CHARACTERISATION OF THE HD6845 CRT CONTROLLER

UTTERCHAOS

1. Introduction

The HD6845 is commonly used as the CRT Controller (CRTC) in IBM CGA cards and whilst its documented behaviour is well-known, there are many questions remaining about how it operates.

Exploitation of "glitches" in the CRTC can lead to interesting effects used in demos such as 8088MPH and Area5150.

Many of the internals of the 6845 can be guessed by examining the behaviour of the chip in a cycle exact way.

We can infer that the chip consists of registers Rx and a set of internal counters Cy which are compared with the values in registers Rx by comparators. Actions may occur when the comparator goes high when some Cy reaches a register value Rx and when it goes low again after reaching Rx+1.

We know such internal counters must exist, otherwise the values in the registers would be lost during operation.

Of course the possibility remains that comparators compare with half or double values or that internal counters are incremented by 2, etc.

Much of the internal construction can be inferred from block diagrams in the datasheet, from the fact that transistors were not wasted and from the fact the 6845 contained no circuitry for addition/subtraction, other than incrementing of the various internal counters.

2. Notation

We use the terminology more or less as standardised in the blog of Reenigne.

(1) hdot = time for one pixel in hires mode (640 horizontal pixels)
(2) ldot = 2 hdots = time for one pixel in 320 pixel mode
(3) cycle = 3 hdots = one processor cycle (aka ccycle)
(4) hchar = 8 hdots = one character in 80-column mode
(5) lchar = 16 hdots = one character in 40-column mode

3. Lockstep

There are four things that can operate with a relative phase to one another: the CPU, the Programmable Interval Timer (PIT), the CGA card, the CRTC.

All are driven from the same clock 14.318MHz clock (each cycle is 1 hdot), but different dividers are used on the mainboard and on the CGA card.

Note that the CRTC derives its input from the CGA card, so these are by definition in sync. Also, the CGA card timing is relevant because of the wait states it asserts on the bus if the CPU tries to access VRAM. This is to avoid conflict with the CGA card accessing VRAM as it continually strobes through CGA memory for

display and also VRAM refresh (which cannot be fully disabled, even in the blanking region).

At boot time the CGA card and PIT start with a random phase with respect to one another. As the PIT is run with a cycle of 12 hdots and the CRTC lchar is 16 hdots the best we can do is lockstep of the CRTC and PIT within 4 hdots, meaning there is an unknown phase of 0-3 hdots between the two, which can only be changed (randomly) on reboot.

This is relevant even if the PIT is not used as a system timer, as system DRAM refresh is run on timer 1. DRAM refresh can be slowed or sped up from its standard value of 18 PIT cycles (each scanline in standard modes is 76 PIT cycles exactly), or temporarily turned off.

As the CPU clock is every 3 hdots and the 6845 lchar takes 16 hdots and these are relatively prime, it is possible to align these.

A full CRT frame (262 scanlines of 57 lchars as defined by the IBM BIOS) is precisely 79648 CPU cycles and exactly 19912 PIT cycles so once aligned, in the absence of external interference, these will remain aligned in standard modes.

This process of alignment is called *lockstep* by Reenigne (a consistent phase of the CRTC and CPU with respect to one another). Reenigne's method of obtaining lockstep consists of a series of delays and VRAM accesses (causing a sequence of wait states that reduces the number of possible phases at each step).

Reenigne discovered this method by analysing the wait state algorithm of the IBM CGA card although some experimentation was apparently required to find a sequence that worked in practice.

Once lockstep is achieved, any fixed sequence of code, CGA register writes (other than certain mode changes), CRTC register writes (other than those which change the size of the CRT frame), DRAM writes and so long as there is no reboot, PIT pulses, will result in precisely the same result down to the hdot. If the PIT is not used (including for DRAM refresh) then the results will be the same even after reboot.

Of course interrupts from external devices such as the keyboard, disk drives and so on, may cause a loss of lockstep, so these are usually switched off. The keyboard can be polled by checking the IRR and accessing the keyboard buffer directly via a port. Lockstep can also be lost by changing the screen geometry of course.

