VIXEN-32 Chipset Reference

Contents

About This Reference 2

Chapter 1 — System Memory Map 5

- The CPU's world 6
- Chip RAM (SDRAM) layout 7
- Chip RAM from the CPU (core v0.39) 7
- Running code from chip RAM (core v0.39) 8
- Trapdoor RAM (core v0.87) 9
- Zorro RAM (core v0.89) 10
- VIH register space at a glance 10

Chapter 2 — VIH: The Display Engine 11

- Features 11
- Layer stack (bottom to top) 11
- The framebuffer 11
- Registers 12

Chapter 3 — Tile Map & Fonts 17

- The tile layer 17
- Fonts 18
- Hi-res text (core v0.95) 18

Chapter 4 — Hardware Sprites 19

- Features 19
- Registers 19
- Timing note 20

Chapter 5 — The Copper 21

- Copper register IDs 21
- Timing 22

Chapter 6 — The Blitter 22

- Operations 22
- The CPU's read window 23
- Registers 23
- Z buffer (core v0.90) 26
- Perspective spans (core v0.91) 27
- Triangles (core v0.93) 28
- Software 3D 30

Chapter 7 — SID 30

Chapter 8 — PCM: The Sample Engine 30

- Control block 31

Beeper 31
Mixer 31
Chapter 9 — SYSTEM: CPU, Interrupts, I/O 32
The CPU 32
8-bit CPU block (core v0.95) 33
Memory protection 41
Interrupts (the "KERNAL" model) 41
Keyboard 42
Joystick 42
Chip RAM access 42
File loading 43
OWN_RELEASE — `$3A00019C` (W) 43
SDRAM_PHASE — `$3A0001E0` (W) — platform tuning 43
Chapter 10 — Booting from the SD Card 44
The chain 44
The reset button (core v0.80/v0.81) 44
Which SD slot (core v0.84) 44
The kernel image must carry its own reset page 45
There is no fallback 45
Error screens 45
Border colors 45
Reading the card from your own code 46
Chapter 11 — Ethernet 47
The buffers are memory, not a port 47
Registers 48
The protocol stack 48
Chapter 12 — Virtual Floppy Drives 49
How a sector actually moves 49
Registers 49
The sector buffers 50
From C 50
Bootable disks 50
Appendix A — Register Index (by address) 50
Appendix B — BASIC POKE Quick Reference 55

About This Reference

This is the register-level hardware reference for the VIXEN-32 computer, in the tradition of the *Commodore 64 Programmer's Reference Guide*, the *Amiga Hardware Reference Manual* and the *MEGA65 Chipset Reference*: one chapter per functional chip, an introduction to each chip's programming model, and a bit-exact table for every register.

VIXEN-32 is a single FPGA design; the "chips" below are subsystems, but they behave — and are documented — like a classic chipset:

"Chip"	Role
VIH	Video Interface Hardware: display engine, layers, palette
COPPER	Per-scanline register processor (the display list)
BLITTER	Block transfer, lines, textured & shaded spans, SDRAM window
SID	MOS 6581 sound (3 voices, filters)
PCM	4-voice sample player (Paula-style)
SYSTEM	CPU, interrupts, keyboard, joystick, file loading, chip RAM

Conventions:

- `$` prefixes hexadecimal. Register addresses are 32-bit CPU addresses (the RISC-V's view). All registers are 32 bits wide; write the whole word.
- "POKE n" refers to the BASIC machine's 6502-side address map (see `docs/BasicMachine.txt`), *not* these CPU addresses.
- Bit ranges are written `[high:low]`, bit 0 = least significant.
- (R) = readable, (W) = writable, (R/W1C) = read status, write 1 to clear.
- The C API in `risc/vih_regs.h` wraps every register described here; the struct/union names are given with each register.

Host platform: MEGA65 R6 (Xilinx Artix-7 XC7A200T), MiSTer2MEGA65 framework up to core v0.93; from v0.94 the core runs without it: no F12 menu, the kernel loads files itself, and the core loads `/VIXEN-32/KERNEL.BIN` (v0.93 and older: `BOOT.BIN`). Video: 15 kHz analog PAL on the VGA port as always, and HDMI 720x576 at 50 Hz with sound (v0.94: the same picture, all 384 pixels across the 720, each line twice, locked to the PAL output – 624 lines a frame, 50.08 Hz). CPU: VexRiscv "Murax", RV32IM+Zicsr, 100 MHz, **512 KB** zero-wait RAM at address 0; full result forwarding and a single-cycle barrel shifter (core v0.37/v0.38). Backing store: 64 MB onboard SDRAM, of which the first 16 MB is chip RAM — readable, writable **and executable** by the CPU (core v0.39), so a program is no longer limited to the 512 KB of zero-wait RAM. Since core v0.89 the other 48 MB is **Zorro RAM**: more memory for the CPU alone.

New since core v0.76 (all of it either invisible to existing programs or off until a program switches it on):

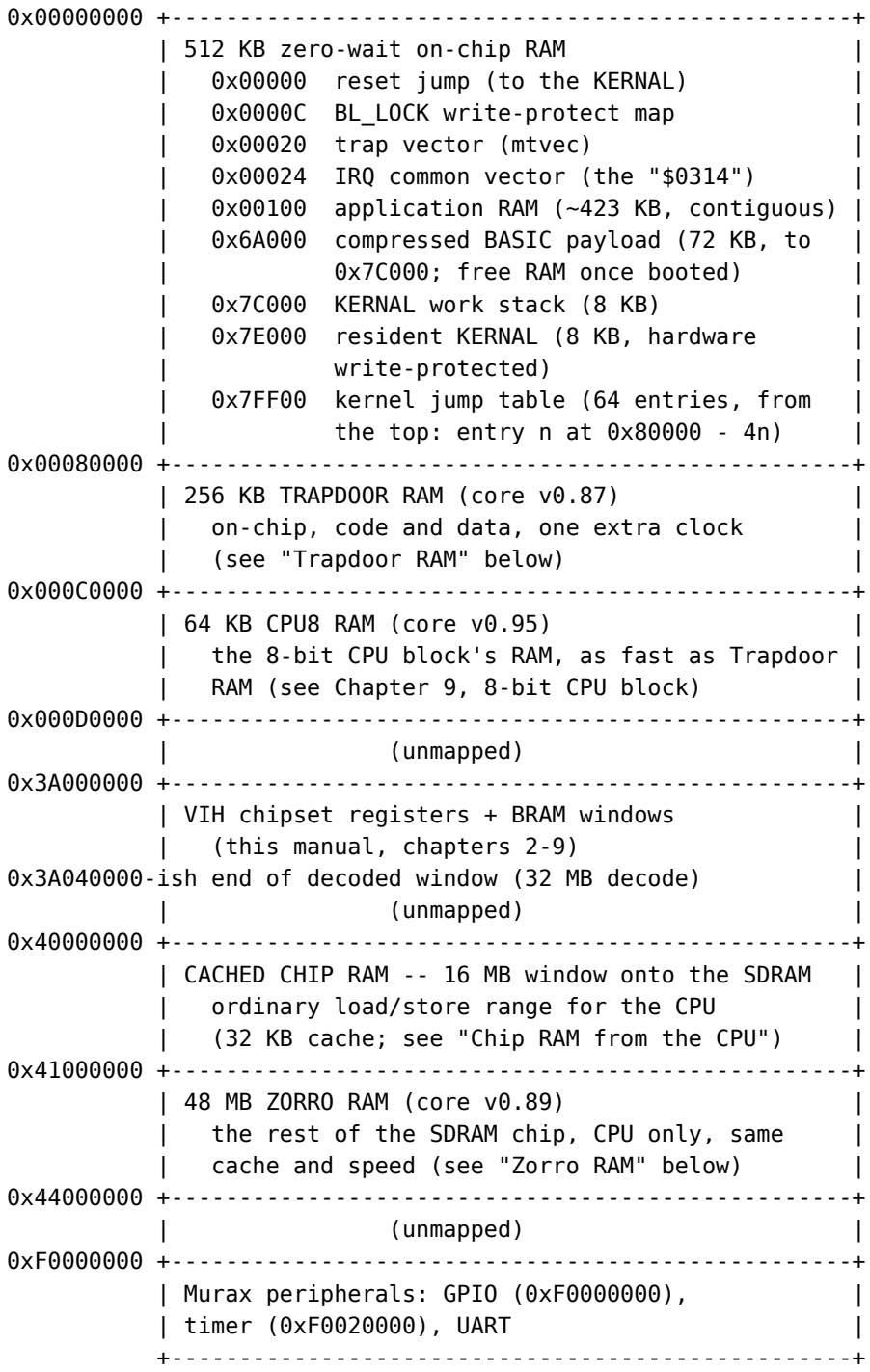
Core	Change	Where
v0.77	A running blit pauses, instead of being cut off, while the CPU writes chip RAM	Chapter 6
v0.78	Top and bottom border never 0 (a 0 is stored as 1)	Chapter 2, BORDER_DIMS
v0.79	Every CPU write reaches the SID, one per 1 MHz tick	Chapter 7

Core	Change	Where
v0.80–v0.81	The reset button reloads the kernel from the card; the screen starts clean	Chapter 10
v0.82	FPGA temperature register	Chapter 9, TEMP
v0.83	HDMI: HyperRAM controller placed next to its pins	—
v0.84	Boots from the internal SD slot when the back slot is empty	Chapter 10
v0.85	Optional 256-deep blitter op queue	Chapter 6, BLIT_QUEUE
v0.86	1024 characters in the 8bpp font (tile cell bit 27)	Chapter 3
v0.87	256 KB **Trapdoor RAM** above the 512 KB	Chapter 1
v0.88	Optional 8 KB instruction cache for code above the 512 KB	Chapter 9, ICACHE
v0.89	48 MB **Zorro RAM**: the rest of the SDRAM chip, for the CPU only	Chapter 1
v0.90	Optional blitter **Z buffer**: polygons need no sorting	Chapter 6, Z buffer
v0.91	Optional **perspective-correct** textured spans	Chapter 6, Perspective spans
v0.92	One-color polygons with the Z buffer on draw at plain-fill speed	Chapter 6, Z buffer
v0.93	Optional **blitter triangles**: one op draws a whole triangle	Chapter 6, Triangles
v0.95	HDMI **zoom** (the picture fills the height, sent as 16:9) and **scanlines**; zoomed, **soft CRT scanlines**	Chapter 2, HDMI
v0.95	**8-bit CPU block**: a 6502 and a Z80 with 64 KBytes of their own RAM, run by the RISC-V	Chapter 9, 8-bit CPU block
v0.95	The VIC-20's two **6522 VIAs** for the 6502, with its keyboard and joystick	Chapter 9, 8-bit CPU block
v0.95	The VIC-20's **VIC** (6561, PAL): its picture into chip RAM, its sound into the mix; the 6502 on the VIC's clock	Chapter 9, 8-bit CPU block
v0.95	The **YM2151** (OPM): 8 FM voices of 4 operators in stereo, sample for sample as ymfm; the Z80 drives it on its own through two ports or addresses, its timers on the Z80's INT, as on arcade sound boards	Chapter 9, 8-bit CPU block

Core	Change	Where
v0.95	The 8-bit CPU block's **page map** (each 1 KByte RAM, ROM, absent or trapped: every VIC-20 memory layout and cartridge) and **context switch** (every bit of the VIC, the VIAs and the CPU read and written; resume in the VIC cycle where it stopped)	Chapter 9, 8-bit CPU block
v0.95	**8 x 256 palettes**: each layer, tile cell and sprite in one of eight palettes, the copper switching them per line	Chapter 2, PAL_SEL
v0.95	**Hi-res text**: the tile layer at 768 dots a line, 80 columns, over the unchanged picture and sprites	Chapter 3, Hi-res text
v0.95	The picture and the sprites **exactly on their coordinates**: framebuffer column x and sprite x at screen x (the picture was one pixel right, sprites one pixel left)	Chapter 2, The framebuffer; Chapter 4
v0.95	The 8-bit CPU block's **64 KBytes at `$000C0000`**, straight after Trapdoor RAM and as fast; with no 6502 or Z80 program it is 64 KBytes more RAM for the RISC-V	Chapter 1, The CPU's world; Chapter 9, 8-bit CPU block
v0.95	**32-bit stores to chip and Zorro RAM almost twice as fast**: one SDRAM burst instead of two writes (byte and halfword stores unchanged)	Chapter 1, Chip RAM from the CPU
v0.96	A **Commodore 64** in the 8-bit CPU block: a VIC-II (6569, PAL), two 6526 CIAs and the PLA beside the 6502, on the machine's SID, whose OSC3 and ENV3 now read back	Chapter 9, 8-bit CPU block
v0.96	The C64's **cartridge port**: 16 KBytes of ROM, a second ROMH bank for Zaxxon, and a **trap map** (a bit for every 16 bytes) that gives `$DE00-$DFFF` to the RISC-V	Chapter 9, 8-bit CPU block
v0.96	**Paddles and the 1351 mouse**: the C64's POTX / POTY on control port 1 or 2, the VIC-20's `$9008` / `$9009`	Chapter 9, 8-bit CPU block
v0.97	The C64's **CIAs never stand**: they count on while the 6502 waits for the RISC-V's answer to a trap, as a C64's do	Chapter 9, 8-bit CPU block
v0.98	The C64's **tape**: a Datasette's READ line into CIA 1's FLAG from queued pulses, cycle for cycle, PLAY and the motor on the 6510's port – turbo loaders load	Chapter 9, 8-bit CPU block
v0.98	**CORE_VERSION**: the core's version, read-only, set by the build	Chapter 9, CORE_VERSION

Chapter 1 — System Memory Map

The CPU's world



The SDRAM is the chipset's "chip RAM". Architecturally it belongs to the chipset — the display, blitter and audio read it directly. Put your code and data in the zero-wait BRAM above and the display never competes with code fetches at all. That is deliberate, Amiga-style, and it is still the fastest place to be.

Since core v0.34 the CPU can also reach chip RAM **directly**, as an ordinary pointer range at `0x40000000` (see "Chip RAM from the CPU" below), and since **core v0.39 it can run code from there too** — which is how a program gets to be larger than the 512 KB of BRAM. Code fetched from chip RAM does share the memory with the display, so it is slower than BRAM; that is the trade, and it buys megabytes. The older indirect paths — the VIH write FIFO, the blitter, and the blitter's read window (Chapter 6) — all still work and are still the right tool for bulk transfers.

Chip RAM (SDRAM) layout

64 MB total; the chipset's byte-addressed bus covers the first 16 MB ("chip RAM"). Since core v0.89 the remaining 48 MB is Zorro RAM: the CPU reaches it at `0x41000000` (see "Zorro RAM" below); the blitter, display and audio still see only the first 16 MB.

Byte address	Contents
`0x000000`	Sprite image slots: slot *n* at *n*·4096, 64x64 bytes
…	(slots, textures, samples, framebuffers — app layout)
`0x800000`	File-loading staging area (Chapter 9)
`0xFFFFFF`	end of chip RAM

Framebuffers may live at any 4096-byte boundary (Chapter 2). PCM samples at any word address (Chapter 8). There is no hardware-enforced layout apart from the staging area convention.

Chip RAM from the CPU (core v0.39)

A 16 MB window at `0x40000000` maps chip RAM into the CPU's address space through a **32 KB cache**, so chip RAM is just memory (since core v0.89 the window is 64 MB: the 48 MB above chip RAM is Zorro RAM, same cache):

```
volatile uint8_t *p = VIH_SDRAM_MEM;    /* 0x40000000 */

p[addr] = 0x42;                          /* write */
v = p[addr];                             /* read  */

/* or, address-masked into the window: */
*(volatile uint32_t *)vih_sdram_ptr(0x600000) = 0xDEADBEEF;
```

The cache is direct-mapped with 32-byte lines and is **write-through**: every CPU write goes to SDRAM immediately, so the display, blitter and audio always see what the CPU wrote. A read costs three cycles on a hit; a miss fetches one 32-byte line.

A store does not stop the CPU: it goes on at once, and only its next read or write of chip or Zorro RAM waits for the store to reach SDRAM. A 32-bit store takes about 23 cycles there, a byte or halfword store about 21 (core v0.95; before, a 32-bit store was two writes, about 42).

So fill and copy chip RAM a word at a time, not a byte at a time: a word loop clears 1 MB of chip RAM in about 62 ms (17 MB/s).

The one rule. The cache only watches the CPU. If *anything else* writes a region the CPU has read before, the CPU may still see the old bytes — invalidate first. There are two things that count:

```
/* 1. after the blitter writes */
vih_blit_fill(addr, 0, 64, 1, 0x77);
vih_blit_wait();
vih_cache_invalidate();      /* now the CPU sees the new bytes */

/* 2. after a file is loaded into chip RAM */
vih_cache_invalidate();      /* before reading the loaded data */
```

A loaded file is put into chip RAM by the loader, not by the CPU, so any line the CPU already holds for those addresses is stale — exactly like a blitter write. Skip this and a freshly loaded file reads back as a mixture of old and new bytes, which looks like data corruption rather than a missing invalidate.

CPU writes never need this; write-through makes them visible to everyone immediately. Invalidate takes about 10 µs (it walks all 1024 lines), so do it once after a batch of blitter work rather than per operation.

Running code from chip RAM (core v0.39)

Chip RAM is not just for data — **a program can run from it**. That lifts the ceiling on program size from the CPU's 512 KB of fast RAM to megabytes: a 1.5 MB program runs today.

It is slower than running from CPU RAM, because instructions arrive through the same 32 KB cache as everything else. The arrangement that works:

put here	what
CPU RAM (`0x00000`)	hot loops, frame buffers you write per pixel, anything touched every frame
chip RAM (`0x40000000`)	bulk code, level data, textures, sounds

Nothing special is needed to do it — link the program for the window and jump there. Two things to know:

- **Load-time invalidate.** The program arrives in chip RAM by the loader, so the invalidate rule above applies before the first instruction is fetched, not only to data.
- **Size, not speed, is the reason to do it.** A program that fits in CPU RAM should stay there; it will be faster.

`risc/cachetest4` is the test that proves the path, and is worth re-running after any change here: it generates 160 KB of code — five times the cache — runs it, repeats it, and then runs it again while data traffic evicts it.

Address	Name	Access	Meaning
`0x40000000`	cached chip RAM	R/W	16 MB window, ordinary memory
`0x41000000`	Zorro RAM (v0.89)	R/W	48 MB, CPU only, same cache
`0x3A0001E4`	`VIH_CACHE_INVAL`	W	any write invalidates the cache

`vih_sdram_read()` (Chapter 6) is still worth using for one-shot bulk reads you do *not* want to evict the cache with.

