ENGINEERING PORTFOLIO

Kalan *Brunell*

FPGA and embedded systems. Securing avionics data buses at MITRE; leading the battery-management redesign for Tufts Electric Racing.

B.S. Computer Engineering · Tufts University · Class of 2027 · Boston, MA · Seeking new-grad roles, May 2027

kalanbrunell.site · kbrunell06@gmail.com · github.com/kalanbrunell · LinkedIn

August 2026

About.

Hi, I'm Kalan.
I'm a fourth-year Computer Engineering student at Tufts University. I enjoy working where hardware meets software, using skills within FPGA design, embedded systems, and hardware security. Currently, I'm designing hardware to secure avionics data buses as a cybersecurity engineering intern at MITRE, and building a custom battery-management board to replace the off-the-shelf modules for the next iteration of our electric race car at Tufts Electric Racing.

Outside of professional and academic work, I enjoy tinkering with personal projects, like custom keyboards, fun homelab setups, and small embedded systems challenges. Otherwise, you'll find me outside on a bike, hiking, or skiing.

Skills

PROGRAMMING	C · C++ · Python · MicroPython · ARM Assembly · OpenCV
DIGITAL DESIGN & FPGA	VHDL · SystemVerilog · FPGA · Vivado / Vitis HLS · GTKWave · cocotb
EMBEDDED	STM32 / ARM Cortex-M · ATmega · RTOS · PlatformIO
BUSES & PROTOCOLS	CAN · RS-485 · UART / SPI / I2C · OBD-II · Ethernet/IP · Custom Protocol Design
SECURITY	Applied Cryptography · Firmware Hardening · Fault Injection Research
PCB & ELECTRONICS	KiCad · Altium Designer · LTspice · Test & Measurement Instrumentation
SYSTEMS & NETWORKING	Linux / Ubuntu Server · Docker & Compose · Tailscale · Caddy · restic / Backblaze B2
ENGINEERING TOOLS	SolidWorks · Onshape · 3D Printing · Fanuc Robot Programming

Education & Awards

B.S. Computer Engineering
Expected May 2027

Tufts University — School of Engineering

GPA: 3.6 / 4.0 · Dean's List

Relevant coursework: Advanced Computer Organization, Digital Logic, Real-Time Embedded Systems, General Embedded Systems, Data Structures, Circuits & Electronics, MITRE Embedded Capture the Flag.

Dean's List
2023 · 2024 · 2025

FRC World Championship Qualifier
2023

Led Lakerbots 8046 to the FIRST Robotics World Championship as Team Captain and Lead Engineer.

Xerox Award for Innovation & Information Technology
2023

Contents.

Experience

Avionics Cybersecurity Intern MITRE · Resilient Cyber Aerospace Testbed · Bedford, MA · May 2026 — Present
FPGA and embedded development at MITRE's RCAT Lab: SystemVerilog hardware that secures avionics data buses, plus the companion microcontroller platform that configures and runs it.

Manufacturing Automation Engineering Intern Gentex Corporation · Manchester, NH · Summer 2025
Summer 2025 internship at Gentex Corporation: vision-system development, prototype automation cells, and manufacturing engineering support.

Projects

01 FSAE Accumulator BMS & Embedded Electronics IN PROGRESS
450 V lithium-ion accumulator for Tufts Electric Racing's Formula-Hybrid car: custom STM32 BMS control board, isolated RS-485 sensor network, and cell-voltage monitoring chain.

02 5-Stage Pipelined ARM Processor
A complete LEGv8 pipelined processor in VHDL, with hazard detection, data forwarding, and full waveform verification.

03 Custom Mechanical Keyboard IN PROGRESS
A fully custom 65% mechanical keyboard: STM32G0B1 PCB designed in KiCad with USB-C, hot-swap sockets, ESD protection, and a diode matrix for N-key rollover, running QMK.

04 MITRE Embedded CTF
2026 MITRE eCTF: Tufts' secure Hardware Security Module in firmware for the TI MSPM0L2228, plus attack-phase work: a UART man-in-the-middle attack, and timing groundwork toward fault injection.

05 Post-Quantum Cryptography Acceleration
Benchmarking an ML-KEM-512 Vitis HLS accelerator on an AMD Alveo U280 against CPU baselines simulated in Sniper, then attempting two hardware optimizations of our own.

06 FPGA 2048
The 2048 game implemented entirely in VHDL on an iCE40 FPGA: a custom VGA controller, tile-ROM graphics pipeline, and NES-controller input.

07 Autonomous Color-Following Vehicle
Junior design project: ATmega-based autonomous vehicle that follows a colored line with a custom photodiode array and a PID control loop, plus WebSocket telemetry.

08 Competitive Robotics Vision & Control
Real-time vision-guided targeting, swerve-drive control, and IMU-based platform balancing for FIRST Robotics, 2022 and 2023 seasons.

09 Car RPM Shift Indicator IN PROGRESS
A 16-LED shift indicator driven by live OBD-II engine data: SK9822 bar, STM32 firmware, working on a breadboard, with a custom CAN + BLE board designed for version 2.

10 Homelab & Self-Hosting
A single always-on mini PC running my personal services in Docker: photos, documents, monitoring, and a set of web apps I built and deploy with an AI-assisted workflow.

EXPERIENCE · AVIONICS CYBERSECURITY

Avionics Cybersecurity Intern, MITRE

Role: Avionics Cybersecurity Intern · **Lab:** RCAT Lab · **Location:** Bedford, MA · **Dates:** May 2026 — Present

> Building operational prototypes that bring modern cybersecurity to avionics data buses, both legacy and new. Most of the work is FPGA development in SystemVerilog, as well as embedded software development and physical hardware design.

Where I work: MITRE is a not-for-profit that operates federally funded R&D centers, doing research in the public interest. The RCAT Lab sits within this, performing cybersecurity research for aerospace and embedded systems, among other domains. I primarily work on the hardware side of the lab's projects.

The problem

Aircraft have shifted from a small number of analog systems on dedicated point-to-point wiring to many more digital systems, far more interconnected. That change buys enormous capability, and it also means connectivity now matters in places where it originally didn't. Much of it runs over avionics data buses designed for reliability and determinism in a closed environment, long before aircraft were thought of as cyber targets. The interesting part is figuring out what modern security principles look like under those constraints, rather than importing IT solutions wholesale.

Two constraints shape everything. These buses run under hard real-time guarantees, so whatever security gets added cannot disturb their timing. And aircraft stay in service for decades, so ripping everything out and starting over is not realistic. But in order to build cyber mitigations capable of protecting every platform, each design decision has to work both ways: retrofittable onto the legacy systems that already fly, and worth building into new designs from the start.

That is the job: build prototype hardware that secures avionics data buses under tight, real-world design constraints.

What I build

Looking at the hardware I work on, the security processing runs on an FPGA, written in SystemVerilog. This work has to live in hardware: meeting bus timing means handling traffic at wire speed, with latency that is fixed and predictable, and that is exactly what an FPGA is for.

Not all of the FPGA work is the security logic. A design nobody can configure or read data out of is a demo, so I also implement the host-facing interfaces on it, USB and SPI, which carry configuration down to the design, control it while it runs, and move captured data back up to a host.

Next to the FPGA sits a companion STM32 running a firmware package I write, with a MicroPython GUI layer on top of it. That is what configures the FPGA, controls it during testing, and runs demos for people who want to see the approach working live, not just a passing testbench.

How the work actually goes

Nothing goes on the board until it has survived simulation. Each piece of IP gets written in SystemVerilog and verified with cocotb testbenches, Python test code that drives the design in simulation and checks its behavior cycle by cycle. Only then does the design go onto real hardware, brought up in Vivado on AMD Xilinx FPGAs.

Hardware rarely behaves exactly like the simulation, and the gap between the two is where most of the debugging lives. My main tool there is the ILA, a logic analyzer built into the FPGA fabric that captures real signals on-chip. For this work the ILAs earn their keep in one place above all: checking the design against the real timing of the data bus, not the idealized timing of a testbench.

Experimenting with AI in the lab

Separate from the avionics work, I've been running an experiment: how far can AI agents go in a hardware lab? Most AI-assisted engineering stops at the code editor, and I wanted to see what happens when the agent can reach the bench. Through SCPI/VISA scripting, an agent can drive an oscilloscope or logic analyzer directly: set up a measurement, trigger a capture, read back the data, and interpret what it sees. If a capture raises a question, it can adjust the instrument and look again, the same loop I would run by hand.

Three uses have stuck. Repetitive measurement sequences that used to mean standing at the bench turning knobs now run on their own. During bring-up, the agent works as a second set of eyes, watching captures alongside me and flagging things worth a closer look. And an automated documentation workflow, where an agent takes notes on what I'm working on as I go, means context that used to evaporate at the end of a debug session gets written down.

What I'm learning

The biggest shift is the difference between building something for a class and building something real. A class project ends when it works once. Here the work has to be something other people can use, understand, and build on.

Security work also rewires how a design gets read. Coursework trains you to ask whether a circuit works; this job trains you to ask how it could be broken or abused. That reflex, looking at an interface and wondering what an attacker could do with it, never came up in any class, and it changes how everything else gets designed.

The bar for correctness is different too. Working once in simulation counts for nothing; a design is done when the testbench proves it handles the cases nobody expects. And hours of staring at ILA captures have built a feel for real on-chip behavior that simulation alone never taught, the kind of intuition that says where to look when the hardware misbehaves.

EXPERIENCE · MANUFACTURING AUTOMATION

Manufacturing Automation Engineering Intern, Gentex Corporation

Role: Manufacturing Automation Engineering Intern · **Location:** Manchester, NH · **Dates:** Summer 2025

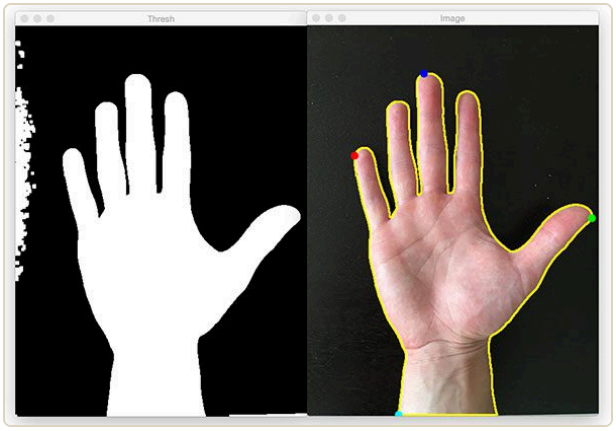
> A summer building a standalone 2D computer vision system for production-line inspection, integrating it with Fanuc robots and existing PLC infrastructure, and supporting day-to-day manufacturing engineering across multiple product lines.

The project

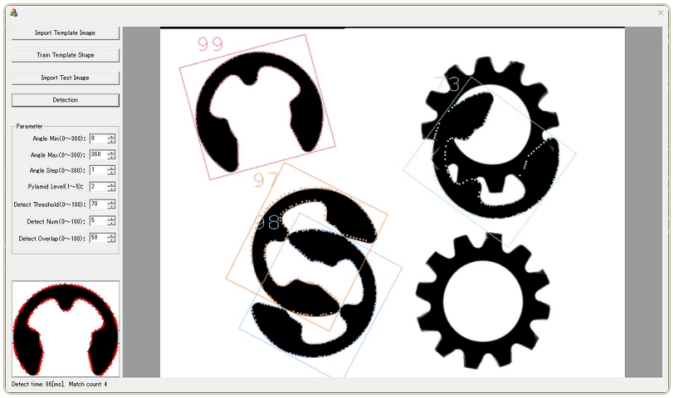
The project was an automated vision inspection system for a production inspection step done by hand. Working with the manufacturing engineers, I identified the points where computer vision could improve accuracy and throughput, then built the system around them, knowing it would eventually have to hold up in a real manufacturing environment rather than a lab.