Unfortunately the IBM CGA card used for the experiments in this document did not respond correctly to Reenigne's lockstep sequence (we have not yet determined the reason). Therefore we use a slightly different approach.

Similarly to the last part of Reenigne's method, we set up a CRTC frame of four horizontal two characters by two scanlines in CGA graphics mode so that there are 8 lchars per frame. We set horizontal displayed and vertical displayed to 1 so that there is exactly one lchar in the active region.

We first use Reenigne's loop of exactly 144 cycles which is exactly 27 lchars to step through lchars until we reach the active lchar. At this point there are exactly 16 phases (one for each hdot of the lchar). We then waste some cycles until we are just back into the lchar immediately following the active lchar. We then use a loop of precisely 85 CPU cycles which is 255 hdots, or one hdot less than a multiple of 4 lchars. This allows us to step back to the active lchar one hdot at a time until it is again detected. At this point we are in full lockstep.

The lockstep algorithm we use appears to be reliable. The display enable signal appears to be consistent down to the hdot, though obviously we don't know about

any delays due to internal circuitry. By the time the CPU knows about the display enable signal, some time may have passed. But at least this delay is absolutely consistent, to the hdot (modulo hardware defect).

Note that there are three possible hdot phases of the first hdot of the active region in a standard mode with respect to the CPU cycle that occurs at that time (0, 1 or 2 hdots).

To precisely control the phase we again step a fixed number of times by 85 cycles to get a phase of 0, 1 or 2 hdots as desired.

4. Limitations

In order to determine what time various register writes take effect, and the precise time hdot marginal behaviour is exhibited, there are three methods that could be used.

- Assume the CRTC is a black box and analyse any delays in the CGA card and system board schematics
- Assume the CRTC is a black box and instrument the system with a sniffer
- Determine timings from their visual effect

In this document we have decided to take the third approach. However, in order to do so, there must be some reference event that is visible that we can compare everything to.

As CGA background register writes take effect down to the hdot (visible using an RGB2HDMI adaptor with a large LCD monitor) and probably take effect with very little delay, we decided to give all timings with respect to when a background register change causes a visible change onscreen.

To this end, the precise hdot where the colour change becomes visible for the first time is taken to be the time that the background colour register change took effect. Of course this subsumes a constant delay due to internal CGA circuitry which is unknown.

To write a CRTC register takes one byte to be sent to port 3D4H to specify the CRTC register and another byte to be sent to port 3D5H to specify the value. Obviously the register value cannot change until the second byte is received, and again we don't know about any internal delays.

It's conceivable that the delay depends on which bits are changed or on whether the value is being changed *to* a value that corresponds to a character position about to be encountered imminently or *from* such a value. We refer to the latter as a *to-imminent* or *from-imminent* changes, respectively.

Each write of a byte to a port takes approximately 4-5 lchars including the time to load the port address and value into CPU registers (the precise number of cycles depends on at least the state of the CPU prefetch, bus activity and whether it has been idle, interrupts and possibly other things). Thus a write of both bytes to change a CRTC register is a relatively slow process.

In light of unknown constant delays associated with both background register writes and CRTC register writes we cannot talk in absolute terms about when these took place. However, we can talk about when one occurs relative to the other.

Obviously the relative delays will depend on model of video card, but the timings we give in this document are for one of the two IBM CGA cards which behave identical up to differences in the CRTC chip.

In order to measure the relative timings of various effects that occur on a CRTC register write we first do *two* background register changes (each of which can be done with a write of a single byte to a CGA port), e.g. we change the background colour to red on the first write and then back to black on the second write. We then use exactly the same code, but change the two background register writes to a single CRTC register write (recall this takes two bytes to be written to ports).

There are no wait states associated with any of these writes on the IBM PC, therefore from the CPU's perspective the code will take precisely the same number of cycles in both cases when this exchange is made in the code.

