(Unofficial) Vera Interface Documentation

Emulators r27 through r32

READ ME FIRST: I've been pretty lax about updating this with specifics from r33 onwards, but little has changed… until very, very recently. New documentation for the Vera shows that it's about to experience an overhaul, **beware using this doc for emulator r37 and later**.

Disclaimer

I can't emphasize enough that this document is subject to change. This documentation is based on the r27 through r31 emulator implementations. I have probably made at least one mistake. The emulator may have bugs. Decisions may be made that cause final hardware to behave differently.

Especially as time goes on, do not take this document as gospel. I did this for my own benefit, I just figured I may as well try to help others while I'm at it.

Differences between emulator versions

r27 and r28

The memory layout of sprite data changed dramatically.
The composer VRAM addresses were moved from $04:$006X to $04:$004X.

r28 and r29

There are no changes to the Vera between these versions.

r29 and r30

Fixed bugs with drawing 2bpp and 4bpp graphics data, both tiles and sprites.
Added Vera IRQ enable and reset registers at $9F26 and $9F27.

r30 and r31

r31 changes to the "Vera 0.8" specification. This is a fairly significant remapping of addresses, but few functions have actually changed, aside from how the "memory access stride" works on the Vera's memory-mapped registers.

When porting code from r30 to r31, be aware of the following changes:

The Vera's interface through its memory-mapped registers has changed.

Address	Description
$9F20	**Low byte of VRAM address**
$9F21	High byte of VRAM address
$9F22	**Bank select and memory access stride**
$9F23	Data at VRAM address 0
$9F24	Data at VRAM address 1
$9F25	Toggle between setting VRAM address 0 and 1, and/or reset video to defaults
$9F26	Enable or disable interrupts from the Vera
$9F27	Interrupt flag

Memory access stride/auto-increment is changing from a direct value to power-of-two, as in the following table:

Value	Actual Stride/Auto-increment
0	0
1	1
2	2
3	4
4	8
5	16 ($10)
6	32 ($20)
7	64 ($40)
8	128 ($80)
9	256 ($0100)
10	512 ($0200)
11	1024 ($0400)
12	2048 ($0800)
13	4096 ($1000)
14	8192 ($2000)
15	16384 ($4000)

The layout of VRAM within the VERA is changing substantially.

Bank	Start	End	Purpose
$00	$0000	$FFFF	Video RAM

$01	$0000	$FFFF	Video RAM
-	-	-	Everything from $020000 to $0EFFFF is unused.
$0F	$0000	$001F	Display composer registers
$0F	$1000	$11FF	Palette
$0F	$2000	$200F	Layer 1 registers
$0F	$3000	$300F	Layer 2 registers
$0F	$4000	$400F	Sprite control registers
$0F	$5000	$5FFF	Sprite graphics data

Notice that there is no longer a charset ROM, as well. The charset starts at $01F000, the tail end of Video RAM, which also means characters are fully editable and now consume part of VRAM that we had previously gotten “for free”.

SPI access has been added, mapped through the Vera’s VRAM.

r31 and r32

Fixes a bug when drawing h-flipped and v-flipped sprites.

r35 and r36

The kernal now operates on layer 1 by default. This doesn’t affect VERA programming directly, but programs written prior to r36 that operate on layer 0 may depend on layer 1 being disabled by default.

Things to look forward to

The official documentation promises the ability to enable interrupt requests and poll for them, as well as control which scanline will generate interrupts. The IRQ toggling has been implemented as of r30, but it is not yet possible to set which line on which the IRQ triggers as of r32: the Vera will always generate an interrupt when its “scan pixel” rolls over to the beginning of the front porch for the current display mode. This is expected to go in with r33.

There is also promised to be some sprite collision detection added at some future point, limited to lines rendered to screen. This will include collision masks added to sprite data, and a VRAM address to poll for collision between groups.

We may also get the ability to override the transparent color index on sprites to another index value.

Other things which appear in the documentation:

- Query the composer for which field is being drawn in NTSC modes!
- SPI access
- UART

Writing to (and Reading from) VRAM

Using BASIC, with VPOKE and VPEEK

Through BASIC, the commands are VPOKE and VPEEK:

Command	**Description**
VPOKE X,Y,Z	Write value Z to the VRAM address on bank X, address Y.
VPEEK(X,Y)	Read the value from VRAM address at bank X, address Y.

Example:

```
5 REM”Flip the colors of the logo, if it’s still at its starting position!
10 DIM A%(6)
20 FOR Y=0 TO 6
25 ADDR=$100*Y+1
30 A(Y)=VPEEK(0,ADDR)
40 NEXT Y
50 RESTORE
60 FOR Y=0 TO 6
70 FOR X=0 TO 10
75 ADDR=($100*(6-Y))+(2*X)+1
80 VPOKE 0,ADDR,A(Y)
90 NEXT X,Y
100 END
```

Using the memory-mapped addresses

The Vera’s interface is mapped to the memory addresses $9F20 through $9F25:

Address	**Description**
$9F20	Low byte of VRAM address
$9F21	High byte of VRAM address
$9F22	Bank select and memory access stride
$9F23	Data at VRAM address 0
$9F24	Data at VRAM address 1
$9F25	Toggle between setting VRAM address 0 and 1, and/or reset video to defaults
$9F26	Enable or disable interrupts from the Vera

$9F27	Interrupt flag

These can be written to through BASIC or through assembly.

A note about the nomenclature: This document refers to the most significant byte of a 3-byte address as the “bank” byte. In descending order, this makes the three bytes “bank”, “high”, and “low”. This seems to be common usage in other 65XX contexts. 3-byte addresses in this document are also typically written as $00:$0000, separating the “bank byte”. This is convenient because the Vera has laid out its memory such that the “bank byte” could be interpreted as having special meaning. However, the official Vera documentation refers to the bytes as “high, mid, low” instead of “bank, high, low”.

