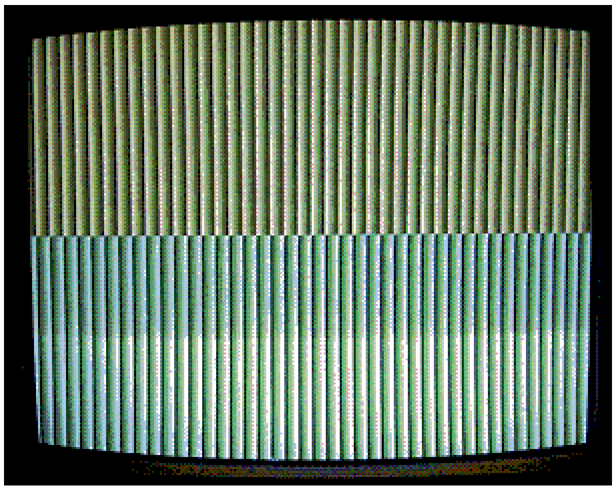
A 50 line graphics driver for the HYDRA

1.0 Introduction

This chapter is aimed at the beginner who wants to understand the basic idea of how to write a graphics driver.
To write a driver in only 50 lines of code is quite a challenge in itself, it can be done, but it leaves very little code to spare for anything useful. The good news is that there are no “fillers” or “extra code” in the driver which makes it easier to understand.
If you have not done so already, open up “BAM_50_Line_Driver_01.spin” and give it a try.
The driver was designed to do something that is easy to understand and at the same time introduce a few basic concepts.
It outputs 188 pixels per scan line in four shades of yellow on the top part of the screen and four shades of green on the bottom part. The shades of colors are then animated by shifting the colors around.
This will introduce concepts such as line loops for the split screen, tile loops for the 188 pixels that are divided in 47 tiles and animation for the scrolling colors.
Do not feel overwhelmed, this is still on a beginner level.

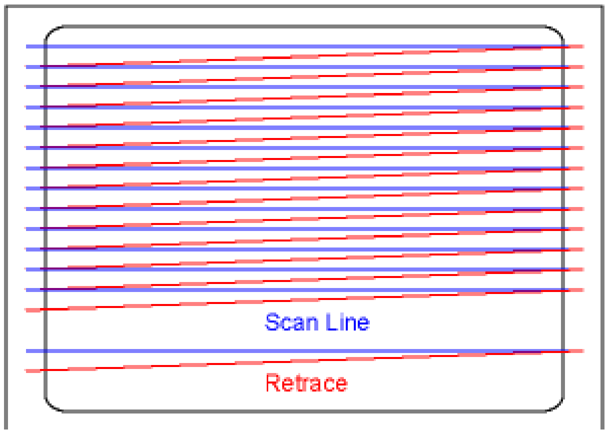
Figure 1.0 – The video driver.



1.1 NTSC introduction for the video driver

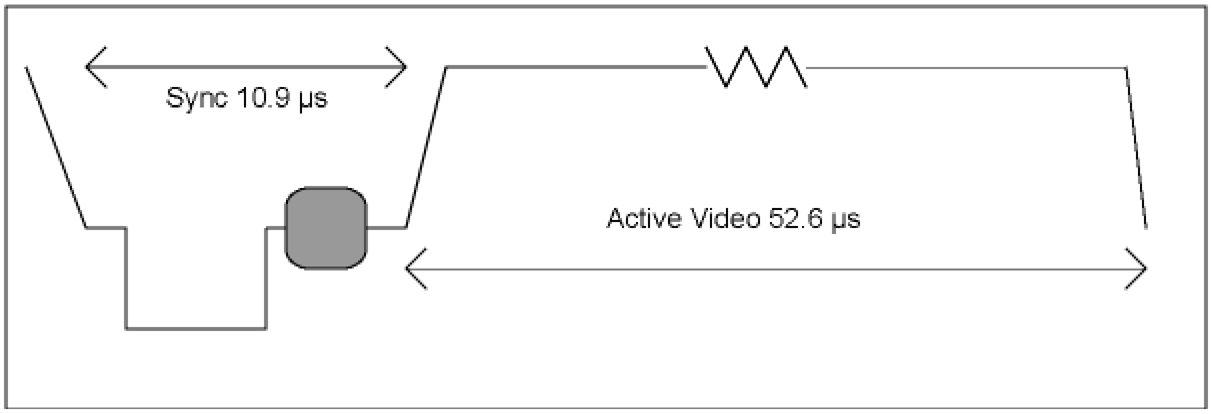
This is an introduction to the NTSC standard and pretty much all you need to know to get started. Some details have been left out and will be explained later so you can concentrate on the overview rather than the details.