Obviously we are interested in when the CRTC register change takes effect, and we *define* it to have been written when the second background colour change was visible onscreen (which we will have previously determined before the code was switched).

What we are interested in is when a CRTC register value has to be written (as measured in the way just specified) in order for it to take effect, or to exhibit some glitch or other marginal behaviour.

In this way we aren't really measuring when such register writes take effect because of the unknown internal delays, but we have a relative reference point with which to compare.

This method can't tell us anything about any constant internal delays, but it can conceivably tell us about any delays that depend on which bits are changed or on whether a to-imminent or from-imminent change is being made.

Obviously if instrumentation or circuit analysis or simulation subsequently reveal something about absolute delays associated with a background register write then we can use the relative information recorded in this document to say something in absolute terms about when the CRTC register writes actually take effect. However, from the programmer's perspective this information isn't needed to be useful.

5. The Horizontal Displayed Register

This controls the number of characters that are displayed horizontally on each scanline, but many questions remain.

(1) When does the hdisp register value take effect?
(2) Is there any edge case behaviour?
(3) What resets the count to zero?
(4) What triggers when the comparator goes high/low?
(5) What happens if it is set multiple times in a scanline?
(6) What is its behaviour near other register values, such as hsync pos, and htotal?
(7) When is the horiz. disp. value added to the address?

5.1. **Methodology.** The program HDISP1.ASM does the following to achieve lockstep:

(1) Enter CGA graphics mode (equivalent of BIOS mode 4)
(2) Set up a CRTC frame with 2 scanlines of 4 lchars, with one lchar in the active region
(3) With a cycle exact loop of 144 cycles $= 4n - 3$ lchars ($n = 7$) wait until in the active region
(4) Waste some cycles until back in the inactive region
(5) With a cycle exact loop of 85 cycles, wait until in the active region again

This puts the CPU in a known state with respect to the CRTC down to the hdot. As far as we can tell, this is entirely consistent between runs if DRAM refresh is disabled.

In order to handle DRAM refresh the program executes 256 consecutive bytes of code in each 64kb block of system RAM, repeated four times per CRT frame. This causes those 256 bytes to be read approximately every 4ms.

The 256 bytes per 64kb block are alternated between offset 0 and offset 256. The reason for this is that on the XT 4164 and 41256 DRAM chips are typically used. The 4164's require refresh every 4ms, or roughly four times per CRT frame. The 4164's are 64kbits and the 41256's are 256kb. The former will be refreshed if reads are done on 256 rows (corresponding to the low 8 address bits) and the latter require reads on 512 rows (corresponding to the low 9 address bits).

The alternating behaviour of the reads from offset 0 to offset 256 ensures that even for the 41256 chips, all 512 rows are energised.

To execute code in all 64kb blocks, small segments of 256 bytes in each block are saved by our program and overwritten by NOPs and a RETF. These are replaced before returning to DOS.

Note that this method does not work on early PCs, as these used 16kbit 4116 chips which need refreshing every 2ms. As there are so many of them in a 640kb system, there isn't time in a CRT frame to refresh them all.

The program turns off the keyboard and timer interrupts (the only ones firing on a quiet XT) and polls the IRR to see if a keyboard IRQ has occurred. If this is detected, the entire program restarts, usually modifying some program values before doing so, in response to the keypress. Lockstep is re-run on every restart to ensure consistent behaviour between runs.

The main loop can be adjusted to a precise number of cycles and defaults to the exact number of cycles for a standard CGA CRTC frame.

The code for changing background colour/CRTC registers and for polling the keyboard is interleaved with the refresh code so that there is no interference between DRAM refresh and these other operations.

The register writes always happen at the top of the main loop, however their exact position relative to the CRTC frame down to the hdot can be changed by altering a delay after lockstep and before the start of the main loop to adjust the cycle the loop starts on, and an additional delay of some multiple of 85 cycles after lockstep but before a normal CRTC frame size is reestablished in order to control the hdot phase of 0-2 hdots.

Longrunning instructions such as SHL (with a high count) and MUL are used for timing within 4 and 1 cycles respectively.

