Emulating a Synthesizer Before It Arrives

Recovering an undocumented CPU from its firmware and running the M-VAVE FM-1 in a browser

Joakim Eriksson

https://github.com/joakimeriksson/fm1-emu

Technical report · 13 October 2026, with Section 9 on the real hardware added 24 October and extended 26 October 2026

Abstract

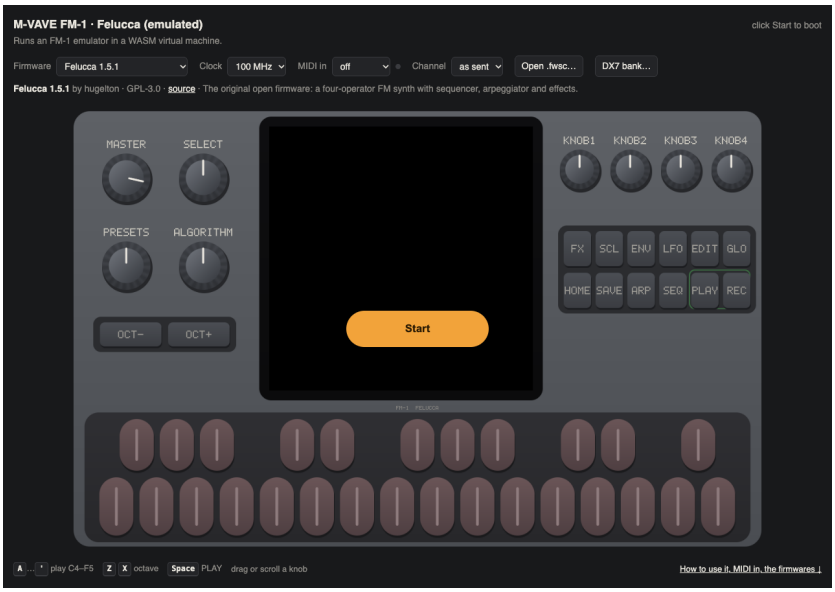
The M-VAVE FM-1 is a pocket FM synthesizer for which a small community writes its own firmware. It is built on a JieLi AC791N, a dual-core system-on-chip whose CPU, pi32v2, has no public instruction-set manual. This report describes, step by step, how an emulator of the whole instrument was built in five days, before the physical unit had been delivered: recovering the instruction set from vendor disassembly of 25 firmware images, solving operand encodings as exact linear models, validating semantics with differential probes in which the vendor compiler's own constant folding is the oracle, modelling the peripherals the firmwares actually touch, and making it fast enough for real time with two just-in-time compilers (AArch64 natively, WebAssembly in the browser). The emulator runs M-VAVE's stock firmware on both cores and fifteen community firmwares unmodified, takes MIDI in, and is published as a web page. Every performance change was accepted only if the emulated audio stayed byte-identical across three execution modes. The emulator was then used as the development platform for a new firmware, *FM-1 SID*, which runs a Commodore 64 tune's 6502 code and a reSID model of the SID chip on the FM-1's CPU. The work was done by one person directing an AI coding agent; the report closes with how that collaboration was organised and which rules made its output trustworthy. A section added once the hardware arrived tells what releasing FM-1 SID before anyone had run it on an FM-1 cost its first users, and how it was made to work on the device in four rounds. Their tool is *fm1-dock*, a host firmware installed once, through which synths are deployed and probes are run from RAM. The probes compare the emulator with the real FM-1, and their findings changed the emulator: CPU guards, LCD timing, a measured cycle and flash-cache model, and a survey of what 80 firmwares execute.

1 Introduction

The M-VAVE FM-1 (Fig. 1) is a battery-powered six-operator FM synthesizer with a 240×240 colour screen, 27 keys, 14 buttons, 7 encoders, a MIDI IN jack and stereo audio. Its stock sound engine is a port of the msfa/Dexed DX7 core. Because its update channel is USB-MIDI SysEx with only a CRC check, and because the community had documented its hardware and published a GPL hardware layer (Felucca), people began writing replacement firmwares within weeks of launch: drum machines, grooveboxes, ambient generators, a chord harp.

The goal of this project was an emulator accurate enough to run those firmwares *unmodified*, so that anyone can try a firmware before flashing it, a developer can write one without the synth on the desk, and the firmwares can be played in a browser. Three facts shaped the work:

1. **The CPU is undocumented.** pi32v2 is JieLi's proprietary 32-bit ISA with Blackfin-style algebraic assembly (`r0 += r1`, `if (r2 != 0) goto L`), variable-length encodings of 2, 4 and 6 bytes, dual-issue pairs, predicated blocks and hardware loops. No manual exists; the existing Ghidra module is described by its author as "very early stage".
2. **The vendor toolchain is public.** JieLi distributes a Clang/LLVM 4.0.1 fork with a pi32v2 back end and an `objdump` that disassembles it. It runs only on Linux x86-64; here it ran on an Apple-silicon Mac through `docker run -platform linux/amd64`.
3. **There was no hardware.** The FM-1 had been ordered but had not arrived. Everything had to be established from published firmware images, the toolchain and the community's notes.



(a) The web page: the panel is drawn by the emulator's own code.



(b) The emulated LCD: Felucca 1.5.1 playing a chord, 3.0 s after boot.

Figure 1: The emulator in use. The page fetches each firmware from its author's site and boots the same update package a real FM-1 would install.

What was built. A Rust workspace (`fm1-isa`, `fm1-core`, `fm1-soc` and three front ends) of about 12 600 hand-written lines of Rust plus a generated decoder table; a native player (`fm1-live`) running the real 240 MHz clock at 2.6–6.3× real time; a batch runner for tests; and a WebAssembly build in an AudioWorklet that runs every firmware at 240 MHz in a browser (1.6–3.7× real time in Node). It boots M-VAVE's own firmware (V13 and V15) with both cores, and the fifteen community firmwares listed on `fm1-editor.com`.