Since core v0.88 the optional **instruction cache** (Chapter 9) makes code in chip RAM much faster: a loop that fits its 8 KB runs at the speed of CPU RAM.

Trapdoor RAM (core v0.87)

256 KB of extra on-chip RAM directly above the 512 KB of CPU RAM, at `0x00080000`–`0x000BFFFF`. Named after the Amiga 500's trapdoor expansion: in speed it sits between CPU ("fast") RAM and chip RAM.

- Plain memory for **code and data**, nothing else in the machine uses it.
- Every read costs **one extra clock**; writes are single-cycle. Measured on the board: code runs at 66% of CPU-RAM speed, data at 90%. With the instruction cache on (Chapter 9) code runs at 99%.
- Not cleared at reset or power-on; contents are undefined until written.
- Straight after it, at `0x000C0000`–`0x000CFFFF`, are the 8-bit CPU block's 64 KBytes (core v0.95, Chapter 9): the same speed and the same instruction cache. While no 6502 or Z80 program uses them they are plain RISC-V RAM too, so `0x00080000`–`0x000CFFFF` is 320 KBytes at one speed; a 6502 or Z80 running there never slows Trapdoor RAM down.
- The 512 KB below, its speed and the released memory map are unchanged — existing programs cannot notice it.
- On cores before v0.87 nothing answers there. Check first:

```
if (vih_trapdoor_present())
{
    uint32_t *big_table = (uint32_t *)TRAPDOOR_RAM_BASE;    /* 256 KB */
}
```

Address	Name	Meaning
`0x00080000`	`TRAPDOOR_RAM_BASE`	first byte
—	`TRAPDOOR_RAM_SIZE`	`0x40000` (256 KB)

A `.rvk` cannot yet place blocks here directly (the kernel's .rvk loader only accepts the 512 KB and chip RAM); copy code or data in at start-up.

Zorro RAM (core v0.89)

The other 48 MB of the SDRAM chip, straight after chip RAM at `0x41000000–0x43FFFFFF`. Named after the Amiga's Zorro expansion slots, whose memory boards only the CPU could use — the same deal here.

- **CPU only.** The blitter, the display, the sprites and the audio cannot see it. Keep assets here — levels, textures, samples, music, the next scene — and copy what is needed into chip RAM with the CPU. A plain word copy from Zorro RAM to chip RAM runs at about 13 MB/s on the board (core v0.95; 8 before), over thirty times faster than loading the same data from the SD card.
- **Same path as chip RAM.** It goes through the same 32 KB data cache and the same instruction cache, at the same speed: data reads at 100% of chip RAM; code runs at 46% of CPU-RAM speed with the instruction cache off and up to 100% with it on.
- **The same rules as chip RAM.** Write-through cache; after writing NEW code here, call `vih_cache_invalidate()` before running it.
- Not cleared at reset or power-on; contents are undefined until written. Nothing else in the machine uses it.
- Chip RAM, the chipset and the released memory map are unchanged — existing programs cannot notice it.
- On cores before v0.89 nothing answers there. Check first:

```
if (vih_zorro_present())
{
    uint8_t *assets = (uint8_t *)ZORRO_RAM_BASE;    /* 48 MB */
}
```

Address	Name	Meaning
`0x41000000`	`ZORRO_RAM_BASE`	first byte
—	`ZORRO_RAM_SIZE`	`0x3000000` (48 MB)

A `.rvk` cannot yet place blocks here directly (the kernel's .rvk loader only accepts the 512 KB and chip RAM); load or copy data in at start-up. `risc/zorro_ram_test` is the board test: all 48 MB, code run from every MB with the instruction cache off and on, while the blitter, sprites and a GFX picture run in chip RAM.

VIH register space at a glance

Full per-register detail is in the chapters; Appendix A is the sorted index. Regions:

Address range	Contents
`$3A000004..$3A00000C`	Raster position (read)
`$3A000100..$3A00019C`	Control registers (write) + status (read)
`$3A0001A0..$3A0001CC`	Blitter

Address range	Contents
`$3A0001E0`	SDRAM clock phase (platform tuning)
`$3A038A00`	Tile map 0 window (4096 cells x 32 bit)
`$3A040200`	Tile map 1 window

Chapter 2 — VIH: The Display Engine

Features

- 384 x 224 active pixels, 8 bits per pixel, indexed.
- 256 simultaneous colors from 65,536 (RGB565 palette, fully writable); since core v0.95 eight such palettes, each layer (and each tile cell and sprite) showing one of them, the copper switching them per line: up to 2,048 colors on screen.
- Since core v0.95 the tile layer can be **hi-res text**: 768 dots a line (a dot = half a pixel), 80 columns over the picture (Chapter 3).
- True PAL: 15,625 Hz horizontal, 50.08 Hz vertical, total raster 500x312.
- Three layers + border, composited per pixel (Chapter order below).
- Pixel-accurate horizontal and vertical smooth scrolling.
- Output: 15 kHz RGB with composite sync on the analog port (CRT-first), HDMI 720x576 at 50 Hz with sound in parallel (core v0.94; the framework's 720p50 before).

Layer stack (bottom to top)

```
Border (outside the active window)
Background color (transparent index 0)
Sprites with ON_TOP = 0
Framebuffer (GFX layer)
Tile map                    <- order vs GFX selectable (TILE_PRIORITY)
Sprites with ON_TOP = 1
```

The framebuffer

The GFX layer streams from chip RAM. `GFX_PTR` holds a *block index*: the framebuffer starts at `gfx_ptr * 4096` in SDRAM.

- One byte per pixel, row stride **512** bytes (a circular 512-pixel line: 384 visible, 128 of scroll headroom).
- **256 rows** per buffer: 224 visible + 32 off-screen for vertical scroll.
- A buffer is therefore 128 KB (256 x 512).
- `GFX_PTR` is latched at frame start: writing it mid-frame flips buffers **tear-free**. Double/triple buffering is pointer arithmetic.
- Palette index 0 is transparent: `BG_COLOR` (or a behind-sprite) shows.

- Column *x* of a line shows at screen x, border included: with the default border, columns 32-351 are the 320-pixel picture (core v0.95; before, every column showed one pixel to the right).

Registers

RASTER_HPOS — `$3A000004` (R)

RASTER_VPOS — `$3A00000C` (R)

Current beam position. VPOS counts 0..311 (write races with the copper are impossible — these are read-only). The frame-sync idiom is a raster spin:

```
static void vblank(uint32_t line)      /* returns exactly once per frame */
{
    while (VIH_RASTER_VPOS != line) {}
    while (VIH_RASTER_VPOS != (line + 1u)) {}
}
```

Call `vblank(216)` before buffer flips (just ahead of the frame-start latch), `vblank(250)` for general once-per-frame pacing.

CTRL_WORD — `$3A000114` (W) — `vih_ctrl_word_t`

Bit(s)	Name	Contents
0	active_buf	legacy framebuffer select (use GFX_PTR instead)
1	pal_reload	with bit 2: see below
2	tile_reload	with bit 1: see below
[31:3]	frame_counter	increment = “present a frame”

Writing bits 2+1 **together** triggers the chipset **soft reset**: every register returns to its boot default and all ownership latches clear (the app-launch path uses this). Since core v121 the frame counter is a legacy field: `GFX_PTR` latches at every frame start on its own, so apps never need to touch CTRL_WORD for display or buffer flips.

BORDER_DIMS — `$3A000130` (W) — `vih_border_dims_t`

Bit(s)	Name	Contents
[7:0]	left	border width in pixels
[15:8]	right	
[23:16]	top	
[31:24]	bottom	

Boot default 32/32/12/12 = a classic 320x200 window.

NEVER SET TOP AND BOTTOM BOTH TO 0.

The vertical border is not just blanking — it is when the display engine re-latches the sub-8-pixel scroll offset and starts the line fetch. Take every one of those lines away and two things go wrong:

- **Fine X scrolling stops working.** The offset never re-latches, so the picture jumps a whole 8-pixel cell instead of moving a pixel at a time. Smooth scrolling is impossible.
- **The layer can come up displaced sideways** by a multiple of 8 pixels, and by a different amount from one power-on to the next.

top >= 1 and bottom >= 1. One line each is enough; the boot default of 12 is comfortably clear of it. This is a rule about the hardware, not a style preference.

Since core v0.78 the core enforces it: a 0 for top or bottom — from the CPU or from the copper — is stored as 1, so the two faults above can no longer happen. Left and right may still be 0. On older cores a program that asks for 0 gets 0, and gets the behaviour above.

BORDER_COLOR — `$3A000104` (W)

`[7:0]` = palette index. Peer register: the copper may also write it (last write wins) — raster bars work from either side.

BG_COLOR — `$3A000108` (W)

`[7:0]` = palette index shown through transparent (index 0) pixels. The `POKE 53281` of this machine.

GFX_PTR — `$3A000158` (W)

`[12:0]` = framebuffer block index (x 4096 bytes). Latched per frame.

GFX_SCROLL — `$3A00015C` (W) — `vih_gfx_scroll_t`

Bit(s)	Name	Contents
[8:0]	x	0..511, circular on the 512-px line
[23:16]	y	0..255, circular on the 256-row buffer

Since core v125 the X scroll is **per-scanline exact**: a copper program rewriting `X_SCROLL` mid-frame gives each band its own pixel-smooth speed (the fine phase travels with each fetched line, so fine and coarse can never disagree). Banded bitmap parallax and Out Run-style roads are one copper entry per band — see `risc/tuts/tut13`.

GFX_ENABLE — `$3A000134` (W)

`[0]` = GFX layer on/off (off shows BG_COLOR). Peer with the copper.

PALETTE — `$3A000154` (W) — `vih_palette_t`

Bit(s)	Name	Contents
[15:0]	rgb565	color value
[23:16]	index	palette entry 0..255

Bit(s)	Name	Contents
[26:24]	palette	which of the 8 palettes (core v0.95; 0 before)

One write sets one entry; loop for a full palette. The copper can write this too (per-scanline gradients: register ID `$14`).

8 x 256 palettes (core v0.95). There are eight palettes of 256 entries. Palette 0 is the one every older program writes (bits [26:24] = 0) and sees; all eight power up the same (the C64 colors, the grey ramp, black). `vih_set_palette8(palette, index, rgb565)`. Which palette a pixel is looked up in depends on the layer it comes from (`PAL_SEL`).

PAL_SEL — `$3A000294` (W) — `vih_pal_sel_t` (core v0.95)

Bit(s)	Name	Contents
[2:0]	picture	palette of the GFX picture (and of `BG_COLOR`)
[6:4]	tiles	palette of the tile layer
[10:8]	sprites	palette of every sprite
[14:12]	border	palette of the border

A tile cell's own palette (cell bits [30:28]) and a sprite's own (`SPRITE_ATTR` [24:22]) are **added** to their layer's, mod 8: with the layer at 0 they pick the palette directly. 0 at power-on and from `vih_init_registers()`: every layer in palette 0, as before. The copper writes the same value (register ID `$16`), so the picture can be shown in bands of different palettes – each band up to 256 colors of its own, e.g. a video whose strips each have their own palette. Write-only; cores before v0.95 ignore the write.

TILE_PRIORITY — `$3A000124` (W)

`[0]` = 1: tile layer above GFX (default); 0: below.

HDMI — `$3A000290` (W) — `vih_hdmi_t` (core v0.95)

How the HDMI output shows the picture. The user's display setting: only the HDMI output reads it, the 15 kHz analog PAL picture never changes, and it costs no CPU time.

Bit	Name	
`[0]`	zoom	1: VIX-EN's 224 lines fill all 576 HDMI lines (ev-

Bit	Name	
		ery 7 lines become 18, so 2 or 3 HDMI lines each) and the frame is sent as 16:9 (VIC 18), so a TV widens it back to within 4% of the nor- mal shape. 0: each line twice, 64 black lines above and be- low, 4:3 (VIC 17)
[1]	scanlines	1: the sec- ond HDMI

Bit	Name	
		line of each VIXEN line at half brightness. Zoomed: soft CRT scanlines – hard ones would be uneven, the lines being 2 or 3 tall, so each VIXEN line is drawn as a beam, bright in its middle and darker at its edges, an HDMI line showing the two nearest VIXEN lines weighted by

Bit	Name	
		its distance from each

Taken at the start of a frame, so a frame is always whole. 0 after a reset; `vih_init_registers()` and a .rvk launch leave it alone, so the kernel can restore the user's choice at boot. Switching zoom moves the HDMI frame once against the machine (as a reset does), so some screens blank for a moment; the rate stays 50 Hz. Write-only; cores before v0.95 ignore the write.

Chapter 3 — Tile Map & Fonts

The tile layer

One merged 128-row tile BRAM holds two 64x64 maps' worth of contiguous rows. 8x8 (or 8x16) tiles; 48 x 28 tiles visible.

The displayed window starts at a **row-granular base pointer** (0..127): `+64` is a classic two-screen flip, `+1` scrolls the window one tile-row through the whole 128-row space ("tall scroll").

Tile cell format (32 bits per cell)

Bit(s)	Name	Contents
[7:0]	char	glyph index low 8 bits
[15:8]	fg	foreground palette index (1bpp mode)
[23:16]	bg	background palette index (1bpp mode)
24	h-flip	mirror horizontally
25	v-flip	mirror vertically
26	char8	glyph index bit 8 (512 glyphs total)
27	char9	glyph index bit 9 — **8bpp font only** (1024 glyphs, core v0.86); ignored by the 1bpp, 8x16 and ROM fonts
[30:28]	palette	the cell's own palette (core v0.95), added to `PAL_SEL`'s tile palette, mod 8

Cells are ordinary memory to the CPU: **tile map 0** at `$3A038A00`, **tile map 1** at `$3A040200`, 4096 cells x 4 bytes each, read/write.

Registers

TILEMAP_PTR — `$3A00016C` (W): `[6:0]` = display base row 0..127.

TILE_SCROLL — `$3A00011C` (W) — `vih_tile_scroll_t`:

Bit(s)	Name	Contents
[8:0]	x	pixel scroll 0..511
[24:16]	y	pixel scroll 0..511

TILES_ENABLE — `$3A00012C` (W): `[0]` = layer on/off.

TILE_COLOR_MODE (copper ID `$06` / FONT_CTRL): 0 = 1bpp fg/bg, 1 = 8bpp glyphs (every glyph pixel is a palette byte).

TILE_BG_TRANS (copper ID `$07`): 1 = bg index 0 transparent (GFX shows through the text).

FONT_CTRL — `$3A000110` (W) — `vih_font_ctrl_t`

Bit(s)	Name	Contents
0	font_src	0 = ROM font, 1 = BRAM font
1	font_height	0 = 8x8, 1 = 8x16

Fonts

- **ROM font**: the built-in character set, 8x8, 1bpp — the MEGA65 ASCII font **by deft**, used with permission and with our thanks. Greetings, deft — this is the face VIXEN-32 wears. (v0.42; it was the C64 set before, which had no lowercase.) ASCII-ordered, so the char code *is* the ASCII code. Codes `$A0-$D4` hold the V32 boot logo as 8x8 cells; `$D5-$F7` are blank and free.
- **BRAM font**: 512 uploadable glyphs, 8x8 or 8x16, 1bpp.
- **font8**: 512 uploadable 8bpp glyphs (a palette byte per pixel); **1024 since core v0.86** (64 KB). Tile cell bit 27 picks 512–1023 — enough for a 40x25 screen of all-different 8x8 glyphs. Glyphs 0–511 are where they always were.

FONT_WR_ADDR — `$3A000170` (W) / FONT_WR_DATA — `$3A000174` (W)

`FONT_WR_ADDR`: `[16]` selects the BRAM (0 = 1bpp font, 1 = font8), `[11:0]` the entry index (`[12:0]` since core v0.86: font8 entries 4096–8191 = characters 512–1023; the 1bpp font uses `[9:0]` as before). Then write `FONT_WR_DATA` in pairs — low 32 bits, high 32 bits — each pair commits one 64-bit entry and auto-increments. Helpers: `vih_font1_glyph()`, `vih_font1_glyph_8x16()`, `vih_font8_glyph()`.

Hi-res text (core v0.95)

With `VIDEO_MODE` [0] = 1 the tile layer is drawn at twice the horizontal resolution: 768 **dots** a line (a dot every 64 ns, 15.625 MHz), 640 of them across the picture, so 80 columns of 8 x 8 text fit where 40 did. Everything else stays exactly as in lo-res: the GFX picture, the sprites and the border keep their pixels (each pixel is two dots, in the same place), the line and frame rates do not change, and the 15 kHz analog PAL picture on a 1084S keeps its size and position.

- **The map is 128 cells wide.** A text row is two map rows: cells 0–63 in map row 2r, 64–127 in row 2r + 1 (plus the base row, `TILEMAP_PTR`, mod 128). So cell (row, col) is

`VIH_TILE_MAP0_BASE[row * VIH_HIRES_MAP_W + col]` (128 cells a row), rows 0–31 in tile map 0's window, 32–63 in tile map 1's.

- **Columns count dots from the screen's left edge**, the way pixels do (dot = 2 x pixel). With the usual 32-pixel border, column 8 is the picture's left edge and 80 columns are columns 8–87. Rows are as in lo-res (`TILE_SCROLL` y and the 8-line cell).
- `TILE_SCROLL` x counts **dots**: 0 puts the cells exactly on the 8-dot grid.
- Cells, fonts and colors are as in lo-res: fg/bg, the cell palette, flips, 512/1024 glyphs, the 8bpp font, `TILE_PRIORITY`. The glyphs are the same, half as wide. 8 x 8 only: `font_height` is ignored.
- Sprite-picture collisions (`SPRITE_COLL` [31:16]) are checked every dot.
- HDMI shows hi-res dots one to one: dots 24–743 on its 720, every column whole, 40 of the 64 border dots each side.

VIDEO_MODE — `$3A000298` (W) — `vih_video_mode_t`

Bit(s)	Name	Contents
0	hires	1 = hi-res text, 0 = lo-res

Taken where the next frame starts, so a frame is all lo-res or all hi-res. 0 at power-on, at every .rvk launch (`VIH_OWN_RELEASE`) and from `vih_init_registers()`; `vih_video_mode(1)` switches it on. Write-only; cores before v0.95 ignore the write.

Chapter 4 — Hardware Sprites

Features

- **16 channels**, all composited on every scanline — there is **no per-line limit** (all 16 can overlap on one line).
- 64 x 64 pixels per sprite; 1bpp (one color) or 8bpp (256-color).
- Signed positions, grid-absolute like the C64: (0,0) tucks the sprite behind the top-left border; the compositor clips. A sprite at x starts at screen x (core v0.95; before, one pixel to the left).
- Per-sprite: priority (front/behind the playfield), H/V flip, index-0 transparency (8bpp may disable it).
- Pixel-perfect collision, both sprite-sprite and sprite-background.