BASIC accesses these addresses through regular POKE statements. In fact, VPOKE is essentially just an alias to a particular set of regular POKEs that hide the Vera’s memory mapped interface from you and allow you to pretend you’re writing to the Vera directly. Any given VPOKE X,Y,Z is basically implemented like this under the hood:

```
5 REM”Select primary VRAM address
10 POKE $9F25,0
15 REM”Set bank to X
20 POKE $9F22,X
25 REM”Set primary address high byte to the high byte of Y
30 POKE $9F21,(Y-(Y AND $FF))/256
35 REM”Set the primary address low byte to the low byte of Y
40 POKE $9F20,(Y AND $FF)
45 REM”Write the byte Z to bank $X, address $Y
50 POKE $9F23,Z
```

A VPEEK(X, Y) looks very similar, differing only by doing a PEEK at the very end:

```
5 REM”Select primary VRAM address
10 POKE $9F25,0
15 REM”Set bank to X
20 POKE $9F22,X
25 REM”Set primary address high byte to the high byte of Y
30 POKE $9F21,(Y-(Y AND $FF))/256
35 REM”Set the primary address low byte to the low byte of Y
40 POKE $9F20,(Y AND $FF)
45 REM”Read bank $X, address $Y and store in variable Z
50 Z = PEEK($9F23)
```

Through assembly, you’ll write to the memory addresses just like writing to any other memory address:

```
LDA #0
STA $9F25 ; Select primary VRAM address
```

```
STA $9F22 ; Set primary address bank to 0, stride to 0
STA $9F21 ; Set primary address high byte to 0
STA $9F20 ; Set Primary address low byte to 0
LDA #$FF
STA $9F23 ; Writing $FF to primary address ($00:$0000)
```

And, of course, reading can look very similar, except we're retrieving the value instead of writing to it:

```
LDA #0
STA $9F25 ; Select primary VRAM address
STA $9F22 ; Set primary address bank to 0, stride to 0
STA $9F21 ; Set primary address high byte to 0
STA $9F20 ; Set Primary address low byte to 0
LDA $9F23 ; Reading $FF to primary address ($00:$0000)
```

$9F20 and $9F21 - Memory access address

If you think of VRAM as a 16-bit address, these two memory-mapped registers contain the low ($9F20) and high ($9F21) byte of the VRAM address you want to access on the currently selected channel.

This byte order ("little-endian", or "smallest byte to largest byte") is a standard convention for the 6502 processor and its derivatives. For instance, in assembly, when accessing an indirect reference (an address written to some other location in memory), the 6502 assumes that the lower byte of the address comes first. Further specifics of this convention fall outside of the scope of this document.

$9F22: Setting the VRAM "bank" and "stride"

The value at $9F22 uses the format %SSSSBBBB:

Field	Description
SSSS	4 bits for the "stride" or "automatic increment" after accessing the current channel's VRAM address.
BBBB	4 bits for the "bank" of the current channel's VRAM address.

The "bank" of a VRAM address is only 4 bits' worth of address space, from $00 to $0F, and much of that goes unused. This makes up the lowest 4 bits of what we write to $9F22.

The upper 4 bits define the "memory access stride" or "automatic memory increment". This means that when we read from or write to that address, the Vera will automatically increment the address we'll access by whatever the stride is set to. If we wanted to write a string of characters to the screen without changing the colors at those screen locations, we might set the stride to 2 because each character on screen is actually 2 bytes wide: 1 byte for the character and 1 byte

for color information. So to automatically skip over the color bytes of VRAM, we'd set the stride to 2.

As of r31, the stride is not literally the value written to the upper 4 bits. Instead, the stride value can be found in the following table:

Value	Actual Stride/Auto-increment
0	0
1	1
2	2
3	4
4	8
5	16 ($10)
6	32 ($20)
7	64 ($40)
8	128 ($80)
9	256 ($0100)
10	512 ($0200)
11	1024 ($0400)
12	2048 ($0800)
13	4096 ($1000)
14	8192 ($2000)
15	16384 ($4000)

In BASIC, this would look like this:

```
5 REM"Select primary VRAM address
10 POKE $9F25,0
15 REM"Set the primary address low byte 0
20 POKE $9F20,0
25 REM"Set primary address high byte to 0
30 POKE $9F21,0
35 REM"Set bank to 0, stride to 2
40 POKE $9F22,$20
45 REM"Write $73 to primary address ($00:$0000)
50 POKE $9F23,$73
55 REM"Write $73 to primary address ($00:$0002)
60 POKE $9F23,$73
65 REM"Write $73 to primary address ($00:$0004)
70 POKE $9F23,$73
```

In assembly, this would look like this:

```
LDA #0
STA $9F25 ; Select primary VRAM address
LDA #0
STA $9F20 ; Set Primary address low byte to 0
STA $9F21 ; Set primary address high byte to 0
LDA #$20
STA $9F22 ; Set primary address bank to 0, stride to 2
LDA #$73  ; PETSCII heart <3
STA $9F23 ; Writing $73 to primary address ($00:$0000)
STA $9F23 ; Writing $73 to primary address ($00:$0002)
STA $9F23 ; Writing $73 to primary address ($00:$0004)
```

As you can imagine, this is much faster when you want to access multiple locations that are a fixed interval from each other, because there's no need to reconfigure any part of the Vera's memory-mapped interface once you start reading.

So suppose you have a side-scrolling game, where new tiles scroll in from the side of the screen. Each tile is defined by 2 bytes, making each "row" of tiles 256 bytes long. We can write "9" for the stride of our address, and then efficiently write a column of tiles, instead of needing to reset the Vera's memory access address for each tile. Very convenient!

