

VGA6800 Programmers Manual

**A VGA Video Controller in
Field Programmable Gate Array
designed for the M6800 8-bit CPU**

Document Version 1

Author and Designer Kevin Bertram

Project Number F0012

Copyright © 2002-2003 Kevin Bertram
(www.cate.com.au)

Table of Contents

Introduction.....	3
Hardware Drawing Acceleration.....	5
Screen Mapping.....	6
Video Drawing Control Registers.....	7
Video Page Register.....	8
Line Drawing.....	9
Managing Pixels.....	10
Writing a pixel.....	10
Reading a pixel from VRAM.....	10
Pixel read using Line Read mode.....	10
Pixel read using Blit operation to Program Memory.....	11
Using the Blitter.....	12
Block Write using a Blit to VRAM.....	12
Block Read using Blit to Program memory.....	13
CPU Memory Map.....	14
Non Displayable Video RAM.....	14
FPGA Register Map.....	16
UART Registers.....	18
UART Receive Status Register Description.....	18
UART Transmit Status Register Description.....	19
Interrupt Controller Register.....	20
LED display Registers.....	22
Appendix A.....	24
VGA Monitors.....	24
VGA 9 pin.....	24
VGA 15-pin.....	24
Appendix B.....	26
Revision Levels.....	26
Glossary of terms.....	27

Introduction

This project is an effort to attach a Video Graphics Adapter (VGA) to an old, and now obsolete, 8-bit micro controller once manufactured by Motorola. The microprocessor, used in the prototype, is an M6800 8-bit CPU clocked at 1Mhz. It became available August 1974 as one of the first microprocessors used to build personal computers. They were obsoleted by Motorola from its product line around 1989 (approximately). Making it an ancient device and a good candidate for the task.

The VGA controller, to be attached to the M6800, has a displayable pixel field of 640 pixels across and 480 lines down. Each pixel is 4-bits wide with two pixels contained in each byte of video RAM. The screen refresh rate is 60Hz, which can be otherwise stated as one screen image redrawn every 16.6 milliseconds.

So why bother? The CPU is obsolete, slow by todays standard, 8-bit data bus, and has a tiny 16-bit address bus. The latter being far to small to address all 153,600 bytes needed for a single video page. In fact an entire video page, including off screen area, is 262,144 bytes. Where would the program RAM and ROM fit?

These problems were all part of the challenge. It was hoped that with such restrictions in place, complacency would be avoided hopefully resulting in an efficient, fast, and capable VGA controller.

After a lot of thinking, and development work the result is a VGA controller that can not only be driven by any small 8-bit micro controller, but draw at very respectable speeds. Memory range addressability is not an issue, CPU speed is a minor consideration, and it is portable from one type of CPU to another. To add it is a compilable FPGA symbol that can be directly placed into any Altera FPGA schematic.

To illustrate its graphics drawing speed: the prototype M6800 could clear an entire video page, draw a cross hatch grid, colours bars and 16*8 fixed font face to the screen in about half a second. Of course there could have been special work done to the software to speed this up in areas such as font drawing, but just for the exercise that was not done. A 1MHz M6800, that shared time between two processes (multitasking) controlled a VGA video page as if it were a much much faster, more efficient CPU.

All this is done by employing a hardware style drawing engine. The M6800 CPU, by itself, could not achieve these high drawing speeds and in practice the M6800 had taken approximately 20 seconds to draw the same screen image pixel by pixel. Hardware drawing acceleration is not a new concept as it is used in every video card developed for the Home PC. So this one allows the small 8-bit microprocessor a way to speed up its drawing efforts of the entire VGA pixel map..

Therefore, this graphics drawing accelerator is designed to hook up to small 8-bit microprocessors and takes care of all the repetitive work of drawing lines, and copying images from program memory to video memory. As well as addressing the large memory space needed by a VGA video page.

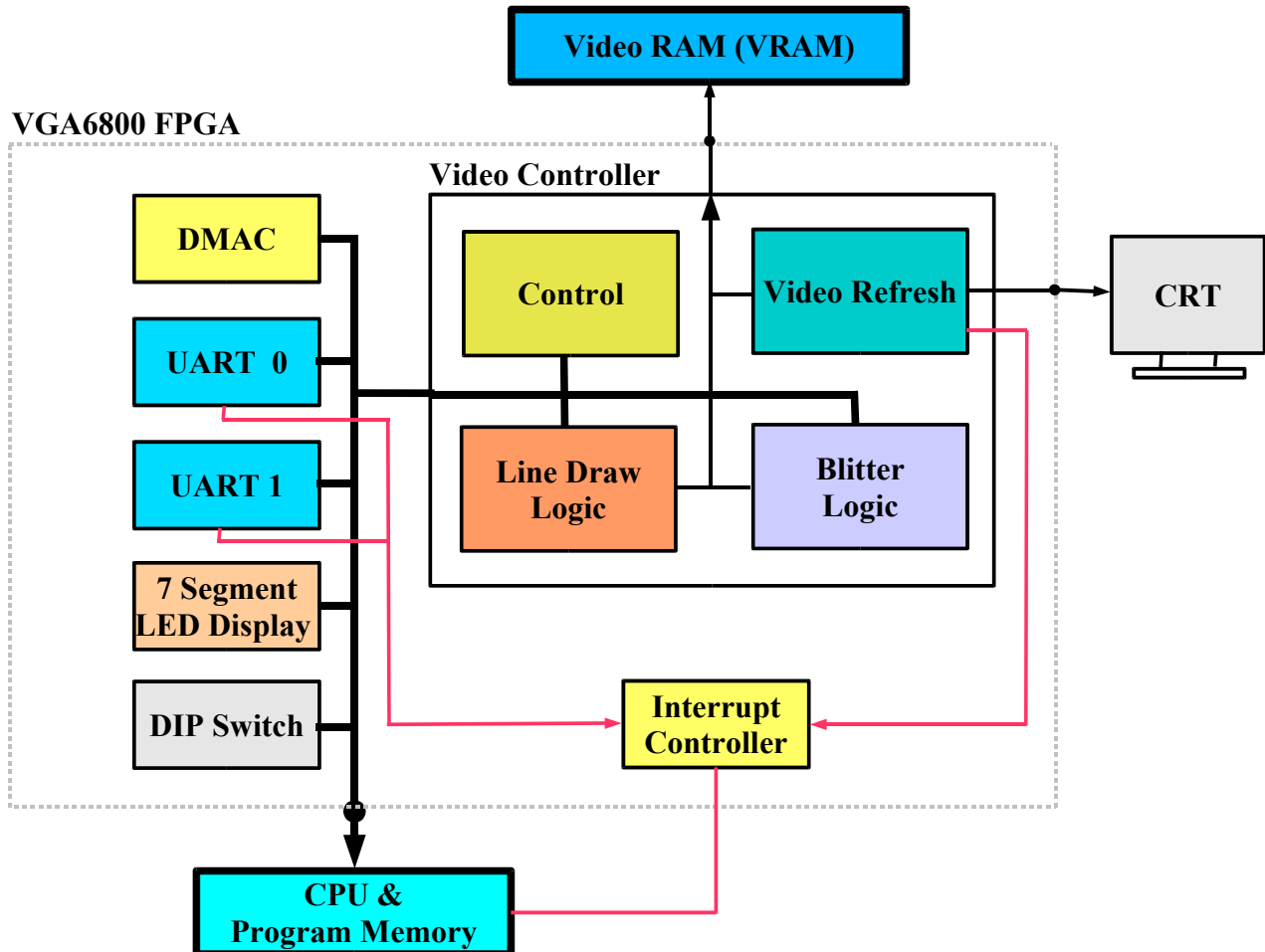
The other side of the project was to see just how fast the old M6800 CPU could drive a VGA screen, and more importantly, could it be done. It was an exercise that, I believe, has not been done before, and was a lot of fun to do all while walking down memory lane.

