Hauke Aakmann-Visher

Mechatronics Engineer — Robotics & Automation

Profile

Mechatronics Engineering graduate (Stellenbosch, Washington Accord) with German–Namibian citizenship and full EU work rights. Experienced in embedded control, automation, and mechanical design. Designed and implemented an autonomous warehouse robot guidance system integrating IoT communication, A* path planning, and real-time sensing. Proficient in C, Python, MATLAB, and Simulink, with hands-on experience in STM32, Raspberry Pi, and PLCs. Strong leadership, documentation, and analytical skills; native German and fluent English.

Contact

Otjiwarongo, Namibia
+264 81 392 9876
hauke.visher@outlook.com
LinkedIn:
hauke-aakmann-visher-064064342
GitHub:
github.com/HaukeAakmannVisher/Portfolio

Contents

Acronyms

PWM Pulse-width modulation (motor speed control)

ADC Analog-to-digital converter

UI User interface

UART Serial interface (microcontroller comms)

MPP Maximum Power Point

SP Solar-panel power measurement routine (duty sweep)

EN Environmental measurement routine (light/temperature)

TRSensors
Line sensors used for tracking/intersections

Autonomous Warehouse Robot Guidance System

2024

Role & context

- **Sole designer & implementer:** planning, A* planner, Flask web UI, Pi firmware, sensor integration, and testing.
- **Course context:** Final-year Mechatronic project (Project 478).
- **Platform:** Raspberry Pi on AlphaBot2; heavily adapted vendor examples to support a web-controlled planner/executor architecture.

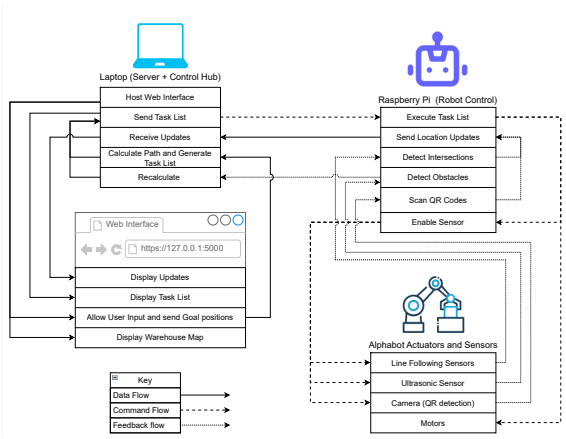
Summary: A laptop computes A* routes on a discrete warehouse grid, serves a browser UI, and converts routes into robot actions (*left/right/straight/stop*). Actions are sent via HTTP/JSON to a Raspberry Pi (AlphaBot2), which performs PWM motor control, line following (TRSensors), intersection detection, and obstacle monitoring. The Pi streams live updates back to the UI.

- **End-to-end architecture:** Web UI (operator) → laptop planner (A*, task list) → Pi executor (motion + sensing).
- **Planning & actions:** Grid paths are translated into intersection actions for three legs (Home→G1, G1→G2, G2→Home).
- **Control & sensing:** PWM differential drive; TRSensors for line/intersections; ultrasonic for obstacle stop → notify workflow.
- **Calibration:** Left motor (linear voltage fit); right motor (4th-order fit) with a small start-up torque boost to reduce veer.

