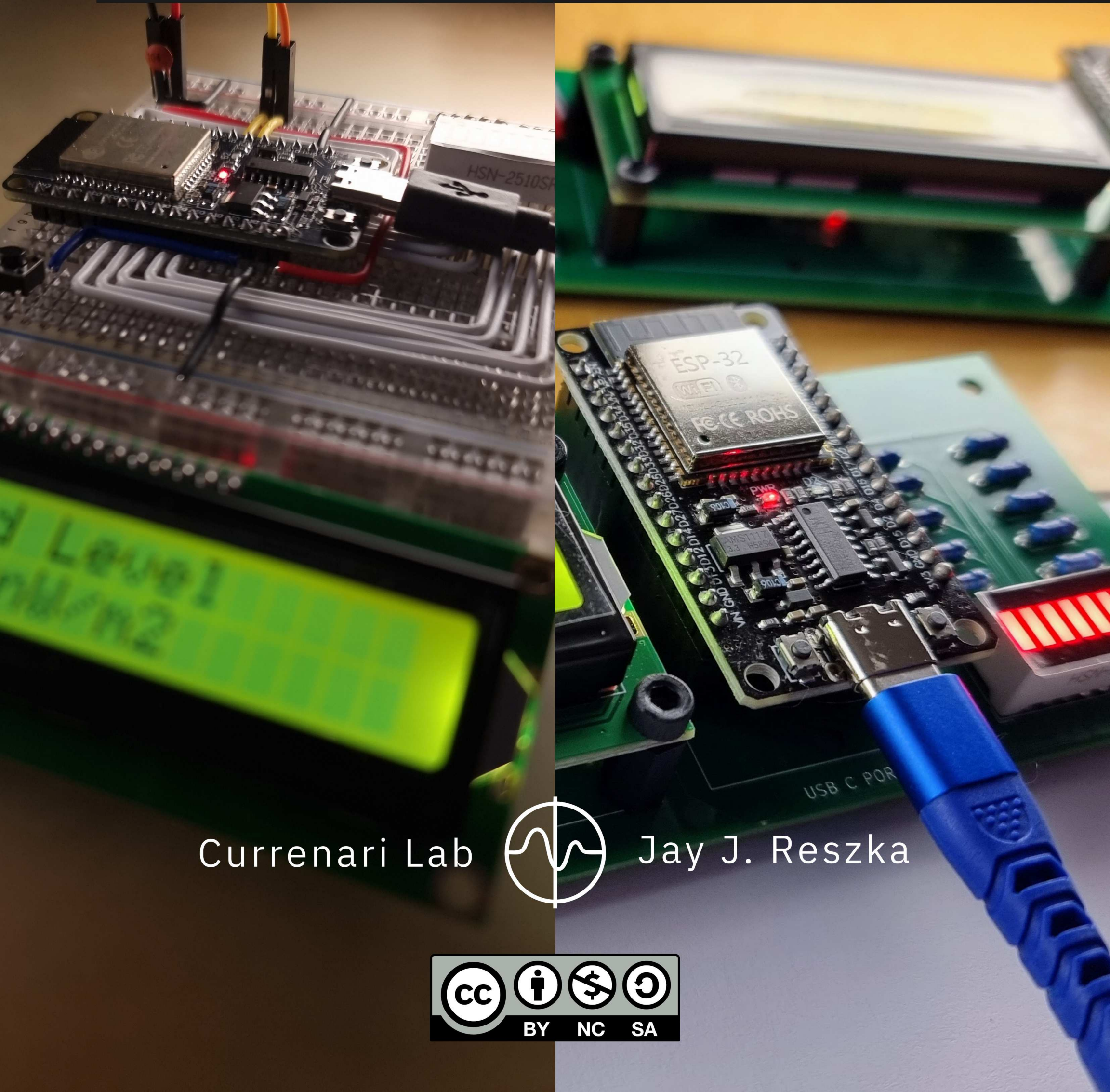
From Breadboard Prototype to Custom PCB

ESP32

Spectrum One WiFi Activity Monitor



Currenari Lab

Jay J. Reszka

ESP32 WiFi Activity Monitor

Spectrum One

From Breadboard Prototype to Custom PCB

Jay J. Reszka
Currenari Lab
England
February 2026

Copyright

Second edition

Dedication

For my granddaughter Amelia.
So you have something made by my hands, not just my words.
You build things. You take things apart.
You try, you fail, and you try again.
Never give up. If you want something, you go for it.
If one day you open this and recognise that feeling,
then this book has done its job.
Keep building. Stay curious. I’m proud of you.
For my future wife Ina,
for patience, support, and for making space for this to exist.
Without your love and support, this book would not exist.
— Jay

Preface

This document is a reference build and design archive for Spectrum One.

It exists as a practical record of a hardware project that can be returned to after notes are lost, files are buried, or details fade. A physical book remains one of the most reliable forms of technical reference, and this manuscript is intended to serve that role.

The material documents a complete ESP32-based hardware project from early idea prototyping through to a finished, reproducible build. The project is presented as it actually developed. Design decisions, constraints, and trade-offs are preserved rather than replaced with an idealised or optimised narrative.

The system began as a loose prototype assembled with jumper wires and an expansion adapter kit. As features were added, revised, and sometimes discarded, mechanical instability became a limiting factor. This progression ultimately led from ad-hoc prototyping to a consolidated breadboard build and, finally, to a dedicated printed circuit board.

For clarity and ease of reference, all build figures are collected in a dedicated section at the end of the book. Figures are referenced by number at the point where they are contextually relevant.

Contents

1 Introduction

Spectrum One is a small electronic device that observes Wi-Fi activity in the 2.4 GHz band and presents it as a live visual display.

The system is built around an ESP32, a ten-segment LED bar, and a two-row character LCD. As wireless activity in the surrounding environment changes, the display responds. The goal is not to analyse packets or measure RF characteristics, but to make otherwise invisible activity visible in a simple, immediate way.

The project begins on a breadboard to make changes uncomplicated and fast during early development. At the start, I had no clear plan for how the system should behave. Most decisions were made through trial and error, experimentation, and reading documentation from Espressif. The breadboard made it easy to change wiring, replace parts, and undo mistakes without friction.

As the device became more stable, the limits of the breadboard became obvious. The system worked, but it was mechanically awkward to handle, move, or use for any length of time. Wires came loose, parts shifted, and the build was not something that could reasonably be treated as finished.

At that point, the project moved beyond exploration. The breadboard build became a working baseline, and the next step was to translate that configuration into a PCB that would hold everything in place as a single, stable unit. The PCB version is not a redesign of the system. It preserves the same pin assignments, logic, and firmware behaviour developed during the breadboard phase.

This book documents the project as a complete build reference. It covers the breadboard implementation in detail, including wiring, pin usage, firmware, and observed behaviour. A corresponding PCB version exists and is derived directly from the same design.

Spectrum One is an observational device. It reacts to changes in Wi-Fi activity and displays relative behaviour over time. It is not intended for calibrated measurement, regulatory testing, or RF analysis.

1.1 Why I Built It

I built Spectrum One out of curiosity.

I wanted to understand how strong the Wi-Fi signal from my router actually was around my home. Not in theory, but in practice. I was looking for two things: a place as free from Wi-Fi interaction as possible for rest, and a place with reliable coverage for work.

Yes, I could have downloaded an app and been satisfied with numbers on a screen. But I wanted something physical on my desk. A standalone device that worked independently of a mobile phone, without background processes, notifications, or distractions.

Building it myself mattered. Projects like this expand my understanding and force me to learn things I would otherwise ignore. So instead of installing another app, I built a device and let the answers emerge from using it.

I also publish the source code, firmware binaries, schematics, and PCB Gerber files on my GitHub. They are open source and free for anyone to use.

The project repository is available at:

github.com/currenari

1.2 Who This Project Is For

This project is for people who want to build and understand a complete electronic system, not just assemble a module.

The device can be built, powered, modified, and reworked using basic tools. During development, all connections remain accessible, and the behaviour of the system can be understood directly from the hardware and firmware.

The focus is on seeing how a system behaves as a whole and how that behaviour changes as the design evolves.

1.3 What This Book Covers

This book provides a complete build reference for the Spectrum One device.

It includes:

- Breadboard wiring layouts and ESP32 pin usage
- A tested firmware build
- LED and LCD display behaviour
- Descriptions of how the device responds in use
- A bill of materials with part references
- The transition from a working breadboard build to a stable PCB version

The material is intended to support rebuilding, modification, and further development.

2 Project Requirements

This project is documented around a specific, tested reference build.

The requirements listed in this chapter reflect the exact hardware and software used during development and verification of the reference implementation. All wiring diagrams, pin assignments, build stages, and behaviour described in this book refer to this configuration unless explicitly stated otherwise.

Keeping a single, known baseline avoids ambiguity and makes it easier to reason about behaviour when something does not work as expected.

Where substitutions are practical or known to work, they are discussed later. Builders are encouraged to understand the implications of any changes they introduce.

2.1 Hardware

ESP32 Development Board

ESP32 Dev Kit V1 30-pin variant fitted with an ESP32-WROOM-32 module

This development board is used throughout the reference build. It provides predictable pin access, stable power regulation, and a form factor that works reliably on a breadboard.

Other ESP32 boards may differ in pin mapping, physical layout, antenna placement, or electrical behaviour. Those differences matter in a project that relies on repeatable wiring and consistent GPIO usage.

For clarity and consistency, this book assumes the ESP32 Dev Kit V1 unless stated otherwise.

Breadboards

a pair of half-size solderless breadboards, mounted side by side

If half-size breadboards are not available, two full-size solderless breadboards can be used instead. Both arrangements provide enough space for the reference build and are functionally equivalent.

The detailed breadboard layout, mounting, and mechanical considerations are covered in a dedicated chapter later in this book.

LED Bar Array

10-segment LED bar reference part: HSN-2510SR

The LED bar must consist of ten electrically independent LED segments. Each segment is driven individually through its own current-limiting resistor.

As an alternative, ten individual LEDs may be used instead of a bar module. In that case, resistor values must be chosen to match the LEDs being used.

LCD Module

16×2 character LCD (LCD1602) I2C backpack based on the PCF8574 I/O expander

LCD1602 modules are visually similar but mechanically inconsistent across manufacturers. Backpack layout, pin orientation, board spacing, and backlight circuitry often vary.

The reference build uses a PCF8574-based I2C backpack with a known address and predictable behaviour. Other backpacks may work, but differences are common and may require wiring or firmware changes.

Push Button

momentary tactile switch SPST, normally open 6 mm × 6 mm footprint

The push button is used for basic mode changes and interaction. No special characteristics are required beyond reliable contact and mechanical durability.

Passive Components

10 × 220 ohm resistors, 0.25 W, 5 % used for LED current limiting

1 × 100 nF ceramic capacitor (marked “104”) used for local power supply decoupling for the LCD module

The resistor value is not critical, but 220 ohms provides a good balance between brightness and GPIO current limits.

The capacitor helps stabilise the LCD supply and reduces display artefacts.

Wiring Materials

Dupont jumper wires or single-core solid wire 22 AWG recommended

The reference build uses 22 AWG single-core wire. This improves mechanical stability, visual clarity, and fault tracing during inspection and rework.

Standard jumper wires are often used during rapid prototyping for short periods. In this build, unstable jumper contacts were a frequent cause of LCD artefacts and intermittent behaviour.

2.2 Software

No proprietary software is required for this project.

The reference build uses standard, widely available tools that run on common operating systems.

Development Environment

a computer running Linux, macOS, or Windows USB access to the ESP32 development board

The reference build was developed using Linux, but all tools used are cross-platform.

ESP32 Toolchain

Espressif ESP-IDF

ESP-IDF is used to build, flash, and debug the firmware. The project uses ESP-IDF rather than the Arduino framework to avoid additional abstraction layers and to provide direct access to ESP32 features.