Sprite images live in chip RAM in **4096-byte slots**: slot n starts at byte address $n \cdot 4096$ (a 64x64x1 byte image fills a slot exactly).

Registers

SPRITE_POS — `$3A00013C` (W) — `vih_sprite_pos_t`

Bit(s)	Name	Contents
[13:0]	x	signed pixel X

Bit(s)	Name	Contents
[27:14]	y	signed pixel Y
[31:28]	channel	target channel 0..15

SPRITE_ATTR — `$3A000140` (W) — `vih_sprite_attr_t`

Bit(s)	Name	Contents
[7:0]	slot	image slot number (address / 4096)
[15:8]	color	palette index (1bpp sprites)
[21:16]	flags	see VIH_SPR_* below
[24:22]	palette	its own palette (core v0.95), added to `PAL_SEL`'s sprite palette, mod 8
[31:28]	channel	target channel 0..15

Flag bits (within the 6-bit flags field):

Value	Name	Meaning
`0x02`	VIH_SPR_8BPP	8bpp image (else 1bpp)
`0x04`	VIH_SPR_NO_TRANSP	index 0 is opaque (8bpp)
`0x08`	VIH_SPR_FLIP_H	mirror horizontally
`0x10`	VIH_SPR_FLIP_V	mirror vertically
`0x20`	VIH_SPR_ON_TOP	in front of GFX + tiles

SPRITE_ENABLE — `$3A000128` (W)

`[15:0]` = one enable bit per channel (the `$D015` of this machine).

SPRITE_COLL — `$3A000150` (R) — `vih_sprite_coll_t`

Bit(s)	Name	Contents
[15:0]	sprite	sprite-sprite collision mask (like `$D01E`)
[31:16]	bg	sprite-background mask (like `$D01F`)

Snapshot taken once per frame; background = any non-transparent framebuffer pixel or opaque tile pixel.

Timing note

`SPRITE_POS`/`SPRITE_ATTR` take effect **immediately** (not frame-latched). For guaranteed tear-free movement update during the reader-idle gap, beam lines ~211..222: `vblank(216)` (see RASTER_VPOS). The copper writes the same channels per-scanline (IDs `$0F`/`$10`) to multiplex one channel into many sprites down the screen.

Chapter 5 — The Copper

A 256-entry display-list processor. Each entry fires when the beam reaches its scanline, and **any number of entries may share a line** (core v121): the list drains at one entry per two clocks, ~3200 entry slots per scanline — the entire 256-entry list could legally fire on a single line. Border + background pairs, register batches, sprite multiplex bursts: all land on the same line together. Each entry:

Field	Bits	Contents
line	[63:48]	scanline at which to fire
reg	[47:32]	copper register ID (below)
value	[31:0]	value to write

Entries must be sorted by line. Writing is by auto-incrementing pairs:

- **COPPER_WR_ADDR — `$3A000178` (W)**: entry index 0..1023.
- **COPPER_WR_DATA — `$3A00017C` (W)**: two writes per entry — first the low 32 bits (value), then the high 32 (`{line,reg}`); commits and advances the index.
- **COPPER_CNT — `$3A00010C` (W)**: number of active entries; arms the list (0 = copper off).

1024 ENTRIES SINCE CORE v0.72 (it was 256). Since the copper applies *every* entry due on a scanline, a per-line effect costs one entry per line per thing it moves, and the budget is entries-per-line x lines rather than just lines. A four-layer starfield that sets a sprite's X on each of 192 visible lines is 768 entries on its own — which is what the old limit could not hold.

> Arm no more entries than you have written. Unwritten entries read as zero, and a zero entry means "line 0, register 0, value 0" — register 0 is BORDER_DIMS, so an over-long count blanks all four borders.

Copper register IDs

ID	Name	Value
`$00`	BORDER_DIMS	{bottom,top,right,left} bytes
`$01`	BORDER_COLOR	palette index
`$02`	X_SCROLL	framebuffer X (0..511)
`$03`	BG_COLOR	palette index
`$04`	FONT_MODE	[0] font_src, [1] height
`$06`	TILE_COLOR_-MODE	[0] 1bpp/8bpp glyphs
`$07`	TILE_BG_TRANS	[0] tile bg transparency

ID	Name	Value
$08	TILES_ENABLE	[0] tile layer on/off
$09	SPRITE_ENABLE	[15:0] channel mask
$0C	GFX_ENABLE	[0] GFX layer on/off
$0D	AUDIO_ENABLE	[0] master audio mute
$0E	TILEMAP_ROW_PTR	[6:0] tile display base row
$0F	SPRITE_MOVE	sprite position word (as SPRITE_POS)
$10	SPRITE_SLOT	{palette[28:26], flags[25:20], color[19:12], slot[11:4], ch[3:0]}
$11	GFX_PTR	[12:0] framebuffer block (mid-frame splits!)
$12	Y_SCROLL	[7:0] framebuffer Y
$13	TILE_SCROLL	{y[24:16], x[8:0]}
$14	PALETTE	{palette[26:24], index[23:16], rgb565[15:0]}
$15	TILE_PRIORITY	[0] tiles above/below GFX
$16	PAL_SEL	each layer's palette, as PAL_SEL (core v0.95)

Timing

The copper compares against the **reader's** line counter, which runs about 12 lines ahead of the visible beam: schedule entries ~12 lines higher than the visual target. The copper is the cycle-exact per-line tool; for CPU-side work at a line, use the raster interrupt (Chapter 9).

Classic recipes: raster bars ($01/$03 per line), side-border opening ($00), split playfields ($11 mid-frame), sprite multiplexing ($0F/$10), per-line palette gradients ($14).

Chapter 6 — The Blitter

A chunky-8bpp streaming engine on chip RAM: one pixel = one byte, so arbitrary alignment is free (no barrel shifter needed, per-byte write masks do the work) and color-key transparency replaces bitplane masking.

Operations

Operation	Description	Speed
FILL	constant byte rectangle	~40-50 MB/s
COPY	rectangle copy, optional color-key, any alignment/width	~25-35 MB/s
XOR	dest = src^dest or const^dest; self-inverse	as COPY

Operation	Description	Speed
LINE	hardware Bresenham, any slope, any color	~7 Mpix/s
TEXLOAD	SDRAM region -> 256x256 internal texture cache (64 KB)	~40 us full
TEXSPAN	affine textured horizontal span (u,v,du,dv in Q16.16)	~60 Mpix/s
GOURAUD	palette-index-interpolated span (smooth shading)	fill speed

In practice: ~250-300 animated 32x32 objects erased+redrawn per 50 Hz frame (Amiga OCS managed ~60-70; the MEGA65's DMAgic moves 40/20 MB/s but only within 384 KB — the blitter works across all 16 MB).

Every width down to a single pixel is exact (core v126 retired a legacy 1-word read path that fed the narrowest keyed blits garbage), and a fully-transparent word never touches memory.

XOR rule: XOR overlays and backup-restore objects must not overlap — the two restore mechanisms don't commute.

The CPU's read window

TEXLOAD any region into the cache, then read it back word by word: write a word index to `BLIT_TEXIDX`, read `BLIT_TEXDATA`. This is how the CPU reads *anything* in chip RAM (`vih_sdram_read()`, ~5 MB/s; `vih_sdram_peek8()` for single bytes). It clobbers the texture cache. Contract: discard the first TEXDATA read after writing TEXIDX.

Registers

Address	Name	Contents
`$3A0001A0`	BLIT_SRC	[23:0] source byte address
`$3A0001A4`	BLIT_DST	[23:0] destination byte address
`$3A0001A8`	BLIT_DIMS	{height[31:16], width[15:0]} pixels
`$3A0001AC`	BLIT_MOD	{dst_mod[31:16], src_mod[15:0]} bytes/row
`$3A0001B0`	BLIT_CTRL	control + GO (below)
`$3A0001B4`	BLIT_STATUS	(R) [0] = busy, [1] = pend (queue full), [2] = Z clear busy (v0.90)
`$3A0001B8`	BLIT_TEX_U	span start U (Q16.16); texrect: per-row start
`$3A0001BC`	BLIT_TEX_V	span start V (Q16.16)
`$3A0001C0`	BLIT_TEX_DU	per-pixel dU (Q16.16, signed)

Address	Name	Contents
$3A0001C4	BLIT_-TEX_DV	per-pixel dV (signed); texrect: per-ROW step
$3A0001C8	BLIT_TEXIDX	(W) read-window word index
$3A0001CC	BLIT_TEX-DATA	(R) read-window data [15:0]
$3A0001D0	BLIT_CBASE	(W) [7:0] color base for COPY/TEXSPAN (v0.30)
$3A0001D4	BLIT_LUT	(W) color remap table (v0.31; see below)
$3A0001D8	BLIT_QUEUE	(W) [0] 1 = 256-deep op queue (v0.85; see below)
$3A0002C0	BLIT_ZCTRL	(W) [0] 1 = Z buffer on (v0.90; see below)
$3A0002C4	BLIT_ZBASE	(W) [23:0] byte address of screen pixel (0,0), even
$3A0002C8	BLIT_ZSTART	(W) 1/z at the op's first pixel, Q16.16 unsigned
$3A0002CC	BLIT_ZSTEP	(W) 1/z step per pixel, Q16.16 signed
$3A0002D0	BLIT_Z-CLEAR	(W) any write clears the Z buffer
$3A0002D4	TRI_XL	(W) long edge x at the first row, Q16.16 (v0.93; see Triangles)
$3A0002D8	TRI_DXL	(W) its step per row, Q16.16 signed
$3A0002DC	TRI_XS	(W) first short edge x at the first row
$3A0002E0	TRI_DXS	(W) its step per row
$3A0002E4	TRI_XS2	(W) second short edge x at row N1
$3A0002E8	TRI_DXS2	(W) its step per row
$3A0002EC	TRI_ROWS	(W) [9:0] N1, [25:16] N2 rows
$3A0002F0	TRI_DZL	(W) 1/z step per row along the long edge
$3A0002F4	TRI_DUL	(W) u step per row along the long edge
$3A0002F8	TRI_DVL	(W) v step per row along the long edge
$3A0002FC	TRI_CLIP	(W) [9:0] first pixel drawn, [25:16] first pixel not drawn

BLIT_CTRL bits

Bit(s)	Name	Contents
0	GO	start (queued if busy — one deep, v0.32)
1	FILL	1 = fill, 0 = copy
2	KEY_EN	color-key enable (copy: source==key skipped)

Bit(s)	Name	Contents
3	XOR	XOR mode
4	LINE	line mode (DIMS =
5	LINE_XN	line X direction negative
6	TEXSPAN	textured span
7	TEXLOAD	texture cache load
[15:8]	KEY	key color / fill byte / line color
16	GOURAUD	interpolate palette index along the span
17	TEXRECT	textured RECT — hardware row loop (v0.33)
18	PERSP	perspective-correct TEXSPAN (v0.91; not with 17)
19	TRI	whole triangle (v0.93; with 1, 6, 6+18 or 16)

Widths: COPY/XOR/TEXSPAN <= 1024 px per op; FILL uncapped; LINE major axis <= 2047. Completion: BLIT_STATUS[0] polls busy, [1] the op-queue pend flag; blit-done also raises IRQ source bit 1 (Chapter 9).

Color base / remap (v0.30 / v0.31). BLIT_CBASE[7:0] is added to every non-keyed COPY / TEXSPAN pixel before storing (the key still tests the raw source), sliding a 0..15 source bob onto any 16-color block; 0 = legacy. BLIT_LUT is a 256-byte table applied *after* CBASE on the same ops — written byte = `lut[(src + cbase) & 0xFF]` — landing each source value on an arbitrary palette entry (global-palette schemes). Write `[31]=1` to set enable = `[0]`, `[31]=0` to load `lut[[15:8]] = [7:0]`. FILL/LINE/XOR/GOURAUD ignore both. The LUT is read live (not snapshotted by the op queue): drain before rewriting entries a queued blit still references.

Op queue (v0.32). A GO issued while busy is latched, not dropped, and auto-starts when the current blit ends (one deep). Poll BLIT_STATUS[1] (pend) = 0 for "queue has room" to overlap CPU and blitter; code that waits for busy = 0 is unaffected.

Deep op queue (v0.85), opt-in. `BLIT_QUEUE[0] = 1` (`vih_blit_queue(1)`) puts a 256-entry queue of complete blit setups in front of the one above, so a polygon renderer can hand over hundreds of spans and get on with the next polygon. The two status bits keep their meaning: **busy** = something is running *or queued* (`vih_blit_wait()` still waits for everything), **pend** = the queue is *full* (`vih_blit_wait_pend()` still means "wait for room"). Queued blits run strictly in order. It is **off** after every reset, after every `VIH_OWN_RELEASE` write (the kernel does that at each `.rvk` launch) and in `vih_init_registers()` — a program that never asks gets exactly the v0.32 queue. Rule in deep mode: `vih_blit_wait()`, not `vih_blit_wait_pend()`, before flipping the screen or touching pixels that queued blits write.

CPU writes during a blit (v0.77). A long burst of CPU writes into chip RAM (memset/memcpy) outranks the blitter on the bus. Before v0.77 a blit that got no bus for ~1.3 ms was aborted by its own watchdog and came out cut off after a line or two; since v0.77 it

simply **pauses** and finishes when the burst ends. Issue each GO only when there is room (`vih_blit_wait_pend()`): a running blit can now still be waiting when the next one is issued.

Textured rect (v0.33). CTRL bit 17 draws a whole scaled sprite in one op: the row loop runs in hardware (u restarts at TEX_U each row, v steps by TEX_DV per ROW, dst advances width + dst_mod per row). Color-key / CBASE / LUT apply as on a normal span. Replaces one-TEXSPAN-per-row sprite loops.

Z buffer (core v0.90)

Opt-in. The blitter owns its own **Z RAM** (block RAM inside the FPGA, not chip RAM): one cell per 2 pixels of the 384 x 224 screen — the pixel pair of one SDRAM word, which the blitter already writes with a per-pixel mask — so 192 x 224 = 43008 cells. Each cell holds:

Bits	Contents
[15:0]	Z = 1/z: **bigger = nearer**, 0 = empty
[16]	left (even) pixel drawn this frame
[17]	right (odd) pixel drawn this frame

With `BLIT_ZCTRL[0] = 1`, every **FILL, TEXSPAN, TEXRECT and GOURAUD** op tests and writes it; COPY, XOR, LINE and TEXLOAD never do. For each cell the op touches, the op's 1/z **at the cell's left pixel** is compared with the cell (so two polygons on one plane tie at their shared edge — `>=` wins):

- **wins**: this op's pixels in the cell are drawn, the cell's Z becomes the new value, their drawn flags are set;
- **loses**: only the pixels whose drawn flag is still 0 are drawn, those flags are set, and the cell keeps its Z.

So every pixel gets drawn once per frame and a renderer can drop its polygon sort. The test is per 2 pixels: where two polygons meet at different depths inside one cell, the edge is decided for both pixels of the cell together.

Register	Use
`BLIT_ZCTRL`	[0] Z on. Taken with each op (queued ops keep the value they were issued with). Off after every reset, every `VIH_OWN_RELEASE` write (each `.rvk` launch) and in `vih_init_registers()`
`BLIT_ZBASE`	byte address of the framebuffer's pixel (0,0), even, 512-byte rows. The op's destination minus ZBASE gives its cells. Read **live**: change it only while the blitter is idle
`BLIT_ZSTART`	1/z at the op's first pixel, Q16.16; the stored and compared Z is `[31:16]`. Taken with each op
`BLIT_ZSTEP`	1/z step per pixel, Q16.16 signed. Taken with each op

Register	Use
`BLIT_ZCLEAR`	any write sets every cell to 0 (Z empty, flags clear), one cell per clock = **~0.43 ms**. It runs **beside** the blitter: a framebuffer FILL can run at the same time. Z ops wait for it on their own (letting go of the bus while they wait). `BLIT_STATUS[2]` = 1 while it runs

Rules:

- 1/z values below 0 count as 0 (empty) and from 65536 up as 65535: scale your 1/z so the visible depth range uses the 16 bits.
- A row outside the 384 x 224 screen at ZBASE (and the part of a row past x = 383) is drawn as if Z were off.
- A multi-row op (TEXRECT, a FILL with height > 1) starts Z at ZSTART on every row.
- The Z RAM is **not** cleared at reset: clear it before the first frame.
- Speed with Z on: a FILL row costs a plain fill plus ~10 clocks (core v0.92: the Z test runs a few words ahead of the stream; on v0.90/v0.91 a Z FILL went through the word buffer at about half speed); a textured or Gouraud row costs ~7 clocks more.
- Write BLIT_ZCLEAR only after `vih_blit_wait()`; it waits for a Z row that is already being drawn, and Z rows issued after it wait for it.

Per frame:

```
vih_blit_wait();                        /* last frame's spans are done  */
vih_blit_z(0u);                         /* background without Z ...     */
vih_blit_zbase(fb);
vih_blit_zclear();                      /* ... while the Z RAM clears   */
vih_blit_fill(fb, 512u - 384u, 384u, 224u, 0u);
vih_blit_z(1u);
/* per span: */
vih_blit_wait_pend();
vih_blit_zspan(z0, dz);                 /* 1/z at the first pixel, step */
vih_blit_texspan(dst, len, u, v, du, dv);
```

On cores before v0.90 the Z registers are ignored and ops draw without Z.

Perspective spans (core v0.91)

Opt-in per span: `BLIT_CTRL` bit 18 together with bit 6 (TEXSPAN; not with the TEXRECT row loop). An affine span steps u and v evenly across the screen, which bends textures on polygons seen at an angle; a perspective span divides by the depth. The span takes the three values that *are* linear across the screen:

Register	Perspective meaning
`BLIT_TEX_U`	u·w at the first pixel, Q24.8 signed (texels × w × 256)
`BLIT_TEX_V`	v·w at the first pixel, Q24.8 signed

Register	Perspective meaning
BLIT_TEX_DU	step of u·w per pixel, Q24.8 signed
BLIT_TEX_DV	step of v·w per pixel, Q24.8 signed
BLIT_ZSTART	w = 1/z at the first pixel, Q16.16 — the same value the Z buffer takes
BLIT_ZSTEP	step of w per pixel, Q16.16 signed

The core cuts the span into 32-pixel pieces. At both ends of each piece it works out u = (u·w) / w and v = (v·w) / w — a 256-entry reciprocal table, one Newton step, two multiplies, within 0.001 texel of exact division — and steps u and v straight between the two ends (as Quake did in software). The next piece's ends are computed while the current piece draws, so a span costs ~16 clocks more to start and nothing per piece. With the Z buffer on, the same w values are the span's Z.