The prototype was built in Python using OpenCV and deployed on a Raspberry Pi 5 as a controlled-environment proof of concept. Under the hood it combined contour detection and template matching to compare captured images against predefined quality tolerances. A custom GUI added real-time monitoring, inspection logs, and operator controls, so the prototype could run as a self-contained station during evaluation.

In that controlled setup I measured the system's detection accuracy against the accuracy typically reported for a person doing the same repetitive visual check, and the camera came out up to **60%** better. The two halves of that comparison came from different places: my figure off the bench, the human figure from published data rather than a measurement of the line the system would replace. It was enough to justify moving to an industrial platform.



OpenCV contour creation, used for feature isolation in the inspection pipeline.



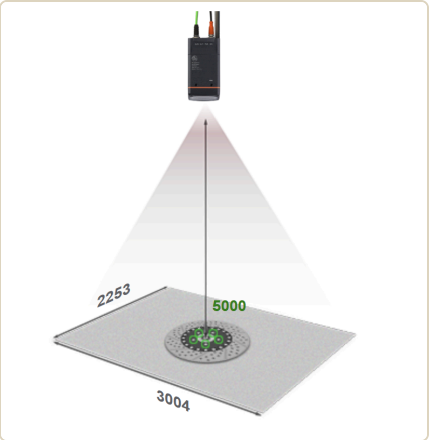
Template matching: comparing live captures against a stored reference.

Moving to an industrial platform

Once the prototype had proved out the approach, the next step was an industrial-grade vision platform. The selection balanced performance, ease of integration, and budget against Gentex's manufacturing environment, and landed on a system that could talk to the existing PLCs and robot controllers over Ethernet/IP and industrial I/O modules.

The industrial platform added adaptive thresholding, morphological operations for feature enhancement, and lens-distortion correction via camera calibration, so parts could be measured precisely under real production lighting.

With the vision system in place, I worked with the manufacturing engineering team to design and prototype automation cells: programming Fanuc robots to handle parts, integrating the vision system for real-time inspection, and designing custom end-effectors for part handling.



Vision-based inspection example: the kind of in-line measurement the system was built for.

Two vendors with no supported way to talk

The vision platform and the Fanuc controller had no plug-and-play path between them, or at least not one that could be stood up inside the internship's timeframe. Until they could talk, the cell could not act on an inspection result, which stalled everything downstream of the camera.

So I wrote Python firmware for a microcontroller that sat between the two and translated Ethernet/IP traffic into discrete I/O the Fanuc could consume. Nobody would ship that as the permanent integration and it was never meant to be one. What it did was give the cell real-time inspection results and object-location feedback during testing, so the automation work kept moving while the clean integration stayed a later problem.

That is where the most useful lesson of the summer came from. Off-the-shelf components from different vendors do not automatically work together, and the gaps between them get closed by whoever needs the system to run. The schedule turned out to be a design constraint like any other: the real choice was not between a good integration and a bad one, it was between the clean integration that would not land in time and the workaround that would.

What I learned

This was my first industry experience, and plenty of the learning had nothing to do with the vision system. A class project has one audience: a grader. Here the daily audience was manufacturing engineers, operators, and technicians, and progress depended on knowing whose concerns mattered at which stage.

The lesson I use most came from sitting on the receiving end of other people's designs. Manufacturing sees parts as they actually arrive: geometry that fights the fixture, features that are hard to present to a camera, steps that are easy for a hand and awkward for a robot, drawings that have drifted from what is on the floor. Each of those becomes someone's custom fixture, end-effector, or workaround. Design for manufacturing stopped being a chapter heading and became a habit.

Computer vision in a working factory is also a different problem from computer vision in a lab. The prototype ran in a controlled environment and behaved. Production offered changing lighting, parts arriving in slightly different positions, and surfaces that reflect or vary enough to break clean feature extraction. Much of the real engineering was making the pipeline tolerant of that variation instead of assuming it away.

IMAGE CREDITS

- **Contour example:** PyImageSearch — "Finding extreme points in contours with OpenCV" (Apr 11, 2016).
- **Template matching example:** ren482, "Fast Shape-based Template Matching," DEV Community (Dec 22, 2024).
- **Vision-based inspection example:** ifm electronic gmbh, "Area calculator illustration" (© ifm).

Images used for portfolio demonstration purposes only. Gentex logo used for identification only — Gentex Corporation is not affiliated with this site.

01 / FSAE · BMS & EMBEDDED SYSTEMS

FSAE Accumulator BMS & Embedded Electronics

Team: Tufts Electric Racing · **Competition:** Formula-Hybrid Electric · **Role:** Accumulator Project Manager (2025 — Present), prev. Electrical Engineer (2024 — 2025) · In Progress

> Designing the safety-critical electronics for a competition-grade EV accumulator: a custom STM32 BMS control board, an isolated RS-485 sensor network, and the cell-voltage monitoring chain that keep a 450 V pack safe, serviceable, and FSAE-compliant.

Current accumulator May 2025 — Present

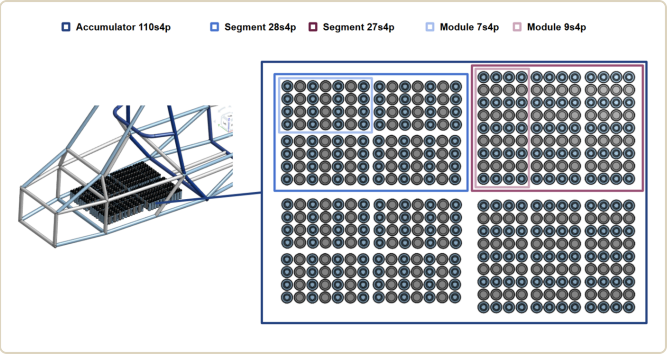
As Accumulator Project Manager, I lead the electrical and embedded design for our Formula-Hybrid Electric accumulator: a fully removable **440-cell, 450 V** pack built around Molicel P42A cells. My focus is the pack's electronics, a custom STM32 BMS control board, an isolated RS-485 network to four distributed thermistor modules, and the cell-voltage sensing chain, replacing the chain of off-the-shelf BMS modules our previous pack relied on. It's still in progress, with the full system targeting competition readiness by the end of 2026: the architecture and wire protocol are settled, the boards are through schematic, and the active work right now is firmware.

PACK & TOPOLOGY
440-cell, 450 V pack split into 4 segments, each with local sensing, built around Molicel P42A cells.

BMS CONTROL BOARD
STM32G474 master board that aggregates sensor data, runs the AMS fault state machine, and reports telemetry over vehicle CAN.

RS-485 THERMAL NETWORK
Custom master-polled protocol over isolated RS-485, linking 4 distributed thermistor modules for real-time temperature mapping.

CELL-VOLTAGE CHAIN
8 Enepaq tinyAFEs (2 per segment) daisy-chained over isolated UART, feeding per-cell voltage into the same fault logic.



Current pack arrangement, packaged for full-unit removability and serviceable segments.

The pack itself stays focused on serviceability: segments are removable as a unit, and the harnessing is laid out to keep the control board and sensor network accessible without breaking down the whole accumulator.

The sections below go deeper on the electronics I own directly: the control board, the RS-485 sensor network, and the cell-voltage chain.

BMS control board

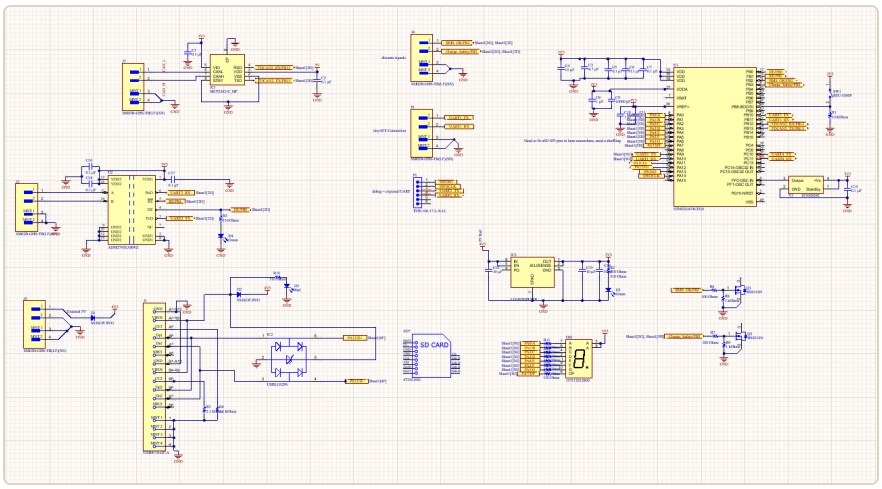
The control board is built around an STM32G474CEU6, sitting between the sensor network and the rest of the car. It owns the RS-485 bus to the thermistor modules and the isolated UART link to the cell-voltage AFE chain, evaluates both against configured thresholds, and drives `BMS_OK` and `Charge_Safety`. Vehicle telemetry goes out over FDCAN2 through an MCP2562-E/MF transceiver.

Both safety outputs are driven through low-side N-channel MOSFETs, so the line is only held while firmware is actively driving the gate. A board that loses power, hangs, or never finishes booting releases the FET on its own and the safety loop opens. The fail-safe behavior falls out of the circuit instead of depending on software noticing that something has gone wrong. The trip is then latched by the physical safety loop rather than in firmware, which means the BMS only has to report its current error state continuously and correctly. It never has to remember a fault across a reset.

Everything crossing into the pack is isolated. The thermistor bus runs through an ADM2795EARWZ, an isolated RS-485 transceiver with fault-protected bus pins, and the cell-voltage data arrives over an isolated UART link, so the low-voltage control side and the 450 V side never share a ground.

For diagnostics, the board has USB-C and SWD for config and flashing, a single-digit display for an at-a-glance fault code without a laptop attached, and a microSD socket for local telemetry logging. Logging is designed to run asynchronously at low priority so a slow card write can never stall the safety loop.

The schematic below is mid-revision, and the open items are visible in it.



BMS control board schematic, in progress.

Custom RS-485 thermistor network

Rather than home-running every thermistor back to the control board, four distributed STM32G474 thermistor modules, one per segment, sit on a shared isolated RS-485 bus, a multi-drop daisy chain over a single differential pair. I designed the wire protocol myself: half-duplex and master-polled, so the control board always owns the bus and the remote modules never have to arbitrate. The bus runs at 115200 bps over isolated transceivers, and a full four-module poll cycle costs about 16 ms at that rate, which leaves the bus idle most of the time at the refresh rate the fault logic actually needs. The baud can go up later without touching the protocol.

```
+------+------+------+------+-----------+--------+--------+
| SOF  | ADDR | CMD  | LEN  |  PAYLOAD  | CRC_HI | CRC_LO |
|  0   |  1   |  2   |  3   |   4..N    |  N+1   |  N+2   |
+------+------+------+------+-----------+--------+--------+
```

FIELD	SIZE	WHAT IT CARRIES
SOF	**1 byte**	Fixed 0xAA sync marker, so a receiver that comes up mid-stream can find a frame boundary.
ADDR	**1 byte**	0x00 to 0x03, one per segment module, set in hardware. 0xFF is a broadcast with no reply.
CMD	**1 byte**	Command identifier. A module's reply is the request with the high bit set, so a response is never mistaken for a poll.
LEN	**1 byte**	Payload length, zero when there is none. The parser knows how much to read before it reads it.
PAYLOAD	**LEN bytes**	Command-specific. A temperature summary is 8 bytes; a ping response is 1 byte of status flags.
CRC	**2 bytes**	CRC-16/CCITT-FALSE over bytes 0 through N, high byte first, computed in the STM32G4's CRC peripheral.