ESP-IDF is well integrated with Visual Studio Code through Espressif's official extension. This provides project configuration, building, flashing, and serial monitoring within a single environment.

The exact ESP-IDF version used for the reference build is documented in the project repository.

Serial Monitor

Any serial terminal capable of 115200 baud communication can be used.

During development, the integrated terminal in Visual Studio Code was used and proved stable and convenient. It is sufficient for observing status messages and runtime behaviour.

Standalone terminal programs such as minicom, screen, or picocom provide the same functionality if preferred.

Optional Tools

Tools used during development include:

- Visual Studio Code
- ESP-IDF tooling
- KiCad for schematic and PCB design

Alternatives may be used freely. The project does not depend on any specific editor or CAD tool.

2.3 Notes on Component Sources

Well-documented components are available from established electronics suppliers. Cheaper alternatives can often be sourced elsewhere, but they usually arrive without documentation and sometimes with inconsistent quality control.

In practice, low-cost versions of common modules almost never include datasheets, and it is common to find no usable documentation online at all.

3 The ESP32 Development Board Reference

Spectrum One is built around a breadboard-compatible ESP32 development board based on the ESP32-WROOM-32 module.

Before assembling any hardware, it is worth understanding what an ESP32 development board actually is, how boards built around the same module can differ, and why this project relies on a specific, well-defined variant. Many problems that surface later in a build are often attributed to wiring mistakes or firmware bugs. In practice, they usually originate much earlier, from assumptions made at the board level. This chapter surfaces those assumptions while they are still easy to correct.

3.1 What an ESP32 development board is

An ESP32 development board is not a single, fixed product.

It is a carrier board designed to make an ESP32 module practical to power, program, and wire. The ESP32 itself is contained inside a module such as the ESP32-WROOM-32. That module integrates the microcontroller together with flash memory, RF components, shielding, and an antenna connection. It is intended to be soldered onto a larger board rather than handled directly.

The development board exists to bridge that gap. The ESP silicon has no native USB interface. The development board provides it. It also handles voltage regulation, a reset and boot interface, and a physical layout suitable for breadboards or test fixtures. It makes decisions on the builder's behalf, including how the module is powered, how reset is generated, how USB interacts with the chip, and which signals are exposed on headers.

Two development boards can use the same ESP32-WROOM-32 module and still produce different results. Assuming equivalence at the board level is a common source of subtle failure in ESP32 projects. The differences are rarely obvious and usually surface only through timing, startup order, or conditions that appear unrelated.

3.2 What “ESP32-WROOM-32” refers to

The designation ESP32-WROOM-32 refers to a specific ESP32 module, not a development board.

Breaking the name down:

- ESP32 identifies the microcontroller family from Espressif Systems.
- WROOM indicates a general-purpose, shielded RF module intended for external integration.
- 32 identifies the module generation within the WROOM family and implicitly fixes the internal silicon and feature set.

In practical terms, ESP32-WROOM-32 means:

- Dual-core ESP32 SoC
- Integrated 2.4 GHz WiFi and Bluetooth
- External antenna connector or onboard PCB antenna, depending on variant
- 4 MB flash in the standard configuration

The key point is that the module name defines what is inside the metal can, not how it is exposed to the user.

Other ESP32 modules exist that may look similar but behave differently, for example:

- ESP32-WROOM-32E and ESP32-WROOM-32UE, which use revised silicon and different antenna arrangements
- ESP32-WROVER modules, which add external PSRAM

- ESP32-S, C, and H series modules, which belong to different ESP32 generations entirely

Development boards are built around these modules. Two boards may both advertise "ESP32" while using different modules with different electrical, memory, and startup characteristics.

That distinction matters later, when pin behaviour, boot stability, and peripheral availability stop being interchangeable.

The ESP32-WROOM-32 module defines what silicon is present and how radio functionality is implemented. It does not define how the device is powered, programmed, reset, or physically accessed. Those characteristics are entirely determined by the carrier board built around the module.

For this reason, identifying the module alone is not sufficient to determine board compatibility. Two boards using the same module may expose different GPIOs, apply different pull-ups or pull-downs, use different regulators, or handle reset and boot differently.

3.3 Why development boards differ

While the ESP32 chip inside the module may be identical, development boards differ in how that chip is supported and exposed.

Typical differences include header pin order, power routing and regulation, reset and boot handling, additional onboard components, and mechanical dimensions that affect breadboard mounting. These differences are not always documented clearly and are often revised silently between board versions.

A board may appear stable when powered via USB but operate unpredictably once external wiring is added. Another may flash firmware reliably yet fail to start consistently under load.

In many cases, the firmware is unchanged and the wiring appears correct.

The difference lies in the assumptions the board makes about power delivery, reset timing, and pin state during startup. No development board is neutral in this respect. Each one encodes design choices that shape the ESP32's startup sequence before firmware execution begins.

Spectrum One is designed, built, and tested against a single reference board so that these assumptions remain explicit and consistent.

3.4 Implicit electrical contracts of development boards

Every development board encodes a set of expectations about how it will be used. These expectations are rarely stated, but they matter.

A board may assume that power arrives via USB before any external signals are present. It may assume that certain pins remain floating or lightly loaded during reset. It may assume short ground paths, low source impedance, or minimal noise on control signals. When these assumptions are met, the board operates predictably. When they are violated, stability degrades without producing a clear failure mode.

These are not firmware assumptions. They are electrical contracts between the board and its environment.

Breadboards, jumper wires, external modules, and sensors often violate these contracts unintentionally. Long wires introduce inductance. Shared ground rails introduce coupling. External modules drive signals before the ESP32 is ready to receive them. None of these conditions are unusual, but they interact poorly with unstated board-level assumptions.

Recognising that these contracts exist changes how problems are diagnosed. Instead of asking why firmware appears unreliable, the focus shifts to whether the board is being used in a way its design implicitly assumes.

3.5 Reference board used in this project

This project uses the ESP32 Dev Kit V1 (30-pin) development board.

All wiring diagrams, GPIO pin assignments, physical placement rules, and described operation assume this exact board. The results observed while building Spectrum One come from this specific combination of ESP32 module, carrier board, and power source.

Using such boards is possible, but it requires manual adaptation, including reassessment of pin states before startup and during reset, verification of power routing, and confirmation of reliable reset and flashing under load.

Other boards are not prohibited. They are simply not treated as drop-in replacements.

3.6 Why breadboards change ESP32 operation

Breadboards are convenient, but they are electrically imprecise.

Contacts introduce variable resistance. Ground rails are shared across unrelated parts of the circuit. Long jumper wires act as inductors and antennas. Adjacent rows couple capacitively. These effects are often negligible for simple microcontrollers, but the ESP32 operates at higher speeds and tighter tolerances.

The ESP32 is particularly sensitive during startup. During this phase, internal regulators stabilise, boot configuration pins are sampled, and subsystems initialise in a fixed sequence. Electrical noise or unintended loading during this window can alter outcomes without producing an obvious fault.

This is why a circuit may operate correctly after a reset but fail on a cold power-up. It is also why adding a single external module can destabilise a previously working setup.

Spectrum One's reference configuration is chosen to minimise these variables so that observed results reflect the system itself rather than the construction environment.

3.7 Understanding pin roles

An ESP32 development board exposes many pins, but not all of them serve the same role.

Some pins are intended for general-purpose input and output. Others participate in boot configuration, flash access, or internal subsystems that are active before firmware execution begins. These distinctions are documented in datasheets and schematics, but they are easy to overlook once the board is mounted on a breadboard.

Pins that share similar positions on the header can serve different roles internally. Choosing the wrong pin may appear to work at first, but often fails during power-up, reset, or when external hardware is added.

Typical symptoms include unreliable startup, unexpected resets, inconsistent flashing, or systems that operate correctly only after repeated resets. These effects are usually caused by pin states during power-up rather than errors in the firmware itself.

3.8 Pin state across reset

Each GPIO pin passes through three functional phases during startup:

- Power-off and power ramp
- Reset and boot configuration
- Normal firmware-controlled operation

Most documentation focuses on the third phase. Many failures originate in the second.

During reset, certain GPIOs are sampled to determine the boot mode. Others may be temporarily undriven or float briefly before internal pull-ups or pull-downs become effective. External circuitry connected to these pins can influence their state at precisely the wrong moment.

A pin that behaves perfectly as an output during normal operation may prevent the system from starting if it is driven externally during reset. This is why a design that works reliably after a warm reset may fail intermittently or consistently after a full power cycle.

3.9 Why pinouts look complex

ESP32 pinout diagrams appear complex because many internal functions share the same physical pins.

A single GPIO may support digital input or output, analogue-to-digital conversion, touch sensing, or peripheral signalling depending on configuration. These functions are multiplexed internally. Development boards expose the physical pins directly, which is why the same pin may appear under multiple labels in reference diagrams.

Pinout tables describe capability, not suitability. They show what a pin can do, not whether it should be used in a given role. Startup operation, internal pull-ups, and boot sensitivity are often omitted or visually de-emphasised.

Input-only pins

Some ESP32 pins are input-only.

These pins can read signals but cannot drive loads. They are suitable for sensors, buttons, and logic-level inputs, but unsuitable for anything that requires current drive or stable output signalling. Using an input-only pin as an output often fails silently. The firmware may compile and flash correctly while the connected hardware never responds.

Spectrum One avoids this class of error entirely. Input-only pins are never used for outputs, and all LED bar connections use GPIOs capable of stable digital output.

Boot-sensitive pins

Certain ESP32 GPIO pins influence startup operation.

If these pins are held high or low while power is applied, the ESP32 may enter programming mode, fail to start normally, or reset repeatedly. These decisions occur before firmware execution begins.

Problems here are often misdiagnosed as firmware faults because they occur inconsistently and disappear once the system is running. In reality, they are caused by external wiring altering pin states during power-up.

Spectrum One avoids boot-sensitive pins for external connections. Startup operation remains consistent, resets occur only when intended, and firmware flashing remains reliable.

The EN pin

The ESP32 Dev Kit V1 exposes an EN pin.

EN is the enable input for the ESP32. When EN is high, the chip runs normally. When EN is pulled low, the chip is held in reset. On this board, EN is connected to the onboard reset button and associated circuitry that defines a clean reset domain.

Although EN appears on the pin header, it is not a general-purpose signal. Noise, loading, or external connections on EN affect the entire system rather than a single GPIO. Even minor disturbances can result in spontaneous resets or failed startups.

In Spectrum One, the EN pin is left untouched. Reset control remains under the control of the board's onboard interface and USB connection.

The VIN pin

The ESP32 Dev Kit V1 exposes a VIN pin intended for external power input.

VIN feeds the board's voltage regulation path, which supplies the 3.3 V rail required by the ESP32. Stability depends not only on voltage level but on how that voltage is delivered. Current capability, wiring resistance, regulator headroom, and thermal conditions all influence operation.

When powered via USB, the board receives a stable 5 V supply that places the regulator within its intended operating range. Startup timing is predictable and operation is stable.

When VIN is powered externally, especially at higher voltages, thermal load increases and sensitivity to wiring quality rises. The system may remain within nominal voltage limits while still operating erratically under load.

Spectrum One powers the ESP32 Dev Kit V1 from USB with 5 V. The USB connection provides a 5 V rail that is present on VIN and feeds the onboard voltage regulator. That regulator supplies a stable 3.3 V rail to the ESP32 silicon inside the module.

3.10 Silkscreen labels and D-pin markings

ESP32 development boards commonly include silkscreen labels printed along the pin headers. These labels are visual aids, not specifications.

Labels beginning with the letter D are board-level aliases chosen by the board designer. They are not GPIO numbers and have no meaning to the ESP32 itself. A pin labelled "D5" may correspond to GPIO 5 on one board and something else on another.