Keep |u·w| and |v·w| below 2^23 (for example u, v up to 256 texels with w up to 32767; w below 1/65536 counts as 1/65536). Texture wrap, color key, CBASE and the LUT work as on an affine span.

C API: `vih_blit_texspan_persp(dst, len, uw, vw, duw, dvw, w0, dw)` and `vih_blit_texspan_persp_ex(..., key_en, key)`. On cores before v0.91 bit 18 is ignored and the span draws affine with u·w taken as u.

Triangles (core v0.93)

Opt-in per op: BLIT_CTRL bit 19 together with a fill (bit 1), a textured span (bit 6), a perspective span (bits 6 + 18) or a Gouraud span (bit 16). One op draws a whole triangle: the blitter walks its rows itself and draws each one exactly as the span a program would have handed over — the same pixels, the same Z test, texture, color key, CBASE and LUT — so the CPU does no work and writes no registers per row.

The CPU gives the triangle split at its middle vertex:

Register	Triangle meaning
BLIT_DST	address of pixel (0, first row) — not of the first pixel
BLIT_MOD	dst field = the row stride in bytes (512 on the 384x224 screen)
TRI_XL / TRI_DXL	long edge (top to bottom vertex): x at the first row, step per row, Q16.16
TRI_XS / TRI_DXS	first short edge (top to middle vertex), used for N1 rows
TRI_XS2 / TRI_DXS2	second short edge (middle to bottom), N2 rows; XS2 = its x at row N1
TRI_ROWS	[9:0] N1, [25:16] N2

Register	Triangle meaning
`BLIT_ZSTART`, `BLIT_TEX_U`, `BLIT_TEX_V`	1/z, u, v on the long edge at the first row (u = the Gouraud color, or u·w; v = v·w for perspective)
`TRI_DZL` / `TRI_DUL` / `TRI_DVL`	their steps per row along the long edge
`BLIT_ZSTEP`, `BLIT_TEX_DU`, `BLIT_TEX_DV`	their steps per pixel, as for a span
`TRI_CLIP`	[9:0] first pixel a row may draw, [25:16] first pixel it may not

A row draws pixels xa = ceil(min(xl, xs)) up to ceil(max(xl, xs)) − 1, cut to TRI_CLIP; its first values are the long edge's plus step per pixel × (xa − xl), with xa − xl taken to 1/128 pixel. Then both edges and the long-edge values step; after N1 rows the second short edge takes over. `BLIT_DIMS` is not used. Rows are **not** clipped: give only rows that lie on the screen. XOR (bit 3) is not meant for triangles.

All of it — edges, rows, clip, values and steps — is taken with each op and travels through the pend slot and the 256-deep queue like the Z values, so the next triangle can be set up while one draws. The clip goes back to 0 .. 1023 at reset, at every `OWN_RELEASE` and in `vih_init_registers()`.

Cost: about 6 clocks a row to work the row out (the blitter lets go of the bus meanwhile), beside the span's own time — instead of the ~1.6 µs a row it costs the CPU to work out and hand over a span.

C API: `vih_blit_tri(&t, row0, stride, ctrl)` with a `vih_blit_tri_t` and `ctrl = VIH_TRI_FILL(color)`, `VIH_TRI_TEX`, `VIH_TRI_TEX_KEY`, `VIH_TRI_GOURAUD`, `VIH_TRI_PERSP` or `VIH_TRI_PERSP_KEY`; `vih_blit_tri_clip(x0, x1)`. `vih_tri.h` sets a triangle up from three vertices — position x, y (Q16.16, pixel centres), z = 1/z, and u, v in the op's format — with `vih_tri_setup()` / `vih_tri_draw()`, rows cut to the range given. On cores before v0.93 bit 19 is ignored and the op runs as the plain fill or span with whatever `BLIT_DIMS` holds — check the core version first.

Bus position: 3rd SDRAM master — video reader > PCM > blitter. The display always wins; watchdogs bound every grant (a stuck blit aborts in ~1.3 ms, never wedging the machine — since v0.77 only time the blitter actually holds the bus counts, so waiting for the bus is never a "hang").

C API: `vih_blit_fill/copy/copy_key/xor/xor_fill/line`, `vih_blit_texload/texspan/texspan_ex/texrect`, `vih_blit_gouraud`, `VIH_BLIT_CBASE`, `vih_blit_lut_set/lut_enable`, `vih_blit_busy()/vih_blit_wait()/vih_blit_wait_pend()`, `vih_blit_z/zbase/zclear/`

`zclear_wait/zspan` (v0.90), `vih_blit_texspan_persp/texspan_persp_ex` (v0.91), `vih_blit_tri/tri_clip` and `vih_tri.h` (v0.93), `vih_sdram_read/peek8`.

Software 3D

The proven recipe (`risc/apptest/apptest_vect.c`): CPU rotates/projects vertices (hardware multiply, Q16.16), walks triangle edges, and emits one blitter op per scanline — flat FILLs for solid polygons, TEXSPANs for texture, GOURAUD spans for lighting. Backface culling is one cross product. Since core v0.90 the Z buffer replaces the polygon sort: give each span its 1/z at the first pixel and the step per pixel (`vih_blit_zspan()`).

Chapter 7 — SID

A cycle-modelled MOS 6581 with multi-mode filters, clocked by a 1 MHz enable. Three voices — triangle/saw/pulse/noise, ADSR, ring modulation, oscillator sync — exactly the register model every SID player expects.

SID — `$3A000160` (W) — `vih_sid_t`

Bit(s)	Name	Contents
[7:0]	value	register value
[12:8]	reg	SID register 0..28

One write pokes one SID register, `POKE $D400+reg`-style. The register file is replayed to the SID continuously, so a value stays set until rewritten. (From BASIC: `POKE 46080+reg,value` — see BasicMachine.txt.)

Since core v0.79 every write reaches the SID, in order — the writes go through a queue that hands the SID at most one per 1 MHz tick, the rate a real C64 CPU can write at. Before v0.79 a gate OFF followed straight by gate ON (100 MHz CPU, two consecutive writes) was overwritten before the SID's envelope ever saw the OFF, so the note did not restart. Now it does, exactly as on a C64. A burst of up to 64 writes is played out in order; a value stays set until rewritten, as before. A full 25-register update takes 25 µs.

The classic register map applies (0/1 = voice 1 freq, 4 = control, 5/6 = ADSR, +7 per voice, 21-23 = filter, 24 = mode/volume).

Chapter 8 — PCM: The Sample Engine

Four independent sample voices, streamed from chip RAM, mixed at ~48 kHz with linear interpolation — Paula-class playback with modern pitch range.

- Per voice: SDRAM base, length, Q16.16 pitch, 8-bit volume, one-shot or loop with **arbitrary loop start** (sustain loops).
- Samples: 8-bit signed (packed) or 16-bit.

- Optional 75/25 stereo panning (Amiga voice routing); mono default.
- Voices stream through per-voice rings refilled by DMA during vblank — playback never fights the display.
- `risc/mod_player.h` is a complete 4-channel ProTracker (.mod) replayer on this engine.

Control block

The engine is programmed by an 18-word control block written through an auto-incrementing window:

- **PCM_IDX — `$3A000164` (W)**: word pointer 0..17.
- **PCM_DATA — `$3A000168` (W)**: write the word, pointer advances.

Word	Name	Contents
0	header	[7:0] master volume, [11:8] per-channel retrigger, [12] stereo enable
1	magic	must be `VIH_PCM_MAGIC` or the engine stays silent
2-5	ch 0	base, len, pitch, flags
6-9	ch 1	(same layout)
10-13	ch 2	
14-17	ch 3	

Per-channel words:

Word	Contents
base	sample start (SDRAM byte address / 2 = word index)
len	sample length in samples
pitch	phase increment (Q-format playback rate)
flags	[7:0] volume, [8] enable, [9] 16-bit, [10] loop, [31:11] loop start

C: `vih_sdram_load()` puts a sample in chip RAM; fill a `vih_pcm_ctrl_t` and `vih_pcm_commit()` writes all 18 words, the trigger last.

Beeper

AUDIO_FREQ (CPU only, no copper ID; C: `VIH_AUDIO_FREQ`): a square-wave divisor against a 24.576 MHz reference; 0 = silent. The C64-style "one POKE makes sound" path.

Mixer

SID + beeper + PCM sum to 16-bit output with saturation, delivered to both the 3.5 mm jack and HDMI audio. The YM2151 (core v0.95, stereo) joins the mix, and so does the VIC-20's VIC

while the VIC-20 chips are on (Chapter 9, 8-bit CPU block). **AUDIO_ENABLE — `$3A000138`**: `[0]` master mute (copper ID `$0D` — silence exactly at a scanline).

Chapter 9 — SYSTEM: CPU, Interrupts, I/O

The CPU

VexRiscv "Murax": RV32IM + Zicsr, 5-stage, DSP multiplier, 100 MHz, ~1 instruction/clock from 512 KB of zero-wait on-chip RAM. GCC toolchain (`riscv-none-elf`), plain C, no OS.

ICACHE — `$3A0001DC` (W) — instruction cache (core v0.88)

`[0]` = 1: an 8 KB instruction cache (direct mapped, 32-byte lines) caches code fetched from **above** the 512 KB of CPU RAM — Trapdoor RAM, chip RAM and (core v0.89) Zorro RAM. Code in CPU RAM is never cached and runs exactly as before, cache on or off. C: `vih_icache(1)`.

Code runs from	cache off	cache on
CPU RAM	100%	100%
Trapdoor RAM	66%	99%
chip RAM (a loop that fits the cache)	46%	100%
Zorro RAM (a loop that fits the cache)	46%	100%

(Board measurements, % of CPU-RAM speed; big code that keeps missing gains less.)

The one rule: the cache does not see data writes. After writing NEW code above CPU RAM — copying code into Trapdoor RAM, chip RAM or Zorro RAM — call `vih_cache_invalidate()` before running it; that write empties the instruction cache as well as the data cache. Switching the cache on or off empties it too. It is **off** after every reset, after every `VIH_OWN_RELEASE` write (every `.rvk` launch) and in `vih_init_registers()`.

TEMP — `$3A0001FC` (R) — FPGA temperature (core v0.82)

Bit(s)	Name	Contents
[11:0]	code	raw 12-bit reading of the FPGA's sensor
31	valid	a reading has arrived since reset

Degrees C = code × 503.975 / 4096 − 273.15. The core reads the sensor (the FPGA's XADC) about every 10 ms. C: `vih_temp_valid()`, `vih_temp_mc()` = thousandths of a degree, integer only (`42449` = 42.449 °C). Reads 0 on cores before v0.82.

CORE_VERSION — `$3A00023C` (R) — the core's version (core v0.98)

Bit(s)	Name	Contents
[7:0]	minor	the minor version, BCD (`$98` for v0.98)
[15:8]	major	the major version, BCD

Bit(s)	Name	Contents
16	test	a test build (the purple power LED)

Set by the build from the core's label, so software can tell which core it runs on: v0.98 reads `$00000098`, and `"%x.%02x"` of major and minor prints `0.98`. A test build without a version label reads `$00010000`. Reads 0 on cores before v0.98. C: `VIH_CORE_VERSION`, `vih_core_version_t`.

8-bit CPU block (core v0.95)

A 6502 and a Z80 inside the core, next to the RISC-V. They share 64 KBytes of their own RAM; one of them runs at a time, from 1 MHz (or slower) up to 25 MHz. The 6502 is VIXEN's own: an NMOS 6502 with every opcode, documented and undocumented, and the real chip's bus cycle for cycle (dummy reads, the double write of a read-modify-write, NMOS decimal mode). The Z80 is T80. The RISC-V loads code and data into the RAM, sets the speed, and lets the CPU run; nothing in the RISC-V slows down.

Address	Name		
`$000C0000-$000CFFFF`	CPU8 RAM	R/W	the 8-bit CPU's 64 KBytes: bytes, halfwords, words, on the RISC-V's own bus straight after Trapdoor RAM and as fast (a read one extra clock, a write none; code there goes through the instruction cache).
`$3B010000-$3B0103FF`	CPU8 STATE	R/W	the state window: one item a word (the page map and the context switch, below)
`$3A000300`	CPU8_CTRL	R/W	`[0]` run `[1]` 1 = Z80, 0 = 6502 `[2]` reset (held while 1) `[3]` IRQ / INT asserted `[4]` NMI asserted `[5]` VIC-20 chips on `[6]` the 6502 runs on the VIC's clock (1,108,405 Hz; CPU8_SPEED unused) `[7]` C64 chips on (core v0.96)
`$3A000304`	CPU8_SPEED	R/W	the CPU clock: Hz x 2^32 / 50,000,000 (`vih_cpu8_speed_hz()`); 25 MHz and above = full speed
`$3A000308`	CPU8_TRAP	R/W	`[15:0]` the 4 KBytes blocks whose accesses go to the RISC-V; `[16]` every Z80 IN / OUT goes to the RISC-V (else IN reads `$FF`, OUT is ignored)
`$3A00030C`	CPU8_ROM	R/W	`[15:0]` the 4 KBytes blocks the 8-bit CPU cannot write

Address	Name		
$3A000310	CPU8_STATUS	R	[0] stopped [1] an access waits for the RISC-V [2] it is a write [3] it is Z80 I/O [4] 6502 JAM [5] Z80 HALT
$3A000314	CPU8_TRAP-INFO	R	[15:0] the waiting access's address (Z80 I/O: the port in [7:0]), [23:16] the data it writes
$3A000318	CPU8_REPLY	W	[7:0] the data a waiting read gets; any write lets the waiting access finish
$3A00031C	CPU8_PC	R	[15:0] the CPU's PC (between instructions)
$3A000320	CPU8_KEYS0	R/W	VIC-20 keyboard, columns 0-3: byte c = the keys on column c (VIA2 PB bit c), bit r = row r (VIA2 PA bit r); 1 = pressed
$3A000324	CPU8_KEYS1	R/W	columns 4-7, the same way
$3A000328	CPU8_JOY	R/W	[0] up [1] down [2] left [3] right [4] fire [5] the RESTORE key [12:8] the C64's joystick 1, the same order (core v0.96); 1 = pressed
$3A00032C	CPU8_VICFB	R/W	the VIC's picture in chip RAM: [0] on [7:4] palette bank (a pixel's byte = bank × 16 + VIC color) [23:8] the chip RAM address / 256; taken at the start of each VIC frame
$3A000330	CPU8_VICPOS	R	[8:0] the VIC's raster line (0-311) [15:9] its raster cycle (0-70) [31:16] VIC frames since the VIC-20 chips went on
$3A000334	CPU8_RE-SUME	R/W	[8:0] line [15:9] cycle [31] armed: the stopped CPU starts by itself in that VIC cycle (CPU8_CTRL run then reads 1, armed 0)
$3A000338	CPU8_STOP-POS	R	[8:0] line [15:9] cycle: the VIC cycle in which the stopped CPU would have gone on
$3A00033C	CPU8_YM	W	the YM2151: [15:8] register, [7:0] value (queued, see below)
		R	its status: [0] timer A [1] timer B [7] busy (8 or more writes waiting)

Starting. Power-on, reset and every .rvk launch leave both CPUs in reset at 1 MHz with nothing trapped or protected, the VIC-20 chips off, no key pressed and no VIC picture written. Load the program, set CPU8_SPEED, CPU8_TRAP and CPU8_ROM, then write CPU8_CTRL with reset = 0 and run = 1. The 6502 starts at its reset vector ($FFFC), the Z80 at $0000. run =

0 stops the CPU between two instructions, never inside one; CPU8_STATUS bit 0 says when it has.

Talking to the RISC-V. An access to a trapped block (or a trapped Z80 IN / OUT) stops the CPU in the middle of that access until the RISC-V answers: poll CPU8_STATUS bit 1, read CPU8_TRAPINFO, write CPU8_REPLY (the byte for a read, any value for a write). The 6502's dummy reads and writes trap too, exactly as they would touch real I/O.

Timing. A 6502 cycle or a Z80 T state is one step of a 50 MHz clock divided down: 1 MHz is exact, 25 MHz is the maximum.

VIC-20 chips (core v0.95, CPU8_CTRL bit 5, the 6502 only). The VIC-20's VIC and two 6522 VIAs, all VIXEN's own. The VIAs, cycle for cycle with the 6502: both ports with their direction registers and input latching, CA1/CA2/CB1/CB2 with every interrupt, handshake and pulse mode, timer 1 one-shot or free running with its PB7 output, timer 2 one-shot or counting PB6 pulses, the shift register in all eight modes, IFR and IER. Timer 1 written with N reads N in the next cycle and `$FFFF` (with its interrupt) N + 2 cycles after the write, then reloads, in one-shot mode too, as the real chip does. The chips sit where the VIC-20 has them: the VIC at `$9000-$90FF` (16 registers, repeated), and anywhere in `$9000-$93FF` address bit 4 selects VIA1 (`$9110`) and bit 5 VIA2 (`$9120`); chips selected at once read ANDed. Those accesses never reach the RAM or a trap; the rest of `$9000-$9FFF` stays RAM (or trapped) – color RAM is `$9400-$97FF`, of which the VIC uses the low four bits.

Line	VIA1 (`$9110`, drives NMI)	VIA2 (`$9120`, drives IRQ)
PA	`[0]` serial CLK in `[1]` serial DATA in `[2]` joystick up `[3]` down `[4]` left `[5]` fire `[6]` tape sense (high) `[7]` serial ATN out	the keyboard rows
PB	user port (nothing connected)	the keyboard columns; `[7]` also joystick right
CA1	the RESTORE key (low while pressed)	tape read (high)
CA2	tape motor	serial CLK out (the bus line is its inverse)
CB1, CB2	user port	serial SRQ in (high), serial DATA out

The keyboard is wired both ways round, as on the VIC-20: a held key joins its column line to its row line, and a line reads low when anything on it drives it low, so a program may drive the columns and read the rows or the other way. Nothing else is on the serial bus: the KERNAL finds no device. The RISC-V writes the held keys to CPU8_KEYS0/1 (the VIC-20 matrix: column 0 holds `1 3 5 7 9 + £ INST/DEL` on rows 0-7, row 0 holds `1 ← CTRL RUN/STOP SPACE C= Q 2` on columns 0-7) and the joystick and RESTORE to CPU8_JOY. The VIAs count only while the 6502 runs: a trap or a stop freezes them with it.

