# P32W Unix

Using it, and how it works inside: the kernel, boot, the filesystem, devices, networking

P32W — Ken Rother's 32-bit TTL computer

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

# Contents


## Using P32W Unix `docs/unix.md`

- Running it
- Logging in
- The shell
- Commands
- Moving files with Kermit
- Later, and regularly
- More filesystems
- Debugging
- Programming tools (M7)
- Writing programs
- The system's sources and make
- The manual
- The date and time
- Devices

## 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

## P32W Unix internals `docs/kernel.md`

1. Where things are
2. Memory
3. Traps and interrupts
4. Processes
5. System calls
6. Files
7. Terminals (`tty.c`)
8. Other drivers

## The P32W Unix filesystem `docs/filesystem.md`

1. The disk
2. The layout inside the filesystem
3. Inodes
4. Directories
5. How the kernel uses it
6. Crash safety
7. Building and inspecting images on the Mac

8. Devices

## P32W Unix — design proposal `docs/os-design.md`

1. What the machine already gives us
2. The plan in one paragraph
3. Toolchain
4. The kernel
5. The filesystem
6. Users and security
7. The commands
8. Emulator support
9. How it is tested
10. Effort and phases
11. For review (D22)

## P32W Unix kernel `kernel/README.md`

Files

How it works

Still to come (M5–M6)

## p32cc — the P32W C compiler `cc/README.md`

The language

How it works

Tests

## USB thumb drive — design study (2026-10-16) `docs/usb-design.md`

1. The controller
2. Hardware
3. Software
4. Tests
5. Questions for Ken

## Internet over Ethernet with the W5500 — design study (2026-10-17) `docs/net-design.md`

1. What the W5500 is
2. The numbers that decide it
3. Three ways to build it
4. The emulator first, as with USB
5. P32W Unix's side (the same for A, B and C)
6. The hardware, wherever it goes
7. Work, in order (once the questions are answered)
8. Questions for Ken

## The network coprocessor's mailbox (D45) `docs/net-mailbox.md`

1. Registers
2. Messages
3. Commands
4. Events and waiting
5. Status codes
6. The emulators' model (`-N`)

# Using P32W Unix

docs/unix.md

P32W Unix is the P32W's own multi-user operating system (D22; the design is os-design.md, the kernel kernel/README.md). It is a small Unix in the style of V6/V7: several users at once, one on each serial port and one at the video console, each process in its own protected address space, a Unix filesystem on the CompactFlash card, and the classic commands.

## Running it

```
make -C emulator                      # the emulators
make -C software disk                 # the C library, the commands, the kernel, software/build/disk.img
python3 assembler/p32asm.py software/monitor/monitor.asm -o monitor.bin
emulator/p32emu -d software/build/disk.img monitor.bin
```

`tools/p32w` does the last two lines for you (and builds what is missing). At the monitor's `>` prompt type `b`: the monitor loads the kernel from the card and the kernel boots. Your terminal is UART 0 (the console); **^]** stops the emulator. When idle, the emulated machine waits in real time.

**A second user** gets UART 1 over TCP:

```
emulator/p32emu -T 2323 -d software/build/disk.img monitor.bin
python3 tools/p32term.py              # in another window: login: ...
```

`tools/p32term.py` sends each key at once, as the console does (so `q` in `top`, `vi` and ^C work) and leaves with ^]. Plain `nc localhost 2323` also connects, but the Mac then holds each line until Return, and ^C ends `nc`. (For scripts: `-A FILE` feeds UART 1 and `-a FILE` records it; `-w` feeds input a line at a time, each after a prompt — `software/tests/run_session.sh`.)

On the real machine: write `software/build/disk.img` to a CF card (`dd if=software/build/disk.img of=/dev/diskN bs=1m`), boot, `b`.

## Logging in

| User | Password | Home |
|---|---|---|
| `root` | `root` | `/` |
| `ken` | `p32` | `/usr/ken` |
| `guest` | (none) | `/usr/guest` |

Change a password with `passwd`. `su` becomes root (with root's password). `who` lists who is logged in; `id` / `whoami` show who you are. Log out with `exit` (or ^D); `init` starts a new `login:` on that terminal.

**Shut down** as root with `shutdown [-r] [now | +minutes] [message]`: it warns both terminals (each minute, with the message), stops the logins, ends every process, writes everything to the disk and marks the filesystem clean, then halts — or with `-r` restarts the machine into the monitor (type `b` to boot again). `halt` and `reboot` do the same at once, without the warnings. (Switching off without it is like any Unix without sync: the next boot warns that the filesystem was not unmounted cleanly.)

## The shell

`sh` reads a command per line:

| | |
|---|---|
| `cmd arg ...` | run a command (found on `$PATH`: `/bin`, `/usr/bin`) |
| `cmd < in > out`, `>> out` | redirection |
| `a | b | c` | a pipeline |
| `a ; b`, `a &` | a list; run in the background (the shell prints its pid) |
| `'...'`, `"..."`, `\c` | quoting (`$` is expanded inside `"..."`) |
| `$NAME`, `${NAME}`, `$?`, `$$`, `$!` | variables, the last status, the shell's pid, the last background pid |
| `$0`–`$9`, `$#`, `$*` | a script's name and arguments (`sh script args`) |
| `NAME=value` | set a variable (all variables are passed to commands) |
| `` `command` `` | its output, in place (split into words unless in `"..."`) |
| `* ? [...]` | wildcards (in the last part of a path) |
| `if list; then list; elif list; then list; else list; fi` | conditions: a command's exit status (`test` / `[`) |
| `while list; do list; done`, `for NAME in words; do list; done` | loops; `break`, `continue` |
| `cd dir`, `exit [n]`, `unset NAME`, `:` | built-ins |

More, as in the Bourne shell (`man sh`): functions (`name() { list; }`, `return`), `case word in pattern) list ;; esac`, here-documents (`<<END`), subshells `( list )` and groups `{ list; }`, redirection of any descriptor (`2> err`, `2>&1`, `>&2`), `set`, `shift`, `read`, `eval`, `.`, `exec`, `wait`, `umask`, `NAME=value cmd` for one command, `#!` scripts, and `/etc/profile` plus `$HOME/.profile` for login shells.

A construct may span lines: the shell prompts `> ` until it is complete. `test` (also `[ ... ]`) checks files (`-f -d -e -r -w -x -s`), strings (`-z -n = !=`) and numbers (`-eq -ne -lt -le -gt -ge`); `expr` does arithmetic and comparisons ( `n=`expr $n + 1`` ).

^C interrupts the running command, ^\ quits it, ^U erases the line, backspace / DEL erase a character, ^D ends input.

**Editing and history** (at a terminal):

| Key or command | What it does |
|---|---|
| ← → (or ^B ^F) | move along the line; typing inserts at the cursor |
| ^A, ^E (Home, End) | go to the start / end of the line |
| backspace, ^D, Delete | delete before / at the cursor (^D on an empty line ends input) |
| ^K, ^U, ^W | delete to the end, the whole line, the word before the cursor |
| ↑ ↓ (or ^P ^N) | step through earlier commands |
| ^L | redraw the line; ^C abandons it |
| Tab | complete a command name (first word) or a file name; Tab again lists the choices |
| `history` | list the last 100 commands, numbered (works in pipelines: `history \| grep cc`) |
| `!!`, `!n`, `!-n`, `!text` | run the last command again, number n, the n-th last, the last starting with `text` (the expanded line is shown first) |

History is kept per user in `$HOME/.sh_history`, so it survives logging out. A `!` followed by a space or `=` (as in `test a != b`) is left alone, as is anything inside `'...'`.

## Commands

`ls [-1ACFRSadhilnrst]` `cat` `cp` `mv` `rm [-rf]` `ln` `mkdir` `rmdir` `pwd` `touch` `chmod` `chown` `find` `du` `ed` `grep [-vnci]` `wc [-lwc]` `sort [-rn]` `uniq [-c]` `head [-n]` `tail [-n]` `tee` `more` `od [-c]` `cmp` `echo` `basename` `dirname` `test` / `[` `expr` `fsck` `ps` `kill [-sig]` `sleep` `date` `stty` `uname [-a]` `who` `id` `whoami` `su` `passwd` `login` `sync` `true` `false`; `ps [-u user] [-t tty] [-f]`; `/etc/adduser [-p] [-u uid] [-g group] [-c comment] name` (root: a passwd line, the next free uid from 100, group staff, a home `/usr/name` of their own — `-p` makes it private, 700; then `passwd name`); `time command` (real, user and system time), `cycles command` (the same, with the measuring gate up around it: `man cycles`, D85), `uptime` (with the load average); `top [-d secs] [-n count] [-b]` (the busiest processes, redrawn; keys `q`, `k` kill, `n` how many, `s` delay); `kill -l`, `kill -HUP pid` (names or numbers); `shutdown`, `reboot`, `halt`. `ps` shows each process's CPU time.

**Text tools:** `sed` (scripts, `-n -e -f -i`, `s///` with groups, ranges, hold space), `tr`, `cut`, `paste`, `join`, `comm`, `xargs`, `diff` (`-u -c -e -r`, and more; its output works with patch and ed), `bc` (an arbitrary-precision calculator; `bc -l` for the math library), and `awk` (POSIX awk: fields, arrays, functions, printf, getline, pipes). Each has a man page; `man -k word` finds commands by subject.

**Compiler tools:** `yacc` (LALR(1) parsers, with `-ly`) and `lex` (scanners, with `-ll`); their output compiles with P32W's own `cc`.

**Lua:** `lua` (Lua 5.4: 32-bit integers, double floats, the string, table, math, io, os, coroutine and utf8 libraries, a REPL) and `luac` (compile to bytecode). `require` looks in `/usr/lib/lua/5.4` and the current directory. Both are built on the host, since the native `cc` can't yet hold Lua's largest files in memory.

**Users:** `write user` and `wall` send to terminals, `mesg y|n` allows or refuses them (terminals are group `tty`; write and wall are setgid tty); `mail user` sends and `mail` reads (`/usr/spool/mail`, setgid `mail`; login says "You have mail."). Root has `chgrp` and `/etc/mknod`.

**Administration** (root): `df` (any user), `mount` / `umount`, `/etc/mkfs`, `/etc/fsck [-y]` (repairs; after repairing / the machine restarts), `su [-] [user] [-c cmd]`, `at` and `cron` (`/etc/crontab`).

**Time zones:** the clock keeps UTC; programs show local time by `TZ` (a POSIX rule, e.g. `TZ=PST8PDT,M3.2.0,M11.1.0`, `TZ=CET-1CEST,M3.5.0,M10.5.0/3`) or, without it, the first line of `/etc/TZ` (`echo PST8PDT,M3.2.0,M11.1.0 > /etc/TZ` sets it for everyone); neither: UTC. `date` names the zone; `date -u` shows and sets UTC; `ls -l`, `who`, `uptime`, `top`, zip's DOS times and Kermit's dates are local; tar keeps UTC, as its format says.

**Archives** (D25), exchangeable with the Mac: `tar c|t|x[vzf]` (POSIX ustar; `z` through gzip), `gzip` / `gunzip` / `zcat` (`-cdfkqtv1..9`), `zip [-rq0..9] archive files` (adds or replaces entries) and `unzip [-lqtoan] [-d dir] archive [names]`. Modes, owners and modification times travel with the files. On the Mac, make tar archives with `COPYFILE_DISABLE=1 tar ...` (or the `._name` resource-fork files come along); `unzip` leaves out Finder's `__MACOSX/` entries unless `-a`.

## Moving files with Kermit

`kermit` (D25) moves files over a terminal line: `kermit -s file ...` sends, `kermit -r` receives, and `kermit -x` **makes P32W a server** that the other side drives entirely. Options: `-a name` (as), `-T` (a text file: CRLF on the line, LF here), `-e length` (longest packet, default 1000), `-w window` (default 3), `-l /dev/tty1` (use the other line), `-q`. Long packets, sliding windows, CRC checks and attribute packets (type, size, modification time) are negotiated with the other Kermit; a lost or damaged packet is sent again.

From the Mac, with **C-Kermit** (`brew install c-kermit`):

- In the emulator, start `p32emu ... -T 5000` (the second terminal on TCP); in C-Kermit: `set host /raw-socket localhost 5000`, then `connect`.
- On the hardware: `set line /dev/cu.usbserial-…`, `set speed 115200`, `set flow none`, `set carrier-watch off`, then `connect`.

Log in, type `kermit -x`, and escape back to C-Kermit (Ctrl-\ then `c`). Then: `send file` (or `send *.c`), `get name` (wildcards `*` `?`), `remote dir`, `remote cd /tmp`, `remote delete name`, `remote type name`, and `finish` to end the server (you are back at the shell). C-Kermit decides text or binary by the file name and says so in its attribute packet; P32W stores text with LF line ends. Times: P32W keeps UTC and C-Kermit sends local time, so received files are off by the time zone until P32W has one.

## Later, and regularly

`/etc/cron` (started by `/etc/rc`) runs `/etc/crontab`'s lines — `minute hour day month weekday command`, fields `*`, `n`, `a-b`, `a,b`, steps `/n`, local time — as root, each minute. `at HH:MM [MM/DD]` or `at now + N minutes|hours|days` queues the commands on its input to run once, as you, in the current directory, output appended to `$HOME/at.out`; `atq` lists your jobs, `atrm job` removes one. `sh -c 'commands'` runs a command line.

## More filesystems

`/etc/mkfs special nblocks [ninodes]` makes an empty filesystem on a block device; `/etc/mount special dir` mounts it, `/etc/umount special|dir` unmounts it (not while a file or a current directory on it is in use); `/etc/mount` lists them (`/etc/mtab`); `df` shows the space on each; `/etc/fsck [special]` checks one. `/dev/ram` is a RAM disk of 256 blocks (empty after a reboot): `/etc/mkfs /dev/ram 256; /etc/mount /dev/ram /mnt`. Links cannot cross filesystems (`EXDEV`); `cp` the file instead. Halt, reboot and shutdown write out and mark clean every mounted filesystem.

## Debugging

`db program [args]` runs a program under the debugger, stopped before its first instruction: `b where` (a function, `sym+off` or `0xADDR`), `c`, `s [n]` (step instructions), `n [n]` (over calls), `f` (finish the function), `bt`, `r` (registers), `set REG VALUE`, `x where [n]`, `p sym`, `l [where] [n]` (disassemble), `bl` / `d [n]` (list / delete breakpoints), `sym [prefix]`, `q`; an empty line repeats `s`, `n` or `l`. It works at the machine level, with the names from the a.out's symbol table (`ld` writes one; `ld -s` leaves it out). Underneath: `ptrace` (system call 26) — a traced program stops at every signal and after `exec`, and `wait` reports the stop (`WIFSTOPPED`).

## Programming tools (M7)

`cc [-c | -S | -E] [-o OUT] [-I DIR] [-D NAME[=VAL]] [-w] files...` is the C compiler (the language of `cc/README.md`: C89 without floating point, `long long`, bit-fields or struct arguments and results): `cc -o prog prog.c` compiles, assembles and links with `/usr/lib/libc.a`; headers are in `/usr/include`. Underneath: `as [-o file.o] file.s` (the syntax of the host's `p32asm.py`), `ld [-o prog] /usr/lib/crt0.o file.o ... /usr/lib/libc.a`, `ar lib.a file.o ...` (or V7 keys: `ar r|q|t|x|d[v] lib.a [files]`). All four write exactly what the host tools write, and `cc` compiles itself. Each `.c` file is compiled in a process of its own, which holds the file's tokens after preprocessing (28 bytes each) and its largest function's tree. Lua's biggest file, `lvm.c` (53,000 tokens, one function of 35,000), takes 3 MB of a process's 8 MiB (D33, D34); split much bigger programs into several files.