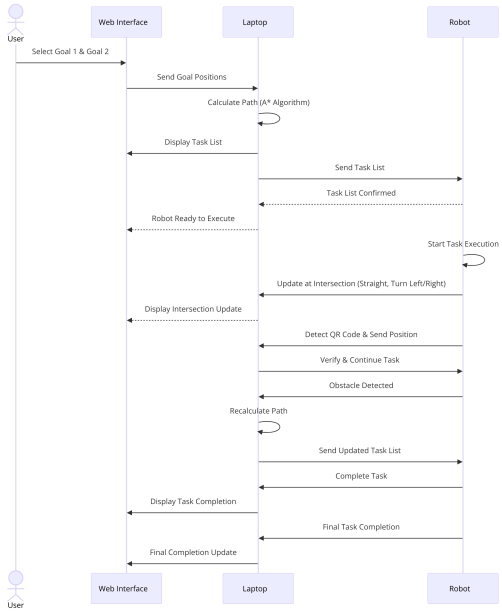
Repository & report links

- Repo root (project folder): https://github.com/HaukeAakmannVisher/Portfolio/tree/main/Warehouse%20Robot%20Guidance%20System
- Full report — *direct download (raw)*: https://raw.githubusercontent.com/HaukeAakmannVisher/Portfolio/main/Warehouse%20Robot%20Guidance%20System/warehouse-robot-guidance-system-full-report.pdf
- Laptop code (planner & web UI): https://github.com/HaukeAakmannVisher/Portfolio/tree/main/Warehouse%20Robot%20Guidance%20System/laptop
- Pi code (task execution): https://github.com/HaukeAakmannVisher/Portfolio/tree/main/Warehouse%20Robot%20Guidance%20System/pi
- PWM–voltage compensation (`AlphaBot2.py`): https://github.com/HaukeAakmannVisher/Portfolio/blob/main/Warehouse%20Robot%20Guidance%20System/pi/AlphaBot2.py

Architecture & Information Flow



Report Fig. 4.1 — System Architecture Overview



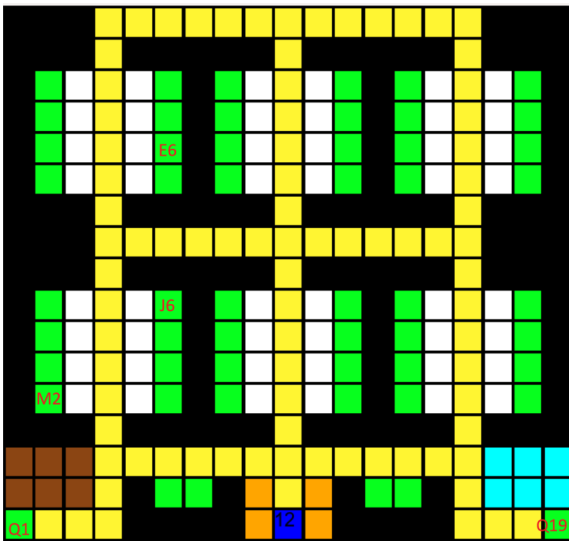
Report Fig. 4.7 — Task and Information Flow

Figure 4.1 positions the three actors: the web interface (operator), the laptop (A* + task derivation), and the Pi (execution + sensing). Figure 4.7 shows the rhythm: goal selection → path calculation → task list display → execution updates, with clear operator feedback at each intersection or exceptional event.

Warehouse Model & A* Path Planning

```
# Define the grid with obstacles, highways, and possible goal cells
grid = [
    [1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1],  # A
    [1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1],  # B
    [1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1],  # C
    [1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1],  # D
    [1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1],  # E
    [1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1],  # F
    [1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1],  # G
    [1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1, 1],  # H
    [1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1],  # I
    [1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1],  # J
    [1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1],  # K
    [1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1],  # L
    [1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1, 3, 0, 2, 0, 3, 1],  # M
    [1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 1, 1, 1],  # N
    [4, 4, 4, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 6, 6, 6],  # O
    [4, 4, 4, 2, 1, 3, 3, 1, 5, 2, 5, 1, 3, 3, 1, 2, 6, 6, 6],  # P
    [3, 2, 2, 2, 1, 1, 1, 1, 5, 3, 5, 1, 1, 1, 1, 2, 2, 2, 3]   # Q
]#   1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
```

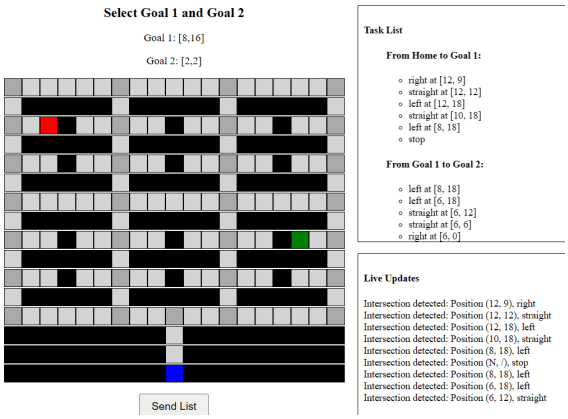
Report Fig. 3.2 — Grid with obstacles, highways, possible goals



