From VB6 Floppy Disk to Rust Daemon: A 25-Year Port Story

Jeremy Anderson July 18, 2026 12 min read **Rust** **iced** **retrocomputing**

The .frm File on a Shelf

In 1999, I opened Visual Basic 6 and built a desktop calculator called **Project for Electronics**. It solved Ohm's Law and Watts Law — enter any two of voltage, current, resistance, or power, and the program computed the other two. It worked. It was useful. Then it sat on a floppy disk for a quarter century.

I found the source — a single `Main.frm` file, 13 KB, packed with every classic VB6 antipattern you'd expect from that era. `Long` integers for electrical values. A `For` loop brute-forcing square roots. A 1 ms `Timer` control polling text boxes because VB6 had no event-driven input binding. `GoTo` labels scattered like confetti. Wrong in every measurable sense, but the intent was solid.

This is the story of porting that program to Rust, fixing every bug, expanding it from one calculator to eight, adding a headless daemon mode, and putting the whole thing under AGPL-3.0 — partly for principle, mostly for the joke.

What Was Wrong (Everything)

Before writing a single line of Rust, I catalogued every defect in the VB6 source. Some were bugs; others were architectural decisions that made sense in 1999 but are radioactive today.

VB6 (1999)	RUST (2026)
`Long` (16-bit int) for watts, volts, amps, ohms — overflows above 32,767	`f64` throughout — 64-bit double precision, no overflow
Brute-force `For` loop to approximate square roots	`(x).max(0.0).sqrt()` — one call, hardware-accurate
1 ms `Timer` control polling all text boxes	iced's reactive `update()` — recalculate only on change
No input validation (negative resistance accepted silently)	Negative power/resistance rejected with error output
`GoTo Calculate`, `GoTo ClearFields` for flow control	Data-driven dispatch tables replace all branching
`Format(Value, "0.00")` — fixed 2 decimal places always	5-decimal precision with trailing-zero trimming via `trim_f()`
Ohm symbol displayed as "Ohms" (no Unicode)	Proper Unicode Ω everywhere

The Eight Calculators

What started as a single Ohm's Law tool grew into a suite I'm calling **MCP-Relational-Data** that bridges two worlds: electronics engineering and modern business/AI economics. The GUI presents all eight calculators in a two-column layout.

CALCULATOR	DOMAIN	INPUTS
Ohm's Law & Watts Law	Electronics	Any 2 of P, I, R, V
Margin & Markup	Ecommerce	Cost, Selling Price
ROI	Business	Investment, Revenue
AI Token Cost	LLM Economics	Model, Input/Output Tokens
Electricity Cost	Infrastructure	Watts, Hours, Days, Rate
Break-Even	Business	Fixed Costs, Price, Variable Cost

Solar Panel Revenue	Energy	Panels, Watts, Sun Hours, Rate, Cost
3D Print Cost	Manufacturing	Filament, Weight, Time, Power, Fail Rate

The Ohm's Law solver uses a data-driven dispatch table — six entries, each containing a pair of non-zero checks and a solve function. The first matching entry wins. The AI Token Cost estimator cycles through six models (GPT-4o, GPT-4, GPT-3.5 Turbo, Claude 3.5 Sonnet, Claude 3 Opus, Claude 3 Haiku) with per-million-token pricing compiled into the binary.

Architecture: calc.rs Is the Soul

The most important design decision was separating pure calculation logic from every presentation layer. `calc.rs` contains zero I/O, zero GUI imports, zero serde. It's just functions: numbers in, numbers out. Both the iced GUI and the JSON daemon call these same functions.

```
// calc.rs — the Ohm's Law solver uses a dispatch table
pub fn ohm_calculate(p: f64, i: f64, r: f64, v: f64) -> (f64, f64, f64, f64) {
    let pairs: [((bool, bool), fn(...) -> (...) ); 6] = [
        ((i != 0.0, r != 0.0), |_, i, r, _| (i * i * r, i, r, i * r)),
        ((i != 0.0, v != 0.0), |_, i, _, v| (i * v, i, v / i, v)),
        // ... 4 more entries
    ];
    for (check, solve) in pairs {
        if check.0 && check.1 { return solve(p, i, r, v); }
    }
    (p, i, r, v)
}
```

The file structure mirrors this separation:

```
src/
  main.rs    — CLI flag parsing, data-driven flag table
  calc.rs    — Pure functions (no dependencies)
  gui.rs     — iced 0.12 desktop app (behind #[cfg(feature = "gui")])
  daemon.rs  — stdin/stdout JSON API (behind #[cfg(feature = "daemon")])
             — tool registry array + macro-based response builder
```

The Daemon: JSON Over Stdout

The original VB6 program required a human sitting at a Windows PC. The daemon mode changes that. It reads one JSON object per line from stdin and writes one JSON object per line to stdout. Status messages go to stderr so they never pollute the pipe.

```
$ echo '{"tool":"ohm","current":2,"resistance":4}' | mcp-relational-data --daemon
{"tool":"ohm","power":"16 Watts(W)","current":"2 Amps(A)",
 "resistance":"4 Ohms(\u03A9)","voltage":"8 Volts(V)"}
```

This makes it trivial to call from Python, Node.js, shell scripts, or any MCP-compatible server wrapper. Build a slim daemon-only binary that skips the iced GUI dependency entirely:

```
cargo build --release --no-default-features --features daemon
```

The daemon uses a `TOOLS` constant array as its dispatch registry. Each entry holds the tool name, description, parameter list, and a handler function pointer. Adding a new calculator means appending one entry — no match arms, no dispatch logic to touch. A `tool_response!` macro handles the repetitive BTreeMap boilerplate.