The rest of this manual is a testament to that effort and the results obtained.

VGA6800 Programmers Manual

The VGA controller is quite modular being flexible enough to be interfaced to many other types of CPU's, not just the M6800 and for that reason adds a usefulness that can be reused anywhere a VGA driver is needed.

This illustration provides a reasonable idea of what is actually integrated into the FPGA software.



As can be seen there are two sections to it. The first logic block is the VGA video controller and hardware drawing acceleration. It forms a function within the whole FPGA.

Then there are the input output peripherals integrated so that the CPU, and software, have methods to communicate to the outside world. It is much easier to build these into a single FPGA than to design them into the hardware externally.

Hardware Drawing Acceleration

There are two types of accelerated drawing operations that form the basis of the whole accelerator.

1. Pixel manipulation. This is the act of reading, or writing, a single pixel to the Video RAM.
2. Pixel Copy. This is the act of moving a single pixel between Video RAM and Program RAM.

Using these two fundamental components larger and more complex operations have been built on top. The existing list of drawing primitives managed, and accelerated by the hardware is:

1. 2D line drawing. A line can be one pixel wide and one high, upto 640 pixels wide and 320 pixels high. The dimensions of the line is controlled by the program. A screen clearing operation, for example, is essentially to draw a line 640 pixels wide by 480 pixel high in the colour Black (0x00).
2. 2D diagonal line drawing. Works the same as (1) except the X and Y gradients are loaded too. The CPU's software has to calculate what *mx* and *my* are before the line is drawn and then let it draw.
3. Copy a 2D image from program memory (linear addressable range) to video RAM (X,Y address range).
4. Copy a 2D image from video RAM (X,Y address range) to program memory (linear addressable range)

Program memory is defined as a linear addressable region, meaning the next byte is at the current address plus one. And continues that way until the end is reached where it will wrap back to zero. This is a typical arrangement for most CPU memory space.

Video RAM is also linear however, it is calculated from, and accessed by, an X and Y co-ordinate value loaded into an X and Y register. From the software's perspective all video RAM appears as a cartesian co-ordinate grid where the origin is at the top left corner of the screen. That makes the screen upside down to the accepted method of video screen mapping (origin at lower left corner) but can easily be adjusted in software.

Because Video RAM is addressed through an X,Y cartesian co-ordinate system requires only two 16-bit registers to be used to represent any place on the video pixel grid, while a third, the drawing page number, gives access to any video page in the entire video RAM. This eliminates the need for the CPU to have, or use, an 18-bit address register ($2^{18} = 256K$ addressable bytes) to get at any pixel on the screen. The software is also relieved of the effort to construct the final video RAM address from internal X and Y co-ordinates. All conversions are performed, very quickly, in hardware.

By directly accessing the video page using 16-bit X and Y co-ordinate registers eliminates the need for a large 18-bit address register. This is more than acceptable for small 8-bit microprocessors and micro-controllers that can be attached to a VGA screen. Most can manipulate 16-bit numbers very easily but could find it difficult for anything larger.

Screen Mapping

A video screen is essentially a block of 256KB of RAM constantly read by a video controller at a fixed rate. The data read from video RAM is considered pixel colour information and is there for transferred to the RED, GREEN, and BLUE colour guns of the video monitor, or display device.

Colour translation used by this video controller is "TTL Colour" where each colour can be turned on, or off only. That results in eight different displayable colours. Only six bits are needed to control the colour guns allowing two pixels to be used as flashing attribute controls.

Bit-7	Bit-6	Bit-5	Bit-4	Bit-3	Bit-2	Bit-1	Bit-0
P2	P2	P2	P2	P1	P1	P1	P1
Flash	Blue	Green	Red	Flash	Blue	Green	Red

Bit allocation in a single Video RAM byte

A video monitor starts scanning its image from top left of the screen across to the right edge and sequentially draws the next lower line until the bottom of the screen is reached. It dictates then that the first pixel from video RAM is situated at the top left corner of the screen.

Each pixel read from video RAM and displayed to the screen comes from a incrementing address starting at 0x00000 and finishing at 0x1E000. Each line on the screen maps to video RAM at an aligned address on 512 byte boundaries. Not all of the memory in a 512 byte block is used either, 128 bytes is left unused by the video controller simply because 320 bytes is all that is required to draw a single line. Video line = 640 pixels = (640/2) bytes = 320 bytes.

0x00000	X=0	Y=0	X=639	X=640	Offscreen RAM	X=1023
0x00200	X=0	Y=1	X=639	X=640	Offscreen RAM	X=1023
0x00400	X=0	Y=2	X=639	X=640	Offscreen RAM	X=1023
0x1DC00	X=0	Y=478	X=639	X=640	Offscreen RAM	X=1023
0x1DE00	X=0	Y=479	X=639	X=640	Offscreen RAM	X=1023

Video Drawing Control Registers

These registers relate to graphic manipulation operations. Depending on the work to be done defines which registers need to be loaded, or read.