Every frame on the bus, in both directions, has this shape.

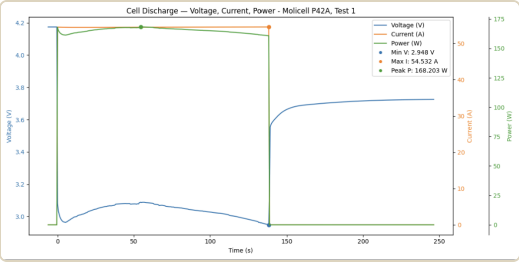
The command set covers polling for a temperature summary, a ping/health check, pushing threshold configuration to a module, and a broadcast command to clear latched faults across all four modules at once. Each module reports back its min, max, and average temperature plus a fault bit field (over-temp, under-temp, sensor fault, stale data), so the control board's fault logic sees per-segment context instead of a single aggregate flag. If a module misses three consecutive polls it's marked `COMMS_LOST` and treated as a hard fault, but the control board keeps polling the rest of the chain so one dead node can't stall the bus for everyone else.

The two ends of that bus run different concurrency models, for reasons that come from what each one has to do. The control board runs FreeRTOS because it juggles four interfaces at once, RS-485, CAN, the AFE UART, and USB, with the fault logic sitting on top of all of them. A thermistor module does exactly two things: scan its sensors on a timer, and answer when it is polled. It runs a bare-metal superloop. An RTOS there would buy nothing and add one more thing to audit on a board whose value is being simple enough to trust.

Cell-voltage chain

Cell voltage comes from 8 Enepaq tinyAFEs, two per segment, on their own isolated UART daisy chain, separate from the thermistor bus. Each AFE is referenced to the lowest cell it monitors, and the head AFE in the chain talks to the control board through Enepaq's proprietary-to-UART converter. That voltage data feeds into the same fault logic as the thermal data, so over-temperature and over-voltage conditions are evaluated together rather than as two separate systems bolted side by side.

Cell testing & validation



One load profile on the single-cell testbench: voltage, current and power against time, run at the peak draw the car was calculated to ask for.

The pack architecture started as arithmetic, and the arithmetic ran off the end of the datasheet. Sizing peak current around the motor we were running at the time put our worst case above the range Molicel characterizes the P42A over, so the published curves could not tell us what a cell would actually do at our peak. That is not something to find out with a finished pack.

So we built a single-cell testbench and measured it. Cells get discharged under profiles shaped like an event rather than a constant-current sweep, at the current the car would really pull, and we watch how far the voltage sags the moment the load lands and how much of it comes back when the load releases.

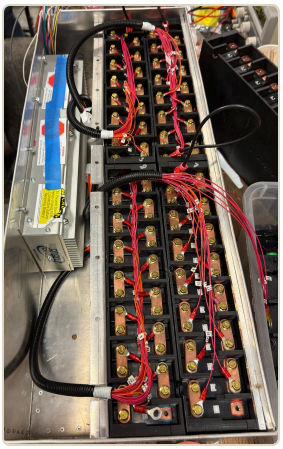
That measurement is what the pack gets configured against. How far a cell sags decides how much of the pack's nominal voltage is still there at the end of a run, so the series and parallel arrangement has to be chosen from measured behavior rather than a datasheet curve, inside what the Formula-Hybrid Electric rules allow and what the car needs. How hard the same profile heats the cells feeds the cooling design and the temperature thresholds the BMS gets configured with.

Previous accumulator 2024 — 2025

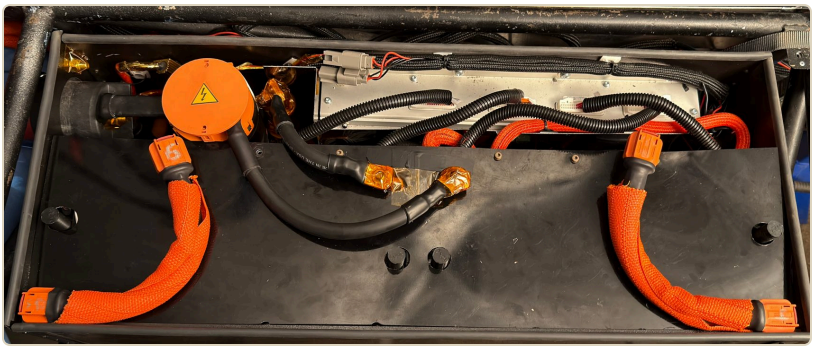
I joined the team as an Electrical Engineer on the previous generation of the pack. My work sat on the high-current side: the path from the cells out to the pack connector, the modules that path runs through, and getting a module built and safely energized for the first time. I also designed the two low-voltage boards the car's safety systems ran through.

First assembly came after months of design and build, and it is the point where the paper pack becomes a real one. Getting a module assembled, checked, and energized for the first time is a deliberate, procedural exercise: it is the first time the safety systems are asked to do their job with live cells behind them rather than a bench supply.

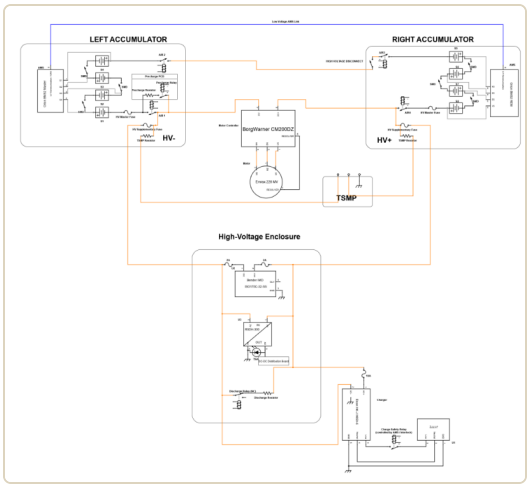
From there the module went into the car's sidepod as a prototype, which tested the mechanical packaging and the high-voltage connections in their real positions and let us run charge and discharge testing under load.



First successful assembly of an accumulator module.



Prototype module deployed in the sidepod, testing packaging and high-voltage connections in place.



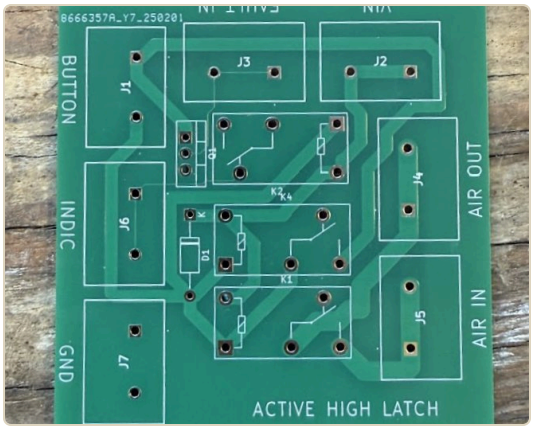
The previous car's tractive system. The high-current path leaves the two accumulators through the AIRs and master fuses on its way to the motor controller, with the insulation monitor, discharge circuit and charger interlock in the high-voltage enclosure. Click for full resolution.

Low-voltage safety boards

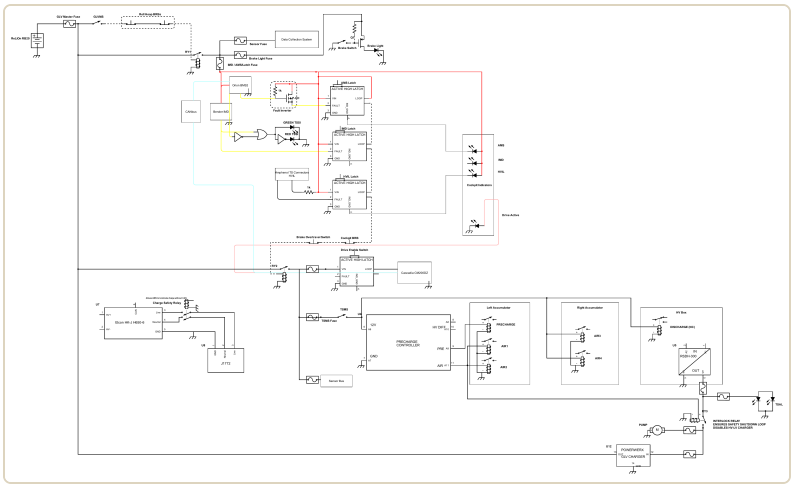
The fusebox splits and protects the car's low-voltage supply. Every circuit gets its own blade fuse with an indicator LED beside it, so a blown fuse is visible at a glance instead of being tracked down with a meter. The same board carries the shutdown-circuit signals alongside the power rails, including the TSMS and AIR lines, the big red button loop, the ESOK line, and the BMS and IMD outputs, plus the 3.3 V and 5 V regulators feeding the boards downstream.

The interlock latches keep the safety loop energized only while no faults are present. I designed active-high and active-low versions so the same latch logic could sit behind sensors of either polarity, covering motor controller, BMS, and high-voltage connection fault signals. Each latch powers up in the disconnected state and only closes once every safety condition is met, and any fault or loss of power drops it immediately. They are fail-safe by construction: the de-energized state is the safe one, so nothing has to run correctly for the car to end up safe.

The populated fusebox. One fuse and one indicator LED per circuit, with the shutdown-circuit signals labeled on the silkscreen.



The active-high latch board, bare. FAULT IN gates the AIR path, and the board powers up open.



The car's low-voltage and shutdown circuit. The active-high latch blocks through the middle are those boards, gating the loop on the BMS, insulation-monitor and high-voltage interlock fault signals before the precharge controller is allowed to close the AIRs. Click for full resolution.

What I've learned

FSAE was the jump from class projects that only have to work once to engineering for a real system. An accumulator has to survive the conditions it was designed for, fit the car, follow the rules, and stay understandable to the people who build, test, and service it. The BMS is not a board in isolation: the control board, the thermistor network, the cell-voltage chain and the safety outputs all have to work together at the moment the car needs them.

The technical work changed the order I think in. The happy path turns out to be the easy part of a protocol. The design decisions live in what happens when a module goes quiet, when a frame arrives corrupted, when a response is late, and none of them would have come out of the feature list.

Leading the program has made the non-schematic parts of engineering just as real. Budget and part availability can change an option, serviceability can rule out an elegant layout, and a competition deadline decides what gets designed now and what is deferred. Underneath all of it is execution: the team is building a safety-critical system in its spare time while carrying full course loads, so complexity has to come down to something we can actually finish, validate and support. That is why the architecture favors clear ownership and bounded interfaces, a master-polled sensor bus instead of distributed arbitration, repeatable module designs instead of one-off

assemblies. The lesson is not to make a design simpler for its own sake. It is to make the right complexity manageable for the team that has to deliver it.