$9F23 and $9F24 (and $9F25) - Two memory access channels

Until this point, this guide has only configured and used the primary VRAM access address, but the Vera has the ability to configure $9F23 to point at one memory address, and point $9F24 to a different address. These channels act completely independent from each other - reading or writing from one causes no changes to the other.

We set which data access register to configure by writing to $9F25, the Vera control register.

This is useful for a couple of purposes. First, it allows for relatively fast copying of strips of data in VRAM. Consider the following BASIC:

```
5 REM"Select and configure primary VRAM address to $00:$0000 with stride of 2
10 POKE $9F25,0
20 POKE $9F22,$20
30 POKE $9F21,0
40 POKE $9F20,0
45 REM"Select and configure the secondary VRAM address to $00:$0100
46 REM"(the location of the next line in VRAM) with a stride of 2
50 POKE $9F25,1
60 POKE $9F22,$20
70 POKE $9F21,$01
80 POKE $9F20,0
90 FOR I=0 TO 2
95 REM"Read the character at $00:$0000 + I*2
100 X=PEEK($9F24)
105 REM"Write that character to $00:$0100 + I*2
```

```
110 POKE $9F25,X
120 NEXT I
```

This reads the three characters from $00:$0000, $00:$0002, and $00:$0004, and copies them to $00:$0100, $00:$0102, and $00:$0104, respectively.

And the equivalent in assembly:

```
LDA #0
STA $9F25 ; Select primary VRAM address
LDA #$20
STA $9F22 ; Set primary address bank to $00, stride to 2
LDA #$00
STA $9F21 ; Set primary address high byte to $00
STA $9F20 ; Set primary address low byte to $00

LDA #1
STA $9F25 ; Select secondary VRAM address
LDA #$20
STA $9F22 ; Set secondary address bank to $00, stride to 2
LDA #$01
STA $9F21 ; Set secondary address high byte to $01
LDA #$00
STA $9F20 ; Set secondary address low byte to $00

LDA $9F23 ; Reading the character at ($00:$0000)
STA $9F24 ; Writing that character to ($00:$0100)

LDA $9F23 ; Reading the character at ($00:$0002)
STA $9F24 ; Writing that character to ($00:$0102)

LDA $9F23 ; Reading the character at ($00:$0004)
STA $9F24 ; Writing that character to ($00:$0104)
```

It may not be terribly convincing, looking at the code for 3 bytes, that this is a big win. But in many cases, such as scrolling background layers, it is desirable to copy an entire row of tiles from one place in VRAM to another. When multiplying that simple write/reads loop to a size of 80 or 160 (depending on whether we need color data), the benefits become obvious.

$9F26 and $9F27 - Interrupts

These are the interrupt control and handling registers.

When the emulator triggers an interrupt, it moves the program counter to the address contained at $FFFE and $FFFF, which is inside ROM. The address contained here points to the kernal's interrupt handler, itself also in ROM. The kernal stows the A, X, and Y registers (in that order), and then checks whether it's dealing with a regular IRQ or a non-maskable interrupt. We care about the former, in which case the kernal sets the program counter to the address contained in

$0314 and $0315, which are fixed RAM addresses that we can use to plug in our own IRQ handler. See also, the balloon example from Frank Buss. This all matches the behavior of the Commodore 64's interrupt handler. Note that if the default IRQ handler at $0314 and $0315 is overridden, the program should either stow the values and jump to the default handler when done, or else restore the processor's registers before the RTI instruction.

They are the best tool we have for identifying when a new frame begins.

When IRQs are enabled on the Vera, they are generated at the beginning of the VGA front porch, and its equivalent in NTSC display modes for each field.

$9F26 only has two valid values:

Value	Description
0	Disable IRQs from the Vera
1	Enable IRQs from the Vera

Reading from $9F27 is similar:

Value	Description
0	No IRQ has been generated by the Vera
1	An IRQ has been generated by the Vera

Writing to $9F27, however, is reversed:

Value	Description
0	Does nothing
1	Clear the IRQ that has been generated

This means that when you are about to exit from an interrupt handler, write a 1 to $9F27 to clear the Vera's interrupt status flag (this will allow it to generate future interrupts).

VRAM Memory Layout

Bank	Start	End	Purpose
$00	$0000	$FFFF	Video RAM
$01	$0000	$FFFF	Video RAM
-	-	-	Everything from $020000 to $0EFFFF is unused.
$0F	$0000	$001F	Display composer registers
$0F	$1000	$11FF	Palette

$0F	$2000	$200F	Layer 1 registers
$0F	$3000	$300F	Layer 2 registers
$0F	$4000	$400F	Sprite control registers
$0F	$5000	$5FFF	Sprite graphics data
$0F	$7000	$7FFF	SPI data

The enormous quantities of “space” between the VRAM locations almost certainly do not reflect any real memory on the chip, they’re just mappings. But it does introduce a useful subtlety: Attempting to write to some location *close to, but greater than* an address in VRAM will cause the write to wrap around to the beginning of that address space. Attempting to write to $0F1200 (the first byte past the useful palette memory), for instance, will actually write to $0F1000, the first byte in palette memory.

On the one hand, this can save you a few instructions when overwriting certain portions of Vera memory, because you don’t always have to explicitly reset the Vera to the start address of a range. On the other hand, if you’re not careful you can clobber data you didn’t intend to. *It is not recommended that you depend on this behavior, as it may differ from final hardware.*

Composer Settings

Emulator r31 moves the composer memory mapping from $04:$004X to $0F:$000X. Nothing else has changed, as far as I can tell.

The composer adjusts global video effects, in particular the output video mode, border color, and the size of the display area.