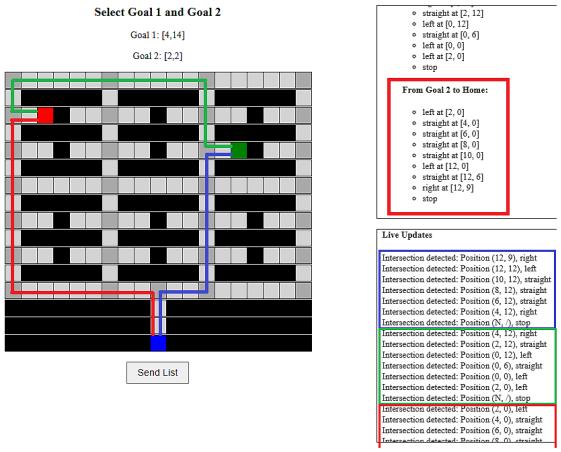
Report Fig. 3.3 — Warehouse model visualised in Pygame

The planner works on a discrete grid with blocked cells, fast "highway" aisles, and eligible goal cells. A* uses cost heuristics to select the shortest feasible route. The right-hand visual confirms the grid model matches the topology used by the UI.

Operator UI, Task Lists & Execution Feedback



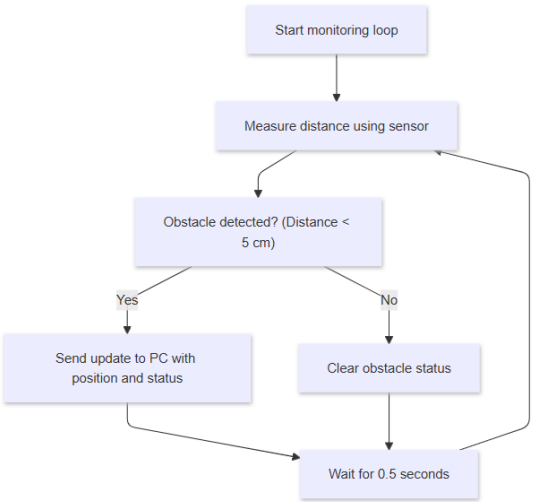
Report Fig. 4.4 — Goal selection, path, and task list



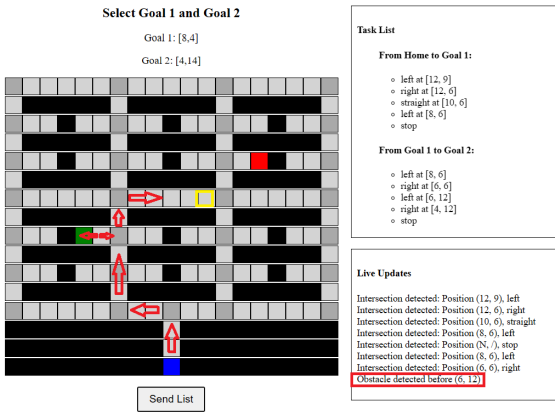
Report Fig. 5.2 — Live intersection updates during tests

After selecting Goal 1 and Goal 2, the UI displays the computed path and the derived per-leg task lists (intersection actions). During execution, each intersection triggers a log entry; the operator can verify progress and intervene if needed.

Obstacle Handling



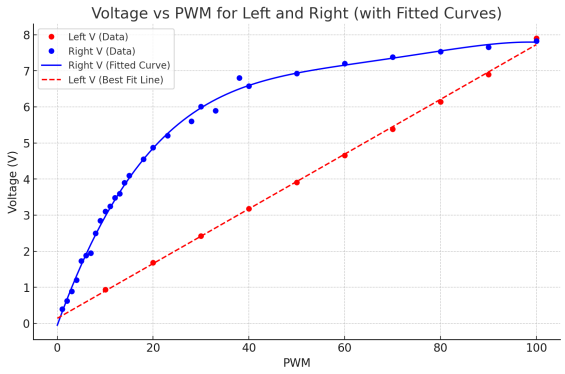
Report Fig. 4.5 — Obstacle detection logic flow