The VIC (core v0.95). The VIC-20's video and sound chip, VIXEN's own, as the PAL 6561: a 4,433,618 Hz pixel clock, 71 cycles of 4 pixels a line, 312 lines, 50.04 frames a second. It runs whenever the VIC-20 chips are on, also while the 6502 stops or waits on a trap. Everything a VIC-20 program does with it works as on the chip: the text window (`$9000` X origin in 4 pixels, `$9001` Y origin in 2 lines, `$9002` columns and screen address bit 9, `$9003` rows and 8×8 / 8×16 characters, `$9005` screen and character addresses), hires and multicolor characters, reverse mode, border, background and auxiliary colors, `$9003` bit 7 / `$9004` the raster line, and registers rewritten in the middle of a line or a frame (a color written in cycle n shows from pixel 4 × (n – 7) + 1 on, reverse two pixels later). `$9006`/`$9007` (light pen) read 0, `$9008`/ `$9009` (paddles) item `$C6`'s values, or `$FF` with none. It sees the RAM as the VIC-20's VIC does: its 14 address lines are `$8000-$9FFF` and `$0000-$1FFF`; the RISC-V puts the character set at `$8000-$8FFF` (and protects it with CPU8_ROM, as a ROM). Its reads come between the 6502's: at full speed a 6502 cycle now and then waits one 50 MHz clock; on the VIC's clock never.

CPU8_CTRL bit 6 runs the 6502 on the VIC's clock, as in a VIC-20: one CPU cycle per VIC cycle, 1,108,405 a second, 22,152 a frame, so raster loops and timer-timed effects land on the same pixel every frame. Clear, the 6502 runs at CPU8_SPEED while the VIC keeps its own time.

The picture. With CPU8_VICFB on, the VIC writes everything it shows into chip RAM, one byte a pixel: bank × 16 + its color (0-15), so the VIC-20's 16 colors are 16 palette entries of the RISC-V's choosing. Line y (0-311) is at address + 256 × y, pixels 0-255 the visible part of the line (the VIC-20's default screen starts at pixel 48 of line 76, 176 × 184 pixels); 78 KBytes a frame. Each 64 pixels go to chip RAM as one burst a few microseconds after the VIC draws them. The address, bank and on take effect at the start of the VIC's next frame: switching between two addresses every frame double-buffers. The blitter copies the picture anywhere on the screen, or loads it into its texture cache (`smod` = 256 – width) to texture a 3D object with it.

The sound. Bass, alto and soprano (`$900A-$900C`, bit 7 on, bits 6-0 the pitch) and noise (`$900D`), with the master volume in `$900E` bits 0-3: square waves of 16 steps, a step every (127 – pitch) × 16 cycles for bass, × 8 alto, × 4 soprano (pitch 127 gives 128, not 0), noise from a 16-bit shift register stepped every (127 – pitch) × 2 cycles, all through the VIC-20's output filter (low-pass 1.6 kHz, high-pass 160 Hz). It joins the machine's audio while the VIC-20 chips are on.

The page map (core v0.95). Each 1 KByte page of the 64 KBytes is RAM, ROM, absent or trapped: write `VIH_CPU8_STATE[0x80 + page]` = 0 RAM, 1 ROM (the 8-bit CPU cannot write it), 2 absent (writes are lost; reads give the last byte the VIC read when the VIC-20 chips are on – what a VIC-20's CPU reads where nothing is fitted – and `$FF` when they are off), 3 trapped (as a CPU8_TRAP block). It adds to CPU8_ROM and CPU8_TRAP, and every page is RAM at power-on, reset and every .rvk launch. The RISC-V reads and writes every page through the CPU8 RAM whatever the map says, so a cartridge is loaded into RAM and then made ROM. Every VIC-20 memory layout is a page map:

VIC-20	Address	Pages
RAM1, RAM2, RAM3 (the 3K expansion)	`$0400-$0FFF`	1, 2, 3
BLK1	`$2000-$3FFF`	8-15
BLK2	`$4000-$5FFF`	16-23
BLK3	`$6000-$7FFF`	24-31
I/O2, I/O3	`$9800-$9BFF`, `$9C00-$9FFF`	38, 39
BLK5 (cartridges, A0CBM at `$A004`)	`$A000-$BFFF`	40-47

An unexpanded VIC-20 is pages 1-3, 8-31 and 40-47 absent; the KERNAL's RAM test then finds 3583 bytes free, as on the real machine.

The C64 chips (core v0.96, CPU8_CTRL bit 7, the 6502 only). A Commodore 64 on the core's own chips. The CPU8 RAM is its 64 KBytes of RAM; beside them are 20 KBytes of ROM and 1 KByte of color RAM, loaded through items `$C1` / `$C2` (`vih_c64_mem_write()`). The core holds no Commodore ROM: c64hw loads the user's own from `/VIXEN-32/C64/`. The VIC-II (a PAL 6569) is at `$D000`, the SID at `$D400`, the color RAM at `$D800`, CIA 1 at `$DC00`, CIA 2 at `$DD00`, the 6510's port at `$00` / `$01`, with the PLA's memory map, all as in a C64. The SID is the machine's own: the C64 owns and clocks it while its chips are on. Bit 7 wins over bit 5. CPU8_SPEED, CPU8_ROM and the page map's ROM and absent pages do nothing then; traps still do. The 6502 runs on the VIC-II's clock, 985,249 Hz, and the VIC-II stops it on bad lines and for sprites as on a C64. To start it: set bit 7 with `[2]` reset held, load the ROMs, then clear reset and set run; it starts at its KERNAL's reset vector.

CPU8_KEYS0 / 1 hold the C64's matrix: byte c = the keys on CIA 1 port A bit c, bit r = port B bit r. That is the MEGA65's key number: keynum = c × 8 + r for keys 0-63. In CPU8_JOY `[4:0]` is joystick 2 (port A), `[5]` RESTORE (the NMI) and `[12:8]` joystick 1 (port B).

With CPU8_VICFB on, the VIC-II writes its picture into chip RAM: 448 pixels a line (sprite X `$1D0-$1F7`, then `$000-$197`: everything a PAL TV shows), all 312 lines, line y at address + 512 y (156 KBytes), one byte a pixel = palette bank × 16 + C64 color. The 40 × 25 text window starts at pixel 64 of line 51. CPU8_VICPOS is the VIC-II's (cycle 1-63).

The trap map (core v0.96). A bit for each 16 bytes of the 64 KBytes, 128 words of 32: word w bit b is the 16 bytes at 512 w + 16 b. Write `VIH_CPU8_STATE[0xC4]` = w (the map off), then `VIH_CPU8_STATE[0xC5]` once for each word from w on (each write moves on to the next word), then `VIH_CPU8_STATE[0xC4]` = `0x100` (on). From then on an access to a block whose bit is 1 goes to the RISC-V just as a CPU8_TRAP block's does – the 6502's and the Z80's memory accesses, never the Z80's IN / OUT. It adds to CPU8_TRAP and the page map's trapped pages: an access any of them catches is one trap. With the C64 chips on, a block traps only while the PLA puts the cartridge there: its ROML or ROMH, or I/O – a cartridge's I/O 1 (`$DE00`) and I/O 2 (`$DF00`) lines are only pulled then. So `$01` hiding the cartridge hides its traps too, and a write the PLA sends to the RAM (8K and 16K modes) is never trapped. Off at power-on, reset and every .rvk launch, but the words are kept: write every word before switching it on. Item `$C4`

reads back `[6:0]` the next word and `[8]` on; `$C5` reads 0. c64hw plays a cartridge's registers and RAM with it, and answers the odd read of a bank that is not in the cartridge ROM. While the 6502 waits for the answer the VIC-II goes on, and (core v0.97) so do both CIAs, as a C64's never stand: a game that keeps an NMI on the raster with a free-running timer stays on it. Stopped (run = 0) the CIAs stand with the 6502.

The C64's cartridge ROM (core v0.96). Beside its own ROMs the C64 has 16 KBytes for a cartridge: ROML and ROMH, loaded like the ROMs through items `$C1` / `$C2` (`$C000` ROML, `$E000` ROMH). Item `$C3` is the cartridge's GAME (`[0]`) and EXROM (`[1]`) lines, 1 = high; the PLA reads them as a C64's does, so ROML shows at `$8000` and ROMH at `$A000` (or at `$E000`, and at the VIC-II's `$3000` of each bank, in Ultimax mode) exactly where a real cartridge would, `$01` hiding and showing them. A write there goes to the RAM, as on a C64 (Ultimax mode: nowhere). Both lines high at power-on, reset and every .rvk launch: no cartridge. After a bank switch c64hw answers the new bank's reads through the trap map, and copies the bank in once it has been read 256 times.

Zaxxon's bank switch (core v0.96). A third 8 KBytes, ROMH's second bank, loads through `$C1` / `$C2` at `$6000`. With item `$C3`'s `[2]` set, each 6502 read of ROML picks ROMH's bank, as the Zaxxon cartridge does: a read in `$8000-$8FFF` the first (`$E000`), in `$9000-$9FFF` the second (`$6000`). The switch happens in the core on the read itself – the cartridge's program runs from ROML, so the RISC-V could never keep up with it. `[2]` clear (power-on, reset, every .rvk launch): always the first.

Paddles and the 1351 mouse (core v0.96). Item `$C6` holds the C64's POTX (`[7:0]`) and POTY (`[15:8]`) and the control port they are on (`[16]` port 1, `[17]` port 2). The SID's `$D419` / `$D41A` read them while CIA 1 port A selects that port – PA6 port 1, PA7 port 2, as the C64's 4066 switch does – and `$FF` otherwise, as with nothing plugged in. The RISC-V writes them: c64hw (b9) turns the Amiga mouse into two paddles or a 1351 mouse (F3 in its menu), the fire buttons on the joystick lines (a paddle's on left / right, a 1351's on fire / up). The port's pins are never sampled – the paddle drain stays off, so the mouse's X axis and buttons are untouched. `$FF`, `$FF`, no port at power-on, reset and every .rvk launch. The VIC-20 uses the same item: with its chips on `$9008` / `$9009` read POTX / POTY while `[16]` is set, else `$FF`, and vic20hw (b18) makes the mouse two paddles (F5 in its menu), their fire buttons on the joystick's left and right lines.

The C64's tape (core v0.98). A Datasette's READ line, played from pulses the RISC-V queues, so a turbo loader times them as it would on a C64. Item `$C8` (write only) queues one pulse: `[23:0]` the C64 cycles from the last pulse to this one – a .TAP file's byte × 8, or its long pulse as it stands. 512 wait in a queue of block RAM (half a RAMB36); a pulse written to a full queue is lost. Item `$C7` written: `[0]` PLAY held down, `[1]` = 1 empties the queue; read: `[9:0]` the pulses queued (the tape holds up to two more: the one counting and the next), `[16]` PLAY, `[17]` the motor on. While PLAY is down and the 6510's port drives the motor (bit 5 low, as the KERNAL and every loader do) each pulse counts down one cycle each C64 cycle the 6502 is not stopped in – the CIAs' cycles, so the tape stands while c64hw's menu is open – and at its end pulls CIA 1's FLAG low for 4 cycles: ICR bit 4, the cassette's interrupt. PLAY is also the cassette sense:

the 6510's port bit 4 reads 0 while it is down. Queue empty and PLAY up at power-on, reset, every .rvk launch and while the C64's chips are off. C: `vih_c64tape_t`, `VIH_C64TAPE_PLAY`, `VIH_C64TAPE_FLUSH`; c64hw (b20) plays .TAP files with it.

The context switch (core v0.95). The state window holds every bit of the machine the 64 KBytes do not, one item a word:

Items	What	Written
`$00-$0F`	the VIC's 16 registers as written (`$9003` / `$9004` without the raster)	any time
`$10-$1F`	VIA1: ORB ORA DDRB DDRA, T1 counter, T1 latch, T2 counter, T2 latch, SR ACR PCR IFR IER, the PA and PB input latches, and the rest of the chip in item 15	CPU stopped
`$20-$2F`	VIA2, the same	CPU stopped
`$30-$36`	6502: A X Y S P PC, and `[0]` an interrupt is taken next `[1]` NMI pending `[2]` NMI line as last seen `[3]` the reset sequence comes next `[4]` JAM (read only)	CPU stopped
`$40-$4E`	Z80: AF BC DE HL IX IY SP PC AF' BC' DE' HL', I << 8 | R, WZ, and `[1:0]` IM `[2]` IFF1 `[3]` IFF2 `[4]` HALT `[5]` NMI pending `[6]` NMI line as last seen	CPU stopped
`$80-$BF`	the page map	any time
`$C0`	the YM2151 for the Z80: `[0]` on `[1]` 0 = two I/O ports, 1 = two memory addresses `[2]` its timer flags pull the Z80's INT `[31:16]` base	any time
`$C1`	the C64 memory pointer: `$0000` BASIC, `$2000` KERNAL, `$4000` character ROM, `$8000` color RAM, `$C000` ROML, `$E000` ROMH, `$6000` ROMH's second bank	any time
`$C2`	the byte there; a write stores it and moves the pointer on by one	any time
`$C3`	the C64's cartridge lines: `[0]` GAME, `[1]` EXROM (1 = high); `[2]` Zaxxon's bank switch	any time
`$C4`	the trap map: `[6:0]` the word `$C5` writes next, `[8]` on	any time
`$C5`	the trap map's word there (write only); moves on by one	any time

Items	What	Written
`$C6`	the paddles (C64: and the 1351): `[7:0]` POTX, `[15:8]` POTY, `[16]` on port 1 (the VIC-20's one port), `[17]` on the C64's port 2	any time

Reads never have side effects (a VIA's timer flag stays set). "CPU stopped" means out of reset, run = 0 and CPU8_STATUS bit 0 set; other writes to those items are dropped. The 6502 stops at an opcode fetch, the Z80 at T2 of an opcode fetch that starts a whole instruction (not after a prefix, not in an interrupt acknowledge), where all its registers hold their final values. Stopped right after an EI, IFF1 and IFF2 read as set: the Z80 sets them one T state later, and that EI is already past.

To freeze a machine: run = 0, wait for CPU8_STATUS bit 0, then save the 64 KBytes, the items and CPU8_STOPPOS. To bring it back: load all of it and write CPU8_RESUME = armed + the saved position. The CPU starts by itself in that VIC cycle, so a 6502 on the VIC's clock finds its timers and the raster line exactly as far apart as when it stopped – raster splits timed by a VIA stay on their line. The VIC itself is never stopped or saved beyond its registers: it shows whatever machine is in the 64 KBytes.

The YM2151 (core v0.95). Yamaha's OPM, VIXEN's own: eight FM voices of four operators, the eight connection algorithms with feedback, key code and key fraction pitch with DT1, DT2 and MUL, envelopes with key scaling, the LFO (saw, square, triangle, noise) with AM and PM, noise on channel 7's last operator, timers A and B with their flags and CSM, left and right per voice, and the YM3012 DAC's 10.3 floating-point steps – as the chip at 3,579,545 Hz, a sample every 64 clocks (55,930 a second), into the machine's audio in stereo. Its registers are the chip's own, `$01`- `$FF`: write `(register << 8) | value` to CPU8_YM. The writes wait in a 16-entry queue and take effect between samples; status bit 7 (busy) is set while 8 or more wait, so a program that waits for busy = 0 before each write (`vih_ym_write()`) never loses one. The timer flags are set only while their IRQ enable bits in `$14` are on, as on the chip; there is no interrupt line. Each sample is the one ymfm (Aaron Giles' YM2151 model, the reference it was written against) computes for the same writes. Power-on, reset and every .rvk launch: silent, every register 0 except `$20`-`$27` = `$C0` (both outputs on).

The YM2151 for the Z80 (core v0.95). As on arcade sound boards, the Z80 can drive the YM2151 itself: state window item `$C0` = 1 (on) + 2 (memory instead of I/O) + 4 (the timer flags pull the Z80's INT) + base << 16. The Z80 writes the register number to base and the value to base + 1, and reads the status at base + 1 (base + 0 reads `$FF`): two I/O ports, A7-A1 = base (Sega's sound boards: `$00`/`$01`), or two memory addresses, A15-A1 = base (Capcom's: `$F000`/`$F001`; the RAM there is not touched). Its writes go through the same queue as the RISC-V's, so a Z80 driver that waits for busy before each value, as drivers for the real chip do, never loses one. With INT on, the timer flags are the Z80's IRQ (IM 1 or IM 2), which is how sound drivers keep their tempo. Off at power-on, reset and every .rvk launch.

Memory protection

BL_LOCK (RAM word `0x0C`): 16 bits, one per 8 KB block of the **top 128 KB** (`0x60000`–`0x7FFFF`); set = read-only (writes silently dropped, reads always pass). Bit 0 = the top block (the KERNAL, `0x7E000`), locked at reset — C64 `$01`-style: poke it clear for intentional patching, and put it back. The lower 384 KB is always writable.

Changed in core v0.55: nothing is locked at reset. Protection is now opt-in — a resident kernel that wants to shield itself writes its own mask here. The old default existed because the bootloader lived at `0x7E000` and was baked into the bitstream; with the kernel loaded from the SD card, that block belongs to whatever the image puts there.

A blocked write does not fault — it is silently discarded, and reads always pass. A locked block therefore looks like RAM that accepts writes and forgets them. Anything failing mysteriously in the top 128 KB is worth checking against this register first.

Interrupts (the "KERNAL" model)

Hardware trap ⟶ bootloader dispatcher (saves state, reads mcause) ⟶ calls **through the RAM vector at `0x24`** — the machine's `$0314`. Install a handler by writing one function pointer; no rebuild.

Handler: `bool irq(uint32_t status)` — return true = acknowledged.

Source bit	Cause
0	raster compare (RASTER_TARGET reached)
1	blitter done
2	keyboard event queued
3	Ethernet frame received (chapter 11)
4	vertical blank started

Source 4 fires once per frame at the start of vertical blank, which is the moment to flip buffers or move sprites: the display is not reading, so a change made then cannot tear. It is not the same as a raster compare on the last visible line, which fires while the display is still fetching.

The `status` word the handler gets is the kernel's own order (`irq_status_t`, `shared/boot_protocol.h`), one place up from the source bits because bit 0 is the 50 Hz timer:

`status` bit	field	Cause
0	`st.f.timer`	50 Hz machine timer
1	`st.f.raster`	source 0, raster compare
2	`st.f.blitter`	source 1, blitter done
3	`st.f.key`	source 2, keyboard event queued
4	`st.f.eth`	source 3, Ethernet frame received

status bit	field	Cause
5	st.f.vblank	source 4, vertical blank started

Bits 3-5 are filled in from **kernel build 125**. Before that the kernel passed only timer, raster and blitter, and a key, Ethernet or vblank interrupt reached the handler with every bit clear — an app had to read the pending register at $3A000188 itself (that still works).