From the ESP32’s perspective, only GPIO numbers exist.

For this reason, Spectrum One uses GPIO numbers exclusively. All references are verified against the board pinout and the project’s GPIO assignments. Silkscreen markings may assist with orientation, but they are never treated as a source of truth.

3.11 Pin selection and design discipline

The ESP32 provides many GPIO pins, but only a small subset is required for this project.

Spectrum One uses only the pins needed to drive the LED bar, read the push button, and communicate with the display. All selected pins have predictable startup operation and stable electrical characteristics.

This simplifies wiring, makes faults easier to isolate, and allows the circuit to be understood without constant reference to datasheets. Designs that are clear on a breadboard translate more directly into clean schematics and predictable PCB layouts.

3.12 Why two identical boards differ

Boards sold under the same name are not always electrically identical.

Revisions change silently. Regulators are substituted. USB interface chips are replaced. Pull-up values change. None of these modifications are visible from the silkscreen.

As a result, advice that works for one board may fail on another, even when both appear identical. This is a common source of confusion in forums and tutorials.

Anchoring Spectrum One to a specific reference board avoids this ambiguity. Observed results can be trusted as repeatable rather than incidental.

3.13 The ESP32 ecosystem in context

The ESP32 appears frequently in hobby projects, but it exists at industrial scale. Espressif Systems has shipped well over one billion IoT chips worldwide. The ESP32 is embedded in consumer products, industrial systems, monitoring equipment, and appliances that are never identified as ESP32-based to the end user.

At this scale, variation is inevitable. There is no canonical ESP32 development board. The term “development board” describes convenience, not standardisation.

Spectrum One is designed with this reality in mind.

3.14 What this chapter is protecting you from

This chapter exists to prevent a specific class of failure.

It is not about learning every ESP32 feature. It is about avoiding failures that feel mysterious, intermittent, or resistant to debugging. By fixing assumptions early, later results become explainable rather than frustrating.

3.15 Terminology used in this chapter

Pinout

The physical header map of a development board. It describes which labels appear on which physical pin positions along the board edges. Pinout is a property of the board itself and is documented in Figure 1 for the ESP32 Dev Kit V1 reference board.

GPIO pin assignments

The project-specific mapping of GPIO numbers to functions. It defines which GPIOs Spectrum One uses for the LED bar, push button, and display interface. This mapping is reflected consistently in wiring diagrams, firmware definitions, and tables, and is documented in Figure 2.

3.16 Reference figures and verification

Figure 1 shows the complete physical pinout of the ESP32 Dev Kit V1 (30-pin) reference board. Its purpose is orientation. It establishes the fixed relationship between the ESP32 module, the carrier board, and the exposed header pins.

Figure 2 shows the GPIO pin assignments used by Spectrum One on the same board. The highlighted pins indicate which GPIOs are actively used by the project and which are intentionally avoided. These selections ensure predictable startup behaviour and reliable interaction with external hardware.

The two figures do not represent different boards or alternative configurations. They describe the same hardware from different perspectives. Figure 1 answers what the board exposes. Figure 2 answers what the project uses.

If a different ESP32 development board is used, the physical pin order, available GPIOs, and startup characteristics must be re-evaluated against these roles. In that case, the reference assumptions described in this chapter no longer apply directly.

3.17 Real-world examples of board-level differences causing failure

The consequences of treating ESP32 development boards as interchangeable are not theoretical. They appear repeatedly in commercial products, field deployments, and large community projects.

What follows are representative examples where outcomes initially blamed on firmware were later traced to board-level differences, power assumptions, or undocumented hardware variation.

Product failures caused by ESP32 board and variant assumptions

In commercial product development, ESP32 failures are frequently caused by incorrect assumptions about the hardware platform rather than by code defects. Industry post-mortems document cases where products reached late development or early production before hardware limitations became visible.

Common failure modes include incorrect power budgeting, unsuitable voltage regulation, misjudged flash memory size, and peripheral availability that differed subtly between ESP32 variants or board revisions. In several cases, products required redesign after PCB fabrication because the selected ESP32 board or module did not operate as expected under real load.

These failures were not caused by misuse of the ESP32 itself. They resulted from treating a development board as a neutral abstraction rather than as a specific electrical design with constraints.

Field devices rendered inoperable by board-dependent boot operation

ESP32 devices deployed in the field have been reported to become permanently unresponsive following power cycles or firmware updates, despite working reliably during development and testing.

In documented cases, affected devices could not be recovered through reset or reflashing. Investigation pointed toward boot-time conditions influenced by board-level factors such as power ramp characteristics, reset timing, or pin states during startup. The same firmware continued to operate normally on development units in the lab.

This class of failure is particularly dangerous because it does not present as a firmware crash. The device appears electrically alive but fails before user code executes. In production environments, this results in bricked units rather than debuggable faults.

Instability reported across "identical" ESP32 boards

Large user communities consistently report ESP32 systems that reset, disconnect from WiFi, or operate erratically on some boards but not others, even when running identical firmware.

Patterns emerge in these reports. Problems correlate with certain board manufacturers, regulator designs, USB interfaces, or power input methods. Switching to a different ESP32 board often resolves the issue without any code changes.

These cases are often dismissed as poor power supplies or faulty units. In reality, they demonstrate that development boards sold under the same name are not electrically equivalent. Differences in component choice, layout, or revision history directly affect operation.

Evidence from broader IoT failure analysis

Academic analysis of IoT system failures shows that hardware integration issues are a dominant source of bugs in deployed devices. These include power integrity problems, undocumented interactions between components, and incorrect assumptions about platform behaviour.

The ESP32 is not unique in this respect. However, its high integration level, RF subsystems, and boot-time complexity amplify the impact of small hardware differences. Systems that appear robust in controlled environments can fail when exposed to real-world power and wiring conditions.

3.18 Why these examples matter for Spectrum One

Each example shares a common pattern:

- Firmware was initially suspected
- Results were inconsistent or environment-dependent
- The root cause lay in board-level assumptions

These failures are not edge cases. They are the natural outcome of treating development boards as interchangeable abstractions rather than as concrete electrical designs.

Spectrum One avoids this class of problem by fixing the reference board, power source, and pin usage from the outset. This does not eliminate complexity, but it makes results observable, repeatable, and diagnosable.

This ensures that when results change, the cause is identifiable rather than mysterious.

3.19 References

1. Espressif Systems Community Forum esp32.com (ESP32 boot behaviour, brownout and startup issues)
2. Predictable Designs (2021) “ESP32 Design Mistakes That Kill Your Product” predictabledesigns.com
3. A. M. B. et al. (2021) “An Empirical Study of Bugs in IoT Systems” dl.acm.org

4 When the ESP32 Only Knows Itself

This chapter is about a boundary that is easy to miss.

After learning how development boards, modules, pinouts, and GPIO assignments work, it is tempting to assume that anything unclear can be resolved through software, inspection, or experimentation. Many ESP32 problems persist because of this assumption.

The ESP32 does not see the system the way a human does.

It has no awareness of the board it is mounted on, no knowledge of silkscreen labels, and no understanding of how its signals are routed once they leave the silicon. Understanding where this knowledge ends changes how pin-related problems are interpreted and prevents entire classes of failure before they occur.

4.1 The ESP32 only knows itself

Internally, the ESP32 identifies digital signals using GPIO numbers only. These GPIO numbers are fixed in the silicon. They do not change. They are the only pin-related identifiers the chip understands.

What the ESP32 does not know:

- which development board it is mounted on
- how GPIOs are routed to headers or connectors
- what text is printed next to those pins
- how a manufacturer chose to group or name them
- which pins are convenient, awkward, or dangerous to use

From the ESP32's perspective, the development board does not exist as a concept.

There is no internal model of headers, pin order, silkscreen, USB connectors, or board layout. There is only silicon, internal routing, and numbered signals.

This single fact explains most ESP32 pin confusion encountered in practice.

4.2 Pins are not GPIOs

On the ESP32, as on all modern microcontrollers, pins and GPIOs are not the same thing.

A pin is a physical connection point. It is a bonded pad on the package that allows electrical signals to enter or leave the chip.

A GPIO is an internal digital signal controlled by the microcontroller. It exists entirely inside the silicon and is configured by software.

The connection between the two is intentional, not automatic.

Inside the ESP32 is an internal routing structure that connects GPIO signals to physical pins and to internal peripherals. From the chip's point of view, GPIOs are abstract resources. Pins are merely places those resources may appear.

This distinction is often blurred in documentation, which is why pins and GPIOs are frequently treated as interchangeable. Electrically and behaviourally, they are not.

A useful way to think about it is this:

- GPIOs are what the ESP32 controls
- pins are where those signals emerge
- Some GPIOs are input-only.
- Some pins are shared with internal functions.
- Some pins behave differently during startup.
- Some GPIOs cannot be freely used at all times.

This is why a pin that looks reasonable on a diagram may fail silently or behave unpredictably in practice.

4.3 Why this feels counter-intuitive

From a human point of view, the development board is the device. You hold it, wire it, and read the labels printed next to the pins. It is natural to assume the chip shares this understanding.

It does not.

The ESP32 cannot tell whether a GPIO is connected to an LED, a sensor, nothing at all, or several things at once. Once a signal leaves the silicon, it disappears from the chip's awareness.

Accepting this mismatch between human perception and chip reality removes a great deal of confusion.

4.4 Silkscreen labels are not authoritative

Silkscreen labels are added by board designers to help humans assemble hardware. They are not part of the ESP32 specification and are invisible to the chip.

Labels vary for many reasons:

- limited PCB space
- legacy naming conventions
- attempts to resemble other platforms
- aesthetic or marketing decisions

None of these choices affect ESP32 behaviour.

Two boards using the same ESP32 module may label pins differently while remaining electrically equivalent. Conversely, two boards with similar-looking labels may behave very differently.

Silkscreen is a convenience layer. It is never authoritative.

4.5 The translation step that always matters

Every ESP32 project requires an explicit translation step:

Human-facing label → GPIO number

That translation must happen before wiring hardware or writing firmware.

Firmware operates exclusively on GPIO numbers. The ESP32 never sees the labels printed on the board. Skipping this translation produces systems that appear correctly wired but behave inconsistently or only under specific conditions.

Once this step becomes habitual, most pin-related problems disappear.

4.6 What the development board actually defines

Pin mapping is a property of the development board, not the ESP32.

The board designer decides:

- which ESP32 pins are routed to headers
- which pins are omitted entirely
- which pins are shared with onboard components
- how pins are grouped and labelled

There is no universal ESP32 board-level pin layout.

Even boards sold under the same name may change subtly between revisions. Header order, shared connections, or available pins can shift without notice.

This is why pin identification is always a documentation task. It cannot be solved purely through inspection or software.

4.7 What software cannot tell you

The ESP32 has no mechanism to report how its GPIOs are routed once they leave the silicon.

There is no register, API, or diagnostic interface that exposes board-level routing. Firmware can configure GPIO behaviour, but it cannot discover where those signals go.

This is not a tooling limitation. It is a structural boundary by design.

Software cannot tell you:

- which header pin corresponds to a GPIO
- whether a GPIO is shared with onboard hardware
- whether a pin is safe during startup

Those questions belong outside the chip.

4.8 The GPIO matrix and the illusion of rerouting

The ESP32 includes an internal GPIO matrix.

This allows many internal peripheral signals such as UART, SPI, I2C, and PWM to be connected to different GPIO numbers than their default ones. From software, this can look like "remapping pins".

What is actually happening is more subtle.

Inside the silicon, an internal signal is being connected to a different GPIO number. The physical pin associated with that GPIO does not change.

In other words:

- software can change which GPIO carries a signal
- software cannot change where that GPIO exists physically

No firmware setting can alter which package pin a GPIO is bonded to, which header it appears on, or how the development board routes it.

