of this bit position of the ALU is not to be propagated to the next significant bit position. A '1' in the mask means that propagation is to take place. Therefore the pixel value can be 'split' into sections within the pixel.

Only the corresponding number of bits-per-pixel in the Destination are required in this register (for example, if the Destination is 4 bits-per-pixel, only the 4 low order bits of this register are used). Therefore the bits of this register more significant than the number of bits per pixel need not be written. Note that there is no carry out of the most-significant bit of the Pixel irrespective of the setting of the corresponding Carry Chain mask bit.

See page 61 for details on the Carry Chain function.

Foreground Color Register (Offset: 58)

This register is a **WRITE ONLY** register.

It holds the foreground color to be used during Coprocessor operations. The foreground color can be specified as the Foreground Source by setting up the appropriate field in the Pixel Operation Register.

Only the corresponding number of bits-per-pixel in the Destination are required in this register (for example, if the Destination is 4 bits-per-pixel, only the 4 low order bits of this register are used) Therefore the bits of this register more significant than the number of bits per pixel need not be written

Background Color Register (Offset: 5C)

This register is a **WRITE ONLY** register.

It holds the background color to be used during Coprocessor operations. The background color can be specified as the Background Source by setting up the appropriate field in the Pixel Operation Register.

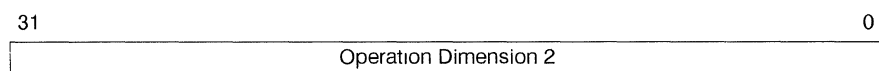
Only the corresponding number of bits-per-pixel in the Destination are required in this register (for example, if the Destination is 4 bits-per-pixel, only the 4 low order bits of this register are used) Therefore the bits of this register more significant than the number of bits per pixel need not be written.

Operation Dimension 1 (Offset: 60)

This register is a **WRITE ONLY** register.

It specifies the width of the rectangle to be drawn by the PxBlt function, or the length of line in a line draw operation. The value is an unsigned quantity, and should be 1 less than the required width. Thus to draw a line 10 pixels long, the value 9 should be written to this register

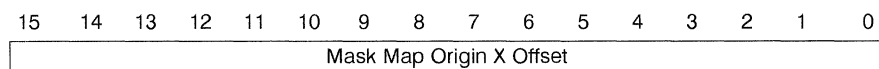
The value written to this register must be within the range 0 to 4095.

Operation Dimension 2 (Offset: 62)

This register is a **WRITE ONLY** register

It specifies the height of the rectangle to be drawn by the PxBlt function. The value is an unsigned quantity, and should be 1 less than the required height. Thus to draw a rectangle 10 pixels high, the value 9 should be written to this register.

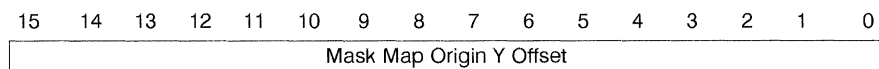
The value written to this register must be within the range 0 to 4095.

Mask Map Origin X Offset (Offset: 6C)

This register is a **WRITE ONLY** register

It specifies the X offset of the Mask Map origin relative to the origin of the Destination Map

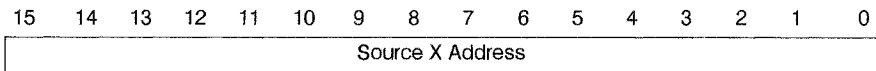
The value written to this register must be within the range 0 to 4095.

Mask Map Origin Y Offset (Offset: 6E)

This register is a **WRITE ONLY** register

It specifies the Y offset of the Mask Map origin relative to the origin of the Destination Map.

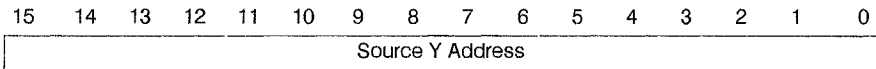
The value written to this register must be within the range 0 to 4095.

Source X Address (Offset: 70)

This register can be written and read.

It specifies the X coordinate of the Coprocessor operation Source pixel.

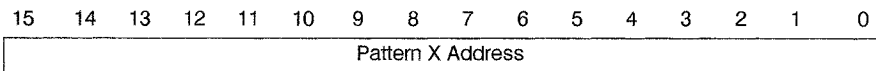
The value written to this register must be within the range 0 to 4095.

Source Y Address (Offset: 72)

This register can be written and read.

It specifies the Y coordinate of the Coprocessor operation Source pixel.

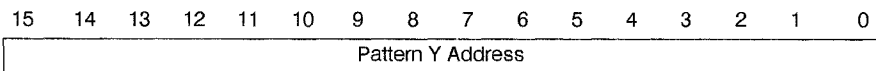
The value written to this register must be within the range 0 to 4095.

Pattern X Address (Offset: 74)

This register can be written and read.

It specifies the X coordinate of the Coprocessor operation Pattern pixel.

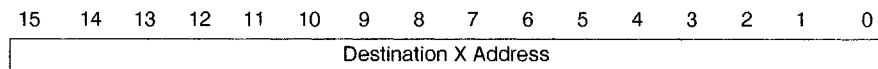
The value written to this register must be within the range 0 to 4095.

Pattern Y Address (Offset: 76)

This register can be written and read.

It specifies the Y coordinate of the Coprocessor operation Pattern pixel.

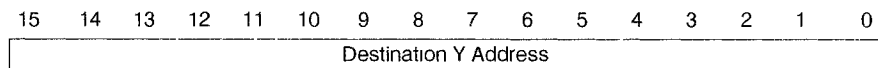
The value written to this register must be within the range 0 to 4095.

Destination X Address (Offset: 78)

This register can be written and read.

It specifies the X coordinate of the Coprocessor operation Destination pixel. The Destination X coordinate can be negative if required.

This register **must** be written as a 16 bit sign extended two's complement number in the range -2048 to +6143

Destination Y Address (Offset: 7A)

This register can be written and read.

It specifies the Y coordinate of the Coprocessor operation Destination pixel. The Destination Y coordinate can be negative if required

This register **must** be written as a 16 bit sign extended two's complement number in the range -2048 to +6143

Pixel Operations Register (Offset: 7C)

This register is a **WRITE ONLY** register.

It is used to define the flow of data during an operation, specifies the address update function that is to be performed, and initiates PxBit and Line Draw operations

The byte order of this register is independent of whether the Intel or Motorola register interface is enabled. For convenience, this register is also shown as it would appear in a 32 bit register in a Motorola style processor.

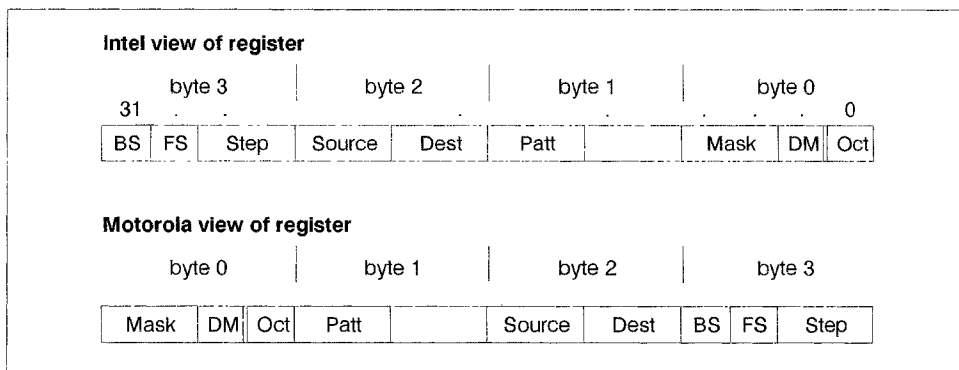


Figure 4.24 Pixel Operation Register fields

The Pixel Operation Register is a 32-bit register that controls the function of the Coprocessor. Its contents define address update operation to be performed and the path of data during the operation.

All operations, with the exception of Draw and Step, are initiated by writing to the **most significant byte** of the Pixel Operations register. Therefore an operation can be initiated by a single 32 bit write, two 16 bit writes where bytes 2 and 3 are written last, or 4 byte writes where byte 3 is written last. The contents of the Pixel Operation register are preserved throughout an operation.

The field in the Pixel Operation Register are coded as follows.

Background Source (BS, bits 31–30): These bits determine the Background Source that is to be combined with the Destination when the Pattern pixel equals 0 (Background mix used).

Background Source	
00	Background color
01	- reserved -
10	Source pixel map
11	- reserved -

Foreground Source (FS, bits 29–28): These bits determine the Foreground Source that is to be combined with the Destination when the Pattern pixel equals 1 (Foreground mix used).

Foreground Source	
00	Foreground color
01	Reserved
10	Source pixel map
11	Reserved

Step function (Step, bits 27–24): These 4 bits determine how the Coprocessor address is modified as pixel data are manipulated. These bits could be regarded as the Coprocessor function code. It is writing to these bits that starts the Coprocessor operation, except for Draw and Step functions. Draw and Step operations are started by writing to the Direction Steps register.

Step	
0000	Reserved
0001	Reserved
0010	Draw and Step Read
0011	Line Draw Read
0100	Draw and Step Write
0101	Line Draw Write
0110	Reserved
0111	Reserved
1000	PxBlt
1001	Inverting PxBlt
1010	Area Fill PxBlt
1011	Reserved
.	
1111	Reserved

Source Pixel Map (Source, bits 23–20): These 4 bits determine the location of pixel map Source data. The combination of these bits and the Foreground and Background Source fields determine the data that is to be used as the Source data for ALU functions.

Source	
0000	Reserved
0001	Pixel Map A
0010	Pixel Map B
0011	Pixel Map C
0100	Reserved
1111	Reserved

Destination Pixel Map (Dest, bits 19–16): These 4 bits determine the location of Destination data to be modified during an operation

Destination	
0000	Reserved
0001	Pixel Map A
0010	Pixel Map B
0011	Pixel Map C
0100	Reserved
1111	Reserved

Pattern Pixel Map (Patt, bits 15–12): These 4 bits determine the Pattern data to be used during an operation. Code 1000 causes the Coprocessor to assume that the Pattern is 1 across the whole operation and therefore to use the Foreground function on all pixels. This effectively turns off the use of the Pattern. Code 1001 causes the Pattern to be generated from Source data. Every "0" pixel in the Source generates a Background Pattern pixel, every non-zero pixel in the Source generates a Foreground Pattern pixel.

Step	
0000	Reserved
0001	Pixel Map A
0010	Pixel Map B
0011	Pixel Map C
0100	Reserved
0101	Reserved
0110	Reserved
0111	Reserved
1000	Foreground (fixed)
1001	Generated from Source
1010	Reserved
1111	Reserved

Mask Pixel Map (Mask, bits 7–6): These bits determine how the Mask Map is used. See page 48 for details of Mask Map modes.

Mask	
00	Mask Map Disabled
01	Mask Map Boundary Enabled
10	Mask Map Enabled
11	Reserved

Drawing Mode Register (DM, bits 5–4): This 2-bit field determines the attributes of Line Draw and Draw and Step operations.

Drawing Mode	
00	draw all pixels
01	draw first pixel null
10	draw last pixel null
11	draw area boundary

Direction Octant (Oct, bits 2–0): This 3-bit field specifies the Octant for Line Draw and PxBit operations. The coding of the octant is illustrated for Line Draw in Figure 4.18, and for PxBit in Figure 4.20.

Direction Octant Bit Definition	
0	DZ
1	DY
2	DX

5 XGA System Interface

5.1 Multiple Instances

Up to eight instances of an XGA subsystem can be installed in a system. The addressing of the I/O registers, Memory Mapped Registers, and video memory for each instance is controlled by the contents of the XGA POS registers. See Section 5.2.

5.1.1 Multiple XGA Subsystems in VGA Mode

The VGA has only one set of addresses allocated to it. Therefore it is not possible to have multiple XGA subsystems in VGA mode, responding to update requests, simultaneously. However, more than one XGA subsystem may be in VGA mode as long as only one has VGA address decoding enabled using the Operating Mode Register (Address: 21x0). Subsystems with VGA address decoding disabled continue to display the correct picture. See Section 6 for further information.

Note: At no time should the XGA be disabled using the card enable bit in the XGA POS registers.

5.1.2 Multiple XGA Subsystems in 132 Column Text Mode.

When in 132 Column Text Mode, the XGA responds to VGA address decodes. Therefore, the same rules apply as for Multiple XGA Subsystems in VGA mode. See Section 6 for further information.

5.1.3 Multiple XGA Subsystems in Extended Graphics Mode

The Extended Graphics Modes are controlled by a bank of 16 I/O registers. These registers are located in one of eight possible locations. As a result up to eight XGA subsystems can be installed in a system. Each 'instance' of XGA installed is positioned at a unique I/O and memory location and so each can be used independently in the system. See Section 6 for details on controlling multiple XGAs.

Similarly the XGA Coprocessor memory-mapped registers occupy a bank of 128 contiguous register addresses that are mapped in memory space. These registers can also be relocated allowing up to 8 instances of the XGA coprocessor to coexist in a system.

The locations of these registers are controlled by the XGA POS registers. See Section 5.2 for the register details and see Section 7 for programming considerations on reading and using the data contained in them.

5.2 XGA POS Registers

The XGA subsystem has movable I/O addresses for the display controller, allowing more than one XGA subsystem to be installed in a system.

All the POS registers detailed in this section are set-up during system configuration and **must never** be written. All the registers are specified relative to a 'Base' address. Details of how to locate the base address and read the registers are given in Section 7.

5.2.1 Register Usage Guidelines

- All registers are 8 bits long.
- All registers are READ ONLY.
- All undefined register bits (marked with '-') should be masked out if the register contents are being tested.
- All Reserved register bits (marked with '#') should be masked out if the register contents are being tested.

5.2.2 Subsystem Identification Low Byte (Base + 0)

When read this register returns 'DB' hex as data.

5.2.3 Subsystem Identification High Byte (Base + 1)

When read this register returns '8F' hex as data.

5.2.4 POS Register 2 (Base + 2)

The fields in this register are as follows:

7	6	5	4	3	2	1	0
ROM Addr				IODA		EN	

XGA Enable (EN, Bit 0)

This bit when '1' identified that the subsystem is enabled for address decoding for all non POS addresses. When '0', only POS registers can be accessed, all other accesses to the subsystem have no effect.

I/O Device Address (IODA, Bits 1-3)

This field specifies which set of I/O addresses has been allocated to the Display Controller Registers. The lowest address of each set of addresses is referred to as the 'I/O Base Address'.

IODA	I/O Base Address (hex)
000	2100
001	2110
010	2120
011	2130
100	2140
101	2150
110	2160
111	2170

Table 5.1

ROM Address (ROM Addr, Bits 4-7)

This field specifies which of sixteen possible 8 Kbyte memory locations has been assigned to the XGA ROM. The ROM occupies the first 7 Kbytes of this 8 Kbyte block, the other 1 Kbyte being occupied by the coprocessor memory-mapped registers.

The 'IODA' field above, specifies which 128 byte section within this 1 Kbyte block is allocated to the subsystem. For example XGA instance 2 has its coprocessor registers located in the third 128 byte section of the 1 Kbyte block. See Table 5.2.

ROM Address Field	ROM Address Range (hex)		Coprorocessor Register Base Address (hex)					
			Instance:					
			1	2	3	4	5	6
0000	C0000	C1BFF	C1C80	C1D00	C1D80	C1E00	C1E80	C1F00
0001	C2000	C3BFF	C3C80	C3D00	C3D80	C3E00	C3E80	C3F00
0010	C4000	C5BFF	C5C80	C5D00	C5D80	C5E00	C5E80	C5F00
0011	C6000	C7BFF	C7C80	C7D00	C7D80	C7E00	C7E80	C7F00
0100	C8000	C9BFF	C9C80	C9D00	C9D80	C9E00	C9E80	C9F00
0101	CA000	CB BFF	CBC80	CBD00	CBD80	CBE00	CBE80	CBF00
0110	CC000	CDBFF	CDC80	CDD00	CD D80	CDE00	CDE80	CDF00
0111	CE000	CFBFF	CFC80	CFD00	CFD80	CFE00	CFE80	FFF00
1000	D0000	D1BFF	D1C80	D1D00	D1D80	D1E00	D1E80	D1F00
1001	D2000	D3BFF	D3C80	D3D00	D3D80	D3E00	D3E80	D3F00
1010	D4000	D5BFF	D5C80	D5D00	D5D80	D5E00	D5E80	D5F00
1011	D6000	D7BFF	D7C80	D7D00	D7D80	D7E00	D7E80	D7F00
1100	D8000	D9BFF	D9C80	D9D00	D9D80	D9E00	D9E80	D9F00
1101	DA000	DBBFF	DBC80	DBD00	DBD80	DBE00	DBE80	DBF00
1110	DC000	DCBFF	DDC80	DDD00	DDD80	DDE00	DDE80	DDF00
1111	DE000	DFBFF	DFC80	DFD00	DFD80	DFE00	DFE80	DF F00

Table 5.2 XGA ROM, Memory Mapped Register Assignments

5.2.5 POS Register 4 (Base + 4)

7	6	5	4	3	2	1	0
Video Memory Base							VE

Video Memory Base Address (Bits 7-1)

This register contains the most significant 7 bits of the address at which the XGA memory is located. Three more bits are provided by the I/O Device Address in POS byte 1. This gives a Video Memory Base address on a 4 Mbyte boundary.

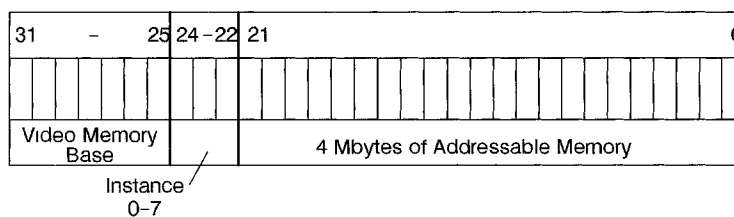


Figure 5.1 XGA Video Memory Base Address.

For example, if the Video Memory Base address is set to 1 and I/O Device Address 6 (instance 6) has been selected, the XGA Video Memory is located, starting at '03800000'h (see diagram above).

Video Memory Enable (VE, Bit 0)

This bit signifies whether the 4 Mbyte Aperture is available for use. When this bit is set to '0'b the 4 Mbyte Aperture is disabled, and when set to '1'b the 4 Mbyte Aperture is enabled.

5.3 POS register 5 (Base + 5)

7	6	5	4	3	2	1	0
#	#	#	#	1 Mbyte Base			

1 Mbyte Aperture Base Address (1 Mbyte Base, Bits 3-0)

This field specifies where the 1 Mbyte Aperture has been positioned in system address space or if the aperture has been disabled. The following table describes the use of this field.

1 Mbyte Base	1 Mbyte Aperture Lochn.(hex)
0000	Disabled
0001	00100000
0010	00200000
0011	00300000
0100	00400000
0101	00500000
0110	00600000
0111	00700000
1000	00800000
1001	00900000
1010	00A00000
1011	00B00000
1100	00C00000
1101	00D00000
1110	00E00000
1111	00F00000

5.4 Virtual Memory Description

The XGA Coprocessor can address either real or virtual memory. When addressing real memory, the linear address calculated by the Coprocessor is passed directly to the host system or local video memory. When addressing virtual memory, the linear address from the Coprocessor is translated by on-chip Virtual Memory Translation logic before the translated address is passed to the host system, or local Video Memory. Virtual Address Translation is enabled or disabled by a control bit in the XGA

The Coprocessor uses two levels of tables to translate the linear address from the Coprocessor to a physical address. Addresses are translated through a Page Directory and Page Table to generate a physical address to memory pages that are 4 Kbytes in size. The Page Directory and Page Tables are of the same form as those used by the 80386 processor Paging Unit.

5.4.1 Address Translation

The linear address from the Coprocessor is divided into 3 fields that are used to look-up the corresponding physical address. The fields are called the Directory Index, the Table Index and the Offset, and are illustrated in Figure 5.2

31	22	21	12	11	0
Directory Index		Table Index		Offset	

Figure 5.2 Linear Address Fields

The location of the Page Directory is at a fixed physical address in memory that must be on a page (4 Kbyte) address boundary. The Coprocessor has a Page Directory Base Address register that should be loaded with the address of the Page Directory Base.