Figure 1.1 – Scan lines.



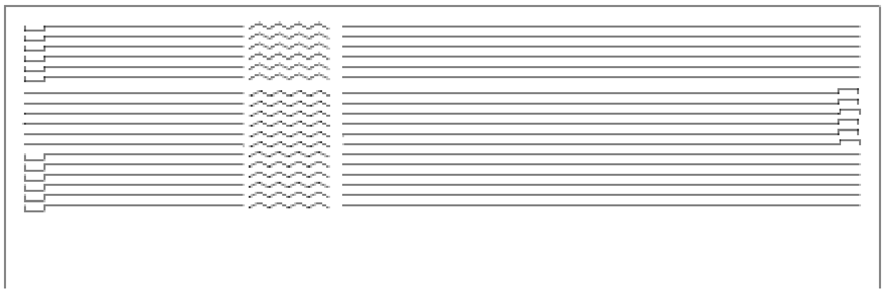
A TV picture is made up of scan lines that draw the whole picture 60 times per second, a full picture is called a frame. A scan line is drawn from left to right and then retraced back to the left side but a little bit below the previous line and then starts over again. This is repeated 262 times per frame which means that we have a total of 262 scan lines. Of these 262 lines, 18 are used for the Vertical sync which leaves us with 244 usable scan lines. The vertical sync tells the TV when it is time to go back to the upper left corner and start a new frame. There is also a horizontal sync per scan line, this tells the TV when to retrace from the right side back to the left.

Figure 1.2 – Scan line details.



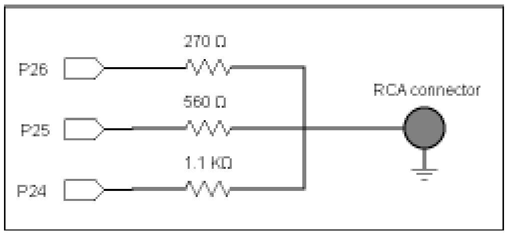
Every video line is 63.5 µs, with a Horizontal blanking period of 10.9 µs and Active video of 52.6 µs. The blanking period is the actual sync signal and divided into a Front porch, Sync tip, Breezeway, Color burst and the Back porch. Do not worry about the time of the individual sections of the sync for now. This is what we have to generate per scan line. The good news is that there is functionality built into the Propeller chip that will take care of a lot of this for you. Generating the Horizontal blanking period can be done with a pretty smart trick and is not as hard as it looks. The rest of the video line is up to you to generate, this is where you actually do something that shows up on the TV screen. You can pretty much choose any resolution you want. The ideal resolution for NTSC is 188 pixels, however a resolution of 160 or 256 pixels might be more desirable.

Figure 1.3 – Horizontal sync



The bottom 18 lines are used for the Vertical sync, these are divided into 3 sections of 6 lines each. First we have 6 lines of Pre equalizing pulses that are no more than a video line with a sync tip and then black level for the rest of the line. The following 6 lines are of Serration pulses that are inverted and reversed. This means they are low as in the sync tip with a black pulse level at the end of the line. Finally another 6 lines of Post equalizing pulses, the same as the Pre equalizing pulses.

Figure 1.4 – Hydra hardware



These signals are generated with a simple 3 bit D/A resistor network connected to the Propeller pins P24 to P26. The resistors are chosen to generate a signal from 0 to 1 Volt in 8 steps. The D/A network is used by the so called Video Streaming Unit to generate these signals, you actually do not have to access the pins at all.

NOTE The reason the ideal resolution for NTSC is 188 pixels is because of the color frequency of 3.579545 MHz. This frequency has a period of 279 ns and since a scan line is 52.6 µs, we get 52.6 µs / 279 ns = 188. So if you try to change the color more than 188 times per scan line the period of 279 ns does not get to finish and the color might become out of phase.

This is it for now, a very brief introduction. Later we will learn how to generate these signals and also look into the timings of the sync signal itself.

