F18A Documentation

Contents

Unlocking 2

Detection 2

Detection 2 5

Enhance Color Mode (ECM) 5

Setting palette registers 6

Standard palette register values 8

Multi-color sprites 9

Scrolling 11

Scrolling pages 14

Scroll Limit Registers 15

Bitmap Layer 15

Plotting Pixels on the Bitmap Layer (BML) and GM2 15

Scanline interrupt 17

VDP Registers 17

Wait for vsync 18

GPU instruction set 18

Catalog 19

SPI Flash 19

Unlocking

The F18A defaults to a "locked" mode of operation to prevent legacy software from accidentally enabling any of the enhanced features. I added the lock because during testing some ColecoVision games were causing strange behavior which I discovered was due to the software writing to VDP registers over register 7. Since the 9918A only has 8 registers (0 to 7), it did not matter, and the higher values were simply masked to a number between 0 and 7.

But, the F18A supports VDP register values from 0 to 63 which is how you take advantage of the new features. This is also how the 9938/58 add additional features, and the datasheets for the 9918A indicates registers over 7 are reserved. However, that didn't stop some software from not following the rules, and on the 9918A/9928/9929 the bad behavior did not have any impact. But, the F18A has to protect itself from that old software, thus it powers up locked.

Since the unlocking sequence has to be performed "in band", i.e. using the standard 9918A registers, I had to come up with a way that would would never happen on the real 9918A. VR1 is probably the most critical VDP register since it contains most of the mode bits plus the memory size bit, thus it is VR1 in the form of VR57 (VR57 the same as VR1 on a non-F18A system) that is used to unlock the F18A.

Unlocking is done by writing >1C to VR57 twice, which on a real 9918A VDP is the same as writing to VR1. The value >1C was chosen because it sets the bits in VR1 to something you would never do on a real 9918A, even accidentally because it makes the VDP almost useless. And to write such a value twice, consecutively, is hopefully beyond all probability of happening accidentally.

Value >1C in VR1 looks like this:

```
|4/16K|BLANK| IE0 | M1  | M2  |  X  |SIZE | MAG |
|  0  |  0  |  0  |  1  |  1  |  1  |  0  |  0  |
```

By writing >1C on the real 9918A you are setting 4K VRAM, blank the screen, no interrupts, both M1 and M2 to '1' which is an illegal mode, and a '1' to the unused bit-5 that the datasheet indicates should always be '0'. This would pretty much make the real 9918A useless, and any working software would never operate with this combination of bits in VR1.

Writing to VR57 (binary: 111001) is VR1 on the 9918A which only sees the low 3-bits "001", and must be done twice in a row with no other CPU-to-VDP access. On the F18A you will be writing to VR57, not VR1, and after two consecutive writes the ERM (Enhanced Register Mode) will be unlocked. Any further writes to VR57 after being unlocked will re-lock the F18A.

Because writing >1C to VR1 on the real 9918A would mess up the video mode and other critical VDP configuration, a write to VR1 should immediately follow the unlock sequence if you care to detect the F18A and write software that works on both the 9918A and F18A.

Thus you would have something like:

```
VDPERM
       LIMI 0                   * Interrupts must be off
       LI   R0,>391C            * VR1/57, value 00011100
       BL   @VWTR               * Write once
       BL   @VWTR               * Write twice, unlock
       LI   R0,>01E0            * VR1, value 11100000, a sane setting
       BL   @VWTR               * Write
```

Note that I'm using my version of VWTR here, not the E/A (or XB) versions.

Now you can test for the F18A, which is coming up in my next post.

Detection

To test for the F18A, I'm going to use Tursi's idea of using the GPU, which should make for a smaller test. Assuming the F18A unlock sequence has been performed, a small GPU program will be loaded to the VRAM and executed that will change 1 byte in VRAM. If the byte changed, the F18A is present, otherwise the system is running a stock VDP.

The GPU is a slightly modified 9900 CPU so you can use any standard 9900 assembler to write code for the F18A's GPU. Since the GPU is inside the VDP it can only access the VRAM, plus an additional 2K of memory above the normal 16K of VRAM. The GPU's memory map looks like this:

```
VRAM 14-bit, 16K @ >0000 to >3FFF (0011 1111 1111 1111)
GRAM 11-bit, 2K  @ >4000 to >47FF (0100 x111 1111 1111)
PRAM  7-bit, 128 @ >5000 to >5x7F (0101 xxxx x111 1111)
VREG  6-bit, 64  @ >6000 to >6x3F (0110 xxxx xx11 1111)
current scanline @ >7000 to >7xx0 (0111 xxxx xxxx xxx0)
blanking         @ >7001 to >7xx1 (0111 xxxx xxxx xxx1)
32-bit counter   @ >8000 to >8xx6 (1000 xxxx xxxx x110)
32-bit rng       @ >9000 to >9xx6 (1001 xxxx xxxx x110)
F18A version     @ >A000 to >Axxx (1010 xxxx xxxx xxxx)
GPU status data  @ >B000 to >Bxxx (1011 xxxx xxxx xxxx)
```

"GRAM" means GPU-RAM and has nothing to do with "GROM or GRAM" of the TI console. It is just a coincidence. PRAM is the palette RAM in the F18A, and VREG is the VDP registers to which the GPU has full read/write access.

The program will be loaded up high in VRAM. I like >3F00 for no particular reason, other than it is 256 bytes from the top of VRAM and probably unused unless there is disk access going on (which there won't be during the test).

This is the code that will be loaded into VRAM for the GPU to execute:

```
0000 3F00        DEF  MAIN
                 AORG >3F00
          MAIN
3F00 04E0        CLR  @>3F00
3F02 3F00
3F04 0340        IDLE
3F06 0000        END
```

That is a total of 6 bytes of assembly, which is pretty small for the test. The GPU will clear the word at >3F00, which in this case is the CLR instruction's opcode itself. You have to love self modifying code. :-) After the code runs, the value at VRAM >3F00 be >00 if the F18A is present, otherwise it will be >04 on a stock VDP.

This is the code to load the program to VRAM. I'm including all the support routines here too so it is a complete program:

```
       DEF MAIN

* VDP Memory Map
*
VDPRD  EQU  >8800             * VDP read data
VDPSTA EQU  >8802             * VDP status
VDPWD  EQU  >8C00             * VDP write data
VDPWA  EQU  >8C02             * VDP set read/write address

* Workspace
*
WRKSP  EQU  >8300             * Workspace
R0LB   EQU  WRKSP+1           * R0 low byte reqd for VDP routines

GPU
       DATA >04E0             * 3F00 04E0        CLR  @>3F00
       DATA >3F00             * 3F02 3F00
       DATA >0340             * 3F04 0340        IDLE
GPUEND

MAIN
       LIMI 0
       LWPI WRKSP

*      F18A Unlock
       LI   R0,>391C          * VR1/57, value 00011100
       BL   @VWTR             * Write once
       BL   @VWTR             * Write twice, unlock
       LI   R0,>01E0          * VR1, value 11100000, a real sane setting
       BL   @VWTR             * Write reg

*      Copy GPU code to VRAM
       LI   R0,>3F00
       LI   R1,GPU
       LI   R2,GPUEND-GPU
       BL   @VMBW

*      Set the GPU PC which also triggers it
       LI   R0,>363F
       BL   @VWTR
       LI   R0,>3700
       BL   @VWTR

*      Compare the result in >3F00
       LI   R0,>3F00
       BL   @VRAD
       MOVB @VDPRD,R0
       JEQ  PASS

*
FAIL

*
PASS


*********************************************************************
*
* VDP Set Write Address
*
* R0   Address to set VDP address counter to
*
VWAD   MOVB @R0LB,@VDPWA      * Send low byte of VDP RAM write address
       ORI  R0,>4000          * Set the two MSbits to 01 for write
       MOVB R0,@VDPWA         * Send high byte of VDP RAM write address
       ANDI R0,>3FFF          * Restore R0 top two MSbits
       B    *R11
*// VWAD / VRAD
```

```
************************************************************************
*
* VDP Set Read Address
*
* R0   Address to set VDP address counter to
*
VRAD   MOVB @R0LB,@VDPWA      * Send low byte of VDP RAM write address
       ANDI R0,>3FFF          * Make sure the two MSbits are 00 for read
       MOVB R0,@VDPWA         * Send high byte of VDP RAM write address
       B    *R11
*// VRAD


************************************************************************
*
* VDP Multiple Byte Write
*
* R0   Starting write address in VDP RAM
* R1   Starting read address in CPU RAM
* R2   Number of bytes to send to the VDP RAM
*
* R1 is modified by the value of R2
* R2 is changed to 0
*
VMBW   MOVB @R0LB,@VDPWA      * Send low byte of VDP RAM write address
       ORI  R0,>4000          * Set the two MSbits to 01 for write
       MOVB R0,@VDPWA         * Send high byte of VDP RAM write address
VMBWLP MOVB *R1+,@VDPWD       * Write byte to VDP RAM
       DEC  R2                * Byte counter
       JNE  VMBWLP            * Check if done
       ANDI R0,>3FFF          * Restore R0 top two MSbits
       B    *R11
*// VMBW


************************************************************************
*
* VDP Write To Register
*
* R0 MSB    VDP register to write to
* R0 LSB    Value to write
*
VWTR   MOVB @R0LB,@VDPWA      * Send low byte (value) to write to VDP register
       ORI  R0,>8000          * Set up a VDP register write operation (10)
       MOVB R0,@VDPWA         * Send high byte (address) of VDP register
       ANDI R0,>3FFF          * Restore R0 top two MSbits
       B    *R11
*// VWTR

       END
```