Address	**Description**
$0F:$0000	Output mode and chroma toggle.
$0F:$0001	Horizontal Scale
$0F:$0002	Vertical Scale
$0F:$0003	Border color palette index
$0F:$0004	Horizontal start of display area
$0F:$0005	Horizontal end of display area
$0F:$0006	Vertical start of display area
$0F:$0007	Vertical end of display area
$0F:$0008	Course adjustments to display area

$0F:$0000 - Output mode and chroma toggle

This field takes bytes in the format %00000CMM.

Field	Description
00000	Unused.
C	1 bit to disable chroma, a.k.a. “Enable black-and-white display”
MM	2 bits Mode parameter

The possible Mode settings are:

Value	Description
0	Disable video. Outputs a blue screen.
1	VGA mode
2	NTSC mode

$0F:$0001 - Horizontal Scale

In general, this byte holds the inverse of the display’s scale, with a 1:1 scale at 128. Although any value is recognized, these are the integer scale values:

Value	Description
128	1:1 Horizontal scale
64	2:1 Horizontal scale
32	4:1 Horizontal scale
16	8:1 Horizontal scale
8	16:1 Horizontal scale
4	32:1 Horizontal scale
2	64:1 Horizontal scale

$0F:$0002 - Vertical Scale

In general, this byte holds inverse of scale relative to 128. Although any value is recognized, these are the integer scale values:

Value	Description
128	1:1 Vertical scale
64	2:1 Vertical scale

32	4:1 Vertical scale
16	8:1 Vertical scale
8	16:1 Vertical scale
4	32:1 Vertical scale
2	64:1 Vertical scale

$0F:$0003 - Border Color Palette Index

This value contains the index into palette memory for the display border. If the Output mode is set to 0, this is ignored.

$0F:$0004 to $0F:$0008 - Display Area Settings

This group of addresses effectively controls the system display area. Regions of the display that fall outside of these bounds are colored with the system border color. This is after accounting for VGA and NTSC blanking times, so this system border subtracts from the 640x480 viewable display area.

The Vera also offsets all other displayed video by these settings, so that the top-left corner of the system display area always maps to 0,0.

$0F:$0008, in particular, is special. It contains the highest 2 bits of the horizontal start and end of the display area, and the highest bit of the vertical start and end of the display area, in the format: %00vVhhHH.

Field	**Description**
HH	Highest 2 bits of the Horizontal start of display area.
hh	Highest 2 bits of the Horizontal end of display area.
V	Highest bit of the Vertical start of display area.
v	Highest bit of the Vertical end of display area.

When appended at the top of the appropriate byte values in addresses $0F:$0004 to $0F:$0007, this makes the Horizontal start and stop 10 bits wide, and the Vertical start and stop 9 bits wide.

Palette

The Vera’s palette is 512 bytes, containing 256 colors, each 2 bytes in size and following the given format:

Byte 1	Byte 0
%0000RRRR	%GGGGBBBB

Field	Description
0000	Unused.
RRRR	Red component of palette color.
GGGG	Green component of palette color.
BBBB	Blue component of palette color.

This provides 16 shades of each color component for a total of 4096 distinct colors, of which 256 can be worked with at any given time.

Through programming treachery, it is technically possible to overwrite the palette partway through drawing to the screen and thereby extend the number of colors output in a single VGA frame or NTSC field. At present, this requires very careful timing!

Palette memory can also be thought of as being “subdivided” into 16 groups of 16 colors, with each group of 16 acting as a “16-color palette” for tiles and sprites using 4bpp color settings. These 4bpp tiles and sprites are assigned a 4-bit “palette index” which indicates which group of 16 colors make up their palette.

Layer Settings

Where X can be 2 or 3, representing Layer 0 or Layer 1, respectively, the following memory-mapped addresses control display layer behavior:

Address	Description
$0F:$X000	Mode and Enable flag for layer X.
When layer X in tile or text mode	
$0F:$X001	Layer X tilemap size.
$0F:$X002	Layer X tilemap base address (low byte)
$0F:$X003	Layer X tilemap base address (high byte)
$0F:$X004	Layer X tile data base address (low byte)
$0F:$X005	Layer X tile data base address (high byte)
$0F:$X006	Layer X horizontal scroll (low byte)
$0F:$X007	Layer X horizontal scroll (high byte, max of $0F)
$0F:$X008	Layer X vertical scroll (low byte)

$0F:$X009	Layer X vertical scroll (high byte, max of $0F)
When layer X in bitmap mode	
$04:$X001	Layer X bitmap size
$04:$X007	Layer X palette offset (max of $0F)

$0F:$X000 - Layer modes and enable flag

The least significant bit of $04:$00X0 is an enable bit. If set (1), the layer is drawn. If reset (0), the layer is not drawn.

The 3 most significant bits of $04:$00X0 represent the layer's "mode" setting, with the following possible values:

Mode	Description
0	Text mode, 1bpp glyphs with 16-color foreground and 16-color background
1	Text mode, 1bpp glyphs with 256-color foreground and 1-color background
2	Tile mode, 2bpp
3	Tile mode, 4bpp
4	Tile mode, 8bpp
5	Bitmap mode, 2bpp
6	Bitmap mode, 4bpp
7	Bitmap mode, 8bpp

Note: r27-r29 have a bug dealing with 2bpp and 4bpp image data, causing the draw routine to not draw any pixels of a tile after the first byte's worth of columns. This is fixed in r30.

Text and Tile mode settings

$0F:$X001 - Text and Tile format

Text and tile modes allow for 32x32 through 256x256 character/tile displays. At 1x scale, the 640x480 display allows for 80 columns and 60 rows of 8x8 character glyphs.

$0F:$X001 contains the tilemap settings for layer X, in the format %00ABCCDD:

Field	Description
00	Unused.
A	1 bit to enable 16-pixel tile height. Ignored in text modes.