The program can be made to switch from background register writes to a horizontal displayed (hdisp) register write and back again. When doing background register writes the program displays pixels on every second scanline, with alternating on/off pixels, starting with the first on each scanline. This allows the background colour changes to be observed when measured against the coloured pixels in the neighbouring scanlines.

Different coloured pixels are used in each lchar in a repeating pattern.

When doing CRTC register writes all pixels on the screen are filled (again repeating a pattern to delineate the lchars) except for every tenth scanline which is left black. The latter is done to make it easier to see when the CRTC has stopped updating the VRAM address at the end of each scanline.

Counting is done on a large LCD monitor after conversion using an RGB2HDMI adaptor, which makes every hdot visible.

5.2. **When does the hdisp register value take effect?** For a to-imminent change from a higher hdisp value we see three behaviours:

(1) The new hdisp value (n) takes full effect, blanking all but the first n characters
(2) An intermediate behaviour where the new hdisp value only takes effect from the second half of the lchar where it is supposed to start blanking, leaving an extra hchar displayed before blanking the remainder of the scanline
(3) The hdisp value does not take effect, causing all characters to be displayed right out to htotal+1 characters, with the exception of the final character which only displays the first hchar of the full lchar, the right hand hchar being blank

These behaviours occur when the write happens on the fourth last, third last or second last hdot before the imminent character, respectively.

A to-imminent change to a strictly greater value has no visible effect on the current scanline, as we are already blanking at that point, so it continues to do so. However, there is a revealing behaviour.

So long as the higher value is written to the register strictly before the second last hdot of the higher hdisp is reached (and we are on the last scanline of the character row), the address will be updated to the higher value, even though characters have not been displayed since the lower hdisp value was reached.

This behaviour occurs so long as the higher value is written to the register strictly after the fifth hdot before the lower value is reached.

A from-imminent change to a lower hdisp value causes the third behaviour in the list above if the change is made early enough (strictly before the 8th hdot of the character). As the hdisp value is never reached in this case the address is not updated.

As might be expected, a from-imminent change to a higher hdisp value results in no unusual behaviour (other than the address incrementing behaviour just described). If the change is made in time the larger value for hdisp is used, otherwise the smaller value continues to be used.

However, the cutoff between the two cases changes according to a very intriguing rule which we have verified in at least 3 HD6845 chips (and not in other CRTC chips).

Generally, the larger value must be written strictly before the 8th hdot of the character before the smaller value is reached. However in very specific cases there is an extra one hdot delay meaning that the write must be done one hdot earlier to take effect.

If we are making a from-imminent change of hdisp from m to some $n > m$ then the extra hdot delay exists if all of the bits of m which are set are also set in n, with the exception of bit 0 which does not affect timings.

Note that it doesn't make any difference if additional bits of n are set. The only ones that affect the timing are those corresponding to bits set in m (with the exception of bit 0 of course).

We have not been able to speculate what causes this unusual timing behaviour, but it has been very carefully measured.

5.3. **Is there any edge case behaviour?** If hdisp is set strictly greater than htotal the final character of the scanline is only half displayed; the second hchar of that final lchar is blank.

This is probably a general behaviour of this CRTC (the final hchar before the end of the scanline is blanked).

5.4. **What resets the count to zero?** Unknown, however a single internal counter is likely used for comparison against the hdisp, htotal and hsync pos registers, and this horizontal counter is needed for comparison against htotal, so we know the counter can't be reset before htotal is reached. It must be 0 by the time htotal+1 is reached as this is the start of the next scanline where hdisp may be 0, so the counter must be 0 by that point.

5.5. **What triggers when the comparator goes high/low?** The address update for the start of a scanline (at least the active ones) seems to be triggered by the hdisp comparator. But this only happens if hdisp is reached before the horizontal counter reaches htotal+1 characters, otherwise the start of scanline address is not advanced (see below).

Interestingly, if hdisp is not reached on the final (active?) scanline of the frame the start of scanline address is not reset to the start address but continues to be increased in the next frame.

