VGA Hardware

The VGA is a complex piece of hardware. Even though it's old, many modern graphics cards are compatible with it, including NVidia and ATI cards. This can make writing a VGA driver rather attractive. The amount of compatibility varies however, so never assume a card is compatible without proper hardware detection. Apart from real machines, several emulators and virtual machines provide VGA emulation: Bochs, QEMU and Microsoft Virtual PC to name a few.

WARNING: Improperly changing CRTC or hardware settings can be harmful to the video card and attached monitor. Configuring graphics hardware may be a rewarding process, and many have achieved their goals with success and have even tampered with them well beyond their original specifications. Fact remains that there are *several* known instances of monitors - CRT and LCD alike - burning out after being fed poor data. Similarly, the video card might lack similar safeguards against accidental misconfiguration.
DISCLAIMER: The information provided might not be accurate, and using it is to be done entirely at your own risk. There have not been any reported mishaps in the entire history of OSDev.org, but in the odd chance you break something, we are not responsible.

Overview

While the VGA chip is quite a simple piece of hardware compared to modern video equipment, it is possibly one of the more complicated devices to program for, and especially in the old days knowing your way around this particular device was sufficient for establishing quite a reputation. While currently a legacy device, it is a good place to begin practicing your video driver skills. While a full-blown VGA driver might make an USB controller look trivial, there are fortunately many shortcuts available for taking.

What's not covered

While this page tries to be a complete overview on what the VGA can do, it does not fully cover the whole set of graphics. After all, a video card only turns bytes in it's memory into a signal on the connector on it's backside. Determining what bytes to put in memory is only barely touched in the wiki in general - there are examples of plotting pixels and setting individual characters but your OS will determine what pixels are formed by an image and which characters are part of your title screen. On the remote end, monitors have their own way of dealing with signals. A lot of those settings dictated by monitors are needed by the video card, and each resolution comes with its own set of settings. You can find out your own set of settings by using a set of equations, but you can skip that step and reuse one of the examples provided at the example settings instead. The CRTC chapterexplains them in detail.

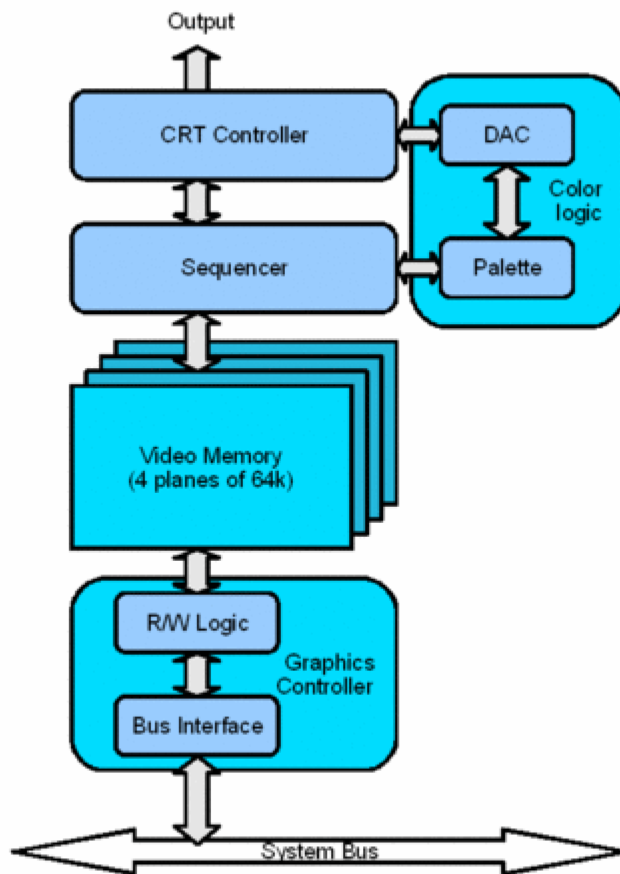
Getting started

There's a fair share of modesetting code available around the web. The basic steps involve calculating the needed register values, writing them to the VGA, then continue with drawing. You'll need:

- Port I/O: The VGA needs 8-bit read/writes, and 16-bit writes.
- MMIO: The VGA uses uncached byte accesses to 0xA0000-0xBFFFF. In several cases, larger writes are also allowed.
- Functions to read and write registers for each VGA component - since there are many more registers than there are ports you will need a wrapper for this.
- A structure that contains the VGA display settings. For a nice list of things you want to set, you can use the example register settings part. Keep in mind that you might also want a structure for things that change during drawing, such as colours and offsets.
- A function that writes that structure to the device
- A function that fills out that structure. You can also use a hardcoded structure initially.
-

Hardware components

The VGA can be divided in several parts. Historically, the predecessor of the VGA - the EGA - had several chips to perform each part in the system. These chips could be configured to your liking using the I/O Bus. On the VGA, these have been merged into one chip (with the exception of the DAC).
The following diagram shows which units are responsible for which parts:



This diagram is, however, a simplification for the ease of programming, and should not be considered correct.