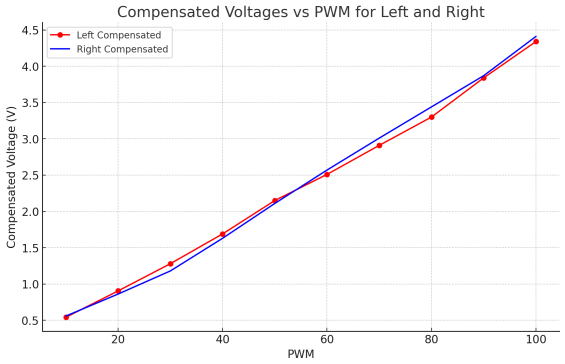
Report Fig. 5.3 — Obstacle flagged before next intersection

A simple ultrasonic check pauses motion if an obstacle is detected shortly before the next intersection. The Pi emits a structured message (*"Obstacle detected before [row, col]"*); the laptop can mark the cell as blocked and recompute an alternative route.

Calibration & Motor Compensation



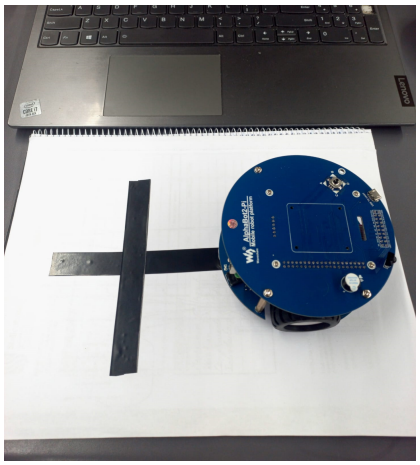
Report Fig. 4.10 — Voltage vs PWM with fitted curves



Report Fig. 4.12 — Compensated voltages (left vs right)

Bench tests showed the left motor followed a near-linear PWM–voltage curve, while the right motor saturated non-linearly due to a recurring driver fault. I compensated in software by solving for a right-motor PWM that matches the left motor's target voltage and adding a brief start-up torque boost. The left plot shows the raw curves (linear left vs. saturating right); the right plot shows the post-compensation result. This mitigated the hardware issue but wasn't perfectly reliable, so I validated the system with controlled tests in the next section. See `AlphaBot2.py` for the compensation functions.

Test Rig



Report Fig. 5.1 — Simulated intersection rig for repeatable tests

The manual rig let me validate the end-to-end software chain (planner, UI, logs, intersection triggers) while the physical motor driver fault was being mitigated.

Outcomes (evidence)

- Verified end-to-end chain (planner → UI → robot) on a simulated intersection rig; correct actions were raised at the right cells and echoed live to the dashboard.
- Obstacle workflow validated: obstacle detected, position logged, grid cell blocked, shortest path recomputed, and a new task list executed.

- Hardware limitation documented (motor-driver fault); software voltage/torque compensation mitigated veer and enabled reliable intersection/obstacle tests.

Notes & disclaimers

Running the Pi code requires the **AlphaBot2 platform** (motors, driver, TRSensors, ultrasonic) and a Raspberry Pi with GPIO access. The repository includes the laptop planner/UI and Pi execution code used for the project; hardware-specific wiring and environment setup are assumed.

PV System Efficiency Monitor

2024

Role & context

- **Sole designer & implementer:** hardware choices (protection, dividers, shunt/amp, rails, UI) and STM32 firmware (calibration, EN, SP).
- **Course context:** Individual E-Design capstone (2024).
- **Intent:** low-cost proxy next to a large array to flag cleaning needs via measurable output degradation under comparable conditions.

Summary: A portable instrument that characterises a small PV panel by sweeping a programmable load, sampling panel voltage/current, computing $P{=}VI$, and reporting peak power and comparative metrics on an LCD; the hardware integrates a PV front-end (protection and scaling), a shunt-based current path, temperature sensing, regulated rails, and a simple UI, while the firmware orchestrates calibration, environmental reads, a duty sweep (*SP* routine), and result display; designed to run in parallel with a utility-scale PV array as a low-cost proxy for panel health, the system tracks output degradation under comparable irradiance/temperature to signal when soiling (dust/debris) has reduced efficiency enough to warrant cleaning.