1.2 Video driver overview

In order to generate these signals you need to set up a few things.
First you need a clock that is adjusted to the NTSC Frequency of 3.58 MHz, without this clock there is nothing for the TV to sync on to. This clock signal is generated by Counter A in the cog you have chosen for your driver.
Second you need to set up the Video Streaming Unit to use the clock and to generate the signal on Pins P24 to P26.
Do not forget to set these pins to output, the Video Streaming Unit will not set these to output automatically.
You are now ready to start generating the video signals.
Since each frame is 244 usable scan lines, generate 244 scan lines as in the Scan line details.
Each scan line consists of the sync signal and the active video which is where you do something.
Once all these lines are done, generate the 18 Vertical Sync lines.
Start the frame over again and repeat indefinitely.

1.3 Setting up counter A

In order for the Video Streaming Unit (VSU) to be able to generate color we need to program Counter A with a multiple of the NTSC color frequency of 3.579545 MHz. In this example we are going to program it with a multiple of 16 and make it 57.27272 MHz, in order to do this we need to update the FRQA and CTRA registers.
It might be a little bit strange that we calculate the values for Counter A using the 3.58 MHz in stead of 57.27 MHz but it will make sense at the end.
The way to calculate the FRQA setting for counter A is FRQA=(Fsmb*2^32)/Fm.
Where FRQA is the value for the Counter A Frequency register, Fsmb is the desired output Frequency and Fm is the system clock.
This formula is taken from the Hydra Game Programming book but can also be found in the Propeller Application Notes “AN001 – Propeller Counters”.
In our example, FRQA = 3579545*4294967296/80000000 => FRQA = 192175359 ($B745CFF).
There is a standard way in assembler to calculate this so we only need to specify our desired Frequency and let the Propeller do the math for us.

```
            mov     r1, NTSC_color_freq     ' r1: Color frequency in Hz (3.579_545MHz)
            rdlong   v_clkfreq, #0          ' Copy system clock from main memory location 0. (80MHz)
            mov     r2, v_clkfreq           ' r2: CLKFREQ (80MHz)
           ' perform r3 = 2^32 * r1 / r2
            mov     r0,#32+1
:loop       cmpsub  r1,r2           wc
            rcl     r3,#1
            shl     r1,#1
            djnz    r0,#:loop
            mov     v_freq, r3              ' v_freq now contains frqa.
            mov     FRQA, r3                ' set frequency for counter A

NTSC_color_freq  long  3_579_545
```

We also need to set the mode and the PLL divider for Counter A, these are the upper 9 bits of the Control register, CTRA.

The upper 9 bits are organized as "%xCCCCCPPP" where x is unused and C is the CTRMODE setting and P is the PLLDIV setting. For the counter mode, there is really only one setting that is interesting for video, %00001 is the "PLL internal routed to video" setting. For the PLL mode I want to multiply the FRQA with 16, which means %111.
The rest of the bits in CTRA (bits 0-22) are not used when counter A is in "PLL internal routed to video" mode. They are used to specify output pins in other modes.

PLLDIV	***PLL Mode***
%000	X 1/8
%001	X 1/4
%010	X 1/2
%011	X 1
%100	X 2
%101	X 4
%110	X 8
%111	X 16

This is why we set the FRQA to 3.58 MHz rather than 57.27 MHz. Even though we programmed the counter for 3.579545 MHz, the output is multiplied by 16. If we really wanted 3.579545 MHz, we should have programmed the PLL mode with %011. This makes it easy to switch between multiples of a base frequency.

TIP

In assembler there are short cuts to set bits 0-8, 9-17 and 23-31 of a long and leave the rest of the bits unchanged. This is done with the movi, movd and movs instructions. These instructions are used to set bits in the Instruction, Destination and Source fields of a long.

Instruction	***Bits affected***
movi Destination, #%000011111	000011111_xxxxx_xxxxxxxxx_xxxxxxxxx (23-31)
movd Destination, #%111110000	xxxxxxxxx_xxxxx_111110000_xxxxxxxxx (9-17)
movs Destination, #%101010101	xxxxxxxxx_xxxxx_xxxxxxxxx_101010101 (0-8)

Here you can see which bits the instructions changes, unaffected bits are marked with x.