The translation process is illustrated in Figure 5.3.

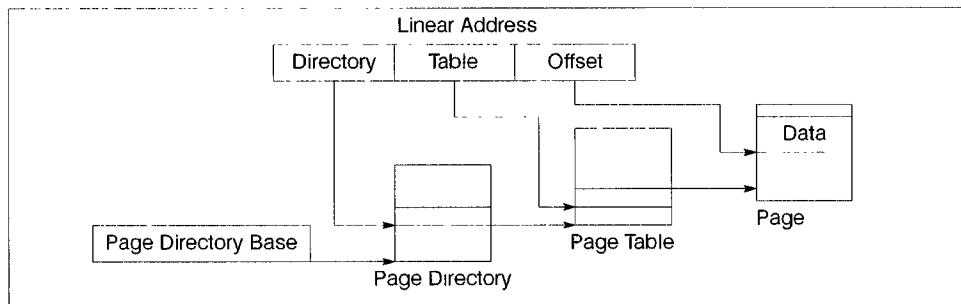


Figure 5.3 Linear to Physical Address Translation

The Directory Index field of the linear address is used to index into the Page Directory. The entry read from the Page Directory contains a 20-bit Page Table address and some statistical information in the low order bits.

The 20-bit Page Table address points to the base of a Page Table in memory. The Table Index field in the Linear address is used to index into the Page Table. The entry read from the Page Table contains a 20 bit Page Address and some statistical information in the low order bits.

The 20-bit Page address points to the base of a 4 Kbyte page in memory. The Offset field in the Linear address is used to index into the Page. The entry read from the Page contains the actual data required by the memory access.

Page Directory and Page Table Entries

The entries of the Page Directory and Page Table are very similar. The format of an entry is shown below.

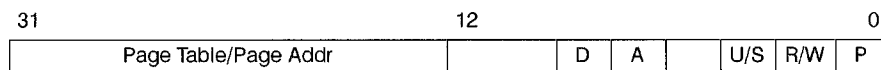


Figure 5.4 Page Directory and Page Table Entry

The top 20 bits of the entry are either the Page Table address or the Page address. The low order bits are as follows:

Dirty Bit (D, bit 6). This bit is set before a write to an address covered by that Page Table entry occurs. The D bit is undefined for Page Directory entries.

Accessed Bit (A, bit 5). This bit is set for both types of entry before a read or write access occurs to an address covered by the entry.

User/Supervisor and Read/Write (U/S, R/W bits 2,3). These bits prevent unauthorized use of Page Directory and Page Table entries. Accesses by the Coprocessor can be defined as a Supervisor or User access depending on the status of the application using the Coprocessor. The access type is defined by a bit in the VM Control Register. If the access is defined as Supervisor, no protection is provided and all accesses to the Page Directory and Page Tables are permitted.

If the access is a User access, the U/S and R/W bits are checked to ensure that access to that entry is permitted. The meaning of these bits is shown below.

U/S	R/W	Access rights of User
0	0	Access not permitted
0	1	Access not permitted
1	0	Reads permitted, Writes not permitted
1	1	Reads and Writes permitted

Table 5.3 Page Directory and Page Table access rights in User mode

Present bit (P, bit 0). The Present bit indicates whether a Page Directory or Page Table entry can be used in translation. If the bit is set it indicates that the Page Table or Page to which the entry refers is present in memory

5.4.2 The XGA Implementation of Virtual Memory

The XGA Coprocessor operates with a Page Directory and Page Tables in the format described above. The Coprocessor contains its own internal cache of translated addresses that avoids it having to perform the two-stage translation process on every Coprocessor access. In the following text this cache is referred to as a Translate Look-aside Buffer or TLB.

The TLB

The TLB is shown in the following figure.

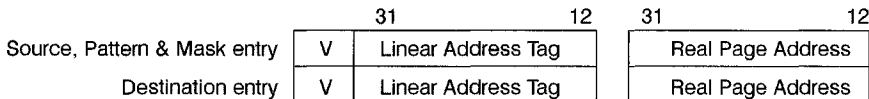


Figure 5.5 Translate Look-aside Buffer (TLB). The 'V' bits are the entry Valid bits.

The TLB has two entries, one entry for the Source, Pattern and Mask Pixel Maps and another for the Destination Pixel Map. Each entry is specifically reserved for use by one of these Maps. Each entry in the TLB contains the top 20 bits of a Linear address (the 'Address Tag'), an 'Entry Valid' flag bit, and the top 20 bits of the Physical address (the 'Real Page address') corresponding to that linear address. When a Linear address is passed from the Coprocessor to the Virtual Address hardware, the top 20 bits of the Linear address are first compared against the appropriate TLB entry Address Tag. If they match and the TLB entry flag bit is 'Valid', the Real Page Address in that TLB entry is used as the top 20 bits of the Physical address for that access. The bottom 12 bits of the Physical address are provided from the bottom 12 bits of the linear address (the Offset).

If the Linear address from the Coprocessor matches the Address Tag in the TLB for the particular map in use, the access is said to have caused a TLB "hit". If the Tag does not match, a TLB 'miss' occurs.

Actions that Flush the TLB. The TLB contents are flushed by the hardware under the following circumstances:

- Whenever the Page Directory Base Address Register is written
- Whenever a Coprocessor operation is suspended (by setting the Suspend Operation bit in the Coprocessor Control Register. (See page 69)).

TLB Misses

If a TLB miss occurs, the Coprocessor automatically performs the two-level translation required to form the required Page Address. The contents of the XGA Page Directory Base Address Register are used to access

the appropriate Page Directory entry, that is in turn used to access the appropriate Page Table entry. The Real Page Address that results from the translation process is stored in the TLB for use by subsequent accesses that address the same Page

Memory access performed by the Coprocessor can be categorized as follows:

Read accesses Performed on the Source, Pattern, Mask, and Destination Maps

Write accesses Performed only on the Destination Map

When the Virtual Memory hardware accesses the Page Directory and Page Tables for a TLB miss, it examines and updates the flags in the low order bits of the entries, as follows:

Accessed bit. Any access (read or write) sets this bit in both the Page Directory and Page Table entries.

Dirty bit. Write accesses set the Dirty bit in the Page Table entry. The Dirty bit is undefined in Page Directory entries.

User/Supervisor and Read/Write bits. These bits are examined by the Coprocessor. The Coprocessor has a bit programmed by the host operating system that indicates whether it is being used by a Supervisor or User. If in User mode, the Coprocessor determines whether access is permitted depending on the state of the User/Supervisor and Read/Write bits in the Page Directory and Page Table entries. Table 5.3 indicates the meaning of these bits. If access is not permitted, the Coprocessor raises a **VM Protection Violation** interrupt to the host system and terminates the access cycle. It is up to the host operating system to then take appropriate action to recover from the Protection Violation Interrupt.

Present bit. The Coprocessor examines the Present bit of both the Page Directory and Page Table entries. If this bit is not set, it indicates that the Page Table or Page corresponding to that entry may not be resident in memory. Should this be the case, the XGA raises a **VM Page Not Present** interrupt to the host system. The host operating system should then fetch the Page Table or Page and place it in memory. The access then completes.

Remaining Page Directory or Page Table entry bits. All the other bits in the Page Directory or Page Table entry are ignored by the Coprocessor. The Coprocessor does not modify these bits in any way, and they can therefore be used by the operating system. Note however that it may be desirable to keep entries in the same format as 80386 Page Directory and Page Table entry formats, and so the Intel rules on use of the remaining bits should be followed.

System Coherency

In any Virtual Memory system where more than one device is accessing Virtual Memory contents, problems arise over coherency. It is vital that one device does not corrupt the other's tables or Pages, and that the tables and TLBs are kept coherent (in step) with the physical allocation of storage. The hardware mechanism provided by the Coprocessor is sufficient to implement coherent Virtual Memory systems. However, system designers should take care to avoid coherency problems.

In particular, it is recommended that the 80386 and the Coprocessor do not share Page Directories or Page Tables. Pages should not be marked as present unless they are locked in place in memory. This maintains coherency between TLB entries in the Coprocessor and the true current allocation of real memory. It prevents the Operating System from moving these Pages out of memory while the Coprocessor is accessing them.

VM Page Not Present Interrupts

When the Coprocessor detects that a Page Table or Page is not present, it raises a Page Not Present interrupt to the host system. The host operating system then fetches the required Page Table or Page (usually from disk) and places it in memory. The host system can determine the faulting address by reading the Current Virtual Address register. After the required Page Table and/or Page has been fetched, the operating system restarts the faulting memory access by clearing the Page Not Present interrupt bit in the XGA. This action causes the hardware to retry the access to the faulted entry.

It is probable that the host operating system will want to switch tasks on receiving a Page Not Present interrupt. In this case it should suspend the Coprocessor operation in the normal way (see page 63) before clearing the Page Not Present interrupt. The Coprocessor state should then be saved. When the task in which the Page Not Present interrupt occurred is restarted, the Coprocessor state should be restored, the required Page Table or Page should be placed in memory, the interrupt cleared and the Coprocessor operation restarted. The interrupt must be cleared before the Coprocessor operation is restarted, otherwise further interrupts can be lost.

VM Protection Violation Interrupts

If the Coprocessor is told to access Tables or Pages that it is not permitted to (as defined by the User/Supervisor and Read/Write entry bits), the Coprocessor generates a Protection Violation interrupt. This is generally indicative that something major is wrong with either the Virtual Memory system (as set up by operating system software), or that the Coprocessor has been incorrectly programmed.

In either case the most likely course of action required is for the operating system to terminate the Coprocessor operation and possibly terminate the faulting task. The Coprocessor operation can be terminated by writing to a control bit in the Coprocessor Control Register. The Coprocessor behaves in a similar manner for Protection Violation Interrupts as it does for Page Not Present Interrupts in that clearing the interrupt causes the hardware to retry the memory access. To avoid a repeated interrupt, the Coprocessor operation should be terminated before the Protection Violation Interrupt is cleared.

The XGA in Segmented Systems

In a segmented system design all memory is allocated in blocks called segments. Memory within a segment is guaranteed to be contiguous, and can therefore be addressed directly by the Coprocessor using physical addresses (that is, VM is turned off). The segment must be locked in place before any Coprocessor operation to ensure that the operating system does not reuse the memory during the operation.

When using 16 bit addressing in the 80386 (for example, under OS/2) it is not possible to define a segment of more than 64 Kbytes. Provided that the Coprocessor data in system memory is restricted to being no more than 64 Kbytes in length, then a single segment can be used and the Coprocessor can directly address the data using physical addresses.

Under OS/2 larger areas of memory can be requested, but are given in blocks of 64 Kbytes maximum that are unlikely to be contiguous in real memory. If larger areas in system memory are required it is possible for the driving software to turn on the Coprocessor VM address translation and perform its own memory management using memory allocated to it.

5.5 Virtual Memory Registers

The following registers provide virtual memory support for the Pixel Interface.

5.5.1 Page Directory Base Address Register (Coprocessor Registers, Offset: 0)

31	12	0
Pointer	-	

This register is a **WRITE ONLY** register.

It contains a 20-bit pointer to the page in physical memory containing the current Page Directory for the current task. Bits 31–12 are used, bits 11–0 are reserved and should be set to 0 when writing.

Loading this register causes the TLB to be cleared.

Note : this register can only be loaded after the XGA has been put into 'Supervisor' mode.

5.5.2 Current Virtual Address Register (Coprocessor Registers, Offset: 4)

31	12	0
Faulting Page Address	-	

This register is a **READ ONLY** register.

In the event of the VM hardware raising a Not Present or Protection Interrupt, the faulting page address can be read from this register. Only bits 31–12 are significant. Bits 11–0 should be masked out when read.

5.5.3 Virtual Memory Control Register (I/O Address: 21x6)

This register is directly mapped to I/O address space. It can be written and read.

7	6	5	4	3	2	1	0
NP	PV	-	-	-	US	-	EV

The fields in the **Virtual Memory Control Register** are shown in Figure 5.6.

NP	0 1	Page Not Present Interrupt Enable Interrupt not raised on VM Page Not Present Interrupt raised on VM Page Not Present
PV	0 1	Protection Violation Interrupt Enable Interrupt not raised on VM Protection Violation Interrupt raised on VM Protection
US	0 1	Protection Level Supervisor controls XGA User controls XGA
EV	0 1	Virtual Address Lookup Disable Lookup Enable Lookup

Figure 5.6 Virtual Memory Control Register Fields

VM Page Not Present Interrupt Enable (NP, bit 7): This bit controls the raising of an interrupt when a VM Page Not Present condition is detected. When this bit is set to '1', an interrupt is raised to the host system when the Not Present condition is detected. When this bit is set to '0', the Not Present condition does not cause an interrupt to be raised. In both cases the contents of the appropriate VM Interrupt Status Register status bit are updated when the Not Present condition is detected.

VM Protection Violation Interrupt Enable (PV, bit 6) This bit controls the raising of an interrupt when a VM Protection Violation condition is detected. When this bit is set to '1', an interrupt is raised to the host system when the Protection Violation condition is detected. When this bit is set to '0', the Protection Violation condition does not cause an interrupt to be raised. In both cases the contents of the appropriate VM Interrupt Status Register status bit are updated when the Protection Violation condition is detected.

U/S bit (US, Bit 2) This bit should be set to '0' if the executing task is at privilege levels 0, 1, or 2, (a Supervisor task) or set to '1' if the executing task is at privilege level 3 (a User task). If set to Supervisor (0), then no protection checking is performed by the coprocessor on Page Directory

and Page Table protection bits. If set to User (1), checking is performed, and a Protection interrupt raised if permitted access rights are violated. The user access rights are shown in table 5.3.

Enable Virtual Address Lookup (EV, Bit 0) Setting this bit turns on the virtual address translation, and subsequent addresses generated by the Pixel Interface hardware are looked up in page tables. If this bit is not set:

- Bitmaps must be resident and contiguous
- The Pixel Map Base Addresses are physical addresses
- All addresses generated by the Coprocessor are physical addresses
- Non-paged operating systems are supported

5.5.4 Virtual Memory Interrupt Status Register (I/O Address: 21x7)

This register is directly mapped to I/O address space. It can be written and read.

7	6	5	4	3	2	1	0
NP	PV	-	-	-	-	-	-

The fields in the **Virtual Memory Interrupt Status Register** are shown below.

NP	0	Page Not Present Interrupt
	1	Interrupt not caused by VM Page Not Present Interrupt caused by VM Page Not Present
PV	0	Protection Violation Interrupt
	1	Interrupt not caused by VM Protection Violation Interrupt caused by VM Protection Violation

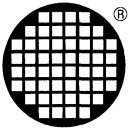
Figure 5.7 Virtual Memory Interrupt Status Register Fields

VM Page Not Present Interrupt (NP, Bit 7): When a VM Page Not Present condition occurs, this bit is automatically set to '1'. This bit can be reset to '0' by writing a '1' to it. This allows the value just read to be written back to clear the bits that were set. Writing a '0' to this bit has no effect

The act of resetting this bit (by writing a '1' to it) causes the VM hardware to retry page translation. If this bit is to be reset before the Not Present condition has been repaired, the Coprocessor operation should first be suspended or terminated, otherwise a further Not Present Interrupt is generated by the same Not Present condition.

VM Protection Violation Interrupt (PV, Bit 6): When a VM Protection Violation condition occurs, this bit is automatically set to '1'. This bit can be reset to '0' by writing a '1' to it. This allows the value just read to be written back to clear the bits that were set. Writing a '0' to this bit has no effect.

The act of resetting this bit (by writing a '1' to it) causes the VM hardware to retry page translation. If this bit is to be reset before the Protection Violation condition has been repaired, the Coprocessor operation should first be suspended or terminated, otherwise a further Protection Violation Interrupt is generated by the same Protection Violation condition. Most operating systems do not attempt to recover from a Protection Violation condition, and the guilty Coprocessor operation is terminated.



XGA programming considerations

6 Adapter Co-existence

6.1 Co-existence with VGA

As the VGA traditionally uses fixed register and mapped memory address spaces, it is characteristic of the VGA adapter that only one VGA may “exist” at any one time without a system failure. When the XGA subsystem is installed alongside either a VGA or another XGA subsystem, this condition is fulfilled by only enabling one of the VGA capable subsystems as an active VGA

An application can be written to use multiple coexisting VGA or XGA subsystems in VGA mode only by alternately disabling and re-enabling the various VGA's **Never** enable more than 1 VGA concurrently. A disabled or inactive VGA retains its visible displayed data, and the overall effect is that of a multiple VGA application.

To successively enable and disable multiple coexisting XGA subsystems in VGA mode, use the Operating Mode Register (21x0).

6.2 Co-existence with Other XGA Subsystems

Up to 8 XGA subsystems may be installed in a system.

Multiple XGA subsystems can co-exist in extended graphics mode, each occupying its own separate ranges of IO and memory space. An application written to exploit multiple XGA subsystems in this mode can access each instance of the subsystem without enabling and disabling the subsystem(s) between accesses.

To comply with the restriction on VGA co-existence, such a multiple display subsystem application should record, on initialisation, which XGA subsystem (if any) was originally in VGA mode. On termination only **that** subsystem should be returned to VGA mode.

7 Locating the XGA Subsystem

Before using the subsystem, it is first necessary to locate the subsystem in I/O and memory space. This is done by interrogating and interpreting the POS data for the subsystem.

7.1 Reading POS Data

To do this, selectively enable for setup each adapter in the system, including the system board video subsystem, and examine the POS ID. Do this using the *System Services BIOS call INT 15h, AH = C4h Programmable Option Select* as documented in the IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference. The following calls are relevant, and are used as follows:

AL = 00h Return Base POS card register address Use this call once only to determine the base I/O register address for reading POS data from all adapters in the system

AL = 01h Enable Slot for Setup Used to selectively enable each adapter to enable the POS data to be read for that adapter.

If this call is used for adapter 0 (the system board), the POS data returned using the I/O ports is that for the system board itself, rather than that for the system board Video subsystem. To access the system board Video subsystem POS data **do not** use this call, but instead write 0DFh to Port 94h using the OUT instruction.

The POS data bytes for the selected adapter may then be read by reading (IN) from the 6 consecutive I/O port addresses starting with the Base register address returned by the "Return Base POS card register address" subfunction.

Base, Base + 1 POS ID

Base + 2 - 5 POS data bytes 0 - 3

Interrupts should be disabled while each card is *Enabled for Setup*

AL = 02h Card Enable Used to restore each adapter to its normal *Enabled* state **immediately** after POS data has been read for that adapter.

To re-enable the system board video adapter, **do not** use this call, but instead write 0FFh to Port 94h using the OUT instruction.

Interrupts should remain disabled until each adapter has been "Enabled" in this way.

If the POS ID matches one of the list of POS IDs allocated to the XGA, and future register compatible subsystems, then the search is complete. The POS IDs allocated to the XGA subsystems are as follows:

8FD8h
8FD9h
8FDAh
8FDBh

On successfully matching POS ID's read the remainder of the POS data bytes for that subsystem as described above. This data can then be used to calculate the location of the XGA subsystem's registers and display buffers in IO and physical system memory address space. Detailed descriptions of the POS Data bit assignments are available in Section 5.2. As also discussed in Section 12, it is important for reasons of future compatibility to mask out all reserved and unused POS data bits before using the data for these calculations.