B	1 bit to enable 16-pixel tile width. Ignored in text modes.
CC	2 bits to adjust tilemap height.
DD	2 bits to adjust tilemap width.

The text and tilemap height and width options are:

Value	Description
0	32 tiles or characters
1	64 tiles or characters
2	128 tiles or characters
3	256 tiles or characters

$0F:$X002 and $0F:$X003 - Tilemap base address

These addresses contain the base address, or starting address, of the tile or character indices that make up the layer's tilemap. These must be aligned to 4-byte intervals, and are stored as the interval index starting from $00:$0000.

$0F:$X004 and $0F:$X005 - Tile Data base address

These addresses contain the base address, or starting address, of the tile or character graphics data. Like the tilemap base address, these must be aligned to 4-byte intervals, and are stored as the interval index starting from $00:$0000.

Video RAM addresses, alignment, and interval indices

To allow fewer bits to address relatively large blocks of memory, some VRAM data cannot start at arbitrary addresses such as $0001, $0002, or $0003. They must start at a VRAM address that is a multiple of some value, and are specified to the Vera chip as the "index" of that multiple from some starting offset address.

For instance, Tilemaps and Tile Data must be aligned to 4 byte intervals. Some valid addresses to start tile indices and tile data at are $0000, $0004, $0008, and so on. Multiples of 4. These would be written, respectively, as $0000, $0001, and $0002 when writing the Tilemap base address and Tile Data base address.

If a tile map is 32x32 and is 2 bytes per tile, that is 2048 bytes. The data could start at $0000 and we would write $0000 as the Tilemap base address for that layer. The other layer could start at $0800 (address 2048), and we would write $0200 (index 512) as the Tilemap base address for that layer. Tile graphics data could then start at $1000 (address 4096), and we would write $0400 (index 1024) as the Tile Data base address for one or both of the layers.

$0F:$X006 and $0F:$X007 - Layer horizontal scroll

These addresses contain the horizontal scrolling offset of the layer, in pixels. $00X6 contains the low byte, and $00X7 contains the high byte. A layer is scrolled left as the value increases, and can be scrolled by up to $0FFF pixels.

$0F:$X008 and $0F:$X009 - Layer vertical scroll

These addresses contain the vertical scrolling offset of the layer, in pixels. $00X6 contains the low byte, and $00X7 contains the high byte. A layer is scrolled up as the value increases, and can be scrolled by up to $0FFF pixels.

Scrolling, wrapping, and negative numbers
Due to wraparound behaviors, $0FFF is conveniently equivalent to being scrolled -1 pixels. This means you can safely “underflow” a 16-bit value and write it to these addresses, and see the “correct” scrolling behavior as if the image had been scrolled to the right or downwards, instead.

Bitmap mode settings

$0F:$X001 - Layer bitmap size

This address sets the bitmap size of the layer in bitmap mode. If set to $10, the layer will be a 640x480 bitmap. If set to $00, the layer will be a 320x480 bitmap.

$0F:$X007 - Layer bitmap palette offset

Bitmap modes do not interleave palette data into the image data, the way tile modes do. Instead, $X007 stores the palette offset for the entire layer bitmap.

Bitmap palettes must be aligned to 32-byte intervals, and are presented to the Vera as the index of the interval starting from $0F:$1000 (start of the Palette Data section in VRAM).

Palettes
Colors in palette memory are 16-bit colors, represented as %ggggbbbb followed by %0000rrrr. Palette memory has exactly enough space for 256 colors, which allows any bitmap or sprite operating in 256-color mode to access the entire palette memory, as long as it’s set to a palette offset of 0.

When bitmaps and sprites use a more restrictive color selection, their palette address is specified as a 32-byte interval index, or 16 colors. This means there are 16 “palettes” for 4bpp images to choose from, with no overlap.

Text and Tilemap Data

Text Modes

In text modes, the Vera reads pairs of bytes from VRAM starting at the Tilemap base address for the layer. The first byte is treated as a “tile index” of the character at that location.

In mode 0, the second byte uses the top 4 bits as the foreground color from palette index 0, and the bottom 4 bits as the background color from palette index 0.

In mode 1, the second byte is treated as the palette color index for the foreground color from palette index 0, and the background color is always the first color at palette index 0.

Tile Modes

In tile modes, the Vera reads pairs of bytes from VRAM starting at the Tilemap base address for the layer. The data in these bytes is laid out like this:

Byte 1	Byte 0
%PPPPVHTT	%TTTTTTTT

Field	**Description**
TTTTTTTTTT	10-bit index of the tile data from the Tile Data base address. The low 8 bits are in Byte 0, the highest 2 bits are in Byte 1.
H	1 bit toggle whether the tile is horizontally flipped
V	1 bit toggle whether the tile is vertically flipped
PPPP	Palette index of the tile. This is 16-byte aligned and written as the interval index from $04:$0200 (Palette Data).