In assembler we can then use the movi instruction to set these two registers.

```
        movi    CTRA,#%00001_111    ' Pll internal routed to Video, PHSx+=FRQx (mode 1) + pll(16x)
```

We have now programmed Counter A for an output Frequency of 57.27272 MHz, the period of this frequency is 17.46 ns which we will be using in calculations later on.

If you want to know more about the counters, read the Propeller Chip Architecture chapter in the Hydra book or the Propeller Application Notes "AN001 – Propeller Counters".

1.4 The video streaming unit

This is the actual hardware that generates the video signal, you can think of it as a pixel pusher.
The Video Streaming Unit, VSU is doing all the hard work for you when generating pixels on the screen. To load pixels and colors into the VSU use the waitvid [Palette],[Pixels] instruction, the VSU will then push them out on the screen with the speed set in Counter A. When using the waitvid instruction execution stops and waits for the VSU to finish pushing the current pixels on the screen, then reload the palette and pixel data. The Palette is made up of four bytes that represents a color each, see the "The color encoding" section on how the color byte is made up. The Pixel data is explained later in this section.
In other words, the VSU will push pixels on to the screen for you but it is your job to keep feeding it with the waitvid instruction.
If you do not feed it fast enough, you are going to start seeing flicker on the screen and might even loose sync. If you feed it too fast you are wasting execution time.
The hardware to generate the composite video signal consists of the three resistors on pins 24 to 26 hooked up to the video socket. We need to tell the VSU which pins are going to be used by setting the Video Configuration register VCFG.

Bits	***VCFG function***
x_HH_xxxxxx_xxxx_xxxxxxxxx_xxxxxxxxx (29-30)	Hardware Select
xxx_C_xxxxx_xxxx_xxxxxxxxx_xxxxxxxxx (28)	Color Mode
xxxx_R_xxxx_xxxx_xxxxxxxxx_xxxxxxxxx (27)	Chroma on Broadcast
xxxxx_A_xxx_xxxx_xxxxxxxxx_xxxxxxxxx (26)	Chroma on Base band
xxxxxx_SSS_xxxx_xxxxxxxxx_xxxxxxxxx (23-25)	Cog for Sub carrier Frequency

Bits	VCFG function
xxxxxxxxx_xxxx_xxxxxxGGG_xxxxxxxxx (8-10)	Pin group
xxxxxxxxx_xxxx_xxxxxxxxx_xMMMMMMMM (0-6)	Pin Mask

The three bits in the Destination field (GGG) is the Pin group setting, for the Hydra, this is %011 since the hardware is connected to pin group 3, pin 24-31.
The eight bits in the Source field (MMMMMMMM) is the mask and should be set to %0000_0111 since we use the lower three bits corresponding to pin 24-26. Pin 27 is used for audio modulation when the cog is in RF modulation mode.
Do not forget to set the actual pins 24-26 to output. The VSU will not automatically set these to output and the screen will be blank.
Now for the topmost bits in the Instruction field. The HH bits are hardware select, a value of %00 disables the output of video, %01 enables the VGA mode which uses all 8 bits in the pin group, %10 enables the bottom four pins (24-27) of our pin group for video and %11 would enable the top four pins (28-31) of our pin group. Since the Hydra uses pin 24 to 26 we need to set these to %10.
The C bit is the color mode, 0 is two color mode and 1 is four color mode, this bit will tell the waitvid instruction how many bytes of the Palette is used. For two color mode only the two lower bytes are used and each pixel is one bit, in four color mode all four bytes are used and each pixel is two bits. The R bit is Chroma on broadcast which we do not use but the A bit is Chroma on the base band which we do use. Chroma tells the VSU to generate the color signal used for NTSC so unless you set this bit, the screen will be in black and white.
Lastly the S bits, which tells the VSU which cog is generating the sub-carrier frequency for FM broadcast. We do not plan on generating this signal so we can just set these to %000.
In order to initialize the VSU for the Hydra, you can use the following instructions,

```
            movs    VCFG, #%0000_0111       ' VCFG'S = pinmask (pin31: 0000_0111 : pin24)
            movd    VCFG, #3                ' VCFG'D = pingroup (grp. 3 i.e. pins 24-31)
            movi    VCFG, #%0_10_101_000    ' Baseband on bottom bits, 2-bit color, Chroma is on.
' Once the VSU is setup to output the video on the Hydra, you also need to make the pins output.
            or      DIRA, tvport_mask       ' set DAC pins to output

' Video Registers (belongs in a DAT section)
tvport_mask            long                    %0000_0111<<24
```

At this point the VSU is fed by the 57 MHz signal from Counter A, video is enabled on pins 24-26, the pins have been set to output, the four color mode is set and Chroma on the video signal is enabled.
Time to have a look at the Video Scale register, VSCL. This register controls the speed that the VSU output the pixels and how many pixels it will output before it is reloaded with the next waitvid instruction. By using different settings we can control how many pixels of the pixel data is actually used.