02 / DIGITAL DESIGN · VHDL

5-Stage Pipelined ARM Processor

Course: Computer Architecture · **Discipline:** Digital design in VHDL · **Date:** 2025

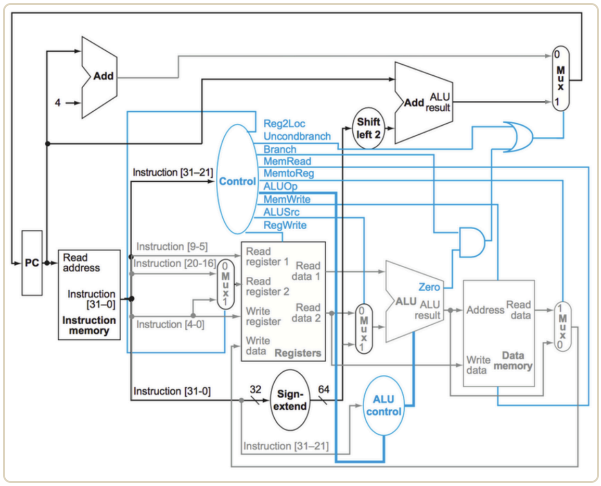
GitHub: github.com/kalanbrunell/5-Stage-Pipelined-Processor

> A complete LEGv8-subset ARM-style processor in VHDL, built from the ground up as a single-cycle datapath, then pipelined into five stages with hazard detection and data forwarding, and verified via waveform analysis.

Overview

As the term project for my Computer Architecture course, I designed and implemented a simplified ARM-like processor using VHDL. The processor supports a subset of ARM instructions (specifically LEGv8), including arithmetic operations, memory access, and control flow.

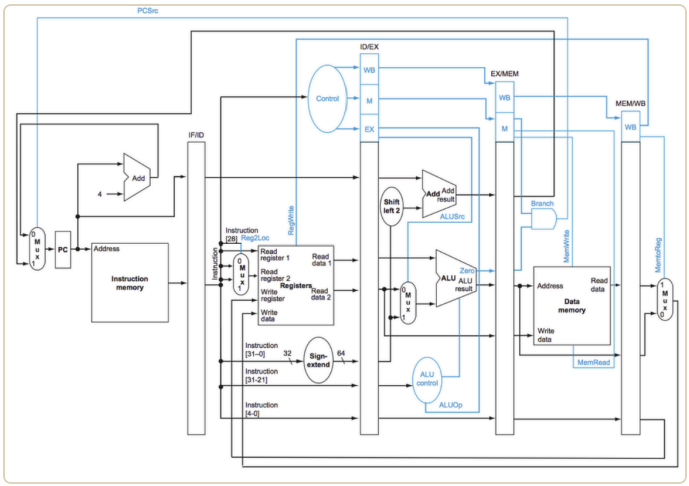
The project began with the foundational components: PC, instruction and data memory, register file, and ALU. I first wired these together into a single-cycle architecture to validate basic functionality before transitioning to a pipelined design. That shift introduced challenges around instruction dependencies and timing, which I addressed through hazard detection and data forwarding.



Single-cycle datapath, the starting point: one instruction per clock, no pipeline registers.

Pipelining the datapath

The pipeline follows the classic five-stage approach: Instruction Fetch, Instruction Decode, Execute, Memory Access, and Write Back. Each stage operates concurrently with the others, creating a processing pipeline that can theoretically complete one instruction per clock cycle. The real complexity comes from data hazards: situations where instructions depend on results from previous instructions still moving through the pipeline.



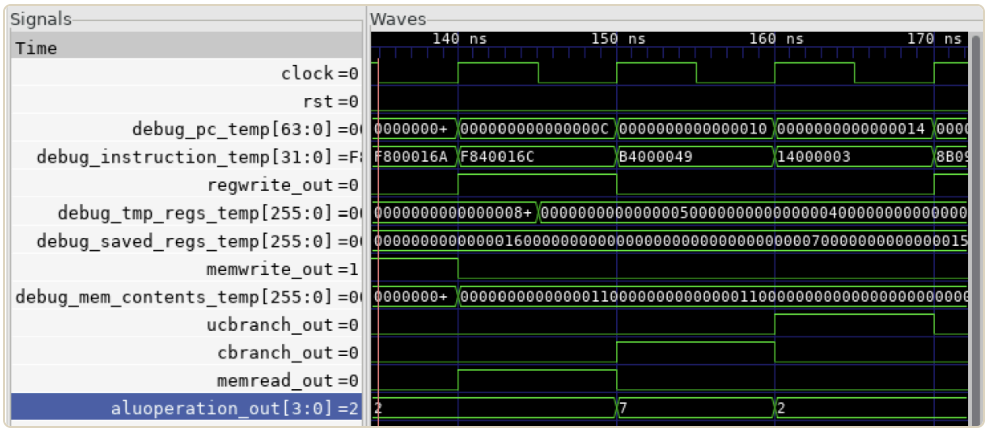
The same datapath cut into five stages by the IF/ID, ID/EX, EX/MEM, and MEM/WB pipeline registers. Control signals ride along in the registers rather than being regenerated per stage.

Most of the bugs that actually cost time weren't in the hazard logic itself. Several pipeline registers and internal signals weren't initialized to a known value at reset, so simulations would run clean until a specific reset timing exposed garbage propagating through the datapath. The rest were classic hazard bugs: read-after-write cases where a dependent instruction read a stale register value before forwarding caught up, and write-after-read cases where the register file wrote its new value before an earlier stage had read the old one. Both took cycle-by-cycle waveform tracing to pin down.

```
library IEEE;
use IEEE.std_logic_1164.all;
  use IEEE.numeric_std.all;

entity SingleCycleCPU is
  port(clk :in STD_LOGIC;
       rst :in STD_LOGIC;
      --Probe ports used for testing
    --The current address (AddressOut from the PC)
     DEBUG_PC : out STD_LOGIC_VECTOR(63 downto 0);
   --The current instruction (Instruction output of IMEM)
     DEBUG_INSTRUCTION : out STD_LOGIC_VECTOR(31 downto 0);
       --DEBUG ports from other components
    DEBUG_TMP_REGS : out STD_LOGIC_VECTOR(64*4 - 1 downto 0);
    DEBUG_SAVED_REGS : out STD_LOGIC_VECTOR(64*4 - 1 downto 0);
   DEBUG_MEM_CONTENTS : out STD_LOGIC_VECTOR(64*4 - 1 downto 0)
  );
end SingleCycleCPU;
```

The single-cycle top level, before it was pipelined. Past the clock and reset, every port on it is a debug probe, so the testbench can watch the program counter, the current instruction, and the register and memory contents from outside the design.



GTKWave simulation waveform, instruction execution traced through the pipeline.

Verification

The debug ports are what let the testbenches check themselves. A testbench loads an instruction sequence, runs it, and compares the register file and memory contents against the end state that sequence should produce, so a failing program reports which value is wrong instead of leaving me to read a waveform and decide. GTKWave came after that, once a check had told me something was broken and I needed to find which stage broke it.

IMAGE CREDITS

- **Single-cycle and pipelined datapath diagrams:** Patterson, David A., and John L. Hennessy. *Computer Organization and Design: The Hardware/Software Interface (Arm® Edition)*. ISBN 978-0-12-801733-3.

Textbook diagram used for educational purposes under fair use. Code screenshots and simulations are personal development work.

03 / HARDWARE · PCB DESIGN

Custom Mechanical Keyboard

MCU: STM32G0B1CEUx · **Firmware:** QMK · **PCB tool:** KiCad · **Date:** 2025 — Present · In Progress

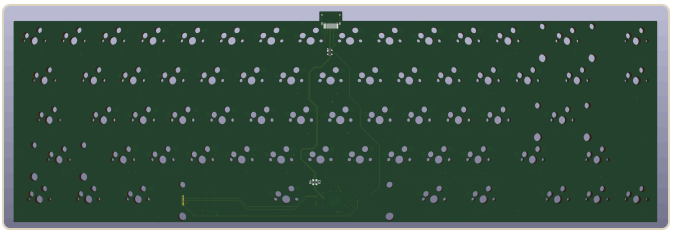
> A fully custom 65% mechanical keyboard designed from the ground up: a KiCad PCB around an STM32G0B1 running QMK, hot-swap sockets, USB-C with ESD protection, and a diode matrix for true N-key rollover. Case design is underway in Onshape, targeting CNC aluminum.

Why build it.

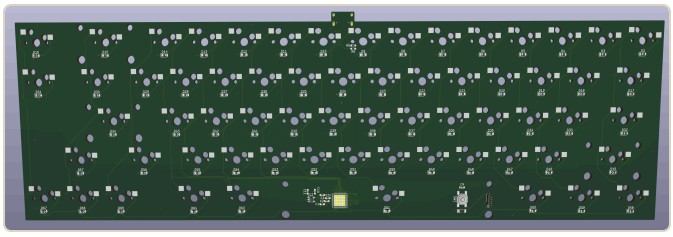
Off-the-shelf keyboards make some decisions for me: the layout, what's on the PCB, the shape of the case. Building one from scratch flips that order. I pick the layout first, decide what goes on the board, and design the case around it instead of the other way around.

It's also a way to learn the full PCB workflow end to end: laying out the board in KiCad, placing the STM32, routing USB-C with ESD protection, and generating gerbers for fabrication. The appeal is that it's immediately verifiable, either the keyboard works or it doesn't. The 65% layout keeps the build compact, but the arrow cluster stays. That's the one thing I'm not willing to drop.

PCB design.



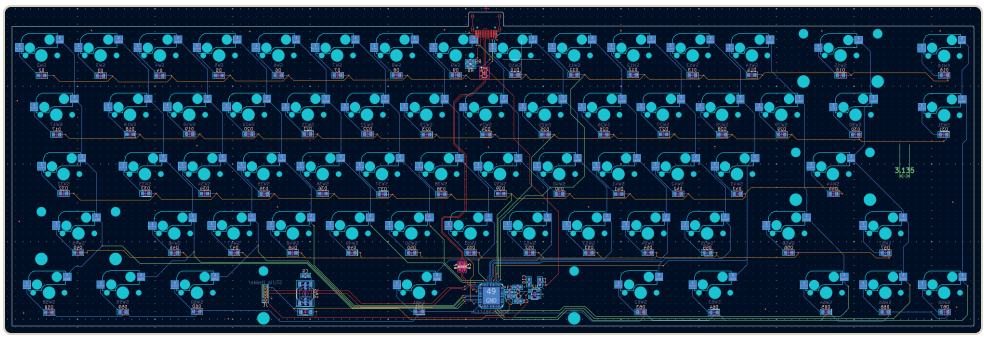
3D render, front side. The USB-C connector sits centered at the top edge.



3D render, back side. STM32G0B1 is center-bottom; diodes and STLink header are visible.

Controller

The board is centered on an **STM32G0B1CEUx**, an ARM Cortex-M0+ with native USB 2.0 full-speed built in. Native USB matters here: the MCU enumerates directly as a USB HID device without a USB-to-serial bridge, keeping the hardware simple and giving QMK full control over the USB descriptor. I'm familiar with the STM32G-family from other projects, so reaching for the G0 here made sense: it has everything a keyboard needs without the cost or complexity of a larger part. QMK supports it via ChibiOS, which handles the low-level RTOS and USB stack.