Repository & report links

- Repo folder (project root):
  https://github.com/HaukeAakmannVisher/Portfolio/tree/main/PV%20Efficiency%20Monitor
- Full report — *view on GitHub*:
  pv-efficiency-monitor-full-report.pdf
- Full report — *direct download (raw)*:
  https://raw.githubusercontent.com/HaukeAakmannVisher/Portfolio/main/PV%20Efficiency%20Monitor/pv-efficiency-monitor-full-report.pdf
- STM32 code folder:
  https://github.com/HaukeAakmannVisher/Portfolio/tree/main/PV%20Efficiency%20Monitor/code

System Overview (Hardware & Firmware)

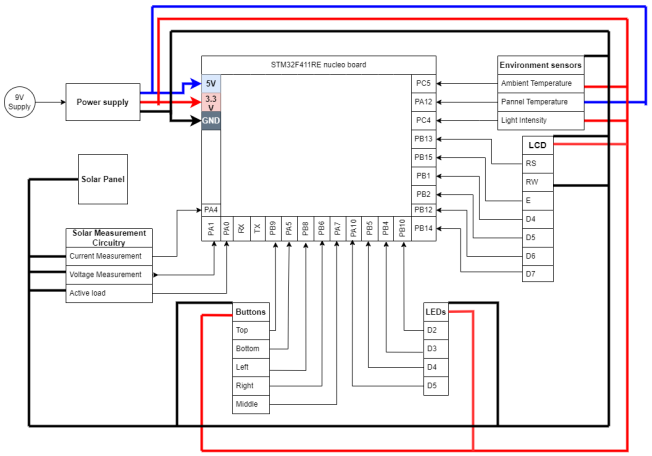


Figure 1. Whole-board assembly.

The board integrates the PV input, measurement front-end, microcontroller, regulators, and LCD UI. This compact arrangement keeps measurement wiring short and stable for repeatable runs.

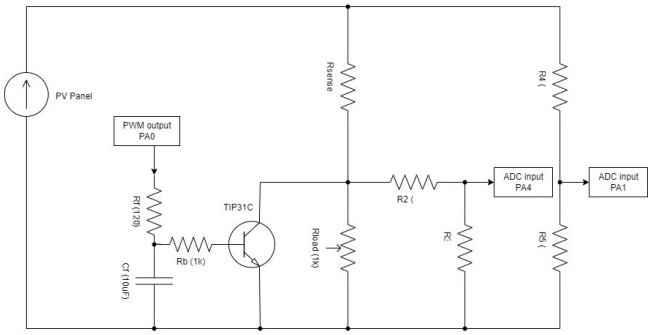


Figure 9. PV panel circuitry (front-end).

Input protection and scaling feed V_{PV} to an ADC channel, while a current shunt path conditions I_{PV}. These two measurements drive the power curve used to identify peak operating points under test conditions.

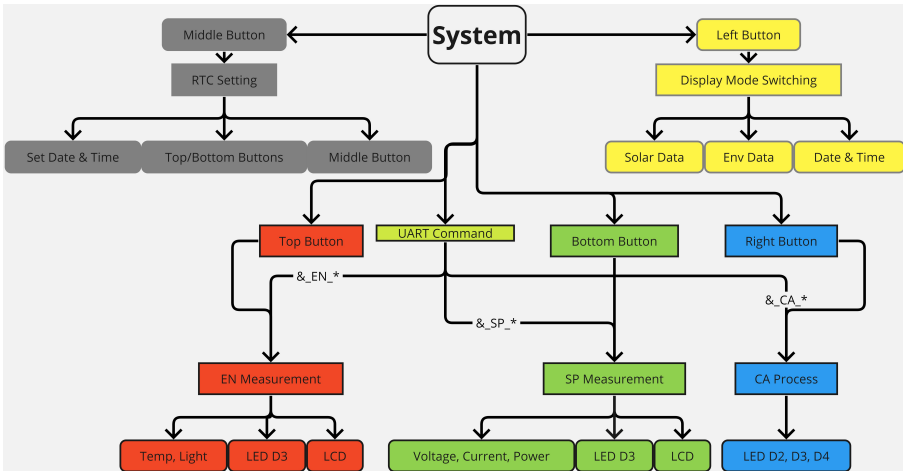


Figure 12. High-level system diagram.

Figure 12 shows sensing (PV, temperature), actuation (active load), computation (MCU), and pre-

sentation (LCD/buttons). The firmware cycles through init → (optional) calibration → SP sweep → compute/report.