2 Starting point: reconnaissance

The first day went to reading, not coding. The relevant public sources were: AL-255's *FM-1-RE* (boot chain, memory layout, a V13 disassembly, firmware images), ip2k's *lunar-modulator* (hardware, update protocol, recovery risks), kagaimiq's JieLi knowledge base and tools, and Felucca's `hal/` directory, which is in effect the peripheral documentation: key matrix, encoders, display, audio. Recon established that the SoC is an AC791N/WL82 (not the "JL-BR22" string in the firmware), that flash is 1 MB mapped for execute-in-place at `0x02000000`, and that several GPL firmwares built from C with the same toolchain exist, which makes them golden references.

Two early checks made the plan concrete. First, AL-255's V13 disassembly was reproduced byte-for-byte by wrapping the raw application in an ELF `.fw` section and running the vendor `objdump` in Docker; so the vendor disassembler, not Ghidra, would be the authority on encodings. Second, a C file compiled to pi32v2 gave (source → bytes → mnemonic) triples, e.g. `r0 += r1` = `10 18`, `rts` = `80 00`. The conclusion, written into the plan on day one, was that ISA recovery would be *a mining exercise, not black-box discovery*, with differential probing as validation.

3 Recovering the instruction set

3.1 Mask mining from disassembly corpora

The vendor `objdump` prints every instruction's bytes next to its text. Running it over five corpora (the V13 and V14 applications, two bootloader builds and the USB OTA loader, about 421 000 instructions) gives the raw material. `tools/build_isa.py` normalises each line into a *printed shape*: registers become `R`, decimal and hex immediates `#i` and `#h`, register lists are canonicalised, and the vendor's `<_fw+0x...>` annotations are stripped. (That last step had a costly bug: stripping un-anchored ate the `<` operator in `ifs (r6 < 3) goto`, creating phantom forms that covered 3 % of the corpus. It is anchored at the end of the line now.) Instructions with the same shape and length are grouped, and for each group the bits that never vary become the class's `mask/match`.

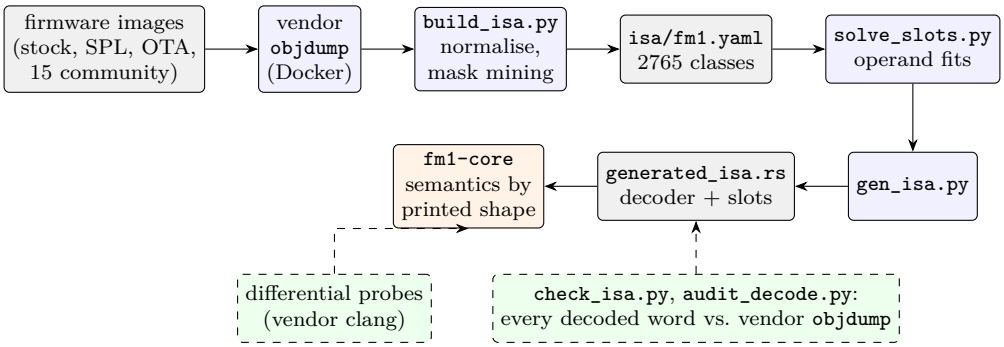


Figure 2: The ISA pipeline. The YAML table is the single source of truth: the Rust decoder is generated from it and never edited by hand. Dashed: the checks that gate every regeneration.

The result on the first day was 1067 classes. A streaming decoder over the same corpora, choosing by longest match, then mask popcount, then corpus count, decoded all of them with 0 mismatches and 0 undecodable words. When the fifteen community firmwares were added (their raw applications wrapped in ELF for `objdump` by `tools/disasm_app.sh`), the table was re-mined from 25 listings into 2765 classes; `tools/isa_stable_names.py` keeps class names stable across re-mining because parts of the core key on them.

3.2 Operand fields as exact linear models

Masks say which bits are opcode; they do not say how the variable bits become operands. The first operand fitter (`derive.py`) tried per-field templates and solved 206 classes. The approach that worked, `tools/solve_slots.py`, fits every printed operand of a class as an *exact* linear model over the class's varying bits, by rational Gaussian elimination, with collinear bits resolved by a power-of-two search. Operand kinds are `imm`, `rel` (sign-extended, scaled, pc-relative), `scaled`, `bit/nbit`, `composed`, `const`, `zero32` and register blocks. A fit is accepted only if it reproduces *every* sample of the class. This raised coverage to 764 classes, 86.8 % of corpus instructions.

For instruction families whose encodings span several classes, *family-level fitting* worked best: take every corpus line of one printed shape across classes (e.g. all 5302 `if (rN op #imm) goto` lines), propose the bit fields, and verify against all of them. Five families went to zero mismatches in one session this way. Two examples of what came out:

- **Compare-immediate branches.** Register in bits 0–3; byte 1 selects the family (`f8` ==/!=, `f9` unsigned $\geq/<$, `fc` unsigned $>/\leq$, `fd/fe` signed); bit 7 the second operator of the family; a 10-bit immediate assembled as bits(25,31) | bits(4,6)$\ll$7; offset $= 2 \cdot \mathrm{sext}_9(\ldots)$ from $pc + 4$.
- **The composed 12-bit immediate**, shared by `R = #h`, `R = R op #h` and memory read-modify-write forms. With code c = bits(24,27) and m = bits(16,23): $c = 0$ is the literal m; 1 replicates m per halfword; 2 places $m \ll 8$ per halfword; 3 replicates per byte; $c \geq 4$ is float-like, $E = 2c + (m \gg 7)$, value = (`0x80` | (m & `0x7f`)) $\ll (32 - E)$. This one rule reproduces a 70-entry table of observed constants (IEEE floats among them) and about 1650 ALU and move samples.

Bits the corpora never vary cannot be fitted from them. For those the *assembler oracle* was used: generate assembly text sweeping one operand at a time, assemble thousands of lines per Docker run with the vendor Clang, and read the bytes back. This is how the FPU was found: the stock firmware uses float instructions that `objdump` prints as `<unknown instruction>`; compiling with `-mcpu=r3` and disassembling the assembler's output gave the `3f e5` encoding (add, sub, mul, div, min, max, conversions).