VGA Registers

The VGA has a lot (over 300!) internal registers, while occupying only a short range in the I/O address space. To cope, many registers are indexed. This means that you fill one field with the number of the register to write, and then read or write another field to get/set the actual register's value.
All registers are accessed as 8 bit. The parts of a register that are not used should officially be preserved, although a lot of programs simply set them to zero. However, not all fields present in the VGA are documented here, so you will either look up a different reference, or preserve the undocumented fields.
In the documentation below, a port number and possibly an index is provided. The port is usually the base port for indexed registers, or the actual port for single registers.
Note that PCI boards do *not* report the VGA addresses in their configuration space, and that the addresses can not be remapped. It is therefore not possible to properly operate two cards in VGA mode at the same time.

Port 0x3C0

This port is a confusing one: you write both the index and data bytes to the same port. The VGA keeps track of whether the next write is supposed to be the index or the data byte. However, the initial state is unknown. By reading from port 0x3DA it'll go to the index state. To read the contents, feed the index into port 0x3C0, then read the value from 0x3C1 (then read 0x3DA as it is not defined whether the VGA expects a data byte or index byte next).

Port 0x3C2

This is the miscellaneous output register. It uses port 0x3C2 for writing, and 0x3CC for reading. Bit 0 of this register controls the location of several other registers: if cleared, port 0x3D4 is mapped to 0x3B4, and port 0x3DA is mapped to 0x3BA. For readability, only the first port is listed and bit 0 is assumed to be set.

Port 0x3C4, 0x3CE, 0x3D4

These are the most used indexed registers. The index byte is written to the port given, then the data byte can be read/written from port+1. Some programs use a single 16-bit access instead of two byte accesses for writing, which does effectively the same. (take care of byte ordering when doing so)
Port 0x3D4 has some extra requirements - it requires bit 0 of the **Miscellaneous Output Register** to be set before it responds to this address (if cleared, these ports appears at 0x3B4). Also, registers 0-7 of 0x3D4 are write protected by the protect bit (bit 7 of index 0x11)

Port 0x3C6
Port 0x3C6 only contains the DAC Mask Register, which can easily be accessed by a simple read/write operation on this port. Under normal conditions it should contain 0xff.

Port 0x3C8
Port 0x3C8, 0x3C9 and 0x3C7 control the DAC. Each register in the DAC consists of 18 bits, 6 bits for each color component. To write a color, write the color index to port 0x3C8, then write 3 bytes to 0x3C9 in the order red, green, blue. If you want to write multiple consecutive DAC entries, you only need to write the first entry's index to 0x3C8 then write all values to 0x3C9 in the order red, green, blue, red, green, blue, and so on. The accessed DAC entry will automatically increment after every three bytes written. To read the DAC entries, write the index to be read to 0x3C7, then read the bytes from port 0x3C9 in a similar fashion (as with writing, the index will increment after every three bytes read)

Video Memory Layout

The video memory consists of four 'planes' (individual units) of memory, each with a size of 64KB, giving the VGA 256k of video memory. Connected to it is the Sequencer, which interprets this memory to generate colors which are fed to the subsequent stages. The way colors are organized in this memory mainly depends on the color depth.
Specific details about how memory is accessed from the host is can be found by reading about the Graphics Controller, detailed information about video memory is rendered can be found by reading about the Sequencer.

Memory Layout in 16-color graphics modes

16 colors means there are 4 bits used per color. The VGA has four planes, and for each pixel, and each plane holds one bit of each pixel drawn. Since the information for each pixel is scattered over four memory locations, this is the most difficult memory model.
Bit 7 of each address contains information about the first pixel, Bit 6 has information about the next pixel, and so on.
Plane 0 contains the first bits of all pixels, Plane 1 the second bits and so on.

Example:

	one byte							
Bits	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Plane 0	0	0	0	0	1	1	1	1
Plane 1	0	0	1	1	1	1	0	0
Plane 2	0	1	1	0	0	1	1	0
Plane 3	0	1	0	1	0	1	0	1
Colors displayed	0000 (0)	1100 (12)	0110 (6)	1010 (10)	0011 (3)	1111 (15)	0101 (5)	1001 (9)

The Plane Write Enable register is used to choose the plane to be written, then the memory can be written by written by accessing the corresponding address in memory.

Memory Layout in 256-color graphics modes

In this mode, each byte of video memory describes exactly one pixel. Pixels are generated by increasing address in linear mode, with all colors taken from plane 0. In planar mode (Also known as Mode X) each address describes 4 consecutive pixels, one from each plane. Plane 0 describing the first pixel, plane 1 the next, and so on. Technically speaking this is what always happens, but the standard 320x200x256 mode "chains" the planes such that 2 lowest order bits select the plane and the memory thus appears linear.
In linear mode, each byte in host memory corresponds to one pixel on the display, making this mode very easy to use. Mode X requires the use of Plane Write Enable register to select the plane to be written.

Memory Layout in text modes

In text mode, the screen is divided into character cells rather than pixels. Only three of the four planes are actually in use. Plane 0 contains the character codes for each cell, Plane 1 contains the respective attributes.
Plane 2 contains the font data. For each of the 256 available characters this plane has 32 bytes reserved. Each byte represents one horizontal cross section through each character. The first byte of each group defines the top line, each next byte describes the rows below it. For every set bit, the foreground color is used, For every cleared bit, the background color is used.
Although 32 bytes are reserved for each character, only 16, 14, or 8 of them are commonly used, depending on the character height.

