3. Actually calculating a mode

If you look at the fbdev driver you think yikes. Yes, it's complex, but not as much as you think. A side note about standard modes -- It's a common misconception that graphics cards cannot do anything but the VGA/VESA induced "standard" modes like 640x480, 800x600, 1024x768, 1280x960, ... With most cards, about any resolution can be programmed, though many accelerators have been optimized for the above mentioned modes, so that it is usually NOT wise to use other widths, as one might need to turn OFF accelerator support. So if you write a driver, don't cling to these modes. If the card can handle it, allow any mode.

Here, the type of monitor has a big impact on what kind of modes we can have. There are two basic types of monitors: fixed frequency (they usually can do multiple vertical frequencies, though, which are much less critical, as they are much lower) and multifrequency.

3.1 Fixed frequency monitors

The monitor manual says the horizontal frequency (hfreq) is 31.5 kHz.

We want to display a given mode, say 640x480.

We can now already determine the absolute minimum dotclock we need, as

    dotclock = horiz_total * hfreq

and

    horiz_total = width + right_border + sync + left_border > width

The minimum dotclock computes to 20.16 MHz. We will probably need something around 20% "slack" for borders and sync, so let's say we need about a 24MHz clock. Now we look at the next higher clock our card can handle, which is 25.175 MHz, as we assume we have a VGA compatible card.

Now we can compute the horizontal total:

    horiz_total = dotclock / hfreq = 799.2

We can only program this as multiples of 8, so we round to 800. Now, we still need to determine the length and placement of the sync pulse, which will give all remaining parameters.

There is no clean mathematical requirement for those. Technically, the sync must be long enough to be detected, and placed in a way that the mode is centered. The centering issue is not very pressing anymore, as digitally controlled monitors are common, which allow to control that externally. Generally, you should place the sync pulse early (i.e. keep right_border small), as this will usually not cause artifacts that would arise from turning on the output again too early when the sync pulse is placed too late.

So, if we as a "rule-of-thumb" use a half of the blanking period for the sync and divide the rest as 1/3 right-border, 2/3 left border, we get a modeline of:

"640x480" 25.175 640 664 744 800 ??? ??? ??? ???

While this is not perfectly the same as a standard VGA timing, it should run as well on VGA monitors. The sync is a bit shorter, but that shouldn't be a real problem.

Now, for the vertical timing. At 480 lines, a VGA monitor uses 60Hz.

hfreq = vfreq * vert_total

This yields a vert_total=525. The vertical timings usually use much less overhead than the horizontal ones, as here we count whole lines. This means much longer delays than just pixels. 10% overhead will suffice here and we again split the borders 1/3: 2/3, with only a few lines (say 2) for the sync pulse, as this is already much longer than a HSYNC and thus easily detectable.

"640x480" 25.175 640 664 744 800 480 495 497 525

Let us compare that to an XF86 modeline that claims to be standard VGA:

Modeline "640x480" 25.175 640 664 760 800 480 491 493 525

Not much difference - right? They should both work well, just a little shifted against each other vertically.

3.2 Multiscan monitor

Here we will consider a theoretical monitor that can do hfreq 31-95kHz and vfreq 50-130Hz. Now, let's look at a 640x480 mode. Our heuristics say, that we will need about 768x528 (20% and 10% more) for borders and sync. We want a maximum refresh rate, so let's look what limits the mode:

```
    hfreq = vfreq * vtotal = 130 * 528 = 68.6 kHz

Oh - we cannot use the full hfreq of our monitor... well no
problem. What counts is the vfreq, as it determines how much
flicker we see.

O.K. - what pixelclock will we need?

    pixclock = hfreq * htotal

The calculation yields 52.7MHz.

Now we look what the card can do. Say we have a fixed set of
clocks. We look what clocks we have close by. Assume the card can
do 50 and 60 MHz.

Now we have a choice: We can either use a lower clock, thus
scaling down the refresh rate a bit (by 5% ... so what...): This
is what one usually does.
```

Or we can use a higher clock, but this would exceed the monitor specifications. That can be countered by adding more border, but this is usually not done, as it is a waste of space on the screen. However, keep it in mind as a trick for displaying 320x200, when you do not have doubling features. It will display in a tiny window in the middle of the screen, but it will display.

O.K. - what will our calculation yield?

```
"640x480"    50   640 664 728 768   480 496 498 528 #
65kHz 123Hz
```

I just mentioned doubling features. This is how VGA does 320x200. It displays each pixel twice in both directions. Thus it effectively is a 640x400 mode. If this would not be done, you would need a pixelclock of 12.59MHz and you would still have the problem of needing a 140Hz refresh, if hsync should stay at 31.5kHz.