KiCad routing view, traces, pads, and component placement across the full 65% layout.

USB-C and ESD protection

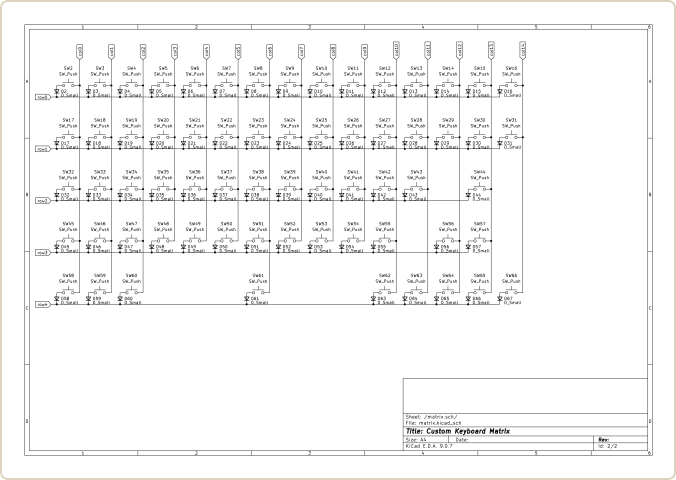
The USB connector is USB-C with ESD protection on the data lines via a PRTR5V0U2X TVS array. Keyboard connectors take a lot of plug cycles and get exposed to whatever static charge walks up to the desk, so protecting the USB differential pair is straightforward insurance.

Hot-swap sockets

The PCB uses hot-swap sockets rather than soldered switches, so I can change switch feel without reflowing the board. From a layout standpoint this means placing socket footprints precisely, keeping the underside of the board clear of traces in the socket area, and orienting each socket consistently so the seating force goes the right direction.

Diode matrix and N-key rollover

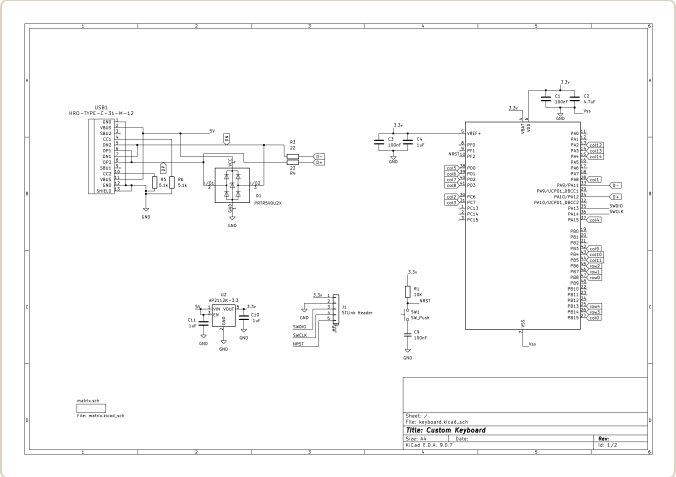
Every switch in the 5×15 grid carries a diode in series, so the scan can't be fooled into a phantom keypress by a reverse path through a fourth key when three corners of a rectangle are held down. Any combination can be held at once, which QMK exposes as N-key rollover over USB HID.



Matrix schematic (sheet 2/2): each switch paired with its series diode, across the 5×15 grid.

Programming header

The PCB includes an STLink SWD header and a dedicated boot-mode button. During development, that means I can flash firmware and step through code with a debugger attached rather than relying purely on the USB DFU bootloader, a convenience that pays for itself the first time something doesn't behave as expected.



Main schematic (sheet 1/2): USB-C, ESD protection, 3.3 V LDO, STLink header, and STM32G0B1 with matrix pin assignments.

Case and plate.

The case is being designed in Onshape to fit the PCB footprint exactly, targeting CNC-machined aluminum. Aluminum also gives better acoustic control than printed plastic: the weight and rigidity damp the higher-frequency ping that cheaper cases emphasize.

Getting a case machined is a stretch goal after the PCB is verified. For the first functional build I'll use a printed case to confirm fit and feel, then move to aluminum once the layout and PCB are locked in. The Onshape model is parametric, so adjustments from the prototype carry forward without a full redraw.

Where it stands.

The PCB design is complete and ready to send to fabrication. Case design is actively in progress in Onshape. Switch and keycap selection is still open; both will be picked once the board is in hand and the layout is confirmed to feel right.

PCB
Design complete in KiCad. Ready for fabrication.

CASE
In progress in Onshape. CNC aluminum is the target; printed prototype first.

SWITCHES & KEYCAPS
TBD, selecting after the board is in hand and the layout is confirmed.

04 / EMBEDDED SECURITY

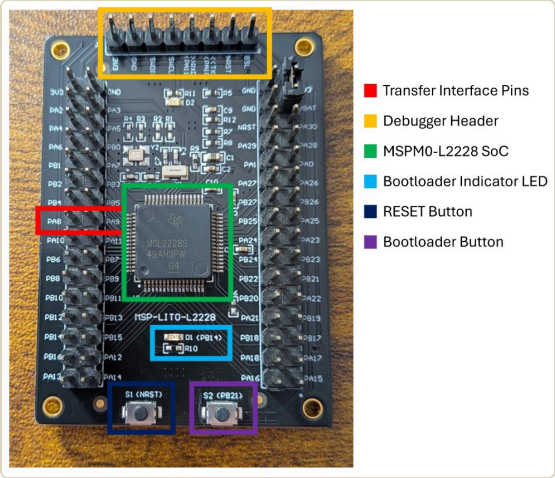
MITRE Embedded CTF

Organizer: MITRE Corporation · **Team:** Tufts University · **Hardware:** TI MSPM0L2228 (Arm Cortex-M0+) · **Dates:** January — April 2026 · **Result:** 19th place

> A two-phase embedded security competition: design a cryptographically secure Hardware Security Module in firmware, then turn around and attack every other team's implementation.

The scenario

The 2026 eCTF challenge tasks each team with building a **Hardware Security Module (HSM)**, a dedicated embedded device for secure file storage and encrypted device-to-device file transfer. The HSM runs as firmware on the TI MSPM0L2228 microcontroller and exposes two distinct interfaces to the outside world.



The TI MSPM0L2228 board used for the competition, with the Transfer Interface pins and debug/bootloader controls labeled. Photo via the eCTF program.

MANAGEMENT INTERFACE
PIN-protected channel for user interactions such as creating files, managing permissions, and retrieving status.

TRANSFER INTERFACE
Device-to-device channel for secure file exchange between HSMs over a direct UART connection.

HOST INTERFACE
Serial connection to a host computer running organizer-provided tools to initiate operations and read results.

The security model is built around **permission groups**. Every file stored on an HSM belongs to a group, and each HSM holds differentiated permissions per group: receive (request files from other devices), write (create new files), and read (return file contents to an authorized user). Crucially, an HSM can store files it has no read permission for, deliberately separating *data possession* from *data access*.

The competition runs in two phases. During the **design phase**, teams implement and harden their HSM firmware, earning flags through an automated testing service that validates both functional correctness and security properties. In the **attack phase**, teams receive boards provisioned with every other team's firmware and attempt to break their security implementations to capture flags. Each team therefore defends its own design while actively attacking everyone else's.

What I worked on

Three pieces of work this season: one hardening change during the design phase, then in the attack phase, one attack I ran against other teams' devices and one I researched without finishing. The unfinished one went after the PIN check, which is the same gate the team spent the design phase trying to make unbreakable.

Hardening: unpredictable memory layout

On the defensive side, I explored **randomizing the order of functions in the compiled binary** so the firmware's memory layout is less predictable. Because the device runs a fixed image, runtime address randomization (ASLR) isn't available, but shuffling function placement at build time is a static analogue that makes it far harder for an attacker to reliably overflow a buffer into a known, useful location.

Attack phase: UART man-in-the-middle

During the attack phase I targeted teams that left their Transfer Interface unencrypted. By sitting in the middle of the UART link between two HSMs, I could intercept and relay traffic to capture sensitive data in transit, directly exploiting the missing confidentiality our own design was built to provide.

Attack research: timing work toward fault injection

The PIN check is the single gate in front of every privileged operation on the device, and a constant-time comparison does nothing to stop you if you can make the processor skip the comparison altogether. That makes **voltage fault injection** the obvious thing to point at it: drop the supply rail for a very short window, at exactly the right moment, and the core mis-executes.

I did not get a glitcher built inside the competition window. What I did do was the measurement work underneath one: bench work on an actual eCTF board, using an external light source as a repeatable trigger reference to establish when the operations I cared about actually happen. The hard part of glitching is not the glitch, it is knowing the moment to fire it, and that timing is the piece that carries forward.

I looked at electromagnetic fault injection as the alternative and settled on voltage glitching, on accessibility grounds: I could build the rig myself, I would have direct physical access to the board's supply rail, and as a first attempt at fault injection it was the easier of the two to reason about. Building it is the plan for next year's competition.

The team's design

For context on what I was defending: the HSM the team shipped encrypted every stored file under a freshly generated AES-CBC key, then wrapped that key with the file group's ECC public key, so a device can hold a file it is cryptographically unable to read. Transfers used a challenge-response handshake, with an AES-CMAC-signed nonce bound to the slot and file-group ID, so a device cannot pull files it has no receive permission for. Both were built on **wolfSSL / wolfCrypt**. Those pieces were a collaborative design effort across the team.

The PIN path is the part worth detail, because it is what the attack phase comes for. PINs are stored only as SHA-256 hashes and verified with a **constant-time comparison**, so a flash leak never exposes the PIN and timing analysis cannot walk it out byte by byte. A brute-force lockout with a cooldown, designed to persist across power cycles, closes the power-cycling shortcut. UART read and write paths are bounded by explicit maximum lengths and slot indices are range-checked, which is what keeps a malformed request from overflowing a buffer or the file table.

Tufts finished **19th** out of a field of roughly 100 teams in the 2026 competition.

05 / COMPUTER ARCHITECTURE · FPGA

Post-Quantum Cryptography Acceleration

Algorithm: ML-KEM-512 (NIST FIPS 203) · **Target:** AMD Alveo U280 (UltraScale+) · **Tools:** Vitis HLS 2023.1, Sniper 7.3 · **Course:** Advanced Computer Organization, Tufts · **Date:** Spring 2026

Paper: kalanbrunell.site/projects/pqc-acceleration-paper.pdf

> Measuring what an FPGA actually buys you on the key exchange that replaces RSA. An ML-KEM-512 accelerator ran a full key-encapsulation cycle 11 times faster than a simulated out-of-order CPU core and 43 times faster than an in-order one, at roughly a ninth of the clock rate and about half the power of the faster core. Then we tried to speed it up further, and mostly found out why we could not.

The problem

Nearly all public-key cryptography in use today rests on factoring large numbers (RSA) or the discrete logarithm over an elliptic curve (ECC), and Shor's algorithm removes the asymmetry both depend on once a large enough quantum computer exists. The threat does not wait for that hardware: traffic captured today can be stored until a machine exists that can open it, so anything with a long secrecy lifetime is already exposed. NIST's answer is ML-KEM, standardized in FIPS 203 and formerly known as CRYSTALS-Kyber. Its hardness comes from the Module Learning With Errors problem over polynomial rings with modulus 3329, believed to resist classical and quantum attacks alike.

