P32W Programmer's Guide

Programs on the bare machine: registers, devices, the boot ROM and its monitor, the tools

P32W — Ken Rother's 32-bit TTL computer

Generated 2026-11-13 from the repository at 3cacc22 2026-11-13

Contents

P32W programmer's guide `docs/programmers-guide.md`

1. The machine in one page
2. Registers
3. Memory map (physical)
4. Instructions
5. Programs, the boot ROM and reset
6. The devices
7. Exceptions and interrupts
8. User mode and virtual memory
9. Tools

The boot sequence `docs/boot.md`

1. Power and reset
2. The reset microcode
3. The monitor (the boot ROM)
4. The kernel loader: the monitor's `b`
5. The kernel's start-up
6. User space
7. Shutting down and rebooting
8. In the emulators

The boot ROM monitor `docs/monitor.md`

1. Where it lives
2. Using it
3. Booting: `b`
4. Programs the monitor calls
5. Exceptions
6. How it works

software/ `software/README.md`

The monitor

assembler/ `assembler/README.md`

microcode/ `microcode/README.md`

P32W programmer's guide

docs/programmers-guide.md

How to write programs for the P32W: the registers, the memory map, the instructions, the devices on the I/O card, exceptions and interrupts, virtual memory, the boot ROM — and the tools that check your program before any hardware exists. The authorities behind this guide: isa-v0.md (instructions and encodings), isa-system.md (traps, interrupts, the MMU), instruction-reference.md (generated from the ISA table), and the decision log decisions.md. How the machine does it in hardware: architecture.md (the cards and the address map) and microarch.md (the microcode).

1. The machine in one page

- **32-bit**, load/store, 16 general registers, fixed 32-bit instructions, **byte addressing, little-endian**, aligned accesses only (a misaligned word or halfword traps).
- **No condition flags:** branches compare two registers (`BEQ`, `BLT`, …).
- **Two modes:** supervisor (reset, the kernel) and user. User programs run in a **virtual** address space of 2 GiB, translated by a software-managed TLB; supervisor code sees physical memory directly.
- **Memory:** 1 MiB boot ROM at address 0, RAM above it on one to four 16 MiB memory cards (up to 64 MiB), the I/O page at the top of the address space (D52).
- **Speed:** a 3 MHz microcycle, about 6–7 microcycles per instruction — roughly 0.5 MIPS. Every instruction, including multiply and divide, takes a fixed, small number of cycles.

2. Registers

Register	Use
`r0` (`zero`)	always reads 0; writes are discarded
`r1`–`r14`	general purpose
`r15` (`sp`)	stack pointer — **banked**: the supervisor SP in supervisor mode, the user SP in user mode; switching mode switches the bank
`PC`	instruction address (a multiple of 4)

Control registers — read with `MFCR rd, cr`, written with `MTCR cr, rs`, supervisor mode only:

#	Name	Contents
0	`STATUS`	bit 0 **S** (supervisor), 1 **IE** (interrupts enabled), 2 **PS** (previous S), 3 **PIE** (previous IE)
1	`EPC`	the PC of the instruction that trapped (or the next one, for an interrupt)
2	`CAUSE`	exception code; bit 31 set for an interrupt (then bits 2:0 = the IRQ line)
3	`BADADDR`	the faulting address (misaligned, bus error, privilege, TLB, protection); 0 otherwise
4	`VBR`	address of the one exception handler (multiple of 4)
5	`USP`	the user-mode `r15`, as seen from supervisor mode
6	`IPEND`	read-only: bit n = interrupt line n asserted
7	`IMASK`	bit n = 1 enables interrupt line n
8	`ASID`	the current address-space ID (8 bits)
9	`TLBHI`	a TLB entry's tag: VPN (31:12), I (8: instruction side), ASID (7:0)

#	Name	Contents
10	`TLBLO`	a TLB entry's data: PFN (31:12), X (2), W (1), V (0)

Reset state: supervisor mode, interrupts disabled (`IE` = 0, `IMASK` = 0), `VBR` = 0, PC = 0. **No other register is initialised** — set `sp` first.

3. Memory map (physical)

Range	Size	What
`0x0000_0000` – `0x000F_FFFF`	1 MiB	**boot ROM** (flash on the memory card); writing it is a bus error
`0x0010_0000` – `0x00FF_FFFF`	15 MiB	**RAM** on the first memory card (window 0)
`0x0100_0000` – `0x03FF_FFFF`	up to 48 MiB	**RAM** on more memory cards, 16 MiB each (windows 1–3, by jumpers; D52)
`0xFE00_0000` – `0xFE07_FFFF`	64 KiB a card	kept for video memory, at `0xFE00_0000 + n × 64 KiB`; the video card (revision 2, §6.6) has none on the bus, so nothing answers
`0xFFFF_8000` – `0xFFFF_FFFF`	32 KiB	**I/O page** (§6): eight 4 KiB card windows
everything else	—	nothing answers: **bus error**

A memory card that is not there does not answer either, so a program that sizes memory must be ready for a bus error (the kernel's probe is `kernel/vm.c`).

The I/O page is the top 32 KiB so that a sign-extended 16-bit offset from `r0` reaches it: `lbu r2, 0xFFFFFF05(r0)` is one instruction. The assembler accepts the full address as the offset.

4. Instructions

The full table, generated from the ISA source, is instruction-reference.md. In outline:

Group	Instructions
Arithmetic, logic (register)	`ADD SUB AND OR XOR NOR SLL SRL SRA SLT SLTU`
… with an immediate	`ADDI SLTI SLTIU` (sign-extended), `ANDI ORI XORI` (zero-extended), `SLLI SRLI SRAI`, `LUI`
Multiply, divide	`MUL MULH MULHU DIV DIVU REM REMU` — division by zero does not trap: `DIV`/`DIVU` give all ones, `REM`/`REMU` the dividend
Load, store	`LB LBU LH LHU LW`, `SB SH SW` — `rd, offset(rs)`
Branch	`BEQ BNE BLT BGE BLTU BGEU ra, rb, label` (±128 KiB)
Jump, call	`JMP label`, `CALL label` (±32 MiB), `JR rb`, `CALLR rb`, `RET`
System	`SYSCALL BREAK RTE MFCR MTCR WAIT TLBW HALT`

Assembler pseudo-instructions (`assembler/p32asm.py`): `NOP`, `MOV rd,rs`, `NEG`, `NOT`, `LI rd,value` (1 or 2 instructions), `LA rd,address` (always 2), `PUSH rs`, `POP rd`, `BEQZ/BNEZ ra,label`, and `BGT BLE BGTU BLEU` (the operands swapped).

Calls. `CALL` pushes the return address and jumps; `RET` pops it. The stack is **full-descending**: `sp` points at the last word pushed, and `PUSH` decrements before storing. Keep `sp` a multiple of 4.

A calling convention (used by the library and the examples; the ISA does not impose one): arguments in `r2`–`r5`, the result in `r1`; a routine may clobber `r10`–`r14`; it preserves `r1`–`r9` except the result. System calls (the test kernels): number in `r1`, arguments `r2`–`r5`, result `r1`.

Large constants and addresses: `LI r1, 0x12345678` becomes `LUI r1, 0x1234` + `ORI r1, r1, 0x5678`; `LA` always uses two words so a label's address may change between assembler passes.

Example: print a string

```
        li    sp, 0x01000000    ; top of the first memory card
        la    r2, msg
        call  uart_puts
        halt
msg:    .asciz "Hello, P32W\n"
        .include "../software/lib/p32io.inc"
```

5. Programs, the boot ROM and reset

At reset the CPU fetches from address 0 in supervisor mode: **the boot ROM is the first code that runs.** A ROM program must:

1. set `sp` (e.g. `li sp, 0x01000000`, the top of the first memory card);
2. initialise the console UART (`uart_init`, §6.1) — the emulators work without it, the hardware does not;
3. set `VBR` before anything can trap, if it wants to handle exceptions.

The assembler's output is a flat image starting at address 0: that is the ROM image, for the emulators (`p32emu image.bin`) and for the flash programmer alike.

The boot ROM monitor (software/monitor) is the standard ROM program: a console prompt to dump and modify memory, call a program, load sectors from the CompactFlash card, and **boot** a program from it (sector 0: `"P32B"`, sector count, load address, entry point). Its exception handler reports `CAUSE`, `EPC` and `BADADDR` and returns to the prompt.

Programming the ROM on the real machine: the four flash chips (SST39SF0x0) on the memory card hold byte lanes 0–3 of each word. Program them in a TL866-class programmer (split the image into lanes). (Writing them in place through the debug card, with a PROG jumper, was planned in D50 and dropped in D101: the memory card ties the chips' write line high.)