Bits	VSCL function
xxxxxxxxxxxx_PPPPPPPP_xxxxxxxxxxxx (12-20)	Clocks per pixel
xxxxxxxxxxxxxxxxxxxx_FFFFFFFFFFFF (0-11)	Clocks per frame

The 12 F bits are Clocks per Frame, cpf and the 8 P bits are Clocks per Pixels, cpp, the cpf controls the time between the waitvids and the cpp controls the number of pixels that are output during this time. Remember that the clock signal comes from Counter A at 57 MHz that means every clock signal is 17.46 ns.
Say we want to output 8 pixels of the 16 available in four color mode over a period of 80 clock pulses. To do this, load the cpf bits with 80 and the cpp bits with 80/8 = 10, this can be done in assembler with the following code,

```
         mov   VSCL, NTSC_80_clocks_8_pixels

NTSC_80_clocks_8_pixels   long  10 << 12 + 80       ' This belongs in a DAT section.
```

One way of achieving high color modes is to use only four pixels of the 16 available, use the same value for the pixels and only reload the palette with each waitvid.
For example, say we want a driver that outputs 188 pixels over one video line with 188 individually colors.
We know that each scan line is 52.6 µs and the output from counter A is 17.46 ns per clock, 52.6µs/17.46ns gives us 3013 clock pulses per line, to make things easier round this to 3008 clock pulses per line since 16*188 = 3008.
This means we have 16 clocks per pixel and since we want to output 4 pixels at a time, which are 64 clocks per frame.

```
         mov   VSCL, NTSC_64_clocks_4_pixels

NTSC_80_clocks_8_pixels   long  16 << 12 + 64       ' This belongs in a DAT section.
```

Since each palette is 4 colors, you would need 188/4 = 47 longs per scan line to have 188 individually selectable colors. If we select the pixels to be the colors 0, 1, 2 and 3 next to each other we only need to reload the palette.
For example,

```
DAT
NTSC_Pixel_data                 long  %%0123                ' The four pixels 0 to 3
NTSC_User_Palette               long  $00_00_00_00   ' Set this palette to user colors.

       Initialize a counter to 47, start of scanline
       Start loop
       Load NTSC_User_Palette with a palette for the current position of the four pixels
       waitvid NTSC_User_Palette, NTSC_Pixel_data   ' Output 4 pixels
       Decrease counter and if not zero, continue the loop.
```

TIP The %% (Quaternary number) is used to group two bits together into a number.
So if you would like to convert %%0_1_2_3 into a byte it would be %00_01_10_11.
This is very useful when specifying pixels in two bit mode.

This is it for now, there will be more details later when I explain the demo code.
To learn more about the VSU, you can read chapter 14, Video Cog Hardware in the Hydra book.

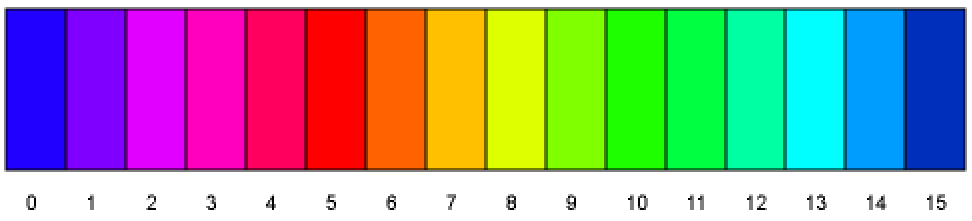
1.4.1 The color encoding

Let us have a look at the color encoding in the Hydra, each color is made up of one byte.