`sh` lists: `a ; b`, `a & b`, `a && b` (b if a succeeded), `a || b` (b if it failed); `NAME=value` and `export NAME=value` set variables.

Without C-Kermit, `tools/pykermit.py --tcp 5000 send|get|dir|finish ...` does the same against a P32W server (the tests use it).

`vi [file]` is the screen editor, for a VT100-style terminal (the Mac's Terminal or `nc`; `LINES` / `COLUMNS` set its size, 24 × 80): counts; `h j k l` and the arrows, `w b e 0 ^ $ G gg H M L`, `^F ^B ^D ^U`, `f t F T ; ,`, `%`; `x X r J ~`; `d c y` with a motion or doubled, `D C Y p P`; `i a I A o O R`; `u` (undo), `.` (repeat); `/re ?re n N`; `ZZ`; `:w :q :q! :wq :x :e file :r file :N`, `:[range]s/re/text/[g]`, `:[range]d`, `:!cmd` (ranges `N,M . $ %`; regular expressions `. * ^ $ [...]`). Long lines scroll sideways instead of wrapping.

`ed` is the editor (as on V6: no screen needed): `a` appends lines until a line with `.`, `p` / `n` print (`,p` the whole file), `d` deletes, `s/old/new/g` substitutes, `w` writes, `q` quits; addresses are line numbers, `.`, `$`, `/pattern/`. See the head of `software/cmd/ed.c`.

## Writing programs

On P32W Unix itself: `cc prog.c` (see `man cc`; `cc -h` lists the options), then `./a.out`. The C library is Unix-shaped (`stdio`, `string`, `stdlib`, `unistd`, `fcntl`, `signal`, `dirent`, `pwd`...; `man 3 intro`), the system calls are in section 2 (`man 2 intro`), and `db` debugs the result.

On the host, the same language is compiled with `cc/p32cc.py` (C89 without floating point; cc/README.md). To make a program part of the system, add it to `software/cmd/`, to `CMDS` in `software/Makefile` and to `software/root.manifest`, and `make disk`.

## The system's sources and make

`/usr/src` holds the sources of P32W Unix, each directory with a makefile for `make` (`man make`):

| Directory | `make` | `make install` (as root) |
|---|---|---|
| `/usr/src/cmd` | rebuilds the commands whose source changed (`make ls`: one) | copies each rebuilt one that differs into `/bin` or `/etc` |
| `/usr/src/cc` | rebuilds the C compiler (`make test`: it compiles itself, identically) | `/bin/cc` |
| `/usr/src/libc` | rebuilds `libc.a` and `crt0.o` | `/usr/lib` |

The disk comes with everything already built there, so a first `make` says there is nothing to do and later ones rebuild only what was edited. `make` in `/usr/src/kernel` builds the kernel, `unix.bin`, byte for byte the host's (D86), but it has no `make install` yet: only the host can put a kernel on the disk's boot area (`tools/p32fs.py --kernel`). The group `staff` (ken) may build in `/usr/src`. For your own programs, `make hello` builds `hello` from `hello.c` with no makefile at all.

## The manual

`man name` shows a command's page (a page at a time, through `more`); `man 2 name` and `man 3 name` the system call or library function of that name; `man -k word` lists the pages about it; `man intro` explains the sections: 1 commands, 2 system calls, 3 the C library, 5 file formats, 8 administration. The pages are plain text in `/usr/man` (the source is `software/man/`; `tools/mancheck.py` checks their layout). Every command also answers `--help` (alone) with its page: `ls --help` is `man ls`.

## The date and time

The I/O card's real-time clock (D23) keeps the time with the power off; P32W Unix reads it at boot and keeps UTC. `date` prints it; as root, `date 2610141230` sets it to 2026-10-14 12:30 (`[cc]yymmddhhmm[.ss]`, as in Unix V7) — the clock is set too. In the emulator the clock follows the Mac's (`-R SECONDS` fixes a start time, as the tests do).

## Devices

Block devices (through the buffer cache): `/dev/disk`, the filesystem area of the CF card (root only). Character devices: `/dev/tty0`–`/dev/tty3` (the UARTs), `/dev/tty4` (the video card's screen and keyboard, D53), `/dev/null`, `/dev/mem` (root), `/dev/port` (reading gives the input port, writing sets the output port), `/dev/rdisk` (the whole card, raw; root only).

# 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`.

# P32W Unix internals

docs/kernel.md

**Status:** DESCRIBES THE KERNEL AS BUILT (2026-10-26), `kernel/`, about 6,100 lines of C and 220 of assembly. The design it grew from is `os-design.md` (D22, written before the code); this chapter is what the code does. Its companions: the boot sequence, the filesystem, using P32W Unix.

P32W Unix is a **Unix V6/V7-style kernel**: one address space per process, processes made by `fork` and changed by `exec`, everything a file, a non-pre-emptive kernel that sleeps on events. The hardware's MMU (a software-refilled TLB with address-space IDs, D13) shapes its memory code, and its I/O cards shape its drivers.

## 1. Where things are

| File | What |
|---|---|
| `start.s` | the entry point; the trap entry and exit; the TLB-miss fast path; `swtch`; the privileged instructions as C functions |
| `main.c` | `kmain`: start-up, in order (boot §5) |
| `vm.c` | physical frames, page tables, copy-on-write, the lazy heap, the growing stack, user-memory access |
| `proc.c` | processes, the scheduler, sleep and wakeup, fork / exit / wait, signals |
| `trap.c` | traps and interrupts that `start.s` does not finish; the clock |
| `syscall.c` | the system calls |
| `exec.c` | loading a program |
| `fs.c`, `bio.c` | the filesystem and the buffer cache (filesystem) |
| `file.c` | open files, pipes, the device switches, `/dev/mem`, the raw disk |
| `cf.c`, `ramdisk.c`, `usb.c` | the block devices: the disk card's drives, a RAM disk, USB drives |
| `tty.c`, `video.c` | terminals: the UARTs, the video console, pseudo-terminals |
| `net.c` | sockets, through the network coprocessor |
| `rtc.c` | the real-time clock |
| `kprintf.c`, `klib.c` | console messages, `panic`, string routines |

**Limits** (`param.h`):

| | |
|---|---|
| processes | 64 (`NPROC`); slot *k* has ASID *k* + 1 |
| open files | 20 per process, 128 in the system |
| in-memory inodes | 96 |
| buffer cache | 48 blocks of 1 KiB |
| clock | 50 Hz (`HZ`); a time slice is 2 ticks (40 ms) |
| user address space | 8 MiB (`USERTOP`, D34); the stack up to 1 MiB (D41) |
| exec arguments | 4 KiB of strings, 250 arguments and 250 environment strings |

# 2. Memory

## 2.1 Physical memory

The kernel is loaded at `0x0010_0000`. It runs in supervisor mode, where addresses are **physical** (isa-system.md C2), so it reads and writes any RAM or device register directly.

At start-up `kinit` probes every 2 MiB bank of the four memory windows (D60), and puts every free 4 KiB **frame** above the kernel's end, in the banks it found, on a free list. Each frame on the list holds the address of the next. Frames in missing banks are marked in use for good, so nothing ever takes them. A byte per frame (`fref`) counts how many address spaces share it (§2.4).

- `frame_alloc` / `frame_free`: one zeroed frame / give one back (or drop one reference to a shared one).
- **Page tables** need two *contiguous* frames (below), so `pt_new` searches the reference counts for two free frames side by side and then takes them off the free list.

## 2.2 A process's address space

| User addresses | What | Permissions |
|---|---|---|
| `0x0000_0000` – `0x0000_0FFF` | never mapped: a null pointer faults | — |
| `0x0000_1000` – | **text** (the program's code) | read + execute |
| next page – | **data**, then **bss** (zeroed), then the **heap** up to the break | read + write |
| … | unmapped: the heap grows up, the stack down | |
| up to `0x0080_0000` | **the stack**, growing down on demand, at most 1 MiB | read + write |

Addresses `0x8000_0000` and above trap in user mode, and so do the device registers. User programs reach devices only through the kernel.

## 2.3 Page tables and the TLB

The MMU card has a **split TLB**, 64 instruction and 64 data entries, each tagged with an 8-bit **ASID**. Software fills it (D13): a miss is a trap.

Each process has **one page table**: 2,048 entries, one per 4 KiB page of its 8 MiB, each **in** `TLBLO` **format** (frame number, X, W, V, plus the software bit `PTE_COW`, which the TLB ignores). The table fills two physically contiguous frames, so the entry for virtual page *n* is simply `pt[n]`.

**The refill fast path** (`trapvec` in `start.s`): on a TLB miss (causes 8–10) the handler saves only r12 and r13, takes the page number from `BADADDR`, loads `pt[n]` from the current table (`curpt`), and if it is valid writes it with `TLBW` (TLBHI was already loaded by the miss) and returns with `RTE`. That is about 20 instructions and one memory load. A miss on an unmapped page goes the slow way, to `trap()` (§3), which may grow the heap or the stack, or deliver `SIGSEGV`.

**Address-space IDs.** Process slot *k* always uses ASID *k* + 1, and the TLB matches only entries of the current ASID. A context switch is just `MTCR ASID` and a new `curpt`: no flush. The whole TLB is flushed (`tlbflush`) only when a mapping is removed or changed (`munmap`-like shrinking of the break, exec, fork's write-protection) or a slot is reused by a new process.

## 2.4 fork: copy-on-write

`pt_copy` gives the child a page table pointing at **the same frames**:

- read-only pages (text) are simply shared;
- writable pages become read-only with `PTE_COW` set, **in both processes**, and the frame's reference count goes up.

The first write to such a page is a `PROTECT` fault. `cow_break` copies the frame for the writer (or, if it is the last sharer, just makes it writable again) and writes the new TLB entry directly. The kernel's own writes into user memory (`copyout`) break copy-on-write the same way. A `fork` followed by `exec`, the common case, therefore copies almost nothing.

### 2.5 The lazy heap and the growing stack

- `sbrk` only *promises* pages: it raises the break and **reserves** a frame for each new page (`heap_reserve`), so that no other allocation can take them. The page is mapped (zeroed) on its first touch: a TLB-miss fault in the heap, or the kernel reaching it with `copyin` / `copyout`. A program that asks for 4 MiB and uses 100 KiB costs 100 KiB, yet `sbrk` still fails at once, as it should, when memory is short.
- **The stack** grows on demand: a fault below the lowest stack page but within 1 MiB of the top (and above the heap) maps the pages down to it (`grow_stack`).
- **Statistics:** each process counts its TLB refills, copy-on-write copies and stack pages (and its children's when they are reaped), for `ps`, `top` and `time`.

### 2.6 Reaching user memory from the kernel

The kernel never trusts a user pointer. `copyin`, `copyout`, `copyinstr` and `user_ok` translate each page through the process's page table (`va2pa`), growing the heap or stack or breaking copy-on-write as a user access would, and return `-EFAULT` for an address the process could not reach itself.

## 3. Traps and interrupts

**Entry.** Every exception, `SYSCALL` and interrupt goes to `VBR = trapvec`, in supervisor mode with interrupts off. r15 is banked (D6), so `sp` is already the supervisor stack: the current process's **kernel stack**, one 4 KiB frame per process. Apart from the TLB-miss fast path, `trapvec` builds a **trap frame** on it (r1–r14, the user `sp`, `EPC`, `STATUS`, `CAUSE`, `BADADDR`: 80 bytes) and calls `trap()`.

`trap()` (`trap.c`) sorts it out:

| What | Handling |
|---|---|
| an interrupt (`CAUSE` bit 31) | dispatched by the **vector** the card put in `CAUSE` bits 15:8 (D57): window 7, device 0 the clock; 1–4 the UARTs; 8 the network; window 5 the keyboard. No vector (`0xFF`): by the level, asking each device on it. |
| a bus error while `probing` | `kinit`'s memory probe: skip the access |
| anything else in the kernel | `panic("kernel trap")` |
| `SYSCALL` from user mode | `syscall()` (§5), with interrupts on |
| a TLB miss in the heap or below the stack | a heap page made, or the stack grown; `RTE` retries the access |
| `PROTECT` on a copy-on-write page | the page copied; `RTE` retries |
| any other user fault | a signal: `SIGILL` (illegal or privileged instruction), `SIGTRAP` (`BREAK`), `SIGBUS` (misaligned, bus error), `SIGSEGV` (the rest), with a console message unless the program catches it |

**On the way back to user mode** `trap()` writes delayed disk blocks if the 30-second flush is due, delivers a pending signal, and gives up the CPU if the clock asked for a reschedule. Then `trapret` restores the frame, and `RTE` returns to the user program.

**The clock** (`clockintr`, 50 Hz, the I/O card's timer reloading 20,000 microseconds): it acknowledges the timer and counts `ticks`. It charges the tick to the interrupted process (user or system time), asks for a reschedule after 2 ticks, fires `alarm` signals, and polls the network. Every 5 s it updates the load average (classic Unix's 1-, 5- and 15-minute exponential averages, 11-bit fixed point), and every 30 s it sets the flush flag.

## 4. Processes

### 4.1 The process table (`proc.h`)

A `struct proc` per slot holds:

- the state: `UNUSED`, `EMBRYO`, `SLEEPING`, `RUNNABLE`, `RUNNING`, `ZOMBIE`, `STOPPED`;
- the pid and parent, the ASID, the page table, the break, the stack bottom, the text end;
- the kernel stack, its trap frame and its saved **context** (the callee-saved registers r8–r14, `sp`, the return address);
- 20 open-file slots, the current directory, the user and group IDs (real and effective), the umask and the controlling terminal;
- signals (a pending bitmask, a handler per signal), the alarm, the CPU times, the ptrace state.

### 4.2 Scheduling: not pre-emptive in the kernel

The kernel follows V6: **a process running in the kernel keeps the CPU until it sleeps** or returns to user mode. The clock never switches processes in the middle of kernel code; it only sets `need_resched`, and the switch happens in `trap()` on the way out. So kernel data needs no locks between processes, only protection from interrupt handlers. `splhi()` turns interrupts off and `splx()` restores them, around the few structures an interrupt handler also touches (terminal queues, the buffer list, the process table).

**The scheduler** runs in its own context on the boot stack. It loops over the table **round-robin** from where it last stopped, picks the next RUNNABLE process, loads its ASID and page table (`switchuvm`), and calls `swtch(&schedctx, &p->ctx)`. When nothing is runnable it turns interrupts on and executes `WAIT`: the CPU idles until an interrupt (D11).

`swtch` saves the callee-saved registers, `sp` and the return address in one context and loads another. A process gives up the CPU with `sched()`: `yield` (its slice is over: RUNNABLE), `sleep` (SLEEPING) or `exit` (ZOMBIE). It resumes inside its own `sched()` call when the scheduler picks it again.