3.3 Semantics keyed by printed shape

`fm1-core` executes instructions by their printed shape (`Cpu::exec_syntax`), taking operands from the generated slots in printed order, so one arm for `R = R + #i` serves every class that prints that way. Raw-bit decoders exist only where one class mixes encodings. When the solver began covering a class, its hand-written decoder was deleted; three of those turned out to be silently wrong (one stored `r9` instead of `r1` because it took bit 7 as part of the register number). Behaviours established while booting real code include:

- **Dual-issue pairs**: an instruction printed with a trailing `#` pairs with the next one, which reads registers from *before* the first wrote them (deferred register commit).

- **Predicated blocks** `if (rA op x) { ...} else { ...}` whose then/else lengths are counted in *units* (a pair is one unit, a nested block one unit).
- **Hardware loops** `rep N rR {`: block length $2(\text{bits}(4,7)+1)$ bytes, count in a register or immediate.
- **Interrupt nesting** follows the Blackfin rule: an ISR can be interrupted only while its `reti` is saved on the stack. (Without this, a tick between an ISR's `reti` pop and its `rti` made the `rti` return to itself.)
- **Two stacks**: in user mode an interrupt saves `sp` to `usp` and continues on `ssp`. The stock RTOS's context switch depends on it.
- **Flags**: N, C (set means no borrow, as on ARM), Z, V in `psr`, read by Clang's 64-bit arithmetic right after the low-word add or subtract.

4 Validation: how we knew it was right

Without hardware, "right" had to be defined by things that could be checked mechanically. Four methods were used, and every change had to pass the ones that applied.

1. Decoder audit against the vendor. `tools/audit_decode.py` decodes every word of a vendor listing with the Rust decoder and compares operands with the vendor text. It found, for instance, that a sp-relative pair store fitted on V13 alone broke for offsets ≥ 512 (keys stayed lit because LED arrays were stale).

2. Differential probes. A probe is a small C program built with the vendor Clang in which each check computes an expression at run time from `volatile` inputs and compares it with the same expression folded at compile time by LLVM (Listing 1). The compiler's constant folder is thus the oracle for the compiler's own model of the ISA, and a mismatch is an emulator bug. Probe suites cover integer arithmetic (121 checks), Felucca's Huffman font decoder on data from its own packer (139), the FPU (1464 cases against LLVM's folding), and 64-bit and soft-double arithmetic.

```
volatile int32_t a = 1234567, b = -89, c = 7;
volatile uint32_t ua = 0xDEADBEEFu, uc = 13u;
T(a / c,  1234567 / 7);
T(b % c,  -89 % 7);
T(ua % uc, 0xDEADBEEFu % 13u);
T(a * 241, 1234567 * 241);
```

Listing 1: From `probes/arith.c`. `T(runtime, folded)`: the first argument compiles to pi32v2 instructions, the second to a constant.

3. Same code, two machines. Where a firmware's C source was available, a piece of it was built both for the device and natively and compared word for word. X0X's TR-909 code exposed the register-pair addressing modes (a pre-increment store overwrote one voice's settings with another's) and a bitfield-extract instruction that had been executed as an add. Later, reSID's double-precision setup computed on the emulator differed from the host; tracing it found that `psr` always read 0, so every 64-bit add lost its carry and every 64-bit subtract borrowed.

4. Byte-identical audio across execution modes. The emulator has three ways to run the same code: one instruction at a time (the reference), a block interpreter with micro-ops, and a JIT. A bench mode (`fm1-live -bench MS` with `FM1_BENCH_WAV`, plus a scripted exercise of every key, encoder and note) writes the raw audio. Every optimisation was accepted only if the audio was byte-identical in all three modes *and* to the build before the change, for every firmware. This turns "did the JIT break something subtle?" into a file comparison. (Dual-core firmwares are deterministic per mode but interleave cores at slightly different points, so for them the comparison is per mode against the previous build.)

Table 1 lists some of what these methods caught. Most bugs were invisible at the point of failure: the symptom appeared much later, in a picture or a sound.

5 The system-on-chip

Peripherals in boot order, lazily. The SoC layer started with RAM, the XIP window and a stub that logs every access to an unknown address and returns a benign value. The rule was: *the access log is the reverse-engineering roadmap.* A peripheral was modelled when a firmware was seen to depend on it, from Felucca's

Defect	Symptom	Found by
2-byte shift: bit 7 taken as register bit	CRT copy loops ran 4× too long; RAM corrupted	boot trace
`[R+#i] = R`: bit 7 in source register	stored `r9` instead of `r1`	solver replacing hand code
16-bit read at word offset 2	every 16-bit canvas sheared	probe (lcdtest)
unsigned compare immediates sign-extended	wrong branches on `if (r2 < 2111)`	probe
4-byte `R = R « #n` run as `»>`	wrong arithmetic in drawing code	probe
sp-relative pair offset ≥ 512	stale LED arrays, keys stayed lit	decoder audit
interrupt nesting	`rti` returned to itself; hang	live play
register-pair pre-increment modes	X0X's 909 settings overwritten	device vs. native build
`b4 e1` extract executed as add	an address used as an exponent	device vs. native build
no arithmetic flags	64-bit add lost carry; reSID filter wrong	soft-double probes
random-number unit read as 0	stock UI froze after 8 s	debugging the freeze
unknown-access log shifted 8192 entries per access	one firmware ran far too slowly	investigating the slowdown

Table 1: A selection of defects and how each was found.

HAL headers and the AC79 SDK where possible. In order: the interrupt controller (priority nibbles, software latch, RAM vectors), TIMER4/TIMER5 on a 240 MHz cycle clock, SPI NOR flash (fonts), an ST7789-style LCD on SPI1 with DMA, the I2S output (ALNK0, double-buffered DMA; Felucca's samples turned out to be 24-bit words in the *low* bits, not left-justified as a HAL comment says), the SAR ADC (MASTER pot, battery), and the key matrix behind two 74HC595 shift registers with seven quadrature encoders.

Felucca first, then the stock firmware. Felucca (open source) ran end to end on day three. The stock firmware needed more, each piece found from a concrete symptom: SPI2 DMA to scan the key columns (keys ignored), a hardware lock between cores for `lockset`/`lockclr` (a task queued twice), a CRC-16 unit (the patch file system failed every CRC), `idle` that waits for an interrupt (0.44× real time), and a random-number generator: a routine in the audio driver parks its task when a random word equals a fixed descrambled value, which a constant 0 always did, so the UI froze about eight seconds after boot. The factory DX7 patch bank lives in a flash area that updates never write; since it cannot be redistributed, a blank area gets the emulator's own small bank, and `-syx` loads a DX7 bank.