This code triggers the GPU:

```
*      Set the GPU PC which also triggers it
       LI   R0,>363F
       BL   @VWTR
       LI   R0,>3700
       BL   @VWTR
```

The PC (program counter) in the GPU is 16-bit, just like the normal 9900, so it takes two bytes to set up the address. VR54 (>36) is the MSB and VR55 (>37) is the LSB. After writing the LSB to VR55, the GPU automatically triggers and begins execution as the address just set up. In this case it executes the CLR instruction, then goes idle via the IDLE instruction which is perfectly fine on the GPU (don't use IDLE in the 9900 in your 99/4A though!)

Now the value at >3F00 is tested. The VRAD routine sets up a VDP read address without doing a read.

```
*      Compare the result in >3F00
       LI   R0,>3F00
       BL   @VRAD
       MOVB @VDPRD,R0
       JEQ  PASS
```

The MOVB moves the byte at >3F00 in VRAM into the MSB of R0. R0 will now be >0000 if the GPU was present, or >0400 on a stock VDP. The MOVB instruction will automatically compare R0 to zero, so the JEQ will cause a jump if the R0 == 0, i.e. the F18A is present. Or you can put JNE if you need the opposite jump.

Note that writing to VR54 and VR55 is the same as VR6 and VR7 on a stock VDP, so if the test for the F18A fails, you should restore those values to something sensible, or simply set up your VDP accordingly now that you know if you have an F18A or stock VDP (9918A/9928/9929).

Detection 2

If you don't want to go through all the mess of setting the SAT to disable sprites and wait for VSYNC, then another method would be:

1. Unlock sequence
2. Write to VR1 to disable the interrupt
3. Read VDP status
4. Set status register to SR1 (stays on SR0 for the 9918A)
5. Read VDP status
6. Set status register back to SR0
7. Enable interrupts

SR1 is the identity register and returns:

```
0 1 2   3 4 5     6        7
1 1 1 | 0 0 0 | blank | horz_int
```

For the 9938/58, bit-2 will never be set, so you can determine a 9918A, F18A, 9938, 9958 with this method.

Enhance Color Mode (ECM)

The F18A has 64 12-bit programmable color registers. Depending on the ECM selected for tiles and sprites (which can be independently set), the final color index is derived differently. Below is the part of the actual HDL used to set the index:

```
tile_ps & pix_colr when ecm = 0 else              -- Original color mode
tile_ps(0) & pal_sel & pix0 when ecm = 1 else     -- 1-bit color
pal_sel & pix1 & pix0 when ecm = 2 else           -- 2-bit color
pal_sel(0 to 2) & pix2 & pix1 & pix0;             -- 3-bit color
```

If that does not mean anything to you, read on...

To reference any of the 64 palette registers you need a 6-bit number (2^6 = 64) which is "000000" to "111111" or >00 to >3F hex. Based on the ECM, a certain number of bits come from the pattern data, and the rest come from the per-tile attribute table and the global "tile palette select" bits from VR24.

ECM0:

In ECM0 the pattern data is not used directly in color selection, and this is the main difference between ECM0 and ECM1. Just like in the original 9918A, in ECM0 a '1' bit from the pattern specifies that the pixel will be the foreground color, and the actual 4-bit color index comes from that tile's color group byte. Since the 9918A has 16 colors indexed with 4-bits, you still need two additional bits to specify the 6-bit index in the F18A. These two additional bits come from the two Tile Palette Select bits in VR24:

TPS0:TPS1 | COL0:COL1:COL2:COL3

COL0:COL3 come from the high-nybble or low-nybble of the color byte for each group of eight tiles. Thus, the 64 palette registers are sectioned into four groups of sixteen color each. You can change the set of sixteen palette registers used for ECM0 by simply changing the TPS bits in VR24.

ECM1:

In ECM1 (and ECM2 and ECM3) the pattern data becomes color information directly used to select one of two colors from the palette. The color table changes to the per-tile attribute table which contains 4-bits of palette selection data for the tile. Thus, the tile's attribute byte provides 4-bits, and the pattern data provides 1-bit, so only 1-bit is used from the global TPS bits in VR24:

TPS0 | COL0:COL1:COL2:COL3 | PATTERN-BIT

COL0:COL3 are from the per-tile attribute entry for the tile. This sections the 64 palette registers into 32 2-color palettes, and the pattern bit, '0' or '1', selects which of the two colors to use. This is also where the transparent bit from the tile attribute tables comes in to play. If the transparent bit is enabled, a '0' pattern bit will be displayed at transparent instead of the color at the index palette register. This is no different than the special color "0" in the 9918A being transparent instead of black. This is really just a way to specify the difference between black and transparent.

ECM2:

ECM2 works just like ECM1 but picks up a additional color bit from the expanded pattern data. ECM2 makes the pattern table 4K-bytes long instead of 2K-bytes. With 2-bits per pixel, the global TPS bits from VR24 are no longer used:

COL0:COL1:COL2:COL3 | PB0:PB1

PB = pattern-bit

COL0:COL3 are from the per-tile attribute entry for the tile. This sections the 64 palette registers into 16 4-color palettes, and the

pattern bits, '00', '01', '10, and '11', select which of the four colors to use. The transparent bit from the attribute table is used to determine if index '00' (from the pattern data) is transparent or the color at that index.

ECM3:

ECM3 adds a final bit and expands the pattern table out to 6K-bytes. There are now 3-bits per pixel for a total of 8 possible colors for any given pixel in any of the 256 tiles. Only 3-bits are used from the per-tile color bits:

COL0:COL1:COL2 | PB0:PB1:PB2

COL0:COL2 are from the per-tile attribute entry for the tile. This sections the 64 palette registers into 8 8-color palettes, and the pattern bits, '000', '001', '010, '011', '100', '101', '110, and '111', select which of the eight colors to use. The transparent bit from the attribute table is used to determine if index '000' (from the pattern data) is transparent or the color at that index.

The pattern tables expand consecutively in VRAM directly following the original 2K pattern table. Each new 2K pattern table are configured as "bit-planes" and add 1-bit of color data per table (or plane):

```
Bytes in VRAM, offset from the Name Table Base Address from VR2:

 NTBA + 0 . . . .  NTBA + 1
|0|1|2|3|4|5|6|7| |0|1|2|3|4|5|6|7| Pattern Plane 1 LSbit (ECM1,ECM2,EM3)
 1 0 1 0 1 1 1 1   0 0 0 0 0 1 1 0

 NTBA + 2048 . . . NTBA + 2049
|0|1|2|3|4|5|6|7| |0|1|2|3|4|5|6|7| Pattern Plane 2 (ECM2,ECM3)
 0 0 0 0 1 1 1 1   1 0 0 0 1 0 0 1

 NTBA + 4096 . . . NTBA + 4097
|0|1|2|3|4|5|6|7| |0|1|2|3|4|5|6|7| Pattern Plane 3 MSbit (ECM3)
 1 1 1 1 0 0 0 1   0 1 0 1 0 1 0 1

Pixel index values.  Pattern Plane 0 is the LSbit:

 1 0 1 0 1 1 1 1   0 0 0 0 0 1 1 0  ECM1 color indexes (same as pattern plane 1)

 1 0 1 0 3 3 3 3   2 0 0 0 2 1 1 2  ECM2 color indexes, pattern planes 2+1

 5 4 5 4 3 3 3 7   2 4 0 4 2 5 1 6  ECM3 color indexes, pattern planes 3+2+1

For example, the first "4" index above is formed by taking 1-bit from each plane
and forming a binary value that becomes the color index:

P3 P2 P1 - pattern plane
 4  2  1 - binary place value
--------
 1  0  0 - data from each plane
```

Every 8-bytes in each pattern plane describes one tile, just like the 9918A. Shown here, the tables are defining the color indexes for the first two rows of tile 0.