Software cannot reroute copper.

This distinction explains why remapped peripherals still fail when the chosen GPIO is not exposed on the board, is input-only, or is boot-sensitive. The internal routing succeeded. The physical constraints did not.

4.9 What software can do instead

Software can confirm assumptions, not discover facts.

Typical confirmation methods include:

- toggling a GPIO and observing which header pin changes state
- enabling internal pull-ups and measuring voltage externally
- configuring known input-only GPIOs and observing output failure

These techniques work because electrical behaviour is observable.

They are useful when documentation is incomplete, but they are slow and risky when boot-sensitive pins are involved.

Example: confirming a GPIO mapping

Connect an LED with a resistor to the header pin you believe corresponds to GPIO18.

```
const int testPin = 18;

void setup() {
  pinMode(testPin, OUTPUT);
}

void loop() {
  digitalWrite(testPin, HIGH);
  delay(500);
  digitalWrite(testPin, LOW);
  delay(500);
}
```

If the LED blinks, the mapping is confirmed.

If it does not, the assumption was wrong.

A silent failure

Some ESP32 GPIOs are input-only.

Using such a GPIO as an output produces no error. The code compiles. The firmware runs. The hardware does nothing.

Without a correct mental model, it is easy to blame wiring or firmware when the cause is neither.

Boot-sensitive pins and early failure

Some GPIOs influence how the ESP32 starts.

Their state during power-up determines whether the chip boots normally, enters programming mode, or fails to start.

Failures caused by these pins occur before firmware execution begins. Debugging them in code is futile.

Correct pin identification prevents this entire class of problems.

The RF shield and misleading clues

The metal enclosure on the ESP32 module is an RF shield.

It is electrically grounded and part of RF and EMC compliance. Markings on the shield vary and do not describe pin behaviour or routing.

Removing the shield provides no useful insight and risks permanent damage. It is not a diagnostic tool.

4.10 Core insight

Pin identification is not a software problem. It is not a firmware problem. It is a development-board problem.

Once this boundary is understood, ESP32 operation becomes predictable rather than frustrating. Problems stop feeling random because their causes are no longer invisible.

4.11 Reference figure 3

Figure 3 illustrates two ESP32 development boards populated with ESP32-WROOM-32 modules. Differences in silkscreen, shield markings, and board layout do not imply differences in GPIO behaviour.

5 Breadboard layout and mechanical preparation

Wide range of development boards expose a practical limitation of standard solderless breadboards.

Boards such as the ESP32 Dev Kit V1 occupy nearly the full width of a breadboard. When inserted in the usual way, the development board covers nearly all available tie points on the breadboard, leaving little or no access for wiring. This limitation is mechanical rather than electrical. It arises from the fixed geometry of the breadboard and the width of the development board, not from any characteristic of the ESP32 itself.

The approach documented here alters the breadboard layout to remove this limitation. It applies equally to half-size and full-size solderless breadboards.

The goal is simple: reposition the breadboard terminal strips so the existing tie points beside each pin row are no longer obstructed by the development board body.

5.1 Physical layout and spacing

Figure 4 shows the physical arrangement used to achieve this.

Two solderless breadboards are mounted side by side on a flat mounting surface. One power rail is removed from each breadboard so that the inner edges face an open space, forming a central clearance gap between the two terminal strip areas.

The clearance gap does not define pin spacing and is not tied to a specific dimension. The ESP32 Dev Kit V1 has a pin-row spacing of 22.86 mm, but the breadboards are positioned closer together than this value. The development board bridges the gap while inserting cleanly into the tie points on each side.

Keeping the gap narrower than the ESP32 pin spacing is intentional. It allows the same modified base to be reused with development boards that differ in width or pin arrangement. The Raspberry Pi Pico 2 shown later in Figure 5I is included only as an example of this reuse with other development boards.

This layout restores full access to tie points on both sides of the development board while keeping the board mechanically relaxed and free from lateral stress.

Figure 4 describes the resulting layout concept. The following chapter documents the physical modification steps used to create it.

6 Breadboard modification sequence

This solution exists because buying adapters became a dead end.

Every development board seems to require a different carrier, breakout, or mounting adapter. Over time this turns into a drawer full of expensive, single purpose parts that only solve one mechanical problem and create another. Switching boards means ordering more hardware, waiting, and rebuilding the same setup again.

This modification came out of frustration with that cycle. Instead of adapting the breadboard to a specific board, the breadboard is stripped back to the minimum needed and the development board itself defines the spacing. Removing only the inner power rails creates enough clearance for a wide range of boards without locking the layout to any single footprint.

The result is a base that works across projects, boards, and revisions without additional adapters or accessories.

Figure 5 documents the physical modification sequence and resulting layout.

- 5A Two solderless breadboards before modification
- 5B Inner power rails identified for removal
- 5C Breadboard inverted on a flat cutting surface
- 5D Power rail separated by cutting through the self-adhesive layer
- 5E Power rails removed from both breadboards
- 5F Development board placed across the gap for a soft alignment check (side view)
- 5G ESP32 Dev Kit mounted across the clearance gap (top view)

- 5H Close-up showing restored access to tie points beside the mounted board
- 5I Alternative development board mounted on the same base as an example of reuse

One power rail is removed from each breadboard to create the clearance gap. No further modification is required. The base can be reused indefinitely.

6.1 Mounting and alignment

After modification, the breadboards are placed on a flat mounting surface such as aluminium dibond, plastic sheet, or any rigid, dimensionally stable material. At this stage, they are positioned but not fixed.

6.2 Alignment check

Figure 5F shows a development board placed across the clearance gap while the breadboards remain movable. This step is used to verify alignment before committing to adhesive fixing.

The development board is inserted to confirm that both pin rows enter the terminal strips cleanly and without lateral force. Any resistance at this stage indicates misalignment, which leads to unreliable electrical contact and long-term mechanical stress.

Figure 5G shows the ESP32 Dev Kit mounted after correct alignment has been achieved.

Once alignment is confirmed, the breadboards are fixed in place using their self-adhesive backing.

Figure 5H shows a close-up view of the mounted board, illustrating unrestricted access to tie points on both sides.

6.3 Multi-board compatibility

The modified breadboard base is not tied to a single development board.

Figure 5I shows a different development board mounted on the same base without further modification. This confirms that the mechanical layout supports boards with different widths and pin arrangements, provided their pin rows align with standard terminal strips.

7 Breadboard component placement

This chapter documents the physical component placement used in the Spectrum One reference build.

The placement reflects deliberate decisions made during the build. Nothing here is presented as a rule or requirement. The purpose of this chapter is to explain why components were placed where they are, not to prescribe a layout that must be copied.

Figure 6 shows the physical component placement used in the Spectrum One reference build, drawn to match the actual breadboard layout.

7.1 Placement as design thinking

Component placement on a breadboard is often treated as arbitrary. In this build, it is used as an early design exercise.

Each placement decision considers mechanical access, electrical context, and the later transition to a PCB layout. The layout is not optimised for appearance or symmetry. It is optimised for understanding.

7.2 ESP32 development board placement

The ESP32 development board is mounted across the central clearance gap of the modified breadboard base.

The board is mounted across the clearance gap between the two breadboards. This keeps the board body out of the tie point area and leaves the existing tie points beside each pin row available for wiring.

The antenna end of the ESP32 faces away from the majority of surrounding components and wiring. This follows Espressif hardware design guidance, which recommends keeping the antenna region clear of nearby metal, copper, and other components.

On a solderless breadboard this condition cannot be fully met, but orienting the antenna away from other components reduces detuning and performance loss. The same antenna intent is applied without compromise in the PCB version of the project.

7.3 LED bar array placement

The 10 segment LED bar array is placed across the central channel of the right hand breadboard.

The package exposes one pin row on each side. Each LED segment has its own anode pin on one side of the package and its own cathode pin on the opposite side. There is no internal common connection.

Functionally, the LED bar array behaves as ten independent LEDs packaged side by side in a single housing.

Placing the array across the central channel on the right hand breadboard uses the breadboard geometry to physically separate the anode and cathode pin rows.

Each cathode pin aligns directly with its corresponding current limiting resistor through the breadboard's internal conductive strip. This direct alignment removes the need for jumper wires between the LED bar and the GND power rail.

This placement reduces wiring complexity and makes the electrical structure of the LED array immediately visible.

7.4 Resistor placement

Each LED segment uses its own current limiting resistor.

The resistors are placed directly adjacent to the LED bar array so that each cathode pin lines up with its corresponding resistor. This alignment is intentional and reflects deliberate use of the breadboard's internal design.

The resistors appear angled relative to the main terminal grid because the power rails are segmented in groups of five tie points and offset from the grid. This angle is dictated by breadboard construction rather than assembly choice.

The placement is mechanically correct and avoids unnecessary lead deformation.

7.5 Push button placement

The tactile push button is placed on the left hand breadboard in a position that allows comfortable finger access during operation.

The switch is not placed across the central channel. For this type of tactile switch, doing so provides no electrical benefit.

This placement avoids imposing constraints that are not required by the component itself.

Internal switch structure and operation

The tactile switch used in this build is a standard four pin, momentary, normally open device.

Internally, the two pins on one side of the package form a permanently connected pair, and the two pins on the opposite side form a second permanently connected pair. Pressing the button bridges these two internal pin pairs.

Electrically, the switch behaves as a two terminal component. The additional pins provide mechanical stability only. Separating the pins across a breadboard channel does not improve operation.

7.6 Capacitor placement

A single ceramic capacitor is placed in the LCD power path.

The capacitor provides local decoupling between VCC and GND at the point where the display draws current. On a solderless breadboard, long power paths and shared rails make this especially relevant.

Placing the capacitor close to the LCD wiring reduces transient voltage drops caused by display switching activity.

No additional capacitor is used for the push button. The button is debounced in software to handle mechanical contact bounce, and an RC network is not required for reliable operation in this build. This separation reflects an intentional design choice rather than component grouping by proximity. Mechanical bounce occurs on the millisecond scale and can register as multiple transitions unless filtered.

7.7 Wiring length and intent

Some connections are kept short where the breadboard layout allows it. Others are necessarily longer due to the physical constraints of the breadboard format.

Power decoupling and display related paths benefit from short loops, but on a solderless breadboard these cannot always be achieved. Other signal lines are tolerant of longer routing and remain reliable in this context.

This layout reflects the limitations of the breadboard rather than an attempt at full routing optimisation, which is addressed later in the PCB design.

7.8 Interpretation of the reference layout

The placement shown is one valid solution. It is not the solution.

Components may be moved, reoriented, or replaced without breaking the project, provided their electrical roles are respected. The value of this chapter lies in understanding why the placement works, not in copying it blindly.

This mindset becomes critical when transitioning from breadboard builds to PCB design.

7.9 What this layout teaches before PCB design

This breadboard layout acts as an early design filter. It forces decisions that matter later when the circuit becomes fixed.

Lessons that transfer directly to PCB work include:

- antenna placement matters early
- component proximity follows electrical context
- not all signals require equal optimisation
- mechanical access is part of design
- breadboard geometry exposes hidden assumptions
- placement is reasoning, not decoration

Treating breadboard placement this way reduces rework and shortens the transition from prototype to PCB.

8 Breadboard Wiring

This chapter documents how the Spectrum One breadboard build is wired, step by step, from an empty base to a fully working reference system.

The wiring process described here is not about copying shapes from photographs. It is about translating a schematic into physical connections in a controlled and repeatable way, while keeping track of progress and avoiding hidden errors.

Breadboard wiring is often treated as an informal activity. In this project, it is treated as an engineering task.

8.1 What this chapter covers

This chapter explains how the schematic for Spectrum One is translated into a physical breadboard build, and how wiring is executed in a way that remains understandable, verifiable, and repeatable over time.