Two cores. The stock firmware renders sound on core 1. Core 1 starts when core 0 sets a bit at `0x10008` (latched at the write: some firmwares hold it only briefly), at the address in `0x01C7FFF8`. It has no clock of its own: core 0 runs in slices of 1024 instructions, then core 1 runs as many instructions as cycles passed, and stops early when it returns to the same pc with the same registers (a job loop with nothing to do).

6 Making it fast

FM-1 firmwares do their audio in an interrupt every 128 samples, and live play needs the full 240 MHz in real time. Speed was added in stages, each gated on the byte-identical audio rule:

1. **Decode cache** and cheap shape matching: from 1 M to 17 M instructions/s.
2. **Micro-ops and blocks**: common forms pre-decoded with operands and branch targets resolved once; straight-line blocks; RAM/XIP fast paths; idle skipping of timer-polling loops. Interrupt delivery stays exact at block granularity: blocks end at due ticks, peripheral writes, and the ends of pairs, predicated blocks and loops.
3. **AArch64 JIT** (macOS): hot blocks compiled to native code with chaining, guest state in the `Cpu` struct, inline RAM/XIP loads and stores, helpers that sync the clock to the exact instruction for everything else.
4. **Shrinking the slow path**: 1.7–3.8 % of instructions fell back to the generic path and cost half the time, because each also ended a block. New micro-ops (shared with the interpreter through one `exec_raw`, so the two cannot drift), dual-issue pairs with no dependency run as two plain ops, and a remembered "needs step" marker brought it to 0.1 %.
5. **WebAssembly JIT**: in the browser, a hot block's *region* (blocks reachable over fixed edges, at most 32 blocks or 384 ops) becomes a small Wasm module that the host compiles and installs in the exported

function table; Rust calls it by index. Per-block modules gained nothing (blocks average 3.5 instructions); regions doubled the speed. Chrome compiles them synchronously inside the AudioWorklet in about 25 µs, at most 16 per audio quantum.

Real-time factor at 240 MHz	slowest (X0X)	Felucca	fastest (Melodee)
native, AArch64 JIT	1.14 → 2.65	1.63 → 4.4	3.1 → 5.9
WebAssembly (Node)	0.53 → 1.62	0.66 → 2.3	0.97 → 3.5

Table 2: The speed pass of day four (before → after); every firmware's bench audio byte-identical to before.

One measure that was tried and dropped: per-block tick accounting in the interpreter, paid before any non-RAM access. It was 5–7 % slower, because blocks are short. A/B timing was done interleaved (`tools/bench_ab.sh`), since back-to-back runs drifted 6–8 %.

7 The browser, MIDI and publishing

The web build runs the emulator in an AudioWorklet. The page hosts no firmware: it fetches each update package from its author's GitHub Pages (all allow cross-origin reads) and unpacks it with `web/fwsc.js`, a package reader checked byte-equal against Felucca's own on every package. Authors publish new versions often; when the catalog's URL is gone, the page takes the package the author's installer page names now (four had moved on the day of publication). M-VAVE's server sends no CORS header, so users open their own copy of the stock package.

New versions also test the emulator. On the day this report was written, Felucca 1.6 appeared and the page picked it up; the emulator halted on `d7 ec 51 38`, which the vendor prints as `[r5+-128] = r3`. The table had a class for it with 19 samples but no operand fit. Comparing all word loads and stores in the 25 listings showed that bit 17 selects pre-increment or a plain offset and that byte 0's bit 2 is the offset's sign: one model, 27 884 samples, 0 mismatches. With it Felucca 1.6 plays, and every other firmware's audio stayed byte-identical.

MIDI IN is modelled at the UART, not injected into firmware: UART1 with receive DMA into a ring, a latch that counts received bytes, and a timeout interrupt after a configured silence, one byte per 31 250-baud byte time (320 µs). Each firmware therefore reads MIDI with its own driver; Felucca polls the count from a timer interrupt, the stock firmware takes the timeout interrupt. The emulator found that the stock firmware listens on channel 2, a claim still to be checked on a real unit. In the browser, Web MIDI feeds the bytes; natively, every MIDI input of the computer.

Continuous integration runs the unit tests, builds the site and runs two smoke tests in Node against the built WebAssembly: Felucca must boot to a screen and sound and play a MIDI note; the SID firmware must stay silent until a key is pressed, then sound. The project began inside a larger repository of demos and was moved into its own repository with history kept (`git subtree split`); the old URLs forward with their query strings.

8 A new firmware, developed on the emulator: FM-1 SID

To test the emulator as a development platform, we asked whether the FM-1 could be a Commodore 64 SID synth. The study (Table 3) compared an integer SID model written for the purpose with reSID 0.16, reSID 1.0 and reSIDfp, by recording register streams and replaying them on every engine, each clocked to the exact cycle of each write; the costs on the FM-1 were measured by counting emulated instructions between markers, and the FM-1 output was compared with the host build of the same code (bit-identical in every case).

Engine	instr./sample	of a 240 MHz core	verdict
own integer model	316	5.8 %	fits; lowest fidelity
reSID 0.16, fast	440	8.1 %	**fits; chosen**
reSID 0.16, interpolate	5 562	102 %	too heavy
reSID 0.16, resample	39 226	721 %	far too heavy
reSID 1.0 / reSIDfp	–	–	8–21 MB filter tables; FM-1 has 1 MB flash

Table 3: SID engines per chip (3 voices and filter) at 44.1 kHz, measured on the emulated FM-1.