<i>Offset</i>	<i>Size</i>	<i>Mnemonic</i>	<i>Description</i>
0x00	8-bit	PIXELW	Pixel colour to write when drawing lines to the video RAM.
0x01	8-bit	PIXELR	Actual value of pixel read from video RAM.
0x10	16-bit	X	X co-ordinate for drawing operation. ^{Note 1}
0x12	16-bit	Y	Y co-ordinate for drawing operation. ^{Note 1}
0x14	16-bit	XN	X iteration count. This represents the number of pixel operations in the X direction. ^{Note 1}
0x16	8-bit	VPAGE	Video page register. The lower 4-bits represents the current display page, while the upper four bits represents the current drawing page. They do not have to be the same.
0x17	8-bit	DRAWCTL	Controls the type of drawing operation and acts as a completion status register. All bits are self clearing once the operations has completed.
0x18	16-bit	YN	Y iteration count. This represents the number of pixel operations in the Y direction. ^{Note 1}
0x1A	8-bit	MX	Iteration count for diagonal line drawing in the X direction. (TBA) ^{Note 2}
0x1B	8-bit	MY	Iteration count for diagonal line drawing in the Y direction. (TBA) ^{Note 2}
0x40	16-bit	DMADRS	Direct Memory Access Address. This register holds the starting location in program memory where the Blitter is to read, or write, data being transfered between Video RAM and program memory. ^{Note 1}

Note 1: 16-bit registers are arranged to suit a *Big Endian* microprocessor. This means the high byte of a 16-bit number is stored into the first (lowest) address and the low byte is stored in the next (highest) address.

Note 2: These registers, and the functions they imply, do not exist in version 1 hardware. Read the *HWVERSION* register to determine the level of revision.

The *DRAWCTL* register is the only bit definable control register in the set. It controls all activity for this register set and therefore holds the status of any graphic operation for this register set.

DRAWCTL			
<i>Bit</i>	<i>Default state</i>	<i>Active State</i>	<i>Operation</i>
Bit-0	0	1	Instructs hardware to write to VRAM.
Bit-1	0	1	Instructs hardware to read from VRAM
Bit-2	0	X	Reserved.
Bit-3	0	X	Reserved.

DRAWCTL			
Bit-4		0/1	Mode of Read, or Write, operation:
	0		When set to '0' = Line mode. Draws or reads lines.
			When set to '1' = Blitter mode.
Bit-5	0	X	Reserved.
Bit-6	0	X	Reserved.
Bit-7	0	X	Reserved.

Video Page Register

The Video Controller can access, and use, up to eight separate blocks of memory sufficient enough to hold an entire VGA pixel map and will be referred to as *video pages*. When translated into numbers, that would be a total of 256KB * 8 = 2048KB (2MB) of addressable video RAM.

A register is provided to allow control of the chosen video page to read from , or write to, by the CPU and that video page which is currently being displayed by the video refresh logic to the CRT.

This type of control register is commonly used in graphics applications for “Double Buffering”, or “Tripple Buffering” video pages. In the VGA6800's case it is “Octet Buffering.” It is a large number of video pages and applications may not effectively use all of them.

Its purpose is to show one video page to the viewer while software is drawing another for viewing. When the new screen image has been drawn to an off-screen buffer it is “flipped” into view allowing the viewer to see it. The overall effect is a smooth jitter less animation effect. It also has a side effect of giving the viewer an impression of a very fast drawing speed.

The *VPAGE* register is a 8-bit read/write latch split into two 4-bit counts. This is how it is laid out:

VPAGE	
Bit 0-3	Current displayed video page.
Bit 4-7	Current drawing (off-screen) page.

How would a program use it? Normally at start up the program resets the current display page pointer to video page 0. If the application does not care about the user seeing it draw to the displayed video page then the current drawing page is set to 0 as well. *VPAGE* = 0x00.

However, if the application does not want the user to see the new screen image until it is completely drawn then it would use the next available video page to direct drawing to: *VPAGE* = 0x10. All graphic drawing operations will be directed to video page 1 while the current displayed video page is 0.

When the new screen has been drawn software will swap the current drawing page with the current display page: *VPAGE* = 0x01. Now the user can see the second video page (1) while the first video page (0) is set as the current drawing page.

When the current drawing page (0) has been drawn with a new image they are again swapped over: *VPAGE* = 0x10. And on it goes until the power is removed or the program is stopped.

Another trick would be to prepare all eight video pages with complex animations. The software could time the interval between flipping to the next video page thus producing a very controlled smooth animation. The overhead in maintaining the preloaded animation would be very low while effect is very high.

Line Drawing

Line drawing can be done by software or hardware. Software drawing is not covered here, only the implemented hardware drawing modes are described.

here are three modes of a line draw operation that must be considered and handled:

1. Vertical Line.
2. Horizontal Line.
3. Diagonal Line.

FPGA hardware version 1, as read from the hardware version register, two of the three drawing modes are supported. That is: Vertical and Horizontal hardware assisted line drawing. The third mode, diagonal line, is still to be implemented meaning software will have to take care of these types of lines when needed.

It will be the task of the software to determine how it is going to draw a line. Since vertical and horizontal lines are supported by this hardware the the decision is not very difficult. In fact there is really no need to be concerned about the direction of the line since its dimensions actually form a line that is vertical or horizontal.

The method to drawing a line is achieved by simply loading the starting X, Y co-ordinates and the XN and YN pixel count, followed up by setting the “Draw” bit in the Line drawing control register: *DRAWCTL*.

For example: To draw a horizontal white line starting at (0,56) of 210 pixels long and 2 pixels high these registers need to be loaded:

<i>PIXELW</i>	0x77	Set upper and lower pixel to White: = 7.
<i>X</i>	0x0000	Loads the starting X co-ordinate (0) of the line.
<i>Y</i>	0x0038	Loads the starting Y co-ordinate (56) of the line.
<i>XN</i>	0x00D2	Loads the pixel count (210 pixels) in the X direction.
<i>YN</i>	0x0002	Loads the pixel count (2 lines) in the Y direction.
<i>DRAWCTL</i>	0x01	Loads the drawing mode (write) and starts the hardware drawing. A line has been drawn when this bit is automatically cleared.

Drawing a line is quick and is interleaved between video refresh accesses to the video RAM. Therefore there will not be 100% of VRAM bus availability to line drawing and thus throughput is reduced. But that is not a serious problem since the hardware can draw eighty pixels in the same time the M6800 CPU can write a single bit to an FPGA control register.

Roughly translated, that means the CPU will never be able to overwhelm the drawing hardware or Video RAM bus.

Managing Pixels

Writing a pixel

Writing a single pixel is exactly the same as a line. The main controlling difference is a defined length and height of 1 pixel. Drawing a single pixel is a very heavy operation to setup compared to a line and any advantage gained by hardware drawn lines is lost when trying to do hardware drawn pixels.