6. The devices

All I/O registers are in the top 32 KiB, in **card windows** of 4 KiB (window n at `0xFFFF_8000 + n × 0x1000`, chosen by jumpers on the card; D52). A card's registers are the top 256 bytes of its window, and at `+FF0` / `+FF1` / `+FF4` every card answers with its **card ID** (byte: 1 I/O, 2 disk, 3 video; the debug card answers no bus address, D101), its **revision** (byte) and its **interrupt status** (word, a bit per device). The default jumpers: the I/O card in window 7 (`FFFF_F000`), the disk card in window 6 (`FFFF_E000`), the video card in window 5 (`FFFF_D000`).

Byte registers (UARTs, ports, RTC, USB, network; the CF data register is a halfword) must be accessed with `LB`/`LBU`/`SB`; **timer registers** with `LW`/`SW`. Any other size, or an address no device uses, is a **bus error** — and the I/O page traps in user mode, so only the kernel touches devices.

Address	Device	IRQ
`FFFF_FF00`–`FF07`	UART 0 — the console	2

Address	Device	IRQ
`FFFF_FF08`–`FF0F`	UART 1	3
`FFFF_FF30`–`FF37`	UART 2	2
`FFFF_FF38`–`FF3F`	UART 3	3
`FFFF_FF10`	`TCOUNT` — timer count	0
`FFFF_FF14`	`TRELOAD` — timer period	
`FFFF_FF18`	`TCTRL` — bit 0 enable, bit 1 pending (write 1 to clear)	
`FFFF_FF20`	input port (8 bits)	
`FFFF_FF21`	output port (8 bits, reads back)	
`FFFF_FF40` / `FF41`	real-time clock (DS12C887): register number / register	5
`FFFF_FF50` / `FF51`	USB 0 (CH376S): data / command, status (§6.8)	6
`FFFF_FF52` / `FF53`	network coprocessor: data / status, control (§6.9)	7
`FFFF_FF54` / `FF55`	USB 1, as `FF50` / `FF51`	6
`FFFF_FFF0` / `FFF1` / `FFF4`	the I/O card's ID (1), revision, interrupt status (bit 0 timer, 1 UART 0, 2 UART 1, 3 UART 2, 4 UART 3, 6 USB 0, 7 USB 1, 8 network)	

The disk card (window 6): four IDE ports of 16 bytes, port p at `FFFF_EF00 + 16 p` (0 CF 0, 1 CF 1, 2 IDE A, 3 IDE B; §6.4).

Address	Device	IRQ
`FFFF_EF00`–`EF07`	port 0, the first CompactFlash card: the True-IDE task file	1
`FFFF_EF08`–`EF0F`	port 0's control block (`EF0E` alternate status)	
`FFFF_EF10`–`EF3F`	ports 1–3, the same layout	1
`FFFF_EF40` / `EF44` / `EF48`	the DMA engine: DMAADR, DMACNT, DMACTL (words; §6.7)	1
`FFFF_EFF0` / `EFF1` / `EFF4`	the disk card's ID (2), revision, interrupt status (bit p = port p)	

The interrupt levels are jumpers on the cards; the defaults are in the IRQ columns above and in §7, and §6.10 lists every source with its jumpers and its vector.

`software/lib/p32io.inc` has tested routines for everything below; its symbols (`U0`, `ULSR`, `CFCMD`, …) are used here.

6.1 The UARTs (TL16C550)

Each UART has the PC-standard 16550 registers at consecutive bytes:

Offset	Read	Write	With LCR bit 7 (DLAB) set
0	RBR: received byte	THR: byte to send	DLL: divisor low
1	IER: interrupt enable	IER	DLM: divisor high
2	IIR: interrupt identification	FCR: FIFO control	
3	LCR: line control	LCR	
4	MCR: modem control	MCR	
5	**LSR: bit 0 = data ready, bit 5 = transmitter ready**	—	
6	MSR: modem status	—	
7	SCR: scratch	SCR	

Baud rate = 1.8432 MHz / (16 × divisor): 12 → 9600, 6 → 19200, 1 → 115200. **Set-up** (`uart_init`): LCR = 0x80, DLL/DLM = divisor, LCR = 0x03 (8 data bits, no parity, 1 stop), FCR = 0x07 (FIFOs on and cleared), IER = 0.

Polled I/O:

```
putc:   lbu   r10, 0xFFFFFF05(r0)  ; LSR
        andi  r10, r10, 0x20       ; transmitter ready?
        beqz  r10, putc
        sb    r2, 0xFFFFFF00(r0)   ; THR
        ret
getc:   lbu   r10, 0xFFFFFF05(r0)
        andi  r10, r10, 0x01       ; data ready?
        beqz  r10, getc
        lbu   r1, 0xFFFFFF00(r0)   ; RBR
        ret
```

Interrupts: IER bit 0 interrupts when a byte arrives, bit 1 when the transmitter is empty. Read IIR to find out which (bits 3:0: `0x4` data received, `0x2` transmitter empty, `0x1` nothing pending); reading IIR clears a transmitter-empty interrupt, reading RBR a receive interrupt. Then enable the line in `IMASK` (bit 2 for UARTs 0 and 2, bit 3 for UARTs 1 and 3) and set `STATUS.IE`.

On the I/O card each UART has a DB9 connector at the front edge.

6.2 The timer

A 32-bit down-counter: `TCOUNT` counts down once per tick **while** `TCTRL` **bit 0 is set**; when it reaches 0 it reloads from `TRELOAD` and sets **pending** (`TCTRL` bit 1), which raises IRQ0 until acknowledged by writing `TCTRL` with bit 1 set (keep bit 0 set to keep counting).

```
        li    r2, 10000             ; 10 ms at 1 MHz
        sw    r2, 0xFFFFFF14(r0)    ; TRELOAD
        sw    r2, 0xFFFFFF10(r0)    ; TCOUNT
        li    r2, 1
        sw    r2, 0xFFFFFF18(r0)    ; enable
; in the IRQ0 handler:
        li    r2, 3
        sw    r2, 0xFFFFFF18(r0)    ; acknowledge, keep counting
```

The tick: 1 MHz on the hardware, divided from the bus clock, so it stops when the CPU is halted from the front panel. **In the emulators one tick per instruction**, so interrupt timing is exactly repeatable in tests. On the hardware `TCOUNT` may read 0 for one microcycle at expiry, just before the reload; in the emulators it never reads 0 while counting.

6.3 The ports

`FFFF_FF20` reads the 8-bit input port; `FFFF_FF21` is the 8-bit output port (reads back what was written). Both go to a connector on the I/O card; with nothing connected the input port reads `0xFF` (pull-ups). In the emulators `-p N` sets the input value and `-t` traces writes to the output port.

6.4 The CompactFlash disk

The disk card's port 0: a CF card in a socket at the card's front edge, in **True-IDE mode**; the data register is **16 bits wide** (D26); 512-byte sectors addressed by a 28-bit **LBA**.

Address	Read	Write
`EF00`	data: a **halfword** (`lhu` / `sh`), the first byte in bits 7:0	data

Address	Read	Write
`EF01`	error	features
`EF02`	sector count	sector count
`EF03` – `EF05`	LBA 7:0, 15:8, 23:16	same
`EF06`	drive/head	`0xE0` | LBA 27:24 (LBA mode, drive 0)
`EF07`	**status**: 7 BSY, 6 DRDY, 3 DRQ, 0 ERR	command

Cards start in 16-bit mode; `cf_init` sends features = `0x81`, command `0xEF` (SET FEATURES: no 8-bit transfers) in case earlier software chose 8-bit mode, and ignores a refusal. **Read a sector** (`cf_read`): count = 1, LBA, head, command `0x20`; wait for DRQ; read the data register as a halfword 256 times (into a halfword-aligned buffer). **Write** (`cf_write`): command `0x30`, wait for DRQ, write 256 halfwords, wait for BSY clear. A count of n moves n sectors with one command (wait for DRQ before each). Byte accesses to `EF00` reach only `DD0–7`: they work after SET FEATURES `0x01` on a card that offers 8-bit mode (optional in the CF standard). `0xEC` (IDENTIFY) returns 512 bytes of device information, word 0 = `0x848A` for a CF card. Reading the status register clears the disk interrupt (IRQ1), raised when a sector is ready and when a write or command completes. An empty port reads status `0x00` (not ready). The other ports work the same way, 16 bytes further on each.

In the emulators `-d image.img` gives a disk backed by a file of raw sectors (`dd if=/dev/zero of=disk.img bs=512 count=2048` for 1 MiB); `-d1`, `-d2`, `-d3` fill the other ports.