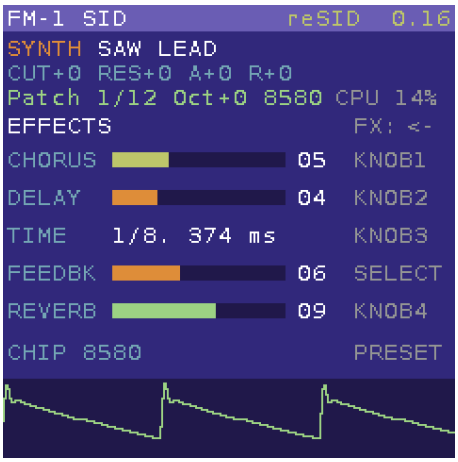


Figure 3: FM-1 SID's effects page on the emulated LCD: the knob assignments, the delay time computed from the sequencer tempo, the chip model, and an oscilloscope of the output.

The firmware that followed nests machines: a C64 tune's own 6502 code runs in floooh's cycle-stepped `m6502.h`, compiled for pi32v2, which runs on the emulated FM-1, which runs in WebAssembly in the browser, whose hot pi32v2 code is compiled at run time into further WebAssembly. The main loop runs the tune's play routine up to three frames ahead and queues each SID write with its C64 cycle; the audio interrupt clocks reSID to each write's cycle. Output correlates 0.984 with host reSID fed the same stream. The web page searches all 61 157 tunes of the High Voltage SID Collection, fetched from `hvsc.c64.org` when picked and appended to the firmware image as a "tune pack".

The firmware then grew into an instrument: a synth mode where the keys play twelve SID patches, a three-track step sequencer, and an effects page (Fig. 3) with stereo chorus, a tempo delay whose right tap answers 1.5 times later, and a room reverb (four damped combs, two allpasses), adapted from Felucca's effects. The same verification style applied: with effects at zero the player's output was compared sample by sample with the firmware before the change (0 of 176 512 samples differed); the audio interrupt's load rose from about 10 % to 14 % of the core, and RAM use to 425 of 432 KB. The firmware is packaged as a real FM-1 update package with Felucca's update loader and three hash-checked files of JieLi's Apache-licensed SDK, and the package boots straight into the synth mode. The firmware image, rebuilt on macOS through the Dockerised toolchain, was byte-identical to the one built on Linux.

Writing this firmware found four more emulator defects (the flags and carry gaps of Table 1 among them), which is itself an argument for developing against an emulator whose every fix comes with a probe.

9 Tuning towards the real hardware

Released before it was tested. FM-1 SID was offered for download as an FM-1 update package on 11 October, a day after the first community firmware ran on the emulator, and for a few days the community's firmware hub listed it beside firmwares that had run on real units. Nobody on the project owned an FM-1 yet. Eight days later, people who installed it found that it played but that their computers no longer saw the synth (GitHub issues #22 and #24, 19 October): FM-1 SID had no USB stack at all, and an FM-1 firmware is updated only through the firmware that is running, so no installer could reach it. The way back was found by the community, not by us: holding OCT− and OCT+ together for about five seconds at power-on enters the chip's own update mode, below every firmware, from which a third-party flasher (M-flasher) wrote M-VAVE's stock firmware back. FM-1 SID was taken off the hub. No unit was damaged permanently, but each owner had to find and perform that recovery themselves.

The emulator could not have caught this. It had no USB model then (one was added after the hardware arrived), and nothing in it asked whether a firmware can be replaced. The failure was one of process rather than emulation: a firmware that had never run on hardware was published as an install. The rule taken from it: what has not run on a real FM-1 is labelled so, and a package is offered only once it has run there, with its way back tested on the device.

Round one: the emulator was not the device. The FM-1 arrived on 23 October, and the first attempt to fix FM-1 SID on it was humbling. USB-MIDI was added, with three exits to the update mode (OCT− and OCT+ held while running, two failed starts in a row, and the installers' upgrade command), and every check on the emulator passed. On the device, the synth mode played but no tune did, and the firmware

received the installer's commands but none of its replies reached the computer. Getting the unit back took three recoveries through the chip's mask-ROM mode, for which no macOS tool exists (a Linux machine and MvaveFM1Unbricker). The question was no longer "is the emulator consistent?" but "where does it differ from the chip?".

fm1-dock: install once, then deploy and measure. The day after the first reports of stuck units, three days before the hardware arrived, the author asked whether the risk could be removed at its source: a thin resident layer, installed once, through which experimental synths are deployed and inspected without a full firmware upgrade each time. That layer was built on the emulator as *harbor* and renamed *fm1-dock* when it was released. It is a host firmware, grown from zvenson's Dinghy, that owns everything a broken experiment must not take down: USB-MIDI, the installers' protocol, the USB rescue (OCT− at power-on), the screen and the keys. Over its own SysEx commands it accepts three kinds of code. A *module* is a synth, compiled against a small host interface, written into a flash slot fm1-dock reserves for it (a deploy takes about ten seconds, erases only that slot, and can be undone by deleting the module), and picked with HOME. A *probe* is a program relocated into RAM and run once; nothing is written to flash. A file sent to the running module's `data()` hook (the MODULE command) is how FM-1 SID 0.3 plays a tune sent over USB, straight into its emulated C64's memory. The emulator speaks the same protocol over an emulated USB device, so every tool runs unchanged against either machine.

Measuring instead of guessing. The probes were what answered the question. A probe writes typed records (numbers, words, TIMER4 timings with the interrupts off and an empty loop subtracted) into a buffer in memory that survives a reset, so a probe that crashes still reports how far it got and where. The same probe runs on the FM-1 and on the emulator (`tools/probe.py run X.c -on emu,fm1`), and the two results are printed side by side with the ratio of every timing. A new question about the hardware costs a C file of a few dozen lines and a few seconds, not a firmware install and a risk to the unit; Table 7 lists the questions asked in the first two days.

The first probe found the first gap. Every probe crashed on the device on its second instruction, at the same addresses each time, while the emulator ran them happily. Reading the crash record (another probe, a memory dump) showed the CPU exception with cause bit 12 of the debug unit: the *PC limit*. Dinghy's start-up code, inherited from Felucca, arms the core's guards: instruction fetches are allowed only within the firmware's own code and its RAM routines, writes into the stack guard bands and the vector area trap, and so do accesses to unmapped space. The emulator modelled none of this. With the guards modelled (the limit registers kept, a block fetched outside both windows ends a run, the instruction executes and interrupt 1 is raised with the cause set), the emulator crashed the same probes at exactly the device's addresses, `0x01C301A2` and `0x01C301BA`. With fm1-dock widening the PC limits around a probe, both ran: the first program loaded and run from RAM on a real FM-1.