Even so, if it is needed the operation can be done. Here is an example of a single Red pixel drawn to the co-ordinates (500,48):

<i>PIXELW</i>	0x01	Set upper and lower pixel to Red: = 1.
<i>X</i>	0x01F4	Loads the starting X co-ordinate of the line.
<i>Y</i>	0x0030	Loads the starting Y co-ordinate of the line.
<i>XN</i>	0x0002	Loads the pixel count in the X direction. ^{Note 1}
<i>YN</i>	0x0001	Loads the pixel count in the Y direction.
<i>DRAWCTL</i>	0x01	Loads the drawing mode (write) and starts the hardware drawing. A pixel has been drawn when this bit is automatically cleared.

Note 1: The *XN* register is loaded with '2' because the hardware needs to work on two pixels at a time. That is, it will write a single byte, that happens to contain two pixels.

Also the internal *XN* iteration counter divides *XN* by two before using it to translate pixel count to byte count. If *XN* were to be loaded with '1' then dividing '1' by 2 results in 0, according to the rules of integer maths.

Reading a pixel from VRAM

There will be times when a pixel needs to be read from VRAM. The hardware allows it and supports two ways of doing it:

1. Read a single pixel using a line read.
2. Read a single pixel, or block, using a blit to program memory.

Either method can be used to read pixels from VRAM and both are just as efficient as the other. Register setup is about the same leaving little time advantages between them. A blitter transfer is much more useful for large bulk transfers, while a two pixel length line read is cheaper on resources when only a single byte needs to be read.

Pixel read using Line Read mode

Here the line drawing hardware is used in reverse to read data from VRAM, rather than writing to it. All other register settings apply as if it were a line write.

A line two pixels long is read from VRAM and placed into the *PIXELR* register. Since *PIXELR* is one byte wide then there is no point to read more than one byte from VRAM, although it is valid to do so, because all data will be stored into the *PIXELR* register ensuring the CPU misses all but the last byte read. To prevent this and ensure an accurate read the *XN* and *YN* registers must be set to 0x02 and 0x01 respectively.

This example reads two pixels from the screen co-ordinate (400, 450):

VGA6800 Programmers Manual

<i>X</i>	0x0190	Loads the starting X co-ordinate (400) of the line.
<i>Y</i>	0x01C2	Loads the starting Y co-ordinate (450) of the line.
<i>XN</i>	0x0002	Loads the pixel count in the X direction. ^{Note 2}
<i>YN</i>	0x0001	Loads the pixel count in the Y direction.
<i>DRAWCTL</i>	0x02	Loads the drawing mode (Read) and starts the hardware off. A pixel has been read when this bit is automatically cleared. ^{Note 1}
<i>PIXELR</i>	0xNN	Byte (Two pixels) read from VRAM co-ordinate (X,Y)

Note 1: Once the *DRAWCTL* register clears to 0x00 signals that the operation has been completed and the resultant read data can be accessed from register *PIXELR*.

Note 2: The *XN* register is loaded with '2' because the hardware needs to work on two pixels at a time. That is, it will write a single byte, that happens to contain two pixels.

Also the internal *XN* iteration counter divides *XN* by two before using it to translate pixel count to byte count. If *XN* were to be loaded with '1' then dividing '1' by 2 results in 0, according to the rules of integer maths.

Pixel read using Blit operation to Program Memory

Here the Blitter hardware is used to read one, or more, bytes from VRAM and place it into CPU program memory. It is valid to perform a block transfer if needed and is recommended for most bulk operations. After a blitter operation is complete the software can use the data stored in Program Memory as it pleases and in its own time.

The Blitter supports data transfers in either direction, that is:

1. Program Memory to VRAM.
2. VRAM to Program Memory.

In this example a single byte, two pixels long, is read from VRAM co-ordinates (400,450) and placed in Program memory at address 0x71AB where the software can then examine it:

<i>X</i>	0x0190	Loads the starting X co-ordinate (400) of the line.
<i>Y</i>	0x01C2	Loads the starting Y co-ordinate (450) of the line.
<i>XN</i>	0x0002	Loads the pixel count in the X direction. ^{Note 1}
<i>YN</i>	0x0001	Loads the pixel count in the Y direction.
<i>DMADRS</i>	0x71AB	CPU RAM address to place VRAM byte (Two pixels.)
<i>DRAWCTL</i>	0x12	Loads the drawing mode (read) and starts the hardware blitter. A pixel has been transferred when this bit is automatically cleared.

Note 1: The *XN* register is loaded with '2' because the hardware needs to work on two pixels at a time. That is, it will write a single byte, that happens to contain two pixels.

Also the internal *XN* iteration counter divides *XN* by two before using it to translate pixel count to byte count. If *XN* were to be loaded with '1' then dividing '1' by 2 results in 0, according to the rules of integer maths.

The contents of memory address 0x71AB will hold a pixel from the currently active drawing video page at co-ordinates (400,450).

There is no restrictions to transferring a larger block and is it only confined by free memory and the *XN* and *YN* register values.

Using the Blitter

Graphic image bulk transfers, or “blitting”, is supported by the video controller. It is very useful when large images have to be transferred to, and from, the VRAM's current drawing page.

To transfer a pixel at a time, using software only, will translate to very slow data throughput and may not be very useful. However, using hardware to do the same work results in very fast bulk transfers and simpler software. There are many benefits to using a hardware blitter and they will become obvious as software is written for it.

This Blitter hardware is essentially a hardware implement Direct Memory Access Controller that knows how to address pixel data in VRAM according to its rules, and address data in Program Memory according to its rules.

When a bulk transfer is initiated from VRAM to Program Memory, for example, the Blitter translates the cartesian co-ordinates of each pixel into an absolute VRAM linear address, applies it and reads a byte (2 pixels) from that address. It then transfers it to the Program Memory DMA Controller, that only works in absolute linear addresses, and writes the byte there.

Transferring data in the opposite direction sees the data flowing from Program Memory to VRAM with all addressing rules being applied.

Block Write using a Blit to VRAM

In this example the blitter hardware is used to place an image onto the current drawing VRAM page sourcing its data from Program Memory. This type of operation can be used for any graphic operation from clearing the region, to drawing a single character font to the screen, to placing a full colour graphic image anywhere on the screen.

The uses are many and varied and defined by the software. A blit to VRAM is fast and automated so once it has been started nothing can stop it until completed. This is further enforced as the CPU is placed into *HALT* mode. Unfortunately the M6800 CPU does not have a nice wait state generating scheme that can survive the potentially long delays imposed by a large blit, so it must be halted.