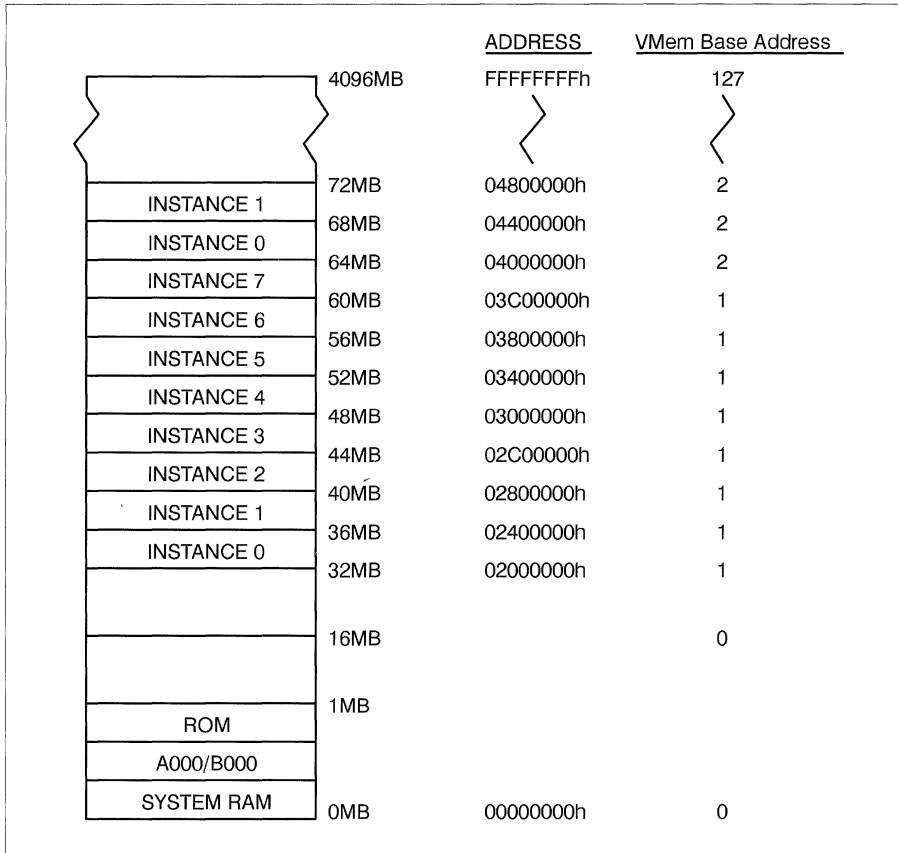


Figure 7.2 The XGA Video Memory Base Address Diagram

The video memory base address field defines a 32 Mbyte address range and the 'Instance' defines a 4 Mbyte address range within the 32 Mbyte range.

For example:

Assuming the Instance = 6
 and the Video Memory Base Address field = 1
 then the Video Memory Base Address is 03800000h.

The Video Memory Base Address, once calculated serves 2 totally separate purposes

4 Mbyte System Video Memory Aperture

If enabled (Read from bit 0 in POS Register 4 to ascertain if the Aperture is enabled), the 4 Mbyte System Video Memory Aperture is located at this address in physical system address space. Provided virtual addressability to this range of physical address space can be achieved, the entire video memory may be accessed through this Aperture at this address.

Video Memory Location in Coprocessor Address Space

The Video Memory Base Address has a special significance to the XGA coprocessor. It defines the location of the video memory, including the display pixel map, in the XGA coprocessor's view of system address

space. The significance of this is that the XGA coprocessor recognizes addresses in this range to be addresses in local video memory rather than general system memory. This is how the XGA coprocessor differentiates video memory from system memory. If an address passed to the XGA coprocessor is in this range, the XGA coprocessor knows that it is operating on a bitmap in video memory. If the address is outside this range, the coprocessor assumes it is operating on a bitmap in normal system memory, and attempts to use DMA busmastership to access it.

As the XGA subsystem operates internally on a 32 bit bus, this address will be a 32 bit address irrespective of whether the XGA subsystem is installed in a 16 or 32 bit slot or system, or whether the 4 Mbyte system video memory aperture is enabled or disabled. This will therefore be a 32 bit address even on systems where such addresses are not otherwise possible.

7.2.5 1 Mbyte Aperture Base Address

The 1 Mbyte Aperture base address is calculated from the 1 Mbyte Base address field in POS Register 5 bits 0 to 3.

If (1Mb Base Field \neq 0)

1Mb Aperture Base Address = 1Mb Base Field * 100000h

If (1Mb Base Field = 0) 1Mb Aperture is disabled

7.3 Display Type and Video Memory Size

The attached display type is determined by reading the Display ID register – Index 52h. The list of display IDs, including the returned value indicating no attached display, is listed in Section 10. It may also be necessary (depending on the application) to determine the video memory size to ensure that the required mode is available. The method for doing this is also described in Section 10

If looking for multiple coexistent XGA subsystems, (for instance for a multiple display adapter application) continue until sufficient instances of the XGA have been located.

8 VGA Primary Adapter Considerations

In situations where a single XGA subsystem is providing both VGA and Extended Graphics functions, particularly on a system with single display subsystem/monitor, an application using the subsystem in Extended Graphics Mode takes on a number of additional systems responsibilities.

Having found an instance of the XGA subsystem, and before switching the subsystem into Extended Graphics Mode, examine the Operating mode register (Address 21x0h) bits 0 and 2 to determine if the XGA subsystem is in VGA mode and enabled as such.

If the XGA subsystem is not enabled in VGA mode, the subsystem is operating an 'auxiliary video subsystem' and systems messages, etc., may safely be left to the primary VGA source. In this case the XGA subsystem must **not** be put into VGA mode unless the current VGA is disabled. If the XGA subsystem is enabled in VGA mode then the subsystem is the system primary video subsystem, and a number of special considerations apply.

8.1 Chaining the Int 10h Video BIOS Handler

The application should chain the Int 10h Video interrupt handler, and monitor calls to the Int 10h handler while the application is using the XGA subsystem in Extended Graphics Mode.

There are a number of hot key and error handlers that may attempt to communicate with the VGA while the XGA subsystem is in Extended Graphics Mode, and code must be written to handle such calls.

The majority of calls to the Int 10h handler can be ignored (simply return to the caller) while the XGA subsystem is in Extended Graphics Mode, but a small number of calls require correct handling.

(Ah) = 00h Set Mode Any attempt to reset the subsystem into VGA mode while in Extended Graphics Mode should be complied with.

As the application should be in control of the subsystem, any attempt to set the subsystem into a VGA mode can only be as a result of a catastrophic error situation, and failure to restore VGA mode will inevitably result in the loss of critical data, and the result will be a blank, or at best unmeaningful, display.

The Int 10h handler should immediately restore the subsystem to VGA mode, and chain on to the saved Int 10h interrupt vector to allow the VGA set mode to be processed.

The Non Maskable Interrupt (NMI) handler traditionally issues a Get Mode followed by a Set Mode to the mode returned by the Get Mode. If the application's Int 10h Video interrupt handler detects an attempt to Set Mode 7Fh, then mode 03h should be substituted, and the Set Mode allowed to proceed, after putting the subsystem in VGA mode as described above.

(Ah) = 0Fh Return current video state While in Extended Graphics Mode, the application's Int 10h interrupt handler should return a current mode of 7Fh in AL, to indicate that the subsystem is in a non-VGA mode.

This is a special mode number assigned for this purpose.

8.2 Int 24h, Critical Error Handler

The application should trap and re-vector the DOS Critical Error Handler Interrupt Vector (Int 24h), as described in the DOS Technical Reference Manual. The application will then be notified on DOS Critical errors.

The application's critical error handler should save the subsystem's video state (as far as necessary), and put the XGA subsystem into VGA mode before chaining on using the saved vector to the original Critical Error Handler. This will allow the Critical Error Handler's dialogue with the user to proceed normally.

On returning from the chained Critical Error handler, the application's Critical Error handler should examine the return code in AL to determine the appropriate action.

0, 1, 3 Control will be returned to the application. Put the XGA subsystem back into extended graphics mode, and restore the video state as necessary.

2 The program will be aborted by the system. Leave the XGA subsystem in VGA mode and return.

Alternatively the application can take over the entire Critical Error handling dialogue while remaining in Extended Graphics Mode.

Note: The C language *signal* function may (in some implementations) be used to intercept the Critical Error handler for this purpose.

8.3 Int 23h Ctrl-Break Exit Address

The application should trap and re-vector the DOS Ctrl-Break Exit Address, as described in the DOS Technical Reference Manual. The application will then be notified on when the Ctrl-Break key combination is entered.

If the application is not otherwise intercepting Ctrl-Breaks, the XGA subsystem should be put back into VGA mode before chaining on using the saved vector to the original Ctrl-Break handler. This will allow the normal Ctrl-Break handler to proceed.

Alternatively the application can take over the entire Ctrl-Break handling while remaining in Extended Graphics Mode.

Note: The C language *signal* function may be used to intercept the Ctrl-Break handler for this purpose.

8.4 Int 21h Function 4Ch Program Terminate function

The subsystem must be left in VGA mode on program termination, irrespective of the how the program terminates, or is terminated.

To ensure that this is done, the application should trap and re-vector the normal DOS program terminate function *DOS Int 21h function 4Ch*, as described in the DOS Technical Reference Manual. On receiving notice of program termination, the application should put the subsystem back into VGA mode, and unhook all other hooked interrupt vectors before chaining on for the remainder of program termination handling.

DOS Int 21h function 4Ch is the conventional method used by all programs to terminate. By trapping the DOS function interrupt (Int 21h) and monitoring calls to the Program terminate function (4Ch), all routes by which a program may terminate normally should be covered.

Note: There are other Program Terminate functions, including:

- Int 20h
- Int 27h
- Int 21h Function 00h
- Int 21h Function 31h

For complete cover, these calls may be similarly re- vectored and trapped, but they are not as commonly used as the Int 21h Function 4Ch.

All other function calls should be passed on to the previous DOS function handler using the saved interrupt vector.

On detecting a call to function 4Ch, put the XGA subsystem into VGA mode before chaining on using the saved vector to the original DOS function Handler. This will allow the DOS Program Terminate function to proceed normally.

Note: The C language *atexit* function may be used for this purpose.

9 General Systems Considerations

9.1 Co-existing with LIM Expanded Memory Managers

The XGA subsystem uses memory mapped registers located in the C0000/D0000 region of physical address space, as described in Section 5.2. Unfortunately this area is now heavily used by Expanded Memory Managers to provide Expanded Memory Services to applications

Once the location of the XGA subsystem's memory mapped register space in the C0000/D0000 region has been determined, the application should interrogate any Expanded Memory Manager to ensure that there is no contention for this range of physical address space.

This should be done as described in the Lotus/Intel/Microsoft Expanded Memory Specification Version 4.0, under Function 25 (Ah) = 58h Get Physical Address Array. If the application detects a clash between the XGA subsystem's use of physical address space and that claimed by the Expanded Memory Manager, a warning should be issued advising the user to resolve this contention by use of the Expanded Memory Manager call parameters, usually on the 'DEVICE =' statement in CONFIG.SYS.

9.2 Screen Switch Notification, Int 2Fh

For the application to work successfully in MVDM (Multiple Virtual DOS Machine) environments, or in the DOS compatibility box of OS/2, it should trap and re-vector the DOS Multiplex vector, looking for (Ah) = 40h. Any other values should be immediately passed to the chained Int 2fh handler.

This multiplex interrupt is used with (Ah) = 40h to notify DOS applications of Screen Switches.

(AI) = 01h DOS Mode application being switched to the background

The application should save its video state and put the display back into VGA mode (if applicable).

(AI) = 02h DOS Mode application being switched to the foreground.

The application can switch the subsystem back into Extended Graphics Mode, and restore the saved video state.

The range of operations permitted within Int 2Fh processing is limited. For instance it is not permissible to issue Disk I/O operations, which therefore precludes an entire save and restore of video memory and state. The only way of using this call is for the Int 2Fh interrupt handler to notify or semaphore the application that a re-draw is required (if application program structure permits).

10 Extended Graphics Modes Selection

10.1 Modes Available

The following table shows the list of modes available according to display type and amount of video memory configured on the XGA subsystem.

Display ID	Example Displays	Size	Color	Maximum addressability	512K memory	1Meg memory
1111b	None				None	None
1101b	8503	12in.	mono	640x480	640x480x64 grays	640x480x64 grays
1110b	8513 8512	12in. 14in.	color	640x480	640x480x256 colors	640x480x256 colors 640x480x65536 colors
1011b	8515	14in.	color	1024x768	640x480x256 colors 1024x768x16 colors	640x480x256 colors 640x480x65536 colors 1024x768x16 colors 1024x768x256 colors
1001b	8604 8507	15in. 19in.	mono	1024x768	640x480x64 grays 1024x768x16 grays	640x480x64 grays 1024x768x16 grays 1024x768x64 grays
1010b	8514	16in.	color	1024x768	640x480x256 colors 1024x768x16 colors	640x480x256 colors 640x480x65536 colors 1024x768x16 colors 1024x768x256 colors

Table 10.1 Availability of Extended Graphics Modes

To ascertain the display type read the XGA subsystem register index 52h, 'Display ID' and examine the Display ID bits returned.

To ascertain the amount of video memory installed, there are two options. Both options rely on a 'write read-back check', whereby a particular value is written to a key location, which is subsequently read to ascertain whether the written value has persisted.

- 1 Use the system processor to write a value through an aperture to the word at offset 768K into video memory. This technique assumes that the system video memory real mode aperture is available. Sample code to do this is shown in figure 10.1.
- 2 Use the XGA subsystem's PxBit capability to perform a similar test to the previous example; PxBit a constant color to the location in video memory, then PxBit that value back from video memory to system memory using DMA busmastership.

This technique works irrespective of the availability of a system video memory aperture, but it does require physical addressability to a location in system memory for the DMA busmastership operation.

```

;
;* Assume GS points to start of A0000 Real mode aperture
;* Where registers are shown as (for instance 21x0h), this should
;* be filled in with the appropriate IO port address after
;* determining the location of the XGA subsystem in IO space
;*
;* First put the adapter PARTIALLY in extended graphics mode
;* to allow use of the system video memory Aperture
    mov     al,0
    mov     dx,21x4h      ; disable XGA interrupts
    out     dx,al
;
    mov     ax,0064h
    mov     dx,21xAh      ; Blank palette
    out     dx,ax        ; indexed XGA register 64h
;
    mov     ax,04h
    mov     dx,21x0h      ; Set adapter in Extended Graphics Mode
    out     dx,al
;
    mov     al,01h
    mov     dx,21x1h      ; Locate video memory Aperture at A0000
    out     dx,al
;
    mov     dx,21x8h      ; System video memory indx reg.
    mov     al,0ch        ; Offset 768K
    out     dx,al
;
    mov     byte ptr gs:[0],0A5h ; Set byte to A5h
    mov     byte ptr gs:[1],0h   ; Avoid shadows on data lines
;
    cmp     byte ptr gs:[0],0A5h ; Test against value written
    jne     vram_512k           ; 512K video memory only
;
    mov     byte ptr gs:[0],5Ah   ; Set byte to 5Ah
    mov     byte ptr gs:[1],0h   ; Avoid shadows on data lines
;
    cmp     byte ptr gs:[0],0A5h ; Test against value written
    je     vram_1Meg           ; 1 Meg if still matches
    jmp     vram_512k          ; Otherwise 1/2 meg found

```

Figure 10.1 Video Memory Size Determination

Having ascertained the monitor type and the video memory configuration available, the modes available can be read from the table above.

11 Mode Setting the XGA Subsystem

The following points should be observed by **all** software when switching between modes.

- All data in the Video Memory is preserved during a mode switch, provided that the CRTIC is halted at the time, using the Display Control 1 register (if switching out of Extended Graphics Mode), or the Reset register (if switching out of VGA mode). The CRTIC is discussed in Section 4.1.5.
- When switching between VGA modes, the mapping of the VGA memory maps to the Video Memory is controlled by two bits in VGA registers:
- Word/Byte Mode (CRTIC Mode Control Register, bit 6)
- Double-Word Mode (CRTIC Underline Location Register, bit 6)

VGA modes can thus be split into three groups: byte modes, word modes and double-word modes.

All switches between modes in the same group are indistinguishable from the same mode switches on VGA.

Switches between modes in different groups produce different effects from those observed on the VGA, but since the bits controlling the mapping are used for display purposes, the picture is scrambled in both cases

Partial mode switches (for example, to load fonts in a text mode) are also possible. As the bits used to control the mapping of the data in the Video Memory are those used to control the displaying of the picture, all partial mode switches to update the Video Memory which don't destroy the picture (and many that do) work correctly.

11.1 Individual Mode Setting Procedures

This section gives the register settings necessary to set the subsystem into the various modes available, subject to the rules described elsewhere in this chapter. It is important to follow the order of register setting as shown here.

11.1.1 Extended Graphics Mode

To set the XGA subsystem into extended graphics mode (subject to the configuration being capable of supporting the required mode as listed in Section 10), write the register values in the sequence shown to the XGA subsystem's registers.

XGA register name	XGA reg. id	Oper	1024x 768x 256 color mode values	1024x 768x 16 color mode values	640x 480x 256 color mode values	640x 480x 65536 color mode values	Comments
Interrupt Enable	21x4	=	00h	00h	00h	00h	Initial Value
Interrupt Status	21x5	=	FFh	FFh	FFh	FFh	
Operating Mode	21x0	=	04h	04h	04h	04h	Set Extended Graphics Mode
Palette Mask	64	=	00h	00h	00h	00h	Blank Display
Video memory Aperture Ctl	21x1	=	00h	00h	00h	00h	Initial Value
Video memory Aperture Index	21x8	=	00h	00h	00h	00h	Initial Value
Virt Mem Ctl	21x6	=	00h	00h	00h	00h	Initial Value
Memory Access Mode	21x9	=	03h	02h	03h	04h	Initial Value
Disp Mode 1	50	=	01h	01h	01h	01h	Prepare for reset
Disp Mode 1	50	=	00h	00h	00h	00h	Reset CRTIC
Horiz Total Lo.	10	=	9Dh	9Dh	63h	63h)
Horiz Total Hi.	11	=	00h	00h	00h	00h)

XGA register name	XGA reg. id	Oper	1024x 768x 256 color mode values	1024x 768x 16 color mode values	640x 480x 256 color mode values	640x 480x 65536 color mode values	Comments
Horiz Display End Lo	12	=	7Fh	7Fh	4Fh	4Fh)
Horiz Display End Hi	13	=	00h	00h	00h	00h)
Horiz Blank Start Lo	14	=	7Fh	7Fh	4Fh	4Fh)
Horiz Blank Start Hi	15	=	00h	00h	00h	00h)
Horiz Blank End Lo	16	=	9Dh	9Dh	63h	63h)
Horiz Blank End Hi	17	=	00h	00h	00h	00h)
Horiz Sync Start Lo	18	=	87h	87h	55h	55h)
Horiz Sync Start Hi	19	=	00h	00h	00h	00h)
Horiz Sync End Lo	1A	=	9Ch	9Ch	61h	61h)
Horiz Sync End Hi	1B	=	00h	00h	00h	00h)
Horiz Sync Posn	1C	=	40h	40h	00h	00h)
Horiz Sync Posn	1E	=	04h	04h	00h	00h)
Vert Total Lo	20	=	30h	30h	0Ch	0Ch)
Vert Total Hi	21	=	03h	03h	02h	02h) XGA CRTC
Vert Disp End Lo	22	=	FFh	FFh	DFh	DFh) parameters
Vert Disp End Hi	23	=	02h	02h	01h	01h)
Vert Blank Start Lo	24	=	FFh	FFh	DFh	DFh)
Vert Blank Start Hi	25	=	02h	02h	01h	01h)
Vert Blank End Lo	26	=	30h	30h	0Ch	0Ch)
Vert Blank End Hi	27	=	03h	03h	02h	02h)
Vert Sync Start Lo	28	=	00h	00h	EAh	EAh)
Vert Sync Start Hi	29	=	03h	03h	01h	01h)
Vert Sync End	2A	=	08h	08h	ECh	ECh)
Vert Line Comp Lo	2C	=	FFh	FFh	FFh	FFh)
Vert Line Comp Hi	2D	=	FFh	FFh	FFh	FFh)
Sprite Control	36	=	00h	00h	00h	00h	Initial Value
Start Addr Lo	40	=	00h	00h	00h	00h	Initial Value
Start Addr Me	41	=	00h	00h	00h	00h	Initial Value
Start Addr Hi	42	=	00h	00h	00h	00h	Initial Value
Buffer Pitch Lo	43	=	80h	40h	50h	A0h	
Buffer Pitch Hi	44	=	00h	00h	00h	00h	
Clock Sel	54	=	0dh	0dh	00h	00h	
Display Mode 2	51	=	03h	02h	03h	04h	
Ext Clock Sel	70	=	00h	00h	00h	00h	
Display Mode 1	50	=	0Fh	0Fh	C7h	C7h	
Note: Initial Palette loading should be done at this point, by writing to the appropriate XGA subsystem palette/sprite registers.							
The video memory should also be initialised at this point, to avoid random data appearing when the palette mask is set to make the current display pixel map contents visible.							
Border Color	55	=	00h	00h	00h	00h	Initial Value
Palette Mask	64	=	FFh	FFh	FFh	FFh	Make visible

11.1.2 VGA Mode

To put the XGA subsystem into VGA mode (subject to the rules for so doing as discussed in Section 8), perform in sequence the operations described here:

- 1 Clear first 256K of video memory contents to avoid screen flash caused by random data being present on switching into VGA mode.
- 2 Write data to the registers in the sequence as shown:

Value	Oper	XGA reg	VGA reg	Comments
00h	=	21x1		Aperture Control
00h	=	21x4		Interrupt disable
FFh	=	21x5		Clear Interrupts
FFh	=	64		Palette Mask
15h	=	50		Enable VFB, Prepare for reset
14h	=	50		Enable VFB, Reset CRTC
00h	=	51		Normal scale factors
04h	=	54		Select VGA Oscillator
00h	=	70		Ext Oscillator (VGA)
20h	=	2A		Ensure No VSync interrupts
01h	=	21x0		Switch to VGA mode
01h	=		3C3	Enable VGA address decode

- 3 Set no. lines in VGA mode (if required) using Video BIOS Int 10h Ah = 12h
- 4 Set required VGA mode using Video BIOS Int 10h Ah = 00h – Set mode.