What the probes measured. Table 4 lists the gaps found, each turned into an emulator change and a test. Three of them explained FM-1 SID's failure on the device. The emulator finished LCD transfers at once, while the real SPI clock is about 11 MHz: FM-1 SID redrew the whole screen (115 KB, ≈82 ms) every 40 ms and waited for it, which starved the main loop that runs the tune ahead of the audio interrupt. With timed LCD transfers the emulator reproduced the silence (peak level 112 instead of 9 086) and now warns when the screen takes most of the time. The second was speed: the emulator charged one 240 MHz cycle per instruction, but FM-1 SID's own load meter read 24 % on the device against 13 % on the emulator. Emulated at 130 MHz, the emulator reproduced the device exactly (GoatTracker songs played, an RSID tune went silent), and the cause was a firmware bug that only a slow enough CPU exposes: once the play routine fell behind the audio interrupt, an unsigned `gen_cyc - play_cyc` wrapped and generation stopped for good. The third, the installer that never answered, is still not explained; one plausible theory (the `idle` instruction in the main loop) was refuted by a probe that sent 40 MIDI notes over USB between `idle`s and saw all 40 arrive.

A cycle model. A probe with one instruction per statement (inline assembly, eight per loop round) gave the cost of each instruction class on the device (Table 5). The core is fast at straight-line arithmetic, which it issues two at a time, and slow at control flow: a taken branch costs five to six cycles, a call and return ten. A second probe swept the footprint and stride of flash reads: a footprint up to 32 KB hits, 64 KB misses; the cost doubles with the stride up to 64 bytes; a miss costs about 1.3 µs. These numbers became a *strict* mode (`FM1_STRICT=1`): the interpreter only, a clock in eighths of a cycle, each instruction its measured cost (taken and not-taken branches apart), a 32 KB, 64-byte-line cache in front of flash for data and code, and misaligned accesses faulting. In strict mode FM-1 SID's load meter reads 21 % (device 24 %, fast mode 13 %),

Gap	On the FM-1	In the emulator now
LCD transfer time	SPI ≈11.2 MHz; a full frame ≈82 ms	timed at 12 MHz; a warning when the screen dominates
PC limit, bus guard	a fetch outside the limits, or a NULL-page access, raises the exception	modelled, cause bits in `DBG_MSG`
Misaligned 16/32-bit RAM access	faults (`EMU_MSG` bit 0, always on); flash reads across bytes	faults in strict mode
Instruction cost	0.58–6.75 cycles by class (Table 5)	per-class costs in strict mode
Flash cache	32 KB, 64-byte lines, ≈300 cycles a miss	modelled in strict mode
TIMER4	24 MHz to 51 ppm against the host's USB frames	as assumed
`trigger` (fault stub)	freezes the trace unit	a no-op (was "unsupported")

Table 4: Differences between the emulator and the FM-1 found with probes, and what was done.

Instruction class	FM-1	strict	Instruction class	FM-1	strict
`nop`, `and`, add	0.58–0.75	0.62	load word / half / byte	1.25–1.34	1.25
6-byte constant	1.08	1.12	store	1.42	1.45
multiply	1.92	1.88	push + pop of 4 registers	10.6	10.5
divide	6.75	6.78	`goto`	5.17	5.25
branch, not taken	0.85	0.88	branch, taken	6.50	6.50
call + return	10.2	10.0	load feeding a load	3.08	1.25

Table 5: Cycles (240 MHz) per instruction measured on the FM-1 (`probes/cycles.c`, run from RAM) and charged by the emulator's strict mode. The load-use stall in the last row is not modelled yet.

every class of Table 5 but the load-use stall is within about 10 % of the device (a store followed by a load of the same word, and independent multiplies, are not modelled either), and the firmware build that failed on the device fails the same way. The fast modes still count one cycle per instruction: changing the three compilers' accounting is left for later.

DSP, the FPU, and what the community uses. The same method looked at the core's DSP-style instructions. Halfword multiply-accumulates with saturation cost 1.2 cycles on the device, three times less than the 64-bit multiply-accumulate chain the compiler emits for `acc += (int64_t)a * b` (3.4 cycles); the saturating shift exposed as `__builtin_pi32v2_qasl` clamps in 0.6 cycles; a cache line prefetch (`pfetch`) issued a microsecond ahead halves a miss. A probe of audio kernels, run on both machines with checksums of the output, showed that a biquad costs about 20 cycles a sample in any form (32-bit, 64-bit MAC, halfword MAC, or single-precision float on the hardware FPU), and that a float state-variable filter is faster than its integer version; every checksum the emulator could compute matched the device's. The probe also exposed a limit of corpus mining: the halfword multiply-accumulate classes were fitted only for the register combinations the corpora contain, so the emulator cannot decode them with any others. Running all 80 published firmwares on the emulator with an executed-instruction histogram showed that none uses those instructions, a few use 64-bit multiplies, and 16 do a substantial part of their work on the FPU.

The cost of a filter turned out not to lie in its arithmetic but in its loop: the taken branch that closes an iteration costs as much as six plain instructions. The core has hardware loops, `rep N rR { ... }`, which no firmware's compiler emits (the 80 firmwares executed eight between them). Probes established three rules that no document states: the body is at most 32 bytes (the assembler counts it), the loop counts its register down to zero, and only counts up to 31 work (for 32 and above the device runs the count modulo 32, plus one, times; the emulator ran the full count, the one semantic error the probes found). Within those rules, a hardware loop with halfword multiply-accumulates runs a 64-tap FIR on 16-bit samples at 2.9 cycles a tap against 11 for the C loop, 3.9 times faster, with an identical result (Table 6): one channel at 44.1 kHz goes from 13 % of the core to 3.4 %. The gain is in tight inner loops (FIR, multi-tap delays, mixing, gain); a biquad's body does not fit 32 bytes. One hoped-for gain did not exist: the emulator's map has a 512 KB "overlay" bank at `0x04000000` and 256 KB at `0x01F00000`, guessed on day one, but on the device the first hangs the bus, the second faults (the stock start-up clears 16 KB there after writing cache control registers: most likely the cache's own memory), and so does anything past the 512 KB of main RAM.