A typical example of a graphic image blit of 10 * 10 pixels and stored at address 0x7200 through to 0x7232 is demonstrated here. It is to be copied to VRAM co-ordinates (400,450) of the current drawing video page as a 10 by 10 image. The control register settings to accomplish this would look like this:

<i>X</i>	0x0190	Loads the starting X co-ordinate (400) of the line.
<i>Y</i>	0x01C2	Loads the starting Y co-ordinate (450) of the line.
<i>XN</i>	0x000A	Loads the pixel count in the X direction. (10 pixels = 5 bytes)
<i>YN</i>	0x000A	Loads the pixel count in the Y direction. (10 lines)
<i>DMADRS</i>	0x7200	CPU RAM address to read graphic image from. (10 * 10 image)
<i>DRAWCTL</i>	0x11	Loads the drawing mode (bulk write) and starts the hardware blitter. A block has been transferred when this bit is automatically cleared.

The Host CPU does not have a lot of program memory to dedicate to sourcing graphic images the size of the VGA pixel map. In fact the whole available CPU program memory space would occupy one quarter the VGA memory space. With this in mind graphic images must be carefully chosen, and in some cases, software could be used to create images on the fly. For example, draw a RED box using four lines rather than blitting a predrawn RED box from Program memory.

Block Read using Blit to Program memory

Here the Blitter hardware is used to read one, or more, bytes from VRAM and place it into CPU program memory. It is valid to perform a block transfer if needed and is recommended for most bulk operations. After a blitter operation is complete the software can use the data stored in Program Memory as it pleases and in its own time.

The Blitter supports data transfers in either direction, that is:

3. Program Memory to VRAM.
4. VRAM to Program Memory.

In this example a single byte, two pixels, is read from VRAM co-ordinates (400,450) of size (10,10) and placed in Program memory at address 0x7000 where the software can then examine it:

<i>X</i>	0x0190	Loads the starting X co-ordinate (400) of the line.
<i>Y</i>	0x01C2	Loads the starting Y co-ordinate (450) of the line.
<i>XN</i>	0x000A	Loads the pixel count in the X direction. (10 pixels = 5 bytes)
<i>YN</i>	0x000A	Loads the pixel count in the Y direction. (10 lines)
<i>DMADRS</i>	0x7000	CPU RAM address to place VRAM byte (Two pixels.)
<i>DRAWCTL</i>	0x12	Loads the drawing mode (bulk read) and starts the hardware blitter. A transfer has been completed when this bit is automatically cleared.

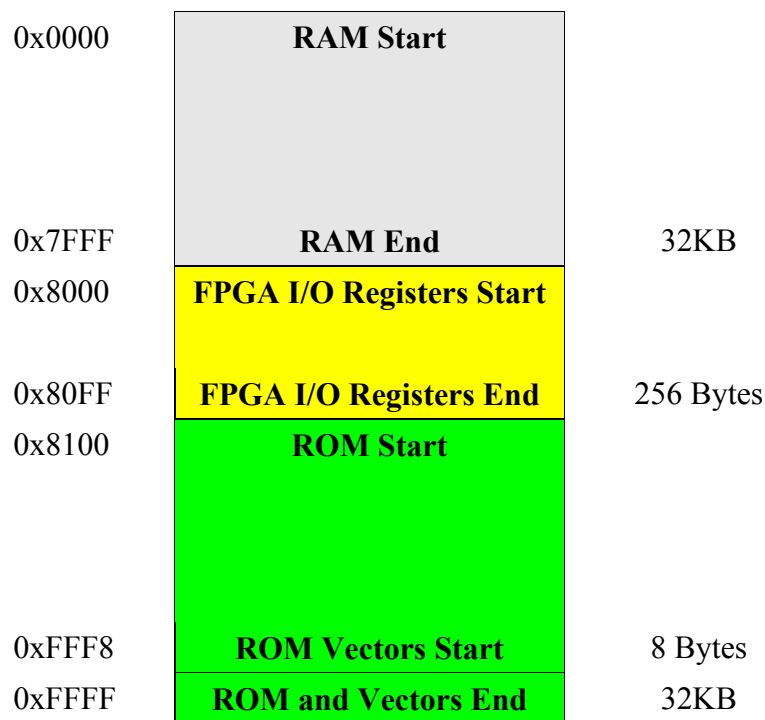
The contents of memory address 0x7000 through to 0x7032 will hold the data from the currently active drawing video page at co-ordinates (400,450).

There is no restrictions to transferring a larger block and is it only confined by free memory and the *XN* and *YN* register values.

CPU Memory Map

The M6800 has a very limited addressable memory space of 65536 Bytes, and does not leave much room for inclusion of a lot of memory, or peripherals. The most readily available memory devices of today can fill the CPU's memory space many times over.

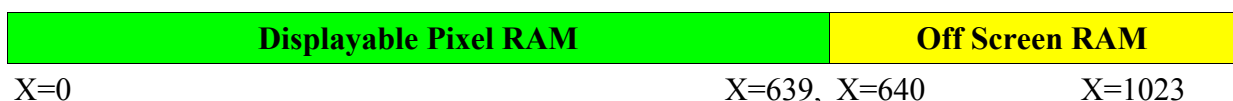
Therefore nearly all addressable memory space is occupied by RAM or ROM. A small section of 256 Bytes has been taken away from the ROM to allow for the FPGA control registers to show through. This means the first 256 Bytes of ROM can not be used to hold program. If there were program written to the device in that region then it would be invisible to the CPU and not accessible.



This memory map shows how full the CPU's memory space is. There is a lot of ROM present because programs may hold a many bitmaps in it for screen graphics. 32KB may be enough to do some basic work, but it will not be enough for serious VGA graphics. So a compromise of program drawn images and hard coded images needs to be found for each application.

Non Displayable Video RAM

For each video page there is a region of RAM that is not displayed to the screen but exists as an unused part each line. This is the off screen video RAM that exists at the end of each line. Each Video line is aligned to a 512 byte boundary, 320 bytes is required to hold displayable pixel data, while 128 bytes is not used and free to be applied in any way. In fact there is 100KB of off screen video RAM for each video page.



Video Line Organisation

From a software perspective the off screen video RAM would be difficult to use as it is so fragmented. However, the

VGA6800 Programmers Manual

FPGA's hardware blitter helps to translate linear addressing to Video RAM X,Y addressing. A block of program memory can be "blitted" to the off screen RAM via the blitter where it would be distributed according to its X,Y coordinates, as any graphic image would be.

To retrieve the image the blitter is reversed and the fragmented data would be read from the off screen RAM, and reassembled into linear program memory space as it originally would have been.