A horizontal doubling feature allows us to use the 25.175MHz clock intended for 640, and a line-doubling feature keeps the vsync the same as 400 lines. Actually the line-doubler is programmable, so you can as well use modes as sick as 640x50.

O.K. - another example. Same monitor, 1280x1024.

Now we need about 1536x1126 total (same rule of thumb). That yields 130Hz*1126lines = 146 kHz. We would exceed the hfreq with that, so now the hfreq is the limit and we can only reach a refresh rate of about (95kHz/1126) 84 Hz anymore.

The required clock is now 146MHz. That would yield:

```
"1280x1024"   146   1280 1320 1448 1536   1024 1058 1060
1126 # 95kHz 84Hz
```

Now the clock might be programmable, but keep in mind that there may be limits to the clock. **DO NOT OVERCLOCK** a graphics card. This will result in the RAMDAC producing blurry images (as it cannot cope with the high speed), and more importantly, the RAMDAC will OVERHEAT and might be destroyed.

Another issue is memory bandwidth. The video memory can only give a certain amount of data per time unit. This often limits the maximum clock at modes with high color depth (i.e. much data per pixel). In the case of my card, it limits the clock to 130MHz at 16-bit depth, which would produce:

```
"1280x1024"   130   1280 1320 1448 1536   1024 1058 1060
1126 # 85kHz 75Hz
```

This is pretty much what my monitor shows now, if I ask it.

3.3 Recipe for multisync monitors

a) Determine the totals by calculating htotal = width*1.2 and vtotal = height*1.1.

b) Check what limits the refresh rate by calculating vfreq2 = hfreqmax/vtotal. If that exceeds vfreqmax, the limit is on the vfreq side and we use vfre = vfreqmax and hfreq = vfreqmax*vtotal. If it doesn't, the mode is limited by hfreq and we have to use vfreq = vfreq2. Note, that if this is smaller than vfreqmin, the mode cannot be displayed. In the vfreq-limited case, you might exceed hfreqmin, which can be countered by using line doubling facilities, if available. You can also add extra blank lines (increase vtotal) as a last-resort alternative.

c) Now that you have hfreq and vfreq, calculate the pixel clock using pixclock=hfreq*htotal. Use the next lower pixel clock. If you are below the lowest clock, you might want to try horizontal doubling features or you will have to pad the mode by increasing htotal.

d) Again, check the monitor limits. You might be violating lower bounds now... In that case you might need to resort to choosing a higher clock and padding as well.

e) You now have pixclock, width, height, htotal and vtotal. Calculate the sync start positions: hss=width+(htotal-width)/2/3; vss=height+(vtotal-height)/3. Make sure to properly align them as required by the video card hardware. hss usually has to be a multiple of 8.

f) SyncEnd is calculated similarly: hse=hss+(htotal-width)/2 and vse=vss+2.

3.4 Recipe for Monosync

a) Calculate the number of lines. As hfreq and vfreq are given, vtotal is fixed: vtotal=hfreq/vfreq. If there are multiple vfreqs allowed, choose them according to your desired vtotal (i.e. one that gives the lowest vtotal above what you really need).

b) Calculate the pixelclock. pixclock=hfreq*htotal. htotal starts at the same estimate (width*1.2) we used above.

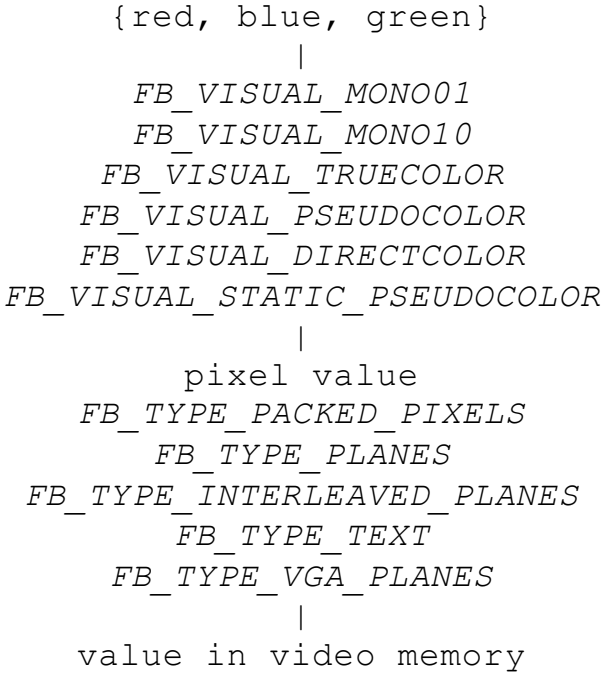
c) Adjust the pixelclock to hardware-limits. Adjust **UP**. Now recalculate the htotal=pixclock/hfreq.

d) Go to 3.3.