### 4.3 Sleep and wakeup

`sleep(chan)` marks the process SLEEPING on an address (a buffer, an inode, a tty queue, a child's parent) and switches away. `wakeup(chan)` makes every process sleeping on that address RUNNABLE. The sleeper checks its condition again when it runs, because another process may have got there first. Interrupts are off from the caller's test of its condition until the process is asleep, so a wakeup from an interrupt handler cannot fall between the two. A signal ends a sleep (`psignal` makes the process RUNNABLE), so slow calls return `EINTR`.

### 4.4 fork, exit, wait

- `fork`: a new slot and pid, a kernel stack with a copy of the parent's trap frame (r1 = 0 in the child), a copy-on-write page table (§2.4), the open files shared (reference counts), the same directory, IDs, umask, terminal and signal

handlers. Its context starts in `trapret`, so the child's first run is a return to user mode.

- `exit`: closes the files, releases the directory, gives orphans to init, becomes a ZOMBIE holding its status, wakes its parent.
- `wait`: finds a ZOMBIE child, adds its CPU times and memory statistics to the parent's, frees its page table, frames and kernel stack, flushes the TLB (its ASID will be reused) and frees the slot. With no zombie it sleeps; with no children it returns `ECHILD`. A traced child that stopped is reported too.

**Process 1** is built by hand at start-up (boot §5.3).

### 4.5 exec (`exec.c`)

1. Look up the file. It must be a regular file the process may execute, and start with the a.out header: magic `0x5033_3277`, text size, data size, bss size, entry point (`assembler/p32ld.py` writes it).
2. Copy the arguments and environment into kernel frames *before* touching the old program, because they live in the old address space.
3. Build a **new** page table: text pages read + execute from `0x1000`, data and bss read + write from the next page boundary. The contents come straight from the disk into the pages (`readi_pages`: runs of consecutive blocks in one ATA command, past the buffer cache). The stack's top page holds the strings, then `argc`, `argv[]`, NULL, `envp[]`, NULL.
4. Only now **commit**: free the old page table, switch to the new one, flush the TLB, set the break after the bss, apply a set-user-ID or set-group-ID bit, reset caught signals to their defaults, and point the trap frame at the entry with the new stack. An exec that fails at any earlier step leaves the old program untouched, and the call returns an error to it.

### 4.6 Signals

V7's numbers and semantics: a caught signal reverts to the default when it is delivered (`signal` must be called again). Pending signals are a bitmask. `kill` (by root, or by a process of the same user) and the terminal (^C `SIGINT`, ^\ `SIGQUIT`, hang-up `SIGHUP`) post them with `psignal`, which also ends an interruptible sleep.

**Delivery** (`issig`, on the way back to user mode): `SIGKILL` first, then the lowest pending.

- Ignored: nothing happens.
- Default: the process exits with the signal number as its status.
- Caught: the kernel copies the whole trap frame onto the **user stack**, points `EPC` at the C library's trampoline (`__sigtramp`, registered as handler 0), with r1 = the signal and r2 = the handler. The trampoline calls the handler, then calls `sigreturn`, which copies the frame back: every register as it was when the signal arrived.

**ptrace** (for `db`, the debugger): a traced process stops at every signal and at exec. Its parent sees the stop through `wait`, can read and write its memory and registers (`pt_private` gives it a private copy of a shared text page for a breakpoint), and lets it continue with or without the signal.

## 5. System calls

A program executes `SYSCALL` with the call's number in r1 and up to four arguments in r2–r5. The trap frame holds them, `syscall()` advances `EPC` past the instruction, calls the handler from the table, and puts the result in the frame's r1: **a value ≥ 0, or −errno**. Every other register is preserved (the C library turns −errno into −1 and `errno`). A call that sleeps simply sleeps inside the handler: the kernel stack keeps its state.

The numbers follow V7 (`software/include/sys/syscall.h`): 1 `exit`, 2 `fork`, 3 `read`, 4 `write`, 5 `open`, 6 `close`, 7 `wait`, 8 `creat`, 9 `link`, 10 `unlink`, 11 `execve`, 12 `chdir`, … 17 `sbrk`, 18 `stat`, 19 `lseek`, … 48 `signal`, 54 `ioctl`, … then P32W's additions: `mkdir`, `rmdir`, `rename`, `dup2`, `uname`, `sigreturn`, `reboot`, `pinfo` (one process-table slot, for `ps` and `top`: there is no `/dev/kmem`), `setsid`, `sysinfo`, `ustat`, `fcntl`, `select`; the 4.2BSD socket calls at their BSD numbers (`socket` 97, `connect` 98, … D45); and `netctl` (160).

## 6. Files

- **The open-file table** (128 entries): an open file is a type (an inode, a pipe or a socket), a reference count, the open flags and the offset. `fork` and `dup` share an entry, so the offset is shared, as in Unix.
- **Pipes**: a 1 KiB ring in a frame, with reader and writer counts. Readers sleep while it is empty, writers while it is full. Writing with no reader left is `SIGPIPE`.
- **Device files** go through two switches, Unix V6's: **block devices** (`bdevsw`: open, close, *strategy*, which transfers one buffer) for anything a filesystem can live on, and **character devices** (`cdevsw`: open, close, read, write, ioctl) for the rest. The table of devices is in the filesystem §8.
- **The filesystem** itself: its own chapter.

## 7. Terminals (`tty.c`)

Nine terminals share one line discipline: the four UARTs (`tty0`–`tty3`), the video console (`tty4`, D53) and four pseudo-terminals (`ttyp0`–`ttyp3`, D46).

- **Input** arrives by interrupt (a UART's received-data interrupt, the keyboard, or a pty master's write) and goes through `input()`: echo, erase (^H or DEL), line kill (^U), ^C → `SIGINT` and ^\ → `SIGQUIT` to every process on the terminal, ^D end of file, CR → LF. The *raw* queue holds the line being typed. Return moves it to the *canonical* queue, which `read` takes from. **CBREAK** delivers each character at once, with the signals still working. **RAW** passes everything through as typed. Modes are set with `ioctl(TIOCSETP)` (`stty`), as in V7.
- **Output** goes to a 4 KiB queue, and the UART's transmit-empty interrupt drains it. A writer sleeps while the queue is full. LF goes out as CR LF (CRMOD).
- **The controlling terminal** is the first terminal a process opens while it has none: `setsid` (getty's parent, init, calls it) clears it, so the terminal getty opens becomes its session's, and children inherit it. `/dev/tty` is always "this process's terminal".
- **Pseudo-terminals:** a slave (`/dev/ttypN`) is an ordinary terminal to the programs on it. Its master (`/dev/ptypN`), held by `telnetd`, writes the slave's typing and reads its output. Closing the master hangs up the slave.

## 8. Other drivers

- **The disk card** (`cf.c`): ATA PIO, polled, 16-bit data (D26). The boot CompactFlash card's filesystem area is minor 0. The second CF socket and the four IDE drives are minors 1–5, probed at start-up. A 1 KiB block is one two-sector command.
- **USB** (`usb.c`, D43): the I/O card's two CH376S controllers in sector mode, polled. A drive is found when first opened (connect, `DISK_INIT`, capacity). Pulled out, the next access fails with `EIO` and its cached blocks are dropped. Every wait has a time limit.
- **The network** (`net.c`, D45): sockets as file descriptors and the BSD calls, with no protocol code in P32W. Each operation is a request and a reply through the coprocessor's mailbox (docs/net-mailbox.md), byte by byte and polled.

The kernel is not pre-empted, so two processes never interleave on the mailbox. A socket that must wait sleeps until the coprocessor's `ATTN` interrupt (IRQ7), or a 200 ms tick, and looks again. `select` covers terminals, pipes and sockets.

- **The clock** (`rtc.c`, D23): the DS12C887, read at boot and set by `stime` (binary mode, 24-hour, UTC).
- **The video console** (`video.c`, D53), when the video card is fitted: the screen as an ANSI-style terminal, drawn by the card's HD63484 from commands (D103, D104). A character is one graphic copy from a picture of the font; a scroll is one copy by the chip. The command words wait in a ring and go to the chip eight at a time from its "write FIFO empty" interrupt, so nothing spins while it draws; kernel messages wait for it. The keyboard's characters come as ASCII from the card's PS/2 section. That section is an ATmega328P with a will of its own (D109): two seconds starting after a reset, and a command written before it has taken the last is lost. So the three commands that start it (interrupts off, which may be dropped; the mouse off; interrupts on) go out from the clock's tick, one each time the card shows BUSY clear (`kbdtick`), and the kernel never waits for it. An interrupt brings one byte.

# The P32W Unix filesystem

docs/filesystem.md

**Status:** DESCRIBES THE SYSTEM AS BUILT (2026-10-26). Two programs read and write this format, and they must agree:

- `tools/p32fs.py` on the Mac: builds disk images, puts files in, lists, checks (`fsck`). Its opening comment is the format's reference.
- **The kernel**, `kernel/fs.c` (files, directories, inodes, blocks), `kernel/bio.c` (the buffer cache) and `kernel/cf.c` (the disk), with `mkfs` and `fsck` on P32W itself (`software/cmd`).

`make test` checks each against the other: an image built on the Mac is used and changed by P32W Unix, and either side's `fsck` must pass on the other's work. The design began in `os-design.md` §5 (D22); this chapter is what was built.

It is a **Unix V7-style inode filesystem**: files are numbered inodes, directories are files of (number, name) entries, and a file's data is found through a table of block numbers in its inode. It differs from V7's in four ways: 1 KiB blocks, 28-character names, a free-block **bitmap** instead of V7's free list, and a triple-indirect block (files to 2 GiB).

## 1. The disk

A CompactFlash card (or any ATA drive) is a row of 512-byte sectors. P32W Unix's boot card has three regions:

| Sectors | Region | Who uses it |
|---|---|---|
| 0 | the **boot header** (`"P32B"`, kernel size, load address, entry, the filesystem's first sector) | the monitor's `b` (boot) |
| 1 … 2047 | the **boot area** (1 MiB): the kernel image | the monitor's `b` |
| 2048 … end | **the filesystem** | the kernel, as block device 0, minor 0 (`/dev/disk`) |

The filesystem's first sector is the boot header's word 4 (2048 unless the image says otherwise; `cf.c` reads it at boot). Everything below counts in **blocks of 1 KiB** (two sectors) from that point: block *b* is sectors `2048 + 2b` and `2048 + 2b + 1`, moved by one two-sector ATA command.

Other drives (the second CF socket, the IDE drives, a USB stick, the RAM disk) hold a filesystem from their sector 0, with no boot area.

## 2. The layout inside the filesystem

| Blocks | Contents |
|---|---|
| 0 | **the superblock** |
| 1 … | **the inode table**: 64-byte inodes, 16 per block |
| then | **the free-block bitmap**: one bit per block of the filesystem |
| then | **data blocks**: file contents, directories, indirect blocks |

**The superblock** (block 0), nine little-endian 32-bit words:

| Word | Field | Meaning |
|---|---|---|
| 0 | `magic` | `0x5033_3246` (the bytes `F23P`) |
| 1 | `nblocks` | blocks in the filesystem |
| 2 | `ninodes` | inodes (a multiple of 16) |
| 3 | `inode_start` | the inode table's first block (1) |
| 4 | `bmap_start` | the bitmap's first block |
| 5 | `data_start` | the first data block |
| 6 | `clean` | 1: written out and unmounted cleanly; 0: in use, or it crashed |
| 7 | `free_blocks` | free data blocks (kept up to date by the kernel) |
| 8 | `free_inodes` | free inodes |

**An example:** the disk `make disk` builds (`DISK_MB = 512`, `DISK_INODES = 16384`): 1,048,576 sectors; the filesystem from sector 2048 has 523,264 blocks; 16,384 inodes fill blocks 1–1,024; the bitmap (523,264 bits) is blocks 1,025–1,088; data starts at block 1,089. On the Mac, `p32fs.py mkfs` gives one inode per 8 blocks by default (at least 64, at most 512 Ki).

**The bitmap:** bit *b* (bit `b mod 8` of byte `b / 8`, counting from `bmap_start`) is 1 if block *b* is in use. The blocks of the superblock, the inode table and the bitmap itself are marked in use when the filesystem is made, so the allocator never hands them out.

## 3. Inodes

Inode *n* is the 64 bytes at offset `(n mod 16) × 64` of block `inode_start + n / 16`. **Inode 0 is never used** (a directory entry with inode 0 is an empty slot). **Inode 1 is the root directory.** An inode whose `mode` is 0 is free.

| Bytes | Field | |
|---|---|---|
| 0–1 | `mode` | the type and permissions (below) |
| 2–3 | `nlink` | how many directory entries name it |
| 4–5 | `uid` | owner |
| 6–7 | `gid` | group |
| 8–11 | `size` | in bytes |
| 12–15 | `mtime` | last modified (seconds since 1970, UTC) |
| 16–19 | `ctime` | created / inode changed |
| 20–63 | `addr[11]` | block numbers: 8 direct, 1 single-, 1 double-, 1 triple-indirect |

`mode` (octal, as in V7):

| Bits | Meaning |
|---|---|
| `0170000` | type: `0100000` regular file, `0040000` directory, `0020000` character device, `0060000` block device, `0010000` pipe (`0140000` socket only in `fstat`, never on disk) |
| `04000` | set-user-ID: the program runs with its owner's user ID |
| `02000` | set-group-ID |
| `01000` | sticky (on `/tmp`: only a file's owner may remove it) |
| `0777` | read, write, execute for the owner, the group, others |

**A device file** has no data: `addr[0]` holds its device number, `major << 8 | minor`. Its type says which device table the major indexes (character or block, §8).

**Where a file's bytes are.** Byte offset *o* is in the file's block `fb = o / 1024`. `bmap` (fs.c) finds its disk block:

| File blocks | Found through | Reach |
|---|---|---|
| 0–7 | `addr[0]`–`addr[7]` directly | 8 KiB |
| 8–263 | `addr[8]`: an indirect block of 256 block numbers | + 256 KiB |
| 264–65,799 | `addr[9]`: an indirect block of 256 indirect blocks | + 64 MiB |
| beyond | `addr[10]`: triple-indirect | + 16 GiB |

A file's size is a signed 32-bit offset, so it stops at 2 GiB before the triple-indirect block runs out. A block number of 0 is a **hole**: it reads as zeros, and the block is allocated only when written.

## 4. Directories

A directory is a file of **32-byte entries**: a 32-bit inode number and a name of up to **28 bytes**, padded with NULs (no NUL when it is exactly 28 long). Inode 0 marks a free slot. Every directory begins with `.` (itself) and `..` (its parent; the root's `..` is the root).

- **Lookup** (`dirlookup`) reads the entries in order and compares names. There is no hashing or sorting, as in V7: a directory of a few hundred entries is still read in a few blocks.
- **Adding** a name (`dirlink`) reuses the first free slot, or appends.
- **Removing** one writes inode 0 into its slot. Directories never shrink.
- **Paths** (`namex`): from `/` or the process's current directory, element by element. Each directory along the way must be searchable (execute permission). A name longer than 28 characters is `ENAMETOOLONG`. Crossing a mount point goes to the mounted filesystem's root, and `..` from that root goes back to the directory it is mounted on.