Planes 0 and 1 are accessible from the host by writing to the video memory range. Plane 0 is accessed on even addresses, plane 1 is accessed on odd addresses, with each consecutive 16-bit value describing the next character. Accessing plane 2 to change fonts requires changes in addressing logic.

Memory Layout in 4-color modes
The CGA was limited to 4 concurrent colors, with two bits each. The EGA adds two extra bits by adding a pair of extra planes, increasing from the old two to the current four planes per pixel. If you want a 4-color mode that means you just should not touch planes 2 + 3.

Todo: determine the b/w/d, shift mode and odd/even mode for CGA compatibility (guesstimated at word mode, interleaved shift, odd/even enabled, i.e. equivalent to text mode except for the alphanumeric bit)

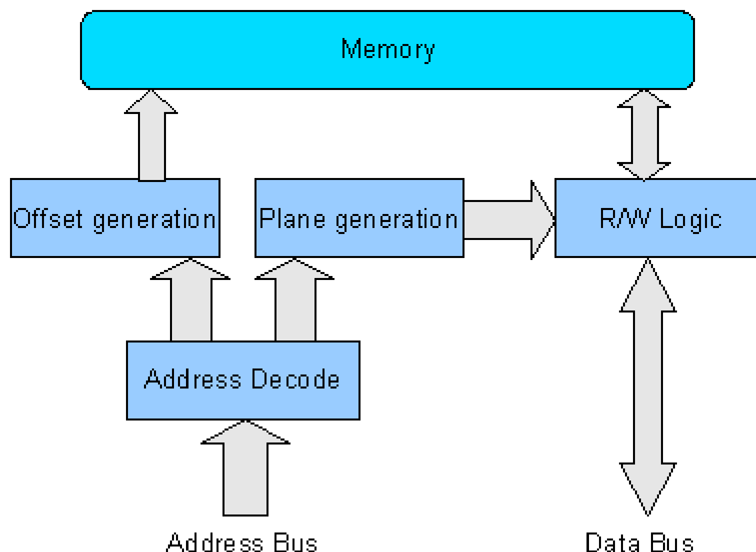
The Graphics Controller

The Graphics Controller (Abbreviated to GC) is responsible for directing memory reads and writes to and from video memory.
The memory consists of four planes of 64k on a standard VGA. Each read and write operation selects one address within all of these planes, then operates on all four planes. This means that for each byte of data written, four bytes of video memory might potentially be changed.
Apart from a few standard modes, the implementation of various control bits vary between implementations. However, the exact details can be probed by performing writes in the mode to check, then reading it out in planar mode.

These modes seem to be consistent among all VGA compatibles and emulators I've tested:

- Mode 3 (Text mode) (right now I can only set it by keeping a state dump of VGA registers of the old mode)
- Mode 11h (Planar 16 color mode)
- Mode 13h (Linear 256 color mode)
- Mode X (Planar 256 color mode)
  I have yet to write code that enters the following standard modes on all hardware
- Mode 04h (4 color mode) - Not tried, should be achieved by setting interleaved shift mode and ignoring planes 2 & 3.

The Graphics Controller consists of of some addressing logic, and some specific read/write logic.
The Addressing logic controls the address in video memory to be accessed, and to some extent, the individual pages within it.
The Read/Write logic controls which planes of the memory are actually read or written, and how these values relate to the value being sent by the CPU.



Addressing Logic

Registers involved:

Register Name	Port	Index	7	6	5	4	3	2	1	0
Sequencer Memory Mode Register	0x3C4	0x04					Chain 4	Odd/Even Disable		

Chain 4 Bit

The Chain-4 bit changes accesses to video mode from a planar mode when clear to a linear mode when set. While under common circumstances this bit is emulated properly, the way this bit actually works is however very different among implementations (especially emulators) and can have strange effects if you are unaware of it. What works in all cases, is if chain-4 matches the other settings that are common for established modes (i.e. if you enable chain-4, also make sure the other registers match the expected values for mode 13). Here is a list of differences between various implementations of Chain-4. Real hardware has so far proven to be consistent with the officially documented behavior.

	Bochs	Qemu	Virtual PC	ATI Card (X300)	NVidia (GeForce 6 6150)
Chain 4 has effect on writes	Yes	Yes	Yes	Yes	Yes
Chain 4 has effect on output	Yes	No	No	No	No
Plane Write Enable takes effect on writes	No	Yes	No	Yes	Yes

In bochs + VPC, as well as the hardware tested, Chain4 writes occur to this address:

```
plane  = addr & 0x0003; // lower bits
offset = addr & 0xfffb; // rest of the bits. multiples of 4 wont get written
```

In qemu writes go here:

```
plane  = addr & 0x0003; // lower bits
offset = addr >> 2;     // only the first 16k of each page gets written.
```

Note the fact that in Bochs, Chain-4 alters the physical display of the screen (equivalent to what doubleword mode normally does). This makes Bochs possibly troublesome since you only need to toggle this bit to enter Mode-X, while real hardware also requires that you change doubleword mode into byte mode.