This is slightly surprising as one might have imagined that this happening at the end of the scanline on which the horizontal total adjust value is reached.

The exact sequence of events that trigger start of scanline address reset to the start address is not clear, but it seems likely that hdisp being reached is a component of it, just as it is for update at other times.

It's unlikely that it's triggered by the vertical total adjust counter reaching the vertical total adjust value as this value could be zero. The associated counter is likely only 5 bits meaning that it isn't counting all the time and is therefore probably zero throughout much of the frame.

It is possible that the reset to start address happens at the end of the active region, but this remains to be confirmed.

5.6. **What happens if it is set multiple times in a scanline?** The program HDISP2.ASM allows two updates to hdisp to be made on the same scanline. Both values can be set independently.

All behaviour appears to be in line with the following:

- Whenever hdisp is reached on the last scanline of a row, the address is updated.
- Reaching hdisp the first time begins blanking and this continues to the end on the scanline regardless if hdisp is reached again or not
- If hdisp is not reached for a first time the entire scanline is filled (with the exception of the last hchar as usual), but the address is not updated

5.7. **What is its behaviour near other register values, such as hsync pos, and htotal?** See the other sections for behaviour near htotal.

There appears to be no special behaviour when changing hdisp near hsync pos.

5.8. **When is the hdisp value added to the address?** If the hdisp value is strictly greater than htotal the address is not updated, otherwise it is updated on the last scanline of a row when hdisp is reached.

6. The Horizontal Total Register

This controls the number of characters horizontally in the scanline. The actual number of characters is always one more than the value in the register. One can speculate that this is because there are tasks that the CRTC needs to do in the final character of each scanline and so the comparator needs to go high one character before the end of the scanline.

There are a number of questions we'd like to answer about the horizontal total register.

(1) When does htotal take effect?
(2) Is there any anomalous behaviour?
(3) What is triggered by the comparator going high?

6.1. **Methodology.**

There are only so many values that we can easily set the horizontal total to. Typically a CRTC frame is 57 lchars wide. Monitors tend to tolerate values very close to this.

One consideration is that if the number of characters in a scanline isn't divisible by 3 there will not be an whole number of CPU cycles in a frame unless the number of scanlines is divisible by 3. We can ensure this is the case by adjusting the vertical total adjust.

The program HTOT1.ASM sets the width of the screen to 56 lchars and for three rasterlines sets the size of a scanline to 28 lchars. In order to compensate for three extra scanlines being used up in this region, we have to add an additional three scanlines to the vertical total adjust, as well as round it to a multiple of 3.

The program HTOT2.ASM sets the width of the screen to 57 lchars and for three rasterlines sets the size of a scanline to 19 lchars. Once again we have to compensate for the extra scanlines being used up in this region by adding 6 in this case to the vertical total adjust.

The program HTOT3.ASM sets the width of the screen to 57 lchars and for two rasterlines sets the size of a scanline to 38 lchars. There's now just one additional scanline over the normal vertical adjust that we need to add to compensate.

6.2. **When does htotal take effect?**

Without exception it seems that htotal can be written up to 18 hdots before the scanline we want to have length htotal+1. However it does not need to be set before the scanline starts. So long as it is set strictly before 18 hdots ahead of the next scanline (both the existing one and the one we are setting up), it will still take effect.

Setting htotal after this point will cause a wraparound of the horizontal counter (which counts to 255 before wrapping around to zero) and for the counter to keep going until it encounters the new htotal value after the wraparound. This very likely results in loss of sync.

6.3. **Is there any anomalous behaviour?**

If one sets htotal to a smaller value than hdisp for the last scanline, the address will not be reset to the start address for the beginning of the next frame. This causes the address to keep incrementing through VRAM even as we go from frame to frame.

6.4. **What is triggered by the comparator going high?** Presumably the horizontal counter is reset to zero ready for the next scanline.

Likewise it seems probable that the row address is incremented.

It's unknown whether the horizontal sync counter could be reset at this time.