Ports 1–3. The four ports are the same circuit (D66), so everything above holds for each of them at its own base, `EF00 + 16 p`:

Port	Registers	What is on it	P32W Unix
0	`FFFF_EF00` – `EF0F`	the boot CF card, in a CF-to-IDE adapter at the front edge	`/dev/disk`
1	`FFFF_EF10` – `EF1F`	a second CF card, the same way	`/dev/disk1`
2	`FFFF_EF20` – `EF2F`	IDE channel A: a 40-pin cable, a master and a slave	`/dev/hda`, `hdb`
3	`FFFF_EF30` – `EF3F`	IDE channel B, the same	`/dev/hdc`, `hdd`

On an IDE channel bit 4 of the drive/head register picks the drive: `0xE0 | LBA 27:24` for the master, as above, and `0xF0 | LBA 27:24` for the slave. A port's offsets +8 to +15 are its control block: +14 is the alternate status (read) and device control (write). Each port's INTRQ is bit p of the card's interrupt status (`EFF4`), all four on the card's one level (IRQ1 by default, vectors `0x60`–`0x63`; §6.10). In the emulators `-d2s` and `-d3s` give the channels' slaves.

6.5 The real-time clock (D23)

A DS12C887 (the PC's MC146818 registers) with its own crystal and battery. Write the register number to `FF40`, then read or write it at `FF41`:

Reg		Reg	
`00`	seconds	`07`	day of the month
`02`	minutes	`08`	month
`04`	hours	`09`	year (00–99)
`06`	day of the week (1 = Sunday)	`32`	century
`0A`	A: bit 7 UIP (update in progress — wait while set); bits 6:4 = `010` runs the oscillator	`0B`	B: bit 7 SET (stop to set), bit 2 binary (else BCD), bit 1 24-hour
`0C`	C: interrupt flags (reading clears them)	`0D`	D: bit 7 = the battery is good

Reg		Reg	
`0E`–`7F`	battery-backed RAM (`32` is the century)		

P32W Unix sets B = `0x06` (binary, 24-hour, no interrupts) and keeps UTC (`kernel/rtc.c`). The chip ships with its oscillator off: write A = `0x20` once. Its interrupt (alarm, periodic, update) is IRQ5. In the emulators the clock follows the host's clock, or `-R SECONDS` (a fixed start).

6.6 The video card and PS/2 (D53, D103, D104)

VGA, 640 × 480, one bit a dot, drawn by an **HD63484 ACRTC**, and two PS/2 ports (keyboard and mouse). The card is video-card-acrtc.md; Hitachi's user's manual has the chip's registers and its 38 commands.

The picture is in the chip's own memory: none of it is on the bus. A program sends the chip commands (clear, copy, line, rectangle, fill, a word read or written), 16-bit words through its write FIFO, and the chip draws. The 64 KiB at `0xFE05_0000`, where the earlier revisions had the picture, does not answer.

Registers (window 5). The chip's two are **halfwords** (`LHU` / `SH`); the PS/2 section's are bytes. Any other size, or anything else in the page, is a bus error.

Address	Read	Write
`FFFF_DF00`	**the chip's status**, in bits 7–0 (mask the rest): bit 0 WFE (the write FIFO is empty), 1 WFR (it has room), 2 RFR (the read FIFO holds a word), 3 RFF (it is full), 5 CED (the chip is idle), 6 ARD (area detect), 7 CER (command error)	**the chip's address register**: which register `DF04` reaches (0: the FIFOs)
`FFFF_DF04`	the register addressed, or the next word of the read FIFO	the register addressed, or a word to the write FIFO
`FFFF_DF08`	**card status**: bit 0 FULL (a PS/2 byte waits), 1 SRC (0 the keyboard's, 1 the mouse's), 2 BUSY (a command not yet taken), 3 TEXT (text method A's parts are fitted), 4 the chip's IRQ	**a PS/2 command**: `01` interrupts on, `02` off, `03` acknowledge (no interrupt until the next byte), `04` mouse on, `05` off, `06` flush, `07` keyboard reset, `10`–`17` the keyboard's LEDs, `FF` start again
`FFFF_DF0C`	**the PS/2 byte**; reading it clears FULL. Read it only after FULL = 1	— (a bus error)
`FFFF_DFF0` / `DFF1`	card ID (3), revision (2)	—
`FFFF_DFF4`	interrupt status, a word: bit 0 the PS/2 section (FULL with its interrupts on), bit 1 the chip (its IRQ)	—

The chip. Registers `r02`–`rFF` are words at even addresses; from `r80` up the address register steps by 2 after each access, so a table loads with one address write. A reset leaves the chip stopped with ABT set (its command control register `r02` reads `8000`): the FIFOs are shut until a program clears it. The write FIFO holds **eight words; a word written when it is full is lost**, so write only while WFR is 1, or eight words after WFE. The status register's high byte is not defined.

The chip's interrupt is whatever its command control register enables, a bit in its low byte for each status flag: `0001` is WEE, an interrupt when the write FIFO is empty. It is a level: it stays until the flag falls or the enable is cleared.

Time. A command's work takes the chip from microseconds to tens of milliseconds: a character copied about 150 to 300 µs, a screen scrolled about 30 ms. CED says when it is done.

The keyboard arrives as ASCII: the card's PS/2 section turns the scan codes into characters (Shift, Caps Lock, Ctrl-letter, Enter as CR, Backspace `08`, Delete `7F`) and the arrows, Home, End, Page Up and Page Down into VT100 sequences. The mouse's 3-byte packets come whole, with SRC set. One queue of 63 bytes for both.

The PS/2 section takes its time (an ATmega328P running a loop; docs/video-card-acrtc.md §4.6). FULL and BUSY say when; a program follows them and no clock:

- Read `DF0C` only when FULL is set. After a byte is read FULL is clear for some 50 µs before the next one is there, so an interrupt brings one byte.
- Write `DF08` only when BUSY is clear. A command written over one not yet taken replaces it. BUSY lasts some 30 µs, or as long as the command before takes: a few milliseconds when the firmware has to talk to the keyboard or the mouse, half a second for a reset of either.
- After a reset the section is about two seconds starting, and drops a command it finds waiting when it begins: write one that does not matter first (02), and never stop the machine to wait for BUSY.
- Command 03 (acknowledge) keeps the interrupt quiet until a device sends something new, also for bytes the firmware already holds: do not use it in a handler. Command 06 (flush) does not empty the latch: a byte that shows FULL is still to be read.

The console (the kernel's `video.c`) keeps 25 rows of 80 characters of 8 × 16 dots on lines 40–439. The font is a picture in the chip's memory under the screen (from the card's font EPROM); a character is one graphic copy from it. Seven more pictures, made by the chip at start, give bold (for bright), underline, inverse and their mixes. The cursor is lines 14 and 15 of its cell in XOR.

At reset the chip is stopped: a black screen.

6.7 The disk card's DMA engine (D75)

The disk card's revision 1 (`EFF1` reads 1) adds a DMA engine: given a port, a memory address and a count, it does the drive's PIO itself — two accesses to the port's data register for each word, as the CPU's — and moves each word to or from memory in one bus cycle as a **bus master**, while the CPU runs on. So every drive and CF card works with it. P32W Unix does not use it yet: its driver moves the data itself.

Its registers are **words** (`LW` / `SW`; any other size is a bus error), and they never wait:

Address	Read	Write
`FFFF_EF40` DMAADR	the physical address of the next word	the first word's physical address (bits 1:0 ignored)
`FFFF_EF44` DMACNT	the words left (bits 15:0; 31:16 read 0)	the words to move, 0–65535 (bits 15:0)
`FFFF_EF48` DMACTL	the bits below (31:8 read 0)	the bits below; **any write clears DONE and ERR**

DMACTL bit	Read	Write
0	**BUSY**: running, with words left	**RUN**: 1 starts; 0 stops after the word in hand
1	DIR	**DIR**: 0 the drive to memory, 1 memory to the drive
3:2	the port	the port, 0–3
4	IE	**IE**: interrupt on DONE or ERR
5	MODE	MODE (multiword DMA): reserved, write 0
6	**DONE**: the count ran out	—
7	**ERR**: a memory cycle no card answered; DMAADR is its address, and that word was not counted	—

At reset DMACTL is 0 (stopped); DMAADR and DMACNT are not initialised.