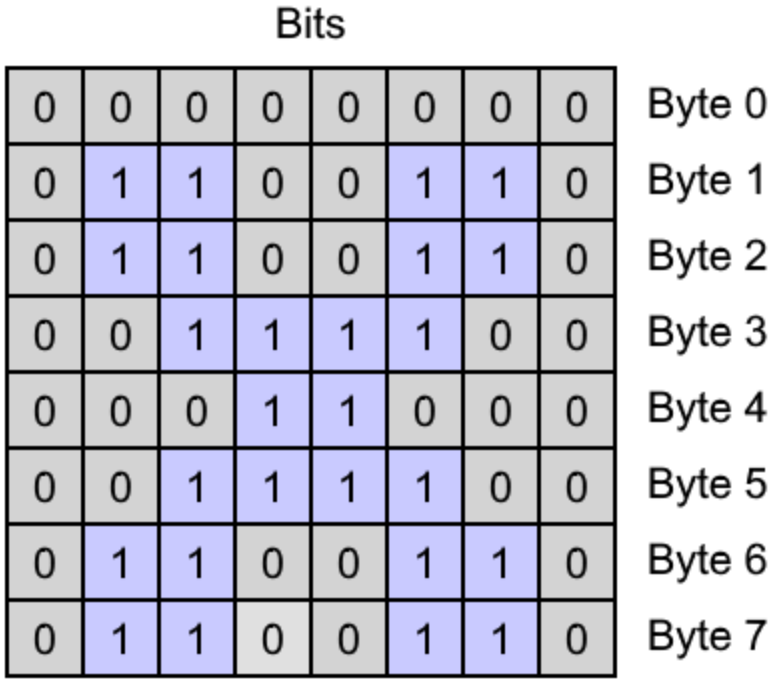
Tile Data

Text Modes

In text modes, all character glyphs are assumed to be 1bpp, 8x8 “tiles”, where a 0 in any given bit shows the background color, while a 1 shows the foreground color. Each character takes 8 bytes in VRAM.

Character glyph data is written into VRAM starting from the top-most row of pixels and working down, with the top bit of each byte representing the left-most pixel of the row, and the bottom bit representing the right-most pixel of the row.

Example layout of a character glyph in VRAM:



In the Acme Cross-Assembler, this might be represented as:

```
CHAR_DATA_X:
      !byte %00000000
      !byte %01100110
      !byte %01100110
      !byte %00111100
      !byte %00011000
      !byte %00111100
      !byte %01100110
      !byte %01100110
```

In X16 BASIC, you might also do this:

```
10 DIM X%(7)
20 X(0)=%00000000
30 X(1)=%01100110
40 X(2)=%01100110
50 X(3)=%00111100
60 X(4)=%00011000
70 X(5)=%00111100
80 X(6)=%01100110
90 X(7)=%01100110
```

Tile Modes

In tile modes, tiles can be 2bpp, 4bpp, or 8bpp. 2bpp tiles represent 4 pixels per byte, with 4bpp tiles representing 2 pixels per byte, and 8bpp tiles represent 1 pixel per byte. The smallest 8x8, 2bpp tile is 16 bytes; the largest 16x16, 8bpp tile is 256 bytes. At present, a color index of 0 is always considered to be “transparent” at that pixel, though it has been mentioned that future versions of the Vera, and thus the emulator, may add the ability to change which color index is considered “transparent.”

Tiles are laid out starting with the top-left most pixel, working right across the row, and then downwards for each row of the tile.

Example layout of an 8x8, 2bpp tile in VRAM:

Byte 0				Byte 1				
0	0	0	0	0	0	0	0	Bytes 0, 1
3	1	1	0	3	1	1	0	Bytes 2, 3
3	1	1	0	3	1	1	0	Bytes 4, 5
0	3	1	1	1	1	0	0	Bytes 6, 7
0	0	3	1	1	0	0	0	Bytes 8, 9
0	3	1	1	1	1	0	0	Bytes 10, 11
3	1	1	0	3	1	1	0	Bytes 12, 13
3	1	1	0	3	1	1	0	Bytes 14, 15

In the Acme Cross-Assembler, this might be represented as:

```
CHAR_DATA_X:
      !byte %00000000, %00000000
      !byte %11010100, %11010100
      !byte %11010100, %11010100
      !byte %00110101, %01010000
      !byte %00001101, %01000000
      !byte %00110101, %01010000
      !byte %11010100, %11010100
      !byte %11010100, %11010100
```

In X16 BASIC, you might also do this:

```
      10 DIM X%(15)
      20 X(0)=%00000000
      30 X(1)=%00000000
      40 X(2)=%11010100
      50 X(3)=%11010100
      60 X(4)=%11010100
      70 X(5)=%11010100
      80 X(6)=%00110101
      90 X(7)=%01010000
      100 X(8)=%00001101
      110 X(9)=%01000000
      120 X(10)=%00110101
      130 X(11)=%01010000
      140 X(12)=%11010100
      150 X(13)=%11010100
      160 X(14)=%11010100
      170 X(15)=%11010111
```

Similarly, in 4bpp modes, one byte represents 2 pixels, each of which can have one of 16 colors. Thus, 4 bytes are used to represent a single row of 8 pixels.

Bitmap Data

Bitmap data in VRAM is represented exactly like Tile Data, except the tile is enormous (320x480 or 640x480). Bitmap pixels are laid out starting with the top-left most pixel, working right across the row, and then downwards for each row of the image.

Sprite Settings

$0F:$4000 - Global sprite toggle

This address enables ($01) or disables ($00) the sprites layer.

$0F:$5000 to $0F:$5FFF - Sprite data

Sprite data layout changed significantly from r27 to r28. Many things moved around, all 8 bytes are used, and the Y position of sprite has been expanded to 10 bits to match the X position.

Each sprite in the sprite data block is 8 bytes in size. This provides room for 256 sprites, however the emulator (r28-r32) currently stops after the first 16. This is expected to be relaxed in r33 to 128 sprites, although additional rules will apply to limit the amount of work done while drawing.