Odd/Even Disable Bit

Seems to be a Don't Care bit in bochs+qemu
VPC generates some form of echo:

```
plane1 = addr & 0x0001; // lowest bit
plane2 = plane1 + 2;    // pick the other odd/even plane
offset = addr & 0xfffe; // generate the offset
write (plane1, offset); // write to the plane
write (plane2, offset); // write to the other plane
```

This matches the NVidia card, however my ATI card (and somewhat older, my V2x00 board) behaves slightly different (its pretty close though):

```
offset = addr & 0xffff; // generate the offset
```

There are probably some more things involved here, todo

Read/Write logic

The Read/Write Logic performs several operations on the written/read data, and a set of internal registers called the latches. Reading from video memory loads these latches with the value emitted by video memory. Write operations use the latches as an additional data source, apart from the data written from the host processor. The read/write logic has several different operation modes. These can be chosen by setting the Graphics Mode register. The VGA has four write modes and 2 read modes, which can be set independently. By default, the VGA operates in read mode 0 and write mode 0 in such a fashion that all written data goes straight to memory, and read data from each plane is ORed together.

Registers involved:

Register Name	Port	Index	7	6	5	4	3	2	1	0
Graphics Mode Register	0x3CE	0x05					Read Mode		Write Mode	
Map Mask Register	0x3C4	0x02					Memory Plane Write Enable			
Enable Set/Reset Register	0x3CE	0x01					Enable Set/Reset			
Set/Reset Register	0x3CE	0x00					Set/Reset Value			
Data Rotate Register	0x3CE	0x03				Logical Operation		Rotate Count		
Bit Mask Register	0x3CE	0x08	Bit Mask							

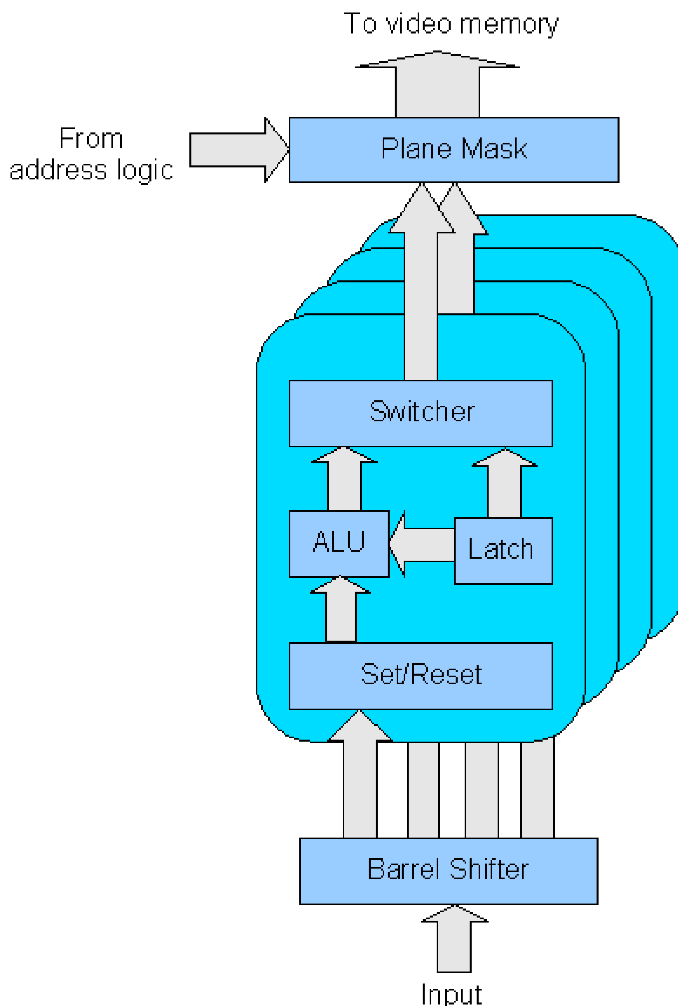
Todo: Study effects of multibyte reads/writes on logic and latch operation

The Latches

Probably the most interesting part of the VGA's internal wiring is the presence of the data latches. In the old times, the VGA could only accept 8 bits at a time. The latches hold 4x8 bits, and are used as a temporary register for VGA reads and writes. By putting that register to good use, a DOS-era programmer could well exceed the data transfer rate that the 8-bit bus was capable of, and instead use the 32-bit pipeline onboard the video card. The latches are written to whenever a load from video memory occurs. An address is supplied to video memory, which emits the 4 bytes, one for each plane, into the latches. From there, the video card determines what to send to the CPU. The latches are used again when writing to video memory. Together with the various read and write modes, the latches allowed video-to-video transfers, pattern and raster operations, as well as supplying original data for doing partial writes.

Write Mode 0

Write mode 0 is the standard write mode.