Off screen RAM can give an additional 100KB of memory over to the software to use as it needs. It is managed through the Video controllers X and Y registers eliminating the need for large address registers. All the software has to do is manage how that extra RAM is used.

FPGA Register Map

Here is a summary of the entire register set support inside the VGA6800 FPGA. The M6800 CPU can read, or write, any of these registers as it pleases. They are all aligned to reside in the "I/O" section of the M6800's memory map (0x8000 - 0x80FF.)

The absolute address of each register is not given, instead an offset is and should be added to the base address of the I/O register set to calculate the absolute address.

For example: to access the *VPAGE* register the software would calculate it like this: $0x8000 + 0x16 = 0x8016$.

<i>Offset</i>	<i>Size</i>	<i>Mnemonic</i>	<i>Description</i>
0x00	8-bit	<i>PIXELW</i>	Colour to write pixel as when drawing lines to the video RAM.
0x01	8-bit	<i>PIXELR</i>	Actual value of pixel read from video RAM.
0x10	16-bit	<i>X</i>	X co-ordinate for drawing operation. ^{Note 1}
0x12	16-bit	<i>Y</i>	Y co-ordinate for drawing operation. ^{Note 1}
0x14	16-bit	<i>XN</i>	X iteration count. This represents the number of pixel operations in the X direction. ^{Note 1}
0x16	8-bit	<i>VPAGE</i>	Video page register. The lower 4-bits represents the current display page, while the upper four bits represents the current drawing page. They do not have to be the same.
0x17	8-bit	<i>DRAWCTL</i>	Controls the type of drawing operation and acts as a completion status register. All bits are self clearing once the operations has completed.
0x18	16-bit	<i>YN</i>	Y iteration count. This represents the number of pixel operations in the Y direction. ^{Note 1}
0x1A	8-bit	<i>MX</i>	Iteration count for diagonal line drawing in the X direction. (TBA) ^{Note 2}
0x1B	8-bit	<i>MY</i>	Iteration count for diagonal line drawing in the Y direction. (TBA) ^{Note 2}
0x20	8-bit	<i>INTEN</i>	Interrupt enable mask. Each bit represents an interrupt from an internal logic block.
0x21	8-bit	<i>INTSTAT</i>	Interrupt Status Register. Each bit set represents an active interrupt from that logic block.
0x30	8-bit	<i>HWVERSION</i>	Holds a hard coded version number of the FPGA image loaded. This is a good way to know if certain logic blocks are supported in the FPGA.
0x40	16-bit	<i>DMADRS</i>	Direct Memory Access Address. This register holds the starting location in program memory where the Blitter is to read, or write, data being transferred between Video RAM and program memory. ^{Note 1}
0x50	8-bit	<i>UART0_RX</i>	UART 0 Receive data register. And data received by the UART logic block will be presented here. Reading this register also clears the UART's receive data read status bit.
0x51	8-bit	<i>UART0_TX</i>	UART 0 Transmit data register. Writing a byte to this register causes the UART transmitter to begin shifting it out. The UART's Transmit data status register will be cleared when transmission is complete.
0x52	8-bit	<i>UART0_RXSTAT</i>	UART 0 Receive Status register.

VGA6800 Programmers Manual

Offset	Size	Mnemonic	Description
0x53	8-bit	<i>UART0_TXSTAT</i>	UART 0 Transmit Status register.
0x60	8-bit	<i>UART1_RX</i>	UART 1 Receive data register. And data received by the UART logic block will be presented here. Reading this register also clears the UART's receive data read status bit.
0x61	8-bit	<i>UART1_TX</i>	UART 1 Transmit data register. Writing a byte to this register causes the UART transmitter to begin shifting it out. The UART's Transmit data status register will be cleared when transmission is complete.
0x62	8-bit	<i>UART1_RXSTAT</i>	UART 1 Receive Status register.
0x63	8-bit	<i>UART1_TXSTAT</i>	UART 1 Transmit Status register.
0x70	8-bit	<i>LED0</i>	Seven Segment LED display 0 (Left hand digit) – Write only.
0x71	8-bit	<i>LED1</i>	Seven Segment LED display 1 (Right hand digit) – Write only.
0x70	8-bit	<i>DIPSW</i>	DIP Switch Read register - Read only.

Note 1: 16-bit registers are arranged to suit a *Big Endian* microprocessor. This means the high byte of a 16-bit number is stored into the first (lowest) address and the low byte is stored in the next (highest) address.

Note 2: These registers, and the functions they imply, do not exist in version 1 hardware. Read the *HWVERSION* register to determine the level of revision.

UART Registers

There are two UART logic blocks compiled into the VGA6800 FPGA. They are provided to allow any sort of communications input, or output that might be needed by software applications.

The UART is a basic type that has most options hard coded to a single mode of operation. That means it supports 8-bit data, 2 stop bits and one start bit. This is a very common configuration for a UART and very rarely changes.

After a hardware reset the UART register are set to a “polled” mode with transmitter and receiver on, interrupts off and ready to go.

Should the interrupts need to be enabled then the software will have to setup the appropriate control register.

<i>Offset</i>	<i>Register Name</i>	<i>Description</i>
0x00	UART_RX	UART Receive data register. All data received and assembled by the serial to parallel converter is stored into this register. There is no FIFO making this register one byte deep.
0x01	UART_TX	UART Transmit data register. All data written here will be serialized and send out the Tx pin. There is no FIFO and is one byte deep.
0x02	UART_RXSTAT	Receiver status register.
0x03	UART_TXSTAT	Transmit status register.

UART Receive Status Register Description

UART_RXSTAT			
<i>Bit</i>	<i>Default state</i>	<i>State</i>	<i>Operation</i>
Bit-0	0	0	Receive Data Register is empty. There is no data to read.
		1	Receive Data Register has data to read.
Bit-1	0	0	Receive Interrupt Disabled. No interrupt will be generated.
		1	Receive Interrupt Enabled. All characters received and transferred to the Receive Data Register will generate an interrupt.
Bit-2,3,4	0	0x00	Internal clock divider set to 2. [38400 bps]
		0x08	Internal clock divider set to 4. [19200 bps]
		0x10	Internal clock divider set to 8. [9600bps]
		0x18	Internal clock divider set to 16. [4800 bps]
		0x20	Internal clock divider set to 32. [2400 bps]

UART_RXSTAT			
		0x28	Internal clock divider set to 64. [1200 bps]
		0x30	Internal clock divider set to 128. [600 bps]
		0x38	Internal clock divider set to 256. [300 bps]
Bit-5	0	0	Receiver Logic enabled.
		1	Receiver Logic Disabled. UART Receiver disabled.
Bit-6	0		Reserved.
Bit-7	0		Reserved.