Bits	***Color function***
xxxxx_LLL (0-2)	Luma
xxxx_M_xxx (3)	Chroma Modulation
PPPP_xxxx (4-7)	Position in color wheel

The three lower bits is the Luma or brightness of the color, the fourth bit is the Chroma modulation bit, set this to 1 to enable the colors and 0 for black and white. The upper four bits is the position in the NTSC color wheel if you have turned the Chroma on.
For black and white this is pretty simple. Set the Luma to a value of 2 to 7 where 2 is black and 7 is white and leave the rest of the bits to 0. The value 0 is sync. If you would set the Luma to 0 for black and white, you risk loosing sync.
If you turn the Chroma modulation on, the upper four bits become the position in the color wheel. Think of the wheel as the color spectrum with blue wrapped around to close the circle.

Figure 1.5 – NTSC Color Wheel laid out as a spectrum



The colors start at blue (position 0) and goes through Purple (2), Red (5), Yellow(8), Green (10), Cyan (13) and back to Blue (0) when 15 is passed. When the Chroma is turned on, the Luma should be set between 3 and 6 to control the brightness of the color.
For example, to generate four shades of blue, use the values 11 to 14 since the upper four bits are zero and the Chroma bit is of value 8 which gives 8+3 to 8+6 or 11 to 14.
To generate four shades of Yellow, use 128+8+3 to 128+8+6 or 139 to 142.
There is also a trick to squeeze out another 16 colors by creating “super saturated” colors, do this by specifying Luma to 0 for our 16 colors. For example a super saturated Red would be 64+16+8+0 = 88.

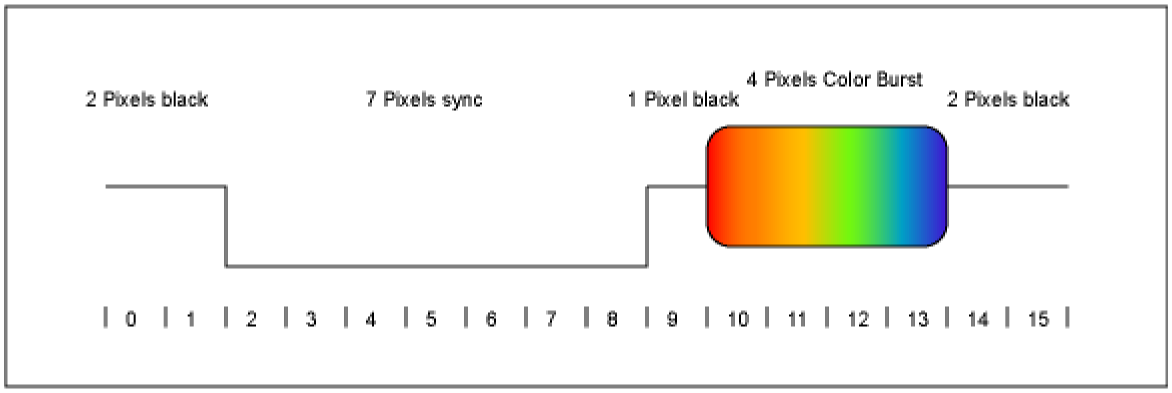
Knowing this we can calculate the number of distinct colors the Hydra can generate.
There are 6 levels of Black and White, 16 colors with 4 brightness level and finally 16 super saturated colors, 6+16*4+16 = 86 colors.

1.5 Generating the video signal

From the Video Driver overview we know the following,
1.We need to generate 244 lines of video that actually display something.
2.Each of these lines have a horizontal sync of 10.9µs and active video of 52.6 µs.
3.Once all 244 lines have been drawn, do the 18 vertical sync rows.

First of all, let us look at the horizontal sync.

Figure 1.6– NTSC Sync made up of pixels



The trick here is to represent these events with a special set of 16 Pixels spread out over 10.9µs, so each pixel would represent 10.9/16 = 0.68µs of this signal.
Let us see how this would work out. Front porch could be two pixels or 0.68 * 2 = 1.4 µs, Sync tip seven pixels or 0.68*7 = 4.8µs, Breezeway 1 pixel, Color burst 4 pixels or 0.68*4 = 2.7µs and Back porch another two pixels for a total of 16 pixels. Compared to the actual NTSC timing, this seems to work out pretty well.