RASTER_TARGET — $3A000184 (W)

Target scanline for IRQ source 0. CPU-timescale (a few lines of latency); use the copper for cycle-exact effects.

IRQ_STATUS — $3A000188 (R/W1C)

Read = pending sources; write 1s to acknowledge.

IRQ_MASK — $3A00018C (W)

Enable mask for the sources above. Helpers: vih_raster_at(), vih_irq_enable(), vih_irq_ack(). A 50 Hz machine timer (Murax APB) is set up by the bootloader as the heartbeat.

Keyboard

KEYBOARD — $3A000190 (R = head, W = pop) — vih_key_t

Bit(s)	Name	Contents
[7:0]	code	key code (see below)
8	extended	PS/2 E0 prefix (legacy PS/2 path)
9	pressed	1 = make, 0 = break
10	valid	1 = event present

8-deep FIFO; a push raises IRQ source 2. On the MEGA65, code bit 7 is set and [6:0] is the **native key number** (0..79, C65 matrix order) — no PS/2 emulation. Up to core v0.93 the Help key is taken by the platform menu; from v0.94 (no platform menu) it is an ordinary key.

Joystick

JOYSTICK — $3A000180 (R)

Active-high, one bit per direction: [0] down, [1] up, [2] fire, [3] right, [5] left. (The scrambled order is historical — use the header's bit names, not guesses.) Port 2 is **JOYSTICK2** at $3A0001E8, identical layout. An Amiga mouse on port 1 reads through **MOUSE** at $3A0001EC.

Chip RAM access

SDRAM_ADDR — $3A000144 (W) / SDRAM_DATA — $3A000148 (W)

Write port into chip RAM: SDRAM_ADDR sets a 24-bit destination byte address (resets the auto-increment), each SDRAM_DATA pushes one 16-bit word (+2). A 256-deep FIFO drains to SDRAM each vblank.

SDRAM_STATUS — $3A00014C (R)

Bit(s)	Name	Contents
0	full	FIFO full (poll before push)
1	empty	FIFO drained
2	overrun	a push hit the 16 MB limit
[24:16]	level	FIFO fill level

For bulk moves use the blitter; for reads use the blitter window (Chapter 6). `vih_sdram_load()/vih_sdram_flush()` wrap this port.

File loading

The platform menu (Help key) streams a picked file into chip RAM staging at `0x800000` and publishes completion:

OSD_STATUS — $3A000194 (R)

Bit(s)	Name	Contents
[6:0]	fidx	file type: 3 = .RVK (app), 4 = .BAS
7	busy	stream in progress
[15:8]	seq	increments per completed load

OSD_SIZE — $3A000198 (R)

File size in bytes. Poll for a `seq` change with `busy` clear, then consume the staging area (BASIC does exactly this).

From core v0.94 there is no platform menu. Both registers are still there and always read 0: `seq` never moves, so a program polling them simply never sees a load. Nothing else about them changed.

OWN_RELEASE — $3A00019C (W)

Returns registers to their boot defaults without a full soft reset (`vih_release_ownership()`), used when handing the display between programs. Since core v0.85 / v0.88 it also switches the deep blitter queue and the instruction cache off, so a newly launched program always starts with both in their classic state. Since v0.90 it also switches the Z buffer off, and since v0.93 it sets the triangle clip back to 0 .. 1023.

SDRAM_PHASE — $3A0001E0 (W) — platform tuning

SDRAM clock fine-phase target, 0..559 steps of ~17.86 ps (one full 100 MHz period). Position/busy read back at Murax GPIO `0xF0000000` bits `[26:16]` = {busy, current}. The shipped default (0 = 180°) sits centred in the measured data eyes; `POKE 45298,1` in BASIC runs the built-in eye-sweep instrument. Normal software never touches this.

Chapter 10 — Booting from the SD Card

From core v0.55 the machine's kernel is **not in the bitstream**. The core holds a small bootstrap whose only job is to load `/VIXEN-32/KERNEL.BIN` off the SD card and hand over to it. Changing the kernel is a file copy — no core rebuild, no reflash.

The bootstrap reads only the kernel image at the front of `KERNEL.BIN` – up to the end of the last block its header lists, plus 64 bytes – so the programs after it in the same file (VMOS, the file browser, the VIC-20, the commander and the C64) cost no boot time; they are read when they are started.

Cores up to v0.93 load `/VIXEN-32/BOOT.BIN` instead – the same kernel image on its own, read whole – so a card can carry both files. Everything below applies to either name (the error screens name the file the core looks for).

The chain

1. **Bootstrap** (in the core, BRAM at `0x0`) — mount the card, find `KERNEL.BIN`, read its kernel image verbatim into SDRAM, copy block 0 to its load address, jump to it.
2. **Trampoline** (block 0, running in SDRAM) — interrupts off, zero all 512 KB of BRAM, unpack the remaining blocks into it, jump to the image's entry point.

The trampoline runs from the cached chip-RAM window because nothing executing in BRAM can overwrite itself. BRAM is **zeroed**, not merely overwritten: a leftover bootstrap routine that still happened to work would be worse than none, because something would come to depend on code that is meant to be gone.

The kernel image is an RVPK container whose **first block is stored uncompressed** — it contains the decompressor, so nothing could have unpacked it. Everything else may be deflated.

The reset button (core v0.80/v0.81)

The bootstrap lives in the same 512 KB that a game may take over completely, so before v0.80 the reset button restarted the CPU into the game's leftovers and only a power cycle brought the machine back. Since v0.80 the core keeps a **read-only copy of the bootstrap** (8 KB of its own block RAM) and writes it back over `0x0000–0x1FFF` after **every** reset, holding the CPU back for the ~20 µs it takes. The reset button therefore always runs the full chain above: the kernel and BASIC come back, whatever ran before. Since v0.81 the bootstrap also clears both tile maps first, so a reset looks exactly like power-on.

The copy holds only the bootstrap, not the kernel, so the kernel may grow freely; the bootstrap itself must stay within 8 KB (its build refuses anything larger).

Which SD slot (core v0.84)

The MEGA65 has two slots: the micro SD at the **back** and a full-size SD slot **inside** (bottom tray). VIXEN boots from — and keeps using — the back slot whenever it has a card; the internal slot is used only when the back slot is empty. That is the same rule the framework uses for the

F12 menu, so both always see the same card (up to v0.93; from v0.94 there is no F12 menu and the rule is unchanged). The slot is chosen **once per reset**, at the first SD access, and kept until the next reset: pull the back card while running and there is simply no card until you press reset, after which the internal card is used. (Switching on the fly would let a kernel that had mounted one card carry on writing to the other.) Both slots run at the same speed; nothing in software selects or needs to know the slot.

The kernel image must carry its own reset page

After the memset, address `0x0` is zeros — an illegal instruction. An image with no block covering `0x0` is a perfectly valid RVPK file that would leave the machine unable to start, so the trampoline **refuses to wipe anything** unless one is present.

There is no fallback

A missing or unusable image gets an error screen and the machine **stops**. It does not retry and does not boot anything else: a machine that quietly booted something other than what is on the card would be worse, because you would believe your `KERNEL.BIN` had loaded when it had not. Fix the card and power cycle.

Error screens

Red border, black background, and a code. All of these occur **before** anything is destroyed, so nothing on the card or in the machine has changed.

Code	Meaning	Cause
E01	NO SD CARD	The card did not respond at all.
E02	NO FILESYSTEM	Card responded; no FAT16/FAT32 on it.
E03	KERNEL.BIN NOT FOUND	No `/VIXEN-32/KERNEL.BIN` on the card (cores up to v0.93: BOOT.BIN).
E04	KERNEL.BIN SIZE WRONG	Far too small or too large to be an image (cores up to v0.93: BOOT.BIN).
E05	CARD READ FAILED	Card stopped responding mid-read.
E06	NOT A BOOT IMAGE	Wrong magic — not an RVPK container.
E07	BOOT IMAGE TOO NEW	Built for a newer core than this one.
E08	BOOT IMAGE DAMAGED	Corrupt; block count implausible.
E09	BOOT IMAGE UNUSABLE	No startup block, so it could not run.

Border colors

Once the trampoline takes over, BRAM — and with it the bootstrap, the screen routines and the font — is gone. The **border** is the only diagnostic left, so every stage sets it. Whatever color the machine stops on names the last step that completed.

Bootstrap, before the handover:

Color	Reached
white	bootstrap running
green	startup settle done
cyan	SD card claimed
purple	card reset succeeded
light green	filesystem mounted
yellow	`KERNEL.BIN` found
light red	image read into SDRAM
light grey	handing over to the trampoline

Trampoline, after it:

Color	Meaning
red	trampoline running
orange	all 512 KB of BRAM zeroed
cyan	image unpacked — jumping to it
pink	REFUSED: bad header, or no reset page

On a healthy boot these flash past in well under a second. Sitting on one means the machine stopped there.

Reading the card from your own code

The bootstrap gives the card back before it hands over, so the kernel and anything it loads may drive it themselves. Claim it with `SD_CTRL` bit 0, work, then release it — hold it and the F12 browser goes blind (up to v0.93; from v0.94 nothing else uses the card, but release it all the same so the same program runs on both).

`SD_CMD` (`$3A0001F4`) is one write per access:

Bits	Meaning
`[15:0]`	data (or, with BURST2, the starting cursor)
`[18:16]`	register: 0/1 block address lo/hi, 2 cursor (no auto-increment), 3 byte at cursor, 4 error, 5 command/CSR
`[19]`	write (0 = read)
`[20]`	BURST2 — **core v0.74 and later**

Command register values are 0 reset, 1 read, 2 write; the CSR answers with `[15]` busy, `[14]` error and `[13:12]` card type. Read results arrive in `SD_STATUS` (`$3A0001F8`), and you **wait on the completion counter in `SEQ[30:24]`, never on busy** — polling busy has a window that returns whole sectors of one repeated byte.

BURST2 exists because every byte of a sector otherwise costs its own clock-domain crossing between the CPU and the 50 MHz card logic. With bit 20 set, one crossing fetches two consecutive buffer bytes, returned together in `SD_STATUS[15:0]` as `{N+1, N}`: 256 crossings per 512-byte sector instead of about 1024, and streaming reads run three to four times faster. Nothing else changed — the status layout is identical and every single-byte access behaves as it did on v0.72.

On an older core bit 20 is simply ignored, and the word decodes as a harmless read of register 0. That is exactly how `fatfs/diskio.c` detects support at run time: it sets register 0 to the complement of two known bytes, issues the burst, and sees which of the two answers comes back. A binary built for burst therefore still runs correctly on a pre-v0.74 core, just at the older speed.

Chapter 11 — Ethernet

100 Mbit Ethernet, on the MEGA65's own RJ45 socket. The core drives the KSZ8081RND PHY over RMII and does the framing itself: preamble, CRC, and a destination-MAC filter in hardware.

The interface is off until you switch it on. The PHY is held in reset until software writes `ETH_CTRL` bit 0, so a machine that never touches these registers puts nothing on the wire and hears nothing from it.

The buffers are memory, not a port

This is the one thing to understand before writing a driver. The SD controller reaches its sector buffer through a cursor register, which costs about 1024 register accesses per sector. A 1500-byte packet through that shape would be slower than the wire it came off. So the packet buffers are plain memory in the CPU's address space and moving a packet is a copy:

Address	Contents
`$3A100000` .. `$3A1007FF`	RX window: the oldest unread packet
`$3A100800` .. `$3A100FFF`	TX window: the packet you are building

32-BIT ACCESSES ONLY. The control bridge carries a byte mask that these memories honour, but a byte or halfword store from C does not reliably reach them as one — what lands is the whole word. Assemble bytes with shifts. `vih_eth.h` does this for you; a `memcpy` onto a `uint8_t *` will appear to work and then corrupt packets.

Four receive slots, one transmit. `ETH_RXBUF` always shows the oldest unread packet; writing `ETH_RXPOP` releases it and the next appears at the same address. Software sees one window either way — the ring only means a frame arriving while you are still copying the last one is not lost. With all four full the next packet is dropped and counted in `ETH_DROPS`.

Registers

Address	Name	Dir	Contents
`$3A000260`	ETH_CTRL	W	[0] enable, [1] promiscuous, [2] keep bad CRC, [3] RX late
`$3A000264`	ETH_STATUS	R	[0] link up, [1] RX packet waiting, [2] TX busy
`$3A000268`	ETH_RXLEN	R	length of the packet in the RX window
`$3A00026C`	ETH_RXPOP	W	any write releases the RX slot
`$3A000270`	ETH_TXLEN	W	length to send; the write starts the transmission
`$3A000274`	ETH_MAC_LO	R/W	MAC bytes 0-3
`$3A000278`	ETH_MAC_HI	R/W	MAC bytes 4-5
`$3A00027C`	ETH_DROPS	R	frames lost because all four RX slots were full
`$3A000280`	ETH_CARRIER	R	times carrier was seen — 0 means nothing arrives at all
`$3A000284`	ETH_FRAMES	R	frames that completed reception
`$3A000288`	ETH_CRCERR	R	frames that arrived and failed their checksum
`$3A00028C`	ETH_PHY	R	[15:0] PHY status reg 1, [31:16] PHY id high — a non-zero id means the management bus works, whatever link says

(`vih_eth.h` is the authority on this table; earlier revisions of this chapter had RXPOP/TXLEN swapped and listed a raw MDIO port that software never sees.) Reset MAC is `02:56:49:58:45:4E` (locally administered). A received frame raises **interrupt source 3** (chapter 9), so a driver need not poll. First things to check on new hardware: `ETH_PHY`, then `ETH_CARRIER` — they tell apart "no cable", "PHY asleep" and "MAC broken", which otherwise look identical.

The protocol stack

`risc/net/` is a small ARP + IPv4 + UDP stack, about 4.8 KB of code, sitting on `vih_eth.h`. **IPv4 only.** It learns MAC addresses from any on-link IPv4 sender, not only from ARP replies, because a host that has already ARPed for you will send data without ever asking again.

Test app: `nettest` — link state, PHY identity, counters and a UDP echo.

Chapter 12 — Virtual Floppy Drives

> **Up to core v0.93 only.** The drives are mounted from the F12 menu and served by the framework's QNICE co-processor; from v0.94 there is neither. The registers stay and read "not mounted, size 0" – start no transfer on an unmounted drive, it would never finish.

Two drives, **A:** and **B:**, each holding a disk image mounted from the F12 menu the way every other MEGA65 core mounts one. Images are ordinary files on the SD card — a 1.44 MB FAT12 floppy image is the intended shape, and one written here is readable by a PC. `.IMG` files appear in the file browser alongside `.RVK`, `.BAS` and `.PRG`.

Mounting is instant. Nothing is copied: the framework opens the file and each sector access seeks into it. Unmount by pressing Space on a mounted drive's menu line.

How a sector actually moves

There is no DMA here, and no SD access by the CPU. The core hands a sector number to the QNICE co-processor, which reads or writes that part of the image file and moves 512 bytes through a shared window. Two consequences:

- **It is not fast.** Every sector is a round trip through QNICE. This is sized for a directory, a `LOAD`, or a boot block — not for streaming a game's assets. For that, use the SD controller directly (`vih_sd.h`).
- **A write is flushed as it happens**, so a sector that reported success is in the file. What it is not is a guarantee about the card's own internal housekeeping, so do not cut the power the instant a save finishes.

Registers

Drive A: is the block below; drive B: is the same layout at `$3A0002B0` through `$3A0002BC`.

Address	Name	Dir	Contents
`$3A0002A0`	DISK_A_LBA	W	sector number for the next transfer
`$3A0002A4`	DISK_A_CTRL	W	1 = read, 2 = write; starts it
`$3A0002A8`	DISK_A_STATUS	R	[0] busy, [1] mounted, [2] read-only
`$3A0002AC`	DISK_A_SIZE	R	image size in bytes

Writing CTRL starts exactly one transfer and is ignored while that drive is busy — a second request mid-transfer would move the sector number under QNICE's feet. Poll `busy` until it clears.

The sector buffers

Address	Contents
`$3A101000` .. `$3A1011FF`	drive A:, 512 bytes
`$3A101200` .. `$3A1013FF`	drive B:, 512 bytes

32-BIT ACCESSES ONLY, the same rule and the same reason as the Ethernet packet buffers above.

From C

```
#include "vih_disk.h"

uint8_t sector[512];

if (vih_disk_mounted(VIH_DISK_DRIVE_A))
{
    if (vih_disk_read(VIH_DISK_DRIVE_A, 0u, sector) == 0)
    {
        /* sector 0 of the image: a FAT12 boot sector */
    }
}
```

`vih_disk_read()` and `vih_disk_write()` return 0 on success, −1 on timeout, −2 if nothing is mounted, −3 if the image was mounted read-only. The wait is bounded rather than infinite: a driver that spins forever on a drive that never answers takes the whole machine with it and looks like a crash.

Test app: `disktest` — mount state, a sector hex dump, and a write test that restores the sector afterwards.

Bootable disks

A disk whose sector 0 carries the signature `V32B` at offset `$3E` BOOTS when mounted in drive A: — the kernel copies the sector to 6502 RAM at `$A000` and enters it at `$A042`, with no prompt and no key held. Only A: boots (the same disk in B: is just data), and only while BASIC sits at Ready. The boot block is 6502 code that reads further sectors through the `$B0B0`–`$B0B8` POKE registers (Appendix B) — enough to load a program or stream a `.RVK` into staging and launch it. The full story, the register detail and the two classic boot-code traps are in `docs/BasicMachine.txt` (“Bootable disks”); working examples are in `risc/bootdisk/`.