UART Transmit Status Register Description

UART_TXSTAT			
<i>Bit</i>	<i>Default state</i>	<i>State</i>	<i>Operation</i>
Bit-0	0	0	Transmit Data Register empty. There is space to send more data.
		1	Transmit Data Register full. It will not accept any more data if written.
Bit-1	0	0	Transmit Interrupt disabled.
		1	Transmit Interrupt enabled. When a character has been sent and interrupt is generated.
Bit-2	0		Reserved.
Bit-3	0		Reserved.
Bit-4	0		Reserved.
Bit-5	0		Reserved.
Bit-6	0	0	Transmitter Enabled. All data written to the Transmit Data Register will be serialized and sent out.
		1	Transmitter Disabled. The whole transmit logic block is disabled.
Bit-7	0		Reserved.

Interrupt Controller Register

All internally generated interrupts, from each logic function that supports it, can be externally enabled, or disabled. Enabled interrupts are then collected together and summarised to form a single interrupt to the CPU.

The CPU also has its own interrupt mask making the interrupt structure very well controlled.

The main interrupt control register provided by the FPGA logic is a read/write register. Its companion status register is also read/write but for different reasons. A read from the status register returns all interrupt sources that have triggered. A write to it will clear those interrupt sources at this level only. Therefore the actual interrupt source has not been cleared, just silenced for a while.

<i>Offset</i>	<i>Size</i>	<i>Mnemonic</i>	<i>Description</i>
0x20	8-bit	<i>INTEN</i>	Interrupt Enable mask. Each bit represents an interrupt from an internal logic block.
0x21	8-bit	<i>INTSTAT</i>	Interrupt Status Register. Each bit set represents an active interrupt from that logic block.

This register is used to enable, or disabled, any of the internal interrupt sources. It is a read/write register.

<i>INTEN</i>			
<i>Bit</i>	<i>Default state</i>	<i>State</i>	<i>Operation</i>
Bit-0	0	0	UART 0 interrupt disabled.
		1	UART 0 interrupt enabled.
Bit-1	0	0	UART 1 interrupt disabled.
		1	UART 1 interrupt enabled.
Bit-2	0	0	Vertical Refresh interrupt disabled.
		1	Vertical Refresh interrupt enabled.
Bit-3	0		Reserved.
Bit-4	0		Reserved.
Bit-5	0		Reserved.
Bit-6	0		Reserved.
Bit-7	0		Reserved.

Internal interrupt status can be read from this register. When an active flag is read it is up to the software to write an arbitrary value back into this register to clear it down. Failing to do so will lock the CPU into an infinite interrupt service loop.

<i>INTSTAT</i>			
<i>Bit</i>	<i>Default state</i>	<i>State</i>	<i>Operation</i>

VGA6800 Programmers Manual

<i>INTSTAT</i>			
Bit-0	0	0	UART 0 interrupt idle.
		1	UART 0 interrupt triggered.
Bit-1	0	0	UART 1 interrupt idle.
		1	UART 1 interrupt triggered.
Bit-2	0	0	Vertical Refresh interrupt idle.
		1	Vertical Refresh interrupt triggered. The interrupt source is tied to the VSYNC signal at a fixed 60Hz period making it a good candidate for a software time base.
Bit-3	0		Reserved.
Bit-4	0		Reserved.
Bit-5	0		Reserved.
Bit-6	0		Reserved.
Bit-7	0		Reserved.

LED display Registers

There are two seven segment LED displays attached to the FPGA to allow user specific information to be displayed on them. They are essentially attached to 8-bit latches that the software can write directly to. The final pattern displayed on them is up to the programmer.

<i>Offset</i>	<i>Size</i>	<i>Mnemonic</i>	<i>Description</i>
0x70	8-bit	LED0	Seven Segment LED display 0 (Left hand digit) – Write only.
0x71	8-bit	LED1	Seven Segment LED display 1 (Right hand digit) – Write only.

LED0			
<i>Bit</i>	<i>Default state</i>	<i>Segment</i>	<i>Operation</i>
Bit-0	0	A	Logic '1' lights segment 'A'
Bit-1	0	B	Logic '1' lights segment 'B'
Bit-2	0	C	Logic '1' lights segment 'C'
Bit-3	0	D	Logic '1' lights segment 'D'
Bit-4	0	E	Logic '1' lights segment 'E'
Bit-5	0	F	Logic '1' lights segment 'F'
Bit-6	0	G	Logic '1' lights segment 'G'
Bit-7	0	DP	Logic '1' lights segment 'Decimal Point'

LED1			
<i>Bit</i>	<i>Default state</i>	<i>Segment</i>	<i>Operation</i>
Bit-0	0	A	Logic '1' lights segment 'A'
Bit-1	0	B	Logic '1' lights segment 'B'
Bit-2	0	C	Logic '1' lights segment 'C'
Bit-3	0	D	Logic '1' lights segment 'D'
Bit-4	0	E	Logic '1' lights segment 'E'
Bit-5	0	F	Logic '1' lights segment 'F'
Bit-6	0	G	Logic '1' lights segment 'G'
Bit-7	0	DP	Logic '1' lights segment 'Decimal Point'

This is an example list of segment configurations that can be written to an LED latch to produce a desired character.

VGA6800 Programmers Manual

<i>Bitmap</i>	<i>Character</i>
0x3F	0
0x06	1
0x5B	2
0x4F	3
0x66	4
0x6D	5
0x7D	6
0x07	7
0x7F	8
0x6F	9
0x77	A
0x7C	B
0x39	C
0x5E	D
0x79	E
0x71	F

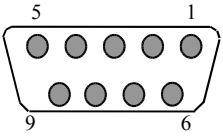
Appendix A

VGA Monitors

(Video Graphics Array)

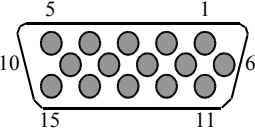
IBM developed VGA in 1987 as one of the first video monitors to employ the use of “Analogue” colour signals. It used a translation memory (Palette table) to convert the pixel code in video memory to colour information that will be feed to the video monitor. Colour palettes provided a way to bridge the gap between limited video memory and displayable colours.