Round two: let the host own USB. The probes refuted the two theories for FM-1 SID's silent USB: `idle` was not it, and neither was buffer alignment, although an emulator rule requiring it reproduced every failed build. A rule fitted to past failures is not evidence until the chip agrees. Rather than keep hunting

Kernel (on the FM-1)	C	hardware loop + DSP	speed-up
loop overhead (a load and an add an iteration)	6.75	2.38	2.8×
64-tap FIR, per tap (C with a 64-bit accumulator: 9.78)	11.06	2.86	3.9×
block gain with a clamp, per sample	12.81	5.70	2.2×

Table 6: Cycles on the FM-1 (`probes/dspx2.c`): C as the vendor compiler builds it, against inline assembly with a hardware loop (`rep`), halfword multiply-accumulates with saturation and the saturating shift. Every fast version gives the C version's result.

Question (probe)	Answer on the FM-1
Does code run from RAM? (the first probe)	No: it faults on its second instruction (the PC limit); it runs once fm1-dock widens the limits
Does `idle` in a main loop stop USB? (`idle`)	No: 40 of 40 MIDI notes sent between `idle`s, each ≤93 µs
Must a USB transmit buffer be 8-byte aligned? (`usbtxoff.c`)	No: fm1-dock's driver sends from offsets 4 and 12, although that rule fitted all six failed FM-1 SID builds in the emulator
Can a probe drive a USB endpoint itself?	Not safely: the computer stopped collecting from the FM-1 until a replug; probes send through fm1-dock
Misaligned 16/32-bit RAM access (`align`)	Faults, always
LCD SPI clock (`lcd`)	≈11.2 MHz: a full frame takes ≈82 ms
TIMER4 against the computer's USB frames (`clock`)	24 MHz within 51 ppm
Instruction costs (`cycles.c`)	0.58–10.6 cycles by class (Table 5)
Flash cache (`cache.c`)	32 KB, 64-byte lines, ≈300 cycles a miss
Hardware loops, DSP instructions (`dspx2.c`)	body ≤32 bytes, counts ≤31; FIR 3.9× faster (Table 6)
Memory banks mapped since day one	`0x04000000` hangs the bus, `0x01F00000` faults

Table 7: Questions answered by probes run from RAM through fm1-dock on 24–25 October. Each took a C file and seconds on the device; none wrote flash.

the bug in FM-1 SID's own USB code, FM-1 SID was rebuilt as an fm1-dock module, so that fm1-dock answers the installers whatever the module does. The probes' findings went in at the same time: the screen sends only the rows that changed, a strip at a time with the tune fed between strips; a play routine that falls behind catches up instead of stopping; and the cycle-stepped 6502 core gave way to one that runs an instruction per dispatch with each memory access at its cycle, fast enough for the slower real CPU (PSID output bit-identical, RSID checked against libsidplayfp). Deployed the same day, it played on the device and USB answered while it played.

Round three: the safety net is hardware too. The standalone FM-1 SID was then deployed into a developer feature of fm1-dock, a *boot slot* that fm1-dock hands the FM-1 to at power-on, to study its USB bug on the chip. Two flaws made it a trap. fm1-dock decided to hand over by comparing names, and the boot firmware had the same name as the selected module. And the escape keys were read only in the first 50 ms after power-on, which the emulator honoured and the device did not. Every start handed over to the firmware whose USB was broken, and the unit needed a fourth recovery from Linux. The fixes: release builds have no boot slot at all (a boot firmware is refused, an old entry is forgotten at start-up), the power-on keys are read for 500 ms, and OCT− is read again while fm1-dock starts. The lesson is the section's in miniature: a rescue path is hardware behaviour, and it is not proven until it has been taken on the device.

Round four: released, verified on the device. On 25 October FM-1 SID 0.2 shipped as a solo firmware built on fm1-dock: one module, with fm1-dock's USB, installer protocol and rescue underneath. It was installed with a community installer and played with the heaviest audio block at 34 % of its time and no overruns, while the installers reached it over USB. The same evening fm1-dock 0.6 and FM-1 SID 0.3 added two features that need the device and the browser together: any SID tune picked on the web page is sent over USB into FM-1 SID's emulated C64 memory and plays at once (Commando in 1.6 s, nothing written to flash), and the FM-1's keys go out as MIDI notes, so the real FM-1 plays the emulated one in the browser. Table 8 summarises the rounds.

What changed. The emulator was right about almost everything that its oracles could reach: of everything the probes tested, one instruction's semantics needed correcting (the hardware loop's count), and every kernel it could run gave the device's output. It was wrong about what no oracle covered: timing, caches, and the

Date	Change	On the FM-1	Recoveries
11–19 Oct	FM-1 SID package, tested on the emulator only; no USB	plays; no computer sees it	owners, chip's mode + M-flasher
23 Oct	USB-MIDI, three exits to the update mode	tunes silent; installer replies never arrive	3, from Linux
24 Oct	probes on fm1-dock; screen strips, catch-up, a faster 6502 core; FM-1 SID as a module	plays; USB answers while it plays	–
25 Oct	the standalone in fm1-dock's boot slot, to study its USB	escape keys missed; handed over at every start	1, from Linux
25 Oct	no boot slot in releases, longer key reads; FM-1 SID 0.2 as a solo app	heaviest block 34 %, no overruns; installers answer	–
25 Oct	fm1-dock 0.6, FM-1 SID 0.3: tunes over USB, keys as MIDI	a tune from the page plays on the device in 1.6 s	–

Table 8: FM-1 SID from its first release to one verified on the device.