ML-KEM was designed to be practical on a general-purpose CPU, and it is. That is a different question from whether it should run there. A data center terminating TLS performs key encapsulation constantly, and full adoption of the standard turns that into an enormous aggregate workload. Two operations dominate the cost: the Number-Theoretic Transform, which moves polynomials into a representation where multiplication is cheap, and the Keccak permutation inside the SHAKE hash functions, which expands a 32-byte seed into the public matrix. The NTT unrolls into layers of independent butterfly operations, and independence is exactly the property that makes a workload worth putting on an FPGA.

Two tracks, measured against each other

The accelerator side started from BSC LOCA's open-source Vitis HLS design rather than RTL of our own, synthesized under Vitis 2023.1 for part xcu280-fsvh2892-2L-e at a 300 MHz target clock. Cycle latency came from a full C/RTL co-simulation rather than a complete Vivado implementation. Driving the real implementation means wrapping the AXI interfaces and supplying clock and reset from a host design, and a first attempt without that wrapper asked for more than 2,000 I/O pins, which no real part provides. Since the comparison is cycles, resources, and power rather than board-level throughput, co-simulation gave the accuracy we needed and left enough time to try different HLS configurations.

The software side used the PQClean implementation of ML-KEM-512, chosen because it is a clean, widely used reference. We wrote a testbench that runs KeyGen, Encap, and Decap in series inside a marked region of interest, so the numbers cover the cryptography and nothing around it. That ran under Sniper 7.3 on two configurations: `gainestown`, a 2.66 GHz out-of-order Nehalem-class core, and `silvermont`, a 1 GHz in-order Atom-class core. Ten trials each, to catch run-to-run variance. I built and ran that software side myself, from the PQClean testbench through both Sniper configurations, and implemented both of the optimization experiments further down this page.

Both sides being simulated is a feature of the comparison, not a compromise in it. No thermal throttling, no OS scheduler, no background load, and neither side gets an advantage the other lacks. The absolute numbers would move on real silicon; the ratio between them is cleaner this way.

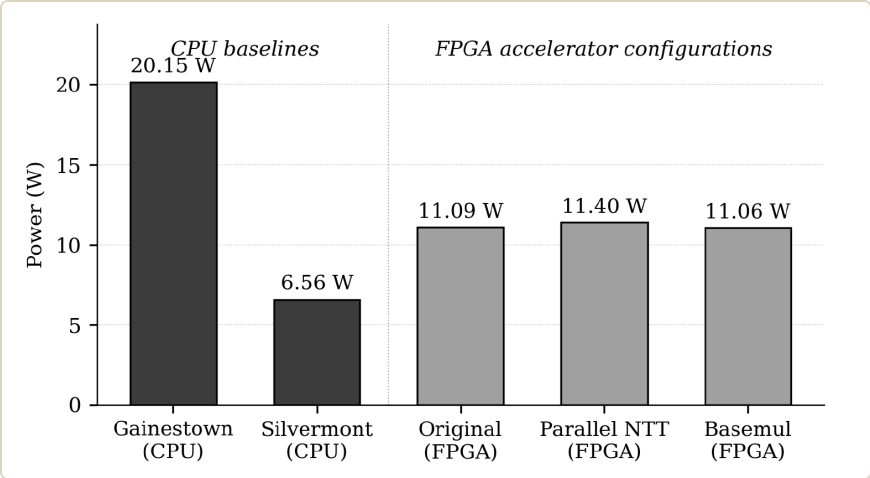
One asymmetry we had to state rather than hide: the HLS design exposes no separate KeyGen kernel. The expensive pieces of key generation, matrix expansion and centered-binomial sampling, happen inside Encapsulate, so its hardware cost is counted, but a KeyGen-to-KeyGen comparison does not exist. Totals still compare.

What the numbers said

Cycle counts flatter the FPGA, which runs at 300 MHz against 2.66 GHz, so the honest number is wall-clock time. On that measure the accelerator finishes 11 times sooner than the out-of-order core and 43 times sooner than the in-order one.

MEASURE	ML-KEM ACCELERATOR	GAINESTOWN, OUT-OF-ORDER	SILVERMONT, IN-ORDER
Cycles	**11,948**	1,195,000	1,724,000
Wall-clock time	**0.0398 ms**	0.448 ms	1.724 ms
Average power	**11.09 W**	20.15 W	6.56 W

One full ML-KEM-512 key-encapsulation pipeline, measured the same way on all three.



Average power across all five configurations. The accelerator draws about half of what the out-of-order CPU core does, and about 70% more than the in-order core.

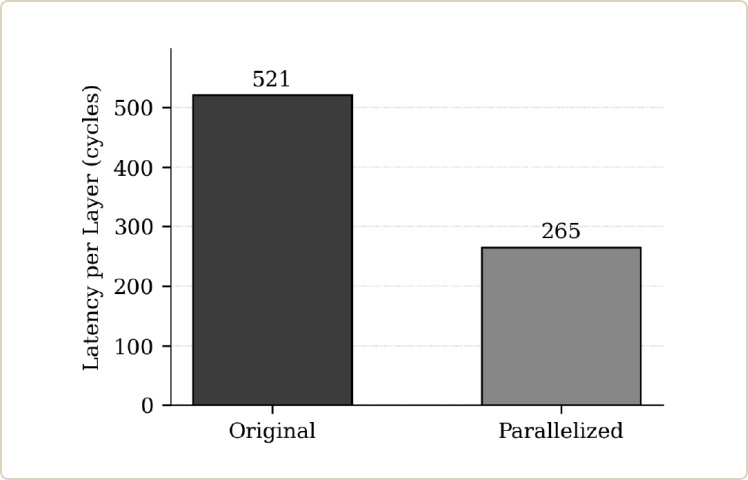
Beating a fast core on both time and power is the result that matters for the data-center case; losing to the low-power in-order core on watts while finishing 43 times sooner is a trade most operators would take. Thermal and environmental parameters came from Vivado's U280 data, with heat sink and airflow set to what a rack deployment would look like.

The resource report is what set up the rest of the project. The design fits in 13% of the U280's LUTs, 3% of its DSPs, and 1% of its block RAM. Most of the chip sits unused, which is an invitation to spend it.

Two optimizations, one halved and neither faster

The obvious lever was unrolling, and the reference design had already pulled it. The NTT and inverse NTT layers and their inner butterfly loops were unrolled about as far as they go. What was left was the module rank `K`, the

parameter that sets the security level. ML-KEM-512 uses K=2, so two polynomials run through the NTT sequentially. We duplicated the module inside `NTT_Layer` and called it once per polynomial, which cut per-layer latency from 521 cycles to 265, a 49% reduction. The same approach extends to K=3 and K=4 for the higher security levels, at the cost of the parameterized modularity we gave up.



Per-layer NTT latency before and after module duplication. The stage got twice as fast; the program did not.

Total cycles for the full pipeline went from 11,948 to 11,967. Halving a stage bought nothing at all, and the resource cost was real: 21% more BRAM, 23% more DSPs, 10% more LUTs, plus a 2.8% increase in power. Whatever the NTT was waiting on, it was still waiting on it.

So the NTT was not on the critical path. Finding what was took reading the design, because the synthesis report does not label which stages run concurrently in a dataflow region. Going through the code alongside the per-stage latencies and initiation intervals pointed at Readmem, Split, `basemul_montgomery`, and `basemul_add`. The two basemul modules handle coefficient-wise multiplication in the NTT domain across 128 to 256 coefficients. Widening their buffers to move four coefficients per pass instead of one should have raised throughput.

It did not move the cycle counts either: 529 to 528 on `basemul_montgomery`, and 515 to 515 on `basemul_add`. The resource numbers confirm the architecture genuinely changed, with flip-flop usage on `basemul_add` going from 69 to 232, so the hardware is different and the schedule is not. The likeliest explanation is data starvation. Widening two modules accomplishes little when most of the ports feeding them still carry one coefficient at a time, which means the buffers would have to be widened through the whole design rather than at the modules that looked slowest. Confirming that needs a co-simulation of the widened build, and that run exhausted the RAM on the machines we had. It stays an unproven hypothesis rather than a conclusion.

The other find was a bug in the reference design: the AXI master port bundles had no stream depths specified. We set them manually to get data flowing at all. Synthesis and implementation take about two hours per configuration, which put a hard ceiling on how many variants we could try, and experimenting with those stream depths never made the cut.

What I took away

A faster block is not a faster program. I knew that as a formula before this project, but watching a 49% latency reduction produce a 0.16% change in total cycles is a different kind of knowing. The interesting part was that Amdahl's law was the wrong explanation. The NTT is not a small fraction of the work; it was already fast enough

that something else in the dataflow architecture set the pace, and no amount of improvement there was going to show up at the top.

Most of the real skill turned out to be reading a design nobody on the team wrote. The tools report latency and resource usage per module and leave the causal structure to you, so locating a bottleneck meant cross-referencing initiation intervals against the code and forming a hypothesis that could be wrong. Ours was wrong twice.

A two-hour build loop changes how you plan, too. Batching configurations and deciding what a run would prove before starting it beats iterating by feel.

Neither optimization worked, and the paper says so. That was the most useful part to write.

CREDITS & SOURCES

- **Co-author:** Patrick Johnson (Tufts University). The design, benchmarking, and analysis were joint work throughout.
- **Accelerator source:** BSC LOCA, *PQC-Crystals-HLS-Accelerators*, GitHub repository, 2024. The accelerator is their open-source design and we did not write the bulk of that code. Our work was building and co-simulating it for our target, measuring it against CPU baselines, and attempting two optimizations on top of it.
- **CPU baseline source:** The PQClean Project, portable implementations of post-quantum cryptography.
- **Standard:** NIST, *Module-Lattice-Based Key-Encapsulation Mechanism Standard*, FIPS 203 (Final), August 2024.

Figures are taken from our own paper, linked at the top of this page.

06 / FPGA · VHDL

FPGA 2048

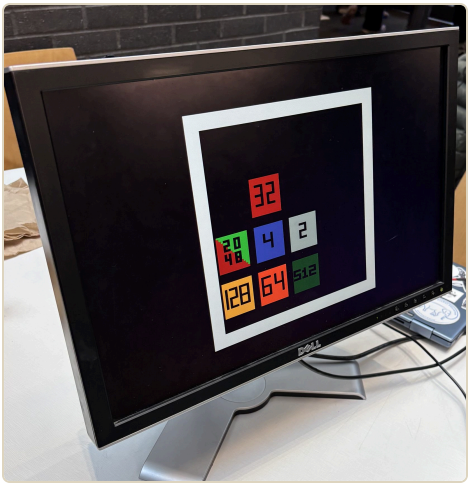
Course: Digital Logic · **Discipline:** FPGA design in VHDL · **Date:** 2024

GitHub: github.com/kalanbrunell/VHDL_Arcade_Game

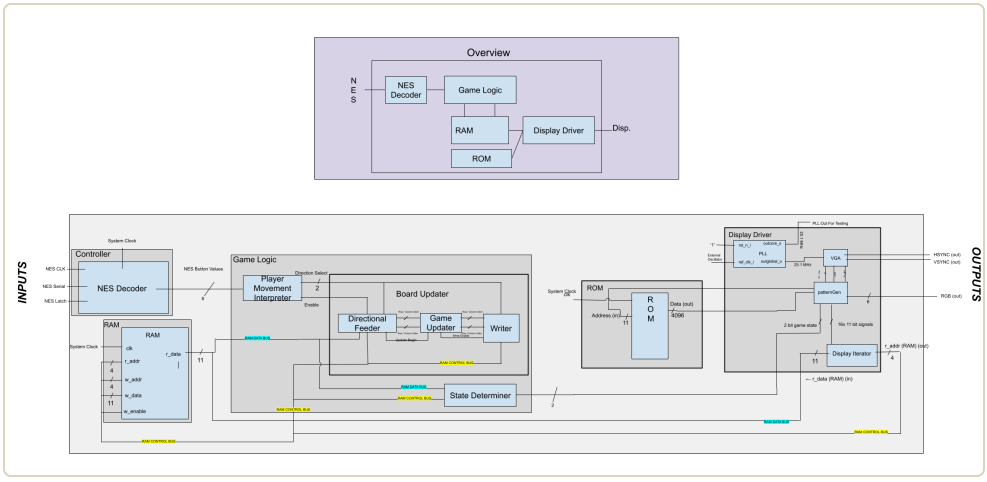
> The complete 2048 game implemented entirely in VHDL on an iCE40 FPGA, with a custom VGA controller, a tile-ROM graphics pipeline, and a classic NES-controller input.