Part of the Sync	*NTSC timing*
Front porch	1.5 µs
Sync tip	4.7 µs
Breezeway	0.6 µs
Colorburst	2.5 µs
Back porch	1.6 µs

Try these pixels,
NTSC_hsync_pixels   long %%11_0000_1_2222222_11
Keep in mind that the VSU shifts out pixels to the right, so the order here is really,
NTSC_hsync_pixels   long %%"Back porch"_"Colorburst"_"Breezeway"_"Sync tip"_"Front Porch."
We need to add a palette with three colors to these pixels as well,
NTSC_sync_signal_palette   long $00_00_02_8A
Here we can see that the Color burst is represented by $8A (Yellow with Luma set to two), the Front and Back Porches plus the Breezeway is $02 (Black) and Sync tip is $00 (Sync).
Now that we have figured out the pixels and the palette we need to set the Video Scale Register (VSCL) to shift out the bits at the correct speed.
The frame length needs to be 10.9µs and since we know that the clock speed of the VSU is 17.46ns this gives us 10.9µs/17.46ns => 624 clocks for the frame and 624/16 = 39 for the clocks per pixels.
NTSC_hsync_VSCL   long 39 << 12 + 624
Once we have all that information our code to generate a Horizontal sync would be as follows,

```
DAT
NTSC_hsync_VSCL                 long  39 << 12 + 624
NTSC_hsync_pixels               long  %%11_0000_1_2222222_11
NTSC_sync_signal_palette        long  $00_00_02_8A
.
.
```

```
.
        mov      VSCL, NTSC_hsync_VSCL                          ' Setup VSCL for horizontal sync.
        waitvid NTSC_sync_signal_palette, NTSC_hsync_pixels     ' Generate sync.
```

While we are focusing on sync, let us look at the Vertical sync.
The first 6 lines consist of Pre equalizing pulses which are a sync signal and then high (black color level) for the rest of the line.
Then 6 lines of Serration pulses which are inverted and reversed, in other words, sync level and then high for 4.7µs at the end of the line.
Finally 6 lines of Pre equalizing pulses the same as the Pre equalizing pulses.
Since the sync settings for horizontal sync and vertical sync are similar we can reuse the same setting for the VSCL and the palette.
The pixels for generating the sync pulse can be approximated with %%1111111_2222222_11, as you can see this is the same as the vertical sync without the color burst.
Since the rest of the line is high or black level, you can program the VSU to output 16 pixels like %%1111111111111111 over a full scan line. A full scan line is 56.2µs and each clock pulse is 17.46ns which gives us 3008 clock pulses, this value was chosen since it's a multiple of 16. To calculate the clocks per pixels, divide this by 16, 3008/16 = 188 clocks per pixel.
The Serration pulses can be done similar, first an active period with pixels like %%2222222222222222 and then a sync period with pixels like %%22_1111111_2222222.
This is the previous signal inversed since we do the active period first and then the sync period where the pixels have been switched around.
This is how the Vertical sync will look like in code,

```
DAT
NTSC_hsync_VSCL                 long  39 << 12 + 624
NTSC_active_VSCL                long  188 << 12 + 3008
NTSC_hsync_pixels               long  %%11_0000_1_2222222_11
NTSC_vsync_high_1               long  %%1111111_2222222_11
NTSC_vsync_high_2               long  %%1111111111111111
NTSC_vsync_low_1                long  %%2222222222222222
NTSC_vsync_low_2                long  %%22_1111111_2222222
NTSC_sync_signal_palette        long  $00_00_02_8A

                        mov     line_loop, #6        ' Line 244, start of first high sync.
vsync_high1             mov     VSCL, NTSC_hsync_VSCL
                        waitvid NTSC_sync_signal_palette, NTSC_vsync_high_1
                        mov     VSCL, NTSC_active_VSCL
                        waitvid NTSC_sync_signal_palette, NTSC_vsync_high_2
                        djnz    line_loop, #vsync_high1    ' Do all 6 sync lines.

                        mov     line_loop, #6        ' Line 250, start of the Seration pulses.
vsync_low               mov     VSCL, NTSC_active_VSCL
                        waitvid NTSC_sync_signal_palette, NTSC_vsync_low_1
                        mov     VSCL, NTSC_hsync_VSCL
                        waitvid NTSC_sync_signal_palette, NTSC_vsync_low_2
                        djnz    line_loop, #vsync_low    ' Do all 6 sync lines.

                        mov     line_loop, #6        ' Line 256, start of second high sync.
vsync_high2             mov     VSCL, NTSC_hsync_VSCL
                        waitvid NTSC_sync_signal_palette, NTSC_vsync_high_1
                        mov     VSCL, NTSC_active_VSCL
                        waitvid NTSC_sync_signal_palette, NTSC_vsync_high_2
                        djnz    line_loop, #vsync_high2    ' Do all 6 sync lines.
```