It describes:

- why breadboard wiring fails without structure
- how wiring progresses through distinct phases
- how correctness is tracked during incomplete states
- why a wiring table is used instead of memory or visual copying

8.2 Why breadboard wiring fails without structure

This chapter exists to address four structural problems that occur when a schematic is translated into a physical breadboard build.

Schematics collapse time

A schematic represents the final electrical state of a circuit. It contains no information about the order in which connections are made, removed, or temporarily absent during construction.

Breadboard builds are stateful

At any moment during wiring, the circuit exists in a partial state. This state may be electrically incomplete, logically inconsistent, or temporarily contradictory to the schematic.

Visual inspection is misleading

During intermediate states, a build can look wrong while being correct for that stage, or look correct while hiding missing or duplicated connections.

Appearance does not map reliably to electrical truth.

Our working memory is fragile under repetition

Long wiring sessions overload short-term memory. This leads to skipped steps, repeated connections, and uncertainty about what has already been done, especially when tasks are interrupted.

At no point during wiring does the breadboard know what stage it is in. Only the builder does, and only if progress is tracked explicitly.

8.3 What exists before wiring begins

Before the first wire is placed, several things already exist:

- a defined idea and functional goal
- selected components with known electrical behaviour
- datasheets reviewed for pin functions and constraints
- a complete schematic developed in advance
- preparation for a wiring table derived from the schematic

At this point, nothing is yet physical. All decisions exist on paper and in the schematic.

The schematic defines what must be connected. The breadboard wiring is the execution of that definition.

8.4 Two wiring phases

Breadboard wiring in this project is split into two distinct phases.

Exploratory wiring

An early physical build is usually fast and untidy.

At this stage:

- any available breadboard may be used
- jumper wires are mixed lengths and colours
- clarity is secondary to basic function

The purpose here is verification and speed.

This phase is used to answer one question only:

Does the circuit behave as intended?

Connections are added, removed, and re-routed freely.

This build is temporary and not preserved.

Reference wiring

Once the intended behaviour is confirmed, the exploratory wiring phase ends.

The temporary build is set aside, and the circuit is rebuilt deliberately as the reference breadboard build.

From this point onward, wiring is no longer exploratory. Correctness, traceability, and repeatability become the goal.

8.5 Complete wiring definition

With behaviour confirmed, the schematic becomes the authoritative reference.

The schematic defines the full electrical system and remains the reference for the rest of the build.

Figure 7 shows the complete schematic for the Spectrum One reference build as developed in KiCad. This schematic provides the wiring definition against which all breadboard work is carried out.

The schematic is available at full resolution in the project repository on the Currenari GitHub account.

8.6 Why a wiring table is used

Some readers are comfortable wiring directly from a schematic. Others find it difficult to maintain order once multiple connections overlap and the physical build no longer resembles the abstract diagram.

To address this, the schematic is translated into a wiring table.

The wiring table preserves schematic intent while imposing sequence and physical reference points.

It replaces memory and visual copying with an explicit, checkable process.

8.7 Wiring states and controlled progress

A schematic represents the final state of a circuit. A breadboard build does not start in that state.

Instead, it moves through intermediate wiring states.

An intermediate wiring state:

- is not complete
- is not expected to function
- may look incorrect if judged visually

This is normal.

Progress is measured against the wiring table, not against photographs or memory.

Photographs serve only as contextual reference. They are not a wiring source and not a wiring target.

8.8 Breadboard identification and orientation

Two breadboards are used throughout the build (see Figure 8):

- **BBL** — Breadboard Left
- **BBR** — Breadboard Right

These identifiers remove ambiguity when connections span both boards.

Tie-point coordinates are always given relative to BBL or BBR and correspond to the physical orientation shown in the reference figures.

8.9 Wiring table reference

This section defines the wiring steps used to execute the build.

Each step represents one physical wiring action and must be completed sequentially.

These steps are the primary source of truth during wiring.

At this stage, the current-limiting resistors for the LED bar and the decoupling capacitor are not yet installed.
They are added later in their dedicated wiring sections.

8.10 Wiring convention

General rules

- All wires are inserted into breadboard tie-points.
- Tie-points adjacent to the ESP32 headers are electrically connected to the ESP32 pins.

- These tie-points inherit the GPIO identity of the connected ESP32 pin.
- GPIO numbers identify the signal carried by that connection.
- Each wiring step describes one wire placed between two specific tie-points.

Ticking steps helps maintain continuity and reduces errors.

Anode and cathode reminder

To help remember the difference between anode and cathode, a simple mnemonic can be used.

A for Anode is the first letter of the alphabet and corresponds to the positive side of a diode, like an **A+** rating.

This applies to diodes in general. An LED is a type of diode.

8.11 ESP32 to 10-LED bar array — signal wiring

The following steps connect the ESP32 GPIO pins to the LED bar anodes.

ESP32 Dev Board — Breadboard Left Side

[ ] Step 1: BBL f6 (GPIO32) → BBR a30

[ ] Step 2: BBL f7 (GPIO33) → BBR a29

[ ] Step 3: BBL f8 (GPIO25) → BBR a28

[ ] Step 4: BBL f9 (GPIO26) → BBR a27

[ ] Step 5: BBL f10 (GPIO27) → BBR a26

ESP32 Dev Board — Breadboard Right Side

[ ] Step 6: BBR d7 (GPIO18) → BBR a25

[ ] Step 7: BBR d6 (GPIO19) → BBR a24

[ ] Step 8: BBR d5 (GPIO21) → BBR a23

[ ] Step 9: BBR d2 (GPIO22) → BBR a22

[ ] Step 10: BBR d1 (GPIO23) → BBR a21

At this point, all LED bar anodes are wired to their corresponding GPIO pins.

The LEDs will not yet light. Their cathodes are not connected.

8.12 10-LED bar array to GND — resistor wiring

Each LED bar segment has two legs:

- an anode, already wired
- a cathode, which must be connected to GND through a resistor

Each segment uses one 220 ohm resistor.

[ ] Step 11: BBR j30 → GND rail

[ ] Step 12: BBR j29 → GND rail

[ ] Step 13: BBR j28 → GND rail

[ ] Step 14: BBR j27 → GND rail

[ ] Step 15: BBR j26 → GND rail

[ ] Step 16: BBR j25 → GND rail

[ ] Step 17: BBR j24 → GND rail

[ ] Step 18: BBR j23 → GND rail

[ ] Step 19: BBR j22 → GND rail

[ ] Step 20: BBR j21 → GND rail

Each resistor completes one LED current path.

8.13 Push button wiring

Placement check

Confirm that the push button legs are inserted into:

[ ] BBL a2

[ ] BBL a4

[ ] BBL d2

[ ] BBL d4

Wiring steps

[ ] Step 21: BBL e2 → BBL GND rail

[ ] Step 22: BBL e4 → BBL g13 (GPIO13)

The button wiring is now complete.

8.14 ESP32 to GND — power wiring

The ESP32 ground pins must be connected to the breadboard ground rails.

[ ] Step 23: BBL f14 → BBL GND rail

[ ] Step 24: BBR e14 → BBR GND rail

This ensures a common reference across both boards.

8.15 LCD backpack wiring

Signal breakout preparation

[ ] Step 25: BBR e9 (GPIO17) → BBR h9

[ ] Step 26: BBR e10 (GPIO16) → BBR h10

Signal wiring

[ ] Step 27: LCD SCL → BBR j9

[ ] Step 28: LCD SDA → BBR j10

Power wiring

[ ] Step 29: LCD VCC → BBR j2

[ ] Step 30: LCD GND → BBR j1

8.16 Decoupling capacitor installation

[ ] Step 31: Capacitor leg 1 → BBR i1

[ ] Step 32: Capacitor leg 2 → BBR i2

The capacitor provides local decoupling for the LCD power path.

8.17 Completion state

This completes the wiring stage of the build.

All wiring now matches the schematic and the physical component layout shown in Figure 6.

If components have not yet been placed, do so now.

The fully wired reference breadboard represents the authoritative physical implementation of Spectrum One.

It demonstrates:

- all pin assignments
- all signal paths
- all functional groupings

The PCB version preserves this design and removes variability introduced by breadboard construction.

9 Bill of Materials (BoM)

This chapter lists the physical components used in the Spectrum One reference breadboard build and its corresponding PCB implementation.

The BoM reflects the exact hardware used during development and validation. It is intended as a reference, not a catalogue of alternatives.

9.1 ESP32 Development Board (1x)

- Type: ESP32 Dev Kit V1
- Module: ESP32-WROOM
- Form factor: 30-pin
- USB interface: USB-C

9.2 LCD 1602 with I2C Backpack (1x)

- LCD module type: LCD 1602
- Controller: HD44780 compatible
- Display format: 16 characters × 2 lines
- Native interface: parallel (HD44780)

I2C backpack:

- Controller: PCF8574
- Interface: I2C
- Pinout: GND, VCC, SDA, SCL
- Default address: 0x27
- Address selection: configurable via A0, A1, A2 jumpers

9.3 10-Segment LED Bar Graph Display (1x)

- Part type: LED bar graph

- Model: HSN-2510SR
- Colour: red
- Segment count: 10 discrete LEDs
- Pin count: 20 pins (10 per side)
- Electrical configuration: no common anode, no common cathode
- Each LED electrically independent
- Package style: single inline bar graph

9.4 Resistor (10x)

- Resistance: 220 ohm
- Tolerance: 5 percent
- Power rating: 0.25 W
- Type: through-hole axial

9.5 Capacitor (1x)

- Capacitance: 100 nF
- Type: ceramic
- Package: through-hole
- Purpose: local supply decoupling

9.6 Push Button (1x)

- Type: momentary tactile switch
- Configuration: SPST, normally open
- Package: through-hole, 6 × 6 mm

9.7 Jumper Wire (4x)

- Type: Dupont jumper wire
- Configuration: female to male
- Pitch: 2.54 mm
- Use: LCD I2C backpack to breadboard connection

9.8 Hookup Wire

- Type: solid core copper wire
- Gauge: approximately 22 AWG
- Insulation: PVC
- Use: general breadboard interconnections

9.9 Solderless Breadboard (2x)

- Size: half size
- Tie points: 400 points
- Power rails: two 50-point rails per board
- Contact pitch: 2.54 mm
- Use: prototyping platform

9.10 Optional mounting base

- Material: rigid insulating sheet
- Examples: acrylic, ABS, or similar
- Approximate size: 100 × 85 mm
- Note: non-conductive materials are preferred to avoid antenna interaction

9.11 LCD I2C address note

The reference build uses an LCD1602 with a PCF8574 I2C backpack configured at address `0x27`.

The firmware is written and validated for this address.

Some visually identical LCD backpacks use address `0x3F` instead. If the display remains blank, change `LCD_ADDR` in `lcd1602_v2.c` from `0x27` to `0x3F` and rebuild the firmware.

Further details are covered in the firmware section.

10 Closing the Breadboard Build

This chapter closes the physical breadboard phase of the Spectrum One project.

At this point, the circuit is complete, functional, and verified. The purpose of this chapter is not to repeat wiring steps or restate the schematic, but to document what was learned during physical construction and to define the boundary between hardware execution and firmware analysis.

Breadboards are temporary by design. They exist to support exploration, validation, and iteration. The value of this phase lies not in visual neatness, but in understanding which behaviours originate in the physical layer and which do not.

10.1 Two Valid Builds, One Electrical System

Figure 10 shows the first phase of rapid development.

This build prioritises speed over stability. Pre-made jumper wires are used extensively, wire lengths vary, and routing is opportunistic. The goal at this stage is immediate feedback: confirming pin mappings, validating peripheral behaviour, and observing system-level interactions as early as possible.

This approach is intentional and effective. It allows design mistakes to surface quickly and reduces the cost of change while the system is still fluid.

However, this wiring style has clear physical limitations.