## 5. How the kernel uses it

### 5.1 The buffer cache (`bio.c`)

Every block the filesystem reads or writes passes through **48 buffers** of 1 KiB, the least recently used reused first (Unix V6's `bio`):

- `bread` returns the block's buffer, read from the disk if it is not already there; `brelse` gives it back. A buffer is **busy** while one process holds it, and another wanting it sleeps.
- **Delayed writes:** `bdwrite` only marks a buffer dirty. It is written when the buffer is reused, by `sync`, or by the kernel's own flush every 30 seconds. `bwrite` writes at once (the superblock does).
- `bnoread` gives a buffer for a block about to be overwritten whole, without reading the disk first (a full-block write, a new block).
- **exec's fast path** (`readi_pages`): runs of consecutive blocks not in the cache go straight from the drive into the new program's pages, up to 64 blocks per ATA command, past the cache.

### 5.2 Inodes in memory (`fs.c`)

Up to **96 inodes** are held in a table (`itab`): `iget` finds or claims an entry by (device, number) and counts references; `iput` drops one. When the last reference goes and the link count is 0, the file is freed: `itrunc` returns its blocks and the inode's `mode` becomes 0. A process **locks** an inode (`ilock`, reading it from disk the first time) while it reads or changes it, so two processes never interleave inside one file's blocks.

### 5.3 Allocation

- **Blocks** (`balloc`): the search in the bitmap starts just after the last block allocated on that filesystem. A file being written therefore gets consecutive blocks, and a large disk is not scanned from the start every time. Bytes that are all ones (eight blocks in use) are stepped over at once. A new block is zeroed in its buffer.
- **Inodes** (`ialloc`): the first inode with mode 0, from inode 1.
- The superblock's `free_blocks` and `free_inodes` follow every change. `df` and the boot line report them.

### 5.4 Permissions

`access_ok`: root (effective user ID 0) may read and write anything, and execute a regular file if any execute bit is set. Anyone else gets the owner's bits if they own the file, else the group's if their effective group matches, else the others'. A set-user-ID program sets the effective user ID at `exec`.

### 5.5 Mounting

`mounts[0]` is the root filesystem (`/dev/disk`). Three more can be mounted (`mount /dev/usb /mnt`), each on a directory. Mounting reads the superblock, checks the magic, and writes `clean = 0`. Unmounting (refused while a file on it is open, or a process's current directory is there) writes everything out with `clean = 1`.

## 6. Crash safety

There is no journal, as in V6 and V7. Instead:

- dirty blocks reach the disk within 30 seconds (the clock sets a flag, and the next process returning to user mode flushes the cache), and at every `sync`;
- **the clean flag**: mounting writes `clean = 0`; `halt`, `reboot` and `umount` write everything and then `clean = 1`. A filesystem found not clean at boot gets the warning `not unmounted cleanly (run fsck)`, and `/etc/rc` runs `fsck -q -c`, which checks fully only in that case (D40), so a large disk does not slow every boot.

`fsck` (on P32W and in `p32fs.py`) checks every invariant:

- every block in use is in range and owned by exactly one file;
- the bitmap matches the blocks in use;
- every directory holds `.` and `..`, and names only allocated inodes;
- every link count equals the number of entries naming the inode;
- the superblock's free counts are right.

With `-y` it repairs: it clears bad inodes and blocks owned twice, removes entries that name free inodes, clears inodes nothing names, sets the link counts, and rebuilds the bitmap and the counts. Repairing the mounted root ends with an immediate restart, without writing the kernel's cache back, as V7's fsck did.

## 7. Building and inspecting images on the Mac

`tools/p32fs.py` works on image files:

```
p32fs.py mkfs IMAGE SIZE_MB [--inodes N] [--kernel FILE]
p32fs.py build IMAGE SIZE_MB ROOT_MANIFEST [--kernel FILE] [--inodes N]
p32fs.py ls IMAGE [PATH]          p32fs.py cat IMAGE PATH
p32fs.py put IMAGE HOSTFILE PATH [--mode OCT] [--uid N] [--gid N]
p32fs.py mkdir / mknod / rm / ln / chmod / chown ...
p32fs.py fsck IMAGE
```

`make -C software disk` runs `build` with `software/root.manifest`, one line per directory, file or device node: kind, path, mode, owner, group, then the host file or the device numbers. On P32W itself, `mkfs special nblocks [ninodes]` makes a filesystem on any block device, and `fsck` checks one.

## 8. Devices

A device number is `major << 8 | minor`. **Block devices** move 1 KiB blocks through the buffer cache, and a filesystem can live on one; **character devices** move bytes straight between the driver and the process (`file.c`, `bdevsw` and `cdevsw`).

| Block | Major | Minors | Driver |
|---|---|---|---|
| `/dev/disk` | 0 | 0 | the boot card's filesystem area (`cf.c`) |
| `/dev/disk1`, `/dev/hda`–`hdd` | 0 | 1–5 | the second CF socket, IDE A master/slave, IDE B master/slave: whole drives |
| `/dev/ram` | 1 | 0 | a 256 KiB RAM disk (`ramdisk.c`) |
| `/dev/usb`, `/dev/usb1` | 2 | 0, 1 | a USB drive in either port (`usb.c`, D43) |

| Character | Major | Minors | Driver |
|---|---|---|---|
| `/dev/tty0`–`tty3`, `tty4`, `ttyp0`–`ttyp3` | 1 | 0–3, 4, 5–8 | the UARTs, the video console, the pseudo-terminals' slave ends (`tty.c`) |
| `/dev/null`, `/dev/mem`, `/dev/port` | 2 | 0, 1, 2 | discard; physical memory (root); the I/O card's ports |
| `/dev/rdisk`, `/dev/rdisk1`, `/dev/rhda`… | 3 | 0–5 | the drives' sectors, raw (root) |
| `/dev/tty` | 4 | 0 | the process's controlling terminal |
| `/dev/rusb`, `/dev/rusb1` | 5 | 0, 1 | a USB drive's sectors, raw (the FAT tools) |
| `/dev/ptyp0`–`ptyp3` | 6 | 0–3 | the pseudo-terminals' master ends (`telnetd`, D46) |

# P32W Unix — design proposal

docs/os-design.md

**Status:** ACCEPTED (2026-10-13, D22): all eight review points as proposed. **Progress:** M1 (objects, linker) and M2 (the C compiler `cc/`, the C library) done; M3 (the kernel, `kernel/`) and M4 (users, logins on both terminals, permissions, the shell's variables and wildcards, `ed`) run in the emulator with ~50 commands (`software/cmd/`) — see kernel/README.md and unix.md.

**Ken's direction (2026-10-13):** "a multitasking OS where each user is in a protected area with their own memory space. We will need a filesystem, system calls into OS and classic unix like commands." Choices made with Ken the same day:

| Question | Choice |
|---|---|
| Language | **C, with our own compiler** (and linker) |
| OS basis | **Our own, in the style of Unix V6** |
| Users | **Real multi-user:** logins, user IDs, rwx permissions, root; one terminal per serial port |
| Filesystem | **A Unix inode filesystem** on the CompactFlash card, plus a host tool for disk images |

Working name: **P32W Unix** (the kernel `/unix`, as in V6).

---

## 1. What the machine already gives us

The hardware was designed for this (D12–D16, D50, D20):

| Need | Already there |
|---|---|
| Protected processes | user / supervisor modes; user mode cannot run privileged instructions or reach the I/O page or the upper 2 GiB |
| Own memory space per process | a software-managed TLB (64 I + 64 D entries, 4 KiB pages, V/W/X bits), **8-bit address-space IDs**: a context switch is one `MTCR ASID`, no TLB flush |
| System calls | `SYSCALL` traps to the kernel; `RTE` returns; the faulting/trapping instruction has no effect, so page faults can restart |
| Pre-emption | the I/O card's timer (IRQ0); interrupt priorities and masks |
| Terminals | four TL16C550 UARTs with interrupts (IRQ2, IRQ3) on the I/O card |
| Disk | CompactFlash on the disk card, True-IDE, 512-byte sectors (IRQ1) |
| Memory | RAM from `0x0010_0000`, 16 MiB per memory card, one to four cards (up to 64 MiB, D52); 1 MiB boot ROM (the monitor) |
| Boot | the monitor's `b` command loads an image from the CF card and jumps to it |

**Speed:** about 450,000 instructions a second (3 MHz microcycle, ~6.5 per instruction). That is the class of the PDP-11/40 that ran Unix V6 — with many times its memory. A context switch is a few hundred instructions, a TLB refill ~15, so time-sharing two users is comfortable.

The I/O card's battery-backed real-time clock (D23) keeps the date over power-off.

## 2. The plan in one paragraph

First a **toolchain**: the assembler learns to emit relocatable object files, a new **linker** joins them into executables, and a new **C compiler** (`p32cc`, Python like the assembler) turns C into P32W assembly. With it, a small **C library**. Then the **kernel**, in C with a little assembly: processes with private address spaces, a scheduler, ~35 V6-style system calls, a terminal driver for both UARTs, a disk driver and buffer cache, and an inode **filesystem**. Last, **user programs**: `init`, `getty`/`login`, a shell, `ed`, and the classic commands. Everything runs first in the emulators, under `make test`, with scripted terminal sessions; the same disk image then boots the real machine.

---

## 3. Toolchain

### 3.1 Objects and the linker (`p32asm -c`, `p32ld`)

- The assembler gains **sections** (`.text`, `.data`, `.bss`), `.global` / `.extern`, local labels, and **relocatable output** (`-c`, a simple `.o` format: sections, symbols, relocations). Three relocation kinds cover the ISA: 32-bit absolute (`.word sym`), the `LUI`/`ORI` pair (`LA`, `hi`/`lo`), and the 24-bit word offset of `JMP`/`CALL` across files.
- `p32ld` links objects and archives (`.a`) into an executable or a flat binary (for the kernel), with a link map. Without `-c` the assembler still makes flat images, so everything that exists keeps working.

### 3.2 The executable format (`a.out`, V7-like)

```
header (32 bytes): magic 0x50333277 ("P32w"), text size, data size, bss size,
                   entry, flags, symbol table size, 1 reserved
then the text, then the data, then the symbol table (debuggers; exec skips it)
```

The symbol table (2026-10-14; `assembler/p32ld.py` gives the format): every symbol of every object linked, global and local, sorted by value; `ld -s` leaves it out.

**User address space** (virtual, 8 MiB per process, D34):

```
0x0000_0000  page 0: never mapped (a NULL pointer faults)
0x0000_1000  text (read + execute, shared between processes running the same program)
             data, bss (read + write)
             heap: grows up with brk/sbrk
     ...     (unmapped gap)
0x0080_0000  top of the stack, growing down (pages added on demand)
```

The page table for 8 MiB is 2,048 entries in two physically contiguous frames (§4.2). The window can grow with the memory without changing programs.

### 3.3 The C compiler (`p32cc`)

A one-pass-per-stage compiler in Python: preprocessor → lexer → parser → type checker → code generator → assembly → `p32asm -c`. No optimiser at first beyond constant folding and sensible register use.

**The language: C89 without floating point** (P32W has no FPU; V6 Unix needed none), growing in stages:

1. `int`, `char`, pointers, arrays, functions, `if`/`while`/`for`/`do`/`break`/`continue`/`return`, expressions, globals, string literals.
2. `struct`, `union`, `enum`, `typedef`, `switch`, `goto`, `sizeof`, casts, `static`/`extern`, initialisers, function pointers, `short`, `unsigned`.

3. The preprocessor in full: `#include`, `#define` with arguments, `#if`/`#ifdef`/`#elif`, `#undef`.
4. Variadic functions (`stdarg.h` for `printf`) — done. Not done: `long long` (64-bit, via library helpers).

Types: `char` 8 (signed, as on the PDP-11), `short` 16, `int` and `long` 32, pointers 32, `long long` 64. Little-endian.

**The calling convention (ABI)** — a proposal, to fix early:

| Registers | Use |
|---|---|
| r0 | zero |
| r1–r4 | arguments 1–4 (more on the stack); r1 the result (r1:r2 for 64-bit) |
| r1–r7 | caller-saved (scratch) |
| r8–r13 | callee-saved |
| r14 | frame pointer (callee-saved) |
| r15 | stack pointer; `CALL` pushes the return address (D6) |

**Tests:** a C test suite (`cc/tests/*.c`), each with its expected output, compiled by `p32cc` and run in `p32emu`; programs that are plain C are also compiled by the host `cc` and must print the same. In `make test`.

### 3.4 The C library

Small and Unix-shaped, in C plus assembly stubs: system-call wrappers (`read`, `write`, `open`, `fork`, `exec`, …), `string.h`, `ctype.h`, `stdlib.h` (`malloc`/`free` on `sbrk`, `atoi`, `exit`), a small `stdio` (`printf` family, `getchar`/`putchar`, `fopen`/`fgets`/`fputs` buffered), `errno`, `signal`, `setjmp`/`longjmp`, `dirent` for reading directories, and the 64-bit arithmetic helpers.

---

## 4. The kernel

Written in C (~8,000 lines expected) with assembly for trap entry, the TLB refill and the context switch (~400 lines). It runs in supervisor mode, untranslated, at physical addresses.

### 4.1 Physical memory

```
0x0000_0000  ROM: the monitor (loads the kernel)
0x0010_0000  kernel text and data (budget 256 KiB)
             kernel stacks (4 KiB per process), process table, buffer cache (64 KiB)
             page frames: the rest of the RAM, 4 KiB each (the boot
             message says how much: "memory: 16 MiB of RAM, ... KiB free")
0x00FF_FFFF  top of the first memory card (16 MiB; up to 0x03FF_FFFF
             with four cards, D52)
```

### 4.2 Processes and memory

- **Process table**, up to 64 processes: state, PID, parent, user IDs (real/effective), ASID, page-table frame, kernel stack, saved registers, open files, current directory, signal state, `sleep` channel.
- **Each process has its own address space:** its own page table, 2,048 entries in two contiguous frames, already in `TLBLO` format (§C7 of isa-system.md), so the **TLB-miss handler is one indexed load**, `TLBW`, `RTE`. A missing or forbidden page is a real fault: the stack grows on demand, everything else kills the process (`SIGSEGV`) — the other users carry on.

- **ASIDs:** each running process gets one of 255; when they run out the TLB is flushed and they are handed out again (a rare, cheap event).
- `fork` is **copy-on-write** (M6): parent and child share every page, writable ones marked read-only, and the first write to one (a `PROTECT` fault) copies it. **Text pages are shared** outright between processes running the same program. (V6 copied the whole image.)
- `exec` reads the `a.out` from the filesystem into fresh pages, sets up the arguments on the new stack, and enters user mode with `RTE`.
- **The kernel checks every user pointer** (`copyin` / `copyout` through the caller's page table): a system call cannot be used to read or write another process's memory or the kernel.
- **Swapping:** not at first. A memory card's 15 MiB of frames holds dozens of small processes; a full memory makes `fork` fail with `EAGAIN`. Swapping to the CF card can come later if it is ever needed.

### 4.3 Scheduling and sleeping

- **Pre-emptive round-robin** on the timer interrupt (proposal: 50 Hz, a 20 ms slice), with V6 `sleep(channel)` / `wakeup(channel)` for waiting on the disk, a terminal, a pipe or a child. `WAIT` idles the CPU when nothing can run.
- The kernel itself is **not pre-empted** (V6 style): it runs a system call to completion or to a `sleep`; interrupt handlers only queue work and wake sleepers. Simple and safe on one CPU.