I set up the data in planes, vs. packed consecutively in bytes, for a few reasons:

First, with 3-bits per pixel in ECM3, color data for a pixel would cross byte boundaries and this would seriously complicate things on the hardware side and for the programmer.

Second, using pattern planes like this means you can use your existing pattern data in ECM2 and ECM3 and just add additional pattern data as necessary to get the extra colors.

However, it does make it harder to define the pattern data, and I'm hoping to produce a dedicated editor for this. In the mean time, sometimers99er has graciously expanded his Flash-based sprite editor to support the 8 color possibility of ECM3.

Setting palette registers

The F18A has 64 palette registers (PR) that are 12-bits each, which gives it a color palette of 4069 colors. Which Palette Register (PR) is used to specify the color of a given pixel depends on a lot of settings. The PRs are grouped into "banks" depending on how many bits are used to resolve a pixel's color. In the 9918A compatible modes (1-bit per pixel (bpp)), there are 4-banks of 16-colors each. VR49 has two bits that control which of the 4-banks will be used for tiles in 1-bpp modes, which means you could set up each of the 4-banks with 16 different colors and change to the new palette with a single register write.

In the Enhanced Color Modes (ECM), there are more bits used to specify a single pixel's color, and thus the number of palette banks grows, but the number of colors in each bank shrinks.

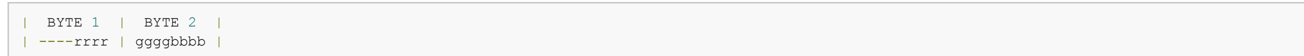
With 2-bpp, there are 16-banks with 4-colors each, and the 2-bits for each pixel select the color from the bank. With 3-bpp, there are

8-banks with 8-colors each.

There are a lot of options for colors, but in this example I'll stick to just updating the palette registers themselves which allows you to use any of the 4096 colors.

NOTE palette changes survive a soft-reset! If you modify the palette and then exit, those changes will remain in effect until the system is power-cycled or hard reset (a cartridge is plugged in, etc.)

Palette registers are numbered 0 to 63 and consist of 12-bits to specify a color in the format:



```
|  BYTE 1   |  BYTE 2   |
| ----rrrr | ggggbbbb |
```

Because there is only one "data write port" to the 9918A (mapped at address >8C00 on the 99/4A) and subsequently to the F18A as well, the F18A has a "Data Port Mode", controlled by a bit in VR47, to select between writing data to VRAM or to Palette Registers.

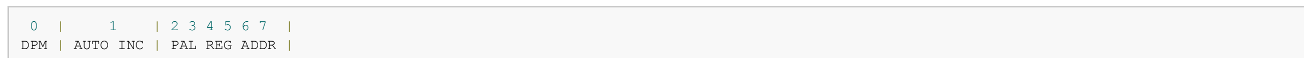
Two byte writes are required to update a single palette register. Palette registers are written to (they cannot be read by the host CPU) by setting the Data Port Mode (DPM) bit in VR47 to 1, then writing to the VDP data port as normal. After the second byte is written, the 12-bit color will be latched into the specified palette register. Side note: a nice advantage of the GPU is that it has full read/write access to the palette registers using normal word instructions like MOV.

If the auto increment bit in VR47 was not set, then the DPM automatically falls back to the default "write to VRAM" mode after the second palette byte has been written.

If a large number of palette registers need to be updated, setting the auto increment flag will keep the DPM in palette mode until VR47 is written again to return to normal VRAM write mode, or the palette address rolls over to 0, which will force the DPM back to VRAM mode. This is a fail-safe to prevent the VDP from inadvertently getting stuck in the write-to-palette DPM.

DMP is also exited any time *any* VDP status register is read, the VDP is externally reset, the palette address rolls over to 0, or by setting VR47 DPM bit to 0.

VR47 controls data-port mode and palette address:



```
 0  |     1    | 2 3 4 5 6 7  |
DPM | AUTO INC | PAL REG ADDR |
```

DPM = Data Port Mode
0 = VDP data writes go to VRAM as normal
1 = VDP data writes go to the palette

AUTO INC = Auto Increment
0 = Do NOT increment the palette address after a single *palette* write, which consists of *TWO* bytes written to the VDP. After the second byte has been received, the addressed palette register will be updated and the Data Port Mode defaults back to normal VRAM for writes to the VDP. This mode of operation is intended for updating a single palette register at a time.

1 = Increment the palette address every time a palette register is updated, which consists of *TWO* bytes written to the VDP with the DPM bit = 1. This allows multiple palette registers to be updated consecutively and quickly.

PAL REG ADDR = Palette Register Address
This is the address of the single palette register to update, or the first palette regiser to update when AUTO INC = 1.

Here is an example of writing PR4, which is normally "dark blue" (RGB: 54F) to a pure blue (RGB: 00F):

```
       LI   R0,>2F84          * Reg 47, value: 1000 0100, DPM = 1, AUTO INC = 0, PR4.
       BL   @VWTR
       LI   R1,>000F          * RGB: 00F, or pure blue in place "dark blue"
*      Two bytes written to the VDP now go to PR1
       MOVB R0,@VDPWD
       SWPB R0
       MOVB R0,@VDPWD
```

After the second write (MOVB instruction), the DPM will fall back to normal VRAM mode since the auto-increment bit in VR47 was not set.

If you were going to update a whole set of registers, then you could use the auto-increment feature. If your update does not cause the PR number to roll over to zero, then you must leave DPM mode after updating:

```
*      Update the first 7 palette values from the host CPU
*      Palette 0 is not updated to keep the screen color stable.
       LI   R0,PAL0+2
       LI   R1,>0111          * Add 1 to each R,G,B value
       LI   R2,7
INCPAL
       A    R1,*R0+           * Update the 12-bit color
       DEC  R2
```

```
       JNE  INCPAL

       LI   R0,>2FC1          * Reg 47, value: 1100 0001, DPM = 1, AUTO INC = 1, start PR1.
       BL   @VWTR
*      Every two bytes written to the VDP now go to the palette registers.
       LI   R0,PAL0+2
       LI   R2,14             * Each 12-bit palette entry requires 2 bytes
UPDPAL
       MOVB *R0+,@VDPWD
       DEC  R2
       JNE  UPDPAL
       LI   R0,>2F00          * Reg 47, value: 0000 0000, exit DMP
       BL   @VWTR


**
* Standard color palette
*
* 12-bit color format: ---rrrrggggbbbb
       EVEN
PAL0
       DATA >0000             * Transparent
       DATA >0000             * Black
       DATA >02C3             * Medium Green
       DATA >05D6             * Light Green
       DATA >054F             * Dark Blue
       DATA >076F             * Light Blue
       DATA >0D54             * Dark Red
       DATA >04EF             * Cyan
       DATA >0F54             * Medium Red
       DATA >0F76             * Light Red
       DATA >0DC3             * Dark Yellow
       DATA >0ED6             * Light Yellow
       DATA >02B2             * Dark Green
       DATA >0C5C             * Magenta
       DATA >0CCC             * Gray
       DATA >0FFF             * White
PAL0E
```

Standard palette register values

When the F18A is powered up, it defaults the four palettes as follows:

#0 standard 9918A colors
#1 an EMC1 version of palette #0
#2 IBM CGA colors
#3 an ECM1 version of palette #2