A sector at a time (D75): issue the ATA command as for PIO (§6.4) and wait for DRQ; then write DMAADR, DMACNT = 128 (one sector) and DMACTL. The first halfword of each pair is the word's low one, so the sector lands in memory just as `cf_read`'s halfword loop would put it. A command for n sectors takes n transfers, each started after its own DRQ; DMAADR has run on past the sector, so the next one needs only DMACNT and DMACTL. A write is command `0x30`, DRQ, the engine with DIR = 1, then a wait for BSY clear, as `cf_write` does.

```
; port 2 (IDE A) has a sector ready (DRQ): move it into buf
        la    r2, buf               ; in RAM: physical, a multiple of 4
        sw    r2, 0xFFFFEF40(r0)    ; DMAADR
        li    r2, 128               ; one sector, in words
        sw    r2, 0xFFFFEF44(r0)    ; DMACNT
        li    r2, 0x09              ; RUN, DIR 0 (to memory), port 2
        sw    r2, 0xFFFFEF48(r0)    ; DMACTL
dmaw:   lw    r2, 0xFFFFEF48(r0)    ; (or set IE and take the interrupt)
        andi  r3, r2, 1             ; BUSY?
        bnez  r3, dmaw
        andi  r3, r2, 0x80          ; ERR?
        bnez  r3, error             ; (else DMACTL reads 0x48: DONE, port 2)
```

The interrupt: with IE, DONE or ERR is the card's fifth source, status bit 4 of `EFF4`, on the card's level (IRQ1 by default), vector window × 16 + 4 (`0x64`); writing DMACTL (with 0, say) acknowledges it. Both emulators model the engine; `emulator/test/diskdma_test.asm` uses all of it.

6.8 USB (D43)

Two USB-A ports for **storage** — a thumb drive as a removable disk — each a CH376S module in sockets on the I/O card, its USB socket at the front edge. The CH376S runs USB and the drive's protocol itself; P32W sees two byte registers a port:

Address	Read	Write
`FFFF_FF50`	USB 0: data	data
`FFFF_FF51`	USB 0: status — bit 7 **low** while an interrupt is pending, bit 4 busy	a command
`FFFF_FF54` / `FF55`	USB 1, as `FF50` / `FF51`	the same

A **command** is a byte written to `FF51`; its data, if any, follows in or out through `FF50`. The chip needs 1.5 µs after a command byte and 0.6 µs between data bytes. It reports each event — a step done, a drive plugged in or pulled out — as an interrupt: its INT# goes low, so status bit 7 reads 0, the I/O card's interrupt status bit 6 (USB 0) or 7 (USB 1) is set, and its level is raised (IRQ6 by default). The event is the chip's interrupt status code (`USB_INT_SUCCESS`, `USB_INT_CONNECT`, `USB_INT_DISK_READ`, …).

The chip's **sector mode**, in outline:

- `RESET_ALL` **first, always:** the card cannot reset the chip (the module resets itself only at power-on), so a reboot finds it as it was left.
- `CHECK_EXIST` with `0x55`: the data register reads back `0xAA` if the controller is there. `SET_USB_MODE` makes it a USB host.
- **A drive plugged in** (`USB_INT_CONNECT`): `DISK_INIT` (some drives need a few tries, about 5), then `DISK_CAPACITY`.

- **Read:** `DISK_READ` with the LBA (4 bytes) and a sector count; each sector then comes in eight 64-byte chunks, each announced by `USB_INT_DISK_READ`, fetched with `RD_USB_DATA0` and followed by `DISK_RD_GO`; `USB_INT_SUCCESS` ends it. **Write:** the same, with `DISK_WRITE`, `WR_HOST_DATA` and `DISK_WR_GO`.
- **Every wait needs a time limit,** so that a drive pulled out in the middle of a transfer is an error, not a hang.

The command and status codes are WCH's (the CH376 datasheet and its `CH376INC.H`); `kernel/usb.c` is a working driver. Expect about 150–250 KB/s (an estimate). P32W Unix has the drive as `/dev/usb`, a block device (`mkfs`, `mount`), and `/dev/rusb`, its raw sectors for the FAT tools (dos(1)). In the emulators `-U usb.img` plugs a drive holding `usb.img` into USB 0 (`-U1` into USB 1), and `-Uout N` pulls it out after N sectors.

6.9 The network coprocessor (D45)

A Raspberry Pi Pico 2 W and a W5500 Ethernet controller (a WIZ850io module, its RJ45 at the front edge) on the I/O card do the networking — TCP, UDP, TLS, DNS — and P32W asks them for socket operations through a **mailbox**: a request latch it writes and a reply latch it reads, each with a full / empty flag. The protocol is net-mailbox.md (version 1).

Address	Read	Write
`FFFF_FF52` `NDATA`	the next reply byte (reading takes it)	the next request byte
`FFFF_FF53` `NSTAT` / `NCTRL`	status: bit 0 **RXF** a reply byte is waiting, 1 **TXE** the request latch is empty, 2 **ATTN** the coprocessor has events, 3 **IEN** as written, 7 **READY** its firmware is running; bits 4–6 read 0	control: bit 3 **IEN** (interrupt while ATTN), bit 7 **RESET** (restart the coprocessor, empty both latches)

Every write to `NCTRL` sets IEN from its bit 3, so a RESET written with bit 3 clear also turns the interrupt off. After a RESET, READY is 0 until the coprocessor has started again; the machine's reset restarts it too, and leaves IEN 0 and both latches empty. Check TXE before each byte you write and RXF before each byte you read, and depend on those flags, never on timing (the Pico answers within a microsecond, the emulators' model at once):

```
nput:   lbu   r10, 0xFFFFFF53(r0)  ; NSTAT
        andi  r10, r10, 0x02       ; TXE: the request latch empty?
        beqz  r10, nput
        sb    r2, 0xFFFFFF52(r0)   ; NDATA
        ret
nget:   lbu   r10, 0xFFFFFF53(r0)
        andi  r10, r10, 0x01       ; RXF: a reply byte waiting?
        beqz  r10, nget
        lbu   r1, 0xFFFFFF52(r0)   ; NDATA: reading takes it
        ret
```

Messages. P32W sends a request, then reads its one reply before it sends the next; nothing else ever arrives in `NDATA`. Both begin with the same 4-byte header:

Byte	Request	Reply
0	the command	the status: 0 `OK`, 1 `AGAIN`, …
1	the socket, 0–7 (0 if none)	the socket (the new one, for `SOCKET` and `ACCEPT`)
2, 3	the payload's length, low byte first	the same
4 …	the payload, at most 1024 bytes	the same

Addresses and ports inside payloads are in network order (big-endian). The commands: `01 HELLO`, `02 IFINFO`, `03 IFCONFIG`, `04 EVENTS`, `10 SOCKET` (TCP, UDP, TLS, a DNS lookup or ICMP), `11 CONNECT`, `12 BIND`, `13 LISTEN`, `14 ACCEPT`, `15 SEND`, `16 RECV`, `17 SENDTO`, `18 RECVFROM`, `19 SHUTDOWN`, `1A CLOSE`, `1B STATE`; their

payloads, the socket states and the status codes are in net-mailbox.md. `HELLO` is `01 00 00 00`; it closes every socket, and its reply is `00 00 08 00` and the text `P32NET 1`.

Events. `CONNECT` only starts a connection. The coprocessor sets ATTN whenever something a program may be waiting for changes on a socket: connected or failed, a DNS answer, data in, room to send, a connection to accept, the peer closing. `EVENTS` replies with a bit per changed socket and clears ATTN. With IEN set, ATTN interrupts: the I/O card's status bit 8, IRQ7 by default, vector `0x78`; without, poll ATTN.

Starting: wait for READY, then send `HELLO`. A driver that loses its place writes RESET, waits for READY and starts again with `HELLO`. Without the I/O card these addresses are a bus error; with it but no running coprocessor, READY is 0. In the emulators `-N` puts the coprocessor on the Mac's network (P32W is 10.0.2.15, the Mac 10.0.2.2; listening ports below 1024 move up by 8000, which `-Nport N` changes; `-Nca FILE` gives TLS other certificates to trust); without `-N`, READY is 0.

6.10 Interrupt sources

Every source of an interrupt, with its default level and its vector at the default windows (the I/O card 7, the disk card 6, the video card 5):

Vector	Source	Status bit	Level (jumpers)	The handler clears it by
`0x70`	timer	I/O 0	0 (`L0_*`)	writing `TCTRL` with bit 1 set (§6.2)
`0x71`	UART 0	I/O 1	2 (`L1_*`)	reading RBR or IIR (§6.1)
`0x72`	UART 1	I/O 2	3 (`L2_*`)	the same
`0x73`	UART 2	I/O 3	2 (`L3_*`)	the same
`0x74`	UART 3	I/O 4	3 (`L4_*`)	the same
`0x75`	real-time clock	I/O 5	5 (`L5_*`)	reading register C (§6.5); the emulators do not model this interrupt
`0x76`	USB 0	I/O 6	6 (`L6_*`)	the CH376S's own protocol (§6.8)
`0x77`	USB 1	I/O 7	6 (`L7_*`)	the same
`0x78`	network coprocessor	I/O 8	7 (`L8_*`)	`EVENTS`, which clears ATTN, or writing IEN = 0 (§6.9)
`0x60`–`0x63`	disk ports 0–3	disk 0–3	1 (`L2 L1 L0`)	reading the port's status register (§6.4)
`0x64`	the disk card's DMA engine	disk 4	1 (the same)	writing DMACTL (§6.7)
`0x50`	PS/2 keyboard and mouse	video 0	4 (its own three jumpers)	reading `DF0C`, which takes the byte (one an interrupt: the next is some 50 µs coming, §6.6)
`0x51`	the video chip (the HD63484's IRQ)	video 1	4 (its own three)	making the flag fall, or clearing its enable in the chip's `r02` (§6.6)

The jumpers are at each card's edge, labelled: a fitted jumper is 0, an open one 1, and a level is three jumpers read as a binary number. On the I/O card every device has its own three, `Ld_2 Ld_1 Ld_0` for status bit d: USB 0's level 6 is `L6_2` and `L6_1` open, `L6_0` fitted. The disk card has one set, `L2 L1 L0`, for all its sources (level 1: `L0` open). The video card has a set for each of its two (both level 4: the high jumper open). The window is `WIN2 WIN1 WIN0`, read the same way (7: all open; 6: `WIN0` fitted; 5: `WIN1` fitted).