Most pre-made jumper wires are typically 24 AWG stranded wire. The smaller gauge, combined with stranded construction and longer unsupported length, makes these jumpers mechanically flexible. When inserted into a breadboard tie point, this flexibility allows slight movement at the contact interface.

That movement can cause intermittent micro-disconnections or brief resistance spikes, especially on signals that are sensitive to timing or voltage stability. The wire may appear fully inserted, yet the electrical contact is not consistently stable.

When this occurs on non-critical digital lines, the effect may go unnoticed. When it occurs on power rails or I2C lines, the consequences become visible.

A common symptom during rapid builds is LCD instability.

Characters may appear incomplete, incorrect symbols may be rendered, or the display may show partially updated lines. This behaviour is not caused by a faulty display module and not by incorrect firmware logic.

The cause is connection instability.

Typical failure modes include:

- brief voltage drops on the LCD supply
- ground reference instability between the MCU and display
- momentary contact instability on SDA or SCL causing incomplete or missed transitions
- unstable pull-up paths affecting I2C rise times
- fast transient voltage spikes on signal lines exceeding logic thresholds
- interrupted command sequences during display updates

I2C failures on breadboards are typically caused by bit-level timing and edge errors rather than complete byte loss, leading to corrupted commands, missed acknowledgements, or unintended START and STOP conditions.

HD44780-compatible LCD controllers do not tolerate unstable power or incomplete transactions well. If a command stream is disrupted or the supply dips during an update, the controller may enter an undefined state. The display then shows corrupted output until it is reinitialised or power-cycled.

This behaviour is expected under marginal electrical conditions. It is not a defect and does not indicate an error in the schematic.

Figure 9 shows the same circuit rebuilt deliberately.

Solid-core wire is used throughout, primarily 22 AWG. The larger gauge and rigid conductor maintain firm pressure inside the breadboard tie points. Connections are mechanically more stable, routing is controlled, and supply paths are consistent. The electrical design is unchanged. Only the physical execution differs.

With stable connections, the LCD behaves correctly and repeatably.

Both builds implement the same schematic. Both are electrically correct. Only one provides sufficient mechanical and electrical margin to serve as a long-term reference.

The purpose of showing both builds is not comparison or judgement. It is to make the transition between exploratory construction and reference construction explicit.

10.2 Connection Integrity Is Part of the Design

Breadboard failures are often misattributed to firmware bugs or component quality. In many cases, the underlying cause is neither.

Connection integrity is a first-order concern in mixed-signal and bus-based systems. I2C, in particular, depends on clean edges, predictable pull-ups, and uninterrupted transactions. Breadboards provide convenience, not guarantees.

A useful diagnostic rule applies here:

If a display recovers after a power cycle, the problem is almost always electrical, not logical.

Recognising this boundary prevents wasted debugging effort and reinforces correct fault isolation.

10.3 Mechanical Stability Progression

The stability of system connections can be progressively improved through mechanical construction choices, ranked from least to most stable:

1. Pre-made jumper wire connections on a breadboard These are the least stable due to thin 24 AWG stranded conductors, flexible insulation, and poor mechanical retention in the tie points.
2. Solid-core wire on a breadboard Thicker 22 AWG solid-core wire improves contact pressure and reduces movement within the tie point, resulting in fewer intermittent faults.
3. PCB design with board-mounted connectors A PCB provides fixed geometry and strain relief. Connectors such as JST headers reduce mechanical stress on signal pins and improve long-term reliability.
4. Directly soldered connections Permanent soldered joints offer the highest mechanical and electrical stability and eliminate contact variability entirely.

Each step reduces uncertainty by removing degrees of mechanical freedom from the system.

10.4 There Is No Single Correct Layout

Breadboards do not impose a single physical solution.

The same electrical connections can be routed in many valid ways depending on component placement, wire length, and personal workflow. Visual symmetry is irrelevant. Electrical correctness and mechanical reliability are not.

The reference layout shown in the figures is one working solution. It is not presented as optimal or canonical.

What makes it a reference is that:

- every connection is deliberate
- every wire has a defined role
- the build can be reproduced consistently
- the system behaves the same after rebuilds

These properties matter more than appearance.

10.5 End of the Physical Phase

With the reference breadboard complete, the physical phase of Spectrum One is finished.

From this point onward:

- the schematic remains unchanged
- the wiring is treated as fixed
- observed behaviour is analysed at the firmware level

Any instability observed beyond this point should be investigated in software, configuration, or interpretation, not in physical construction.

The breadboard build now serves as the authoritative physical reference from which the PCB version is derived.

What the PCB will change is not the design, but the certainty.

11 Firmware Binaries

11.1 What These Binaries Are

The firmware binaries are direct build outputs produced from the same source code published with Spectrum One.

Each file represents a specific firmware state compiled and tested on the reference hardware. These files are snapshots of working firmware captured at a defined moment in the project. They target the ESP32 microcontroller and the Spectrum One device itself.

They exist as firmware images rather than user-facing software. They run on the device and have meaning only within the physical and electrical context of Spectrum One.

Everything contained in these binaries already exists in readable form in the source code. The binaries add nothing on top of that. They are simply the compiled result of the same logic.

The published binary package contains only the following:

The published binary package contains only the following:

```
spectrum_one_firmware_v0.1.0/
  README.txt
  spectrum_one_v0.1.0.bin
  spectrum_one_v0.1.0.elf
  spectrum_one_v0.1.0.map
```

- The `.bin` file is the flashable firmware image.
- The `.elf` file preserves symbols and build context.
- The `.map` file records the linker memory layout.

Together, these files provide a fixed reference point for a known working firmware build.

11.2 Why They Exist

The binaries are provided for anyone who finds them useful.

Some people use them to compare against their own builds. Some use them to confirm that hardware behaves as expected. Some ignore them completely and work only from source.

All of these choices are valid.

The project does not require the binaries, depend on them, or privilege them over the source. They are published artifacts from a working build, nothing more.

11.3 What They Are Not

The binaries are not instructions.

They are not a simplified path. They are not a supported shortcut. They are not tailored to any operating system, flashing tool, or workflow.

No assumptions are made about how or whether they are used.

11.4 Flashing Is Outside the Book

Flashing firmware is an external process.

It depends on the host system, the tools chosen, the connection method, power conditions, and timing. These variables change constantly and differ from system to system.

This book does not attempt to standardise or explain that process.

If you choose to flash a binary, you already know how you intend to do it. If you do not, the source code path remains available.

11.5 Responsibility Boundary

Firmware flashing always carries risk.

Once a binary leaves the repository, what happens next depends entirely on the environment it is introduced into. That environment is not knowable or controllable from here.

The project provides firmware source and build outputs. What is done with them is a user decision.

11.6 Relationship to the Source Code

The binaries do not define Spectrum One.

The source code does.

Every behaviour, decision, and interaction in the firmware exists first in the source. The binaries are only compiled expressions of that work.

The next chapters move into the firmware itself. That is where the logic lives, where changes are made, and where understanding actually begins.

11.7 Architecture Overview

The firmware is intentionally simple.

There is:

- one main execution loop
- one button handling task
- one WiFi scan engine
- a small, explicit UI state machine

There are no interrupts driving logic. There are no hidden background timers. There are no asynchronous side effects.

This simplicity is not naïve. It is defensive.

Complex systems hide cause and effect. Spectrum One refuses to do that.

11.8 Firmware Source Tree

The repository structure mirrors the firmware’s intent.

```
spectrum_one/
  CMakeLists.txt
  sdkconfig
  main/
    CMakeLists.txt
    spectrum_one.c
    wifi_scan.c
    wifi_scan.h
    wifi_one_math.c
    wifi_one_math.h
    led_bar.c
    led_bar.h
    lcd1602.c
    lcd1602.h
    config.h
```

There are no generic helpers. No utils directory. No abstraction layers added “just in case”.

12 Firmware as an RF Observer

Spectrum One is a hardware project, yet the firmware is the part that turns it into an instrument.

Without firmware, the ESP32 sits there as silicon and antenna. With firmware, it becomes a disciplined observer of the 2.4 GHz environment. It scans, collects, reduces, maps, and renders a human-readable account of radio behaviour that is normally reshaped or obscured by operating system policy layers and interface abstractions.

The firmware collects only broadcast WiFi scan metadata returned during active scans: SSID, BSSID, RSSI, channel, and security flags. This data exists transiently in RAM, is processed locally for display, and is overwritten on the next scan or lost on power-off. There is without payload capture, without credentials, without logging, without storage, without transmission, and without external processing.

The RF front end and baseband inside the ESP32 remain vendor implementations. That boundary is fixed and unavoidable. Spectrum One makes no claim beyond it. What the firmware controls is everything above that boundary: scan timing, result handling, aggregation, interpretation, and rendering.

Interpretation does exist in the firmware, and it is intentional. It is explicit, documented, and inspectable. The maths layer converts reported RSSI values into derived indicators so they can be rendered meaningfully on a small display. These transformations are heuristics, not measurements. They claim no physical authority and no calibration. They exist to expose trends, relationships, and change, not to assert truth.

This distinction matters when comparing Spectrum One to smartphone applications. Phones and ESP32 devices both rely on vendor radios. The difference is that smartphones place an operating system policy layer above the radio. Scan timing can be throttled or deferred, results may be cached or filtered, background behaviour is constrained, and identifiers may be suppressed. An application does not own the scan loop, and it cannot fully observe or control the transformation chain that shapes what it displays.

Spectrum One does. The firmware controls when scans occur, how results are handled in memory, how values are derived, and how they are rendered. No external policy layer reshapes the output. All interpretation is visible in code and can be modified, removed, or replaced by the observer.

This chapter explains what that firmware is doing, why it is built that way, and why the output is intentionally unsettling. It also explains why Spectrum One can reveal behaviours that many smartphone tools cannot, even when those tools appear more polished.

The goal is to expose the real behaviour of WiFi by observing it end-to-end, with all interpretation made explicit and inspectable.

12.1 What the firmware is trying to prove

Most WiFi explanations start with theory and end with a speed test. That path builds confidence first, then disappointment.

Spectrum One takes a different approach.

It starts by exposing instability and asks you to treat that instability as the primary signal. The firmware is designed to show:

- WiFi scans are snapshots, not a live feed
- RSSI is a local report, not a distance ruler
- strongest is a temporary winner, not a fixed truth
- small timing changes produce large visible changes

- smoothing can be useful, and it can also hide the story the device is showing

Under the hood, the firmware is a chain of choices. It is not physics. It is not calibrated metrology. It is a set of repeatable transformations that turn scan responses into a display and a moving LED bar.

That chain matters.

12.2 Observation versus participation

Spectrum One observes the environment while staying out of the role people associate with WiFi devices.

It does not authenticate to an access point, negotiate security, request an IP, or move user traffic. It runs in station mode because that is the ESP32 scanning model, yet it behaves as an observer rather than a network participant. In `wifi_scan_init()` the firmware sets WiFi mode to station and starts WiFi, then scans in a loop.

That separation matters because participation changes the system you are trying to observe. Once you join a network, your device becomes part of the airtime competition. It transmits, it retries, it adapts data rates, it sleeps and wakes, it influences roaming, and it makes the environment respond.

An observer still influences the environment a little, because active scanning transmits probe requests. The firmware accepts that reality instead of pretending it can watch without touching. The important point is intent and scope: the firmware stays away from user traffic and stays away from the make-the-connection-work mindset.

12.3 Local processing as a trust boundary

Everything that matters happens on the ESP32.

Scan results arrive, the firmware computes a small set of derived values, then it renders them on the LCD and LED bar.

The software operates without an export path, without background services, without accounts, and without any remote database. The behaviour remains visible, traceable, and reproducible on the bench.

That boundary is practical. It also keeps feedback tight. When you change a constant, you see the result immediately, on hardware, in the same room, under the same RF conditions.