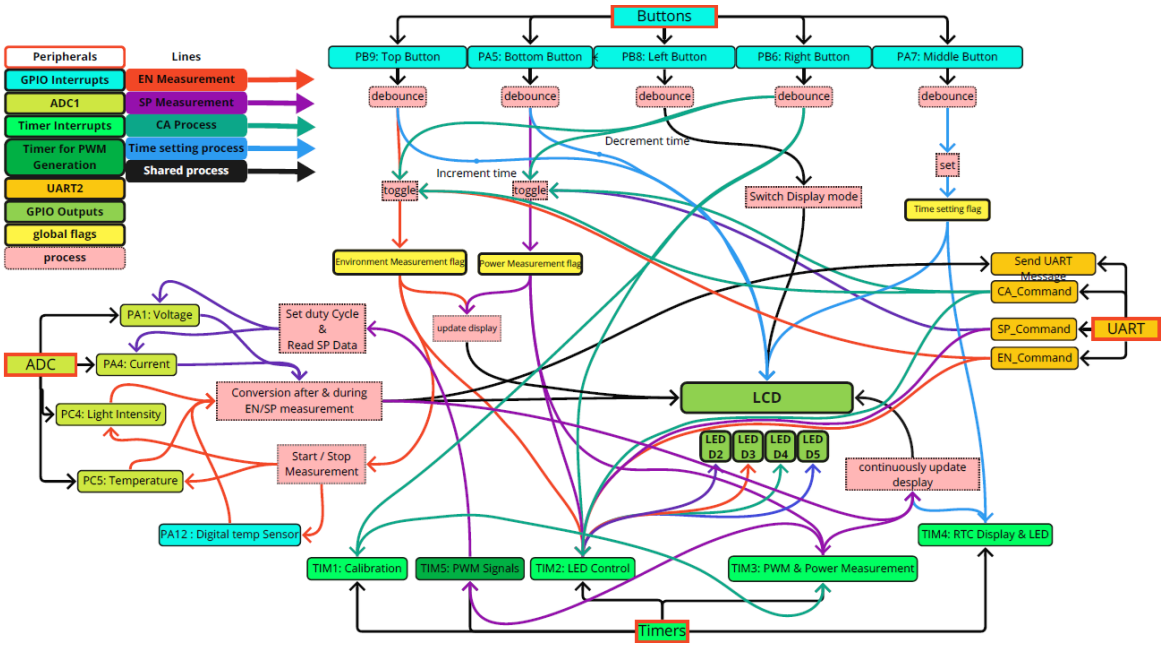


Figure 13. Software interactions and tasking.

Tasks are split into small, predictable services: button handling, measurement scheduling, V/I acquisition, power calculation, and UI updates. This keeps timing and code paths simple on the STM32.

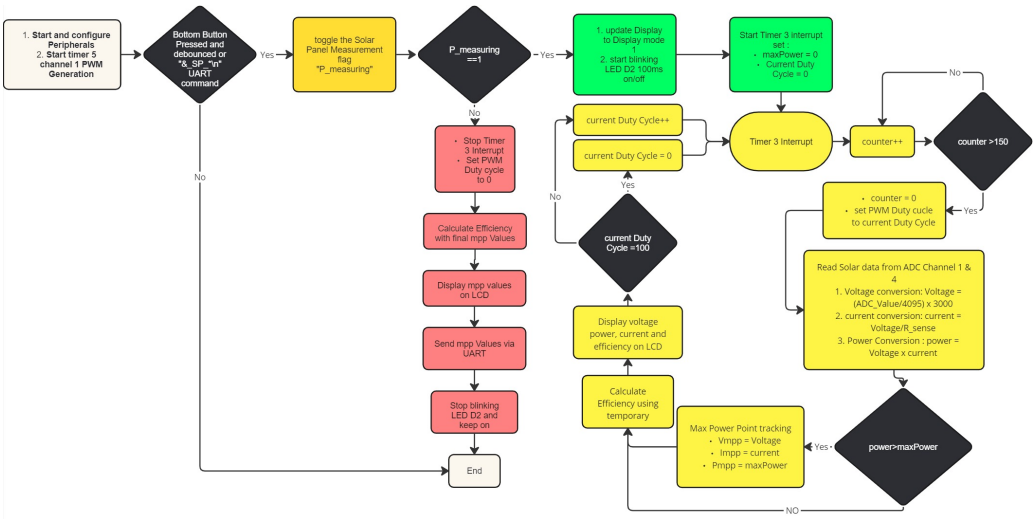


Figure 15. SP routine (duty sweep → V/I sampling → $P{=}VI$).