Overview

For the final project of my Digital Logic course, I collaborated with a small group to implement the popular 2048 game entirely in VHDL on an iCE40 FPGA development board. The game outputs full VGA graphics and supports user input via a classic NES controller, enabling a complete gaming experience on custom hardware.



The finished game on a VGA monitor, driven by the iCE40 board.



System block diagram: VGA controller, game logic, and input processing modules.

Game logic & graphics pipeline

At the core of the project was the game logic, which required careful state management to handle tile movement, merging, and score updates. On top of that, I developed a custom graphics pipeline to render the board and tiles

on a VGA display under strict timing requirements for synchronization signals. Each tile was stored in its own ROM block for efficient fetch and rendering during gameplay. The graphics system also required tightly timed pixel-clock generation, horizontal-sync, and vertical-sync signals to maintain a stable display.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity vga is
    port(
        clk : in std_logic;
        HSYNC : out std_logic;
        VSYNC : out std_logic;
        rowOut : out unsigned(9 downto 0);
        colOut : out unsigned(9 downto 0);
        valid : out std_logic
    );
end;
```

The VGA controller owns the sync signals and hands the rest of the design a pixel coordinate, plus a valid flag saying when that coordinate is inside the drawable region.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity twoBlockROM is
    port(
        clk : in std_logic;
        rowIn : in unsigned(9 downto 0);
        colIn : in unsigned(9 downto 0);
        rowOffset : in unsigned (9 downto 0);
        colOffset : in unsigned (9 downto 0);
        rgbOut : out std_logic_vector(5 downto 0)
    );
end;
```

A tile ROM addressed by that pixel coordinate plus an offset, so one stored tile can be drawn anywhere on the board. It answers with a six-bit RGB word.

07 / JUNIOR DESIGN · EMBEDDED SYSTEMS

Autonomous Color-Following Vehicle

Course: Junior Design Project · **Institution:** Tufts University · **Dates:** September 2025 — December 2025

> An autonomous vehicle built end to end by a small project team: an ATmega-based platform that reads a custom photodiode array through a PID control loop to follow a colored line, with a WebSocket link for telemetry and coordination with the other vehicles on the course. I owned that link, and a large share of the sensor array and the physical build.

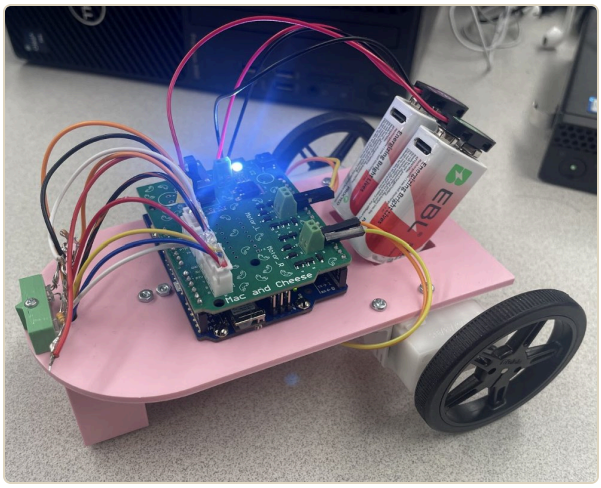
Overview

A small team building a vehicle that follows a colored line around a course on its own, while sharing that course with the other teams' vehicles at the same time. The work split four ways: a custom Arduino shield for power and motor drive, embedded C++ on an ATmega, a discrete photodiode array for sensing, and a WebSocket link carrying telemetry off the vehicle.

The WebSocket link was mine end to end. I did a large share of the photodiode array and of the physical assembly of the vehicle, and I supported the shield design and the PID work rather than leading them. The vehicle followed the course and the project was delivered.

Custom motor & power shield

The vehicle's electronics live on a custom shield stacked on the Arduino, nicknamed "Mac and Cheese" on its silkscreen. Each of the two drive motors gets its own discrete H-bridge, built from individual transistors rather than an off-the-shelf driver chip, so each wheel can run forward and reverse independently. A driver IC would have been fewer parts and less to get wrong. We built it from transistors on purpose, to understand the switching topology instead of calling into a black box. The same board steps a pair of 9V batteries down to 3.3V to power the Arduino, and dedicated screw-terminal blocks connect the photodiode array and its LEDs instead of loose wire splices.



The finished vehicle, with the custom shield stacked on the Arduino.

Embedded detection system

Line detection runs through a custom photodiode array read by the ATmega's ADCs, with no camera or external vision hardware involved. I did a large share of that array, and of the physical build of the vehicle around it.

On top of the raw readings sits a PID control loop, which treats the vehicle's position relative to the line as an error term and continuously corrects the motor outputs to stay centered, rather than reacting to the sensors with fixed on/off logic. I supported that work rather than owning it.

WebSocket communication

The link is the piece I owned. The course requirement was that two vehicles run the course at the same time, on paths that crossed at multiple points. Every team built their robot differently, so neither vehicle could assume anything about how the other one moved or how quickly it got anywhere. That rules out hard-coding the yielding: there is no fixed timing to encode against. Instead the two vehicles shared status over a WebSocket link and cooperated at each crossing.

The same link carried telemetry and debug messages back to a laptop, which is what sped development up. You can watch what the vehicle is deciding while it drives, instead of reconstructing it afterward from a robot that has already left the line.

08 / FRC · VISION & CONTROLS

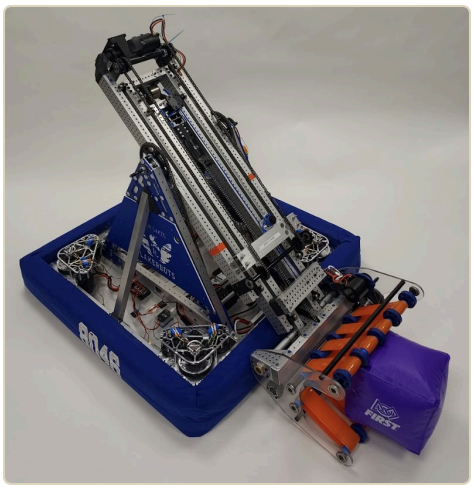
Competitive Robotics Vision & Control

Team: Lakerbots 8046 · **Competition:** FIRST Robotics Competition · **Role:** Team Captain & Lead Engineer · **Seasons:** 2022 — 2023

> Three interlocking systems across two FRC seasons: a real-time vision pipeline for autonomous targeting, a field-centric swerve-drive control stack, and an IMU-based balancing routine for the 2023 Charge Station.

FRC context

The FIRST Robotics Competition challenges high-school teams to design, build, and program 120-pound robots in six weeks to compete in alliance-based matches. Teams engineer solutions for complex game objectives within strict size, weight, and safety constraints.

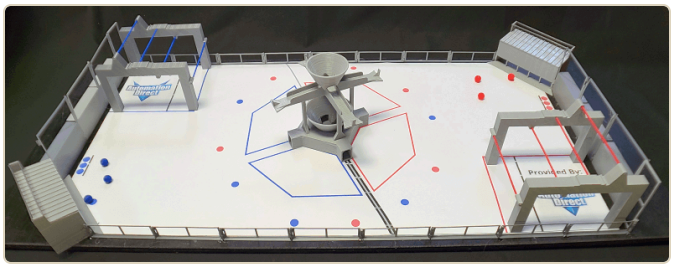


Our 2023 Charged-Up competition robot.

Autonomous vision system

Challenge

The 2022 season required collecting and scoring inflated game pieces through an elevated hoop. Effective play meant autonomous scoring during the first 15 seconds of each match and precision manual control thereafter, and both depend on accurate target acquisition. The field included retroreflective tape around scoring targets to support vision-based solutions.

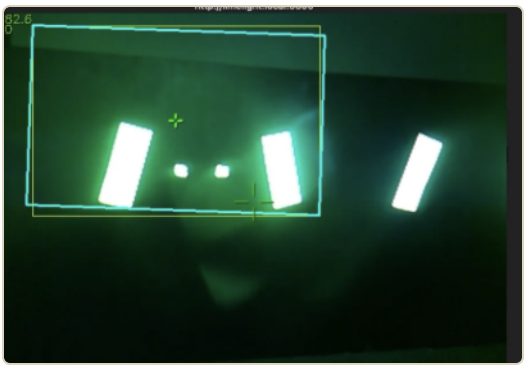


The 2022 Rapid React field: the central scoring target.

Technical approach

I led the development of a real-time vision system using a Limelight vision processor to detect the retroreflective tape on scoring targets. We tuned the camera's mounting angle for maximum target visibility and programmed the Limelight to return 2D pixel coordinates for detected targets.

From that, we built a custom algorithm that used the target's vertical dimension to estimate distance and its horizontal dimension to compute the rotation needed to align. The output drove our swerve-drive autonomous control, letting the robot align itself and adjust launching parameters for efficient scoring.



Limelight interface: real-time target detection on retroreflective tape.

Results

This system enabled consistent autonomous scoring from up to 10 feet away.

Swerve-drive system

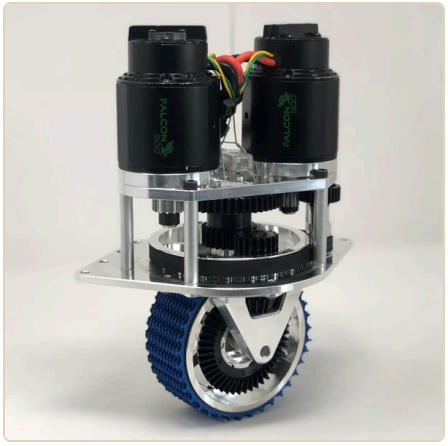
Challenge

Before 2022 our tank-drive system limited maneuverability and prevented simultaneous translation and rotation, which hurt game-piece collection efficiency. I spearheaded the integration of a swerve drive, enabling omnidirectional movement and much higher agility on the field.