12.4 Scanning is a snapshot, and the snapshot window matters

A WiFi scan is an active discovery process. A station sends probe requests and waits for responses, then moves to the next channel and repeats. The result is a set of responses received inside a timing window, not an authoritative catalogue of everything that exists.

That timing window is where people get stuck.

Two scans taken seconds apart can produce different lists and different RSSI winners, even when neither you nor the router has moved. That behaviour is normal in RF. It also gets amplified by:

- multipath reflections within the surrounding environment
- interference from other 2.4 GHz systems
- airtime contention and backoff
- AP response behaviour under load
- scan dwell time choices per channel
- dynamic transmit power and rate adaptation
- client-side filtering, caching, and rate limiting

Multipath alone is enough to break the RSSI equals distance myth. Small changes in path length can push the received signal into constructive or destructive interference at 2.4 GHz, producing rapid apparent swings.

So the firmware treats each scan as an independent snapshot, then builds only the smallest amount of continuity on top, and only where continuity supports a stable interface.

12.5 Why 2.4 GHz exposes behaviour

The 2.4 GHz band is crowded, reflective, and inherently messy. Signals at this frequency attenuate less through common building materials than higher bands, allowing them to travel further within typical homes and overlap more easily. The band is shared by many unrelated wireless systems and is densely populated with overlapping networks operating in close proximity.

That behaviour makes it ideal for observation. It refuses to behave like a controlled laboratory environment and instead exposes interference, contention, and variability as normal operating conditions.

Starting with a cleaner band would hide many of these effects behind better propagation and lower congestion. Spectrum One is designed to reveal the mess, because the mess is where understanding starts.

12.6 Active scan choice and what it implies

The firmware uses active scans. In the ESP-IDF scan configuration, the scan type is set to active and hidden networks are included in the scan result list.

Active scanning means probe requests are transmitted and APs respond with probe responses. That tends to discover more networks than passive listening in a short scan, especially in environments where beacon capture across all channels would take too long.

It also means your scan is part of the environment.

That is not a flaw. It is an honest tradeoff. On a small embedded device, active scanning provides repeatable behaviour and usable datasets. The firmware accepts the resulting variability and exposes it directly rather than hiding it.

12.7 Firmware architecture that stays readable

The reference firmware is intentionally small and direct. The source tree reflects that: a main application file, a scan module, a maths module, an LCD driver, and an LED bar driver.

This is a deliberate architectural statement.

The firmware is not trying to be clever. It is trying to be inspectable. When cause and effect must stay visible, every extra layer becomes a liability.

The core modules are:

- `spectrum_one.c` Application control, UI state, scan loop, and input handling.
- `wifi_scan.c` and `wifi_scan.h` WiFi scan initiation, result capture, strongest selection, and total power estimate.
- `wifi_one_math.c` and `wifi_one_math.h` Conversions and mappings from scan values to UI level values.
- `lcd1602.c` and `lcd1602.h` A driver for a 16×2 HD44780-compatible LCD connected through a PCF8574 I2C backpack, matched to the reference build.

  This layer deserves special attention. LCD backpacks that appear identical externally often differ internally. Address selection, backlight polarity, transistor inversion, and pin mapping can vary between manufacturers. Swapping to a visually identical LCD module can introduce subtle incompatibilities that present as firmware faults.

  In the reference build, the driver reflects the specific backpack wiring used. When a different backpack is fitted, symptoms such as inverted backlight control, incorrect addressing, or non-responsive displays can occur until the driver assumptions are aligned with the actual hardware.

- `led_bar.c` and `led_bar.h` Ten GPIO outputs driving a ten-segment LED bar used as a visual trend indicator.

  This module is intentionally hardware-only. It contains no WiFi logic and performs no calculations. It accepts a single integer level from 0 to 10 and reflects that level directly on the GPIO pins according to the reference wiring.

  All interpretation happens upstream. LED levels are produced by mapping functions in `wifi_one_math.c`, which convert WiFi-derived values into banded display levels. These mappings include both aggregate activity estimates and individual access point RSSI mappings.

  For example, individual access point signal strength is mapped to LED levels using a bounded scale defined in the maths layer. The LED bar therefore visualises interpreted behaviour rather than raw signal values.

12.8 The UI as a state machine

A menu-driven interface changes behaviour by moving through a list. The same button press can mean different things depending on where you are in the menu structure.

Spectrum One does not navigate a menu. It switches between a small number of fixed UI states. Each state defines its own valid inputs and outcomes. Screen changes occur only when a specific event triggers a specific transition.

The firmware defines a small, fixed set of screen states and well-defined transitions between them. In the reference code, `ui_mode` represents primary screens such as Strongest, SUM, and Field, with additional overlay states for browsing and follow mode.

This approach prevents accidental behaviour. Every screen change is the result of a specific input event. Every timeout is the result of a deliberate logic path. Nothing changes implicitly.

As a result, behaviour remains predictable. Cause and effect stay visible in the code, and changes to UI logic remain local and traceable.

Field screen as the home state

The Field screen acts as the default state. It presents a computed field-level value and provides an immediate sense of overall RF activity. The LCD formatting applies unit scaling to keep the value readable across a wide range.

Returning to a consistent home state simplifies interaction. From there, other views can be entered intentionally and exited cleanly, without leaving residual UI state behind.

Strongest screen and browse overlay

The strongest screen answers a simple question: which named network is strongest in the current scan.

Browse mode extends this by allowing you to step through individual access point records and inspect RSSI values per entry. For this reason, `wifi_scan_result_t` stores a fixed maximum number of access point records per scan, keeping memory use bounded and predictable.

Browse mode also supports an optional timeout, controlled through compile-time options. This allows the interface to either return automatically to the home state or remain in browse mode until explicitly exited, depending on the chosen configuration.

SUM screen

SUM tries to show how much WiFi power appears present as a single number. In the reference firmware, the scan result carries both `total_mw` and `total_dbm`.

This screen has two consequences:

- aggregation can be useful
- aggregation can invite false confidence

A single number looks authoritative. The firmware treats it as a compact indicator, not a claim of truth.

12.9 One button input and why polling beats interrupts here

A single button forces clarity. It reduces the UI to events: short click, long press, and context.

The firmware implements button reading via a dedicated FreeRTOS task that polls at a defined interval, tracks stable state, applies debounce, and detects long press timing. The constants make that explicit: polling period, debounce ticks, and long press duration.

Polling is a design choice with consequences:

- timing is visible in the code
- behaviour is easier to reason about
- the button is aligned to human time, not edge noise

Interrupt-driven button logic looks elegant until contact bounce turns one press into many. Polling with a stability threshold makes intent the unit of input.

12.10 WiFi scan pipeline in the reference firmware

The scan pipeline has four stages:

1. initialise WiFi and system services
2. request a scan
3. read scan records

4. compute derived values and update the UI

Initialisation

In `wifi_scan_init()` the firmware initialises NVS (Non-Volatile Storage), network interfaces, event loop, creates the station interface, initialises WiFi, sets station mode, starts, and reads the station MAC.

That is deliberate. It reduces hidden setup steps and keeps the scan loop free of one-time complexity.

Scan request and record capture

In `wifi_scan_run()` the firmware clears the output struct, sets safe defaults, constructs the scan configuration, applies scan time config, then starts a blocking scan. It then reads the number of APs found, caps it to a maximum, and retrieves the AP records into a buffer.

Two design points matter here:

- empty scans are treated as valid snapshots. That prevents a false idea that scans are guaranteed to find something.
- record count is capped. That prevents memory chaos and keeps behaviour stable across environments.

Strongest selection and the stability problem

The strongest access point is selected from the current scan list.

In the default model, the strongest AP is simply the record with the highest RSSI in that scan snapshot. This selection can change frequently when multiple APs report similar signal levels, resulting in visible flicker.

An optional alternative model is implemented in the firmware. This model applies hysteresis and persistence by maintaining a locked winner and requiring a challenger to exceed it by a defined margin for a defined number of consecutive scans before a switch occurs. The logic is documented in code comments and implemented using a challenger counter and configurable thresholds.

The two approaches represent different tradeoffs. Raw selection reacts immediately to changes but can fluctuate rapidly. Smoothed selection reduces visible switching at the cost of delayed response and reduced sensitivity to short-term variation. The behaviour is selected at compile time.

12.11 The maths layer and why it is explicitly heuristic

The maths module exists because the LCD and LED bar need values.

It contains:

- conversions from dBm to mW
- a total power style estimate
- a derived field value expressed as nW/m^2
- mappings from those values to a 0 to 10 LED bar level

The header comments matter. The firmware describes the field conversion as a heuristic derived from scan RSSI and states that it is without calibration.

That honesty is essential.

RSSI values are already a product of vendor decisions. They represent received signal strength at the radio with device-specific scaling. Turning that into a field-style value is inherently interpretive. The firmware does it anyway because it provides a stable visual language for trend and density.

Summing power and why logarithms bite

RSSI is expressed in dBm, a logarithmic unit. Summing dBm values directly is meaningless. If you want a combined power-like figure, you convert each dBm to mW, sum the mW values, then convert back to dBm.

The firmware follows that style by tracking `total_mw` and converting to `total_dbm` through log10.

This produces a number that reacts to both the count of responders and their reported strengths. It remains an estimate because:

- RSSI includes noise and receiver variability
- AP responses are not guaranteed each scan
- scan timing changes which responses arrive

That is why the firmware treats it as a comparative indicator rather than a claim of exposure.

LED bar mapping as perception, not measurement

A 10 segment LED bar is a human perception device. It is better at showing motion than showing absolute truth.

The firmware maps values into bands, clamps them, and drives the bar level. The mapping functions use min and max ranges for normalisation and return banded levels.

This accomplishes three things:

- quick situational awareness
- trend visibility
- a built-in refusal to pretend precision

The bar is a language. It is a compression of complex behaviour into something your peripheral vision can read.

12.12 LCD driver reality and the backpack trap

The LCD in the reference build is an HD44780 compatible 16×2 panel with a PCF8574 I2C backpack. The firmware driver is written to match that wiring, including the bit mapping of RS, RW, EN, and backlight control.

This is where many builds fail.

The controller standard is stable. The backpack implementations vary widely. Backlight polarity, transistor inversion, address strapping, and pin mapping differ between suppliers. The resulting symptoms often resemble firmware faults: blank screens, block characters, random glyphs, or backlight control that operates with inverted logic.

The key insight is that the backpack is an adapter layer with inconsistent conventions. The firmware is correct for the reference backpack. If your backpack differs, the driver must be adapted.

This is why the chapter treats LCD issues as supply chain and assumptions, not as mysterious software bugs.

12.13 Follow mode and identity in WiFi

Follow mode tracks a selected network using an identifier that does not change between scans.

A practical use case is locking onto a known network, such as a home access point, and moving around a property while observing how its reported signal strength and visibility change across locations.

In the reference firmware, the selected SSID is stored in `follow_ssid` and used to locate the corresponding access point record in subsequent scans. If the SSID does not appear in a given scan snapshot, that absence is reported explicitly rather than being replaced by another network.

This is a deliberate design choice about identity.

An alternative approach would be to track a specific BSSID. While more granular, BSSID identity can change in real deployments due to multiple radios, extenders, roaming infrastructure, or modern privacy-related behaviour. As a result, BSSID-based tracking can produce discontinuous or misleading identity shifts.

Follow mode therefore prioritises SSID-based identity, favouring continuity and interpretability over strict per-radio precision.

12.14 Why Spectrum One can beat smartphone apps at truth telling

Smartphones are powerful devices with capable radios. That does not mean their WiFi scan results are neutral or complete.

On a phone, WiFi scanning is mediated by the operating system. Scan timing can be throttled or deferred. Results can be filtered, cached, merged, or delayed. Background scanning can be heavily constrained. Power management and privacy rules can limit what identifiers are exposed to apps. What an app receives is often a policy-shaped view of WiFi, not a direct representation of what the radio could observe in real time.