Byte 3	Byte 2	Byte 1	Byte 0
%000000XX	%XXXXXXXX	%M000AAAA	%AAAAAAAA

Byte 7	Byte 6	Byte 5	Byte 4
%hhwwPPPP	%0000ZZVH	%000000YY	%YYYYYYYY

Field	**Description**
AAAAAAAAAAAA	12-bit address of the sprite's graphics data in VRAM, using 32-byte alignment and specified as the index offset into VRAM starting from $00:$0000.
M	1-bit color mode of the sprite. 0 for 4bpp, 1 for 8bpp.
XXXXXXXXXX	10-bit X position of the sprite. The low 8 bits are in Byte 2, the highest 2 bits are in Byte 3.
YYYYYYYYYY	10-bit Y position of the sprite. The low 8 bits are in Byte 4 the highest 2 bits are in Byte 5.
H	1 bit toggle whether the sprite is horizontally flipped
V	1 bit toggle whether the sprite is vertically flipped
ZZ	Z-Depth of the sprite. %00 to disable the sprite from being drawn. Otherwise, currently unused.
PPPP	Palette index of the sprite. This is 32-byte aligned and written as the interval index from $04:$0200 (Palette Data).
ww	2-bit width specifier for the sprite's graphics data.
hh	2-bit height specifier for the sprite's graphics data.
0	Unused bit.

The height and width specifiers for sprite's graphics data are:

Value	Description
0	8 pixels
1	16 pixels
2	32 pixels
3	64 pixels

Notes:

r27-r29 have a bug dealing with 2bpp and 4bpp image data, causing the draw routine to not draw any pixels of a sprite after the first byte's worth of columns. This is fixed as of r30.

r27-r31 have a bug dealing with h-flipped and v-flipped sprites which incorrectly reads sprite data. For the most part, this results in h-flipped sprites failing to flip the left-most column and instead drawing it one pixel higher than it ought to. V-flipped sprites, similarly, fail to flip the top-most row and instead draw the first "row" from the following "sprite" in VRAM. (A fix may have been identified, but hasn't been tested/committed yet.)

SPI

As of r31, the Vera provides the interface for SPI, which is also a means of accessing the SD card, albeit in a read-only fashion. SPI can also be accessed through the VIA2 chip, which is beyond the scope of this document but suggests that the I/O design is still in-flux.

Proceed with caution.

(I plan to dig into the SDCard's inner workings eventually, but for now that is beyond the scope of this document. Some documentation has been written, but not tested or studied carefully.)

$0F:$7000 - SPI I/O

The SPI I/O address reads from and writes to the selected SPI device. If the device is not busy, this can be written to send that byte to the SPI device.

When the SD card is the selected device, SPI I/O expects to have a 6-byte command written to it, each byte separated by a delay lasting until SPI Select returns 0 from its "busy" flag. As of r31, only the first byte of this command matters. After each command, it expects $FF to be written, following the same delay, but after each delay the port can be read from to receive the SPI command response, one byte at a time. The expected number of bytes to read from this port depends on the command given to the device, and is presently beyond the scope of this document.

If there is no byte in response (or if there is no SDCard mounted or selected), the port returns $FF.

$0F:$7001 - SPI Select

If a “1” is written to the SPI Select address, this selects the SDCard.

When reading from this port, it returns a byte with the following pattern: %B000000S

Field	Description
B	1 if the selected device is busy, 0 otherwise
S	1 if the SDCard is the selected device, 0 otherwise

Historical Reference

Tables and notes from previous versions of the emulator can be found here.

Using the memory-mapped addresses (r27-r30)

Prior to r31, the Vera’s interface was still mapped to the memory addresses $9F20 through $9F25, but had reversed the bank select and low byte VRAM address registers:

Address	Description
$9F20	Bank select and memory access stride
$9F21	High byte of VRAM address
$9F22	Low byte of VRAM address
$9F23	Data at VRAM address 0
$9F24	Data at VRAM address 1
$9F25	Toggle between setting VRAM address 0 and 1, and/or reset video to defaults
$9F26	Enable or disable interrupts from the Vera
$9F27	Interrupt flag

$9F20: Setting the VRAM “bank” and “stride”

The value at $9F20 uses the format %SSSSBBBB:

Field	Description
SSSS	4 bits for the “stride” or “automatic increment” after accessing the current channel’s VRAM address.

BBBB	4 bits for the “bank” of the current channel’s VRAM address.

The “bank” of a VRAM address is only 4 bits’ worth of address space, from $00 to $0F, and much of that goes unused. This makes up the lowest 4 bits of what we write to $9F20.

The upper 4 bits directly represented the memory access stride/auto-increment value, meaning strides could only represent 0-15.

$9F21 and $9F22 - Memory access address

If you think of VRAM as a 16-bit address, these two memory-mapped registers contain the high ($9F22) and low ($9F21) byte of the VRAM address you want to access on the currently selected channel.

Note that this byte order (“big-endian”, or “largest byte to smallest byte”) is the reverse of typical conventions for a 6502 processor and its derivatives (which would be “little-endian”), but that is outside of the scope of this resource.

$9F23 and $9F24 (and $9F25)

These work identically to r31.

$9F26 and $9F27 - Interrupts

These are the interrupt control and handling registers.

When the emulator triggers an interrupt, it moves the program counter to the address contained at $FFFE and $FFFF, which is inside ROM. The address contained here points to the kernal’s interrupt handler, itself also in ROM. The kernal stows the A, X, and Y registers (in that order), and then checks whether it’s dealing with a regular IRQ or a non-maskable interrupt. We care about the former, in which case the kernal sets the program counter to the address contained in $0314 and $0315, which are fixed RAM addresses that we can use to plug in our own IRQ handler. See also, the balloon example from Frank Buss. This all matches the behavior of the Commodore 64’s interrupt handler. Note that if the default IRQ handler at $0314 and $0315 is overridden, the program should either stow the values and jump to the default handler when done, or else restore the processor’s registers before the RTI instruction.

They are the best tool we have for identifying when a new frame begins.

When IRQs are enabled on the Vera, they are generated at the beginning of the VGA front porch, and its equivalent in NTSC display modes for each field.