When a byte is written, it follows the following steps

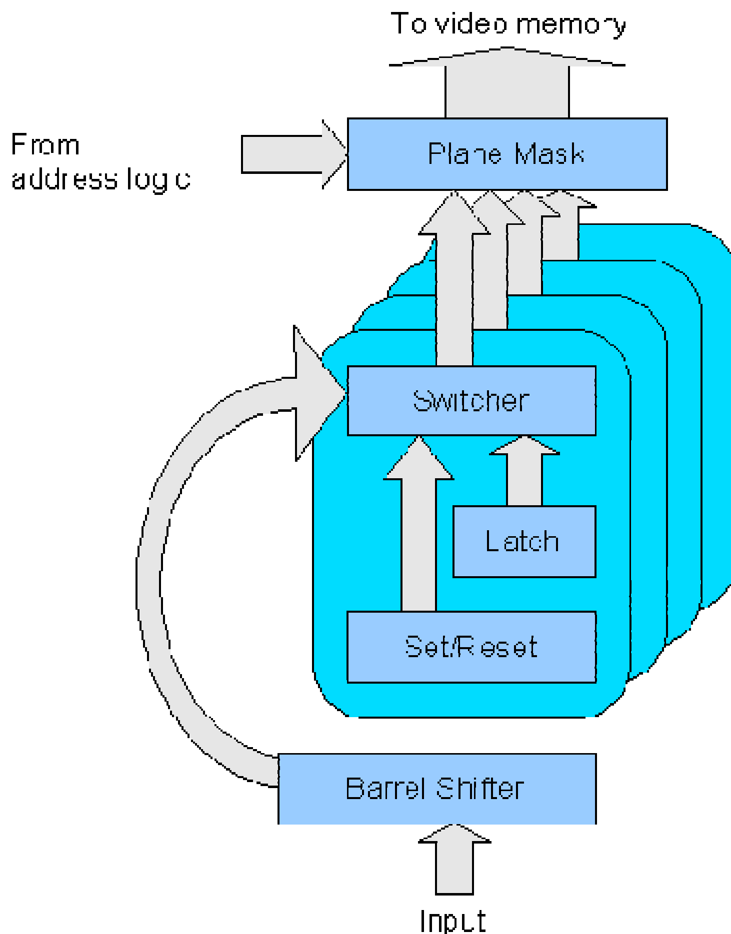
- The input byte is rotated right by the amount specified in Rotate Count, with all bits shifted off being fed into bit 7
- The resulting byte is distributed over 4 separate paths, one for each plane of memory
- If a bit in the Enable Set/Reset register is clear, the corresponding byte is left unmodified. Otherwise the byte is replaced by all 0s if the corresponding bit in Set/Reset Value is clear, or all 1s if the bit is one.
- The resulting value and the latch value are passed to the ALU
- Depending of the value of Logical Operation, the following operation is performed:

Value of Logical Operation	Result
0 (0x00)	The byte from the set/reset operation is forwarded
1 (0x08)	Both inputs are ANDed together
2 (0x10)	Both inputs are ORed together
3 (0x18)	Both inputs are XORed together

- The Bit Mask Register is checked, for each set bit the corresponding bit from the ALU is forwarded. If the bit is clear the bit is taken directly from the Latch.
- The Memory Plane Write Enable field is ANDed with the input from the address logic. For each set bit in the result, the corresponding plane is loaded with the result.

Write mode 3

Write mode 3 can, among others, be used for transparent writes with a constant color



Todo: thorough testing. This is currently an unverified interpretation of existing docs

When a byte is written:

- The input byte is rotated right by the amount specified in Rotate Count, with all bits shifted off being fed into bit 7
- The resulting value is ANDed with the Bit Mask Register, resulting in the bit mask to be applied.
- Each plane takes one bit from the Set/Reset Value register, and turns it into either 0x00 (if set) or 0xff (if clear)
- The computed bit mask is checked, for each set bit the corresponding bit from the set/reset logic is forwarded. If the bit is clear the bit is taken directly from the Latch. The result is sent towards memory.
- Finally, The Memory Plane Write Enable field and the input line from the address logic are ANDed together. The bits that remain set are the planes that are actually written.

Todo: more write modes, read modes

The Sequencer

The Sequencer is responsible to convert video memory to color indexes. Like the graphics controller, it has some special addressing logic, which is designed to iterate over the memory in a sensible manner to produce images out of video memory data.
The Sequencer either operates in text (alphanumeric) mode or graphics mode

Alphanumeric Mode

In alphanumeric mode the four planes are assigned distinct tasks. Plane 0 contains character data, while plane 1 contains Attribute data. In a standard text mode, these planes are interleaved into host memory. Plane 2 holds the font data. When displaying text data, the sequencer loads the character/attribute pair for the current set of eight pixels, after which it uses the value for byte 0 to look up the corresponding character in plane 2 and adds the character line to get the font data needed. It then pops out the bits from MSB to LSB, generating the chosen foreground color when a 1 is encountered, and the background color when a 0 is encountered.
In typical text modes, the stored font is not directly accessible and needs some changes in addressing logic to be read or written.

TODO: schematics
TODO: testing

Graphics Mode

In graphics mode, operations can be divided into two steps: address computation and shift logic. The sequencer computes an address, then reads out the four planes from that address, and generates 8 pixels from these values. The shift logic has three operating modes: single, interleaved and 256-color shift. Although the VGA 'supports' various color depths, these are basically varations on 16-color modes.