The vector. When the CPU takes line n, the card with a source asking on level n answers the acknowledge cycle (D57) with its window × 16 + the lowest asking status bit among its sources on that level, and `CAUSE` = `0x8000_0000 | vector << 8 | n` (§7), so a handler can dispatch on `CAUSE` bits 15:8 alone. Sources that share a level are served lowest bit first: with UART 0 and UART 2 both asking the vector is `0x71`, and once UART 0's request is cleared the line,

still held by UART 2, traps again with `0x73`. A vector of `0xFF` means no card answered: read the cards' status words (`+FF4`) instead. **Two cards must never have sources on the same level** — both would answer; one card may use several (the I/O card uses six). A status word shows its card's requests whatever their levels.

6.11 The debug card (D64)

A bus tester, not a device: the host — the Mac, over USB through the card's Pico 2 — drives it, and no program does. It sits in a slot of its own, slot 0, and works in one of two modes, chosen by a switch at its front edge, never both:

- **LISTEN:** a logic analyzer on every signal of the bus and of the CPU's local link, one sample a bus cycle (or a faster timing view). Its bus drivers are held off by wiring, so it cannot disturb a running program.
- **DRIVE:** it is a bus master. With the CPU and MMU / data cards in, it borrows the bus by DMA (`-BREQ0`, the arbiter's highest priority) one cycle at a time, between the CPU's cycles, and reads and writes memory and device registers with the CPU halted or running: the host loads programs this way. So memory can change under a running program. (It cannot write the boot ROM: the memory card ties the flash chips' write line high, and they are programmed in a chip programmer, D101.) In DRIVE the host can also pull any interrupt line low. With those two cards pulled it makes the clock and reset itself, to test a card alone; no program runs then. It cannot do that against a card that is in: each of the two ties a line to ground that turns the debug card's drivers for its lines off (D102).

In either mode the host can also **clock the machine** (D76): the CPU card then runs from the debug card's 12 MHz (a 1.5 MHz machine) or a phase at a time, one per STEP command. The timer counts bus clocks (§6.2), so time slows with the machine; the devices with clocks of their own (the UARTs' baud rate, the real-time clock, the USB modules, the coprocessor, the drives, the video card) keep real time, and a UART receiving at full speed overruns.

7. Exceptions and interrupts

One handler. Every exception and interrupt enters supervisor mode at the address in `VBR`:

1. `EPC` ← the trapping instruction (an interrupt: the next instruction);
2. `CAUSE` ← the code, `BADADDR` ← the faulting address (or 0);
3. `PS` ← S, `PIE` ← IE; S ← 1, IE ← 0 (and `r15` becomes the supervisor SP);
4. PC ← `VBR`.

Entry writes no memory and no register, so it cannot fault. **The trapping instruction has had no effect** — return with `RTE` to re-execute it (after fixing the cause, e.g. a TLB refill), or add 4 to `EPC` first to skip it (`SYSCALL` and `BREAK` leave `EPC` pointing at themselves).

Code	Cause	`BADADDR`
1	ILLEGAL — undefined opcode or non-zero reserved bits	0
2	PRIVILEGE — privileged instruction in user mode, or a user access to ≥ `0x8000_0000`	0 / the address
3	MISALIGNED — a load/store or jump not aligned	the address
4	BUSERROR — nothing answered (or wrong access size in the I/O page, or a ROM write)	the **physical** address
5	SYSCALL	0
6	BREAK	0
8 / 9 / 10	TLB miss on fetch / load / store	the virtual address
11	PROTECT — store to a read-only page, fetch from a non-executable one	the virtual address

Code	Cause	BADADDR
`0x8000_0000` | v << 8 | n	interrupt on line n; v = the card's vector (its window × 16 + the device's interrupt status bit; `0xFF`: none answered, D57)	0

Interrupts (8 levels, set by jumpers; the defaults: 0 timer, 1 disk, 2 UARTs 0 and 2, 3 UARTs 1 and 3, 4 PS/2, 5 real-time clock, 6 USB, 7 network; every source and its vector: §6.10) are level-triggered and taken **between instructions** when `STATUS.IE` = 1 and `IPEND & IMASK` ≠ 0, lowest line first. The handler must clear the device's request, or it will trap again at once. **Interrupt shadow:** after an `MTCR` (e.g. setting `IE`) or an `RTE`, one more instruction runs before a pending interrupt is taken — so `RTE` always lets the interrupted program make progress.

A handler skeleton:

```
        la    r2, handler
        mtcr  VBR, r2
handler:                        ; supervisor SP in effect
        push  r10
        push  r11
        mfcr  r10, CAUSE
        blt   r10, r0, irq      ; bit 31 set: an interrupt
        ...                     ; an exception: CAUSE, BADADDR, EPC
irq:    andi  r10, r10, 7       ; the line
        ...                     ; acknowledge the device
        pop   r11
        pop   r10
        rte
```

A handler that may itself trap, or that re-enables interrupts, must first save `EPC` and `STATUS` (they are overwritten by the next trap). `WAIT` idles until an enabled interrupt line is asserted — the kernel's idle loop.

8. User mode and virtual memory

User programs run with `STATUS.S` = 0: privileged instructions and the I/O page trap, and **every address is translated**. User addresses `0x0000_0000`–`0x7FFF_FFFF` go through the TLB; anything above traps (PRIVILEGE). Supervisor addresses are physical.

Pages are 4 KiB. The **TLB** is split: 64 entries for instruction fetches (I-TLB), 64 for loads and stores (D-TLB), each **direct-mapped** — the entry for virtual page number VPN is entry `VPN mod 64` — and **tagged with an ASID**, so switching processes is just `MTCR ASID` (no flush).

The TLB is managed by software. On a miss the hardware has already put the faulting VPN, the side (I) and the current ASID into `TLBHI`; the handler looks the page up in its own page table, writes `TLBLO`, executes `TLBW`, and returns with `RTE` to retry the access:

```
tlbmiss:                        ; CAUSE 8, 9 or 10
        mfcr  r10, TLBHI        ; VPN | I | ASID, loaded by the hardware
        srli  r10, r10, 12      ; the VPN
        slli  r10, r10, 2       ; index into a linear table of TLBLO words
        la    r11, pagetable    ; (this process's table)
        add   r10, r10, r11
        lw    r10, 0(r10)       ; PFN | X W V
        beqz  r10, segfault     ; not mapped: the process dies
        mtcr  TLBLO, r10
        tlbw                    ; write the entry TLBHI selects
        rte                     ; retry the instruction
```

`TLBLO` = `PFN << 12 | X << 2 | W << 1 | V`. A store needs W, a fetch X; V = 0 entries never match. See `emulator/test/vmkernel.inc` for a complete kernel (processes in separate address spaces, preemption by the timer, system calls that check user pointers).

9. Tools

Tool	
`assembler/p32asm.py src.asm -o img.bin -l listing.lst`	two-pass assembler; flat ROM image at 0
`emulator/p32emu [-t] [-d disk] [-a aux] [-p n] img.bin`	instruction-level emulator (the reference)
`emulator/p32uemu ...`	microcode-level emulator: runs the real control store, cycle by cycle
`make -C emulator test`	every test: both emulators must agree instruction by instruction

The console is the emulators' stdin/stdout; UART 1 goes to files (`-a`, `-A`); `HALT` ends the run with exit status `r1 & 0xFF`. With `VBR` = 0 an exception stops the emulators with a register dump (on the hardware it would jump to 0). `-t` traces every instruction, trap and interrupt.

The boot sequence

docs/boot.md

Status: DESCRIBES THE SYSTEM AS BUILT (2026-10-26): the CPU card's reset logic (`generators/card_cpu.py`), the microcode (`microcode/p32ucode.py`), the boot ROM monitor (`software/monitor/monitor.asm`), the disk layout (`tools/p32fs.py`) and the kernel (`kernel/`). Each step names the file that does it.