Appendix A — Register Index (by address)

Address	Name	Dir	Chapter
`$3A000004`	RASTER_HPOS	R	2

Address	Name	Dir	Chapter
$3A00000C	RASTER_VPOS	R	2
$3A000100	BOOT_ACK	W	(boot protocol)
$3A000104	BORDER_COLOR	W	2
$3A000108	BG_COLOR	W	2
$3A00010C	COPPER_CNT	W	5
$3A000110	FONT_CTRL	W	3
$3A000114	CTRL_WORD	W	2
$3A000118	TILE_BRAM_SEL	W	3 (legacy select)
$3A00011C	TILE_SCROLL	W	3
$3A000120	TILEPATCH_EN	W	3 (legacy patch)
$3A000124	TILE_PRIORITY	W	2
$3A000128	SPRITE_ENABLE	W	4
$3A00012C	TILES_ENABLE	W	3
$3A000130	BORDER_DIMS	W	2
$3A000134	GFX_ENABLE	W	2
$3A000138	AUDIO_ENABLE	W	8
$3A00013C	SPRITE_POS	W	4
$3A000140	SPRITE_ATTR	W	4
$3A000144	SDRAM_ADDR	W	9
$3A000148	SDRAM_DATA	W	9
$3A00014C	SDRAM_STATUS	R	9
$3A000150	SPRITE_COLL	R	4
$3A000154	PALETTE	W	2
$3A000158	GFX_PTR	W	2
$3A00015C	GFX_SCROLL	W	2
$3A000160	SID	W	7
$3A000164	PCM_IDX	W	8
$3A000168	PCM_DATA	W	8
$3A00016C	TILEMAP_PTR	W	3
$3A000170	FONT_WR_ADDR	W	3

Address	Name	Dir	Chapter
$3A000174	FONT_WR_DATA	W	3
$3A000178	COPPER_WR_ADDR	W	5
$3A00017C	COPPER_WR_DATA	W	5
$3A000180	JOYSTICK	R	9
$3A000184	RASTER_TARGET	W	9
$3A000188	IRQ_STATUS	R/W1C	9
$3A00018C	IRQ_MASK	W	9
$3A000190	KEYBOARD	R/W	9
$3A000194	OSD_STATUS	R	9
$3A000198	OSD_SIZE	R	9
$3A00019C	OWN_RELEASE	W	9
$3A0001A0	BLIT_SRC	W	6
$3A0001A4	BLIT_DST	W	6
$3A0001A8	BLIT_DIMS	W	6
$3A0001AC	BLIT_MOD	W	6
$3A0001B0	BLIT_CTRL	W	6
$3A0001B4	BLIT_STATUS	R	6
$3A0001B8	BLIT_TEX_U	W	6
$3A0001BC	BLIT_TEX_V	W	6
$3A0001C0	BLIT_TEX_DU	W	6
$3A0001C4	BLIT_TEX_DV	W	6
$3A0001C8	BLIT_TEXIDX	W	6
$3A0001CC	BLIT_TEXDATA	R	6
$3A0001D0	BLIT_CBASE	W	6
$3A0001D4	BLIT_LUT	W	6
$3A0001D8	BLIT_QUEUE	W	6 (v0.85)
$3A0001DC	ICACHE	W	9 (v0.88)
$3A0001E0	SDRAM_PHASE	W	9
$3A0001E4	CACHE_INVAL	W	9
$3A0001E8	JOYSTICK2	R	9

Address	Name	Dir	Chapter
$3A0001EC	MOUSE	R	9
$3A0001F0	SD_CTRL	W	10
$3A0001F4	SD_CMD	W	10
$3A0001F8	SD_STATUS	R	10
$3A0001FC	TEMP	R	9 (v0.82)
$3A000200	FPU_A	W	(vih_regs.h)
$3A000208	FPU_RES	R	(vih_regs.h)
$3A000210+	FPU_GO ops 0-10	W	(vih_regs.h; ..$3A000238)
$3A00023C	CORE_VERSION	R	9 (v0.98)
$3A000260	ETH_CTRL	W	11
$3A000264	ETH_STATUS	R	11
$3A000268	ETH_RXLEN	R	11
$3A00026C	ETH_RXPOP	W	11
$3A000270	ETH_TXLEN	W	11
$3A000274	ETH_MAC_LO	R/W	11
$3A000278	ETH_MAC_HI	R/W	11
$3A00027C	ETH_DROPS	R	11
$3A000280	ETH_CARRIER	R	11
$3A000284	ETH_FRAMES	R	11
$3A000288	ETH_CRCERR	R	11
$3A00028C	ETH_PHY	R	11
$3A000290	HDMI	W	2 (v0.95)
$3A000294	PAL_SEL	W	2 (v0.95)
$3A000298	VIDEO_MODE	W	3 (v0.95)
$3A000300	CPU8_CTRL	R/W	9 (v0.95)
$3A000304	CPU8_SPEED	R/W	9 (v0.95)
$3A000308	CPU8_TRAP	R/W	9 (v0.95)
$3A00030C	CPU8_ROM	R/W	9 (v0.95)
$3A000310	CPU8_STATUS	R	9 (v0.95)
$3A000314	CPU8_TRAPINFO	R	9 (v0.95)

Address	Name	Dir	Chapter
$3A000318	CPU8_REPLY	W	9 (v0.95)
$3A00031C	CPU8_PC	R	9 (v0.95)
$3A000320	CPU8_KEYS0	R/W	9 (v0.95)
$3A000324	CPU8_KEYS1	R/W	9 (v0.95)
$3A000328	CPU8_JOY	R/W	9 (v0.95)
$3A00032C	CPU8_VICFB	R/W	9 (v0.95)
$3A000330	CPU8_VICPOS	R	9 (v0.95)
$3A000334	CPU8_RESUME	R/W	9 (v0.95)
$3A000338	CPU8_STOPPOS	R	9 (v0.95)
$3A00033C	CPU8_YM	R/W	9 (v0.95)
$3A0002A0	DISK_A_LBA	W	12
$3A0002A4	DISK_A_CTRL	W	12
$3A0002A8	DISK_A_STATUS	R	12
$3A0002AC	DISK_A_SIZE	R	12
$3A0002B0	DISK_B_LBA	W	12
$3A0002B4	DISK_B_CTRL	W	12
$3A0002B8	DISK_B_STATUS	R	12
$3A0002BC	DISK_B_SIZE	R	12
$3A0002C0	BLIT_ZCTRL	W	6 (v0.90)
$3A0002C4	BLIT_ZBASE	W	6 (v0.90)
$3A0002C8	BLIT_ZSTART	W	6 (v0.90)
$3A0002CC	BLIT_ZSTEP	W	6 (v0.90)
$3A0002D0	BLIT_ZCLEAR	W	6 (v0.90)
$3A0002D4	TRI_XL	W	6 (v0.93)
$3A0002D8	TRI_DXL	W	6 (v0.93)
$3A0002DC	TRI_XS	W	6 (v0.93)
$3A0002E0	TRI_DXS	W	6 (v0.93)
$3A0002E4	TRI_XS2	W	6 (v0.93)
$3A0002E8	TRI_DXS2	W	6 (v0.93)
$3A0002EC	TRI_ROWS	W	6 (v0.93)

Address	Name	Dir	Chapter
$3A0002F0	TRI_DZL	W	6 (v0.93)
$3A0002F4	TRI_DUL	W	6 (v0.93)
$3A0002F8	TRI_DVL	W	6 (v0.93)
$3A0002FC	TRI_CLIP	W	6 (v0.93)
$3A100000	ETH RX BUFFER	R	11 (2 KB, 32-bit only)
$3A100800	ETH TX BUFFER	W	11 (2 KB, 32-bit only)
$3A101000	DISK A: SECTOR	R/W	12 (512 B, 32-bit only)
$3A101200	DISK B: SECTOR	R/W	12 (512 B, 32-bit only)
$3A038A00	TILE MAP 0	R/W	3
$3A040200	TILE MAP 1	R/W	3

Appendix B — BASIC POKE Quick Reference

BASIC sees the VIH through a 6502-side I/O page at $B000 (details in docs/BasicMachine.txt). $B000 is laid out like a C64's $D000, so $B020 is the border exactly as $D020 is on a C64 — port a sprite routine by changing one digit.

VIC-II layout, offset for offset:

POKE (dec)	6502 addr	C64	Function
45056+n*2,v	$B000+	$D000	sprite 0-7 X (n = sprite)
45057+n*2,v	$B001+	$D001	sprite 0-7 Y
45072,v	$B010	$D010	sprite X bit 8, one bit per sprite
PEEK(45074)	$B012	$D012	raster line (read)
45077,v	$B015	$D015	sprite enable, sprites 0-7
45083,v	$B01B	$D01B	1 = sprite behind the background
PEEK(45086)	$B01E	$D01E	sprite-sprite collisions (read)
PEEK(45087)	$B01F	$D01F	sprite-background collisions (read)
45088,v	$B020	$D020	border color (all 256)
45089,v	$B021	$D021	background color
45095+n,v	$B027+	$D027	sprite 0-7 color

VIXEN has more hardware than a VIC-II. Those registers start past $B02E so they can never collide with something mirrored later:

POKE (dec)	6502 addr	Function
45104,n	`$B030`	select sprite 0-15 for the six below
45105,v	`$B031`	selected sprite X, low byte
45106,v	`$B032`	selected sprite X bit 8
45107,v	`$B033`	selected sprite Y
45108,v	`$B034`	selected sprite image slot (4096-byte SDRAM)
45109,v	`$B035`	selected sprite color (1bpp sprites)
45110,v	`$B036`	selected sprite flags (8bpp, flip, on-top…)
45112,v	`$B038`	sprite enable, sprites 8-15
45120,v	`$B040`	tile layer on/off
45121,v	`$B041`	tile layer in front of the bitmap
45122/3,v	`$B042/3`	tile scroll X, low byte / bit 8
45124/5,v	`$B044/5`	tile scroll Y, low byte / bit 8
45126,v	`$B046`	tile map 0 or 1
45136,v	`$B050`	bitmap layer on/off
45137/8,v	`$B051/2`	bitmap pointer, 4096-byte blocks, lo / hi
45139/40,v	`$B053/4`	bitmap scroll X, low byte / bit 8
45141,v	`$B055`	bitmap scroll Y
PEEK(45152)	`$B060`	joystick 1 (read)
PEEK(45153)	`$B061`	joystick 2 (read)
PEEK(45154)	`$B062`	mouse X counter (read)
PEEK(45155)	`$B063`	mouse Y counter (read)
PEEK(45156)	`$B064`	mouse buttons, L/R/M in bits 0-2 (read)
45169/70,v	`$B071/2`	copper: scanline, low byte / bit 8
45171,v	`$B073`	copper: register id
45172-5,v	`$B074-7`	copper: value, four bytes, low first
45176,n	`$B078`	copper: commit the staged entry as entry n
45177,n	`$B079`	copper: run the first n entries
45184,n	`$B080`	sprite image: select slot n, reset the offset
45185/6,v	`$B081/2`	sprite image: offset within the slot, lo / hi
45187,v	`$B083`	sprite image: one byte, offset auto-advances

POKE (dec)	6502 addr	Function
PEEK(45200/1)	`$B090/1`	jiffy clock, 50 Hz; read the LOW byte first
PEEK(45202-5)	`$B092-5`	the 6502's clock, cycles; read `$B092` first
45212,v	`$B09C`	beeper, one byte: 0 = silence, 1-255 = 8660*2^((v-255)/32) Hz
PEEK(45215)	`$B09F`	kernel build number
45220,c	`$B0A4`	cursor column 0-39, 0-79 in 80 columns (read or write)
45221,r	`$B0A5`	cursor row 0-24 (read or write)
45222,n	`$B0A6`	0 = home, 1 = clear and home
45232,d	`$B0B0`	disk sector access: drive, 0 = A: 1 = B:
45233-5,v	`$B0B1-3`	disk: sector number (LBA), low byte first
45236,n	`$B0B4`	disk: sector count
45237/8,v	`$B0B5/6`	disk: target address, low / high
45239,b	`$B0B7`	disk: bank — 0 = 6502 RAM, 1 = SDRAM staging (512-byte units)
45240	`$B0B8`	disk: write = go, read = result (0 = ok)
PEEK(45244-6)	`$B0BC-E`	READF/WRITEF: bytes transferred, low byte first
PEEK(45247)	`$B0BF`	READF/WRITEF status: 0 ok, 1 no file, 2 I/O error, 3 address out of range, 4 bad unit, 5 no disk
45223,c	`$B0A7`	screen: column 0-39, 0-79 in 80 columns (read or write)
45224,r	`$B0A8`	screen: row 0-24 (read or write)
45225,f	`$B0A9`	screen: foreground color
45226,b	`$B0AA`	screen: background color
45227,f	`$B0AB`	screen: bit0 flipX, bit1 flipY, bit2 = char bit 8, bit3 = map 1
45228,c	`$B0AC`	screen: write = put the character and step on; read = the character there
45229/30,hz	`$B0AD/E`	beeper: tone in Hz, low / high; 0 = silence
45241/2/3,v	`$B0B9-B`	palette: red, green, blue (0-255 each)
45231,n	`$B0AF`	palette: entry 16-255 — writing it commits the color
45264,n	`$B0D0`	blitter: 1 fill, 2 copy, 3 copy with key — starts it
45265-7,v	`$B0D1-3`	blitter: source address, 24-bit chip
45268-70,v	`$B0D4-6`	blitter: destination address

POKE (dec)	6502 addr	Function
45271/2,v	`$B0D7/8`	blitter: width, low / high
45273/4,v	`$B0D9/A`	blitter: height, low / high
45275/6,v	`$B0DB/C`	blitter: source stride in bytes
45277/8,v	`$B0DD/E`	blitter: destination stride
45279,v	`$B0DF`	blitter: fill color, or the transparent key
45282,n	`$B0E2`	PCM: channel 0-3
45283/4,v	`$B0E3/4`	PCM: sample start, 4096-byte blocks
45285,n	`$B0E5`	PCM: length in 4096-byte blocks
45286,n	`$B0E6`	PCM: rate in hundreds of Hz (80 = 8000)
45287,v	`$B0E7`	PCM: volume 0-255
45288,n	`$B0E8`	PCM: 0 stop, 1 play once, 2 loop — acts
45289-91,v	`$B0E9-B`	chip RAM: address, 24 bits, low / mid / high
45292,v	`$B0EC`	chip RAM: the byte there; read or write steps on
45326,v	`$B10E`	6502 IRQ enable: bit0 = 50 Hz tick, bit1 = raster
PEEK(45327)	`$B10F`	which IRQ sources fired; reading acks
45328/9,v	`$B110/1`	raster IRQ line, low byte / bit 8
45248,c	`$B0C0`	font: character to define, 0-255 (restarts it)
45249,f	`$B0C1`	font: bit0 = code bit 8, bit1 = 8bpp, bit2 = 8x16 (restarts it)
45263,v	`$B0CF`	font: one glyph byte, offset auto-advances
45262,f	`$B0CE`	font: bit0 = show uploaded font, bit1 = 8x16
PEEK(45280)	`$B0E0`	last key code (128 + C65 keynum)
PEEK(45281)	`$B0E1`	1 while that key is still held
45293,c	`$B0ED`	a key code to ask about
PEEK(45294)	`$B0EE`	1 if THAT key is down right now
45295,v	`$B0EF`	1 = raw keyboard: the editor stops seeing keys

A C64 pokes sprite pixels into 6502 RAM and points the VIC at them. VIXEN keeps sprite images in SDRAM, where slot *n* is chip address *n*×4096, so `$B080` is how BASIC gets them there. A **1bpp** image is 8 bytes per row, 64 rows, 512 bytes; byte 0 is the leftmost 8 pixels and bit 7 the leftmost pixel of that byte — the same order a C64 sprite uses, so C64 sprite DATA maps straight across. An **8bpp** image is one byte per pixel, 4096 bytes.

Joystick bits are active-high and NOT in a compact order: `1` down, `2` up, `4` fire, `8` right, `32` left. `IF (PEEK(45152) AND 4) THEN ...` tests fire on port 1.

Mouse counters free-run and wrap at 256, like the Amiga's `JOY0DAT` — subtract your previous reading to get movement.

Screen cells

A C64 pokes `1024+x` for a character and `55296+x` for its color. VIXEN's cells are 32 bits — character, foreground, background, two flip bits — so they are reached as a cursor instead: set the position and the colors once, then each write to `$B0AC` puts a character and steps on, wrapping to the next row.

```
10 POKE $B0A7,0 : POKE $B0A8,0 : POKE $B0A9,7
20 FOR I=0 TO 39 : POKE $B0AC,65+(I AND 15) : NEXT
```

Colors and flags stick, so a full row is one POKE per cell rather than five — which at 3 MHz is the difference between a screenful appearing and a screenful happening. `PEEK($B0AC)` reads the character already in the cell and does *not* move on, so a program can look at what it just wrote.

`READF 4,"SCREEN.MAP",0` fills the screen from a file: one byte per cell, drawn with the current colors and flags, the address being the starting cell counted the way the cursor moves (0 = top left, 40 = start of row 1).

Coordinates are what you see — 0,0 is the top-left visible character.

Map 0 is the console's. It keeps its own copy of the screen for scrolling, so a cell you poke there is correct until the next scroll redraws that row and wipes it. Bit 3 of `$B0AB` draws into map 1 instead, which nothing else touches; `POKE $B046,1` then shows it. Drawing on map 1 and flipping is the intended way to use the screen as a bitmap, and is verified working on hardware and in the emulator.

Drawing: CANVAS and LINE

```
10 CANVAS $200000,512,384,224      : REM address, stride, width, height
20 LINE 0,0,383,223,1              : REM x0,y0,x1,y1,color
30 LINE 0,223,383,0,2,5            : REM ...and a thickness
```

`CANVAS` is set once and every drawing command uses it. With no `CANVAS`, drawing goes to the bitmap layer — whatever `$B051`/`$B052` points at, 384x224, stride 512 — so `LINE` works as soon as the layer is on.

Coordinates are **clipped**, not trusted: the blitter has no clipping, and a line running off the canvas would write across whatever follows it in chip RAM. Thickness is drawn as parallel lines offset across the direction of travel, so the ends are square.

```
10 RECT 20,20,120,80,2,0        : REM solid; thickness 0
20 RECT 160,20,120,80,7,4       : REM a frame four pixels thick
30 IMAGE 60,40,128,128,64*4096  : REM a 64x64 image, drawn at 128x128
```

`RECT` is one blitter fill when solid and four when it is a frame. `IMAGE` draws a **64x64, 8bpp** image from chip RAM — the same size as a sprite slot (slot N at N*4096) but **not the same**

format: sprite images are 1bpp, an `IMAGE` source is one byte per pixel, 4096 bytes. Color 0 is transparent in both. Giving `w,h` different from `sw,sh` scales in hardware at no extra cost, and the texture cache reloads only when the source address changes.

The arguments are evaluated in the ROM and left at `$B0F6-$B0FF` (coordinates, color, thickness), `$B100-$B108` (the canvas) and `$B109-$B10D` (the image source); machine code can fill those in and call the traps directly, exactly as with `READF`.

The blitter

Fill, copy, and copy-with-transparency, at sixty million pixels a second instead of BASIC's three. Set up the operation, then write `$B0D0` to start it:

```
10 POKE $B0D4,0 : POKE $B0D5,0 : POKE $B0D6,32   : REM dest $200000
20 POKE $B0DD,0 : POKE $B0DE,2                   : REM dest stride 512
30 POKE $B0D7,100 : POKE $B0D8,0                 : REM width 100
40 POKE $B0D9,50  : POKE $B0DA,0                 : REM height 50
50 POKE $B0DF,2 : POKE $B0D0,1                   : REM fill with color 2
```

Give it the stride, not the modulo. The hardware wants "bytes to skip at the end of each row" — stride minus width — and the kernel does that subtraction for you. A stride is a property of the surface (512 for the bitmap layer) and stays put while widths change.