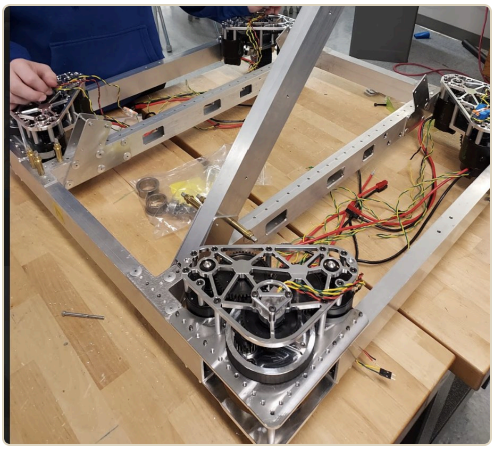
Technical approach

Given our team's limited prior experience with swerve drives, we purchased a COTS module system to shorten the path to a working prototype. We then deployed a FIRST-standard swerve-kinematics library and tuned the control algorithms to match our robot's dynamics.

We implemented field-centric control using a gyroscope for orientation feedback, so each joystick direction always mapped to the field's frame relative to the driver, regardless of the robot's heading. We then tuned the PID loops on each wheel module for precise speed and angle adjustments.



MK4 swerve module: independent steering and drive motors.



Our prototype swerve frame during development.

Results

The swerve drive let the robot translate and rotate at the same time, so repositioning no longer cost a turn. That season produced our highest regional placement at the time and a district-championship qualification.

Dynamic platform balancing

Challenge

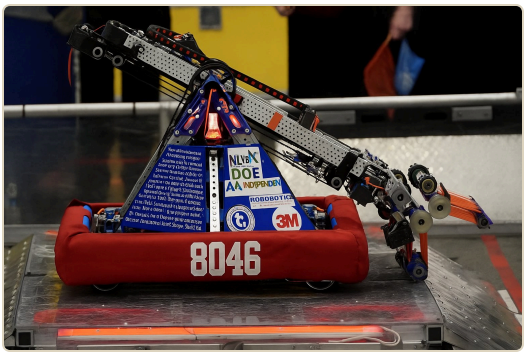
The 2023 endgame asked robots to balance on a dynamic "Charge Station" platform for bonus points, but unpredictable motion caused most robots to slide off. I built an IMU-based control system for stable equilibrium while compensating for disturbances from other robots.

Technical approach

The IMU was already integrated on the robot for field-centric driving. I co-developed a PID-based balancing algorithm that adjusted wheel speeds in real-time to maintain a level orientation, allowing us to program a fully autonomous balancing routine for the endgame. Once balanced, two of the four wheels would rotate 90° to mechanically lock the robot in place, preventing roll-off from external forces.

Results

We achieved an estimated 95% success rate on autonomous balancing attempts during competition matches, consistent enough that alliance partners could count on the endgame points.



Autonomous balancing on the Charge Station, 2023 Charged Up.

IMAGE CREDITS

- **Rapid React field layout:** AutomationDirect — "2022 FIRST Robotics Rapid React 3D Printed Field." AutomationDirect Library, accessed Nov 6, 2025.
- **MK4 swerve module:** Swerve Drive Specialties — "MK4 Swerve Module." Accessed Nov 6, 2025.
- **Limelight interface:** Limelight Vision — "Getting Started with Pipelines." Limelight Documentation, accessed Nov 6, 2025.

Other images courtesy of Lakerbots 8046 FRC Team. Field layout, swerve module, and Limelight documentation images used for educational demonstration purposes.

09 / EMBEDDED · AUTOMOTIVE

Car RPM Shift Indicator

Display: 16× SK9822 LED bar · **MCU:** STM32 (Nucleo-32) · **Data:** OBD-II · **Date:** 2026 — Present · In Progress

> A dedicated shift indicator: a 16-LED bar that sweeps green to yellow to red with engine RPM and flashes when it's time to shift. It reads live engine data from the car's OBD-II diagnostic port, works on a breadboard today, and is headed into a 3D-printed enclosure as a perfboard build.

> **The idea:** a shift cue you can read without looking at it, driven by data the car already has, added without modifying the car at all. The car sees a standard diagnostic tool; everything interesting happens on my side of the port.

The problem

A tachometer is a bad shift cue. It is a small dial low in the instrument cluster, and reading it means taking your eyes off the road, finding the needle, and interpreting a number, all in the moment before the shift. What you actually want near redline is a signal that works in peripheral vision: position and color, not a gauge.

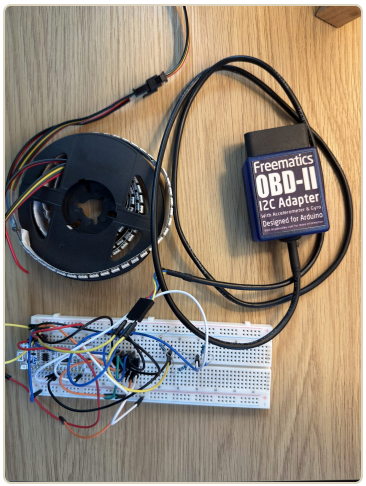
The harder problem is that a modern car is a closed box. The data to drive that signal already exists, the ECU tracks RPM, speed, and coolant temperature continuously, but there is no sanctioned way to add your own electronics to the dash. The one standardized opening is the OBD-II diagnostic port. This project lives entirely on that port: it asks the same questions a scan tool would, writes nothing, and unplugs without a trace.

What's working

The display is a 16-LED bar cut from an SK9822 addressable strip. I picked the SK9822 over the more common WS2812B for three reasons. It is driven over plain SPI with separate data and clock lines, so the microcontroller never has to generate a timing-critical one-wire waveform. Its PWM refresh sits near 5 kHz, where the WS2812B's much slower refresh shimmers visibly in peripheral vision, exactly where a shift light has to live. And it carries a separate 5-bit per-LED current setting, so the bar can dim for night driving without giving up color resolution. A 74AHCT125 buffer shifts the 3.3 V SPI lines up to the strip's 5 V logic.

The controller is an STM32 Nucleo-32 board with firmware built in PlatformIO. RPM comes from an off-the-shelf OBD-II adapter that speaks the diagnostic protocol on its own microcontroller and hands over the latest values, with age stamps, over I2C. The firmware polls it and maps RPM onto the bar as a green-to-yellow-to-red sweep, with a full-bar flash at the shift point.

The data source is an interface in the firmware, not a fixed device. On the bench, a simulated source plays back a synthetic drive loop, so the sweep colors, shift flash, and display logic can all be developed and tested at a desk. The OBD-II source swaps in for the car.



The bench: the Nucleo-32 and its wiring on the breadboard, the Freematics OBD-II I2C adapter that supplies RPM, and the SK9822 strip still on its reel.

Version 1: off the breadboard

Current work is turning the breadboard into a version that can live in a car: the same circuit moved to perfboard, inside a 3D-printed enclosure sized for the top of the steering column. Two open items gate the install. The adapter's 5 V rail is rated for 2 A, which covers the strip's worst case on paper, but the bar stays brightness-capped until that holds up under a real load test. And the diagnostic port is powered even when the car is off, so the firmware has to drop into a low-power state whenever the engine isn't running, or the project becomes a slow battery drain.

Designed for version 2

Version 2 is already specced, and it replaces the off-the-shelf adapter with my own hardware: a custom PCB around an ESP32-S3, chosen because one chip brings a CAN controller, Bluetooth LE, and WiFi. Instead of requesting values through the diagnostic protocol, the board listens directly to the car's CAN bus, receive-only by design, and gets everything the modules on the bus already broadcast.

That opens up more than a faster shift light. The same data can drive indicators the dash doesn't offer: a low-battery-voltage alert, a warm-up mode that holds the shift point down until the coolant is warm, and gear detection computed from the ratio of RPM to speed. A phone app over Bluetooth LE handles configuration and live gauges, and drives get logged to onboard flash, with summaries syncing to my home server when the car is on home WiFi.

10 / INFRASTRUCTURE · SELF-HOSTING

Homelab & Self-Hosted Infrastructure

Server: special-k (HP EliteDesk Mini) · **OS:** Ubuntu Server · **Status:** Always on, ongoing

> A single small mini PC at home, running my personal services in Docker and reachable from anywhere over a private network. It holds my own photos and documents, replaces a few subscriptions, and doubles as the place I practice building and deploying real software with AI.

The idea: own the data and understand the whole stack. Instead of renting cloud storage and stitching subscriptions together, one machine at home runs everything, and none of it is exposed to the public internet. It is equal parts a set of tools I actually use and a sandbox for learning how software gets built and kept running once it ships.

The machine

The whole lab runs on one HP EliteDesk Mini, a small-form-factor desktop picked for low idle power and a quiet fan rather than raw speed. It runs Ubuntu Server with no desktop environment, so everything is managed over SSH from the terminal.

Each service lives in its own Docker container, described in Compose files that can be torn down and brought back exactly as before. Containers keep the services isolated: one app crashing or hogging memory cannot take the others down with it, and adding something new is a matter of another config block rather than another machine.

What's running

Roughly a dozen containers run side by side. Some are well-known open-source projects; several are web apps I wrote and host here myself.

PHOTOS, DOCS & FILES

Immich hosts my photo and video library in place of a cloud subscription, Paperless-ngx keeps scanned documents searchable, and Syncthing mirrors files across my laptop, desktop, and the server.

APPS I BUILT

A personal dashboard that pulls together my calendar, email, and coursework; a grocery and recipe tracker; and a tracker for job applications. Each is a small web app with its own database, running as its own container.

MONITORING

Uptime Kuma watches each service for outages, Beszel graphs host and container health, and a status page I wrote pulls both into one view so I can check the whole lab at a glance.

special-k, the HP EliteDesk Mini that runs the whole homelab, mounted in a 10-inch rack under the patch panel and switch.

Access and backups

Access runs entirely over Tailscale, a private mesh network that connects my own devices directly. Reaching a service from a laptop or phone works from anywhere, but only from a device I have added to the network, which is what keeps the lab off the public internet in the first place. A Caddy reverse proxy sits in front of the containers and gives each app a clean internal address with automatic HTTPS.

Backups run with restic to Backblaze B2, encrypted on my side before anything leaves the house, so the storage provider only ever holds ciphertext. I compared restic against Kopia before settling on it, and the backup set covers the data that would actually hurt to lose: photos, documents, and each app's database.

Built and run with AI

Most of the apps above started as a written spec, not as code. The workflow is to describe what I want in a detailed handoff document, hand that to Claude Code, then read, test, and correct what it produces. The homelab is where I practice that loop on software I rely on every day, which raises the stakes well past a throwaway demo.

That has meant learning where the agent is strong and where it needs a short leash. It moves fast, but a vague spec sends it straight into plausible code that quietly does the wrong thing, so the handoff has to pin down interfaces and failure behavior up front. Reviewing the result matters as much as writing the prompt: one deploy handed back a container that passed every health check and still served a blank page, because the generated code was rejecting the app's own requests. curl could not see the bug; a real browser could. Catching that kind of gap, and knowing to look for it, is the skill I am building here.

What I'm learning

Running this many services on one box teaches things a class project never does. Ports collide, a reverse proxy has to route traffic to the right place, a container that starts is not the same as a service that works, and a backup only counts once a restore has been tested. Most of what I know about operating software, rather than just writing it, came from breaking and fixing this setup.

Self-hosting also means the data is mine to protect. Keeping my own photos and documents instead of renting them puts the whole chain on me: encryption, off-site copies, and a plan for the day a disk dies. Getting that right is more satisfying, and more instructive, than handing it to a provider.

Underneath all of it, the lab has become steady practice with the everyday tools of the trade: Linux and the command line, Docker, networking, and Git.