Figure 11 1 Setting VGA Mode

The XGA subsystem will now be in VGA mode.

11.1.3 132 Column Text Mode

The 132 column text mode should eventually appear as Video BIOS Mode 14h on XGA subsystems and systems units. Before directly setting the mode as described below, issue a Video BIOS Int 10h Return Functionality State Information call, and examine the list of BIOS supported modes for the existence of mode 14h.

If Mode 14h is supported in BIOS, the appropriate Video BIOS Set Mode should be issued in preference to the method described here.

Where Video BIOS mode 14h is not supported in BIOS, the following sequence of operations will put the subsystem into 132 column text mode:

- 1 If necessary put the XGA subsystem into VGA mode as described in Section 11.1.2.
- 2 Write data to registers in the sequence shown:

Value	Oper	XGA reg	VGA 3D4/5	VGA 3C4/5	Other VGA	Comments
15h	=	50				Prepare CRTC for reset
14h	=	50				Reset CRTC
04h	=	54				Select VGA Oscillator

- 3 Set No. lines in VGA mode using Int 10h Ah = 12h (200, 350 or 400)
- 4 Set VGA mode 3 using Int 10h Ax = 0003h. The 132 column text mode is a variation on the VGA text mode, and the table below is the variations from the standard mode.
- 5 Write data to registers in the sequence as shown:

Value	Oper	XGA reg	VGA 3D4/5	VGA 3C4/5	Other VGA	Comments
01h	=	50)
FDh	&=	50) Prepare CRTC for reset
FCh	&=	50				Reset CRTC
03h	=	21x0				132 column text mode
01h	=	54				132 column clock frequency select
80h	=	70				Select internal 132 col clock
EFh	&=	50				Disable VFB
7Fh	&=		11			Enable VGA CRTC reg update
A4h	=		0)
83h	=		1)
84h	=		2)
83h	=		3)
90h	=		4) Variations on VGA CRTC syncs
80h	=		5)
A3h	=	1A)
00h	=	1B)
42h	=		13)
80h	=		11			Disable VGA CRTC reg update
03h	=	50				Remove CRTC Reset
01h	=			01		8 bit characters
**	INP				3DA	Read sets Attr Ctlr flip flop
13h	=				3C0) Sets Attr Ctlr
00h	=				3C0) Reg 13h to 00h
20h	=				3C0	Restore Palette

- 6 MOVE 84h to 40:4Ah in BIOS data area to force Video BIOS recognition of 132 column text mode

Figure 11.2 Setting 132 column text mode

Having set the mode, it is programmed similarly to any other VGA text mode, with a coded text buffer located at B8000 in system address space. Obviously the coded text buffer is now 132 columns wide.

If it is necessary to invoke a mode change using Video BIOS (Int 10h) while in 132 column text mode (for instance to vary the number of lines), the steps shown above from 2 to 6 should be followed.

11.2 System Video Memory Apertures

There are three possible apertures in the system's physical address space. If present, any of them may be used by the system processor to directly access the packed pixel display buffer mapped into system memory. Each aperture has its own rules for existence, advantages and drawbacks as described below. The XGA coprocessor may make the use of an aperture unnecessary.

The precise location of each aperture, including whether it is enabled, may be determined by decoding the XGA subsystem's POS data as described in Section 7.

11.2.1 64K System Video Memory Aperture

This aperture is at either A0000 or B0000 in physical address space. The 64K aperture is insufficient to access the entire subsystem display buffer at a time, so the aperture position over the display buffer is controlled using the Aperture Index register (21x8).

This is the only aperture in 8086 real mode address space.

Other video adapters, such as another adapter or subsystem in either VGA or Extended Graphics mode may contend for the use of this aperture. Only one video subsystem may have this aperture enabled at any one time. Provided there is no contention for the A0000 or B0000 address spaces, this aperture is the only aperture that may be 'enabled' at will by the application.

11.2.2 1 Mbyte System Video Memory Aperture

This aperture may appear at a whole number of megabytes below 16 Mbytes, depending on the hardware configuration. Its position, and whether it is enabled, must be determined by decoding the POS data as described in Section 7.

In the case of multiple coexisting XGA subsystems, each may have its own such aperture. Dependent on hardware configuration it is possible for some but not all coexisting XGA subsystems to have their 1 Mbyte System Video Memory Apertures enabled.

This aperture is sufficiently large that the entire video memory is accessible without using the Aperture Index register (21x8) to move the aperture. The Aperture Index register must be set to zero when using this aperture.

This aperture is only easily accessible in protect mode environments. The operating system must provide addressability to the address range occupied by the aperture. Some operating systems attempt to restrict such addressability to protect or kernel device drivers only. It may be necessary to write a small kernel device driver to provide addressability. For instance, in a 16 bit segmented system such as OS/2, the following steps may be necessary to build GDT addressability to an aperture.

- 1 Allocate a GDT selector
- 2 Modify the GDT entry directly to alter the permission bits to allow user mode (Ring 3) access.
- 3 Alter the GDT segment length to be a 1 Megabyte segment. The entire 1 Mbyte video memory display buffers can be then accessed as a single segment.

Always check that the aperture is enabled before assuming its existence. If this aperture is found to be disabled, it cannot be enabled by the application. The application should then try to use the 4 Mbyte aperture.

11.2.3 4 Mbyte System Video Memory Aperture

This aperture appears at a multiple of 4 Mbytes at or above 16 Mbytes, depending on the hardware configuration. Its position, and whether it is enabled, must be determined by decoding the POS data as described in Section 7.

In the case of multiple coexisting XGA subsystems, each will have its own such aperture.

This aperture is not available in 16 bit systems based on the 80386SX. Neither does this aperture exist when the XGA subsystem adapter card is plugged into a 16 bit (short) slot on a 32 bit system. Always check that the aperture is enabled before assuming its existence. Also, check the Auto-Configuration register as described in 'Auto-Configuration Register (Index: 04)' on page 24 to determine the Bus width

While this aperture is always present when the XGA subsystem is plugged into a 32 bit slot on a 32 bit system, it may not be easily accessible in either real mode DOS or 16 bit protect mode operating systems.

11.3 Physical Addressability to System Memory

The XGA subsystem coprocessor is able to operate as a DMA busmaster. Using this, the coprocessor is capable of bitmap operations on bitmaps up to 4K by 4K pixels anywhere in system address space, including video memory. A PxBlt operation can be defined as a function of 4 separate bitmaps, $D' = f(S, D, P, M)$. That is, the modified destination pel (D') is a function of the source (S), the current destination pel (D), the pattern (P) and the mask (M). Any or all of these bitmaps can be anywhere in memory. The XGA coprocessor handles all bitmaps alike, no special handling of a bitmap in video memory is required.

This flexibility is very powerful, but requires support from the operating system to fully realize the benefits.

DMA busmastership is of necessity on i386 physical address space while applications run on the system processor in virtual or linear address space. The system processor automatically converts such addresses to physical addresses internally via the page tables or segment descriptor tables. An adapter such as the XGA coprocessor has no physical access to either the segment descriptors or the page tables. To use DMA busmastership, the application (or its device drivers) must provide the XGA coprocessor with the physical address of all the bitmaps on which it requires the XGA coprocessor to operate. Methods for providing the XGA coprocessor with physical addressability to all such resources, and the tasks necessary, vary according to the operating system, and mode of the system processor.

11.3.1 Real Mode DOS Environments

The real mode DOS environment is the simplest and easiest in terms of memory management. The application is limited to 640K of real mode DOS memory. Conversion from virtual to physical memory addresses is by means of a simple *Shift left 4 and add*. One problem may be that the application written to run in the real mode DOS environment will be expected to migrate *compatibly* to Multiple Virtual DOS Machine (MVDM) environments. The simple *Shift left 4 and add* has now merely produced a linear, but not a physical address. Hopefully the MVDM hypervisor 'Virtualisation Display Driver' will cope with this, but applications must be tested in individual MVDM environments before full real mode DOS compatibility can be claimed.

Extended Memory

A DOS application can allocate large areas of *Extended* memory as working bitmaps for the application. It is unnecessary to have system processor addressability to such bitmaps, as the XGA coprocessor can do all the necessary accesses, and Extended memory is ideal for this purpose.

The techniques required to allocate and use *Extended* memory in a DOS application are not covered here.

LIM EMS Managers

The commonest memory management technique that gives extra memory in the DOS environment is the Lotus-Intel-Microsoft Expanded Memory Services Manager. Such memory managers implement the LIM 4.0 specification for a software interrupt driven memory management interface via software interrupt 67h. On 80386 and above processors, all the memory is physically allocated as *Extended* memory, and the LIM EMS manager maps this into Expanded memory via the 80386 page tables.

The drawback to this technique is that a simple *Shift left 4 and add* will only yield the linear, but not the physical address of the LIM frame. To determine the physical address, it is necessary to call the *Operating System DMA Services* interface of the LIM EMS driver to convert linear addresses to physical. This inter-

face, based on Software Interrupt 4Bh, is described in the IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference manual.

This interface is of recent origin, and early LIM drivers may be encountered that have not yet implemented it. The application has 2 choices:

- 1 Do **not** locate resources in LIM memory on which the XGA coprocessor is requested to operate
- 2 Specify a dependency in the application documentation on LIM EMS drivers that have implemented this interface.

11.3.2 32 bit DOS Extended Environments

This is the mode of the processor in which full exploitation of the power of the XGA coprocessor is easiest. The application can allocate huge memory bitmaps without needing to account for the behaviour of a memory manager that might change the location of the memory. Calculation of physical addresses is easily accomplished without the system overheads of full blown protect mode operating systems. Access to the XGA subsystem's system video memory aperture and coprocessor register address space can be accomplished easily.

11.3.3 Multiple Virtual DOS Machine Environments

This is a mode where multiple DOS applications can run concurrently (even windowed on the same screen), each application appears internally to be running in the bottom Megabyte of physical address space.

Full compatibility with real mode DOS for a DMA busmaster such as the XGA coprocessor is only provided if each such DOS application using the XGA subsystem in extended graphics mode is locked in the bottom 1 Mbyte of physical address space.

For the extended graphics mode DOS application to run successfully (even if not windowed), the MVDM hypervisor's *Virtualisation Display Driver* must include specific support functions.

A suggested technique is described here, although there may be others equally effective.

On switching to the foreground ("Resurrection") a VDM in which an XGA extended graphics mode DOS application is running, the entire 640K of the VDM's linear address space is locked "discontiguous" by the VDD. The VDD will then use the foreground VDM's Page Directory Entry to provide physical addressability to the VDM's discontiguous linear address space. The XGA coprocessor's virtual address capability can then be used, by giving the XGA coprocessor direct DMA access to the VDM's Page Tables. As the entire 640K DOS region is locked, (except for LIM which will be discussed below), a DOS application will not supply addresses outside the locked 640K linear address range.

The technique relies on the XGA coprocessor Page Directory Base Address, once set by the VDD on resurrection, remaining unmodified by the application. Inadvertant updates to this field can be prevented by placing the XGA coprocessor into User Mode.

It is possible for an application to program the XGA coprocessor to access memory outside of the applications own storage. If this is done, the integrity of the entire system is compromised.

One complication is LIM, where the DOS application may locate a resource such as a font definition in LIM memory, and subsequently give the XGA coprocessor the linear address of the LIM frame, rather than the underlying address. This is normally handled in real mode DOS by calling the *Operating System DMA Services* interface of the LIM EMS driver to convert linear addresses to physical. This will not be appropriate in the MVDM environment, as the Linear address is now required by virtue of the fact that XGA is in VM mode, operating off the System Page Tables for the VDM in question. The obvious solution is for the VDD to monitor the LIM software interrupt (Int 67h), and ensure that any LIM 'logical 16K pages' currently mapped into the VDM's LIM frames or Windows are locked. The VDM's page tables will then naturally reflect the correct physical addresses for the LIM pages at the linear address of the LIM frame. Calls to the *Operating System DMA Services* interface must also be filtered out.

11.3.4 Protect Mode 16 Bit Segmented Environments

An application written for this environments has a range of limitations imposed by the operating system.

64K Segment Limit

No memory object in this environment can be larger than 64K, unless allocated by a kernel device driver on initialisation.

The application cannot assume that 2 adjacent segments are located adjacent in physical address space

Segment Motion

Segments are liable to be moved in physical system memory at any time. Segments may even be 'swapped' out to disk when memory is overcommitted.

All segments must be 'locked' before the physical address is established.

Consideration must be given to the overall impact on system performance of long term locking of large areas of memory. It also increases the minimum physical memory configuration that is required to run the application.

System Overheads

Applications generally run at a low privilege level, and video device drivers must be accessible easily and frequently by the application without large system overheads

Applications using the XGA coprocessor typically need to make use of the operating system's memory management services. These services (used for locking segments and determining physical address of segments) are typically restricted to device drivers at operating at high privilege levels.

The system overhead in reaching these services in such operating systems may be so high as to make the writing of high performance applications difficult.

Access to XGA Registers and System Memory Apertures

Considerable ingenuity is required to provide addressability to the XGA subsystem's I/O and memory space. A technique for this is described in Section 11.2.

Suggested Design Model

A suggested design for an application in this environment is as follows:

Use a kernel or Ring 0 '.SYS' device driver to permanently allocate a range of physical memory (typically 128K). The device driver can then generate a GDT selector to this *Kernel Work Space (KWS)* that is valid in User mode at Ring 3. Both the virtual and physical addresses are passed back to the application in User mode. The Kernel device driver also provides User mode addressability to the XGA coprocessor's register address space.

The application can then operate totally in User mode, passing resources (for example, bitmaps, patterns, etc.) by system processor block moves into the KWS. The application can then drive the XGA coprocessor to access the resources in the KWS without ever suffering the system overheads of switching into kernel mode again. Bitmaps are effectively 'cached' via the KWS to the XGA coprocessor.

The principal feature of this technique is to minimize kernel or system overheads.

11.3.5 Paged Virtual Memory (VM) Environments

This environment shares many constraints with the 16 bit segmented environment. The principle difference is that the unit of granularity of memory objects has dropped from 64K to 4K, the VM support in the XGA coprocessor is intended to support this environment.

4K Discontiguous Pages

In this environment, memory is allocated to applications in 4K pages. The system memory manager looks after all paging, and may swap pages in and out of physical memory transparently to the application. The application can make no assumptions about the relationship between adjacent pages.

There are memory management calls available to the kernel or Ring 0 device driver that will allow such a device driver to build a table containing the physical addresses of all the component pages of a large bitmap. As with 16 bit segmented environments, described in Section 11.3.4, the overhead of the transition to kernel mode to make such calls expensive. It is, however, possible to build such a table, and to operate the XGA coprocessor in *virtual memory* mode. The overall impact on system performance and minimum physical memory configurations should be considered, particularly as a bitmap in this case could theoretically be 4Kx4Kx8bpp, a total of 16 Mbytes of locked physical memory.

It is possible to use the XGA coprocessor to interrupt to indicate a page fault, but this interrupt is a normal shared adapter interrupt rather than a i386 page fault interrupt. As such it is handled at a lower priority. A further complication is that most such operating systems do not allow device drivers to call the Memory Management services (to request the faulting pages) on an interrupt thread.

Page Table Coherency

It would seem obvious that the XGA coprocessor should be able to operate off the system page tables, as the XGA coprocessor uses i386 like page tables.

Unfortunately a typical VM operating system uses a set of page tables per task. In a multi tasking environment, only the currently executing task's page tables remain coherent, while background task's page tables are allowed to become outdated or incoherent.

This implies that the XGA coprocessor can be operating on a set of page tables belonging to what may be a background task, and so it cannot assume that the page table remains coherent unless the component pages have been locked by a call to the system memory management interface by a kernel device driver.

System Overheads

The overheads associated in switching from the applications privilege level to the kernel level have been described earlier, see *System Overheads* in Section 11.3.4.

Access to XGA Registers and System Memory Apertures

Again it is necessary to provide addressability to these XGA subsystem's I/O spaces. The Operating system memory management services must be called to map these ranges of physical system memory into the application's task address space.

Suggested Design Model

The optimum design model is one that minimizes kernel overhead at all cost. A similar model to that suggested in Section 11.3.4 is again appropriate for this environment.

11.3.6 Video Memory Addressability in VM Mode

In 'Video Memory Location in Coprocessor Address Space' on page 98 there is a description of how the XGA coprocessor differentiates video memory from system memory. When operating the XGA subsystem in VM mode, this comparison is done post-translate, on physical address space, while all addresses passed to the coprocessor are pre-translate, that is on linear address space. When building VM addressa-

bility to system memory bitmaps for the XGA subsystem, it is also necessary to map local video memory into the page directory structure to allow the XGA coprocessor to differentiate video memory from system memory.

11.3.7 System Memory Access Limitation

The XGA subsystem can be plugged into any 16 or 32 bit slot in any i386SX, i386DX and i486 systems. In a 16 bit slot, the address range is limited since there are only 24 address lines on 16 bit slots. The range of physical addressability to system memory using DMA busmastership is limited to 24 bit physical address space (or 16 Megabytes) when the subsystem occupies a 16 bit slot.

Systems based on the i386SX are 16 bit throughout, and 16 Mbytes is the limit of addressability of the system processor in any case so there are no constraints.

The constraint applies when

- It is a 32 bit system based on the i386DX or i486
- There is more than 16 Megabytes of physical memory installed
- The XGA subsystem is plugged into a 16 bit slot.

In this case, the XGA coprocessor cannot access memory located above the 16 Mbyte line in physical address space. To determine if the XGA subsystem is in a 16 bit slot, examine the Auto-Configuration register, as described in 'Auto-Configuration Register (Index: 04)' on page 24. The application must ensure (with operating system assistance if necessary) that all memory bitmaps on which the XGA processor is asked to operate are located below the 16 Mbyte line in physical address space.

The alternative is for the application to specify that the XGA subsystem is always plugged into 32 bit slots on 32 bit systems.

12 Upwards Compatibility

Upwards compatibility problems can be minimized by sensible programming practices, and some specific precautions

12.1 XGA Subsystem POS ID Allocations

A number of POS ID's have been pre-allocated to the XGA subsystem and follow-on XGA register compatible subsystems, as follows:

8FD8h
8FD9h
8FDAh
8FDBh

Application writers should check for all these POS ID's when determining the existence and location of the XGA subsystem in the system.

12.1.1 General Register Usage

To avoid conflicts with possible future changes in the use of registers or register fields, applications **must** comply with the *Register Usage Guidelines* at the start of the various register definition sections.

12.1.2 Video BIOS Mode 14h

Video BIOS mode 14h has been reserved to support the 132 column text mode. Applications should plan to use BIOS support for this mode as it becomes available, and should, therefore, query Video BIOS for the existence of the mode, and in the case of a positive response, the Video BIOS Int 10h Set mode should be used for mode setting. Only in the absence of Int 10h Video BIOS support should the direct mode setting procedure described in this chapter be used.