Day	Result
1 (Oct 8)	Recon; toolchain verified in Docker; V13 disassembly reproduced byte-for-byte.
2 (Oct 9)	1067 classes mined, operand solver (86.8 % of instructions); stock V13 boots to RTOS start on two cores.
3 (Oct 10)	Interrupts, timers, flash, LCD, I2S, keys: Felucca plays; live at 240 MHz with the AArch64 JIT; fifteen firmwares, the FPU, the browser build, published.
4 (Oct 11)	Flags and carry; stock firmware with both cores; SID study, player firmware, HVSC page, synth mode, update package, sequencer; WebAssembly JIT.
5 (Oct 12)	MIDI in; a guide on the page; own repository; FM-1 SID boots as a synth, with effects.
Oct 19–20	Users who installed FM-1 SID cannot reach their FM-1 over USB; the community finds the way back. The idea of a host to deploy synths into.
Oct 23–24	The FM-1 on the desk: FM-1 SID fails on it; fm1-dock and probes over USB-MIDI; guards, timed LCD, a cycle and flash-cache model; FM-1 SID as an fm1-dock module (Section 9).
Oct 25	A boot-slot trap and its fixes; fm1-dock 0.6 and FM-1 SID 0.2 and 0.3 released, verified on the device.

Table 9: Timeline. Each commit is a working state that passed the checks.

protections a start-up file switches on. Those are exactly the things a vendor disassembler and a compiler's constant folder cannot know, and a day of probes on the device measured them. The rule we take from it is the one the probe tool makes cheap: a claim about the hardware is a measurement, and a firmware is not ready for the device because it passes on the emulator, but because the emulator has been shown to fail the way the device fails.

10 How the work was organised

The work was done by one person directing an AI coding agent (Claude Code, Anthropic). The division of labour was roughly as follows. The human chose what to build and in which order (the emulator itself, the SID synth idea, adding effects, booting the SID firmware as a synth, moving to a repository of its own), judged results by playing and listening, decided the tone of the public pages ("a better story", less technical), and approved anything outward-facing: pushes, publishing, messages to others. The agent read the sources, ran the vendor toolchain, wrote the mining and fitting tools, the emulator, the probes and the firmware, measured, and kept a running work log (`PLAN.md`) that recorded each phase's state, numbers and open questions, so that work could resume after an interruption or in a new session. Separate agents were occasionally used for self-contained jobs, such as writing the page's guide.

What made the agent's output trustworthy was not review of every line but a small set of rules that turned claims into checks:

- **One source of truth.** The decoder is generated from the YAML table; a second, hand-maintained table was never allowed. When generated slots covered a class, the hand-written decoder was deleted, which is how three silent bugs surfaced.
- **The oracle is the vendor, not a guess.** Encodings are checked against the vendor disassembler; semantics against the vendor compiler's constant folding; unseen bits against the vendor assembler.
- **Byte-identical or it did not happen.** Optimisations must leave audio unchanged across execution modes and against the previous build, for every firmware.
- **Log the unknown, never swallow it.** Unknown memory accesses are logged, unknown instructions halt with their class name; both are to-do lists.

- **State what is unverified.** Statements not yet checked on hardware (the stock MIDI channel, FM-1 SID on a real unit, which instructions set flags) are marked as such in the documentation and on the pages.
- **The device decides, and the way back comes first.** Learned late, from FM-1 SID's first users (Section 9): an emulator pass makes a firmware a candidate, not a release. Nothing is offered for install before it has run on a real FM-1, and nothing risky goes onto the unit before its way back has been taken on the chip itself. Installs on the unit are run by the human; the agent may read from the device (identity, logs) and run probes, which never write flash.

11 Limitations

Section 9 compares the emulator with a real FM-1, but only in part. Instruction costs and the flash cache are modelled in the strict mode only; the fast modes still charge one cycle per instruction, and load-use and store-to-load stalls, sequential prefetch and a possible split of the cache between code and data are not modelled. The write and stack limits of the guards are not modelled, and misaligned accesses fault only in the strict mode. Why the standalone FM-1 SID builds never answered an installer on the device is not explained; FM-1 SID avoids the question by leaving USB to fm1-dock. In the emulator about one USB command frame in 80 is lost after the device has accepted its packets, so the senders repeat a frame after a second; whether the device does the same is not measured. Which instructions update flags on the real core is still assumed from compiled code, which only reads them right after a producer. Two FPU operations in the stock firmware (codes 7 and 8) are unknown, the halfword multiply-accumulate classes decode only for the register combinations seen in the corpora (656 classes in all have no solved operand fields), the hardware loop still runs counts of 32 and more in full, the two memory banks that do not exist on the device are still mapped, and the real entry chain of core 1 is not understood; the emulator starts it at the address the firmware writes. Each of these is now a probe away from a measurement.

12 Conclusion

An undocumented CPU in a consumer device turned out to be recoverable from its firmware in days, because the vendor's own tools could serve as oracles: the disassembler for encodings, the assembler for bits the corpora never vary, and the compiler's constant folding for semantics. The rest was engineering discipline: a generated decoder, peripherals modelled on demand from an access log, and a byte-identical audio rule that let aggressive optimisation (two JITs) proceed without fear. The result runs every published FM-1 firmware in a browser, and was good enough to develop a new firmware on before the synth was on the desk.

Acknowledgements

Built on the community's work: AL-255's FM-1-RE, ip2k's lunar-modulator, kagaimiq's jielie, hugelton's Felucca (whose hardware layer and effects FM-1 SID uses), cicloid's awesome-fm-1, and every firmware author who published their work. reSID is by Dag Lem; `m6502.h` by Andre Weissflog; the SID tunes come from the High Voltage SID Collection.

References

[1] AL-255. *FM-1-RE*. `https://github.com/AL-255/FM-1-RE`

[2] ip2k. *lunar-modulator*. `https://github.com/ip2k/lunar-modulator`

[3] kagaimiq. *jielie*. `https://kagaimiq.github.io/jielie`

[4] hugelton. *Felucca*. `https://github.com/hugelton/Felucca`

[5] cicloid. *awesome-fm-1*. `https://github.com/cicloid/awesome-fm-1`

[6] D. Lem. *reSID*, version 0.16.

[7] A. Weissflog. *chips* (`m6502.h`). `https://github.com/floooh/chips`

[8] *High Voltage SID Collection*. `https://hvsc.c64.org`

[9] J. Eriksson. *fm1-emu*. `https://github.com/joakimeriksson/fm1-emu`