The SP routine steps the programmable load duty while logging V/I. The resulting curve identifies peak power and allows repeatable panel comparisons across conditions.

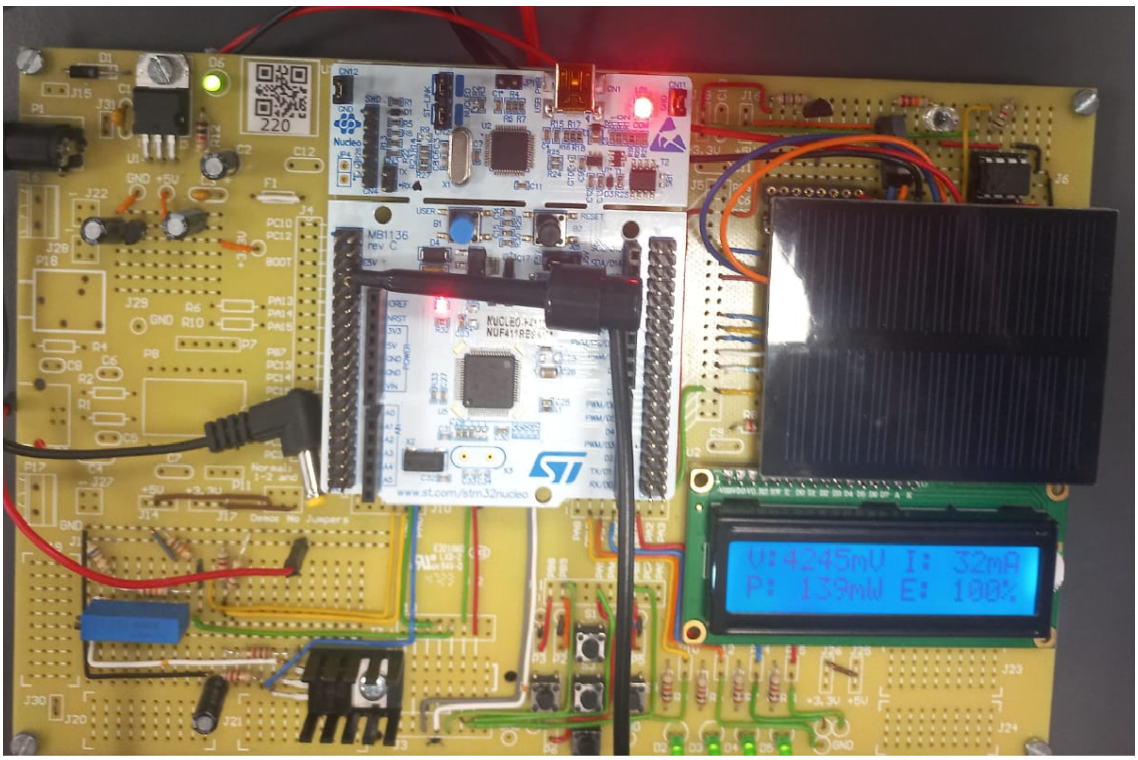


Figure 32. Representative power measurement.

A sample run confirming end-to-end behaviour: the measured $P{=}VI$ trajectory and peak occur where expected given the panel and ambient conditions.

Outcomes (evidence)

- Implemented SP sweep with time-aligned V/I sampling and LCD reporting of MPP/peak power.
- Captured a representative "after-calibration" power curve (portfolio Fig. 32) as end-to-end proof of measurement and computation.
- Disclosed calibration notes and sources of error (supply stability, sensor/ADC tolerances) to frame repeatability and next steps.

Notes & disclaimers

The code folder contains the most relevant STM32 sources (e.g., `main.c`, `lcd16x2.c/h`, interrupt and system stubs). It is not a complete CubeIDE project; paths and generated files are omitted to keep the repository compact.

Final Notes

This portfolio focuses on the two most relevant build–measure projects. For context and full technical details, please use the repository links to open the complete PDFs and browse the corresponding source folders.