$9F26 only has two valid values:

Value	Description
0	Disable IRQs from the Vera
1	Enable IRQs from the Vera

Reading from $9F27 is similar:

Value	Description
0	No IRQ has been generated by the Vera
1	An IRQ has been generated by the Vera

Writing to $9F27, however, is reversed:

Value	Description
0	Does nothing
1	Clear the IRQ that has been generated

This means that when you are about to exit from an interrupt handler, write a 1 to $9F27 to clear the Vera's interrupt status flag (this will allow it to generate future interrupts).\

VRAM Memory Layout (r27-r30)

Bank	Start	End	Purpose
$00	$0000	$FFFF	Video RAM
$01	$0000	$FFFF	Video RAM
$02	$0000	$FFFF	Video ROM (PETSCII Character Data)
$03	$0000	$FFFF	Unassigned
$04	$0000	$000F	Layer 0 Settings
$04	$0010	$001F	Layer 1 Settings
$04	$0020	$003F	Sprite Settings
$04	$0040	$005F	Composer Settings
$04	$0060	$01FF	Unassigned
$04	$0200	$03FF	Palette Data
$04	$0400	$07FF	Unassigned
$04	$0800	$0FFF	Sprite Data
$04	$1000	$FFFF	Unassigned

Sprites (r27)

$04:$0020 - Global sprite toggle

This address enables ($01) or disables ($00) the sprites layer.

$04:$0800 to $04:$0FFF - Sprite data

Each sprite in the sprite data block is 8 bytes in size. This provides room for 256 sprites, however the emulator (r27) currently stops after the first 16.

Sprites currently use only 6 bytes of their 8-byte block. That sprite data is laid out like this:

Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
%hhwwAAAA	%AAAAAAAA	%0000ZZMY	%YYYYYYYY	%PPPPVHXX	%XXXXXXXX

(Again, bytes 6 and 7 are unused)

Field	**Description**
XXXXXXXXXX	10-bit X position of the sprite. The low 8 bits are in Byte 0, the highest 2 bits are in Byte 1.
H	1 bit toggle whether the sprite is horizontally flipped
V	1 bit toggle whether the sprite is vertically flipped
PPPP	Palette index of the sprite. This is 16-byte aligned and written as the interval index from $04:$0200 (Palette Data).
YYYYYYYYY	9-bit Y position of the sprite. The low 8 bits are in Byte 2, the highest bit is in Byte 3.
M	Color mode of the sprite. 0 for 4bpp, 1 for 8bpp
ZZ	Z-Depth of the sprite. %00 to disable the sprite from being drawn. Otherwise, currently unused.
AAAAAAAAAAAA	12-bit address of the sprite's graphics data in VRAM, using 32-byte alignment and specified as the index offset into VRAM starting from $00:$0000.
ww	2-bit width specifier for the sprite's graphics data.
hh	2-bit height specifier for the sprite's graphics data.
0	Unused bit.

The height and width specifiers for sprite's graphics data are:

Value	Description
0	8 pixels
1	16 pixels
2	32 pixels
3	64 pixels

Composer Settings (r27)

The composer adjusts global video effects, in particular the output video mode, border color, and the size of the display area.

Address	Description
$04:$0060	Output mode and chroma toggle.
$04:$0061	Horizontal Scale
$04:$0062	Vertical Scale
$04:$0063	Border color palette index
$04:$0064	Horizontal start of display area
$04:$0065	Horizontal end of display area
$04:$0066	Vertical start of display area
$04:$0067	Vertical end of display area
$04:$0068	Course adjustments to display area

$04:$0060 - Output mode and chroma toggle

This field takes bytes in the format %00000CMM.

Field	Description
00000	Unused.
C	1 bit to disable chroma, a.k.a. “Enable black-and-white display”
MM	2 bits Mode parameter

The possible Mode settings are:

Value	Description
0	Disable video. Outputs a blue screen.
1	VGA mode
2	NTSC mode

$04:$0061 - Horizontal Scale

In general, this byte holds inverse of scale relative to 128. Although any value is recognized, these are the integer scale values:

Value	Description
128	1:1 Horizontal scale
64	2:1 Horizontal scale
32	4:1 Horizontal scale
16	8:1 Horizontal scale
8	16:1 Horizontal scale
4	32:1 Horizontal scale
2	64:1 Horizontal scale

$04:$0062 - Vertical Scale

In general, this byte holds inverse of scale relative to 128. Although any value is recognized, these are the integer scale values:

Value	Description
128	1:1 Vertical scale
64	2:1 Vertical scale
32	4:1 Vertical scale
16	8:1 Vertical scale
8	16:1 Vertical scale
4	32:1 Vertical scale
2	64:1 Vertical scale

$04:$0063 - Border Color Palette Index

This value contains the index into palette memory for the display border. If the Output mode is set to 0, this is ignored.

$04:$0064 to $04:$0068 - Display Area Settings

This group of addresses effectively controls the system display area. Regions of the display that fall outside of these bounds are colored with the system border color. This is after accounting for VGA and NTSC blanking times, so this system border subtracts from the 640x480 viewable display area.

The Vera also offsets all other displayed video by these settings, so that the top-left corner of the system display area always maps to 0,0.

$04:$0068, in particular, is special. It contains the highest 2 bits of the horizontal start and end of the display area, and the highest bit of the vertical start and end of the display area, in the format: %00vVhhHH.

Field	**Description**
HH	Highest 2 bits of the Horizontal start of display area.
hh	Highest 2 bits of the Horizontal end of display area.
V	Highest bit of the Vertical start of display area.
v	Highest bit of the Vertical end of display area.

When appended at the top of the appropriate byte values in addresses $04:$0064 to $04:$0067, this makes the Horizontal start and stop 10 bits wide, and the Vertical start and stop 9 bits wide.