Shift modes

While going through memory, the sequencer reads in a 4 bytes at a time from each of the four planes, then outputs 8 pixel colors. The VGA has three distinct modes of grouping this data into pixel values. The setting depends on two bits in a VGA register: the 256-color shift and interleaved shift bits. when both are off, Single shift mode is selected, otherwise the corresponding mode is used (256 color shift mode takes precedence over interleaved shift mode)

Register Name	Port	Index	7	6	5	4	3	2	1	0
Graphics Mode Register	0x3CE	0x05		256-Color Shift	Interleaved Shift					

- Single shift mode
  This mode is used in 16 color modes. For each pixel, one bit is popped off each plane and put together to form the value of a pixel. An example is given in Memory Layout in 16-color graphics modes
- Interleaved Shift Mode
  This mode makes 4-color modes relatively easy: 2 bits are popped off the most significant side of plane 0. The same is done with plane 2, which become the most significant bits (in 4-color modes, these are zero) After 4 pixels being popped from planes 0 and 2, the same is done with plane 1 and 3, until all 8 pixels have been generated.
- 256-Color Shift Mode
  This mode causes 4 bits to be popped of each time. Plane 0 gives the first two pixels, Plane 1 the next two and so on. However, it is not defined in which order this happens. Because this mode is normally used solely in 256-color modes where the color logic will merge two 4-bits pixels together to form one 8-bit pixel, the communication inbetween is not certain. However, the bits can only be shifted out one of two possible sides, and supporting two possibilities can be overseen. Another problem to this method is, you can not detect which method is used without user intervention or keeping a list. Either way, this method either shifts left (from the msb) or right (from the lsb). If you know the ordering of your video card, you can create a linear 16-bit color mode.

Address Calculation

For each group of pixels, the sequencer calculates the address in video memory where to load the data from.

This address is calculated in three steps:

First the starting address of video memory is calculated: **TODO**

After that a scanline is rendered. The first address is read and the contents is split into pixels. Then the address is incremented by either 1, 2 or 4 for the next set of pixels until the scanline completes (the consequence of this is that each scanline is a multiple of eight pixels wide). The increment depends on wether the VGA is operating in "byte mode", "word mode", or "doubleword mode". These can be set using two bits: (doubleword modes takes precedence over byte/word mode) **TODO: Registers**

Once a scanline is complete, the original value at the start of the scanline is loaded, and the scanline counter is incremented. The VGA then does one of the following:

- Add the virtual width to the offset (and going to the next sequence of pixel data in memory). You can change the virtual width to change the amount of free data between scanlines, which can be useful for scrolling screens.
- Leave the value unchanged (and draw the same scanline again, identically). This is done when doublescanning **TODO**
- Reset the address to 0 (and start rendering from a different location in video memory) This can be used to create splitscreens **TODO**

Color Logic

This block revolves around the Attribute Controller, Palette RAM and DAC, which are together responsible for generating a color signal out of an index generated by the Sequencer.

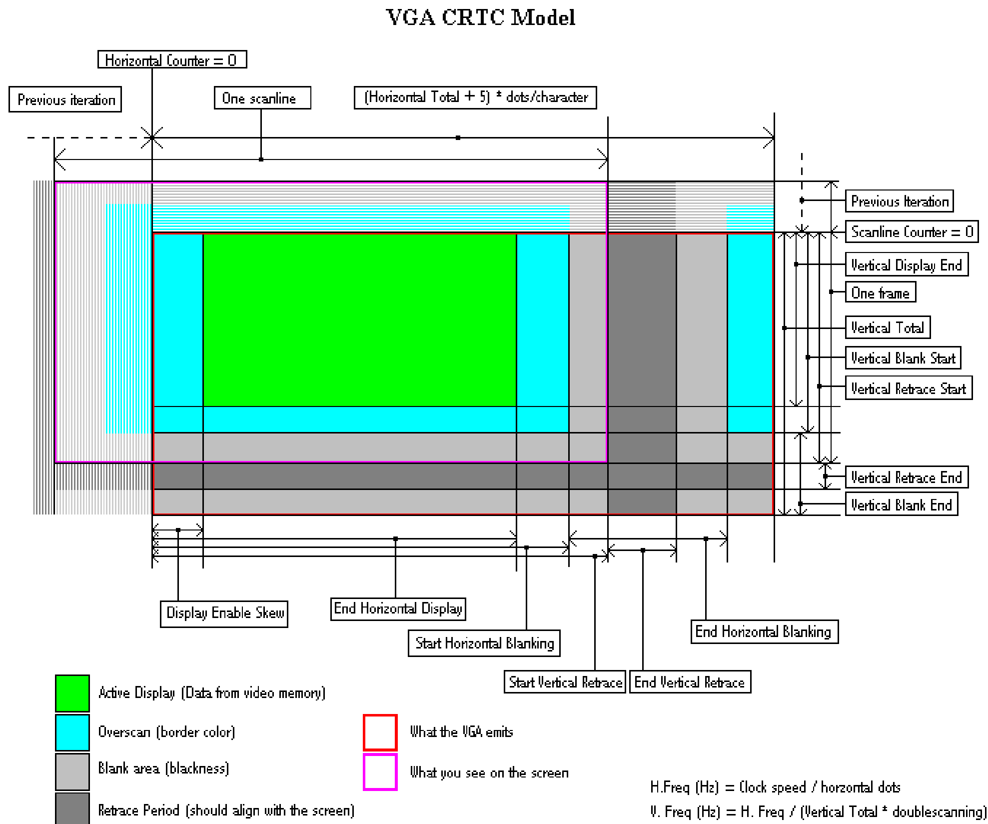
TODO

The CRT Controller

The Cathode Ray Tube Controller, or CRTC, is the unit to create a video signal from the data produced by the DAC. By programming this unit you can control the resolution of your monitor, as well as some hardware overlay and panning effects.