12.1.3 PS/2 Video Memory Apertures

As described in Section 11.2 none of the apertures may always be relied upon to exist, depending on configurations, bus size, etc. For maximum flexibility, applications are recommended to avoid using the apertures, but if this is not possible, the following considerations apply:

- 1 Providing the XGA subsystem is not plugged into a 16 bit slot on a 32 bit system at least one of the two potential protect mode apertures should always be available.
- 2 The user of the application should be instructed that the XGA subsystem must be installed in a 32 bit slot on a 32 bit system.

13 Programming the XGA Subsystem in Extended Graphics Mode

Having set the subsystem into the required mode as described in Section 11.1.1 this section describes using the Extended Graphics Functions of the XGA coprocessor.

13.1 XGA Coprocessor Pixel Interface Registers

All extended graphics functions devolve down to graphics update operations involving up to 4 Pixel Maps. A Pixel Map is defined by five registers:

- 1 Pixel Map Index Register
- 2 Pixel Map n Base Pointer Register
- 3 Pixel Map n Width Register
- 4 Pixel Map n Height Register
- 5 Pixel Map n Format Register

13.1.1 Pixel Map Index Register (OFFSET 12h)

The Pixel Map Index Register defines which of the 4 possible maps is to be defined. The encoding of this 4-bit register is as follows:

Mask Map	0
Pixel Map A	1
Pixel Map B	2
Pixel Map C	3

Example:

To use Pixel Map A, WRITE 01h to `copr_regs` offset 12h.

13.1.2 Pixel Map Base Address Register (OFFSET 14h)

The Pixel Map Base Address Register defines the byte address in memory of the start of the Pixel Map. It is a 32-bit address register and can therefore address up to 4096 Mbytes of memory. A Pixel Map can be defined to be in the XGA video memory or in system memory.

As described in 'Video Memory Location in Coprocessor Address Space' on page 97, to define a Pixel Map as being in XGA video memory, the address put in this register must be in the range:

Video Memory Base Address ↔ (Video Memory Base Address + Video Memory size)

If the Pixel map is in system memory and the Micro Channel interface is a 16 bit interface (for example, if the XGA adapter is installed in a 16 bit slot) then the address of the map must be below 16 Mbytes.

13.1.3 Pixel Map Width Register (OFFSET 18h)

The Pixel Map Width is measured in pixels and is defined as 1 less than the required width.

Examples:

```
To set the Width of a Pixel Map to 640 pixels,
WRITE 027Fh to copr_regs offset 18h
To set the Width of a Pixel Map to 1024 pixels,
WRITE 03FFh to copr_regs offset 18h
```

13.1.4 Pixel Map Height Register (OFFSET 20h)

The Pixel Map Height is measured in pixels and is defined as 1 less than the required height.

Examples:

```
To set the Height of a Pixel Map to 480 pixels,
                               WRITE 01DFh to copr_regs offset 20h
To set the Height of a Pixel Map to 768 pixels,
                               WRITE 02FFh to copr_regs offset 20h
```

13.1.5 Pixel Map Format Register (OFFSET 1Ch)

This register specifies the bits/pixel of the Pixel Map. The encoding of the register is as follows:

```
1 bit/pixel Intel format      ( 00h )
2 bits/pixel Intel format     ( 01h )
4 bits/pixel Intel format     ( 02h )
8 bits/pixel Intel format     ( 03h )
1 bit/pixel Motorola format   ( 08h )
2 bit/pixel Motorola format   ( 09h )
4 bit/pixel Motorola format   ( 0Ah )
8 bit/pixel Motorola format   ( 0Bh )
```

Example:

```
For an 8 bit/pixel Motorola format Pixel Map,
                               WRITE 0Bh to copr_regs offset 1Ch
```

The relationship between Intel and Motorola format Pixel maps is discussed in Section 4.1.1 and Section 13.5.

All four Pixel Maps (Maps A, B and C and the Mask map) can be initialized in this manner ready for later use. Maps A, B and C can be used interchangeably as the source, destination or pattern in all subsequent pixel operations.

13.1.6 Other Registers

For simple operations the Pixel Interface Control register should be cleared.

Example:

```
WRITE 00h to copr_regs offset 11h
```

For simple operations the Destination Color Compare Register should be set so that it has no effect on the operation.

Example:

```
WRITE 04h to copr_regs offset 4Ah
```

To allow all planes of a Pixel Map to be updated, the Pixel Bit Mask should be turned on. That is set all bits to a '1' that are required for the pixel size selected.

Example:

```
WRITE 00FFh to copr_regs offset 50h for 8 bits per pixel
```

For simple operations the Carry Chain Mask should be turned on. That is set all bits to a '1' that are required for the pixel size selected.

Example:

```
WRITE FFh to copr_regs offset 54h for 8 bits per pixel
```

13.2 Using the Coprocessor to Perform a Pixel Blit (PxBlt)

This section describes in detail the actions necessary to use the XGA coprocessor to perform a typical simple PxBlt.

Various types of PxBlt can be performed but the subject of this example is for a PxBlt into video memory using the Foreground Color Register as the Source Data. The effect achieved will be to draw a solid rectangle into the Display Pixel Map.

This section will detail the steps necessary to perform the PxBlt mentioned above with a foreground color of 05h, 100 pixels wide and 60 pixels deep. This is positioned at screen coordinates X=200 and Y=150.

The example is summarized in the table below and each value is explained in detail in the following sections which also give useful information on the other forms of PxBlt available.

Value	Copr_regs Offset
03h	48h
05h	58h
0063h	60h
003Bh	62h
00C8h	78h
0096h	7Ah
08118000h	7Ch

13.2.1 Mixes and Colors

Before a coprocessor operation can be performed, the background and foreground 'mixes' have to be set. Mixes are logical or arithmetic functions performed on the source and destination data when performing a coprocessor operation. The full range of mix functions available are as follows:

Code	Function
0	zeros
1	source AND destination
2	source AND NOT destination
3	source
4	NOT source AND destination
5	destination
6	source XOR destination
7	source OR destination
8	NOT source AND NOT destination
9	source XOR NOT destination
A	NOT destination
B	source OR NOT destination
C	NOT source
D	NOT source OR destination
E	NOT source OR NOT destination
F	ones
10	maximum
11	minimum
12	add with saturate
13	subtract (destination - source) with saturate
14	subtract (source - destination) with saturate
15	average

Foreground and Background Mix Registers

The mixes to be applied to foreground and background pixels are specified in these two registers. The contents of the pattern map determine which pixels are foreground and which are background. For this simple example the PxBlit is solid, and so contains only foreground pixels. The foreground mix register should be set to 'SOURCE' to give a readily understandable result on the screen.

For our example:

```
WRITE 03h to copr_regs offset 48h (Foreground Mix Register)
```

Foreground & Background Color Registers

The colors to be used for foreground and background pixels are specified in these two registers. In this simple example the PxBlit is solid and so only the Foreground Color Register needs to be set up.

For our example:

```
WRITE 05h to copr_regs offset 58h (Foreground Color Register)
```

Other forms of PxBlit (for example, video memory to video memory and so on) PxBlit from a Source Map into a Destination Map and therefore do not use these color registers.

13.2.2 PxBlit Dimensions

The Operation Dimension 1 Register should be loaded with the WIDTH of PxBlit that is to be performed. The value loaded into the register should be 1 pixel less than the required Width (in pixels).

For our example:

```
For a PxBlit 100 pixels wide, WRITE 0063h to copr_regs offset 60h
```

The Operation Dimension 2 Register should be loaded with the HEIGHT of PxBlit that is to be performed. The value loaded into the register should be 1 pixel less than the required Height (in pixels).

For our example:

```
For a PxBlit 60 pixels High, WRITE 003Bh to copr_regs offset 62h
```

13.2.3 Pixel Map, Source & Destination

Source Map X and Y Registers

The Source Map is initialized as detailed in the previous chapter. Two registers exist that contain the X & Y offset positions within the Source map of the start of the source data for a PxBlit. These registers are used if you are performing a PxBlit using a Source Map. In this example these registers are unused.

Destination Map X and Y Registers

The Destination Map is initialized as detailed in the previous chapter. Two registers exist that contain the X & Y offset positions within the Destination map of the start of the PxBlit.

For our example:

```
To position the PxBlit at X=200 and Y=150 in the destination map
WRITE 00C8h to copr_regs offset 78h (Destination Map X position)
WRITE 0096h to copr_regs offset 7Ah (Destination Map Y position)
```

Pattern Map X and Y Registers

The Pattern Map is initialized as detailed in the previous chapter. Two registers exist that contain the X & Y offset positions within the Pattern map of the start of the pattern data for a PxBlit. These registers are used if you are performing a PxBlit using a Pattern Map.

For this example these registers are unused.

Mask Map Origin X and Y Offset Registers

The Mask Map is initialized as detailed in the previous chapter. Two registers exist that contain the X & Y offset positions of the start of the Mask Map relative to the top left corner of the Destination Map. These registers are used if you are performing a PxBit using a Mask Map.

For this example these registers are unused.

13.2.4 Pixel Operations Register

This is a 32-bit register which defines the operation that the coprocessor performs.

31	30	29	28	27	24	23	20	19	16	15	12	11	8	7	6	5	4	3	2	0	
												X	X	X	X				X		
1	2	3			4			5			6			7		8		9			

Figure 13.1 Bit Layout Pixel Operations Register

The bits 0:31 are shown on the top of the diagram above and fields 1–9 are shown on the bottom. The definition of these fields are:

- 1 Background Source
- 2 Foreground Source
- 3 Step Function
- 4 Source Pixel Map
- 5 Destination Pixel Map
- 6 Pattern Pixel Map
- 7 Mask Pixel Map
- 8 Drawing Mode
- 9 Direction Octant

These fields will be described in turn until we have assembled the complete Pixel Operations Register contents.

Background Source

These bits determine the origin of the Background source pixels when an operation is performed.

The encoding for these bits is as follows:

Background Color '00'b (for example, for a fixed register value to video memory PxBit).

Source Pixel Map '10'b (for example, for a video memory to video memory PxBit).

For this example, there is no background color, and the field is ignored.

Background Source = '00'b

Foreground Source

These bits determine the origin of the Foreground Source pixels when an operation is performed.

The encoding for these bits is as follows.

Foreground Color '00'b (for example, for a fixed register value to video memory PxBlt).

Source Pixel Map '10'b (for example, for a video memory to video memory PxBlt).

For this example:

Foreground Source = '00'b (Solid Foreground Color)

Step Function

These bits define the type of operation that the coprocessor is required to do

Draw & Step Read	'0010'b
Line Draw Read	'0011'b
Draw & Step Write	'0100'b
Line Draw Write	'0101'b
PxBlt	'1000'b
Inverting PxBlt	'1001'b
Area Fill PxBlt	'1010'b

For this example:

Step Function = '1000b' (PxBlt)

Source Pixel Map

These bits define which Pixel Map is used as the Source Map in the operation. This enables different maps to be setup in advance, and defined for use as this register is loaded.

The encoding for these bits is as follows:

Pixel Map A	'0001'b
Pixel Map B	'0010'b
Pixel Map C	'0011'b

For this example the contents of this field will be ignored.

Source Pixel Map = '0001'b (must not be a reserved value)

Destination Pixel Map

These bits define which Pixel Map is used as the Destination Map in the operation. This enables different maps to be setup in advance, and defined for use as this register is loaded.

The encoding for these bits is as follows:

Pixel Map A	'0001'b
Pixel Map B	'0010'b
Pixel Map C	'0011'b

For this example:

Destination Pixel Map = '0001'b (Pixel Map A)

Pattern Pixel Map

These bits define which Pixel Map is used as the Pattern Map in the operation. This enables different maps to be setup in advance, and defined for use as this register is loaded.

The encoding for these bits is as follows:

Pixel Map A	'0001'b
Pixel Map B	'0010'b
Pixel Map C	'0011'b
Foreground (fixed)	'1000'b
Generated from Source	'1001'b

For this example:

Pattern Pixel Map = '1000'b (Foreground (fixed), for a solid Pxblt)

Mask Pixel Map

These bits define whether the Mask Map is used, or not, in the operation.

The encoding for these bits is as follows:

Mask Map Disabled	'00'b
Mask Map Boundary Enabled	'01'b
Mask Map Enabled	'10'b

For this example:

Mask Pixel Map = '00'b (Mask Map disabled)

Drawing Mode

These bits only concern line drawing only and so are discussed else where. They are ignored during a PxBlt.

For this example:

Drawing Mode = '00'b

Direction Octant

These bits, when concerned with PxBlts determine the direction that the PxBlt is drawn in.

The encoding for these bits is as follows:

'000'b or '001'b	Start at Top LH corner of Area increasing right and down.
'100'b or '101'b	Start at Top RH corner of Area increasing left and down.
'010'b or '011'b	Start at Bottom LH corner of Area increasing right and up.
'110'b or '111'b	Start at Bottom RH corner of Area increasing left and up.

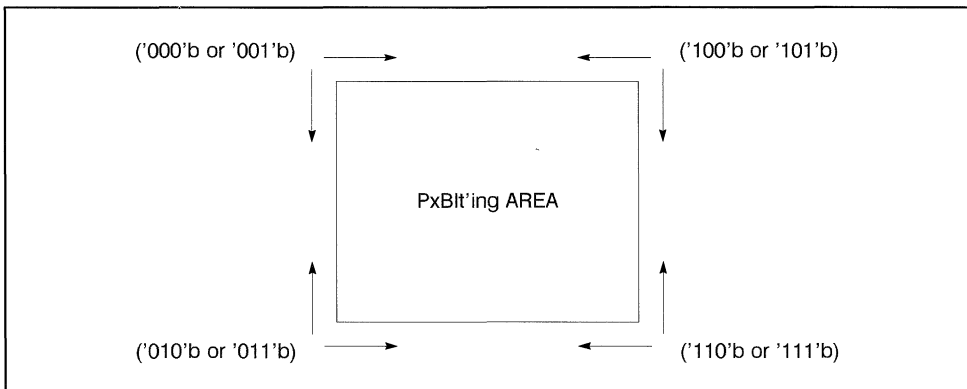


Figure 13.2 Operation Direction Diagram

These bits are normally set to '000'b, but other values are necessary to avoid pixel corruption when Source and Destination rectangles overlap.

For this example:

Direction Octant = `000'b (Top Left)

Conclusion

Putting all these together, for our PxBit the Pixel Operations Register should be set as:

31	30	29	28	27	24	23	20	19	16	15	12	11	8	7	6	5	4	3	2	0							
0	0	0	0	1	0	0	0	0	0	0	1	1	0	0	0	X	X	X	X	0	0	0	0	X	0	0	0

Figure 13 3 Example Definition for Pixel Operations Register

For this example:

WRITE 08118000h to copr_regs offset 7Ch

13.3 Using the Coprocessor to Perform a Bresenham Line Draw

The next example is the detailed steps necessary to draw a line of palette color 05h from (20,15) to (80,35).

The example is summarized in the table below and each value is then explained in detail in the following sections which also give useful information on the other line drawing options available.

Value	Copr_regs Offset
03h	48h
05h	58h
-20d	20h
40d	24h
-80d	28h
59d	60h
20d	78h
15d	7Ah
05118000h	7Ch

13.3.1 Mixes and Colors

Before a coprocessor operation can be performed, the Background and Foreground 'mixes' have to be set. Mixes are logical or arithmetic functions performed on the Source and Destination data when performing a coprocessor operation. The mix functions available are as follows:

Code	Function
0	zeros
1	source AND destination
2	source AND NOT destination
3	source
4	NOT source AND destination
5	destination
6	source XOR destination
7	source OR destination
8	NOT source AND NOT destination
9	source XOR NOT destination
A	NOT destination
B	source OR NOT destination
C	NOT source
D	NOT source OR destination
E	NOT source OR NOT destination
F	ones
10	maximum
11	minimum
12	add with saturate
13	subtract (destination - source) with saturate
14	subtract (source - destination) with saturate
15	average

Foreground and Background Mix Registers

The Foreground and Background Mix registers allow a mix (as detailed in the table above) to be specified. These Registers are discussed in the previous example in Section 13.2.

For the purposes of the simple example being followed here, the Foreground Mix Register should be loaded with 'SOURCE'. The Background Mix Register is not used in this example.

For our example:

```
WRITE 03h to copr_regs offset 48h (Foreground Mix Register)
```

Foreground and Background Color Registers

The Foreground Color register should be set to the color required for the line.

For this example:

```
WRITE 05h to copr_regs offset 58h (Foreground Color Register)
```

13.3.2 Bresenham Line Draw

The algorithm used to perform the linedraw function on the XGA is the Bresenham Line Draw Algorithm. This algorithm operates with all parameters normalized to the first octant (Octant 0).

The first task is to calculate DeltaX and DeltaY as shown in the figure 13.4.

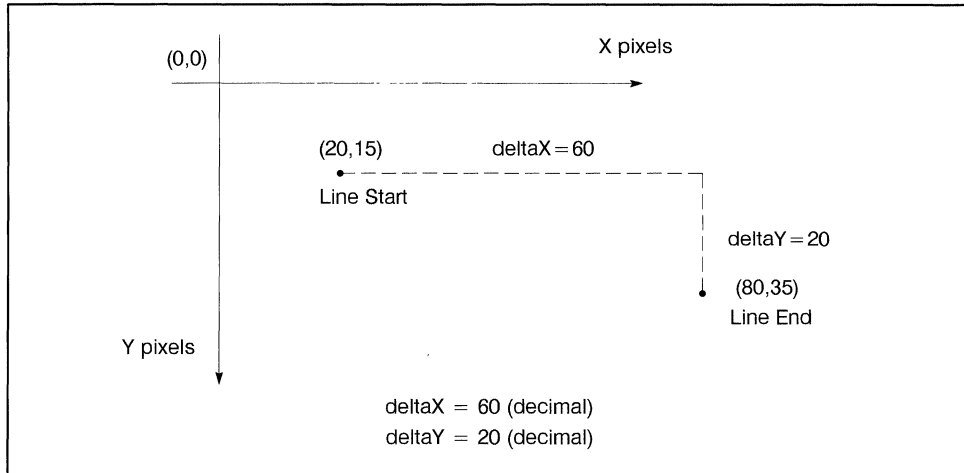


Figure 13.4 Line Draw Example in Octant 0

A line in the first octant as shown above has deltaX greater than deltaY with both deltaX and deltaY positive and deltaX greater than deltaY. If a line is to be drawn in another octant, the octant information is specified in the octant bits of the Pixel Operation Register and the line drawn as if it was in the first octant.

To normalize a line to the first octant the following rules should be followed:

- If deltaX is -ve , set DX in Octant bits of the Pixel Operation Register and make deltaX +ve.
- If deltaY is -ve , set DY in Octant bits of the Pixel Operation Register and make deltaY +ve
- If $\text{deltaY} \geq \text{deltaX}$, set DZ in Octant bits of the Pixel Operation Register and exchange deltaX and deltaY.

The terms deltaX and deltaY referred to below are the lengths of the line after it has been normalized to Octant 0. The algorithm requires several parameters to be calculated. These are.

Bresenham Error Term Register

Bresenham Error Term $ET = (2 * \text{deltaY}) - \text{deltaX}$

For this example:

```
WRITE -20 decimal (FFECh) copr_regs offset 20h
```

Bresenham Constant K1 Register

Bresenham Constant $K1 = 2 * \text{deltaY}$

For this example:

```
WRITE +40 decimal (0028h) copr_regs offset 24h
```

Bresenham Constant K2. Register

Bresenham Constant K2 = $2 * (\text{deltaY} - \text{deltaX})$

For this example:

```
WRITE -80 decimal (FFB0h) copr_regs offset 28h
```

Operation Dimension Registers

The Operation Dimension 1 Register should be loaded with $(\text{deltaX} - 1)$ after normalization.

For this example:

```
WRITE +59 decimal (003Bh) to copr_regs offset 60h
```

The Operation Dimension 2 Register is not used for line draw.

13.3.3 Pixel Map, Source and Destination

Source Map X and Y Registers

The Source Map is defined as described in Section 13.1. Two registers exist that contain the X & Y offset positions within the Source map of the start of the source data for a PxBlt. These registers are used if you are drawing a line using a Source Map. In this example these registers are unused.

Destination Map X and Y Registers

The Destination Map is initialized as described in Section 13.1. Two registers exist that contain the X & Y offset positions within the Destination map of the start of the Line.