The iced 0.12 Adventure

Using iced 0.12 was its own challenge. The version available had API gaps compared to the documentation — no `Container::border()`, no `container::Style` or `container::Status` types, no `rule::horizontal()`. Every border, separator, and styled container I tried to add failed to compile.

The solution was brutal minimalism: stick to the APIs confirmed to work (`container`, `column`, `row`, `text`, `text_input`, `button`, `scrollable`) and use spacing and padding to create visual separation. It ended up looking cleaner than bordered panels would have.

There was also a subtle lifetime error: using `&String::new()` as a static reference for the break-even calculator's empty output field caused a dangling reference at compile time. The fix was trivial — use `""` (a `&'static str`) instead — but tracking it down took a few rounds of compiler error archaeology.

Quality Assurance Pass

After the initial port, I ran a full QA pass with production-readiness standards in mind. The goal: eliminate deep nesting, remove dead code, replace branching with data where appropriate, and make every comment and doc string sound like a deliberate decision rather than an afterthought.

Key changes from that pass:

- **Ohm's Law solver** — replaced six nested `if/return` guards with a dispatch table array. Each entry pairs a boolean check tuple with a solve function. First match wins. Flat, data-driven, no nesting.
- **GUI TokenPrev bug** — both the prev and next buttons called `next_model()`. The prev button now correctly calls `prev_model()`.
- **Dead code removal** — `print3d_calculate` returned a `prints_to_payoff` field that was always `0.0` (printer capital cost was out of scope). The field is gone from both the function signature and the GUI.
- **Parse helper extraction** — eight `recalculate()` implementations each repeated `self.x.trim().parse().unwrap_or(default)`. A single `parse_f64()` helper eliminates that duplication.
- **Daemon tool registry** — the 8-arm `match` dispatch is now a `TOOLS` constant array. A `tool_response!` macro handles the BTreeMap construction boilerplate. Adding a calculator = one array entry + one handler function.
- **CLI flag table** — `main.rs` replaced four cascading `if args.iter().any(...)` checks with a `FLAGS` array evaluated top-to-bottom. First match wins. Step-down flow.

Why AGPL?

> *"If you host this as a network service and modify it, you owe the world the source. Consider it a 25-year-later joke at the VB6 original that was locked in a `.frm` file on a floppy disk."*

The original program was proprietary by default — not because I made a deliberate licensing choice, but because that's what VB6 projects were. One file, one IDE, one OS. You couldn't build on it, extend it, or integrate it into anything else.

AGPL-3.0 is the opposite extreme. Fork this calculator, wrap it in a web service, and charge people to use it? Publish your modified source. It's a deliberately ironic choice for a program whose original incarnation was the least open thing possible: a compiled binary I distributed on a physical disk in 1999.

Practically, it doesn't matter. This is a calculator. Nobody is going to build a SaaS around Ohm's Law. But the license file is 662 lines long and it's there, and that's the point.

What's in the Box

The final deliverable is a Rust project with everything needed to build, run, and extend it:

- `Cargo.toml` — feature-gated dependencies (gui + daemon default, either can be disabled)
- `src/main.rs` — CLI entry point with data-driven flag table (`--daemon` , `--help` , `--version`)
- `src/calc.rs` — eight pure calculation engines + AI model pricing data
- `src/gui.rs` — iced 0.12 desktop app, two-column layout, 8 panels, reactive updates
- `src/daemon.rs` — stdin/stdout JSON protocol, tool registry array, `tool_response!` macro
- `README.md` — full documentation with protocol reference, pair-matching table, VB6 bug list
- `QUICKSTART.md` — zero-to-running in 7 steps with Python, shell, and Node.js examples
- `LICENSE` — the full GNU AGPL-3.0 text, all 662 lines

Try It

```
git clone https://git.dcos.net/dcosnet/MCP-Relational-Data.git
cd MCP-Relational-Data
cargo run                          # GUI mode
echo '{"tool":"ohm","voltage":12,"resistance":4}' | cargo run -- --daemon
```

A 1999 VB6 calculator I wrote as a teenager, now a modern Rust application with a desktop GUI, a headless JSON daemon, eight calculators, and the most aggressively copyleft

license I could find. The floppy disk would be proud.