The following diagram gives a quick overview of how the CRTC is generally configured (with register names)

Resolution and Timing



Horizontal timings are based on character clocks (multiples of 8 or 9 pixels), Vertical timings are per-scanline. Since these easily exceed the 255 limit of one byte, the Overflow Register is used to store the high-order bits. Horizontally and vertically, the area can be separated into four parts:

- Active Display: The data rendered from memory, from the human perspective this area equals the resolution.
- Overscan: the area around the Active Display. Although commonly black, this can be made visible by changing its colour. By default this is 8 pixels in size on each end.
- Blanking Area: When the CRTC disables the colour output. This is the black area around the screen. By changing this you can move, center, and scale the screen.
- Retrace Period: This area is not generally visible. This is just a signal sent to the monitor to go to the next scanline or the next frame. However, monitors may steal some of this time for the blanking area following it or vice versa. Consequently, the blanking sizes on either side are not equal to accommodate.

Registers

Registers involved in horizontal timing:

Register Name	Port	Index	7	6	5	4	3	2	1	0
Horizontal Total Register	0x3D4	0x00	Horizontal Total							
End Horizontal Display Register	0x3D4	0x01	Horizontal Display End							
Start Horizontal Blanking Register	0x3D4	0x02	Horizontal Blanking Start							
End Horizontal Blanking Register	0x3D4	0x03		Horizontal Display Skew		Horizontal Blanking End (bits 0..4)				
Start Horizontal Retrace Register	0x3D4	0x04	Horizontal Retrace Start							
End Horizontal Retrace Register	0x3D4	0x05	H. Blanking End (bit 5)			Horizontal Retrace End				

Registers involved in vertical timing:

Register Name	Port	Index	7	6	5	4	3	2	1	0
Vertical Total Register	0x3D4	0x06	Vertical Total (bits 0..7)							
Overflow Register	0x3D4	0x07	V.Retr.St. (9)	V.Disp.End (9)	V.Total (9)		V.Blank.St. (8)	V.Retr.St. (8)	V.Disp.End (8)	V.Total (8)
Maximum Scan Line Register	0x3D4	0x09			V.Blank.St. (9)					
Vertical Retrace Start Register	0x3D4	0x10	Vertical Retrace Start (bits 0..7)							
Vertical Retrace End Register	0x3D4	0x11					Vertical Retrace End			
Vertical Display End Register	0x3D4	0x12	Vertical Display End (bits 0..7)							
Vertical Blanking Start Register	0x3D4	0x15	Vertical Blanking Start (bits 0..7)							
Vertical Blanking End Register	0x3D4	0x16	Vertical Blanking End (bits 0..6)							

Other registers dealing with timing:

Register Name	Port	Index	7	6	5	4	3	2	1	0
Miscellaneous Output Register	0x3C2	-					Clock Select			
Clocking Mode Register	0x3C4	0x01								9/8 Dot Mode

Timing Model

The horizontal timing registers are based on a unit called 'character' (As they match one character in text mode). Each character equals 8 (**9/8 Dot Mode** is set) or 9 (**9/8 Dot Mode** is clear) pixels. Each scanline contains **Horizontal Total** + 5 characters, zero based. **Horizontal Display End**tells us the last character that is calculated from memory (i.e. the horizontal resolution in characters minus one). **Horizontal Blanking Start** and **Horizontal Retrace Start** give us the the last character before either period is started. **Horizontal Blanking End** and **Horizontal Retrace End** need more explanation, as they only contain part of a number. When blanking or horizontal retrace is enabled the significant bits are checked against the character counter, and if these bits match the respective period will be ended. The quick solution is to calculate the appropriate values, compute the last character clock at which each period should be active, then AND it with 0x3F (Blank) or 0x1F (Retrace) to get the register's value. Note that the periods must be between 1 and 63(Blank)/31(Retrace) character clocks. To be safe, there must be at least one character of overscan on each side of the screen to avoid additional artefacts.
The vertical timing is similar, apart from the fact that these registers operate on scan lines (pixels) instead of characters. The **Vertical Retrace End** and **Vertical Blank End** registers work also similar, although they are different sizes. The Retrace End is 4 bits wide (AND with 0xF, period is 1-15 scanlines), The Blank End size is at least 7 bits (some say its 8, some say its 7), so the value is computed by ANDing with 0xFF, with the period ranging from 1-127 scanlines. As with horizontal timing, at least one scan line of overscan must be present to avoid possible artefacts.
The clock can be selected using **Clock Select**. Only two of four possible clocks are present on all VGAs. A clock of 25MHz is selected when this field is zero, 28MHz is selected when this field equals 1. Some boards have other clocks under values 2 and 3, but you should not write these values unless you know the clock that is there. Note that selecting the 28MHz clock and 9 pixels per pixel results in the same timings as selecting the 25MHz clock and 8 pixels per character, only with different resolutions.