In this example:

```
WRITE +20 decimal (0014h)
  to copr_regs offset 78h (Destination Map X position)
WRITE +15 decimal (000Fh)
  to copr_regs offset 7Ah (Destination Map Y position)
```

Pattern Map X and Y Registers

The Pattern Map is initialized as described in Section 13.1. Two registers exist that contain the X & Y offset positions within the Pattern map of the start of the pattern data for a Line. These registers are used if you are drawing a line using a Pattern Map. In this example these registers are unused.

Mask Map Origin X and Y Offset Registers

The Mask Map is initialized as described in Section 13.1. Two registers exist that contain the X & Y offset positions of the start of the Mask Map relative to the top left corner of the Destination Map. These registers are used if you are drawing a line using a Mask Map. In this example these registers are unused.

13.3.4 Pixel Operations Register.

This is a 32-bit register which defines the operation that the coprocessor performs.

31	30	29	28	27	24	23	20	19	16	15	12	11	8	7	6	5	4	3	2	0	
												X	X	X	X				X		
1		2		3			4			5			6			7		8		9	

Figure 13.5 Bit Layout Pixel Operations Register

The bits 0:31 are shown on the top of the diagram above and fields 1-9 are shown on the bottom. The definition of these fields are:

- 1 Background Source
- 2 Foreground Source
- 3 Step Function
- 4 Source Pixel Map
- 5 Destination Pixel Map
- 6 Pattern Pixel Map
- 7 Mask Pixel Map
- 8 Drawing Mode
- 9 Direction Octant

These fields will be described in turn until we have assembled the complete Pixel Operations Register contents.

Background Source

These bits determine the origin of the Background source pixels when an operation is performed.

The encoding for these bits is as follows:

Background Color '00'b (for example, for a fixed pattern line draw using a fixed register value)

Source Pixel Map '10'b (for example, for a variable color data pattern held in video memory to video memory draw).

In this example the contents of this field are ignored as the line is solid and so has no Background pixels:

Background Source = '00'b

Foreground Source

These bits determine the origin of the Foreground Source pixels when an operation is performed.

The encoding for these bits is as follows:

Foreground Color '00'b (for example, for a fixed pattern line draw using a fixed register value)

Source Pixel Map '10'b (for example, for a variable color data pattern held in video memory to video memory draw).

For this example:

Foreground Source = '00'b (Solid Foreground Color)

Step Function

These bits define the type of operation that the coprocessor is required to do.

Draw & Step Read	'0010'b
Line Draw Read	'0011'b
Draw & Step Write	'0100'b
Line Draw Write	'0101'b
PxBlt	'1000'b
Inverting PxBlt	'1001'b
Area Fill PxBlt	'1010'b

For this example:

```
Step Function = '0101b' (Line Draw Write)
```

Source Pixel Map

These bits define which Pixel Map is used as the Source Map in the operation. This enables different maps to be setup in advance, and defined for use as this register is loaded.

The encoding for these bits is as follows:

Pixel Map A	'0001'b
Pixel Map B	'0010'b
Pixel Map C	'0011'b

In this example the contents of this field is ignored:

```
Source Pixel Map = '0001'b (must not be a reserved value)
```

Destination Pixel Map

These bits define which Pixel Map is used as the Destination Map in the operation. This enables different maps to be setup in advance, and defined for use as this register is loaded.

The encoding for these bits is as follows:

Pixel Map A	'0001'b
Pixel Map B	'0010'b
Pixel Map C	'0011'b

For this example:

```
Destination Pixel Map = '0001'b (Pixel Map A)
```

Pattern Pixel Map

These bits define which Pixel Map is used as the Pattern Map in the operation. This enables different maps to be setup in advance, and defined for use as this register is loaded.

The encoding for these bits is as follows:

Pixel Map A	'0001'b
Pixel Map B	'0010'b
Pixel Map C	'0011'b
Foreground (fixed)	'1000'b
Generated from Source	'1001'b

For this example:

```
Pattern Pixel Map = '1000'b
(Foreground (fixed), for a solid Line)
```

Mask Pixel Map

These bits define whether Mask Map is used, or not, in the operation.

The encoding for these bits is as follows:

Mask Map Disabled	'00'b
Mask Map Boundary Enabled	'01'b
Mask Map Enabled	'10'b

For this example:

Mask Pixel Map = '00'b (Mask Map disabled)

Drawing Mode

These bits determine the attributes of a Line Draw.

The encoding for these bits is as follows:

Draw All Pixels	'00'b
Draw First Pixel Null	'01'b
Draw Last Pixel Null	'11'b
Mask Area Boundary	'11'b

The first three of these options can be used when drawing a line. The fourth option is for use when drawing the outline of a shape to be filled using the AreaFill capability of the hardware.

For this example:

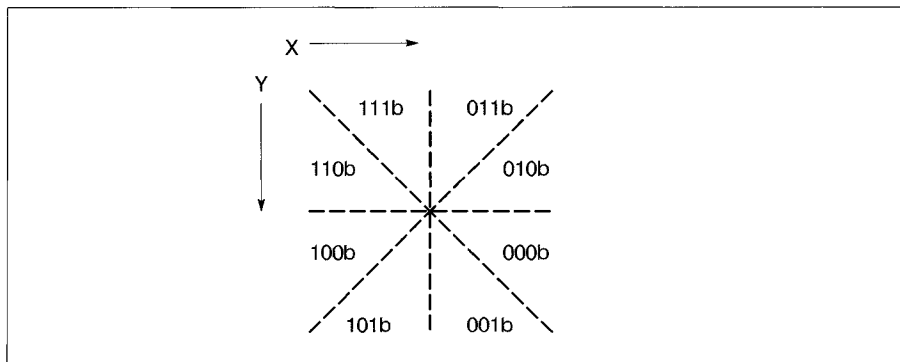
Drawing Mode = '00'b (Draw All Pixels)

Direction Octant

These bits, when concerned with Line Draws determine the direction that the Line is drawn in.

The encoding for these bits is as follows:

Bit 2(DX)	'1'b if Negative X direction
Bit 2(DX)	'0'b if Positive X direction
Bit 1(DY)	'1'b if Negative Y direction
Bit 1(DY)	'0'b if Positive Y direction
Bit 0(DZ)	'1'b if $ X < Y $
Bit 0(DZ)	'0'b if $ X > Y $, (magnitude)



For this example:

Direction Octant = '000'b (X +ve, Y +ve, $|X| > |Y|$)

Conclusion

Putting all these together for our example Line Draw operation, the Pixel Operations Register should be set as:

31	30	29	28	27	24	23	20	19	16	15	12	11	8	7	6	5	4	3	2	0											
0	0	0	0	0	1	0	1	0	0	0	1	0	0	0	1	1	0	0	0	X	X	X	X	0	0	0	0	X	0	0	0

Figure 13.6 Example Definition for Pixel Operations Register

For this example:

```
WRITE 05118000h to copr_regs offset 7Ch
```

13.4 Memory Access Modes (Reg. 21x9)

This register is used to control the format of the data supplied by the system processor through a system video memory aperture. For conventional use, this register should be set to match the format of the data as seen by the system processor (see Section 13.5), and the depth of the video memory bitmap.

It is possible to exploit the different formats available using this register to achieve useful and otherwise difficult conversions.

13.5 Motorola/Intel Format

The internal organization of the video memory is Intel format. However images and bitmaps are traditionally stored in Motorola format. It is important and necessary to understand the format of the application's bitmaps in system memory to get the correct results. The different formats are described in Section 4.1.1

The internal organization of video memory as Intel format can be entirely hidden by appropriate use of the Memory Access Mode register (Section 13.4) and the various Coprocessor Pixel Map format registers.

13.5.1 System Processor Access

When using the system processor to read or write data direct to or from video memory via a system video memory aperture, it is necessary to specify the format of the data via the Memory Access Mode Register (21x9).

13.5.2 XGA Coprocessor Accesses

The format of all bitmaps in system memory must be specified, via the Pixel Map format register. This parameter is ignored for bitmaps in video memory.

13.5.3 Exploitation

Writing data in one format, and reading it back in another is a technique that performs many useful and otherwise difficult and/or expensive bitmap conversions.

14 Other Programming Considerations

14.1 Overlapping BitBlits

14.1.1 Pixel Block Transfer (PxBlt)

The coprocessor *PxBlt* function is used to transfer a rectangular block of pixels from the Source to the Destination subject to a number of modifiers, in other words a standard BitBlit operation. Where the Source and Destination rectangles do not overlap, the order of processing pixels is obviously immaterial. In cases where the rectangles do overlap, it is important to pre-determine that this is the case, and to program the PxBlt direction (via the *Direction Octant*) correctly to ensure the expected result.

14.1.2 Inverting PxBlt

The inverting PxBlt is intended to convert images from the traditional application format of Y increasing upwards to the traditional display hardware format of Y increasing downwards. As such a PxBlt operates from both ends towards the middle, an *Inverting PxBlt* involving overlapping Source and Target rectangles inevitably overwrites pixels. The lesson here is that *Inverting Pxblts* on overlapping rectangles should be avoided, unless for carefully considered special effects.

14.2 Sprite Handling

14.2.1 Sprite Loading

The sprite is loaded as a 64 x 64 x 2bpp Intel format image definition. As the application's Sprite definition is invariably held in 2 separate 1bpp Motorola format bitmaps it is necessary to merge and 'pixel swap' the Sprite definition into 2bpp Intel format before loading the Sprite.

14.2.2 Sprite Positioning

The position of the sprite is then controlled by 2 separate controls, as follows:

Sprite Start Registers The sprite is positioned on the display surface by specifying the position of the upper left corner of the sprite definition relative to the upper left corner of the visible bitmap using the Horizontal and Vertical Start registers.

Sprite Preset Registers The Sprite Start registers only accept positive values, and cannot be used to move the sprite partially off the display surface at the left and top edges. The Sprite Preset Registers are used to offset the start of the displayed sprite definition horizontally and/or vertically relative to the loaded definition.

For example, if it is desired to display a 64 x 64 sprite with the leftmost 32 pels outside the left edge of the display surface, set the Horizontal Start Register to 0, and the Horizontal Preset Register to 32. The Start position has now been preset to the centre of the loaded definition, giving the desired effect.

The Sprite Preset can also be used to display sprites smaller than 64x64.

14.3 Waiting for Hardware Not Busy

The XGA coprocessor operates asynchronously with the system processor. It is necessary to wait for the previous operation to complete before issuing the next operation. There are 2 ways to do this, each with its own drawbacks and advantages.

Polling the Busy bit There is a *Coprocessor Busy* bit provided in XGA coprocessor *PI Control register* that may be polled to ascertain the completion of the previous operation prior to initiating the next operation.

Continuous polling of this bit slows down the coprocessor which must pause in its current operation to process the 'read' of the *PI control register*

If this method is chosen, it is advisable to code a double polling loop, only checking the *Coprocessor busy* bit once for every 100 times around the loop, for example.

Advantages:

- Minimal overhead. For typical PxBlts used to display text, the previous PxBlt operation will be almost complete before the system processor is ready to issue the next operation.
- Simplicity

Disadvantages:

- Frequent use delays the XGA coprocessor. This can be partially reduced by a double loop algorithm.
- The processor is kept busy doing nothing, although it has to be 'doing nothing' for a long time to exceed the interrupt response codepath.

Operation complete interrupt The coprocessor can be programmed to cause an interrupt to the system processor when an operation is completed.

This interrupt is a shared level, and interrupt response time therefore depends on other interrupt handlers chained on this shared level. In protect mode operating systems, in particular, the overheads and restrictions placed on interrupt handlers may make the performance of this technique prohibitive.

Advantages:

- The XGA coprocessor is not slowed while waiting for completion.
- The system processor may be freed up for other tasks.

Disadvantages:

- Program complexity.
- Interrupt response time gives a threshold in size of operation that is only exceeded by large PxBlt operations. The more complex the operating system, the higher the interrupt response time, and the larger the operation must be to benefit from using interrupts to notify the application of operation complete.

14.4 Destination Bitmap Width Restriction

Incorrect results can be obtained if the XGA coprocessor is used to write over the edge of a destination bitmap where the edge of the bitmap is not four byte aligned. To avoid this, either:

- Ensure that all destination bitmaps have a base address that is on a four byte boundary, and are an exact multiple of 4 bytes in width.

The visible display bitmap naturally complies with this restriction.

or

- Where bitmaps are not aligned, software clip all PxBlts in advance so that the destination bitmap boundary is not crossed during the PxBlt.

14.5 Line Length Restriction

The XGA coprocessor *Destination X Address* and *Destination Y Address* register accept coordinates in the range -2048 to +6143. This gives a *guardband* effect, where it is possible to write coordinates anywhere in this range, and the operation is hardware scissored to the edge of the destination bitmap. The limit on bitmap size for coprocessor operations is 0 to 4095.

However, the *Operation Dimension 1* register only accepts values in the range 0 to 4095. It is therefore not possible to draw a line in a single operation across the entire guardband coordinate space.

A 2 stage line draw can be performed easily, since the line parameters (ET, K1, K2, Destination X & Y, Pattern X etc.) are already set up in the hardware at the end of the previous line segment. It is necessary merely to update the new line length in the *Operation Dimension 1* register to draw the remainder of the line.

14.6 System Register Usage

When programming the XGA subsystem, it is often necessary to maintain addressability to:

- 1 XGA coprocessor memory mapped address space.
- 2 XGA state data segment (application dependent) containing the IO base address, in other words the location of the XGA registers in IO space.
- 3 The normal function dependent application data, such as parameter blocks.
- 4 Global application dependent data.

In addition, many of the XGA registers are 32 bit registers.

To program the XGA subsystem efficiently, it is helpful to use the full i386 register set, specifically the FS and GS segment registers and the 32 bit extended data registers.

Use of the extra segment registers allows concurrent addressability to all the separate data areas to be maintained without frequent segment register loading — a particularly expensive operation in protect modes.

14.7 Direct Color Mode

This section deals with matters unique to the Direct Color mode of the XGA subsystem.

14.7.1 Palette Loading

It is necessary to load the palette with a fixed set of values. These are described in Section 4.2.

14.7.2 Coprocessor Support

The XGA coprocessor does not support the 16 bpp mode. This mode is a display mode only and must be programmed using the system processor to access the video memory display buffer directly using one of the system video memory apertures (See also Section 11.2 & Section 13.4).

The processor is not, however, disabled while in this mode. The restriction is rather that the pixel map formats available for coprocessor operations is restricted to 1, 2, 4 or 8 bpp. The graphics coprocessor can be used while in this mode if this is allowed for. Some ingenuity is necessary to achieve useful results using the coprocessor in this way, but the rewards could be justified.

BitBit Operations By using the PxBlt operations on an 8 bpp bitmap, doubling the dimension width of the bitmaps involved and avoiding arithmetic mixes, BitBit operations are possible. Use of the 1bpp pattern and mask maps are possible if carefully considered and calculated.

Text Operations Text operations using the coprocessor PxBlt function rely on 1 bpp patterns. By doubling the width of the individual character bitmap patterns, (interspersing the active bits with zero bits), and writing the high and low order bytes of the required color index separately, Text Operations are possible.

15 Sample Code

15.1 Putting the XGA Subsystem into Extended Graphics Mode

15.1.1 Pseudo Code

Main Program

- Locate first XGA subsystem with attached monitor
- If XGA is current system VGA subsystem
 - Chain Int 10h Video handler
 - Chain Int 21h DOS Function handler
 - Chain Int 23h Ctrl Break Exit Address
 - Chain Int 24h Critical Error handler
- If LIM Expanded Memory Manager installed
 - Call LIM Fn 25.Get Physical Address Array
 - Examine returned list for Memory Contention
 - If contention found
 - Display Warning Message.
 - Terminate Application
- Chain Int 2Fh Screen Switch Notification handler
- Put XGA in highest Extended Graphics Mode for attached monitor (see Section 11.1.1)
- Draw simple rectangle (or whatever)
- Exit

Int 10 Handler

- Examine value of (Ah)
 - 00h Set Mode**
 - Put XGA subsystem in VGA mode (see Section 11.1.2)
 - Chain on to saved Int 10h Video Interrupt handler.
 - 0Fh Return current video state**
 - Set (AL) = 7Fh
 - Interrupt return (IRET)
 - Any other value**
 - Interrupt return (IRET)

Int 21h DOS Function handler

- Examine value of (Ah)
 - 4Ch Program Terminate**
 - Put XGA subsystem in VGA mode
 - UnChain and restore original Int 10h Video handler
 - UnChain and restore original Int 21h DOS function handler
- Chain on to saved Int 21h handler

Int 23h Ctrl Break Exit Address

- Chain on to saved Int 23h handler, using a method that will ensure return of control via this function handler.
- On return from chained handler, Examine Carry Flag (CF). If set
 - Put XGA subsystem in VGA mode
 - UnChain and restore original Int 10h Video handler
 - UnChain and restore original Int 21h DOS function handler
- Interrupt return (IRET)

Int 24h Critical Error handler

- Save video state (or as much as is corrupted by a temporary switch into VGA text mode).
- Put XGA subsystem in VGA mode
- Chain on to saved Int 24h handler, using a method that will ensure return of control via this function handler.
- On return from chained handler, examine AL, as follows:
 - 0, 1, 3**
 - Put XGA subsystem in Extended graphics mode
 - Restore video state
 - 2**
 - UnChain and restore original Int 10h Video handler
 - UnChain and restore original Int 21h DOS function handler
- Interrupt return (IRET)

Int 2Fh Screen Switch Notification Handler

- Examine value of (Ah)

40h Screen Switch Notification

- Examine value of (AL)
 - 01h Impending switch to background.**
 - Put XGA subsystem in VGA mode
 - 02h Impending switch to foreground**
 - Put XGA subsystem in Extended Graphics Mode
 - Semaphore “re-draw required” to application
 - Chain on to saved Int 2Fh handler (if any)

Any other value

- Chain on to saved Int 2F Interrupt vector (if any)

15.1.2 Code Example

Main C Program

```

/* ***** */
/*
/*
/* Program s_ext
/*
/* Description This program is sample code to illustrate entry to
/* Ext Graphics mode and back to VGA mode upon program
/* termination.
/*
/*
/* ***** */

#define POS 0xc4
#define POS_GET_BASE_ADDRESS 0X00
#define FALSE 0x00
#define TRUE 0x01
#define EMM_INT 0x67
#define DEVICE_NAME_LENGTH 0x08
#define MAX_SLOTS 0x09
#define XGA_ID_UPPER 0x8fdb
#define XGA_ID_LOWER 0x8fd8
#define INDEX_SELECT 0x0a
#define INDEX_DATA 0x0b
#define MONITOR_ID 0x52

#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <signal.h>
#include <malloc.h>
#include <memory.h>
#include <stdlib.h>
#include <conio.h>

typedef struct
{
    int page_segment ;
    int page_number ;
} MPAA ; /* Mappable Physical Address Array struct */

typedef struct /* POS Record */
{
    unsigned int pos_id ;
    char pos_byte1 ;
    char pos_byte2 ;
    char pos_byte3 ;
    char pos_byte4 ;
} POS_REC ;

union REGS inregs , outregs ;
struct SREGS segregs ;
unsigned int pos_base_address , IoRegBase , slot_number ;

POS_REC pos_record ;