From power-on to `login:` the machine passes through six stages. Each one hands a small, well-defined state to the next:

Stage	Runs	Hands on
1. Power and reset	the CPU card's GAL PANEL	`-RES` released: every card in its reset state
2. The reset microcode	the CPU card's sequencer	r0 = 0, PC = 0, supervisor mode, interrupts off
3. The monitor	the boot ROM at `0x0000_0000`	the console at 9600 baud; the `b` command
4. The kernel loader	the monitor's `b`	the kernel in RAM at `0x0010_0000`, called
5. The kernel's start-up	`kernel/start.s`, `kernel/main.c`	devices, memory, the root filesystem, process 1
6. User space	`/etc/init`, `/etc/rc`, `getty`, `login`	a shell for each user

1. Power and reset

What starts it. Two things reset the machine:

- `PWROK` (J2 A6), *power good*, from the backplane's supply supervisor (a TL7705A, D78). It stays low until +5 V has been above 4.55 V for about 130 ms, and falls when +5 V dips below that.
- **The RESET button** on the CPU card's front panel (debounced by a 74HCT14).

GAL PANEL on the CPU card is clocked by the 24 MHz oscillator. It registers `RESET = RESBTN or not PWROK` and drives the inverse onto the backplane as `-RES` (J2 A5). Every card sees the same reset, and while it is low:

- The CPU card's registered GALs (STATUS, IMASK, COND and the others) are held in their reset state through their asynchronous reset inputs. STATUS resets to supervisor mode with interrupts off, IMASK to 0 (every level masked), VBR to 0.
- GAL PANEL forces the sequencer's next microaddress to the **RESET internal sequence**. While RESET is high, every commit edge loads that same first microword again.
- The ALU module (a Tang Nano 20K) gets its own reset through a 74LVC244 (`NANO_RES_N`).
- The MMU / data card clears its state: the TLB-write pending flag, the bus lock, the arbiter's grants.
- The device cards reset their devices. On the I/O card (its netlist is in progress) the timer stops (enable 0, pending 0), the UARTs reset (MR) and the network coprocessor's RUN pin is held low.

When the PANEL GAL sees `PWROK` high and the button released, it drops RESET at the next oscillator edge, so `-RES` rises cleanly, in step with the clock.

2. The reset microcode

The first two microwords after reset are the **RESET internal sequence** (`p32ucode.py`, `ISEQ["RESET"]`):

Step	Microword	Effect
0	R = 0 → register r0	r0 is a real register in the register file's SRAM, and its contents after power-on are random. The microcode writes it to 0 here, and no instruction can write it afterwards (D4).
1	R = 0 → PC and MAR, END	The PC is 0, and so is the MMU / data card's MAR (its shadow PC loads with it). END starts the first fetch.

The machine is now in the state `isa-system.md` §1 defines: **supervisor mode** (`STATUS.S = 1`), **interrupts off** (`IE = 0`, `IMASK = 0`), `VBR = 0`, and executing at **physical address 0**. Supervisor addresses are not translated (isa-system.md C2), so the TLB's random contents after power-on do not matter yet.

3. The monitor (the boot ROM)

Address 0 is the **boot ROM**: four SST39SF020A flash chips on the memory card in window 0, the one with its ROM jumper open (D52, D63), one chip per byte lane. The ROM holds the monitor, `software/monitor/monitor.asm`, assembled and split into its four lanes. Its first instructions (`reset:`):

1. `sp = 0x001F_FC00`: the stack, growing down from just below the monitor's 128-byte line buffer at the top of the first 2 MiB bank. Every machine has that bank (D60), whatever else is fitted.
2. `VBR = exception`: from now on any exception (a bad address, an illegal instruction in a program the monitor called) lands in the monitor's own handler. The handler prints `CAUSE`, `EPC` and `BADADDR` and returns to the prompt.
3. UART 0 to 9600 baud, 8 data bits, no parity, 1 stop bit (divisor 12 from the 1.8432 MHz clock), FIFOs on, interrupts off (`uart_init` in `software/lib/p32io.inc`).
4. The banner, `P32W monitor -- ? for help`, and the prompt `> `.

The monitor polls the UART; it never enables interrupts. Its commands are described in the monitor. To boot, the user types `b`.

4. The kernel loader: the monitor's `b`

The disk. The boot device is the CompactFlash card in the disk card's port 0 (window 6, `0xFFFF_EF00`; D50, D52). `tools/p32fs.py` lays it out:

Sectors (512 bytes)	Contents
0	**the boot header**: five little-endian words
1 … 2047	**the boot area** (1 MiB): the kernel image, `kernel/unix.bin`
2048 …	**the filesystem** (its first sector is the header's word 4)

Header word	Meaning	Today's value
0	magic `"P32B"` (`0x4232_3350`)	
1	the kernel's size in sectors	`0xD1` (209 sectors, about 105 KiB)
2	the load address	`0x0010_0000`
3	the entry point	`0x0010_0000` (`_start`)
4	the filesystem's first sector	`0x800` (2048)

What `b` does (`cmd_boot`):

1. `cf_init`. Is a card there? The status register's DRDY bit must be set. A socket with nothing in it reads 0, because the adapter pulls the bus down. If no card answers, the monitor prints `no disk`. If one does, it sends SET FEATURES 81h, which turns off any 8-bit transfer mode earlier software left behind. The data register is 16 bits wide (D26).
2. Sector 0 is read into RAM just below the line buffer, at `0x001F_F800`. If its first word is not `P32B`: `not a boot disk`.
3. `disk_load`: word 1's count of sectors, starting at sector 1, into memory from word 2's address, one READ SECTOR command per sector. Each sector is 256 halfword reads of the data register, polling BSY and DRQ in between. Then it prints `loaded 000000D1 sectors at 00100000` and `entry 00100000`.
4. `callr` to word 3's entry point. It is a *call*, so a program that returns lands back in the monitor, which prints `returned` and r1. The kernel never returns. Its `reboot` system call clears STATUS, IMASK and VBR and jumps to address 0 instead, which restarts the monitor as a reset would, except that the hardware is not reset.

At this point the CPU is still in supervisor mode with interrupts off, and the stack pointer is the monitor's.

5. The kernel's start-up

5.1 `_start` (`kernel/start.s`)

1. `sp` = the kernel's own 4 KiB boot stack (`bootstack_top`, in its .bss).
2. The .bss, `_bss` to `_end`, is cleared. The monitor loaded only the text and data, and RAM after power-on holds anything.
3. `STATUS = 1` (supervisor, interrupts off), `IMASK = 0`, `VBR = trapvec`. From here every trap and interrupt enters the kernel's handler, not the monitor's.
4. `call kmain`.

The kernel is linked at `0x0010_0000` (`kernel/Makefile`: `--text 0x00100000`) and runs **untranslated**: in supervisor mode an address is a physical address, so the kernel needs no page table of its own.

5.2 `kmain` (`kernel/main.c`), in order

Call	What it does
`video_init()`	If the video card answers (its ID read with a probe, D53), the screen becomes a second console.
`kprintf("P32W Unix (M7)")`	The first line on the console (UART 0, polled).
`kinit()`	**Memory** (`vm.c`): probes every 2 MiB bank from 2 MiB to 64 MiB. A bank is there if a test pattern written at its offset `0x100` reads back and no BUSERROR came; a missing bank's access is a BUSERROR, which `trap()` skips while `probing` is set. Every free 4 KiB frame above the kernel's `_end`, in the banks found, goes on the free list. Frames in missing banks are marked in use for good. Holes are allowed (D60).
`procinit()`	The 64 process slots. Slot k gets ASID k + 1.
`binit()`	The buffer cache: 48 one-KiB buffers on an LRU ring.
`fileinit()`	The open-file table (128 entries).
`cf_init()`	**The disk** (`cf.c`): the boot card must answer, or `panic("no disk")`. Reads the boot header's word 4 (the filesystem's first sector) and probes the second CF socket and the two IDE channels for more drives.
`fsinit(ROOTDEV)`	**The root filesystem** (`fs.c`): reads the superblock (block 0 of the filesystem area), checks its magic. If its *clean* flag is 0 it warns `the filesystem was not unmounted cleanly (run fsck)`. Then it writes the flag as 0 (in use) until a clean shutdown sets it again.