### 4.4 System calls

`SYSCALL` with the number in r1 and arguments in r2–r5 (isa-system.md §8); the result comes back in r1, and failure as **−errno** (the library turns it into `−1` and `errno`). The V6 set, with V7 names where clearer:

| Group | Calls |
|---|---|
| Processes | `fork` `exec` (`execve`) `exit` `wait` `getpid` `getppid` `kill` `signal` `pause` `alarm` `nice` `sbrk` |
| Files | `open` `creat` `close` `read` `write` `lseek` `dup` `dup2` `pipe` `stat` `fstat` `ioctl` (terminal modes) |
| Names | `link` `unlink` `mkdir` `rmdir` `chdir` `mknod` `chmod` `chown` `access` `rename` |
| Users | `getuid` `geteuid` `getgid` `setuid` `setgid` `umask` |
| Time, system | `time` `stime` `times` `sync` `uname` |

**Signals,** V7's fifteen, with V7's numbers: `SIGHUP`, `SIGINT` (^C), `SIGQUIT` (^), `SIGILL`, `SIGTRAP`, `SIGABRT`, `SIGEMT`, `SIGFPE`, `SIGKILL`, `SIGBUS`, `SIGSEGV`, `SIGSYS`, `SIGPIPE`, `SIGALRM`, `SIGTERM` (no `SIGCHLD`: a parent learns of its children through `wait`). A handler runs in user mode through a small trampoline on the user stack; a caught signal resets to the default action, as in V7.

### 4.5 Devices

| Device | Driver |
|---|---|
| `/dev/tty0`, `/dev/tty1` | UART 0 (the console) and UART 1, interrupt-driven, with a line discipline: echo, erase (backspace / DEL), line kill (^U), ^C → `SIGINT`, ^\ → `SIGQUIT`, ^D end of file, a raw mode for programs that want characters |
| `/dev/disk` | the CompactFlash card's filesystem area: a block driver (PIO, 512-byte sectors) under a **buffer cache** (V6 `bread` / `bwrite`, write-back, flushed by `sync` and every 30 s) |
| `/dev/rdisk` | the card raw, by its own sector numbers: reads and writes go through one sector buffer, bypassing the cache |
| `/dev/ram` | a 256 KiB RAM disk (block), for a scratch filesystem (`mkfs`, `mount`) |
| `/dev/null` | discards writes; reads give end of file |

| Device | Driver |
|---|---|
| `/dev/mem` (root only) | physical memory: ROM and RAM by physical address |
| `/dev/port` | the ports on the I/O card: the 8-bit input and output ports |
| `/dev/tty` (character 4) | the process's controlling terminal, whichever it is |

### 4.6 Boot

1. The monitor's `b` loads the kernel from the boot area at the start of the card (the existing "P32B" header) and jumps to it.
2. The kernel sets up memory, the TLB and the devices, mounts the root filesystem, and starts `/etc/init` as process 1.
3. `init` runs `/etc/rc` (checks the filesystem, cleans `/tmp`), then a `getty` on each terminal; `getty` runs `login`, `login` checks `/etc/passwd`, sets the user's IDs and directory, and runs their shell.
4. `shutdown` / `sync` before power-off (no journal, as in V6: §5.4).

---

## 5. The filesystem

A V7-style inode filesystem, sized for a CF card, with names longer than V7's 14 characters.

### 5.1 Layout

**Blocks of 1 KiB** (two sectors). The card holds:

```
sectors 0..N   boot area: "P32B" header + the kernel image (the monitor boots this)
then block 0   superblock: magic, size, inode count, the free-block bitmap's place, clean flag
     block 1.. the inode table (64-byte inodes, 16 per block)
     then      the free-block bitmap (one bit per block)
     then      data blocks
```

**Inode (64 bytes):** type and permissions (16 bits: file / directory / character / block device / pipe, set-user-ID, set-group-ID, sticky, rwx × 3), link count, owner uid and gid, size, modification and change times, **11 block numbers: 8 direct, 1 single-, 1 double-, 1 triple-indirect** (files to 2 GiB: the triple block — the former padding word, 2026-10-14 — reaches 16 GiB, the signed 32-bit file offset stops at 2 GiB).

**Capacity** (Ken: "we will want a bigger cf capacity"): the CF driver addresses $2^{28}$ sectors (LBA28, 128 GiB); block numbers are 32-bit. The kernel's block allocator starts each search in the bitmap block where the last one ended and steps over full bytes, so a large disk is not scanned from its start every time. `mkfs` gives one inode per 8 blocks, at most 512 Ki inodes, so the on-machine `fsck` (one bit per block, three bytes per inode) checks a filesystem of several GiB within a process's address space. Still to do for very large cards: LBA48 (past 128 GiB), mount and partitions (several filesystems on one card).

**Directory entries (32 bytes):** inode number (32 bits) + a name of up to 28 characters. `.` and `..` in every directory.

### 5.2 What it supports

Hard links, `mkdir`/`rmdir`, `rename`, device files (`mknod`), pipes, permissions with root override, set-user-ID programs (`passwd`, `su`), `umask`, and **mounting** a second filesystem (`mount`, `umount`; for example the RAM disk on `/mnt`).

### 5.3 The host tool `p32fs`

A Python tool, so disk images are built from the repository by `make`: `p32fs mkfs IMAGE SIZE`, `p32fs put IMAGE host-file /path`, `p32fs get`, `ls`, `mkdir`, `chmod`, `chown`, `fsck` (checks every invariant: free map vs blocks in use, link counts vs directory entries, sizes vs blocks), and `p32fs build IMAGE tree/` to make a complete root filesystem (kernel in the boot area, `/bin`, `/etc`, `/dev`, `/usr`, `/tmp`, home directories) in one step. The same image boots in the emulator (`-d`) and is written to a real card with `dd`.

### 5.4 Crash safety

V6 style: writes are cached and flushed by `sync`, every 30 seconds and at `shutdown`; at every boot `/etc/rc` runs `fsck -q` (on the machine — a C port of the host checker) before going multi-user. `fsck -y` repairs; after repairing the mounted root it restarts the machine without writing the cache back (D28), as V7's fsck did. A journal (as xv6's log) is possible later, but adds complexity the first version does not need.

---

## 6. Users and security

- `/etc/passwd`: `name:password-hash:uid:gid:full name:home:shell`; `/etc/group`. **root** is uid 0 and bypasses permission checks.
- Every process carries a **real and effective uid/gid**; files carry an owner, a group and rwx bits for owner / group / others. The kernel checks them in `open`, `exec`, `chdir`, directory searches, `unlink`, `chmod`, `chown` (root only), `kill` (own processes, or root), `setuid` (root, or back to the real uid).
- **Memory protection** comes from the address spaces (§4.2): one user's processes cannot see another's memory, the kernel's, or the devices.
- **Passwords** are stored as a salted hash. P32W has no fast crypto, so the proposal is a deliberately simple keyed hash (several thousand rounds of a 32-bit mixing function, a 16-bit salt): fine for a hobby machine on two serial lines, not a defence against a stolen disk. `/etc/passwd` can be readable (as in V7) or the hashes moved to a root-only `/etc/shadow`.

---

## 7. The commands

**Phase 1 — a usable system:** `init` `getty` `login` `sh` `ls` `cat` `echo` `cp` `mv` `rm` `ln` `mkdir` `rmdir` `pwd` `chmod` `chown` `chgrp` `ps` `kill` `wc` `grep` `head` `tail` `sync` `date` `who` `passwd` `su` `ed` `mknod` `stty` `true` `false` `sleep` `shutdown`

**Phase 2:** `more` `sort` `uniq` `od` `diff` `cmp` `find` `du` `df` `tee` `tr` `cut` `test`/`[` `expr` `write` (to another user's terminal) `mail` `wall` `fsck` `mount`/`umount` `cc`? (a native compiler is far off; programs are compiled on the host).

*Status (2026-10-14):* both phases are built, and much more (about 85 commands, with `vi`, `make`, `basic`, `sql`, `sed`, `bc` and a native `cc` since M7), except `chgrp`, `mknod` (the system call is there, the command is not), `write`, `mail` and `wall`.

**The shell** (`sh`): V6/Thompson-shell features first — commands with arguments, `<` `>` `>>` redirection, pipelines `|`, `;`, background `&`, `cd`, `exit`, `$PATH` search (`/bin`, `/usr/bin`), wildcards `* ? [...]`; later variables, `if`/`while`, and scripts (Bourne-style subset). *Status:* all of that and more is in: functions, `case`, here-documents, subshells, `#!` scripts, history and Tab completion (`sh(1)`).

`ed` is the editor: it needs no screen control and works on a serial terminal, as on V6.

---

## 8. Emulator support

- `p32emu` (instruction level, fast) is the development platform; the microcode-level `p32uemu` checks the same images more slowly.
- **Two terminals in the emulator:** UART 0 on the emulator's own terminal, UART 1 on a pseudo-terminal or a TCP port (`-2 tcp:2323`), so two people can log in at once on the Mac.
- **Scripted sessions for `make test`:** feed input to each UART from a file (with waits for prompts), record the output, compare it with a recorded transcript — the way `monitor.expect` works today.
- A per-process trace (system calls, faults, context switches) for debugging.

## 9. How it is tested

| Layer | Test (in `make test`) |
|---|---|
| Assembler objects, linker | link tests: cross-file calls and data, relocations at the limits, archives |
| C compiler | the C test suite against expected output and the host compiler |
| C library | unit programs in `p32emu` (strings, printf, malloc) |
| Kernel | scripted sessions on a fresh image: boot, login on both terminals, the commands, pipes, background jobs, permission refusals, a program that faults is killed while the others run, `fork` until memory is full and recover, `fsck` clean afterwards |
| Filesystem | `p32fs fsck` after every session; the kernel and the host tool must agree on every image |
| The real machine | the same image on a CF card; booting to `login:` is the bring-up goal |

## 10. Effort and phases

| Phase | Delivers | Size (rough) |
|---|---|---|
| **M1** | relocatable assembler, linker, `a.out` | ~1,000 lines Python |
| **M2** | `p32cc` (stages 1–3) and its test suite; the C library | ~5,000 lines Python, ~2,500 C |
| **M3** | kernel I: boot, console, processes, `fork`/`exec`/`wait`, the filesystem read path, `p32fs`, a shell on one terminal | ~5,000 lines C |
| **M4** | kernel II: writing files, pipes, signals, both terminals, users and permissions, `init`/`getty`/`login` | ~3,000 lines C |
| **M5** | the phase-1 commands, `ed`, the on-machine `fsck`; manual pages | ~6,000 lines C |
| **M6** | copy-on-write `fork`, shared text, phase-2 commands, mount | done |
| **M7** | **the native toolchain** (D24): `as`, `ld`, `ar` and `cc` in C, running on P32W Unix; `cc` compiles itself there | ~8,000 lines C |

### M7 — the native toolchain (D24)

Programs are built on the host today (`cc/p32cc.py`, `assembler/`). M7 puts the toolchain on the machine, written in C and compiled by `p32cc`:

1. `as` — the assembler, reading the same syntax as `p32asm.py` and writing relocatable objects; `ld` and `ar` — the linker and archiver, with the same layout rules as `p32ld.py` (a.out, data on the next page). Objects and archives get

a compact **binary** format (the host's JSON is too bulky to parse on a 0.45 MIPS machine); the host tools learn to read and write it too, so either side can link the other's output.

2. `cc` — a C rewrite of `p32cc`, with the same design (token preprocessor, recursive-descent parser, the same code generator and ABI) so one test suite checks both; a driver runs `cpp` → `cc1` → `as` → `ld` with `/usr/include` and `/usr/lib/libc.a` on the disk.
3. **The test:** the host and native tools produce byte-identical objects and executables for every program in `cc/tests/` and the library; then **self-hosting** — `cc` compiles its own source on P32W Unix and the result is identical to the host-built `cc`.

Before it: keeping scalar locals in callee-saved registers in the code generator (both compilers), since compile time on the machine depends on the quality of the compiler's own code. **Done in `p32cc` (2026-10-14):** up to four scalar locals or parameters whose address is never taken (the most used, uses inside loops weighted ×8, at least 3) live in r13–r10; reads use the register in place, an assignment's result is not copied when unused, and a function that then runs short of temporaries is generated again with fewer. Session 1 (boot, login, 31 commands, halt): 50.3 M → 35.9 M instructions (−29 %); the kernel 62,344 → 57,172 bytes. `P32CC_NOREGVARS=1` turns it off, for comparisons. The C `cc` (step 3) does the same.

**Step 1 (as / ld / ar) done (2026-10-14):** objects and archives are binary (the format is specified in `assembler/p32obj.py`; the host tools write it and still read M1's JSON; `p32obj.py dump` shows one as JSON). `software/cmd/as.c`, `ld.c` and `ar.c` (with `objfmt.h`) are the host tools in C; the ISA table they use (`software/include/p32isa.h`) is generated from `p32isa.py` like `p32opc.h`. Checks: built on the Mac, the C tools give byte-identical output to the Python ones for all 104 assembly files of the system (libc, commands, kernel, tests), all 62 command links and libc.a; on P32W Unix (`software/tests/toolchain_test.sh`, in `make test`) they assemble, link and archive compiler output identically to the host, and the program runs. Speed on P32W (emulated time): `as` of `sort.s` (≈ 900 lines) 4.2 s, `ld` of `echo` against libc 5.2 s. (p32cc does not yet return structs from functions; `as.c` passes results by pointer.)

**Steps 2 and 3 done (2026-10-14): M7 is complete — P32W is self-hosting.** `software/cc` (`lex.c cpp.c type.c parse.c gen.c main.c`, ≈ 3,300 lines with `cc.h`) is `p32cc` in C, stage for stage; built on the Mac it writes the same assembly as `p32cc` for all 106 C files of the system and the same `-E` text. On P32W Unix `cc` (with `/usr/include`, `/usr/lib/crt0.o`, `/usr/lib/libc.a`) compiles, assembles (`as`) and links (`ld`) programs that run, and **compiles its own source into a `cc` byte for byte identical to the host-built one** (`software/tests/cc_test.sh`, in `make test`). Each `.c` file is compiled in a process of its own (as `cc1`), so each starts with a whole address space: the largest, `parse.c`, needed 3.1 MB of heap (1.6 MB since D29: each function is generated as soon as it is parsed, and its tree and tokens freed). Self-hosting costs about $1.0 \times 10^9$ instructions: 16 minutes at the emulators' 1 MIPS, about half an hour on the hardware.

Each phase ends with `make test` green and a push.

## 11. For review (D22)

1. **The ABI** (§3.3): arguments in r1–r4, result in r1, r8–r13 + r14 callee-saved, `char` signed.
2. **The user address space:** 8 MiB per process (D34) with a two-frame page table and a one-load TLB refill; page 0 unmapped.
3. **The C subset:** C89 without floating point, staged as in §3.3.
4. **Kernel model:** V6 — non-pre-emptive kernel, `sleep`/`wakeup`, copy on `fork` first (copy-on-write later), no swapping.