Note that palette changes will survive a reset, i.e. plugging in a cartridge or software reset. They only assume these defaults at power-on.

```
-- Palette 0, original 9918A NTSC color approximations
x"000", --  0 Transparent
x"000", --  1 Black
x"2C3", --  2 Medium Green
x"5D6", --  3 Light Green
x"54F", --  4 Dark Blue
x"76F", --  5 Light Blue
x"D54", --  6 Dark Red
x"4EF", --  7 Cyan
x"F54", --  8 Medium Red
x"F76", --  9 Light Red
x"DC3", -- 10 Dark Yellow
x"ED6", -- 11 Light Yellow
x"2B2", -- 12 Dark Green
x"C5C", -- 13 Magenta
x"CCC", -- 14 Gray
x"FFF", -- 15 White

-- Palette 1, ECM1 (0 index is always 000) version of palette 0
x"000", --  0 Black
x"2C3", --  1 Medium Green
x"000", --  2 Black
x"54F", --  3 Dark Blue
x"000", --  4 Black
x"D54", --  5 Dark Red
x"000", --  6 Black
x"4EF", --  7 Cyan
x"000", --  8 Black
x"CCC", --  9 Gray
x"000", -- 10 Black
x"DC3", -- 11 Dark Yellow
x"000", -- 12 Black
```

```
x"C5C", -- 13 Magenta
x"000", -- 14 Black
x"FFF", -- 15 White

-- Palette 2, CGA colors
x"000", --  0 >000000 (  0   0   0) black
x"00A", --  1 >0000AA (  0   0 170) blue
x"0A0", --  2 >00AA00 (  0 170   0) green
x"0AA", --  3 >00AAAA (  0 170 170) cyan
x"A00", --  4 >AA0000 (170   0   0) red
x"A0A", --  5 >AA00AA (170   0 170) magenta
x"A50", --  6 >AA5500 (170  85   0) brown
x"AAA", --  7 >AAAAAA (170 170 170) light gray
x"555", --  8 >555555 ( 85  85  85) gray
x"55F", --  9 >5555FF ( 85  85 255) light blue
x"5F5", -- 10 >55FF55 ( 85 255  85) light green
x"5FF", -- 11 >55FFFF ( 85 255 255) light cyan
x"F55", -- 12 >FF5555 (255  85  85) light red
x"F5F", -- 13 >FF55FF (255  85 255) light magenta
x"FF5", -- 14 >FFFF55 (255 255  85) yellow
x"FFF", -- 15 >FFFFFF (255 255 255) white

-- Palette 3, ECM1 (0 index is always 000) version of palette 2
x"000", --  0 >000000 (  0   0   0) black
x"555", --  1 >555555 ( 85  85  85) gray
x"000", --  2 >000000 (  0   0   0) black
x"00A", --  3 >0000AA (  0   0 170) blue
x"000", --  4 >000000 (  0   0   0) black
x"0A0", --  5 >00AA00 (  0 170   0) green
x"000", --  6 >000000 (  0   0   0) black
x"0AA", --  7 >00AAAA (  0 170 170) cyan
x"000", --  8 >000000 (  0   0   0) black
x"A00", --  9 >AA0000 (170   0   0) red
x"000", -- 10 >000000 (  0   0   0) black
x"A0A", -- 11 >AA00AA (170   0 170) magenta
x"000", -- 12 >000000 (  0   0   0) black
x"A50", -- 13 >AA5500 (170  85   0) brown
x"000", -- 14 >000000 (  0   0   0) black
x"FFF"  -- 15 >FFFFFF (255 255 255) white
```

Multi-color sprites

The color enhancements to sprites and tiles, as far as the pattern representation goes, works the same way. As you know, the original VDP could support 2-colors per sprite with one of the colors always being transparent, thus we tend to think of sprites a having one color. The sprite's color could be any of the sixteen original colors, including transparent which presents some interesting possibilities.

In the original sprite pattern data, a '0' bit specifies a transparent pixel and also does not count in collision detection. However, a '1' bit in the pattern specifies that the sprite has a pixel at that location, regardless of the color. That is an important distinction to keep in mind, because is allows you to have a pixel (1-bit in the pattern) that is transparent, i.e. the sprite's main color is set to 0.

So, the bit patterns in the original sprite (and tile) modes don't actually represent the color, they represent if the sprite has a pixel at that location. The color data is derived from a different location, and in the case of sprites the color comes from the Sprite Attribute Table (SAT) entry for the given sprite.

When moving to the ECM (enhanced color modes) for sprites and tiles, the pattern data itself *does* actually select a color from a palette. However there is a distinction between tiles and sprites when it come to the "zero-index", i.e. color data "0", "00", or "000". For sprites the zero-index is *ALWAYS* transparent, so the number of actual colors you can display is 1, 3, or 7 vs 2, 4, or 8. In the ECMs for tiles there is an attribute byte for each tile "name" (0-255) were you can specify if the zero-index is transparent or the color at that index.

To provide more pixel data, the extra pattern bits need to come from some where. To keep some sort of compatibility with existing patterns, I chose to implement the extra bits via "bit planes". This allows you to start off with some existing sprite or tile patterns, and expand them to support more colors at a later time in your development. Also, the 3-bpp mode does not pack neatly into a single byte had I tried to use a linear bit-packing method.

The down-side to bit-planes is that making patterns is more of a pain. Luckily sometimes99er has implemented some initial support for the multi-color sprites via his sprite editor.

For example, for a 2-bpp pattern you need two bits to specify which of the 4-colors to use for a given pixel. There are four possible values:

```
bits | index color
00   | 0
01   | 1
10   | 2
11   | 3
```

.
This is simply binary representation, and for 3-bpp it becomes "000" to "111". Note that the least significant bit comes from the original pattern table (bit plane), and the 2nd or 3rd bits come from subsequent bit-planes.

For example, here are two pattern bytes that will have four pixels next to each other, each being one of the four possible colors in a given palette (which is specified by a tile's or sprite's attribute byte).

```
01010000 pattern-plane 0
00110000 pattern-plane 1, 2K (2048) byte-offset from bit-plane 0 (the original pattern table)
--------
01230000 color index values.
```

.

For each byte that makes up a sprite's or tile's pattern, there are one or two more bytes in the additional pattern-planes that are used to make the final color index for a given pixel. The additional pattern-planes are always 2K (and 4K for 3-bpp) bytes offset from the Sprite Pattern Generator Table (SPGT). This means that in 1-bpp the SPGT is the normal 2K, for 2-bpp the SPGT is 4K, and 3-bpp it is 6K.

To get a pixel's color index you combine the bits *vertically* from each pattern byte in each plane. So, the second pixel is "01", or index 1. The byte from the first pattern-plane represents the LSbit in the final index value. Shown below is the sprite data using 3-bpp to show all eight colors in a single row:

```
bits | color index
000  | 0 (0 is always transparent for sprites)
001  | 1
010  | 2
011  | 3
100  | 4
101  | 5
110  | 6
111  | 7

the bits are combined vertically
 | | | | | | | |
 V V V V V V V V
|0|1|0|1|0|1|0|1| LSbit pattern-plane 0, 2K total
|0|0|1|1|0|0|1|1|       pattern-plane 1, 4K total, 2048 bytes offset from the SPGT
|0|0|0|0|1|1|1|1| MBbit pattern-plane 2, 6K total, 4096 bytes offset from the SPGT
 | | | | | | | |
 V V V V V V V V
|0|1|2|3|4|5|6|7| color index values.
```

.

The first step to making a multi-color sprite is to make a pattern that has data for all the bit-planes, which depends on the number of colors you want. You set up the sprite tables as you normally would, load the patterns, set up the SAT, and finally enable the ECM for sprites.

VR49 controls the ECM for both tiles and sprites:

```
|    0     |   1   |   2     3   |    4   |  5   |   6     7   |
FIXED_EN | ROW30 | ECMT0 ECMT1 | Y_REAL | LINK | ECMS0 ECMS1 |
```

.

The bit fields for ECM(T)iles and ECM(S)sprites are:

00 - 0 - original 9918A mode
01 - 1 - 1-bpp
10 - 2 - 2-bpp
11 - 3 - 3-bpp

Thus, to enable sprites to use 2-bpp just write >02 to VR49. Heh, all that talking just to say that... Really all the detail is in setting up the patterns, which is just additional data written to the VRAM.

> **RasmusM, on 12 Sept 2013 - 9:17 PM, said:**
> Do all sprites share the same palette of 4 or 8 colors, or can each sprite use a different palette?

In the enhanced color modes (ECM), each sprite can reference any "palette".

There are 64 Palette Registers (PR) that are 12-bits each. Each PR is programmable and you can set any PR to any value (color) from >000 to >FFF (4095), i.e. there are 4-bits per red, green, and blue. The number of PRs is fixed, but the number of "palettes" at any time depends on the current ECM.