```

```
long int ROS_add_rec[16] = { 0xc0000 ,
                             0xc2000 ,
                             0xc4000 ,
                             0xc6000 ,
                             0xc8000 ,
                             0xca000 ,
                             0xcc000 ,
                             0xce000 ,
                             0xd0000 ,
                             0xd2000 ,
                             0xd4000 ,
                             0xd6000 ,
                             0xd8000 ,
                             0xda000 ,
                             0xdc000 ,
                             0xde000 } ;

unsigned char vga_data[] = { 0x01 , 0x00 , 0x00 ,
                             0x04 , 0x00 , 0x00 ,
                             0x05 , 0xff , 0x00 ,
                             0x0a , 0xff , 0x64 ,
                             0x0a , 0x15 , 0x50 ,
                             0x0a , 0x14 , 0x50 ,
                             0x0a , 0x00 , 0x51 ,
                             0x0a , 0x04 , 0x54 ,
                             0x0a , 0x7f , 0x70 ,
                             0x0a , 0x20 , 0x2a ,
                             0x00 , 0x01 , 0x00 } ;

unsigned char nm_data[] = { 0x04 , 0x00 , 0x00 , 0x00 ,
                             0x05 , 0xff , 0xff , 0x00 ,
                             0x00 , 0x04 , 0x04 , 0x00 ,
                             0x0a , 0x00 , 0x00 , 0x64 ,
                             0x01 , 0x00 , 0x00 , 0x00 ,
                             0x08 , 0x00 , 0x00 , 0x00 ,
                             0x06 , 0x00 , 0x00 , 0x00 ,
                             0x09 , 0x03 , 0x02 , 0x00 ,
                             0x0a , 0x01 , 0x01 , 0x50 ,
                             0x0a , 0x00 , 0x00 , 0x50 ,
                             0x0a , 0x9d , 0x9d , 0x10 ,
                             0x0a , 0x00 , 0x00 , 0x11 ,
                             0x0a , 0x7f , 0x7f , 0x12 ,
                             0x0a , 0x00 , 0x00 , 0x13 ,
                             0x0a , 0x7f , 0x7f , 0x14 ,
                             0x0a , 0x00 , 0x00 , 0x15 ,
                             0x0a , 0x9d , 0x9d , 0x16 ,
                             0x0a , 0x00 , 0x00 , 0x17 ,
                             0x0a , 0x87 , 0x87 , 0x18 ,
                             0x0a , 0x00 , 0x00 , 0x19 ,
                             0x0a , 0x9c , 0x9c , 0x1a ,
                             0x0a , 0x00 , 0x00 , 0x1b ,
                             0x0a , 0x40 , 0x40 , 0x1c ,
                             0x0a , 0x04 , 0x04 , 0x1e ,
                             0x0a , 0x30 , 0x30 , 0x20 ,
                             0x0a , 0x03 , 0x03 , 0x21 ,
                             0x0a , 0xff , 0xff , 0x22 ,
                             0x0a , 0x02 , 0x02 , 0x23 ,
                             0x0a , 0xff , 0xff , 0x24 ,
                             0x0a , 0x02 , 0x02 , 0x25 ,
```

```

0x0a , 0x30 , 0x30 , 0x26 ,
0x0a , 0x03 , 0x03 , 0x27 ,
0x0a , 0x00 , 0x00 , 0x28 ,
0x0a , 0x03 , 0x03 , 0x29 ,
0x0a , 0x08 , 0x08 , 0x2a ,
0x0a , 0xff , 0xff , 0x2c ,
0x0a , 0xff , 0xff , 0x2d ,
0x0a , 0x00 , 0x00 , 0x36 ,
0x0a , 0x00 , 0x00 , 0x40 ,
0x0a , 0x00 , 0x00 , 0x41 ,
0x0a , 0x00 , 0x00 , 0x42 ,
0x0a , 0x80 , 0x40 , 0x43 ,
0x0a , 0x00 , 0x00 , 0x44 ,
0x0a , 0x0d , 0x0d , 0x54 ,
0x0a , 0x03 , 0x02 , 0x51 ,
0x0a , 0x00 , 0x00 , 0x70 ,
0x0a , 0x0f , 0x0f , 0x50 ,
0x0a , 0x00 , 0x00 , 0x55 ,
0x0a , 0x00 , 0x00 , 0x60 ,
0x0a , 0x00 , 0x00 , 0x61 ,
0x0a , 0x00 , 0x00 , 0x62 ,
0x0a , 0x00 , 0x00 , 0x63 ,
0x0a , 0xff , 0xff , 0x64 } ;

/* colour_default_palette
unsigned char colour_default_palette[] = {
    R      G      B      */
0x00 , 0x00 , 0x00 ,
0x00 , 0x00 , 0xa8 ,
0x00 , 0xa8 , 0x00 ,
0x00 , 0xa8 , 0xa8 ,
0xa8 , 0x00 , 0x00 ,
0xa8 , 0x00 , 0xa8 ,
0xa8 , 0x54 , 0x00 ,
0xa8 , 0xa8 , 0xa8 ,
0x54 , 0x54 , 0x54 ,
0x54 , 0x54 , 0xfc ,
0x54 , 0xfc , 0x54 ,
0x54 , 0xfc , 0xfc ,
0xfc , 0x54 , 0x54 ,
0xfc , 0x54 , 0xfc ,
0xfc , 0xfc , 0x54 ,
0xfc , 0xfc , 0xfc } ;

int      IpInt , cop_instance ;
long int ROS_address ;
char     XGAFound , VRAM_1Meg ;
char     XGAInVGA , ExtG_mode_set , slot_enabled ;
void far *int_10_original_vector ;
void far *int_2f_original_vector ;
FILE     *stream ;
int      delaytime = 1000 ;
char VramIr , A1024x768 ;

unsigned short int VRAM_address_lo , VRAM_address_hi ;

interrupt far int_10( void ) ;
interrupt far int_2f( void ) ;
char *build_ptr( unsigned int , unsigned int ) ;
char emm_installed( void ) ;
int far return_ax( void ) ;

```

```

int far return_7f( void ) ;
void PutXGAInVGA (void) ;
void PutXGAInExtG(void);
int exit_handler( void ) ;
void signal_handler( void ) ;
void co_pro_blit(void);
void CoProWriteByte(int , unsigned char );
void CoProWriteWord(int , unsigned int );
void CoProPrintByte( int , char * );
void WaitForCoProReady(void);
void delay(long int);

/* ***** */
/*
/*
/*      Function main()
/*
/*      Description  This is the program entry point.
/*
/*
/*
/* ***** */

void main( void )
{
    int      index ;
    MPAA     *mpaa ;
    unsigned int  ipdata ;
    unsigned char far *vram_address ;
    unsigned char ip_byte ;

    atexit( exit_handler ) ;
    signal( SIGINT , signal_handler ) ;
    signal( SIGFPE , signal_handler ) ;
    signal( SIGABRT , signal_handler ) ;

    ExtG_mode_set = FALSE ;
    int_l0_original_vector = 0 ;
    int_2f_original_vector = 0 ;

    inregs.h.ah = POS ;                               /* get base POS address */
    inregs.h.al = POS_GET_BASE_ADDRESS ; /* from system services */
    int86( 0x15 , &inregs , &outregs ) ;

    pos_base_address = outregs.x.dx ;
    XGAFound = FALSE ;
    if( outregs.x.cflag )                               /* carry flag set means */
    {                                                     /* not a microchannel */
        printf( "No XGA Installed\n" ) ; /* machine */
    }
    else
    {
        XGAFound = FALSE ;
        slot_enabled = FALSE ;
        for (slot_number = 0 ; slot_number <= MAX_SLOTS ; slot_number++)
        {
            _disable();                               /* Disable interrupts */
            if (slot_number == 0)
            { /* Look at the planar for XGA */

```

```

    outp( 0X094 , 0x0df ) ; /* Enable planar for setup */
}
else
{ /* Look in the slots for XGA */
    inregs.x.ax = 0xc401 ;
    inregs.x.bx = slot_number ;
    int86( 0x15 , &inregs , &outregs ) ;
                                /* enable slot for update */
}
slot_enabled = TRUE ;
/* Get pos record for the slot */
pos_record.pos_id = inpw( pos_base_address ) ;
pos_record.pos_byte1 = (char)inp( pos_base_address + 2 ) ;
pos_record.pos_byte2 = (char)inp( pos_base_address + 3 ) ;
pos_record.pos_byte3 = (char)inp( pos_base_address + 4 ) ;
pos_record.pos_byte4 = (char)inp( pos_base_address + 5 ) ;
IoRegBase = (( pos_record.pos_byte1 & 0x0e ) << 3 ) + 0x2100 ;
if(slot_number == 0)
{
    outp( 0X094 , 0x0ff ) ; /* Enable planar for normal mode */
}
else
{
    inregs.x.ax = 0xc402 ;
    inregs.x.bx = slot_number ;
    int86( 0x15 , &inregs , &outregs ) ;
                                /* enable slot normal mode */
}
slot_enabled = FALSE ;
_enable(); /* Enable interrupts */
/* Check for a valid XGA POS id */
if ( pos_record.pos_id >= XGA_ID_LOWER &&
    pos_record.pos_id <= XGA_ID_UPPER )
{
    /* XGA found in slot */

    /* Look to see if monitor connected to XGA */
    outp( IoRegBase + INDEX_SELECT , MONITOR_ID );
    if ( ( inp(IoRegBase + INDEX_DATA) & 0x0f) != 0x0f )
    {
        /* Monitor connected to XGA */
        XGAFound = TRUE ;

        /* Determine if XGA in VGA */
        ipdata = inp( IoRegBase ) ;
        if( ipdata & 0x01 )
        {
            XGAInVGA = TRUE ;
            /* Chain Int 10H */
            int_10_original_vector = _dos_getvect( 0x10 ) ;
            _dos_setvect( 0x10 , int_10 ) ;
        }
        else
            XGAInVGA = FALSE ;
        /* calculate VRAM address */
        VRAM_address_lo = 0x0 ;
        VRAM_address_hi = ((short int)
            ( pos_record.pos_byte3 & 0xfe )) << 8 ;
    }
}
}

```

```

VRAM_address_hi &lor.= ((short int)
                        (pos_record.pos_bytel & 0x0e)) << 5 ;

/* check VRAM */
outp(IoRegBase + 4 , 0x00 );
outp(IoRegBase + 0 , 0x04 );
outp(IoRegBase + 1 , 0x01 );
outpw(IoRegBase + 0x0a , 0x0064 );

VRAM_1Meg = FALSE ;
outp(IoRegBase + 8 , 0x0c) ;
vram_address = (unsigned char far *)0xa0000000 ;
*vram_address = 0xa5 ;

vram_address++ ;
*vram_address = 0 ;

vram_address-- ;
ip_byte = *vram_address ;

if(ip_byte == 0xa5)
{
    VRAM_1Meg = TRUE ;
}

index = (pos_record.pos_bytel >> 4) & 0xf ;
cop_instance = (pos_record.pos_bytel >> 1) & 0x07 ;
ROS_address = ROS_add_rec[ index ] ;
if( emm_installed() )
{
    /* get number of entries in mappable physical page */
    inregs.x.ax = 0x5801 ;
    int86( 0x15 , &inregs , &outregs ) ;
    if(outregs.x.cx)
    {
        /* entries exist in the list */
        mpaa = (MPAA *)calloc(outregs.x.cx, sizeof(MPAA));
        if(!mpaa)
        {
            printf("Unable to allocate memory\n");
        }
        else
        {
            segregs.es = FP_SEG( mpaa ) ;
            inregs.x.di = FP_OFF( mpaa ) ;
            int86x( 0x67 , &inregs , &outregs , &segregs ) ;

            for( ; outregs.x.cx > 0 ; mpaa++ )
            {
                if(ROS_address == mpaa->page_segment )
                {
                    printf("Extended memory conflict at");
                    printf(" segment address");
                    printf(" %x\n" , mpaa->page_segment ) ;
                    exit(0);
                }
            }
        }
    }
}

```

```

    }
    }
    /* Chain Int 2fH */
    int_2f_original_vector = _dos_getvect( 0x2f );
    _dos_setvect( 0x2f , int_2f );
    PutXGAInExtG();
    co_pro_blit();
}
}
if( XGAFound ) break ;
}
}

/* ***** */
/*
/* Function - pointer = build_ptr( segment , offset )
/*
/* Description - Constructs a pointer from segment and offset.
/*
/*
/*
/*
/* ***** */

char *build_ptr( unsigned int segment , unsigned int offset )
{
    char *ptr ;
    ptr = (char *)(((unsigned long)segment << 16) + offset ) ;
    return( ptr ) ;
}

/* ***** */
/*
/* Function - status = emm_installed()
/*
/* Description - checks to see if extended memory installed
/*
/*
/*
/* ***** */

char emm_installed( )
{
    char *EMM_device_name = "EMMXXXX0" ;
    char *int_67_device_name_ptr ;
    inregs.h.ah = 0x35 ;
    inregs.h.al = EMM_INT ;
    intdosx( &inregs , &outregs , &segregs ) ;
    int_67_device_name_ptr = build_ptr( segreg.es , 0x0a ) ;
    if( memcmp(EMM_device_name, int_67_device_name_ptr,
              DEVICE_NAME_LENGTH )
        return ( FALSE ) ;
    else
        return ( TRUE ) ;
}

```



```

/* ***** */
/*
/* Function - interrupt routine int_10()
/*
/* Description - internal INT 10h interrupt handler
/*
/*
/*
/* ***** */

```

```

interrupt far int_10( void )
{
    unsigned int ax ;
    ax = return_ax() >> 8 ;
    if( ax == 0 )
    {
        /* set xga to vga mode */
        _chain_intr( int_10_original_vector ) ;
    }
    else if ( ax == 0x0f )
    {
        return_7f() ;
    }
    else
    {
        _chain_intr( int_10_original_vector ) ;
    }
}

```

```

/* ***** */
/*
/* Function - Interrupt routine int_2f()
/*
/* Description - Internal INT 2fh interrupt handler
/*
/*
/*
/* ***** */

```

```

interrupt far int_2f( void )
{
    unsigned int ax ;
    unsigned char ah ;
    unsigned char al ;

    ax = return_ax() ;
    ah = (unsigned char)(ax >> 8) ;
    al = (unsigned char)(ax & 0x0f) ;
    if( ah == 0x40 )
    { /* Screen switch notification received */
        if (al == 0x01) /* Switch to background */
            PutXGAInVGA() ;
        else if (al == 0x02) /* Switch to foreground */
            PutXGAInExtG() ;
    }
    if(int_2f_original_vector)
        _chain_intr( int_2f_original_vector ) ;
}

```

```

/* ***** */
/*
/* Function - signal_handler()
/*
/* Description - error handler
/*
/*
/*
/* ***** */

void signal_handler( void )
{
    exit(0);
}

/* ***** */
/*
/* Function - exit_handler()
/*
/* Description - Exit handler
/*
/*
/* ***** */

int exit_handler( void )
{
    /* reset interrupt vectors to original values */
    if ( int_10_original_vector )
        _dos_setvect( 0x10 , int_10_original_vector ) ;
    if ( int_2f_original_vector )
        _dos_setvect( 0x2f , int_2f_original_vector ) ;

    /* reset to VGA if originally in VGA mode */
    if ( ExtG_mode_set && XGAInVGA ) PutXGAInVGA ( ) ;
    printf ("Completed\n");
    return (0) ;
}

/* ***** */
/*
/* Function - co_pr_blit()
/*
/* Description - Use co-processor to blit to the display
/*
/*
/* ***** */

void co_pro_blit(void)
{
    unsigned char i , j , k;
    A1024x768 = TRUE ;

    CoProWriteByte( 0x11 , 0x00 ) ;
    CoProWriteByte( 0x12 , 0x01 ) ;

    /* set up VRAM base address */
    CoProWriteWord( 0x14 , VRAM_address_lo ) ;
    CoProWriteWord( 0x16 , VRAM_address_hi ) ;

    CoProWriteWord( 0x18 , A1024x768 ? 0x03ff : 0x02a8 ) ;
    CoProWriteWord( 0x1a , A1024x768 ? 0x02ff : 0x01e0 ) ;
    CoProWriteByte( 0x1c , (unsigned char)(VRAM_1Meg ? 0x03 : 0x02 ) ) ;
}

```

```

CoProWriteByte( 0x48 , 0x03 ) ;
CoProWriteByte( 0x49 , 0x05 ) ;
CoProWriteByte( 0x4a , 0x04 ) ;
CoProWriteWord( 0x50 , 0xff ) ;
CoProWriteWord( 0x54 , 0x7f ) ;
CoProWriteByte( 0x58 , 0x00 ) ;
CoProWriteByte( 0x5c , 0x00 ) ;
CoProWriteWord( 0x60 , A1024x768 ? 0x03ff : 0x02a8 ) ;
CoProWriteWord( 0x62 , A1024x768 ? 0x02ff : 0x01e0 ) ;
CoProWriteWord( 0x6c , 0x0000 ) ;
CoProWriteWord( 0x6e , 0x0000 ) ;
CoProWriteWord( 0x70 , 0x0000 ) ;
CoProWriteWord( 0x72 , 0x0000 ) ;
CoProWriteWord( 0x74 , 0x0000 ) ;
CoProWriteWord( 0x76 , 0x0000 ) ;
CoProWriteWord( 0x78 , 0x0000 ) ;
CoProWriteWord( 0x7a , 0x0000 ) ;
CoProWriteWord( 0x7c , 0x8000 ) ;
CoProWriteWord( 0x7e , 0x0811 ) ;

WaitForCoProReady() ;

CoProWriteWord( 0x60 , 0x0015 ) ;
CoProWriteWord( 0x62 , 0x02ff ) ;
CoProWriteWord( 0x60 , A1024x768 ? 0x0015 : 0x0010 ) ;
CoProWriteWord( 0x62 , A1024x768 ? 0x02ff : 0x02ff ) ;
for (j=0; j < 30 ; j++)
{
    k=0;
    for (i=1; i<= 34 ; i++)
    {
        if (k > 15 )
            k = 1 ;
        else
            k++ ;
        CoProWriteWord( 0x78 , (0x0030 * i) - 15 +j ) ;
        CoProWriteWord( 0x7a , 0x0000 ) ;
        CoProWriteByte( 0x58 , k ) ;
        CoProWriteByte( 0x7f , 0x08 ) ;
        WaitForCoProReady() ;
    }
}
CoProWriteWord( 0x60 , 0x03ff ) ;
CoProWriteWord( 0x62 , 0x02ff ) ;
CoProWriteWord( 0x70 , 0x0015 ) ;
CoProWriteWord( 0x72 , 0x0000 ) ;
CoProWriteWord( 0x78 , 0x0030 ) ;
CoProWriteWord( 0x7a , 0x0024 ) ;
CoProWriteByte( 0x7f , 0x28 ) ;
WaitForCoProReady() ;
}

```

```

/* ***** */
/*
/* Function - CoProWriteByte(offset , data)
/*
/* Description - Writes a data byte to the co-processor at the
/* supplied offset.
/*
/* ***** */

void CoProWriteByte(int offset , unsigned char ipdata)
{
    unsigned char far *cop ;
    unsigned long tmp ;

    tmp = ROS_address + 0x1c00 + offset + (cop_instance * 128 ) ;
    cop = (void far *)(((tmp & 0xffff0 )<12 ) + (tmp & 0x0f)) ;
    *cop= ipdata ;
}

void CoProPrintByte(int offset , char *string)
{
    unsigned char far *cop ;
    unsigned long tmp ;

    tmp = ROS_address + 0x1c00 + offset + (cop_instance * 128 ) ;
    cop = (void far *)(((tmp & 0xffff0 )<12 ) + (tmp & 0x0f)) ;
    printf(string);
    printf(" - %x\n",*cop);
}

/* ***** */
/*
/* Function - CoProWriteWord(offset , data)
/*
/* Description - Writes a data word to the co-processor at the
/* supplied offset.
/*
/* ***** */

void CoProWriteWord(int offset , unsigned int ipdata)
{
    unsigned int far *cop ;
    unsigned long tmp ;

    tmp = ROS_address + 0x1c00 + offset + (cop_instance * 128 ) ;
    cop = (void far *)(((tmp & 0xffff0 )<12 ) + (tmp & 0x0f)) ;
    *cop= ipdata ;
}

/* ***** */
/*
/* Function - WaitForCoProReady()
/*
/* Description - Waits until co-processor in ready state
/*
/* ***** */

void WaitForCoProReady(void)
{
    unsigned char far *cop ;
    unsigned long tmp ;
    long int count ;
}