Precisely when these events happen hasn't been determined, and we don't know if they occur when the comparator for htotal goes high or low.

7. Start address register

This controls the address that the CRTC is reset to before the new frame is drawn. In the CGA card, each increment of the start address corresponds to an increment of two bytes of VRAM (one character).

As we've already begun to speculate about when the start of scanline address is reset to the start address, it seemed worthwhile figuring out when this happens.

The questions we'd like to answer are as follows:

(1) Can the start address be written anywhere in the frame?
(2) For which frame does the new start address take effect?
(3) Precisely where does the start address need to be written by?
(4) What likely triggers the start of scanline address to be reset to the start address?

7.1. **Methodology.** The file ADDR1.ASM divides the CRT frame up into two CRTC frames in the vertical direction, each exactly half of the CRT frame in size.

The start address is changed twice per CRT frame, exactly half a CRT frame apart, i.e. in exactly the same place on the two CRTC frames.

As well as changing the start address we also follow that up by a change of the vertical sync position. In the top CRTC frame the vertical sync position is changed to a value that is never reached so that there is no vertical sync associated with this frame. This ensures that the only vertical sync is the one that happens in the bottom CRTC frame.

The file ADDR2.ASM does the same as ADD1.ASM except that after changing the start address and vertical sync position it then immediately changes hdisp. In the top CRTC frame it is changed to be greater than htotal and in the bottom CRTC frame it is changed back to a normal value.

This allows us to check which scanlines the reset to start address occurs. If it were set on more than just the final scanline of the CRTC frame then we'd see the start address resetting despite the invalid hdisp never being reached on the final scanline of the frame.

Beware that if the register write position is moved more than half way down the CRT frame, the vertical sync will end up occurring on what was previously the top CRTC frame causing the monitor to briefly lose sync and readjust with the two frames switched.

7.2. **Can the start address be written anywhere in the frame?** It appears that it can be written anywhere in the frame.

7.3. **For which frame does the new start address take effect?** The start address takes effect for the next frame to start, roughly speaking. The precise details are given in the next section.

7.4. **Precisely where does the start address need to be written by?** The start address needs to be written strictly before the second last hdot before hdisp is reached on the scanline before the relevant frame starts.

This confirms what we saw previously, that if hdisp is not reached on the final scanline of a frame, the start of scanline address is not reset to the start address at the beginning of the next frame.

7.5. **What likely triggers the start of scanline address to be reset to the start address?** Based on the information in the previous section it is extremely likely that the hdisp comparator going high on the final scanline of the CRTC frame is what causes the start of scanline address to be reset to the start address.

When we say "final scanline of the CRTC frame" we are not referring to the active region, but the entire frame.

This is mildly surprising because it's not immediately clear how the CRTC recognises that it is on the final scanline of the frame. If there is a vertical adjust period then the final scanline of the frame is certainly nothing to do with vtotal.

Also, if vtotal adjust is zero then it cannot simply be waiting until the vtotal adjust counter reaches the value in the vtotal adjust register, as that value will be zero for most of the frame.

One possibility is that a value of zero in this register causes problems, but there is no mention of this in the documentation.

As the vertical displayed value must be less than the total number of vertical characters it might simply be that the start of scanline address is reset when hdisp is reached on *every* scanline once vtotal is reached. Then so long as the start address is set before the last time this happens, all will be well.

However, tests with our second program in this section confirm this is not the case. The reset to start address occurs only on the final scanline of the CRTC frame and only if and when hdisp is reached on that scanline.

Some fairly complex condition must indicate that the final scanline of the frame has been reached. Hopefully this will be revealed when we study the vertical adjust register.

8. Horizontal sync

There are two registers associated with horizontal sync: the horizontal sync position and the horizontal sync length registers.

The first of these registers sets the position that the horizontal sync pulse will be emitted, along with a blank region whose number of characters is given by the second register.

We'd like to find out the following about the horizontal sync.

- What position does the horizontal blanking region appear?
- When does the hsync register have to be set?
- Is there any marginal behaviour of hsync?
- Can more than one sync region appear on the same scanline?
- What resets the hsync counter?
- Are any things triggered by the hsync counter?