An important final word. Most video card documentations give you the exact equations needed to set a mode. Here, we give approached values. Use the exact values given in the documents.

3.5 Colors

There exist an endless number of colors, but colors have a special property. If you take two colors and mix them together you get a different color. There exist many models to represent colors, but fbdev is based on what is known as the RGB color model. Out of all the colors, there exist three colors in this model which when mixed in different amounts can produce any color. These colors are known as primary colors. There does exist a physical limit to mixing colors. If you take and mix red, green, and blue in equal amounts you get gray. Now if you make each color component equally brighter, the final color will become white. Now their reaches a point when making each component brighter and brighter you will still end up with white. You can increase the intensity of a color component by any amount from some initial value up to this physical limit. This is where the image depth comes in. As you know, on most cards you can have an image depth from one bit to 32 bits. Image depth is independent of the mapping from the pixel to the color components. It is also independent of the memory layout to pixel mapping. Note that some cards have a complex mapping from the pixel values to the color components (red, blue, green) as well as video memory to pixel mapping. If this is the case, you will have to consult your documentation on your video card to see what the mapping exactly is. Here are the mappings defined from top to bottom in fbdev starting with the value of the color components:



The way fbdev tells what this video memory to pixel mapping is, is with the type field in *fb_fix_screeninfo*. Here, I'm going to describe the *FB_TYPE_PACKED_PIXELS* layout since it is the most common. Here, there exists a direct mapping from video memory to a pixel value. If you place a 5 in video memory then the pixel value at that position will be 5. This is important when you have

memory mapped the video memory to userland. Next, we consider the mapping from a pixel value to the colors. This is represented in the fbdev API by the visual field in *fb_fix_screeninfo*. As you can see from the above diagram, this mapping is independent from the mapping from video memory to pixel value. This means that *FB_TYPE_PLANES* could have *FB_VISUAL_MONO01* just like *FB_TYPE_PACKED_PIXELS* can.

To understand visuals, we have to go back to the first types of video hardware. In the beginning there was just monochrome monitors. Here we could only display 2 colors: the foreground and background color. Traditionally, these colors are black and white, but if you are old enough, you would remember the old green monitors. In the fbdev API, we define two types of monochrome modes. The difference between the two is that the foreground and background colors are swapped.

Then computers began to support color. The only problem was they could only display a small number of colors at one time. What if you wanted to have an application display a certain set of colors. Well, the way that was developed to get around this was the idea of a color map. A color map translated a pixel value to the colors needed. If your application needs a specific set of colors it would switch the color maps and you would get the needed colors. Of course, this also switches the other colors in the applications. That was the trade off. This became what is known as pseudocolor mode. In the fbdev API, there exist two types of pseudocolor mappings -- A static one and a dynamic one.

FB_VISUAL_STATIC_PSEUDOCOLOR defines a video card that has a non-programmable color map. What colors you get are what you are stuck with. The other type of color map can be changed.

In time, video cards started to support more colors but this required having a larger color map. Also, video memory prices started to drop and video cards began to sell with more of it. To properly support 256 color intensity levels for each color component, you would need a color map of 16 million colors. New mappings where developed in which specific fields of a pixel where directly proportional to the intensity of a color component. Two types of mappings were developed -- One being truecolor and the other directcolor.

In truecolor, you cannot change the mappings from the pixel value to color intensities. Setting a value of 64 to the red component of the pixel will result in a red intensity of 64. How bright of a red this is depends on the image depth. For directcolor, you can control this. You could make a pixel value in the red field of 64 equal 128 for the intensity. Also some cards support an alpha value, which is used in higher graphics, which for fbdev is of little importance. It should always be set to the highest value it can have. For most cards, alpha shows up for 15-bit

modes where each color component can have up to 32 intensity levels (2^5) and one bit represents the alpha component. It also shows up for 32-bit modes where each component red, blue, green, and alpha are given 256 intensity levels (2^8). 24-bit mode is like 32-bit mode except it lacks the alpha component. An important note is that some cards only support 24-bit mode on certain architectures.

index back forward