Call	What it does
`kprintf("memory: … filesystem: …")`	RAM found, frames free, blocks free.
`rtc_init()`	**The real-time clock** (D23): if the DS12C887 answers, the time of day comes from it (`boottime`). Without it the system starts at 1 January 1970.
`usb_init()`, `net_init()`	The I/O card's two CH376S USB controllers and the network coprocessor, if they answer.
`ttyinit()`	The four UARTs at 9600 baud 8N1 with receive interrupts on, and IMASK levels 2 and 3 enabled (UARTs 0 and 2 share level 2, 1 and 3 share level 3).
timer	`TRELOAD = TCOUNT = 20000`, `TCTRL = 1`: the I/O card's 1 MHz timer interrupts 50 times a second (`HZ`, `param.h`); IMASK level 0 enabled.
`tlbflush()`	Both halves of the TLB invalidated. Its contents after power-on are random, and user mode is about to use it.
`userinit()`	**Process 1** (below).
`scheduler()`	Never returns. The kernel is now a scheduler that runs processes.

Interrupts are still off (STATUS.IE = 0) through all of this. The scheduler's loop is the first code to turn them on.

5.3 Process 1 (`userinit` in `kernel/proc.c`)

Process 1 gets an address space with two pages:

- `0x1000`: a nine-instruction program (`initcode`) that is `execve("/etc/init", { "/etc/init", 0 }, 0)`, then `exit(1)` if the exec fails;
- `0x7F_F000`, the page below `USERTOP` (8 MiB, D34): its stack.

Its trap frame is set so that the first "return from trap" (`RTE`) enters **user mode at `0x1000` with interrupts on** (`status = S | PIE`: the current mode supervisor, the mode `RTE` returns to user, interrupts enabled after it). Its kernel context starts in `trapret`. Its current directory is `/`, and its user and group are 0 (root).

The scheduler finds it RUNNABLE, loads its ASID and page table pointer (`switchuvm`) and switches to it (`swtch`). `trapret` restores the trap frame, and `RTE` runs `initcode`. Its `SYSCALL` for `execve` is the first system call. The kernel reads `/etc/init` from the filesystem, builds a new address space (text at `0x1000`, data, bss, a stack holding `argc` and `argv`), and the same `RTE` path starts `init`'s `main`.

6. User space

6.1 `/etc/init` (`software/cmd/init.c`)

1. Opens `/dev/tty0` (the console, UART 0) as its standard input, output and error. If `/dev/tty4` opens, the video card is fitted and its screen and keyboard get a login too.
2. Runs `/bin/sh /etc/rc` and waits for it:

```
/etc/fsck -q -c          a full check only if the filesystem was not clean (D40)
: > /etc/mtab            nothing but / is mounted
: > /etc/utmp            nobody is logged in
/etc/cron
/etc/ntpdate             only if /etc/ntp.conf exists: the clock from the network (D46)
/etc/telnetd             remote logins, if there is a network
echo P32W Unix: multi-user
```

3. If there is no `/etc/getty`, it runs single-user: a shell on the console, started again whenever it exits.

4. Otherwise it starts **one `getty` per terminal**: `tty0`, `tty1` and `tty4` when the video card is fitted. Each getty runs in a new session (`setsid`), so the terminal it opens becomes its controlling terminal.
5. It then waits forever. When a getty's session ends, init clears that terminal's `/etc/utmp` record and starts a new getty. A getty that ends within five seconds five times in a row waits a minute before the next (a terminal that keeps failing does not spin). Orphaned processes are given to init, which reaps them. `SIGTERM` (from `shutdown`) stops the respawning.

6.2 `getty` and `login`

- `/etc/getty ttyN` opens `/dev/ttyN`, sets it to echo with CR/LF mapping, erase ^H and kill ^U, prints `P32W Unix (ttyN)` and `login:`, reads a name and execs `/bin/login name ttyN`.
- `/bin/login` asks for the password with echo off (three tries, then back to getty), checks it against `/etc/passwd`, and records the login in `/etc/utmp`. It gives the user the terminal (owner, group `tty`, mode 0620), prints `/etc/motd` and `You have mail.` when the mailbox is not empty, sets the user and group IDs, `HOME`, `USER`, `LOGNAME`, `SHELL` and `PATH=/bin:/usr/bin`, changes to the home directory and execs the user's shell. The shell reads `/etc/profile`.

The machine is up.

7. Shutting down and rebooting

- `sync` (and the kernel itself, every 30 seconds) writes the buffer cache's delayed blocks to the disk.
- `shutdown` warns the logged-in users (`wall`), sends `SIGTERM` to init (no new logins), then to every other process, waits two seconds and sends `SIGKILL`. `halt` and `reboot` call the `reboot` system call (root only). It writes every delayed block and the superblocks with the *clean* flag set (`fs_clean`). Then it either executes `HALT`, or stops the clock, masks every interrupt and jumps to the monitor at address 0 with STATUS and VBR cleared, as after a reset button press (`reset` in `start.s`). A machine that loses power without this leaves the flag 0, and the next boot's `fsck -c` checks the whole filesystem.

8. In the emulators

The emulators start exactly as the hardware does: `p32emu` and `p32uemu` take the ROM image (`monitor.bin`) and begin at address 0 in supervisor mode. `-d IMAGE` is the CompactFlash card:

```
make -C emulator
make -C software disk
python3 assembler/p32asm.py software/monitor/monitor.asm -o monitor.bin
emulator/p32emu -d software/build/disk.img monitor.bin
```

At the monitor's `>` type `b`. On the real machine, write `software/build/disk.img` to a CF card (`dd … bs=1m`), power on, `b`.

The boot ROM monitor

docs/monitor.md

Status: DESCRIBES THE PROGRAM AS BUILT (2026-10-26): `software/monitor/monitor.asm` and the routines it includes from `software/lib/p32io.inc`. Tested by `make test` (section 4f) on both emulators against the recorded session in `emulator/test/monitor.expect`.

The monitor is the first program P32W runs: it is what the boot ROM holds, so the CPU starts it at address 0 after every reset (see the boot sequence). It is deliberately small (under 2 KiB, 1,960 bytes today) and does only what is needed to look at a machine that has just been built and to start a bigger program:

- talk on the console (UART 0, 9600 baud, 8N1);
- show and change memory a word at a time;
- call a program in memory;
- load sectors from the CompactFlash card, and boot from it;
- catch any exception and report it, instead of losing control.

It uses no interrupts, no MMU and no RAM beyond its own stack and line buffer, so it works on a machine whose devices, interrupts or virtual memory are not working yet: bring-up starts here.

1. Where it lives

In the ROM. The monitor's image is the boot ROM's contents: four SST39SF020A flash chips on the memory card in window 0 (the card with its ROM jumper open, D63), one chip per byte lane. Byte `k` of the image goes into lane `k mod 4`'s chip at address `k / 4`. Programming them: split `monitor.bin` into four lane files and write each chip in a programmer, or (planned) let the debug card's host link write them in place with the PROG jumper fitted.

In RAM, while it runs:

Address	Use
`0x0000_0000` … `0x0000_07A7`	the monitor's code and strings (ROM)
`0x001F_FC00` … `0x001F_FC7F`	the line buffer: the command being typed (128 bytes)
below `0x001F_FC00`	the monitor's stack, growing down
`0x001F_F800` … `0x001F_F9FF`	`b`'s copy of the boot header sector
`0x0010_0000` … `0x001F_7FFF`	free for programs (the usual load address is `0x0010_0000`)

Everything sits in the first 2 MiB bank, which every machine has (D60). Before D60 the stack was at 9 MiB, and a machine with less memory could not run the monitor. The ROM hides the first MiB of the RAM under it, so RAM starts at `0x0010_0000`.

2. Using it

The monitor prints

```
P32W monitor -- ? for help
>
```

and reads a line: echoed as typed, Backspace (^H) or Delete removes a character, Return ends the line, and anything past 126 characters is ignored. **Numbers are hexadecimal**, without `0x`; upper or lower case. Spaces separate them. An unknown command prints `?`, a missing number `syntax? (? for help)`.

Command	What it does
`d ADDR [LEN]`	**Dump** memory as words, four to a line, from `ADDR` rounded down to a multiple of 4; `LEN` bytes (default 16). Each word is a 32-bit load, so a dump of the I/O windows really reads the device registers.
`m ADDR VALUE`	**Store** the word `VALUE` at `ADDR` (rounded down to a multiple of 4) and print `ADDR = VALUE`.
`g ADDR`	**Call** the program at `ADDR` (below). When it returns, print `returned` and its r1.
`l LBA COUNT ADDR`	**Load** `COUNT` 512-byte sectors from the CompactFlash card, starting at sector `LBA`, into memory at `ADDR` (which must be a multiple of 2). Prints `loaded COUNT sectors at ADDR`.
`b`	**Boot** from the card (§3).
`?`	Help: the command list.
`q`	`HALT` with r1 = 0. In an emulator this ends the run; on the machine the CPU card's HALT LED lights and the clock stops until RESET.