8.1. **Methodology.** To make things a little easier to see on the screen, we increased the standard value of the horizontal sync length by two characters and decreased the horizontal sync position by two characters.

The program HSYNC1.ASM sets the horizontal sync position temporarily to a specified value, then precisely one scanline later sets it back to the the value mentioned above. Immediately after setting it back, it also sets the hsync width to a specified value which by default is the value mentioned above, but can be decremented at will.

The program HSYNC2.ASM is as per HSYNC1.ASM except that instead of returning the hsync position back to its initial value one scanline later, it does it on the same scanline. This will allow us to see if the hsync can fire twice in the same scanline.

8.2. **What position does the horizontal blanking region appear?** Interestingly, the horizontal blanking begins one hchar before the position corresponding to the hsync value is reached. It's not that the machine travels back in time, but obviously there are some delays in the CGA card, e.g. to retrieve graphics information from memory, propagation delays, delays to prevent simultaneous access of the CPU and CGA card to VRAM, etc. This means that the background colour changes and the display of pixels from graphics memory actually happens later than signals on the pins of the CRTC might suggest.

Horizontal blanking actually continues across scanlines. In fact, one can set any value up to and including htotal as the hsync position. If it is set higher than that it will not be reached on a scanline.

8.3. **When does the hsync register have to be set?** The hsync position must be set strictly more than 2 hdots before the position in question. However, extra time is required if it is already about to initiate a horizontal sync at the next character.

If one is pending, it will commit to this 8 hdots earlier than the cutoff just mentioned, i.e. a total of 10 hdots before the pending character and one would need to set the register before it commits itself in order to ensure the upcoming hsync doesn't happen.

We speculate that the CRTC clock goes high for the 8 hdots between the two points and that the comparator going high at any time during this 8 hdots flips an internal flip-flop to say that hsync must occur at the next character.

Thus if the next character matches the position already in the register at the start of those 8 hdots it is committed to an hsync regardless of any changes to the register value. If the comparator isn't already high, any change of the register to a value matching the next character during that 8 hdot period will send the comparator high and likewise commit it to an hsync.

In addition to this, the hsync pos register seems to exhibit the same timing delay of 1 hdot if a from-imminent change to a higher value with at least the same 1 bits set is made. This is not surprising, as the same register and comparator circuitry is probably used, though this is a smaller register than the hdisp case, so it wouldn't have been inconceivable that the same delay not be observed here.

8.4. **Is there any marginal behaviour of hsync?** In fact, if one sets the hsync pos precisely three hdots before the specified character, the CRTC will start blanking one hchar late. Thus an extra hchar will be displayed than would otherwise be the case.

8.5. **Can more than one sync region appear on the same scanline?** It appears that the horizontal sync blanking can happen more than once on a scanline. So long as the hsync pos. value is reached multiple times, the horizontal sync counter will begin multiple times and there will be multiple blanking intervals on the scanline.

8.6. **What resets the hsync counter?** We can infer from the previous section that the hsync counter resets itself when it reaches its terminal count. This enables the hsync to fire multiple times on a scanline.

8.7. **Are any things triggered by the hsync counter?** It's is difficult to say with any certainty what if anything is triggered by the hsync counter.

One might assume something is trigger by it because vertical sync is lost if one sets the horizontal sync so that it doesn't finish before the end of the final scanline before the vertical blanking period.

It's not entirely obvious what's going on here as one would expect hsync to not be reached and for the vertical sync pulse to be unaffected.

According to Reenigne the CGA card rather than the CRTC delays a vsync pulse until the next hsync pulse occurs. This means there is a delay that wouldn't otherwise exist meaning the next frame starts one scanline late.

This doesn't completely explain the phenomenon as the delay to the start of the next frame means that the program will effectively be doing the hsync pos register write one scanline earlier on the next frame, which should no longer be a problem. However a continuous slow loss of vertical sync is observed in practice. This remains unexplained.