5. **Filesystem:** 1 KiB blocks, 64-byte inodes with 8 direct + single + double indirect, a free-block bitmap, 28-character names, no journal (`sync` + `fsck`).
6. **Users:** V7 passwd/group files, real/effective IDs, setuid programs, a simple salted hash for passwords.
7. **Timer at 50 Hz** for the scheduler and the clock; no real-time clock for now (`date` is set by hand, or by a later RTC).
8. **Start with M1 + M2** (the toolchain): nothing in the OS can start until C works.

(This list is the proposal as reviewed. Since then: copy-on-write `fork` and shared text are in (M6); a real-time clock is on the I/O card (D23), so the time survives power-off; the signal list is V7's, without `SIGCHLD`.)

# P32W Unix kernel

kernel/README.md

**Status:** M3 and M4 of D22 (os-design.md; using it: docs/unix.md). Boots from the CF disk in the emulator, mounts the root filesystem and runs `init`, which keeps a `getty` / `login` on each terminal. `make test` runs two recorded sessions: one user working with files, directories, links, redirection, pipelines, background jobs, `ps`, errors and `halt`; and two users logged in at once on the two terminals (a wrong password refused, permissions between users enforced, `su` with root's password, `passwd` refused for another user). After each, the host's `fsck` finds the disk clean.

```
make -C kernel            unix.bin: the kernel, linked at 0x0010_0000
make -C software disk     software/build/disk.img: the kernel in the boot area + the root filesystem
emulator/p32emu -w -d software/build/disk.img monitor.bin       then type b at the > prompt
```

(`monitor.bin`: `python3 assembler/p32asm.py software/monitor/monitor.asm -o monitor.bin`. `-w` waits for a prompt before each input line — for scripts; leave it out when typing.)

On P32W itself, `make` in `/usr/src/kernel` builds the same `unix.bin`, byte for byte (`Makefile.p32w`, D86), in about 20 minutes at 3 MHz. Only the host can put a kernel on the disk's boot area so far.

## Files

| File | What |
|---|---|
| `start.s` | entry; the trap vector with the TLB-refill fast path; `trapret`; `swtch`; wrappers for the privileged instructions |
| `main.c` | `kmain`: memory, processes, buffers, files, disk, filesystem, terminals, the clock, process 1 |
| `trap.c` | system calls, faults (stack growth, or a signal), interrupts (clock, UARTs) |
| `proc.c` | the process table, the scheduler, `sleep` / `wakeup`, `fork`, `exit`, `wait`, signals |
| `vm.c` | page frames; page tables; `copyin` / `copyout`; stack growth; `sbrk` |
| `exec.c` | loading an `a.out`; the argument stack; set-user-id |
| `syscall.c` | the system calls (V7 numbers, `sys/syscall.h`) |
| `fs.c` | the inode filesystem (tools/p32fs.py's format): inodes, block maps, directories, path names, permissions |
| `bio.c` | the buffer cache (48 × 1 KiB, LRU, delayed writes) |
| `cf.c` | the CompactFlash driver (16-bit PIO, D26; a block is one two-sector command): block device 0 (the filesystem area) and the card's layout |
| `file.c` | open files, pipes, the block- and character-device switches, `/dev/null` `/dev/port` `/dev/rdisk` |
| `rtc.c` | the real-time clock (D23): read at boot, set by `stime` |
| `usb.c` | a USB drive through the I/O card's CH376S (D43), polled: block device 2 (`/dev/usb`) and the raw sectors, character device 5 (`/dev/rusb`) |
| `tty.c` | the terminals (UARTs 0–3, and the video console `/dev/tty4`), interrupt-driven, with the V7 line discipline |
| `video.c` | the video card's screen and PS/2 keyboard, the console behind `/dev/tty4` (D53): commands to the card's HD63484, which draws the characters (D103, D104) |
| `kprintf.c`, `klib.c` | console messages, panics, string routines |

## How it works

- **Memory.** The kernel runs untranslated (supervisor addresses are physical). Free RAM after the kernel is a list of 4 KiB frames. Each process has **one page table**: 2,048 entries in `TLBLO` format, in two physically contiguous frames, covering user addresses 0–8 MiB (D34), so the TLB-miss handler in `start.s` refills with one load. Page 0 is never mapped; text is read + execute; data, bss, heap and stack are read + write; the stack grows on demand down to 1 MiB. A process slot's ASID is its index + 1, so a context switch is one `MTCR ASID`; the TLB is flushed only when a mapping is removed or a slot is reused.
- **Processes.** `fork` shares the address space **copy-on-write**: every frame has a reference count; writable pages become read-only + `PTE_COW` (a software bit the TLB ignores) in both processes, and the first write — a `PROTECT` fault, or the kernel's `copyout` — copies the page (or, for the last sharer, just makes it writable); text is shared outright. `exec` builds a fresh one and frees the old only when the new is complete. Each process has a 4 KiB kernel stack holding its trap frame. The kernel is **not pre-empted** (V6): a process in the kernel runs until it sleeps; the clock (50 Hz) only asks for a reschedule, which happens on the way back to user mode. The scheduler is round-robin; it idles with `WAIT`.
- **System calls.** `SYSCALL`, number in r1, arguments r2–r5, result r1 (−errno on failure). Every user pointer is checked through the page table.
- **Signals.** V7 numbers and semantics (a caught signal reverts to the default). A handler runs in user mode through the library's trampoline (`__sigtramp`): the kernel saves the trap frame on the user stack and `sigreturn` restores it. ^C / ^\ on a terminal signal every process on it.
- **Files.** V6/V7: an open-file table shared across `fork`, pipes (1 KiB), inodes cached and locked, a write-back buffer cache flushed by `sync` and every 30 s, `fsck` after a crash (no journal; os-design.md 5.4).
- **Devices** (Unix V6/V7): **block** devices (`bdevsw`: open, close, *strategy*) move 1 KiB blocks through the buffer cache — the CF card is block device 0, the root filesystem lives on its minor 0, and `/dev/disk` reads it through the cache (what `fsck` uses, so it sees what the kernel sees); **character** devices (`cdevsw`: open, close, read, write, ioctl) move bytes straight to the process — `/dev/tty0` `tty1`, `/dev/null`, `/dev/mem`, `/dev/port`, and `/dev/rdisk`, the raw card byte by byte (as V7's `/dev/rrk0` beside `/dev/rk0`). A new disk, a RAM disk or a partition is one more `bdevsw` entry; `mount` (M6) will use them.
- **Terminals.** Input by interrupt with echo, erase, line kill, ^C ^\ ^D, CBREAK and RAW modes (`stty`); output by interrupt from a queue.

## Still to come (M5–M6)

Signals in the session tests; the shell's `if` / `while`; `fsck` repairs (it checks now, at every boot, from `/etc/rc`); `mount`; booting the real machine.

# p32cc — the P32W C compiler

cc/README.md

The same compiler in C, which runs on P32W Unix and compiles itself there, is `software/cc` (M7, D24); the two write the same assembly (`make test`).

**Status:** M2 of the operating-system plan (D22, os-design.md). Compiles the C library and the test programs; every test program behaves exactly like the host compiler's build of it (`make test`).

```
python3 cc/p32cc.py -c file.c [-o file.o]       compile to an object
python3 cc/p32cc.py -S file.c                   assembly (file.s), with each statement's source line
python3 cc/p32cc.py -E file.c                   preprocessed source
python3 cc/p32cc.py --sa prog.c -o prog.bin     a stand-alone program for the emulators
emulator/p32emu prog.bin < /dev/null            run it (exit status = main's result)
```

`-I DIR`, `-D NAME[=VAL]`, `-w` (no warnings), `-M map` (link map).

## The language

C89, with floating point in software (P32W has no FPU): `char` (signed) / `short` / `int` / `long` (32 bits), `unsigned`, pointers, arrays, `struct`, `union`, `enum`, `typedef`, function pointers, `static` / `extern`, initialisers (nested, partial, strings, address constants), `switch`, `goto`, variadic functions (`stdarg.h`), the full preprocessor (`#include`, `#define` with `#` / `##` / `__VA_ARGS__`, `#if` / `#elif`, `defined`, `__FILE__` / `__LINE__`).

From C99: `//` comments, declarations anywhere in a block and in `for (int i = ...)`, `_Bool`, `inline` (accepted, no effect), **designated initialisers** (`[i] =`, `.member =`, chains like `.a.b[2] =`), brace elision in initialisers, **compound literals** (`(struct pt){ 1, 2 }`, at file scope a static object, in a function a fresh one each time it is evaluated) and `__func__`.

**Bit-fields**, laid out as System V does (from a unit's lowest bit, never across a unit the size of the field's type; `:0` ends the unit; plain `int` fields are signed), with `op=` and `++`/`--` (evaluating the struct's place once) and in initialisers.

**Structs by value**, as arguments and results (D30): a function with a struct parameter takes every argument on the stack, each struct copied into words of its own (as a variadic function does, so `va_arg(ap, struct pt)` works); a function returning a struct gets a hidden first argument, where to put it, and returns that address.

`long long` (64 bits, aligned to 4; D30): kept in memory like a struct, so it passes and returns as one; its operations are calls of libc's helpers (`libc/llong.c`: `__lladd` ... `__llcmp`), constants are objects in the data, and constant expressions fold at compile time. printf has `%lld`, stdlib `strtoll` and its family.

`double` (D30; `float` and `long double` are the same 8-byte IEEE type, which C permits): kept in memory like a long long, its operations calls of libc's helpers (`libc/dbl.c`, on softfp, so + - * / are correctly rounded and agree bit for bit with any IEEE machine). Comparisons follow C's NaN rules. Constants (`1.5`, `.5e-3`, `2.f`) are objects in the data and fold at compile time, except a division by zero or a NaN result.

**Not yet:** K&R-style definitions, hexadecimal floating constants. A use is a clear compile-time error. (In progress, D30: the C compiler upgrade.)

## How it works

| File | Stage |
|---|---|
| `clex.py` | tokens with file, line and spacing |
| `cpp.py` | the preprocessor, on tokens (hidesets stop recursive macros) |
| `cparse.py` | recursive-descent parser: a typed syntax tree, C89 conversions, constants folded |
| `cty.py` | types: sizes, alignment, struct layout, the usual arithmetic conversions |
| `cgen.py` | P32W assembly for `p32asm -c` |
| `p32cc.py` | the driver: assemble with `p32asm`, link with `p32ld` |

**Calling convention** (D22, D30): arguments 1–4 in r1–r4, more on the stack (argument 5 lowest); result in r1; r1–r7 caller-saved, r8–r13 callee-saved, r14 frame pointer, r15 stack pointer; `CALL` pushes the return address. A **variadic** function, or one with a struct parameter, gets every argument on the stack (a struct in as many words as it takes), so `va_list` is a plain pointer. A struct result goes where a hidden first argument points. Arguments are evaluated left to right.

**Code:** expressions are evaluated in a stack of temporaries (the lowest free of r1–r13); locals live in the frame (`off(r14)`); immediates are used where they fit; conditions compile to direct compare-and-branch. No optimiser yet.

## Tests

`cc/tests/*.c`, run by `cc/run_tests.py` (in `make test`): each program is built by p32cc and run in `p32emu`, and built by the host `cc` and run natively; output and exit status must match. A `NAME.expect` file makes a test P32W-only.

# USB thumb drive — design study (2026-10-16)

docs/usb-design.md

Status: **decided (D43); the emulator model and the driver are done** (2026-10-16). The model is the CH376S's sector mode in `emulator/p32io.h` (`-U`, `-Uout`; `emulator/test/usb_model.c`). The driver is `kernel/usb.c`: `/dev/usb` and `/dev/rusb`, usb(4), sessions 17 and 18. The FAT tools are done too (2026-10-17, D44): `software/cmd/dos.c`, dos(1), session 19, `software/tests/fat_test.sh`. The hardware is two CH376S ports on the I/O card (D50, architecture.md §3).

**Decided** (Ken, 2026-10-16; D43):

- USB goes on the **I/O card** (section 2): two ports, each a CH376S.
- **The CH376S is plan A**, with the Pico + TinyUSB as plan B.
- **FAT through user-space tools only** (`dosdir`, `doscp`, `dosrm`, `dosformat` on `/dev/rusb`). No FAT inside the kernel, so no VFS.

From Ken: "I also want to explore USB" — for **storage**: a thumb drive as a removable disk, to move files to and from the Mac and for extra storage or backups. Keyboards, hubs and P32W-as-a-USB-device are out of scope.

## 1. The controller

Three ways to give P32W a USB host port were compared. Sources: the WCH CH376 datasheet and its `CH376INC.H`, the Fuzix CH375/CH376 driver, and TinyUSB's `msc_host` and the Pico datasheet. Throughput figures are estimates: nobody publishes sector rates for either.

| | CH376S module | Pico (RP2040) + TinyUSB | bare MAX3421E |
|---|---|---|---|
| USB protocol runs on | the CH376S (fixed firmware) | the Pico (our firmware) | **P32W's CPU** |
| Bus interface | 8-bit parallel, 5 V, connects straight to the HCT bus | 3.3 V: latches, flags, level shifting, PIO firmware | SPI |
| Glue chips | 3–4 | 6–8 | — |
| P32W driver | ~150 lines: a command/status state machine | a protocol we define | a whole USB stack |
| Throughput (est.) | 150–250 KB/s | 300–500 KB/s (P32W is the limit) | — |
| Drive compatibility | fixed; known mount quirks; no hubs | fixable in firmware; hubs work | — |
| Cost | module about €6 | Pico about $5 + chips | — |

**Recommendation: the CH376S in parallel mode.**

- It sits directly on the 5 V byte bus. Its timing needs are tiny next to the I/O card's strobes: write pulse ≥ 30 ns, read pulse ≥ 40 ns, ≥ 1.5 µs after a command byte, ≥ 0.6 µs between data bytes. At about 1 µs an instruction, P32W meets those by itself.
- It needs little glue, and there are working drivers that use exactly this sector command set (Fuzix, RomWBW).
- It is simple to model in the emulators: two ports and a small state machine.

The Pico stays **plan B**, if too many drives turn out not to work with the CH376S. The driver keeps the controller behind one small interface, so changing to the Pico later changes the driver's bottom layer only. Ruled out: VNC2 (end of life), FT311D (Android accessories, not storage), MAX3421E (the USB stack would run on P32W).

## 2. Hardware

**On the I/O card** (D50): two USB-A ports, each a CH376S module, beside the UARTs, the timer, the RTC and the network coprocessor. The card holds, for USB:

- **each device's select** within the I/O card's window, byte accesses only, and `-ACK`, decoded with the card's other devices;
- **the byte lane** `D7–D0`, shared with the card's other byte devices;
- **the interrupt:** both ports onto `-IRQ6` (a jumper sets the level). The interrupt gate and the data buffer are **HCT**, because the module's outputs may reach only about 3.3 V;
- **two CH376S modules:** the common "CH376S U-disk module", with 2.54 mm headers and a USB-A socket, in socket strips (D21), the sockets at the card's front edge. The TXD-to-GND jumper selects parallel mode at reset.

**Registers** (in the I/O card's window, D52):

| Address | Read | Write |
|---|---|---|
| `0xFFFF_FF50` | USB 0: data byte | data byte |
| `0xFFFF_FF51` | USB 0: status (bit 7: interrupt pending, active low; bit 4: busy) | command byte |
| `0xFFFF_FF54` / `FF55` | USB 1: the same | the same |

The I/O card's interrupt status (`FFFF_FFF4`) has bit 6 for USB 0 and bit 7 for USB 1.