```

```

tmp = ROS_address + 0x1c00 + 0x11 + (cop_instance * 128 ) ;
cop = (void far *)(((tmp & 0xffff0 )<<12 ) + (tmp & 0x0f)) ;
count = 0 ;
for(;;)
{
    if ( ((*cop & 0x80)==0) || (count > 20000)) break;
    count++;
}
}

/* ***** */
/*
/* Function - PutXGAInExtG()
/*
/* Description - Puts the XGA into Extended Graphics Mode
/*
/* ***** */

void PutXGAInExtG( void )
{
    int    i , res , palette_size ;

    outp( 0x03c3 , 0x01 ) ;
    ExtG_mode_set = TRUE ;
    /* 1meg VRAM res = 1 if 512K VRAM res = 2 */
    res = VRAM_1Meg ? 1 : 2 ;
    for ( i = 0 ; i < sizeof(nm_data) ; i = i + 4 )
    {
        if (nm_data[i+3])
            outpw( IoRegBase + nm_data[i] ,
                (((int)nm_data[i+res]) << 8) + (unsigned)nm_data[i+3]
                ) ;
        else
            outp( IoRegBase + nm_data[i] , (int)nm_data[i+res] ) ;
    }

    outpw( IoRegBase + 0x0a , 0x0066 ) ;
    outpw( IoRegBase + 0x0a , 0x0060 ) ;
    outpw( IoRegBase + 0x0a , 0x0061 ) ;

    palette_size = sizeof(colour_default_palette) ;
    for ( i=0 ; i <= palette_size ; i++ )
    {
        /* select palette data register */
        outp( IoRegBase + INDEX_SELECT , 0x65 ) ;
        outp( IoRegBase + 0x0b , (int)colour_default_palette[i] ) ;
    }
}

```

```

/* ***** */
/*
/* Function - PutXGAInVGA()
/*
/* Description - puts the XGA into VGA mode
/*
/* ***** */

void PutXGAInVGA( void )
{
    int i ;
    vram_address = (unsigned char far *)0xA0000000;
    /* clear 1st 256k of vram */
    for (i = 0 ; i < 4 ; i++)
    {
        outp(IoRegBase + 8 , i);
        ptr = vram_address ;
        memset(ptr , 0 , 0x8000); /* set 1st 32K of 64K aperture*/
        ptr = vram_address + 0x8000 ;
        memset(ptr , 0 , 0x8000); /* set 2nd 32K of 64K aperture*/
    }

    for (i = 0 ; i < sizeof(vga_data) ; i = i + 3 )
    {
        if (vga_data[i+2])
            outpw( IoRegBase + vga_data[i] ,
                (((unsigned)vga_data[i+1]) << 8)
                + (unsigned)vga_data[i+2]
                ) ;
        else
            outp( IoRegBase + vga_data[i] , (int)vga_data[i+1] );
    }

    outp( 0x03c3 , 0x01 );

    /* select scan lines for alphanumeric modes */
    inregs.h.ah = 0x12 ;
    inregs.h.al = 0x02 ; /* 400 scan lines */
    inregs.h.bl = 0x30 ;
    int86( 0x10 , &inregs , &outregs ) ;
    inregs.h.ah = 0x00 ;
    inregs.h.al = 0x03 ;
    int86( 0x10 , &inregs , &outregs ) ;
}

```

Assembler Subroutines

```

;*****
;**
;** Function _return_ax() **
;** **
;** Description unwinds the stack after a call to an interrupt **
;** routine to obtain contents of ax register, the **
;** value of which is returned. The stack is restored **
;** prior to the return. **
;** **
;*****

.286c
.MODEL SMALL
.DATA
return_segment_address dw ?
return_offset_address dw ?
ret_data_seg dw ?
ret_bp dw ?

.CODE
PUBLIC _return_ax
_return_ax PROC FAR
    mov     ret_bp , bp
    pop     bx
    pop     es
    mov     return_segment_address , es
    mov     return_offset_address , bx
    add     sp,+2
    pop     es
    pop     bx ; data seg
    mov     ret_data_seg , bx
    popa
    pusha
    mov     bx,ret_data_seg
    push   bx ; data seg
    push   es
    sub     sp,+2
    mov     es,return_segment_address
    mov     bx,return_offset_address
    push   es
    push   bx
    mov     bp , ret_bp
    ret
_return_ax ENDP

END

```

```
*****
;
; **
; ** Function return_7f
; **
; ** Description unwinds the stack after a call to an interrupt
; ** routine to obtain contents of ax register, the
; ** value of which is returned. The stack is restored
; ** prior to the return.
; **
; **
; *****

.286c
.MODEL SMALL
.DATA
return_segment_address dw ?
return_offset_address dw ?
ret_data_seg dw ?
ret_bp dw ?

.CODE
PUBLIC _return_7f
_return_7f PROC FAR
    mov ret_bp , bp
    pop bx
    pop es
    mov return_segment_address , es
    mov return_offset_address , bx
    add sp,+2
    pop es
    pop bx ; data seg
    mov ret_data_seg , bx
    popa
    mov ah,7fh
    pusha
    mov bx,ret_data_seg
    push bx ; data seg
    push es
    sub sp,+2
    mov es,return_segment_address
    mov bx,return_offset_address
    push es
    push bx
    mov bp , ret_bp
    ret
_return_7f ENDP
END
```


15.2 Putting the XGA Subsystem into 132 Column Text Mode

15.2.1 Pseudo Code

Main Program

- Locate XGA subsystem with attached monitor in VGA mode
- If none, display error message and return.
- Chain Int 10h Video handler
- Chain Int 21h DOS Function handler
- Chain Int 23h Ctrl Break Exit Address
- Chain Int 2Fh Screen Switch Notification handler
- Put XGA into 132 column text mode (see Section 11.1.3)
- Display simple text
- Exit

Int 10 Handler

- Examine value of (Ah)

00h Set Mode

- Put XGA subsystem in normal VGA mode (see Section 11.1.2)
- Chain on to saved Int 10h Video Interrupt handler.

Any other value

- Interrupt return (IRET)

Int 21h DOS Function handler

- Examine value of (Ah)

4Ch Program Terminate

- Put XGA subsystem in normal VGA mode
- UnChain and restore original Int 10h Video handler
- UnChain and restore original Int 21h DOS function handler

- Chain on to saved Int 21h handler

Int 23h Ctrl Break Exit Address

- Chain on to saved Int 23h handler, using a method that will ensure return of control via this function handler.
- On return from chained handler, examine Carry Flag (CF). If set
 - Put XGA subsystem in normal VGA mode
 - UnChain and restore original Int 10h Video handler
 - UnChain and restore original Int 21h DOS function handler
- Interrupt return (IRET)

Int 2Fh Screen Switch Notification Handler

- Examine value of (Ah)

40h Screen Switch Notification

- Examine value of (AL)
 - 01h Impending switch to background.**
 - Put XGA subsystem in normal VGA mode
 - Chain on to saved Int 2Fh handler (if any)
 - 02h Impending switch to foreground**
 - Put XGA subsystem in 132 column text mode
 - Semaphore “re-draw required” to application
 - Chain on to saved Int 2Fh handler (if any)

Any other value

- Chain on to saved Int 2F Interrupt vector (if any)

15.2.2 Code Example

Main C Program

```

/* ***** */
/*
/*
/* Program s_132n
/*
/* Description This program is sample code to illustrate entry
/* to 132 mode and back to VGA mode upon program
/* termination.
/*
/* ***** */

#define POS 0xc4
#define POS_GET_BASE_ADDRESS 0X00
#define FALSE 0x00
#define TRUE 0x01
#define EMM_INT 0x67
#define DEVICE_NAME_LENGTH 0x08
#define MAX_SLOTS 0x09
#define XGA_ID_UPPER 0x8fdb
#define XGA_ID_LOWER 0x8fd8
#define INDEX_SELECT 0x0a
#define INDEX_DATA 0x0b
#define MONITOR_ID 0x52

#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <signal.h>
#include <malloc.h>
#include <memory.h>
#include <stdlib.h>
#include <conio.h>

```

```
typedef struct                                /* POS Record */
{
  unsigned int pos_id ;
  char pos_byte1 ;
  char pos_byte2 ;
  char pos_byte3 ;
  char pos_byte4 ;
} POS_REC ;

union REGS inregs , outregs ;
struct SREGS segregs ;
unsigned int pos_base_address , IoRegBase , slot_number ;

POS_REC pos_record ;

unsigned char vga_data[] = { 0x01 , 0x00 , 0x00 ,
                             0x04 , 0x00 , 0x00 ,
                             0x05 , 0xff , 0x00 ,
                             0x0a , 0xff , 0x64 ,
                             0x0a , 0x15 , 0x50 ,
                             0x0a , 0x14 , 0x50 ,
                             0x0a , 0x00 , 0x51 ,
                             0x0a , 0x04 , 0x54 ,
                             0x0a , 0x7f , 0x70 ,
                             0x0a , 0x20 , 0x2a ,
                             0x00 , 0x01 , 0x00
                           } ;

char RedrawRequired ;
char XGAFound ;
char XGAInVGA ;
void far *int_10_original_vector ;
void far *int_2f_original_vector ;

interrupt far int_10( void ) ;
interrupt far int_2f( void ) ;
int far return_ax( void ) ;
int far return_7f( void ) ;
void PutXGAInVGA(void) ;
void PutXGAIn132(void);
int exit_handler( void ) ;
void signal_handler( void ) ;
void DisplayText( void ) ;
```

```

/* ***** */
/* Function main() */
/* */
/* Description This is the program entry point. */
/* ***** */

void main( void )
{
    int      index ;
    unsigned int  ipdata ;
    unsigned char far *vram_address ;
    unsigned char ip_byte ;

    atexit( exit_handler ) ;
    signal( SIGINT , signal_handler ) ;
    signal( SIGFPE , signal_handler ) ;
    signal( SIGABRT , signal_handler ) ;

    int_10_original_vector = 0 ;
    int_2f_original_vector = 0 ;

    inregs.h.ah = POS ;          /* get base POS address */
    inregs.h.al = POS_GET_BASE_ADDRESS ; /* from system services */
    int86( 0x15 , &inregs , &outregs ) ;

    pos_base_address = outregs.x.dx ;
    XGAFound = FALSE ;
    if( outregs.x.cflag )          /* carry flag set means */
    {                               /* not a microchannel */
        printf( "No XGA Installed\n" ) ; /* machine */
    }
    else
    {
        XGAFound = FALSE ;
        XGAIInVGA = FALSE ;

        for (slot_number = 0 ; slot_number <= MAX_SLOTS ; slot_number++)
        {
            _disable();          /* Disable interrupts */
            if (slot_number == 0)
            { /* Look at the planar for XGA */
                outp( 0X094 , 0x0df ) ; /* Enable planar for setup */
            }
            else
            { /* Look in the slots for XGA */
                inregs.x.ax = 0xc401 ;
                inregs.x.bx = slot_number ;
                int86( 0x15 , &inregs , &outregs ) ;
                /* enable slot for update */
            }

            /* Get pos record for the slot */
            pos_record.pos_id = inpw( pos_base_address ) ;
            pos_record.pos_byte1 = (char)inp( pos_base_address + 2 ) ;
            pos_record.pos_byte2 = (char)inp( pos_base_address + 3 ) ;
            pos_record.pos_byte3 = (char)inp( pos_base_address + 4 ) ;
            pos_record.pos_byte4 = (char)inp( pos_base_address + 5 ) ;
            IoRegBase = (( pos_record.pos_byte1 & 0x0e ) << 3 ) + 0x2100 ;
        }
    }
}

```

```

if(slot_number == 0)
{
    outp(0X094 , 0x0ff) ; /* Enable planar for normal mode */
}
else
{
    inregs.x.ax = 0xc402 ;
    inregs.x.bx = slot_number ;
    int86( 0x15 , &inregs , &outregs ) ;
    /* enable slot normal mode */
}
_enable(); /* Enable interrupts */
/* Check for a valid XGA POS id */
if ( pos_record.pos_id >= XGA_ID_LOWER &&
    pos_record.pos_id <= XGA_ID_UPPER )
{
    /* XGA found in slot */

    /* Look to see if monitor connected to XGA */
    outp( IoRegBase + INDEX_SELECT , MONITOR_ID );
    if ( ( inp(IoRegBase + INDEX_DATA) & 0x0f) != 0x0f )
    {
        /* Monitor connected to XGA */
        XGAFound = TRUE ;
        /* Determine if XGA in VGA */
        ipdata = inp( IoRegBase ) ;
        if( ipdata & 0x01 )
        {
            XGAInVGA = TRUE ;
            break;
        }
    }
}
}
if (XGAInVGA == FALSE )
{
    printf( "XGA in VGA with attached monitor - not found\n");
}
else
{
    RedrawRequired = FALSE ;
    /* Chain Int 10H */
    int_10_original_vector = _dos_getvect( 0x10 ) ;
    _dos_setvect( 0x10 , int_10 ) ;

    /* Chain Int 2fH */
    int_2f_original_vector = _dos_getvect( 0x2f ) ;
    _dos_setvect( 0x2f , int_2f ) ;

    PutXGAIn132();

    DisplayText();
}
}
}

```

```

/* ***** */
/*
/* Function - interrupt routine int_10()
/*
/* Description - internal INT 10h interrupt handler
/*
/*
/* ***** */

interrupt far int_10( void )
{
    int ax ;
    ax = return_ax() >> 8 ;
    if( ax == 0 )
    {
        /* set xga to vga mode */
        PutXGAInVGA();
        _chain_intr( int_10_original_vector ) ;
    }
    else if ( ax == 0x0f )
    {
        return_7f() ;
    }
    else
    {
        _chain_intr( int_10_original_vector ) ;
    }
}

/* ***** */
/*
/* Function - Interrupt routine int_2f()
/*
/* Description - Internal INT 2fh interrupt handler
/*
/*
/* ***** */

interrupt far int_2f( void )
{
    unsigned int ax ;
    unsigned char ah ;
    unsigned char al ;

    ax = return_ax() ;
    ah = (unsigned char)(ax >> 8) ;
    al = (unsigned char)(ax & 0x0f);
    if( ah == 0x40 )
    { /* Screen switch notification received */
        if (al == 0x01) /* Switch to background */
            PutXGAInVGA();
        else if (al == 0x02) /* Switch to foreground */
        {
            RedrawRequired = FALSE ;
            PutXGAIn132();
        }
    }
    if(int_2f_original_vector) _chain_intr( int_2f_original_vector ) ;
}

```

```

/* ***** */
/*
/* Function - signal_handler()
/* Description - error handler
/*
/*
/* ***** */

void signal_handler( void )
{
    exit(0);
}

/* ***** */
/*
/* Function - exit_handler()
/* Description - Exit handler
/*
/*
/* ***** */

int exit_handler( void )
{
    /* reset interrupt vectors to original values */
    if ( int_10_original_vector )
        _dos_setvect( 0x10 , int_10_original_vector ) ;
    if ( int_2f_original_vector )
        _dos_setvect( 0x2f , int_2f_original_vector ) ;

    if(XGAIInVGA)PutXGAIInVGA ( );    return (0) ;
}

/* ***** */
/*
/* Function - PutXGAIIn132()
/* Description - Puts the XGA into 132 mode
*/
/*
/* ***** */

void PutXGAIIn132( void )
{
    int ipdata ;
    int far *bios ;

    outw ( IoRegBase + 0x0a , 0x1550 ) ;

    outw ( IoRegBase + 0x0a , 0x1450 ) ;

    outw ( IoRegBase + 0x0a , 0x0454 ) ;

    /* select scan lines for alphanumeric modes */
    inregs.h.ah = 0x12 ;
    inregs.h.al = 0x02 ;    /* 400 scan lines */
    inregs.h.bl = 0x30 ;
    int86( 0x10 , &inregs , &outregs ) ;
}

```

```
/* set vga mode */
inregs.h.ah = 0x00 ;
inregs.h.al = 0x03 ;
int86( 0x10 , &inregs , &outregs ) ;

outp ( IoRegBase + 0x0a , 0x50 ) ;
ipdata = inp ( IoRegBase + 0x0b ) ;
ipdata &lor.= 0x01 ;
outp ( IoRegBase + 0x0b , ipdata) ;

    outp ( IoRegBase + 0x0a , 0x50 ) ;
ipdata = inp ( IoRegBase + 0x0b ) ;
ipdata &= 0xFD ;
outp ( IoRegBase + 0x0b , ipdata) ;

    outp ( IoRegBase + 0x0a , 0x50 ) ;
ipdata = inp ( IoRegBase + 0x0b ) ;
ipdata &= 0xFC ;
outp ( IoRegBase + 0x0b , ipdata) ;

outp ( IoRegBase , 0x03 ) ;

outpw( IoRegBase + 0x0a , 0x0154 ) ;

outpw( IoRegBase + 0x0a , 0x8070 ) ;

outp ( IoRegBase + 0x0a , 0x50 ) ;
ipdata = inp ( IoRegBase + 0x0b ) ;
ipdata &= 0xef ;
outp ( IoRegBase + 0x0b , ipdata) ;

outp( 0x03d4 , 0x11 ) ;
ipdata = inp( 0x03d5 ) ;
ipdata &= 0x7f ;
outp( 0x03d5 , ipdata ) ;

outp( 0x03d4 , 0x00 ) ;
outp( 0x03d5 , 0xa4 ) ;

outp( 0x03d4 , 0x01 ) ;
outp( 0x03d5 , 0x83 ) ;

outp( 0x03d4 , 0x02 ) ;
outp( 0x03d5 , 0x84 ) ;

outp( 0x03d4 , 0x03 ) ;
outp( 0x03d5 , 0x83 ) ;

outp( 0x03d4 , 0x04 ) ;
outp( 0x03d5 , 0x90 ) ;

outp( 0x03d4 , 0x05 ) ;
outp( 0x03d5 , 0x80 ) ;

outpw( IoRegBase + 0x0a , 0xa31a ) ;
outpw( IoRegBase + 0x0a , 0x001b ) ;

outp( 0x03d4 , 0x13 ) ;
```



```

    outp( 0x03d5 , 0x42 );

    outp( 0x03d4 , 0x11 );
    ipdata = inp( 0x03d5 );
    ipdata &lor.= 0x80 ;
    outp( 0x03d5 , ipdata );

    outp ( IoRegBase + 0x0a , 0x50 ) ;
    ipdata = inp ( IoRegBase + 0x0b ) ;
    ipdata &lor.= 0x03 ;
    outp ( IoRegBase + 0x0b , ipdata) ;

    outp( 0x03c4 , 0x01 ) ;
    ipdata = inp( 0x03c5 );
    ipdata &lor.= 0x01 ;
    outp( 0x03c5 , ipdata );

    ipdata = inp( 0x03da ) ;

    outp( 0x03c0 , 0x13 );

    outp( 0x03c0 , 0x00 );

    outp( 0x03c0 , 0x20 );

    bios = (int far *)0x40004a ;
    *bios = 0x84 ;          /* tell BIOS we have 132 columns */
}

/* ***** */
/*
/* Function - PutXGAInVGA()
/*
/* Description - puts the XGA into VGA mode
/*
/*
/* ***** */

void PutXGAInVGA( void )
{
    int i ;
    vram_address = (unsigned char far *)0xA0000000;

    /* clear 1st 256k of vram */
    for (i = 0 ; i < 4 ; i++)
    {
        outp(IoRegBase + 8 , i);
        ptr = vram_address ;
        memset(ptr , 0 , 0x8000); /* set 1st 32K of 64K aperture*/
        ptr = vram_address + 0x8000 ;
        memset(ptr , 0 , 0x8000); /* set 2nd 32K of 64K aperture*/
    }

    for (i = 0 ; i < sizeof(vga_data) ; i = i + 3 )
    {
        if (vga_data[i+2])
            outpw( IoRegBase + vga_data[i] ,
                (((unsigned)vga_data[i+1]) << 8)
                + (unsigned)vga_data[i+2]
                ) ;
    }
}

```

```
        else
            outp( IoRegBase + vga_data[i] , (int)vga_data[i+1] );
    }

    outp( 0x03c3 , 0x01 );

    /* select scan lines for alphanumeric modes */
    inregs.h.ah = 0x12 ;
    inregs.h.al = 0x02 ;    /* 400 scan lines */
    inregs.h.bl = 0x30 ;
    int86( 0x10 , &inregs , &outregs ) ;

    /* set vga mode */
    inregs.h.ah = 0x00 ;
    inregs.h.al = 0x03 ;
    int86( 0x10 , &inregs , &outregs ) ;
}

void DisplayText( void )
{
    int j ;
    char ch ;

    for (j=1 ; j < 24 ; j++)
    { /* sample code to fill screen with 24 x 132 chars */
        printf("This line .....") ;
        printf(".....") ;
        printf("..... is 132 chars long" ) ;
    }
    printf("Press Enter to continue.....");
    ch = getchar();
}
```

Assembler Subroutines

See Section 15.1.