Having 64 PRs means you need 6-bits to address a PR for a pixel. In the original color mode, the sprite's color is an index into 1 of 4 main palettes. In the Enhanced Color Modes, the pattern data becomes part of the palette index. To complete the 6-bits to address a PR, there are 2-bits for sprites and 2-bits for tiles that come from the new "palette select" VDP Register (VR) VR24:

```
VR24:
 0  1  2  3  4  5  6  7 (Note, I use TI's bit numbering)
XX XX S0 S1 XX XX T0 T1
```

S0 and S1 are the "sprite palette select" and T0 and T1 are the "tile palette select". These two bits (two for sprites, two for tiles) are used to complete the 6-bit PR address when there are not enough pattern bits (original mode, ECM1 ECM2). Thus, for sprites, there are three places data comes from to select a pixel's color:

VR24: PS0&PS1 (sprite palette select bits)
Sprite attribute table: CS0 CS1 CS2 CS3 (color select)
Sprite pattern data: 0 to 3 bits

PS0&PS1 can be: 00, 01, 10, or 11, and are always the MSbits of the palette address, so they can select 1 of 4 groupings of PRs:

```
00xxxx PR0 to PR15
01xxxx PR16 to PR31
10xxxx PR32 to PR47
11xxxx PR48 to PR63
```

In the original color mode (ECM0), xxxx comes from the Sprite Attribute Table and will thus select 1-of-16 colors in the "palette" specified by the PS0&PS1 bits from VR24. This defaults to "00" and PR0 to PR15 are defaulted to the original 99/4A colors. In original mode, *pattern data* does not contribute to selecting a sprite pixel color, and simply indicates if the pixel is visible or not.

In ECM1 to ECM3, pattern data becomes part of building the PR address. The more pixel data, the less the PS0&PS1 are used, and the less of the sprite's color attribute (CS0 to CS3) is used:

```
PR Address bit: 0   1   2   3   4   5
-------------------------------------
original mode: ps0 ps1 cs0 cs1 cs2 cs3 (VR24 S0&S1 bits and SAT color index)
1-bit (ECM1) : ps0 cs0 cs1 cs2 cs3 px0 (VR24 S0 bit only, SAT color index, pattern bit)
2-bit (ECM2) : cs0 cs1 cs2 cs3 px1 px0 (SAT color index, two pattern bits)
3-bit (ECM3) : cs0 cs1 cs2 px2 px1 px0 (3 SAT color index bits, three pattern bits)
```

In ECM1 there are effectively 32 "palettes" with two colors each. VR24 PS0 and the SAT color (5-bits) make up the palette selection from 0 to 32, and the pattern bit selects one of the two colors in the "palette". Note that a sprite pattern value of zero "0", "00", or "000" is *always* transparent. So ECM1 for sprites is really only useful over original mode to enable the other enhanced sprite features.

In ECM2 there are effectively 16 "palettes" with four colors each. The sprite palette select bits from VR24 are now unused, and only the sprite's color from the SAT and the pattern data are used. So, in ECM2 the sprite's color in the SAT becomes a "palette select" from 0 to 15, and the two pattern bits select one of the four colors in the palette. Again, color "00" is transparent.

In ECM3 there are effectively 8 "palettes" with eight colors each. Only three bits from the sprite's color in the SAT are used to select the palette, and the three pattern bits select one of the eight colors in the palette. Color "000" is transparent.

So it is really a matter of how the 64 PRs are grouped and addressed based on the ECM. You can reprogram any of the PRs at any time, which will change the color of any pixel referencing that PR.

Scrolling

Scrolling is one of the areas that I ended up spending a lot of time on, and subsequently there are a *lot* of options. So, let me start with the basics and build up from there.

Note1 The term "pixel" here refers to what I call a "fat pixel", which is made up of a 2x2 block of VGA pixels, and represents the 9918A's pixel resolution of 256x192.

Note2 Let me say right now that when I was implementing the scrolling, I was not considering the text modes because they use a special pixel counter that is not a multiple of 8 (and that is critical to the scrolling hardware). Scrolling will do *something* in the text modes, but it won't be what you expect. Everything here assumes GM1 for the most part.

The guts of the scrolling are the horizontal and vertical scroll registers. These are a full 8-bits each and cause the display to be adjusted in 1-pixel increments.

When you start scrolling, you immediately become aware of several problems, the first being what should display on the left or right (or top and bottom) as you start to scroll. This is where the idea of "pages" comes in to play.

There are two "page" bits that control how big the display is horizontally and vertically. With a "0" horizontal page size, if you scroll left or right the display wraps at the edges. With a "1" horizontal page bit, you now have two name tables in VRAM that make a virtual display size of 64x24 tiles, of which you can see a 32x24 window:

```
virtual screen  |0<--------------->63|
physical screen   |0<---->31|
```

So, as you increase the horizontal scroll register, instead of the display wrapping, you start to see what is in the second name table. In memory, the name tables are 1K each and are consecutive. The name table base address can still be used to locate the name tables, but the size grows 1K for each additional "page" (of which there can be 1, 2, or 4).

```
VRAM Name Table address example
>0000 start 1st name table, page 0
>02FF end 1st name table, page 0 (767 bytes)

>0300 256 bytes unused
>03FF ...

>0400 start 2nd name table, page 1
>06FF end 2nd name table, page 1

   01234----------31-32-------------63
  +-----//...//-----+-----//...//-----|
 0|0,1,2............|1024,1025,1026...|
 1|
 2|......page 0.....|......page 1.....|
.
.
.
21|
22|
23|.............767|............1791|
  +-----//...//-----+-----//...//-----+
```

Also, in a 1-page setup, you would normally have to mask-off the left-most and right-most columns (0 and 31) and constantly provide new display information to "scroll in" or "scroll out". With the extra horizontal page, you still need to do updates to provide the appearance of endless scrolling, but you don't have to mask the edge columns and you have at least 16 columns of buffer space to work with on either side, or a full screen depending on how you set things up.

The vertical page size works the same way, except instead of the page-1 name table being displayed next to page-0, it would show up below page-0 as you start to scroll vertically:

```
   01234----------31
  +-----//...//-----|
 0|0,1,2............|
 1|
 2|......page 0.....|
.
.
.
21|
22|
23|.............767|
  +-----//...//-----+
24|1024,1025,1026...|
25|
26|......page 1.....|
.
.
.
45|
46|
47|............1791|
  +-----//...//-----+
```

The final configuration would be a "1" for both the horizontal and vertical page size, in which case there are now four name tables:

```
VRAM Name Table address example
>0000 start 1st name table, page 0
>02FF end 1st name table, page 0 (767 bytes)

>0300 256 bytes unused
>03FF ...

>0400 start 2nd name table, page 1
>06FF end 2nd name table, page 1

>0700 256 bytes unused
>07FF ...

>0800 start 3rd name table, page 2
```

```
>0AFF end 3rd name table, page 2

>0B00 256 bytes unused
>0BFF ...

>0C00 start 3rd name table, page 3
>0EFF end 3rd name table, page 3

   01234----------31-32-------------63
  +-----//...//-----+-----//...//-----|
 0|0,1,2...........|1024,1025,1026...|
 1|
 2|......page 0.....|......page 1.....|
.
.
.
21|
22|
23|.............767|............1791|
  +-----//...//-----+-----//...//-----+
24|2048,2049,2050...|3072,3073,3074...|
25|
26|......page 2.....|......page 3.....|
.
.
.
45|
46|
47|............2815|............3839|
  +-----//...//-----+-----//...//-----+
```

The 4-page mode gives you the ability to scroll around a 32x24 window on a virtual 64x48 tile field. Also, with the ROW30 bit, the 24 rows expands to 30 rows, so your display becomes a 32x30 window on a 64x60 tile field. Also with the ROW30 bit set, the name tables go from 768 bytes (32*24) to 960 bytes (32*30) with only 64 bytes of unused memory between the pages. This was one of the main reasons for placing the pages on 1K boundaries (not to mention the technical reasons).

The page layout scheme was inspired by the NES, which has a very similar setup.

```
hsize|vsize|hscroll|vscroll|name tables
-----+-----+-------+-------+-----------
  0  |  0  | wraps | wraps | 1 (page 0)
  1  |  0  | pg0/1 | wraps | 2 (page 0 and 1)
  0  |  1  | wraps | pg0/1 | 2 (page 0 and 1)
  1  |  1  | pg0/1 | pg2/3 | 4 (page 0, 1, 2, 3)
```

There are also two more bits associated with the horizontal and vertical size bits, and those are the horizontal and vertical "page start" bits. These control which page is the "starting" page when the horizontal or vertical size is "1".