**From the datasheet's application circuit** (CH376DS1, section 8.1, "USB Flash Drive applications, 5V power supply"; Ken's copy):

- **Parallel mode:** TXD is tied to GND, the other mode pins are left unconnected. The signals are D0–D7 (pins 15–22), A0 (8), RD# (4), WR# (3), PCS# (27) and INT# (1).
- **RSTI (pin 2) is active high,** with a built-in pull-down: left open, the chip runs. (The research found the pin table contradicting this; the application circuit's text settles it.)
- **INT# drives only a weak high while the chip is in reset;** the datasheet recommends a 2–5 kΩ pull-up.
- **The parts around the chip** are the module's: a 12 MHz crystal, 0.01 µF on V3 (at 5 V), 0.1 µF + 100 µF on VCC, and the USB socket with VBUS fed from VCC through a small resistor (R2). The drive takes its current from the backplane's 5 V (budget 200 mA).

**The module** (its schematic, from Ken; it is drawn with a CH375, which has the same pinout):

- **A 16-pin header (8×2)** carries everything the parallel interface needs: D0–D7 on the odd pins 1–15, GND on 2 and 4, +5 V on 6, INT# 8, A0 10, CS# 12, RD# 14, WR# 16. Parallel mode: the 3-pin jumper P_S, pins 1–2, tying TXD to GND.
- **RSTI is not on the header.** The module has its own power-on reset (0.47 µF to its 3.3 V rail), so the card cannot reset the chip, and a P32W reboot without a power cycle does not reset it either. The driver sends `RESET_ALL` first, always.

- **The chip runs at 3.3 V,** from an LM1117-3.3 on the module (the USB socket gets +5 V directly). Its outputs reach about 3.3 V. That is enough for 74HCT inputs (2.0 V threshold), so the data buffer and the interrupt gate must be **HCT, not HC**. INT#'s pull-up must not go to 5 V.
- **Run at 5 V (Ken, 2026-10-16):** the module's LM1117 is bridged (input to output), or its 5 V/3.3 V jumper set if it has one, so the CH376S runs at 5 V as in the datasheet's own circuit (8.1). Its V3 pin already has that circuit's 10 nF to ground. Then the chip's outputs are full 5 V levels and its inputs take 5 V: no level shifting, and the question of 5 V tolerance at 3.3 V does not arise. (The HCT parts stay: they work either way.)

**To check on a real module before the I/O card's PCB is drawn:**

- the header pinout above, against the real board;
- whether it has a 5 V/3.3 V jumper, or the LM1117 must be bridged; and the datasheet's word on 5 V inputs at 3.3 V (its electrical section);
- that its P_S jumper is on pins 1–2 (parallel mode).

## 3. Software

1. **An emulator model** of the CH376S in `p32io.h` (so in both emulators), backed by an image file: `-U usb.img` plugs a drive in, and a key or option pulls it out. It runs the status codes and the 64-byte-chunk protocol below, so the driver is written and tested before the card exists, as for the RTC (D23).
2. **A block driver,** `bdevsw` 2 (`/dev/usb`) and a character device (`/dev/rusb`). A 1 KiB block is two 512-byte sectors.
   - **Initialise:** `RESET_ALL`; `CHECK_EXIST` (0x55 → 0xAA); `SET_USB_MODE` 7, then 6 (host).
   - **When a drive is plugged in** (`USB_INT_CONNECT`): `DISK_INIT`, which Fuzix uses because `DISK_MOUNT` may reject a stick that isn't FAT; retry about 5 times. Then `DISK_CAPACITY`.
   - **Read:** `DISK_READ` with the LBA (4 bytes) and a count. Then, eight times per sector: wait for `USB_INT_DISK_READ`, `RD_USB_DATA0`, read 64 bytes, `DISK_RD_GO`. Finish on `USB_INT_SUCCESS`.
   - **Write:** the same, with `DISK_WRITE`, `WR_HOST_DATA` and `DISK_WR_GO`.
   - **Timeouts:** every wait has one, so a drive pulled out mid-transfer gives `EIO`, never a hang.
   - **Removal:** a disconnect discards the drive's cached blocks, and later accesses fail until a drive is plugged in again.
   - Polled at first, as the CF driver is. `IRQ6` is used for plug and unplug, and later to sleep instead of poll.
3. **P32W filesystem on a stick:** `mkfs /dev/usb`, then `mount /dev/usb /mnt` (mount and umount exist already). `sync` or `umount` before pulling it out.
4. **FAT tools** for the Mac, mtools-style, in user space on `/dev/rusb`: `dosdir`, `doscp` (both ways), `dosrm`, `dosformat`. FAT32 first; FAT16 for small sticks.
5. **Not planned:** FAT mounted directly (it would need a filesystem switch, a VFS, in the kernel; Ken chose the tools). Plan B, the Pico, only if drive compatibility is poor.

**Speed (estimate):** about 150–250 KB/s, against about 800 KB/s from the CF card (D37). A 1.5 MB file would take under ten seconds.

## 4. Tests

- **The emulator model under** `make test`: `mkfs`, `mount`, copy files both ways, `fsck`, `umount`.

- **Pulling the drive out** mid-copy: the copy fails with an error, the system carries on, and plugging the drive back in works.
- **FAT tools:** a FAT32 image made on the Mac (`newfs_msdos`/`hdiutil`) is read on P32W, and one written on P32W is read back by the Mac's tools, as the `zip` tests do with Finder's archives.
- **On the bench:** a handful of real sticks of different makes and sizes, noting which ones mount.

## 5. Questions for Ken

1. ~~Where does USB go?~~ **Decided: on the I/O card**, two ports (D50).
2. ~~CH376S or Pico?~~ **Decided: the CH376S is plan A**, the Pico plan B.
3. ~~FAT tools or a FAT filesystem?~~ **Decided: user-space tools only.**
4. **Buy one module** to check its pinout before the PCB layout?

# Internet over Ethernet with the W5500 — design study (2026-10-17)

docs/net-design.md

Status: **decided (D45, 2026-10-17): B, the Pico coprocessor, and Ethernet before SLIP.** The hardware: a Pico 2 W and a W5500 on the I/O card, for Ethernet and Wi-Fi (D50, architecture.md §3); the mailbox is net-mailbox.md.

From Ken: "analyze adding internet connectivity, TCP/IP, to the system using the W5500", then "it might need a microprocessor as a coprocessor". This study weighs both. It extends the backlog's "Networking: SLIP first, then Ethernet" (2026-10-14), whose last step already named the W5500.

## 1. What the W5500 is

WIZnet's W5500 is an Ethernet controller with the **TCP/IP stack built into the silicon** (WIZnet calls it "hardwired TCP/IP"):

| | |
|---|---|
| Network | 10/100 Ethernet MAC and PHY on the chip; an RJ45 jack with magnetics outside |
| Protocols in hardware | TCP, UDP, IPv4, ICMP (it answers ping itself), ARP, IGMP, PPPoE |
| Sockets | **8**, each opened as TCP, UDP, IP-raw, or (socket 0 only) MAC-raw |
| Buffers | 32 KiB on the chip: 16 KiB send and 16 KiB receive, shared among the sockets (2 KiB each by default) |
| Interface to the host | **SPI only** (modes 0 and 3, up to 80 MHz), plus an interrupt pin (INTn) and reset (RSTn) |
| Supply | 3.3 V, about 130 mA at 100 Mb/s. Its inputs are **5 V tolerant**; its outputs reach 3.3 V |
| Package | LQFP-48, surface mount, so it comes on a **module** (D21) |
| Not in it | IPv6, DHCP, DNS, TLS. DHCP and DNS are small programs over its UDP sockets |

**The model of use.** The host writes a socket's mode and port into its registers, then gives it a command: `OPEN`, `LISTEN`, `CONNECT`, `SEND`, `RECV`, `DISCON` or `CLOSE`. Data goes through each socket's ring buffer in the chip, with read and write pointers the host moves itself. The chip does the handshakes, retransmissions, windows, ARP and checksums, and raises INTn when something happens (connected, data arrived, sent, disconnected, timeout).

**An SPI transfer** is 3 header bytes (a 16-bit address, then a control byte naming the register block and read or write) and then any number of data bytes, which go to consecutive addresses. So a 1460-byte TCP segment costs about 1463 SPI bytes, and the overhead of the pointer and command registers is a few dozen bytes per packet.

**Two ways to use it:**

1. **Socket offload** (TCP and UDP sockets): P32W never sees a packet, only streams and datagrams. Small and fast on P32W's side.
2. **MAC-raw** (socket 0): the W5500 is just an Ethernet card, sending and receiving whole frames, and P32W runs its own TCP/IP stack (ARP, IP, ICMP, UDP, TCP). That is the "KA9Q route" of the backlog's SLIP plan.

## 2. The numbers that decide it

**P32W's speed:** about **0.5 MIPS** (D14: a 3 MHz microcycle). Moving a byte through an I/O register costs a few instructions (load, store, count, branch). The CH376S driver's figure, an estimate since the emulators answer at once, is **150–250 KB/s** for a polled byte port. That is the ceiling for any design in which P32W moves every byte itself. It is far below Ethernet's 12 MB/s, so the wire is never the limit.

**Compared with SLIP** at 115,200 baud: about 11 KB/s. So any W5500 design is roughly **10–20 times faster** than the SLIP plan.

**A software TCP/IP stack on P32W** (MAC-raw) adds per byte at least a checksum (an add per 16 bits, done twice: in and out of the stack) and a copy or two. A rough estimate is **30–60 KB/s**, in line with KA9Q on a PC/XT. It also needs about 5,000–8,000 lines of kernel C, and its timers (retransmission, ARP) run in the kernel.

**TLS (HTTPS)** is where P32W runs out. A TLS handshake does an elliptic-curve key exchange and checks a certificate chain. On P32W that is on the order of 10^8 instructions or more, which is **minutes per connection** (an estimate), before the cipher itself costs time on every byte. Today most of the internet is HTTPS only: web pages, package servers, GitHub. **Without TLS, P32W can reach the local network, plain HTTP, telnet, FTP, NTP, DNS, and HTTPS only through a proxy on another machine.**

## 3. Three ways to build it

The kernel side above the hardware (sockets as file descriptors, the classic system calls) is the same in all three: section 5.

### A. A TTL SPI master; P32W drives the W5500 itself

The W5500 module sits on the I/O card behind a small **SPI master made of logic**:

- an 8-bit shift register, a 74HCT299 (or a '595 to send and a '165 to receive), loaded by a write and read back by a read;
- a 74HCT161 counting the 8 bits and a flip-flop for "busy", or all of it in a GAL (an ATF22V10 holds the 3-bit counter and the control);
- SCLK from the bus clock: at 3 MHz, a byte shifts in under 3 µs, about one P32W instruction, so P32W never has to wait;
- chip select from a register bit, so P32W holds SCSn low across a burst;
- no level shifting: the W5500 takes 5 V inputs, and its 3.3 V MISO and INTn are a valid high for HCT inputs (2.0 V);
- a 3.3 V regulator for the module (an LM1117-3.3 in TO-220).

About **5–7 chips** plus the module. **Registers:** SPI data (write sends a byte; read returns the byte received), SPI control (chip select, reset), and status (busy, INTn). **IRQ7**.

**Software on P32W:**

- the W5500 driver in the kernel, about 800 lines: socket commands, the ring buffers, the interrupt;
- DHCP and DNS as small user programs (`dhcp`, and a resolver in libc) over UDP sockets.

**Good:**

- the fewest parts, and no second program to maintain;

- all of it is P32W's own code: in keeping with a machine built to be understood;
- MAC-raw is there too, for a P32W stack of its own later (route 2 above).

**Limits:**

- 8 sockets at most;
- the W5500's TCP has no listen backlog: accepting a connection uses up the listening socket, and the server must open a new one;
- no TLS;
- P32W moves every byte (150–250 KB/s at best).

### B. A coprocessor: an RP2040 with the W5500 (Ken's suggestion)

A microcontroller sits between P32W and the W5500. The neatest part is WIZnet's **W5500-EVB-Pico**: a Raspberry Pi Pico board (RP2040, 133 MHz, 264 KiB of RAM) with a W5500 and an RJ45 jack on the same board, on the Pico's 2.54 mm pins. The debug card's host link is a Pico too (D50), so the tools and the 3.3 V-to-5 V interfacing are shared.

**P32W talks to it through a mailbox,** like the CH376S's command and data ports. Its commands are network operations, not W5500 registers:

- `socket`, `connect`, `listen`, `accept`, `send`, `recv`, `close`;
- `dns` (a name to an address) and `status` (link, address from DHCP);
- possibly `tls-connect` (below).

**The firmware** runs WIZnet's ioLibrary (DHCP, DNS, the W5500 driver) or lwIP over MAC-raw, in C with the Pico SDK.

**The mailbox hardware** needs care, because the Pico is 3.3 V and **not 5 V tolerant**, and firmware cannot answer a 333 ns bus cycle in time. So the mailbox is latches, not a live bus:

- P32W to Pico: a 74HCT574 latch that P32W writes. Its 5 V outputs reach the Pico through CD74HC4050 buffers (DIP-16, run at 3.3 V, 5 V-tolerant inputs).
- Pico to P32W: the Pico drives a second 74HCT574's inputs at 3.3 V (a valid HCT high), and P32W reads it.
- A GAL keeps the "full" and "empty" flags and raises IRQ7.

About **6–8 chips** plus the board.

A refinement: **FIFOs instead of single latches** (IDT7201-style, 512 × 9, DIP-28, one per direction), so P32W can move a block without checking a flag between bytes. Their availability needs checking: the 720x family is old.

**Good:**

- **TLS becomes possible.** The RP2040 can run mbedTLS: slowly by PC standards (a handshake in a second or two), but feasible. P32W could then fetch real `https://` URLs, with the certificates checked on the coprocessor. **This is the one thing A cannot do.**
- DHCP leases, DNS, ARP and TCP keepalives are timed by the coprocessor, not P32W's kernel.
- The kernel driver is the simplest of the three (like `usb.c`).
- The coprocessor can be reprogrammed, so protocols can be added later without touching the card.

**Limits:**

- **A second program to write and keep in step,** in another language and toolchain. Its emulator model must speak the same mailbox protocol.
- **P32W still moves every byte through the mailbox,** so throughput is about the same as A (150–250 KB/s at best). A coprocessor saves P32W register bookkeeping, not byte moves.
- The coprocessor is far more powerful than P32W, so "the computer" does less of the networking. (The CH376S already has a microcontroller inside, and so does the W5500's state machine.)

### C. A 5 V microcontroller: an AVR with a W5500 module

The same idea as B with an **ATmega1284P** (DIP-40, 5 V, 20 MHz, 16 KiB RAM, through-hole like the rest of P32W) and a WIZ850io module:

- **no level shifting on P32W's side:** it can sit on the 5 V bus behind plain HCT latches;
- MISO from the W5500 goes through one HCT buffer, because 3.3 V is only just a high for a 5 V AVR;
- programmed with avr-gcc.