Operation 3 skips source pixels equal to `$B0DF`, which is how a sprite with a transparent background is drawn into a bitmap.

The kernel waits for the blitter to finish and invalidates the CPU cache afterwards, so reading a drawn pixel back through the chip RAM window gives what was drawn rather than what was there before.

PCM sample playback

Four channels of sampled sound. `READF` puts the sample in chip RAM; these registers tell a channel where it is:

```
10 READF 1,"SOUND.RAW",$100000
20 POKE $B0E2,0                    : REM channel 0
30 POKE $B0E3,0 : POKE $B0E4,1     : REM start block 256 = $100000
40 POKE $B0E5,2                    : REM two 4096-byte blocks
50 POKE $B0E6,80                   : REM 8000 Hz
60 POKE $B0E7,255                  : REM volume
70 POKE $B0E8,1                    : REM play once (2 = loop, 0 = stop)
```

Samples are signed 8-bit — 0 is silence, 128 is nearly full deflection. **Start and length are in 4096-byte blocks**, the same granularity as sprite slots, which is what keeps a sample word-aligned; an unaligned sample drifts out of tune after its first loop and no emulator will show it.

Changing `$B0E6` between triggers is how one sample becomes a scale.

The chip RAM window

`READF` fills chip RAM and `WRITEF` saves it, but this is how BASIC reads and writes single bytes of it:

```
10 POKE $B0E9,0 : POKE $B0EA,0 : POKE $B0EB,32   : REM address $200000
20 POKE $B0EC,7                                  : REM write a pixel
30 PRINT PEEK($B0EC)                             : REM ...and the NEXT byte
```

Both reading and writing step the address on, because walking a buffer is the common case. Set the address again to look at the same byte twice.

Setting the address also invalidates the CPU cache, since the blitter and the sprite DMA write chip RAM behind the CPU's back — without that, reading back what the blitter just drew returns whatever the cache held.

Which keys are down

`$B0E0`/`$B0E1` report the last key and whether it is still held — enough for a menu, useless for a game, because a second key press throws the first key's state away. `$B0ED`/`$B0EE` ask about any key at any time:

```
10 POKE $B0ED,184 : IF PEEK($B0EE) THEN PRINT "1 IS DOWN"
```

Any number of keys read correctly at once. Unlike a C64 — which scans its matrix through `$DC00`/`$DC01` and suffers ghosting on some three-key combinations — this is a record of what the hardware reported, not a matrix being scanned.

The codes are **C65 matrix numbers**: `1` is 184 but `3` is 136, in no order you can guess. Press the key and `PEEK($B0E0)` to find its number.

Note that `GET` is not a keyboard read: typing edits the screen and only Enter feeds a line to BASIC, so `GET` returns nothing until Enter is pressed.

Turn the editor off while polling. Otherwise every key pressed also prints itself into the middle of whatever the program is drawing:

```
10 POKE $B0EF,1     : REM keys go only to the registers
...
99 POKE $B0EF,0     : REM back to normal typing
```

RUN/STOP still works in raw mode, and a break clears it — a program that crashes with it on cannot leave the keyboard unable to type.

6502 interrupts

`POKE $B10E,1` makes the 50 Hz tick pull the 6502's IRQ line; bit 1 adds a raster-compare source at the line in `$B110/$B111`. EhBASIC's `ON IRQ <line>` / `IRQ ON` / `RETIRQ` handle it at BASIC level; machine code patches the RAM handler at `$020D` (the `$FFFE` vector ends up there, through a `JMP $020D` at `$F3F0` the kernel watches since build 138). `PEEK($B10F)` says which sources fired and acks them. Breaking a program disables the sources. Delivery lands within microseconds — under a scanline — so border effects hold still; cycle-exact per-line work still belongs to the copper.

The palette

256 colors exist; 32 are set at power-on (the 16 C64 colors and a 16-step grey ramp) and the rest are black. Set red, green and blue in any order, then poke the entry number to commit:

```
10 POKE $B0B9,255 : POKE $B0BA,128 : POKE $B0BB,0
20 POKE $B0AF,64                  : REM entry 64 is now orange
```

Entries 0-15 are refused. Those sixteen are the machine's identity — what the boot screen, the console and every C64-style program are drawn in — so a program cannot repaint them by accident. 16-31 (the greys) are allowed.

The hardware stores RGB565, so the low bits of each component are dropped: `255,255,255` and `249,253,249` give the same white.

`READF 5,"NAME.PAL",16` loads a whole palette from a file — three bytes an entry, red/green/blue, starting at the entry the address names. A file that would reach entries 0-15 is refused outright rather than half-applied.

The beeper

The machine's third voice, beside the SID and the PCM engine: one square wave, the same thing a PC speaker is. `$B0AD`/`$B0AE` take the tone **in Hz** — the hardware wants a divisor, but the division happens in the kernel so a program pokes 440 and gets 440 Hz. Either byte sets the tone from the pair, and 0 is silence.

```
10 POKE $B0AE,1 : POKE $B0AD,184 : REM 440 Hz
20 FOR D=0 TO 2000 : NEXT
30 POKE $B0AD,0 : POKE $B0AE,0   : REM silence
```

`$B09C` is the whole tone in one byte (kernel build 124). Two POKEs per pitch means the beeper plays a wrong pitch between them, and a sweep from BASIC crackles; one write has no in-between. 0 is silence, 1-255 is a tone in equal steps, 32 to the octave — every +32 doubles the pitch:

```
Hz = 8660 * 2^((value - 255) / 32)

  1 = 35 Hz     31 = 68 Hz     63 = 135 Hz    95 = 271 Hz
127 = 541 Hz   159 = 1082 Hz  191 = 2165 Hz  223 = 4330 Hz  255 = 8660 Hz
```

PEEK returns the value poked. `$B09C` and `$B0AD`/`$B0AE` drive the same beeper: the last write wins.

```
10 FOR V=88 TO 162 : POKE $B09C,V : NEXT : REM a clean sweep up
20 POKE $B09C,0                          : REM silence
```

It needs nothing else switched on — the master audio enable is already set by the time BASIC is running.

The character set

`$B0C0` picks the character, `$B0CF` takes its bytes — 8 at 8x8, 16 at 8x16, 64 at 8bpp — and `$B0CE` bit 0 shows the result instead of the built-in ROM font. Bit 7 of each byte is the leftmost pixel, exactly as a C64 character, so existing charset data maps straight across.

```
10 POKE $B0C0,65 : REM define "A"
20 FOR I=0 TO 7 : READ B : POKE $B0CF,B : NEXT
30 POKE $B0CE,1 : REM show it
40 DATA 60,66,165,129,165,153,66,60
```

`READF 2,"name",off` fills the 1bpp font from a file and unit `3` the 8bpp one, the address being a byte offset — so `READF 2,"FONT.CHR",0` loads all 512 characters at once and `64*8` starts at character 64. The offset must be a multiple of 8. `WRITEF` cannot go the other way: the font memory has a write port and no read port, so a glyph can never be read back and unit 2 or 3 returns status 4.

Two things will catch you. Switching to an uploaded font that is still empty blanks **every** character on screen, `Ready` included — the console draws with the ROM font, exactly as a C64 blanks when you point the VIC at uninitialised RAM. And the data byte is at `$B0CF`, not the `$B0CD` you would expect: until build 101 the crunch matched keyword text inside hex literals, so `POKE $B0CD` became `POKE $B0` + the `CD` token and died as a syntax error. That is fixed — a `$` literal is now copied verbatim, as quoted strings always were, so `$DEC0`, `$DEF0` and `$B0CD` all work — but the register stays where build 100 put it.

`READF unit,"name",addr` reads a whole file into memory and `WRITEF unit,"name",addr,len` writes memory back out; unit `0` is 6502 RAM (`$0000`–`$FFFF`), unit `1` is chip RAM (`$000000`–`$EFFFFF`), and `READF -1,"name",0` reports a file's size without reading it. The last megabyte of chip RAM holds the kernel modules and is refused in both directions.

ALWAYS CHECK `PEEK($B0BF)` AFTER EVERY `READF` AND `WRITEF`. Neither one stops the program when it fails — a missing file, a full disk or an address out of range sets the status byte and returns, and the next line runs as if nothing happened. A program that does not read `$B0BF` cannot tell loaded data from whatever was in that memory beforehand, and will happily draw a sprite sheet that was never read. That is the price of being able to ask "is this file there?" at all, and it is only a fair trade if the check is written every time:

```
10 READF -1,"LEVELS.BIN",0
20 IF PEEK($B0BF)<>0 THEN PRINT "NO LEVELS":END
30 READF 1,"LEVELS.BIN",$100000
40 IF PEEK($B0BF)<>0 THEN PRINT "LOAD FAILED":END
```

Both are callable from machine code, and that is the whole of the interface — the BASIC keywords are a convenience on top of it, not a privileged path. Fill in the arguments, then `JSR` the entry:

write	address	meaning
`$B0C2/3`		unit, 16-bit: `$0000` 6502 RAM, `$0001` chip RAM, `$FFFF` (-1) size query
`$B0C4`		name length
`$B0C5/6`		name pointer, anywhere in 6502 memory; NOT terminated
`$B0C7-9`		address, 24 bits, low byte first
`$B0CA-C`		length, 24 bits — `WRITEF` only, mandatory
`JSR $BFE8`		do the READF

write	address	meaning
`JSR $BFE6`		do the WRITEF

```
lda #$01
sta $B0C2       ; unit 1, chip RAM
lda #$00
sta $B0C3
lda #name_end-name
sta $B0C4
lda #<name
sta $B0C5
lda #>name
sta $B0C6
lda #$00
sta $B0C7
sta $B0C8
lda #$10
sta $B0C9       ; $100000
jsr $BFE8       ; READF
lda $B0BF       ; 0 = it worked -- NEVER skip this
bne failed      ; nothing was loaded; the memory is untouched
```

The entry addresses are ordinary reads that the machine answers with an `RTS`, which is why a `JSR` to them works and why a stray `PEEK($BFE8)` performs a file read — the same mechanism `DIR`, `DEL` and `COPY` use.

THE REGISTERS PERSIST between calls, so a second transfer only writes what changed: the routine above can follow its `READF` with a new name, a length and `JSR $BFE6` to write the same memory back out, leaving the unit and address alone.

The `$B0B0–$B0B8` block is raw sector access to the mounted drives — it is what a bootable disk's 450-byte boot block loads with (chapter 12, "Bootable disks", and `docs/BasicMachine.txt`), and it works from ordinary BASIC too.

Files from machine code — `$B120–$B133`

`READF` and `WRITEF` move a whole file to or from an address. This block does everything else: list a directory, make one, delete a file or a whole tree, rename, copy in steps, and read a piece of a file from the middle. It is what VIXEN COMMANDER (`VCMD.PRG`) is built on, and it needs kernel build 117 or later.

BASIC's `DIR`, `COPY`, `MOVE`, `DEL` and `MKDIR` do the same jobs, but they read their arguments by scanning the BASIC program text and report by printing on the console — no use to a program with its own screen. Underneath, both reach the same kernel; there is no second filesystem.

write	address	meaning
`$B120`		unit: 0 = `A:`, 1 = `B:` (3 = `C:` is off – see below)
`$B121`		the DESTINATION unit, for COPY

write	address	meaning
`$B122/3`		path pointer — NUL-terminated, anywhere in 6502 memory
`$B124/5`		destination path pointer
`$B126/7`		buffer pointer: where an answer is written
`$B128`		buffer length; 0 means 256
`$B129-C`		file offset, 32 bits, low byte first
`$B12D`		**the command — writing it runs the command**

read	address	meaning
`$B12E`		status: 0 = ok
`$B12F-132`		count: a size, a byte count, or free space
`$B133`		flags: bit0 mounted, bit1 read-only, bit2 directory, bit3 the copy has more to do

command	`$B12D`	what it does
STAT	1	is there a disk, and how much room — count = free bytes
DIROPEN	2	start listing the directory named by the path
DIRNEXT	3	one entry into the buffer; status 10 = end of directory
DIRCLOSE	4	finish the listing
MKDIR	5	make a directory
DELETE	6	delete a file, or an EMPTY directory
RENAME	7	path -> path 2, same drive only
DELTREE	8	delete a directory AND EVERYTHING IN IT
COPYOPEN	9	begin a copy: unit/path -> unit 2/path 2
COPYSTEP	10	copy up to 8 KB; flag bit 3 set = call it again
COPYCLOSE	11	finish the copy

com-mand	$B12D	what it does
COPY-ABORT	12	give up, and delete the half-written destination
FOPEN	13	open a file to read — count = its size
FREAD	14	read from the offset in $B129 into the buffer
FCLOSE	15	close it
CHDIR	16	change that unit's current directory
GETCWD	17	write that directory into the buffer
LABEL	18	write the disk's name into the buffer
SIZE	19	count = size; flag bit 2 says it is a directory

Status values: 0 ok, 1 not found, 2 I/O error, 3 no disk in that drive, 4 no such unit, 5 this kernel has no disk support, 6 refused by the `C:` write jail, 7 a handle is open (or none is), 8 a missing path or buffer, 9 the tree is too deep, 10 end of directory, 11 `C:` refused because `A:` or `B:` is mounted (kernel build 120+: the card and the disk images on it cannot be driven at once, so `C:` is only available with both image drives empty).

A command runs inside the POKE that starts it. The status and the count are already there when the next instruction reads them — there is nothing to wait for and no interrupt to catch. Reading `$B12D` always gives 0.

The arguments persist between commands, exactly as `READF`'s do, so walking a directory is one DIROPEN and then DIRNEXT after DIRNEXT with nothing else written.

DIRNEXT writes **eighteen bytes**: twelve characters of name and a terminator, a flags byte (bit 0 = directory), then the size as four bytes, low byte first.

The name is always 8.3. A long name on the disk is reported as its DOS alias — `LONGFI~1.TXT` — which opens the same file and fits a 40-column screen.

```
; list the current directory of A:
        lda #$00
        sta $B120       ; unit 0 = A:
        lda #<path
        sta $B122
        lda #>path
        sta $B123
        lda #<entry
        sta $B126
        lda #>entry
        sta $B127
        lda #$02
        sta $B12D       ; DIROPEN
```

```
        lda $B12E
        bne failed
next:   lda #$03
        sta $B12D       ; DIRNEXT
        lda $B12E
        bne done        ; 10 = end of directory, anything else is an error
        ...             ; `entry` now holds one name, its flags and size
        jmp next
done:   lda #$04
        sta $B12D       ; DIRCLOSE
```

A copy is done in steps and that is deliberate. A copy off the card can run for seconds, and a machine that stops answering for seconds cannot show a progress bar or be interrupted. Call COPYSTEP until bit 3 of `$B133` clears, then COPYCLOSE; the count is the running total, so the caller can draw it. COPYABORT takes the half-written file with it — a truncated copy left behind looks like a real file.

DELTREE deletes everything underneath. There is no confirmation and no undo; it is the command a file manager puts a warning in front of. It stops at eight levels deep (status 9) and RUN/STOP stops it (status 7), in which case some of the tree is gone and some is not.

DRIVE `C:` IS SWITCHED OFF (kernel build 119) and unit 3 answers status 4, no such unit. The card can be driven by one side at a time and `A:`/`B:` are disk images stored *on* it, so a copy with `C:` at one end and an image drive at the other needs the card twice over and fails part way through. `VX_DRIVE_C` in `risc/basic_test.c` brings it back; while it is 1, writes to `C:` stay jailed inside `/VIXEN-32/` exactly as `SAVE`'s do and a refused one answers status 6. Reading is free anywhere.

While `C:` is enabled, the card is claimed for a command and released after, so the F12 browser (up to core v0.93) never goes blind — except while something of yours is open. An open file, an open directory or a copy in progress holds the card until the matching close, because releasing it unmounts the filesystem those handles depend on. Close what you open.

> **Reading `$B0E0` and `$B0E1` needs care in any program that polls them.** The kernel writes the key code and the held flag from its own loop while your 6502 code runs, so reading one and then the other can pair the PREVIOUS key's code with the NEW key's held flag — a keypress that never happened. It doubled letters in typed filenames and made one press of F7 act like Return. Read both twice and throw the pair away if either changed; the next poll is microseconds later.

Both counters latch on their lowest byte, the way a C64's `TI` has to: read `$B090` (or `$B092`) first and the remaining bytes come from a frozen copy, so a tick landing between two PEEKs cannot give you a half-updated number. Read them out of order and you will eventually get a reading that is wrong by 256.

`$B092` counts the 6502's cycles, so `cycles ÷ seconds` is the speed BASIC is actually running at — independent of how long any particular program takes. Since kernel build 138 EhBASIC runs on the core's own 6502 and this is its 25 MHz clock; up to 137 it counted fake6502's emulated cycles. `$B0A0-3` (instructions run, which only the emulator could count) reads 0 from 138 on. `$B09F` is the kernel build number, which is how you tell whether the card really has the image you just copied onto it.

The cursor moves by number, not by magic character. A C64 makes you print unprintable control codes to place the cursor, which is why its listings are full of characters you cannot type. Here `POKE 45220,col : POKE 45221,row` puts it where you want and `POKE 45222,1` clears the screen — and PEEKing the same two addresses tells you where the cursor is, which a C64 cannot do at all. `PRINT CHR$(147)` and `CHR$(19)` still work, so C64 listings behave.

`$B0E0` is how you find out what a key sends. The MEGA65 reports `128 + C65 matrix keynum`, NOT a PS/2 scancode — F1 is 132, not
5. Rather than working that out from a matrix diagram, press the key and

PEEK.

Move a sprite:

```
POKE 45077,1   : REM enable sprite 0
POKE 45056,160 : REM X
POKE 45057,100 : REM Y
POKE 45095,1   : REM white
```

Machine control:

POKE (dec)	6502 addr	Function
45296,1	`$B0F0`	launch the staged .RVK
45297,1	`$B0F1`	launch without chipset soft reset
45298,1	`$B0F2`	SDRAM eye-sweep instrument
45299,1	`$B0F3`	CPU-RAM test (expansion region + KERNAL shield)
45300,1	`$B0F4`	chip-RAM write speed test (512KB, prints KB/s)
46080+r,v	`$B400+r`	SID register r (0..28), C64 `$D400` style

First sound:

```
POKE 46104,15 : POKE 46080,137 : POKE 46081,65
POKE 46085,0  : POKE 46086,240 : POKE 46084,17   (gate on)
POKE 46084,16                                     (gate off)
```