Think about when you scroll to the right a little (increase the horizontal scroll register) with a horizontal size of "0". If the horizontal scroll is at 16, for example, as the display is drawn the left of the screen starts with data for column 2. But, when you get to column 31 in VRAM you still have 2 columns on the screen to display. So the F18A uses the data from columns 0 and 1 to finish the row. This is the wrapping in a single page setup.

Now, if you have a horizontal size of "1", then the 2 columns of data to finish off the row (using the example above) would come from columns 0 and 1 of the second page.

Here is where the "start page" settings come in to play. In the example above, say you have the horizontal scroll register set to 255 (max). This means you are displaying one *pixel* of data from the first page (page 0) before you start using data from page 1, which is entirely displayed except for 1 pixel. At this point, if you wanted to continue scrolling, you need to be able to "start" on page 1 and use data from page 0 as you continue to scroll. This is what the "start page" bit does.

After all that I came to the realization that you would probably want to keep part of the display still while you scrolled, like for scores and such. So I added a "top border" register that would let you specify a line above which scrolling would not affect the display. So, you could set the top-border to 8 to keep the first row still while the remaining rows would scroll around.

Of course the top-border soon expanded into a bottom border, then a left and right border. Therefore, you also have four registers that can define a "scroll window" on a pixel level. Inside the window the tiles scroll, outside the window the tiles are fixed.

The top-border was originally called the top "banner" (because I was only going to have that one additional register), and I realized you might want to have only what is on the first name table (page 0) display at the top while having a horizontal size of "1" (two pages wide). So there is also a setting to control how "wide" the top border is, i.e. 1 page or 2 pages. It can really make for some strange effects and was really designed with side-scrolling games in mind, but I tried not to do that though, i.e. create features for specific purposes.

On top of all of that, I thought it would be nice to have control over how each tile is affected by the scroll registers. Thus the "fixed map" was created. The fixed map is a bitmap of the 32x24 (or 32x30) tile display, each each bit determines if that tile will be affected by the scrolling or not, i.e. "fixed" in place. The fixed table has a new base-address register and can be located on 128-byte boundaries (I think, I can't remember right off hand). Each byte controls 8 tiles on the screen:

```
.........byte 0.........  ...  .........byte 3.........
|t0|t1|t2|t3|t4|t5|t6|t7| ... |t24|t25|t26|t27|t28|t29|t30|t31|
```

Each row requires four bytes (4*8=32), and there are 24 rows, so a total of 96 bytes for a 32x24 name table and 120 bytes for a 32x30 name table (ROW30 set).

The sprites and the bitmap layer (BML) are *not* affected by the scroll registers. However, the BML can be moved around at a pixel level via is own xy location registers, and can be set above or below the tiles (have priority over tiles or not). So, with the scrolling, scroll borders, fixed map, BML, and sprites you can probably come up with some pretty nice effects.

Scrolling pages

The F18A provides support for multiple "pages" when scrolling, and a "page" is simply additional "name tables" (NT) in the traditional sense of the 9918A VDP. There can be up to four NTs which are always consecutive in VRAM starting with NT1 which begins based on the NT-base in VR2.

When either the horizontal scroll register (HSR) or vertical scroll register (VSR) are incremented, there needs to be something to display at the edges where visual data is coming into view. Since scrolling can take place in two directions at once, there are four options for where the new data can come from:

1. One page, both horizontal and vertical directions wrap.
2. Two horizontal pages, vertical direction wraps.
3. Two vertical pages, horizontal direction wraps.
4. Four pages, no wrapping.

When using scrolling, the Name Table Base Address (NTBA) in VR2 is limited to 2-bits instead of the normal 4-bits. Thus the NT start address can only be located on 4K boundaries instead of 1K boundaries when using scrolling:

```
VR2:
MSB                LSB
0  1  2  3  4  5  6   7
X  X  X  X A0 A1  A2  A3 - Normal
X  X  X  X A0 A1 VPS HPS - Vertical / Horizontal Page Start from VR29
```

When scrolling, the HPS and VPS bits come from VR29:6 and VR29:7 and change from 0 -> 1 or 1 -> 0 depending on the horizontal / vertical page size selections in VR30. Allowing these bits to toggle, but having VR30:1 / VR30:2 set to one (1), is what causes a new name table to be selected when scrolling in either direction. If VR30:1 / VR30:2 are zero (0) for a given direction, then wrapping occurs in that direction instead of using a new name table.

```
VR29:
MSB                LSB
0  1  2  3  4  5   6   7
X  X  X  X  X  X  HPS VPS

VR30:
   MSB                      LSB
   0      1      2    3 4 5 6 7
HBSIZE HPSIZE VPSIZE  SPRITEMAX

HBSIZE = horizontal banner size
HPSIZE = horizontal page size
VPSIZE = vertical page size
```

Because the VPS has a higher bit-value (bit-2) than the HPS bit (bit-3) in the VRAM address, the horizontal name table will always come in memory before the vertical name table. This also means that two-page scrolling in the vertical direction will always use name tables 1 and 3, and skip name table 2. Two-page scrolling in the horizontal direction will always use name tables 1 and 2.

```
VRAM Address, 14-bits:

 14   13   12   11   10  9   8   7  6  5 4 3 2 1 - number of bits
8192 4096 2048 1024 512 256 128 64 32 16 8 4 2 1 - bit place value

 MSB                             LSB
 0 1 2 3 | 4 5 6 7 8 9 10 11 | 12 13 14 - bit number (reverse of "industry standard")
---------------------------------------
VR2:4..7 | y raster counter  | x MOD 8  - normal

  MSB                                           LSB
  0   1  |  2  |  3  | 4 5 6 7 8 9 10 11 |   12 13 14
VR2:4..5 | VPS | HPS |     y modified     | x modified - scrolling
```

It can be seen that the VSP will select between 0 (0K) or 2048 (2K) offset, and HPS will select a 0 (0K) or 1024 (1K) offset. VR2 is reduced to 2-bits and can locate the four name tables at 0K, 4K, 8K, or 12K when scrolling is being used.

The X pixel counter and Y raster counter are modified by the horizontal and vertical scroll registers, and when the counters reach their

limits they reset and cause the VPS or HPS to toggle if the page size is one (1) in VR30, thus causing a new name table to be used, otherwise wrapping occurs in that direction.

Scroll Limit Registers

The F18A has four registers: VR50, VR51, VR52, and VR53 that can be used to limit the area in which the scroll registers will affect the tiles. The registers are a full byte each, thus they can define a Top, Bottom, Left, and Right boundary at the pixel level. The top (VR50) must be less than the bottom (VR51), and the left (VR52) must be less than the right (VR53). A value of zero disables that particular boundary.

For example, if VR50 (top boundary) has a value of 29, then all lines from 0 to 29 will not be affected by scrolling, but lines 30 to 191/239 (for row30) will be affected by scrolling.

Used together, the four registers can be used to define a "window" in which scrolling takes place, but outside the window the display is fixed.

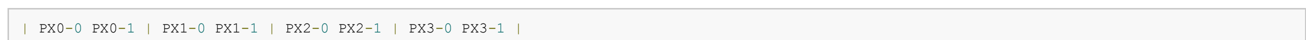
Fixed Map

The "Fixed Map" is used to allow any tile to be "fixed" in place and not affected by scrolling. The Fixed Map is a bit-map, so there is 1-bit per tile to control if the tile is fixed or affected by scrolling. VR10 specifies the Fixed Map Base Address and is 7-bits, so the table can be located on 128-byte boundaries. VR49 contains a "fixed enable" bit which must be one (1) to enable the fixed map.

The scroll limit registers and fixed map can be used at the same time. When a tile, pixel row, or pixel column is not affected by scrolling, it will always display tile information from the first name table, i.e. "Page 1".