VGA 9 pin

 <p>Female (Front View)</p>	1	Red	6	Red Return
	2	Green	7	Green Return
	3	Blue	8	Blue Return
	4	Horizontal Sync	9	Sync Return
	5	Vertical Sync		
Screen resolution 640x480 (16 colours)	ROM Character resolution 40x25 (16 colours) 80x25 (16 colours) 80x30 (Monochrome)			
Colour depth 1-bit (Monochrome) 4-bit (16 colours)	Video memory 256KB standard			
Screen Refresh $F_v = 60\text{Hz}, 70\text{Hz}$	Screen horizontal scan rate $F_h = 31.5\text{KHz}$			

VGA 15-pin

VGA video cards and monitors were also the first to use the higher density *DB* connector that squeezed an extra six pins into the current 9-pin *DB* connector. The connector is still being used on video monitors today and carry much larger bandwidths than VGA ever did.

 <p>Female (Front View)</p>	1	Red	9	Not used
	2	Green	10	Ground
	3	Blue	11	ID Bit-1
	4	ID Bit-0	12	ID Bit-2
	5	Not used	13	Horizontal Sync
	6	Red Return	14	Vertical Sync
	7	Green Return	15	ID Bit-3
	8	Blue Return		

VGA6800 Programmers Manual

Screen resolution 640x480 (16 colours)	ROM Character resolution 40x25 (16 colours) 80x25 (16 colours) 80x30 (Monochrome)
Colour depth 1-bit (Monochrome) 4-bit (16 colours)	Video memory 256KB standard 512K expandable on board.
Screen Refresh $F_v = 60\text{Hz}, 70\text{Hz}$	Screen horizontal scan rate $F_h = 31.5\text{KHz}$

According to the Video Electronics Standards Association (VESA), this is the video standard for VGA monitors and driver hardware.

This pin-out and video signal is also adapted and supported by VESA as a standard for video monitor wiring and connection method. This is also the first use of a denser 15 pin connector housed inside the standard DB-9 shell.

Appendix B

Revision Levels

The FPGA software provides a register that holds a number which reflects the level of revision of functionality. Software can use it to determine functions to be supported by software, or hardware, or both.

Read the register: *HWVERSION* to learn the current FPGA revision level.

<i>Revision Code</i>	<i>Description</i>
0x01	First release of the VGA6800 FPGA design. Missing functionality: <ol style="list-style-type: none">1. Diagonal line drawing support in hardware.2. Some minor hardware bugs that provide no serious side effects to the operation of the controller.
0x02	

Glossary of terms

Analog

In this context, it refers to a voltage level that can be on, off and many steps in between. Often spelt using the American spelling of the word: *Analog*.

Big Endian

Defines how a CPU will store 16-bit data, or larger, into memory. Big Endian dictates that the highest data byte is stored into the lowest address, while the lowest data byte is stored into the last address.

CGA

Colour Graphics Adapter. A type of video monitor.

CRT

Cathode Ray Tube. Type of display technology used in most computer monitors, Televisions and other electronic equipment.

EGA

Enhanced Graphics Adapter. A type of video monitor.

FIFO

First In First Out. Type of memory pipe.

FPGA

Field Programmable Gate Array. A new type of semiconductor that merges hardware speed with software versatility. The chip holds many unconnected logic elements that are eventually connected together using a connecting matrix. The resultant connections form the intended hardware design.

HiColor

A name given to the number of bits allocated to a single pixel. In this case it is fifteen and/or sixteen bits. *HiColor* graphic mode does not require the presence of a Palette translation table to produce the Red, Green and Blue components of the video signal. These are derived directly from the video memory where pixels are allocated in 15-bit mode as: R₅G₅B₅, and 16-bit mode: R₅G₅B₆.

Horizontal Sync

Signal used to indicate the beginning of a horizontal line on a computer video screen.

Hz

Unit of frequency measurement: One cycle per second.

KB

Kilo Byte. 1KB = 1024 bytes, 2KB = 2048 bytes.

KHz

Unit of frequency measurement. 1KHz = 1000Hz.

LCD

Liquid Crystal Display.

Little Endian

Order CPU stores 16-bit data, or larger, into memory. Little Endian defines that the lowest byte is stored into the lowest address, while the highest byte is stored in the last address. When memory is viewed it appears that the data has been stored backwards.

MB

Mega Bytes. 1MB = 1048576 bytes. 2MB = 2097152 bytes.

MDA

Monochrome Display Adapter. Type of video monitor.

Overlay Plane

Overlay Planes allow two, or more, video pixel grids to be displayed together onto a single CRT. The overlay and main pixel planes were aligned so that each pixel sat in front of, or behind, the next layer. Using Black as a transparent colour layers behind it show through. Overlays are excellent ways of producing “Floating” text and images over a main image while reducing CPU loads as it tries to produce more graphic frames.

Pixel

A single point of light emitted from an electronically controlled display device such as CRT or LCD.

PGA

Professional Graphics Array.

RAM

Random Access Memory. An information storage medium for computer memories.

RAMDAC

RAM controlled Digital to Analog Conversion. Commonly used to translate a small subset of colours out of a larger super set. *See also Video Palette.*

RGBHV

Red, Green, Blue, Horizontal, Vertical. A short way of saying Analogue monitor.

TFT

Thin Film Transistor. Often used to refer to the technology used to construct display matrixes on LCD's.

TrueColor

Is a name given to a video mode where 24-bits of video memory data is allocated to each pixel. This mode does not require the presence of a translation palette and will produce 16 million possible colours on the screen at any one time. The organization of the colour bits within a *TrueColor* pixel is: R₈G₈B₈.

In some systems that have 32-bit busses 8-bits of pixel data is not used. But some systems utilised the extra 8-bits as a “Overlay” plane that could display monochrome, 2-bit, 4-bit and 8-bit colour per overlay pixel.

TTL

Transistor Transistor Logic. A type of digital electronic circuit technology.

Vertical Sync

Used to indicate the start of each new screen image drawn onto a video monitor.

Video Palette

Commonly used to translate a small subset of colours from a larger super-set. The video palette is essentially a translation table that accepts a tokenized colour, (pixel) in one end, and looks that up in a preprogrammed RAM array to extract a Red, Green and Blue value ready to feed into the video monitor.

VGA

Video Graphics Array. Type of video monitor.

VRAM [1]

VGA6800 Programmers Manual

Video Random Access Memory. A memory device, or region of main RAM, allocated to holding the entire pixel map.

VRAM [2]

Video Random Access Memory. A family of memory devices designed to remove video refresh data bus from CPU access data bus.

SVGA

Super Video Graphics Array. Type of video monitor.

SXGA

Super eXtended Graphics Array. Type of video monitor.

UXGA

Ultra eXtended Graphics Array. Type of video monitor.

XGA

EXtended Graphics Array. Type of video monitor.

End of Document