**Good:** the most "in period" part: fully through-hole and 5 V.

**Limits:** too small and slow for TLS. Without TLS it offers little that A does not, at the cost of the second program.

### Summary

| | A: TTL SPI master | B: RP2040 + W5500 | C: AVR + W5500 |
|---|---|---|---|
| Chips on the card (plus the module) | 5–7 | 6–8 | 5–7 |
| Second program to maintain | no | yes (C, Pico SDK) | yes (C, avr-gcc) |
| Kernel driver | about 800 lines | about 400 lines (mailbox) | about 400 lines |
| Throughput (P32W moves every byte) | 150–250 KB/s | the same | the same |
| HTTPS from P32W | no (a proxy) | **yes, on the coprocessor** | no |
| DHCP, DNS | user programs on P32W | on the coprocessor | on the coprocessor |
| A P32W TCP/IP stack later (MAC-raw) | yes | yes (frames through the mailbox) | possible, slow |
| Level shifting | none | the Pico's side | MISO only |
| Through-hole (D21) | yes, plus the W5500 module | yes, plus the Pico board | yes, plus the W5500 module |

**My reading:**

- If the goal is **the internet as it is today** (web pages, downloads, GitHub), the deciding fact is TLS, and that points to **B**. P32W keeps everything above the socket: the system calls, the programs, the shell scripts.
- If the goal is **a network P32W runs itself** (the local network, a web server for P32W's own pages, telnet, plain HTTP, with HTTPS through a proxy as the backlog already planned), **A** is smaller, has nothing extra to program, and does not need a microcontroller.
- **C** sits between them without TLS, so it has the least to recommend it.

So yes, a coprocessor is needed, but only for TLS. For TCP/IP itself the W5500 already is the coprocessor.

## 4. The emulator first, as with USB

All three can be built and tested in the emulators before any hardware, and with **real network access**. The model maps P32W's sockets onto the Mac's own sockets: a W5500 `CONNECT` becomes a `connect()` on the Mac, a `LISTEN` a listening socket on a host port.

- **For A:** a W5500 register model behind the SPI registers, with the socket commands done by the Mac. It is large-ish, but the W5500's socket model is simple.
- **For B:** a model of the mailbox protocol, even simpler, since its commands are already socket operations. TLS in the model can use the Mac's TLS library (Python's `ssl`, or `-lssl` in C).

Then P32W Unix can fetch a web page from the emulator on the first day. The tests can run against a small server the test script starts on the Mac, so `make test` needs no internet.

## 5. P32W Unix's side (the same for A, B and C)

- **Sockets as file descriptors:** a new `FD_SOCKET` type beside `FD_INODE` and `FD_PIPE`, so `read`, `write` and `close` work on them, and shell redirection and `cat` work with the network.
- **System calls:** `socket`, `connect`, `bind`, `listen`, `accept`, `send`/`recv` (or just read and write), `shutdown`, `getsockname`, `setsockopt` in a small form; BSD-compatible, so ported programs work. A way to wait on several descriptors at once (`select` or `poll`) is needed for telnet and a server that handles more than one client.
- **A network interface layer underneath,** so the SLIP plan (with P32W's own stack) could become a second interface later, as the backlog wants.
- **Programs, in the backlog's order:**
  - `ifconfig` / `netstat` (small);
  - `ping`;
  - a DNS resolver in libc (`gethostbyname`);
  - `telnet` and `telnetd` (on a pseudo-terminal);
  - `httpd` (static pages and CGI);
  - `ftp`/`tftp` and NTP (the clock set at boot);
  - with B, a `fetch URL` command, http and https.
- **Limits to accept:** 8 sockets on the W5500, each listening port using one. That is enough for telnetd plus httpd plus a few clients, not for a busy server.

## 6. The hardware, wherever it goes

- **On the I/O card** (D50): next to the CH376S ports, with **IRQ7** and registers `FF52` / `FF53` in the card's window (net-mailbox.md).
- **Modules, in sockets (D21):** WIZ850io (W5500 with an RJ45 jack and magnetics, 2 × 6 pins, 3.3 V) for A or C; W5500-EVB-Pico (2 × 20 pins, powered from 5 V through its own regulator) for B. **Before the PCB,** check the module in hand: pinout, supply, and the RJ45 jack's height against the card's front edge, where the jack sits.
- **Power:** about 130 mA more for the W5500 (plus about 30 mA for an RP2040), at 3.3 V from a regulator on the card or on the board.
- **The jack at the card's edge,** as the USB socket is: both modules have it at one end, so the layout puts that end at the front edge.

## 7. Work, in order (once the questions are answered)

1. Ken's choice of A, B or C, and of the order relative to SLIP.
2. The emulator model (both emulators), mapped to the Mac's sockets.
3. Sockets in the kernel, and the driver for the chosen interface.
4. The programs: `ifconfig`, `ping`, the resolver, `telnet`, `fetch`, then `telnetd`, `httpd`, NTP.
5. With B: the coprocessor firmware, tested against the emulator's model on the Mac's side first (the firmware's logic can build on the Mac).
6. The card: part of the I/O card's netlist and GALs, through the generators, after a real module is checked.

## 8. Questions for Ken

1. **A, B or C?** In short: does P32W need HTTPS itself (B), or is "the network P32W runs, HTTPS through a proxy" enough (A)?
2. **SLIP first, as the backlog says, or Ethernet first?** With A or B, the W5500 does TCP, so SLIP would mean writing P32W's own stack as a separate project (still worthwhile for learning, and needed for MAC-raw).
3. ~~**B only:** the W5500-EVB-Pico (one board), or a separate Pico and WIZ850io?~~ **Decided (D50): a Pico 2 W and a W5500, for Ethernet and Wi-Fi.**
4. **Buy a module now** to check its pinout and size before the card's layout, as for the CH376S?

# The network coprocessor's mailbox (D45)

docs/net-mailbox.md

The interface between P32W and the network coprocessor (a Pico 2 W with a W5500, on the I/O card, D50). Three programs implement it and must agree on it:

- the emulators' model (`emulator/p32io.h`, `-N`), which carries it out with the Mac's own sockets;
- the kernel's driver;
- the coprocessor's firmware.

This document is their common reference. Version 1, 2026-10-17.

## 1. Registers

Two byte registers in the I/O card's window (D52), after the first CH376S's. Byte accesses only, as for the card's other byte devices. The I/O card's interrupt status (`FFFF_FFF4`) has bit 8 for the coprocessor.

| Address | Read | Write |
|---|---|---|
| `0xFFFF_FF52` `NDATA` | the next reply byte (reading takes it) | the next request byte |
| `0xFFFF_FF53` `NSTAT` / `NCTRL` | status | control |

**Status** (`NSTAT`, read):

| Bit | Name | Meaning |
|---|---|---|
| 0 | `RXF` | a reply byte is waiting in `NDATA` |
| 1 | `TXE` | the request latch is empty: P32W may write `NDATA` |
| 2 | `ATTN` | the coprocessor has events (section 4); with `IEN`, `IRQ7` is asserted |
| 3 | `IEN` | interrupts enabled (as written) |
| 6 | — | always 0 |
| 7 | `READY` | the coprocessor is running (its firmware has started) |

**Control** (`NCTRL`, write):

| Bit | Meaning |
|---|---|
| 3 | `IEN`: assert `IRQ7` while `ATTN` is set |
| 7 | `RESET`: restart the coprocessor (the Pico's RUN pin) and empty both latches; `READY` drops until it has started again |

**The latches.** On the card, each direction is one byte latch with a full/empty flag, kept in a GAL: writing `NDATA` fills the request latch, and the Pico taking the byte empties it again; the Pico filling the reply latch sets `RXF`, and P32W reading `NDATA` clears it. The Pico answers within a microsecond, which is less than one P32W instruction, so a driver that checks `TXE` and `RXF` before each byte almost never waits. The model answers at once, as all of the emulators' devices do: software must depend on the bits only, never on timing.

**The I/O card's side** (D50): the request latch reaches the Pico through a 3.3 V buffer it enables with `-PRD`; the Pico clocks the reply latch with `PWR`. A GAL clocked by P32W's `CLK` keeps `TXE` and `RXF` and acts on each strobe's leading edge, once. `RESET` is a one-clock pulse on the Pico's RUN, stretched by an RC to about 15 µs.

**Without the I/O card,** the device window does not answer at these addresses (`BUSERROR`). **With the card but no running coprocessor,** `READY` is 0. The emulators give the second case without `-N`.

## 2. Messages

P32W sends a **request** and reads the **reply**, then sends the next request; there is exactly one reply to each request, and nothing else ever arrives in `NDATA`. The two have the same 4-byte header:

| Byte | Request | Reply |
|---|---|---|
| 0 | command | status (section 5) |
| 1 | socket number (0 if none) | socket number (the new one, for `SOCKET` and `ACCEPT`) |
| 2, 3 | payload length, low byte first | payload length, low byte first |
| 4 … | payload | payload |

Payloads are at most **1024 bytes** each way. Addresses and ports in payloads are in **network order** (big-endian), as in `sockaddr_in`: an address is 4 bytes, a port 2.

A driver that loses its place (a P32W reset in the middle of a message, say) sets `RESET` and starts again with `HELLO`.

## 3. Commands

| Code | Name | Request payload | Reply payload |
|---|---|---|---|
| `01` | `HELLO` | — | the text `P32NET 1` (the protocol's version). Closes every socket |
| `02` | `IFINFO` | — | 24 bytes: flags (bit 0 link up, bit 1 address configured, bit 2 by DHCP), 0, MAC (6), address, netmask, gateway, DNS server (4 each) |
| `03` | `IFCONFIG` | mode (1 byte: 0 DHCP, 1 static), then address, netmask, gateway, DNS server | — |
| `04` | `EVENTS` | — | 2 bytes: a bit for each socket whose state has changed since the last `EVENTS` (bit n = socket n), low byte first. Clears `ATTN` |
| `10` | `SOCKET` | type (1 byte): 1 TCP, 2 UDP, 3 TLS, 4 DNS lookup, 5 ICMP (the W5500's IP-raw mode, protocol 1: `SENDTO` and `RECVFROM` carry whole ICMP messages, P32W builds them, the port is 0) | — (the reply's socket byte is the new socket) |
| `11` | `CONNECT` | TCP, TLS: address, port; TLS also the host name, for SNI and the certificate check. DNS: the name to look up | — (started: section 4) |
| `12` | `BIND` | port (0: any) | — |
| `13` | `LISTEN` | backlog (1 byte) | — |
| `14` | `ACCEPT` | — | the peer's address and port; the reply's socket byte is the new, connected socket. `AGAIN` if none is waiting |
| `15` | `SEND` | the data | 2 bytes: how many were taken (`AGAIN` if none could be) |
| `16` | `RECV` | 2 bytes: the most to return | the data. `AGAIN` if none is there yet; `OK` with none at the end of the stream. DNS: the 4-byte address |
| `17` | `SENDTO` | UDP: address, port, the data | 2 bytes: how many were sent |
| `18` | `RECVFROM` | UDP: 2 bytes, the most to return | the sender's address and port, then the data. `AGAIN` if none |
| `19` | `SHUTDOWN` | — | — (TCP, TLS: no more sending; the peer sees the end of the stream) |
| `1A` | `CLOSE` | — | — (the socket is free again) |

| Code | Name | Request payload | Reply payload |
|---|---|---|---|
| `1B` | `STATE` | — | 14 bytes: state, error, bytes ready to read (2; a listening socket: connections waiting), room to send (2), local port (2), peer address (4), peer port (2). The four 2-byte counts and ports here are numbers, low byte first; the address is in network order |

The coprocessor has **8 sockets**, numbered 0 to 7 (one for each W5500 socket). A TLS socket takes one, and a DNS lookup takes one while it runs.

**Socket states** (`STATE`'s first byte):

| Value | State |
|---|---|
| 0 | free |
| 1 | new (made, not yet used) |
| 2 | connecting (TCP and TLS: the handshakes; DNS: the lookup) |
| 3 | connected (DNS: the answer is ready) |
| 4 | listening |
| 5 | closed by the peer (data may still be waiting to be read) |
| 6 | failed (`STATE`'s second byte is the error, a status code) |
| 7 | bound (UDP: ready to send and receive) |

## 4. Events and waiting

`CONNECT` only starts a connection, because a handshake can take seconds and the mailbox is shared by every socket. The coprocessor sets `ATTN` when anything a process might be waiting for changes on any socket:

- a connection made or failed;
- a DNS answer;
- data arriving;
- room to send again;
- a connection waiting to be accepted;
- the peer closing.

The driver then sends `EVENTS` and wakes the processes waiting on those sockets, which look again with `STATE`, `RECV` or `ACCEPT`. Without interrupts, the driver can poll `ATTN`.

## 5. Status codes

The reply's first byte. The driver turns them into P32W's errno values.

| Code | Name | errno | Meaning |
|---|---|---|---|
| 0 | `OK` | | done |
| 1 | `AGAIN` | `EAGAIN` | nothing yet: no data, no room, no connection waiting |
| 2 | `INVAL` | `EINVAL` | a bad request: no such socket, or not in that state |
| 3 | `NOSOCK` | `ENOBUFS` | all 8 sockets are in use |
| 4 | `REFUSED` | `ECONNREFUSED` | connection refused |

| Code | Name | errno | Meaning |
|---|---|---|---|
| 5 | `TIMEDOUT` | `ETIMEDOUT` | no answer |
| 6 | `UNREACH` | `EHOSTUNREACH` | no route to the host |
| 7 | `RESET` | `ECONNRESET` | the connection was reset by the peer |
| 8 | `NOTCONN` | `ENOTCONN` | not connected |
| 9 | `INUSE` | `EADDRINUSE` | the port is in use |
| 10 | `NONAME` | (the resolver's "host not found") | DNS: no such name |
| 11 | `TLS` | `ECONNABORTED` | the TLS handshake failed, or the certificate is not trusted |
| 12 | `NOLINK` | `ENETDOWN` | no network: the link is down, or no address yet |
| 13 | `TOOBIG` | `EMSGSIZE` | a payload over 1024 bytes |
| 14 | `BADCMD` | `EINVAL` | an unknown command |

## 6. The emulators' model (`-N`)

The model carries the commands out with the Mac's sockets, so P32W in the emulator really reaches the network:

- **The interface it reports:** address 10.0.2.15/24, gateway 10.0.2.2, DNS server 10.0.2.3, MAC 02:50:33:32:00:01, link up, by DHCP. These are QEMU's user-networking numbers.
- **10.0.2.2 is the Mac** (connections to it go to 127.0.0.1), so tests can run their own servers. Other addresses are reached as they are.
- **Listening:** P32W port `p` becomes the Mac's `p + 8000` when `p` is below 1024 (80 is 8080, 23 is 8023), else `p`; `-Nport OFFSET` changes the 8000. On the Mac's loopback only.
- **DNS** uses the Mac's resolver (IPv4 answers).
- **TLS** goes through `tools/p32tls.py` (Python's `ssl`: the Mac's certificate store, SNI, and the host-name check). `-Nca FILE` gives another set of trusted certificates, so tests can use a certificate of their own.
- **No loopback:** P32W cannot connect to its own address, as on the W5500.
- **Waiting:** while a socket is busy, an idle P32W (`WAIT`) waits in real time for the Mac's sockets as well as the timer, so emulated time does not run ahead of the network.