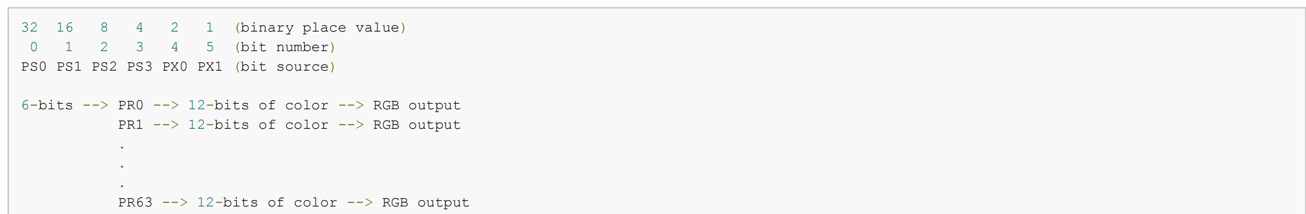
Bitmap Layer

In the F18A, the bitmap layer (BML) uses 2-bits for each pixel, so 1-byte in VRAM holds 4-pixels. That means the bits for each pixel can identify 4 possible colors: "00", "01", "10", and "11" in binary or 0, 1, 2, and 3 in decimal.



```
| PX0-0 PX0-1 | PX1-0 PX1-1 | PX2-0 PX2-1 | PX3-0 PX3-1 |
```

But you need 6-bits to identify one of the 64 palette registers to use, so the extra 4-bits come from a new VDP Register related to the BML and are called the "palette select" bits. So taken together, 4-bits come from the new register, and 2-bits come from each pixel:



```
32  16   8   4   2   1  (binary place value)
 0   1   2   3   4   5  (bit number)
PS0 PS1 PS2 PS3 PX0 PX1 (bit source)

6-bits --> PR0 --> 12-bits of color --> RGB output
           PR1 --> 12-bits of color --> RGB output
           .
           .
           .
           PR63 --> 12-bits of color --> RGB output
```

Since the top 4-bits come from the Palette Select, they can select 1-of-16 groups of 4-colors in the total range of 64 palette registers. The pixel data selects 1 of 4 colors in that group. This is exactly how the "color groups" of the 9918A work for each group of 8 tiles in the color table, only here we are dealing with pixels and groups of 4 instead of groups of 8.

The BML also has an option to use pixel value "00" as transparent instead of an index into a palette register, which gives you 3 colors and transparency (either background or the tile layer) instead of 4-colors. It all depends on what you want to do.

Plotting Pixels on the Bitmap Layer (BML) and GM2

The GPU can plot a BML pixel, given an XY location, in a single instruction. It can also read a pixel, conditionally set a pixel based on the current pixel color, read and write a pixel at the same time, just calculate a pixel's VRAM address, or calculate a GM2 pixel's address!

I call the new instruction PIX, and it uses the same opcode as the 9900's XOP instruction, so you can use any 9900 assembler to code

the PIX instruction. The F18A GPU does not have a Workspace Pointer (since its registers are hard-wired instead of memory-base), so XOP was not implemented.

The XOP format is multi-addressing for the source, and workspace register for the destination. This makes it very flexible for the PIX instruction. Here are the options you can use with PIX:

```
Format: MAxxRWCE xxOOxxPP

M - 1 = calculate the effective address for GM2 instead of the new bitmap layer
    0 = use the remainder of the bits for the new bitmap layer pixels
A - 1 = retrieve the pixel's effective address instead of setting a pixel
    0 = read or set a pixel according to the other bits
R - 1 = read current pixel into PP, only after possibly writing PP
    0 = do not read current pixel into PP
W - 1 = do not write PP
    0 = write PP to current pixel
C - 1 = compare OO with PP according to E, and write PP only if true
    0 = always write
E - 1 = only write PP if current pixel is equal to OO
    0 = only write PP if current pixel is not equal to OO
OO - pixel to compare to existing pixel
PP - new pixel to write, and previous pixel when reading
```

The source value is the XY location as two bytes, the X being the MSB. Since the XOP supports multiple addressing for the source parameter, you can use a register or memory location. XY values are 0 to 255.

The destination parameter is the PIX instruction as indicated above. If you use the M or A operations (calculate addresses only), the destination register will contain the address after the instruction has executed. If you use the R operation, the read pixel will be in PP (over writes the LSbits). You can read and write at the same time, in which case the PP bits are written first and then replaced with the original pixel bits.

Example (this is code running on the GPU):

```
LI  R0,>2020 * xy=32,32
LI  R1,>0001 * write a pixel of "01"
XOP R0,R1 * PIX R0,R1

- or -
EVEN * make sure XPIX is an even address
XPIX BYTE 50
YPIX BYTE 50
.
.
.
LI  R1,>0801 * Read existing pixel at XPIX,YPIX and write a "01" pixel in its place
XOP @XPIX,R1

- or -
LI  R1,>0302 * ONLY write a 2("10") pixel if the current pixel is 0("00")
XOP @XPIX,R1

- or -
LI  R1,>0213 * ONLY write a 3("11") pixel if the current pixel is NOT 1("01")
XOP @XPIX,R1

- or -
LI  R1,>8000 * Get the GM2 effective address of the pixel at XY location
XOP @XPIX,R1 * R1 now contains the VRAM address byte containing the pixel.
* Doing (XPIX AND >07) will isolate the bit in the specified byte.
```

The PIX instruction was really designed to assist with the BML, so using it with GM2 does require a little extra work to update the pixel in the appropriate byte. However, the address of the byte that contains the pixel to be updated it calculated for you, which replaces all this code (from the E/A manual page 336):

```
MOV  R1,R4
SLA  R4,5
SOC  R1,R4
ANDI R4,>FF07
MOV  R0,R5
ANDI R5,7
A    R0,R4
S    R5,R4
```

This is a very nice routine, and it took me a long time to figure out how it worked. But once I did, I was very impressed with what was going on, and I was also intrigued to know that all this code does is bit-twiddling. Since bit-twiddling is something that takes a lot of work via programming, but something that hardware does naturally, all that code can be replaced with a single bit of hardware (shown here as HDL):

```
gm2_addr <= "00" & (
(pgba & src_oper(8 to 12) & "00000" & src_oper(13 to 15)) +    -- y / 8 * 256 + (y % 8)
("0000" & src_oper(0 to 4) & "000"));        -- + (x AND >F8) (mask out the pixel index bits)
```

Two dashes -- are comments in HDL. So, one adder and some bit twiddling and the address is calculated in 10ns. Not that you need to know that, but I thought it was interesting.

Scanline interrupt

Since the 9918A only has one interrupt line, the when you use the HINT it triggers the same interrupt line. Therefore on the host side you have to check for both VINT and HINT when you are using both. This means you probably can't use the 99/4A's console ISR with the HINT.

The HINT works very much like the 9938, and I did try to do some enhancements like the 9938 where possible.

To set up a HINT you set the scan line you want the interrupt to trigger on in VR19. A value of zero (0) will disable the interrupt. You also have to enable the HINT, just like the VINT, in VR0:

```
       MSB                       LSB
       0   1   2   3   4   5   6   7
       -----------------------------
9918A  0   0   0   0   0   0   M3 EXTVID
9938   0  DG  IE2 IE1  M5  M4  M3  0
F18A   0   0   0  IE1  0   M4  M3  0

IE = interrupt enable.  IE0 is in VR1 and is the VINT enable, as per the 9918A.
```

The F18A does not care if the zero bits are zero (0) or one (1), they are ignored. So VR0:3 (IE1) has to be set to one (1) *and* VR19 has to have a value other than zero (0) to enable the interrupt. This way, even if poorly written legacy software sets VR0:3 to one (1), the F18A will still not generate the HINT because VR19 defaults to zero (and the only way to update VR19 is to unlock the F18A.)

The HINT is reported just like the 9938 via status register SR1:

```
       MSB                       LSB
       0   1   2   3   4   5   6   7
       ------------------------------
9938   LPF LPS ID0 ID1 ID2 ID3 ID4 HF
F18A   ID0 ID1 ID2  X   X   X  BLK HF

LPF = light pen flag
LPS = light pen switch
BLK = horizontal or vertical blanking is active
HF = horizontal interrupt flag
```

The HF works the same way as the VINT flag of the 9918A VDP. If VR0:3 (IE1) is set to one (1) then when the scan line reaches the value in VR19 (and VR19 is not zero (0)), the VDP interrupt output is triggered (set low), and SR1:7 (HF) is set to one (1) and stays set until you read SR1.

The F18A GPU also has direct access to the current scan line counter as well as all the VDP registers. The GPU is also very fast.