Now for the really fun stuff, putting the Demo together...
We want two halves of the screen so the upper and lower half will use 122 lines each.
We are also going to need two palettes and two set of pixels, one per half.
Since we are drawing 188 pixels per scan line, we need a loop to be able to draw all pixels per line, with 4 pixels generated at the time, this needs to be done 188/4 = 47 times over the scan line.
Each pixel will now require 3008/188 = 16 clocks per pixel so the clock per frame for 4 pixels will be 4*16 = 64.
Looking at the DAT section of the demo code below,

```
NTSC_Half_of_Graphic_Lines      long  122                   ' Half of our 244 usable lines.
NTSC_Graphics_Pixels_VSCL       long  16 << 12 + 64         ' 16 clocks per pixel, 64 clocks per frame.
NTSC_User_Palette1              long  $8E_8D_8C_8B          ' The yellow shade palette
NTSC_Pixel_data1                long  %%0123                ' Fade pixels from right to left
NTSC_User_Palette2              long  $AE_AD_AC_AB          ' The green shade palette
NTSC_Pixel_data2                long  %%3210                ' Fade pixels from left to right
NTSC_Tiles_Per_Line             long  47                    ' Tiles per line.

' Loop counters.
line_loop                       long  $0                    ' Line counter...
tile_loop                       long  $0                    ' Tile counter...
```

```
‘ ============ Upper part =============

                     mov     tile_loop, NTSC_Tiles_Per_Line
                     ' Set up the tile loop with 47 tiles.
                     mov line_loop, NTSC_Half_of_Graphic_Lines
                     ' Load the line loop with user lines, 122.

user_upper_lines     mov VSCL, NTSC_hsync_VSCL
                     ' Setup VSCL for horizontal sync.
                     waitvid NTSC_sync_signal_palette, NTSC_hsync_pixels
                     ' Generate sync.

‘ ============ Lower part =============

                     mov     tile_loop, NTSC_Tiles_Per_Line
                     ' Set up the tile loop with 47 tiles.
                     mov     VSCL, NTSC_Graphics_Pixels_VSCL
                     ' Setup VSCL for user pixels.

user_upper_tile_loop waitvid NTSC_User_Palette1, NTSC_Pixel_data1
                     ' Draw the tile.
                     djnz    tile_loop, #user_upper_tile_loop
                     ' loop throug the 47 tiles.

                     djnz    line_loop, #user_upper_lines
                     ' Loop through the 122 user video lines.
```

As you can see here, we load the loop counter with 122 since that is half the screen.
Then we generate the horizontal sync since this is needed for every line.
We then load 47 into the tile counter and draw the tile 47 times.
The tile in our example is the four pixels of a Yellow shade.
When all 122 lines for the upper part of the screen is done, we do the same for the lower part.
To make the colors move, we simply use two instructions to shift the palette around,

```
                      rol    NTSC_User_Palette1, #8        ' Shift through the colors.
                      rol    NTSC_User_Palette2, #8        ' Shift through the colors.
```

Since each color is a byte, we have to rotate the palette data with 8.
Once all user lines are drawn we generate the vertical sync as explained earlier.
And finally when all the 244 user lines and the 18 sync lines have been drawn, we start all over again.

1.6 Summary

Although this driver doesn’t do much, it can be used as a template for your own driver.
For example,
Write a high resolution driver by squeezing in more than 188 pixels per line, try 256 pixels like the ZX Spectrum or 320 as the Commodore 64. Use the calculations you learned earlier to get your desired resolution.
Write a high color demo that scrolls all the possible 86 colors over the screen. Add a mask to only have the colors scroll on certain parts of the screen like an Atari logo.
Write a demo that scrolls text smoothly over the screen like a NASDAQ stock market ticker.
Hopefully this article has explained the basics of NTSC video programming and provided you some ideas to try out. Do not forget that there are a lot of demos included on the Hydra CD and the Hydra forum from Parallax is also a great source of help.