The refresh rates can be calculated as follows:

- Horizontal Refresh Rate = Clock Frequency (in Hz) / total pixels horizontally
- Vertical Refresh Rate = Horizontal Refresh Rate / total scan lines

On a VGA monitor, the horizontal refresh rate should equal 31.25 kHz. Vertically, only 400 and 480 pixel resolutions are used.
If your monitor supports it, you can set virtually any crazy resolution you want. (provided the horizontal resolution is a multiple of 8 or 9 - 8 seems to be common) Most modern monitors allow anything between 400x300 and 800x600 being set this way.

Sample timing scheme

640x480 (16 colours) uses the following sizes:

- Timing: 25MHz dot clock, 8 pixels per character
- Totals: 800 pixels horizontally, (100 characters), 524 scan lines
- Active Display: 640x480 (80 characters, 480 scan lines)
- Overscan: 8 pixels (8 scan lines vertically / 1 character clock horizontally) on each side
- Horizontal Retrace: 12 characters (96 pixels)
- Vertical Retrace: 2 scan lines
- Blanking (Left): 2 characters (16 pixels)
- Blanking (Right): 4 characters (32 pixels)
- Blanking (Top): 24 scan lines
- Blanking (Bottom): 2 scan lines

Which should be VGA compatible

Sample Register Settings

These are the register values that can be loaded into the VGA to set a standard mode. Note that you should unlock the CRTC and disable output before loading these registers, and afterwards restoring these to be nice for old monitors that are around.
The pseudocode for changing modes is roughly as follows:

```
DisableDisplay   // disable output
UnlockCRTC       // unlock registers

LoadRegisters    // load registers
ClearScreen      // clear the screen contents
LoadFonts        // and for text mode, load fonts
                 // note that this may need to alter GC settings
                 // so be sure to restore those after that

LockCRTC         // optional: lock the registers again
EnableDisplay    // make sure there is output
```

List of register settings

Register name	**port**	**index**	**mode 3h** (80x25 text mode)	**mode 12h** (640x480 planar 16-bit color mode)	**mode 13h** (320x200 linear 256-color mode)	**mode X** (320x240 planar 256 color mode)
Mode Control	0x3C0	0x10	0x0C	0x01	0x41	0x41
Overscan Register	0x3C0	0x11	0x00	0x00	0x00	0x00
Color Plane Enable	0x3C0	0x12	0x0F	0x0F	0x0F	0x0F
Horizontal Panning	0x3C0	0x13	0x08	0x00	0x00	0x00
Color Select	0x3C0	0x14	0x00	0x00	0x00	0x00
Miscellaneous Output Register	0x3C2	N/A	0x67	0xE3	0x63	0xE3
Clock Mode Register	0x3C4	0x01	0x00	0x01	0x01	0x01
Character select	0x3C4	0x03	0x00	0x00	0x00	0x00
Memory Mode Register	0x3C4	0x04	0x07	0x02	0x0E	0x06
Mode Register	0x3CE	0x05	0x10	0x00	0x40	0x40
Miscellaneous Register	0x3CE	0x06	0x0E	0x05	0x05	0x05
Horizontal Total	0x3D4	0x00	0x5F	0x5F	0x5F	0x5F
Horizontal Display Enable End	0x3D4	0x01	0x4F	0x4F	0x4F	0x4F
Horizontal Blank Start	0x3D4	0x02	0x50	0x50	0x50	0x50
Horizontal Blank End	0x3D4	0x03	0x82	0x82	0x82	0x82
Horizontal Retrace Start	0x3D4	0x04	0x55	0x54	0x54	0x54
Horizontal Retrace End	0x3D4	0x05	0x81	0x80	0x80	0x80
Vertical Total	0x3D4	0x06	0xBF	0x0B	0xBF	0x0D
Overflow Register	0x3D4	0x07	0x1F	0x3E	0x1F	0x3E
Preset row scan	0x3D4	0x08	0x00	0x00	0x00	0x00
Maximum Scan Line	0x3D4	0x09	0x4F	0x40	0x41	0x41
Vertical Retrace Start	0x3D4	0x10	0x9C	0xEA	0x9C	0xEA
Vertical Retrace End	0x3D4	0x11	0x8E	0x8C	0x8E	0xAC
Vertical Display Enable End	0x3D4	0x12	0x8F	0xDF	0x8F	0xDF
Logical Width	0x3D4	0x13	0x28	0x28	0x28	0x28
Underline Location	0x3D4	0x14	0x1F	0x00	0x40	0x00
Vertical Blank Start	0x3D4	0x15	0x96	0xE7	0x96	0xE7
Vertical Blank End	0x3D4	0x16	0xB9	0x04	0xB9	0x06
Mode Control	0x3D4	0x17	0xA3	0xE3	0xA3	0xE3