A session (the test's own, `emulator/test/monitor.expect`), booting a one-sector program that prints `BOOTED` and returns `0x55`:

```
P32W monitor -- ? for help
> b
loaded 00000001 sectors at 00100000
entry 00100000
BOOTED
returned 00000055
> d 100000 8
00100000: 20200042 4820FF00
> g 100000
BOOTED
returned 00000055
> l 0 1 200000
loaded 00000001 sectors at 00200000
> d 200000 10
00200000: 42323350 00000001 00100000 00100000
> q
```

The last dump is the card's sector 0, the boot header: `"P32B"`, one sector, load address and entry `0x0010_0000`.

3. Booting: `b`

The card's sector 0 is the **boot header**, five little-endian words:

Word	Meaning
0	magic `0x4232_3350`, the bytes `P32B`
1	the number of sectors to load
2	where to load them
3	where to start
4	(P32W Unix's disks: the filesystem's first sector; the monitor ignores it)

`b` checks for a card (`no disk` if the CF socket is empty) and reads sector 0. If word 0 is not the magic it prints `not a boot disk`. Otherwise it loads word 1's sectors from sector 1 onwards to word 2's address, prints `loaded …` and `entry …`, and **calls** word 3's address. A disk error at any point prints `disk error` and the card's status byte, then returns to the

prompt.

To make a bootable card from any program:

```
python3 emulator/test/mkbootdisk.py program.bin disk.img [LOAD [ENTRY]]
```

(load and entry default to `100000`). P32W Unix's disk is made by `tools/p32fs.py`, which puts the kernel in the boot area the same way.

4. Programs the monitor calls

`g` and `b` both use `CALLR`. The program starts with:

- **supervisor mode, interrupts off** (STATUS as after reset), IMASK 0;
- `sp` = **the monitor's stack**, with the return address on top (P32W's `CALL` pushes it, D6). A program that returns with `RET` comes back to the monitor, which prints its r1. It may use the stack freely below that.
- `VBR` = **the monitor's exception handler**, until the program sets its own (the kernel does, at once).
- the console UART initialised (9600 8N1, FIFOs on, interrupts off).

Registers are not preserved for it: the monitor needs nothing back except `sp` (restored with the return) and r1. A program that never returns (the kernel) simply takes over the machine.

5. Exceptions

The monitor sets `VBR` to its handler before anything else. Any exception while it, or a program it called, is running (an address nothing answers, which is a BUSERROR; a misaligned word; an illegal instruction; `BREAK`) ends up there. The handler resets the stack, prints

```
exception, CAUSE 00000004 EPC 00100010 BADADDR 0FFFFFF0
```

(`isa-system.md` lists the causes: 4 is a bus error) and returns to the prompt. A program that crashes therefore leaves the monitor running, and its memory can be dumped afterwards. Interrupts stay off throughout, so an interrupt cannot reach the handler.

6. How it works

The monitor is one assembly file of about 330 lines, plus the shared device routines it includes:

- **The command loop** (`prompt`): reset `sp`, print `> `, `readline` into the buffer, skip spaces, dispatch on the first character. Each command parses its numbers with `gethex` (r1 the value, r2 the number of digits: 0 means none), does its work and jumps back to `prompt`. Jumping back, rather than returning, resets the stack every time, so a command that fails part way leaves nothing behind.
- **The console** (`p32io.inc`): `uart_init`, `uart_putc`, `uart_getc`, `uart_puts`, `uart_puthex`. All of them poll the line status register (LSR bit 5 THR empty, bit 0 data ready) and clobber only r10–r14, by the programmer's guide's convention: arguments in r2–r5, the result in r1.
- **The disk** (`p32io.inc`): `cf_init` (is a card there; turn off 8-bit mode) and `cf_read` (one sector: the LBA into the task file, READ SECTOR 20h, wait for DRQ, 256 halfword reads of the 16-bit data register, D26; wait for BSY to clear, check ERR). `disk_load` loops over the sectors.

- **Devices it touches:** UART 0 (`0xFFFF_FF00`) and the disk card's port 0 (`0xFFFF_EF00`) at their default windows (D52). It needs no other card, and no interrupt, timer or MMU.

Building and trying it:

```
python3 assembler/p32asm.py software/monitor/monitor.asm -o monitor.bin
emulator/p32emu -d software/build/disk.img monitor.bin
```

Possible later additions (not built): a write-sector command, a download over the serial line (Kermit or S-records) for a machine with no card yet, an autoboot after a few seconds without a key.

software/

software/README.md

Programs for the P32W itself.

cmd/	**the P32W Unix commands** (C): `init`, `sh`, `ls`, `cat`, `cp`, `mv`, `rm`, `ln`, `mkdir`, `rmdir`, `pwd`, `grep`, `wc`, `sort`, `uniq`, `head`, `tail`, `tee`, `ps`, `kill`, `chmod`, `chown`, `date`, `stty`, `halt`, ...
libc/, include/	**the C library** (M2, D22): strings, ctype, malloc, stdio / printf, stdlib, setjmp, system calls, signals, directories, time
sa/crt0_sa.s	the stand-alone runtime: C programs on the bare machine (the compiler's tests)
root.manifest	the root filesystem: what `make disk` puts where
tests/	scripted P32W Unix sessions (`run_session.sh`, in `make test`)
monitor/monitor.asm	**the boot ROM monitor** — the first program at reset: console on UART 0 (9600 8N1); dump and modify memory, call a program, load sectors from the CompactFlash card, boot from it; catches exceptions and reports them
lib/p32io.inc	UART, CompactFlash and port routines (the programmer's guide's)

`make` builds the library and the commands; `make disk` builds `build/disk.img`: the kernel (`kernel/`) in the boot area and the root filesystem (`tools/p32fs.py build`). Boot it with the monitor's `b`.

The monitor and the library are tested by `make test` on both emulators: `emulator/test/lib_test.asm` exercises the library; a scripted monitor session boots a program from a disk image and must match `emulator/test/monitor.expect` exactly.

The monitor

```
python3 assembler/p32asm.py software/monitor/monitor.asm -o monitor.bin
python3 emulator/test/mkbootdisk.py program.bin disk.img     # a bootable disk
emulator/p32emu -d disk.img monitor.bin                     # try it
```

Command	
`d ADDR [LEN]`	dump memory as words (hex; default 16 bytes)
`m ADDR VALUE`	store a word
`g ADDR`	call a program; it may `RET` to the monitor, which shows its `r1`
`l LBA COUNT ADDR`	load sectors from the card
`b`	boot: sector 0 holds `"P32B"`, the sector count, the load address and the entry point; sectors 1.. hold the program
`?` / `q`	help / `HALT`

An exception (a bad address, an illegal instruction in a called program) prints `CAUSE`, `EPC` and `BADADDR` and returns to the prompt: the monitor sets `VBR` to its own handler. It keeps its line buffer and stack at `0x001F_FC00` down, at the top of the first 2 MiB bank (which every machine has, D60), so programs have `0x0010_0000`–`0x001F_7FFF`.

On the hardware the monitor's image is what goes into the boot ROM: split into byte lanes for the four flash chips (or written in place by the debug card's host link with the PROG jumper fitted).

assembler/

assembler/README.md

`p32asm.py` — the P32W two-pass assembler (ISA v0, ../docs/isa-v0.md).

Mnemonics, opcodes and formats come from `../microcode/p32isa.py`, so the assembler and the emulator cannot disagree about an encoding. Edit the instruction set there, not here.

```
python3 p32asm.py src.asm [-o out.bin] [-l listing.txt]
```

Output is a flat little-endian image starting at address 0 (the ROM). The syntax, directives and pseudo-instructions (`LI`, `LA`, `MOV`, `NOT`, `PUSH`/`POP`, `BGT`/`BLE`, `BEQZ`/`BNEZ`, …) are listed in the module docstring. `.include "file"` splices in another source file (path relative to the including file); `MFCR`/`MTCR` take control-register names (`STATUS`, `EPC`, `CAUSE`, `BADADDR`, `VBR`, `USP`).

microcode/

microcode/README.md

File	Role
`p32isa.py`	the ISA's single source of truth: opcodes, formats, privileged set, control registers, exception causes
`p32ucode.py`	the microcode generator (docs/microarch.md, D14): microword fields, the microcode of every instruction and of reset / trap entry / interrupt entry, the commit-after-access check, the control-store images and `emulator/p32uc.h`

```
python3 p32ucode.py --stats             # check, build the images, print cycles per instruction
```

Outputs (build artifacts, not committed; `make` in `emulator/` builds them): `p32ucode.bin` (16384 words x 8 bytes, the emulator's image) and `u0.bin`-`u5.bin` (one image per flash chip).

The generator refuses to build if any instruction writes programmer-visible state before its last possibly-faulting memory access (precise exceptions, microarch.md section 5).