VDP Registers

The GPU is a modified 9900 so it can access VRAM, the VDP Registers, etc. as 16-bit or 8-bit values. Yes, the VDP Registers are memory mapped into the GPU's address space (they are also readable by the host system), as well as the current scan line, blanking, etc:

```
-- Address building
-- VRAM 14-bit, 16K @ >0000 to >3FFF (0011 1111 1111 1111)
-- GRAM 11-bit, 2K  @ >4000 to >47FF (0100 x111 1111 1111)
-- PRAM  7-bit, 128 @ >5000 to >5x7F (0101 xxxx x111 1111)
-- VREG  6-bit, 64  @ >6000 to >6x3F (0110 xxxx xx11 1111)
-- current scanline @ >7000 to >7xx0 (0111 xxxx xxxx xxx0)
-- blanking         @ >7001 to >7xx1 (0111 xxxx xxxx xxx1)
-- 32-bit counter   @ >8000 to >8xx6 (1000 xxxx xxxx x110)
-- 32-bit rng       @ >9000 to >9xx6 (1001 xxxx xxxx x110)
-- F18A version     @ >A000 to >Axxx (1010 xxxx xxxx xxxx)
-- GPU status data  @ >B000 to >Bxxx (1011 xxxx xxxx xxxx)
```

VRAM = VDP RAM
GRAM = GPU RAM (only accessible by the GPU)
PRAM = Palette RAM, 16-bit access *ONLY*, i.e. you can not use MOVB to PRAM.
VREG = VDP Registers

Wait for vsync

```
NUM192 BYTE 192
.
.
.
WAIT_B CB   @>7000,@NUM192     * Wait for the blanking period
       JL   WAIT_B
```

GPU instruction set

These are the *new* instructions, and they use unused (that looks funny) 9900 opcodes:

```
* CALL 0C80 0000 1100 10Ts SSSS
* RET  0C00 0000 1100 0000 0000
* PUSH 0D00 0000 1101 00Ts SSSS
* POP  0F00 0000 1111 00Td DDDD
*
* SLC  0E00 0000 1110 00Ts SSSS
```

You can use these in any assembler by using DATA statements inline with the code. Examples with hard coded addressing (a pain I realize):

```
CSON   EQU  >03A0             * SPI chip select enable
CSOFF  EQU  >03C0             * SPI chip select disable
CALSYM EQU  >0CA0             * CALL Symbolic 0000 1100 1010 0000
RET    EQU  >0C00             * RET  0000 1100 0000 0000
PUSHR0 EQU  >0D00             * PUSH R0 0000 1101 0000 0000
POPR1  EQU  >0F01             * POP  R1 0000 1101 0000 0001
.
.
.
       LI   R15,>47FE         * Set the stack pointer to the bottom of the GRAM
.
.
.
       DATA CALSYM            * CALL @GETIDX
       DATA IDXNUM
.
.
.
       DATA CSON              * SPI !CE (CKON opcode)
       DATA CSOFF             * SPI CE (CKOF opcode)
.
.
.
       DATA RET
```

The F18A uses R15 as the stack pointer, so it should be set up before any of the stack functions are used. The stack operations grow *down* in memory and the stack is always word operations (16-bit) on even addresses.

Modified instructions:

```
IDLE = IDLE    Forces the GPU state machine to the idle state, restart with a trigger from host
XOP  = PIX     The new dedicated pixel plotting instruction
CKON = SPI !CE Sets the chip enable line to the SPI Flash ROM low (enables the ROM)
CKOF = SPI CE  Sets the chip enable line to the SPI Flash ROM high (disables the ROM)
LDCR = SPI OUT Writes a byte (always a byte operation) to the SPI Flash ROM
STCR = SPI IN  Reads a byte (always a byte operation) from the SPI Flash ROM
RTWP = RTWP    Modified, does not use R13, only performs R14->PC, R15->status flags
```

XOP was chosen for PIX because it uses the workspace pointer similar to BLWP, and the GPU does not have a workspace pointer, and thus XOP would have been unimplemented anyway. Also, it provided useful enough addressing, i.e. general addressing for the source which allows the XY location bytes to be in a register or VRAM, and workspace register addressing for the destination (pixel command).

Unimplemented instructions:

```
SBO
SBZ
TB
BLWP
STWP
```

```
LWPI
LIMI
RSET
LREX
```

Since the F18A does not have a workspace pointer, interrupts, or a CRU, instructions related to those components were not implemented. Any unknown opcode will simply be ignored (NOP if you want to think of it like that), meaning it will not affect the GPU or cause it to go awry.

Catalog

The pre-loaded routines are probably not as much as you might think. Rather than try to guess what everyone might need, I decided to keep it to a minimum (and not hold up the update any longer) and just provide some sort of software library later.

The initial firmware had two pre-loaded routines, a block copy and font load. In the v1.4 update I just added a minimal number of routines, and made room for the user to add their own routines using the same parameter mechanism and vector table.

BLKCPY * Block Copy
FONTLD * Font Load
GETINF * Get catalog version, free memory, vector tables
GETIDX * Get a catalog index entry
BLOBLD * Load a data blob from the catalog

The firmware has a catalog file that I had hoped to fill with more routines, sound data, patterns, etc. but it never happened (too much work, not enough time). The catalog currently contains the pre-loaded code itself (so the F18A can be software reset if desired), the default palettes, and about 22 character sets (patterns for tiles 0 to 255).

SPI Flash

You can definitely brick the F18A via GPU software if you write to the wrong part of the SPI Flash. The SPI Flash contains the bit stream for the FPGA and if you overwrite it then the FPGA will not have anything to load at power-on. Being able to write to the SPI Flash is key to me being able to write a software-based firmware update, as well as the file-system storage idea.

Having said that, you can safely *read* any part of the SPI Flash. It currently contains the FPGA bit stream and some font patterns. You also probably would not be able to accidentally write data to the SPI Flash, it takes a very specific sequence of commands and data.

The GPU-to-SPI interface is low level, so you will need to get the datasheet for the M25P80 SPI Flash chip. All serial flash is "command based", meaning to read data you have to send the "read data" command, followed by the 24-bit address, and finally read out the data. It is actually very similar to how the 9918A works, as well as the GROM.

I re-purposed four opcodes to interface with the SPI Flash:

CKON - clock on - Set the SPI chip enable low. The chip enable is active low.
CKOFF - clock off - Set the SPI chip enable high.
LDCR - load communication register - write a byte (always a byte) to the SPI Flash.
STCR - store communication register - read a byte from the SPI flash.

Did you see the "gpu_preload.asm" file I posted? It contains the pre-loaded GPU code that is part of the FPGA bit stream and has low level SPI functions.

Basically you would follow this sequence:

```
       CKON        Enable the SPI Flash
       . . .
       LDCR        Write to the SPI Flash
       STCR        Read from the SPI Flash
       . . .
       CKOFF       Disable the SPI Flash
```

.
Here is a real example of reading some data from the SPI Flash:

```
CSON   EQU  >03A0             * SPI chip select enable
CSOFF  EQU  >03C0             * SPI chip select disable
.
.
.
* The name, address, length, and info byte
       EVEN
```

```
ENTNAM BYTE 32,32,32,32,32,32,32,32,32,32
ENTADR BYTE >00,>00,>00       * Catalog data 24-bit address
ENTLEN BYTE >00,>00           * Length of data
ENTINF BYTE >00               * Info, 1-byte
       EVEN

.
.
.

*      R1,R0 = Fast read command (1 byte) + 24-bit address of catalog entry to read

       LI   R3,ENTNAM         * Start of the catalog entry structure to load
       LI   R2,16             * Size of the catalog entry structure

       DATA CSON              * SPI !CE (CKON opcode)
       LDCR R1,8              * Send Fast-Read Command in MSB of R1
       SWPB R1
       LDCR R1,8              * Send MSB of 24-bit address
       LDCR R0,8              * Send 2nd byte
       SWPB R0
       LDCR R0,8              * Send LSB of 24-bit address
       STCR R0,8              * Consume (read) Fast-Read dummy byte
IDXG10 STCR *R3+,8            * Read 1-byte into the structure
       DEC  R2
       JNE  IDXG10
       DATA CSOFF             * SPI CE (CKOF opcode)
```
.
There are 16 "sectors" in the SPI Flash, each sector being 64K (>10000) each. The bit stream reserves the first 5 (0 to 4) sectors from >00000 to >4FFFF. The catalog reserves sector 5 from >50000 to >5FFFF. You can safely write data in any of the remaining 10 (6 to 15) sectors from >60000 to >FFFFF. It was this area that I was planning to use for the "disk" storage, but that idea is still vaporware.