As a result, a smartphone WiFi app often shows what the operating system allows it to see at that moment, not the full set of conditions present on the air.

Spectrum One operates differently. It controls the entire chain on the device: when scans occur, how results are handled in memory, how values are derived, and how they are rendered. The display is driven by the scan that just completed, without reliance on an external OS policy layer deciding scan cadence, background behaviour, or the shape of the results.

Scan frequency throttling on Android

On Android, WiFi scanning through public APIs is throttled. Foreground apps are limited in how frequently they can request scans, and background scanning is constrained even further across all background apps. When an app cannot scan at the cadence a user expects, the interface tends to drift toward cached or stale results, partial updates, or smoothing that hides gaps.

Spectrum One runs on a device built for one job. It scans in a fixed loop with a fixed delay, so the snapshot cadence remains consistent and fully under firmware control.

General-purpose WiFi scanning is restricted on iOS

iOS does not provide open, general-purpose WiFi scanning to third-party apps in the way many people expect. WiFi-related APIs are gated and targeted at specific use cases. This limits what typical App Store apps can access and how directly they can present nearby network information.

This is why an iPhone app can disagree with Spectrum One without either device being faulty. The app is operating under a different set of rules and constraints.

Phones optimise for battery, privacy, and user experience

A phone is a general-purpose device designed to preserve battery life, protect privacy, and keep interaction smooth. WiFi scanning is power hungry, and nearby network information can be used for location inference. Platform policies therefore constrain scanning behaviour and the information exposed to apps.

Spectrum One makes a different set of tradeoffs. It is dedicated hardware built to observe and display WiFi behaviour, with scanning cadence and result handling kept explicit and local.

12.15 Direct control of scan timing and processing

Spectrum One owns the entire scan-to-display chain, and that ownership is explicit in code.

When scans happen

Scanning is driven directly by the main loop in `spectrum_one.c`. Each iteration runs a blocking WiFi scan via `wifi_scan_run()` and then delays by a fixed interval. There is no scheduler above it, no background policy, and no adaptive throttling. Scan cadence is therefore predictable and repeatable by design.

How results are stored

Scan results are stored in a bounded result structure that is allocated at compile time. Each scan writes into a fixed-size array with a defined upper limit on the number of access points that can be recorded.

This design is deliberate. By fixing the maximum result count in advance, memory usage remains constant and observable. There is no heap allocation, no resizing, and no hidden growth as the radio environment becomes more congested.

When surrounding WiFi activity increases, the system does not consume additional memory or change its allocation behaviour. Excess results are ignored once the limit is reached, preserving predictable timing and stable operation.

The concrete implementation details for this behaviour can be found in the firmware source:

- WIFI_SCAN_MAX_APS
  Defined in wifi_scan.h. This constant sets the maximum number of access points stored per scan.
- wifi_scan_result_t
  Defined in wifi_scan.h. This structure holds the bounded scan results.
- WiFi scan logic
  Implemented in the WiFi scan module, where scan results are copied into the fixed array and excess entries are discarded.

What is aggregated

All interpretation happens in the maths layer, implemented in `wifi_one_math.c`. This includes:

- converting RSSI (dBm) to linear power (mW)
- summing total received power across all visible APs
- deriving a field-style density estimate in nW/m²
- mapping derived values into discrete 0–10 LED levels

These mappings are explicit, bounded, and documented in code comments. The LED bar never sees raw RSSI. It visualises interpreted values only.

What is rendered

Rendering is handled by hardware drivers that deliberately contain no WiFi knowledge:

- `lcd1602.c` moves characters to a 16×2 HD44780 display
- `led_bar.c` sets GPIO levels for a ten-segment LED bar

Both drivers accept already computed values. They do not smooth, filter, or interpret data. If a value changes on screen, it changed upstream.

12.16 Behaviour switches exposed in firmware

Several behavioural choices are surfaced directly in code as compile-time options:

- strongest AP selection In `wifi_scan.c`, the strongest AP can be selected in two ways:
  – raw strongest per scan (default)
  – optional smoothed winner using hysteresis and persistence (`OPT_STRONGEST_SMOOTHED_WINNER`, `SWITCH_MARGIN_DB`, `REQUIRED_WINS`) The difference between flicker and stability is therefore explicit. It is a single switch with visible consequences.
- scan dwell timing Active scan timing is selectable

(`OPT_SCAN_TIME_ALT`). The comments document how different dwell values change observed behaviour and can introduce or suppress extreme spikes.

- UI timeout behaviour In `spectrum_one.c`, idle timeouts, browse timeouts, and screen-specific timeouts are independently controlled with compile-time flags. Timeout behaviour is not implicit. It is spelled out and can be removed or enforced deliberately.

By contrast, a smartphone app cannot reliably own these stages:

- scan scheduling may be altered or deferred by the operating system
- results may be cached, merged, or filtered before the app receives them
- background execution limits can create long gaps between fresh scans
- privacy and location policies can suppress identifiers entirely

This is why Spectrum One can produce a more honest output despite its small display.

Honesty comes from control of the chain, not from the size of the UI.

12.17 What people commonly misread, and what the firmware is showing instead

Mistake 1: expecting stability

People expect that if they stand still, the strongest network and RSSI should stay fixed.

The firmware contradicts that expectation because the environment is not static. Packets collide, responses arrive late, reflections shift with tiny motion, and scan snapshots include different responders.

Flicker is information.

If the strongest winner flips between two SSIDs at similar RSSI, that often indicates a threshold boundary in a reflective environment.

Mistake 2: treating RSSI as distance

RSSI is a report of received signal strength, not a tape measure. Multipath fading and interference can dominate, especially indoors at 2.4 GHz.

Spectrum One reinforces this by showing that the winner can change without movement.

When you see that happen repeatedly, the distance assumption stops feeling reliable.

Mistake 3: trusting single numbers too much

The SUM figure and the Field figure are useful, yet they are compressions.

The firmware is clear that the field-style value is heuristic rather than calibrated measurement.

So the correct use is comparison:

- compare the same room at different times
- compare different rooms in the same building
- compare with and without specific devices running
- compare before and after channel changes on a router

The output is a lens for relative change.

Mistake 4: thinking a scan is a list of everything

A scan is an interaction. It depends on timing and responses. Probe request and response behaviour is part of the process, and each scan includes only what answered in that window.

Spectrum One treats empty scans as valid and treats changing lists as normal.

Discovery is probabilistic.

12.18 Practical experiments that expose behaviour quickly

These are firmware-supported experiments that reliably reveal how WiFi behaviour changes under controlled conditions.

Experiment 1: vary the scan cadence

Change the scan delay in the main loop and observe:

- how strongest-AP stability changes
- how the number of APs reported per scan changes
- how SUM and Field values respond at different cadences

This demonstrates that scan timing directly influences what appears visible. The environment does not change, but the snapshot you take of it does.

Experiment 2: enable strongest smoothing and observe the tradeoff

Enable the strongest smoothing option and compare it with raw strongest selection.

The smoothed model requires a challenger to exceed the current winner by a defined margin for a defined number of consecutive scans before a switch occurs.

Observe:

- the strongest label remains stable for longer
- genuine changes take longer to appear
- the device output appears calmer
- the RF environment itself remains unchanged

This makes the cost of smoothing explicit. Stability is achieved by delaying or suppressing change, not by improving accuracy.

Experiment 3: browse order versus identity

Use browse mode to step through access point records and observe how record order changes between scans. Then lock onto an SSID using follow mode and observe its behaviour across snapshots.

This exposes the difference between list position and identity. Scan result order is transient. Identity must be resolved explicitly rather than inferred from index position.

Experiment 4: change activity, not topology

Turn off a high-traffic device, such as a streaming client, and observe SUM and Field behaviour.

In many cases, the list of visible access points changes little, while aggregate values shift noticeably. This separates the idea of what exists in the environment from what is actively contributing to RF activity.

12.19 Limits that keep the project honest

Spectrum One is effective because its limits are explicit.

- it is without calibration
- it is without exposure measurement claims
- it is without health inference claims
- it is intended for comparative observation within a local environment

These constraints prevent false authority. The firmware reflects this by treating all derived values as heuristics and by keeping every transformation step visible in code.

12.20 This firmware does one thing

It runs WiFi scans, accepts whatever responses arrive in that window, applies explicit transformations defined in code, and displays the result. Nothing is stabilised for comfort. Nothing is smoothed unless you enable it. Nothing is hidden behind policy or background behaviour.

Every value shown is the direct outcome of:

- the RF environment at that moment
- the scan timing chosen
- the transformations applied in the firmware

If the output changes, something in that chain changed.

The firmware does not claim correctness, calibration, or completeness. It exposes process. It makes the transformation from scan response to displayed value visible and inspectable.

13 Printed Circuit Board

13.1 From Breadboard to PCB

The PCB version of Spectrum One is a direct continuation of the reference breadboard build.

Nothing essential changes:

- Pin assignments remain the same
- System logic is unchanged
- Firmware behaviour and codebase are unchanged

This is not a redesign. It is the stabilisation of a proven reference build.

The breadboard establishes correctness and intent. The PCB preserves that intent while removing the variability inherent in temporary wiring.

In practical terms, the PCB improves:

- Mechanical stability
- Build repeatability
- Ease of assembly, inspection, and repair
- Safe handling and transport
- Confidence during through-hole soldering
- Structural clarity through a simple two-layer layout

13.2 PCB Design Philosophy

The PCB is designed to prioritise learning, clarity, and longevity.

Its goals are intentional and extensible:

- readable layout through clear spacing and grouping
- repairability without specialised tools
- reproducible fabrication using standard processes

- safe modification without disturbing the core system
- extension paths that allow future expansion without redesign

There are no RF optimisation techniques, impedance-controlled traces, buried layers, or density-driven compromises.

One deliberate difference from the breadboard build is the implementation of a copper-free keep-out zone beneath the ESP32 antenna, following Espressif guidance. This constraint is practical on a PCB and prevents avoidable detuning that cannot be controlled on a breadboard.

The keep-out region is not an optimisation. It is a visible, verifiable constraint that avoids introducing unnecessary RF variables.

This board favours understanding over compactness. Routing and spacing decisions support visual inspection, manual soldering, and confident modification rather than production optimisation.

The PCB is intended to be understandable by inspection, without requiring prior PCB design experience.

13.3 Socketed ESP32 Module

The ESP32 development board is mounted using sockets rather than being soldered directly.

This allows:

- replacement of the ESP32 without rework
- recovery from damage without risking the PCB
- experimentation with alternative ESP32 modules
- removal without stressing pads or traces

Socketing trades compactness for long-term serviceability. That trade is intentional.

The antenna region beneath the ESP32 module is kept free of copper. No pours, traces, or components intrude into this area. The constraint is visible in the layout and verifiable by inspection.

13.4 LCD Mounting and Connector Choice

The LCD1602 module is connected using short wires terminated with JST-XH connectors.

LCD1602 modules share a name but vary mechanically. I2C backpacks differ in header placement, board thickness, connector offset, and pin mapping between manufacturers.

Using short wired connectors provides:

- compatibility across LCD backpack variants
- stress-free mounting without forced alignment
- straightforward replacement of the LCD module
- mechanical isolation from PCB tolerances

This approach prioritises real-world variability over rigid mechanical assumptions.

13.5 Component Spacing and Soldering Experience

All through-hole components are spaced generously.

This provides:

- comfortable soldering access
- clear visual inspection of joints
- reduced risk of accidental bridging
- straightforward rework and repair

The PCB is designed to feel approachable. It rewards careful work rather than punishing inexperience.

13.6 Physical Layout and Assembly Cues

Figure 12 shows the populated PCB alongside the unpopulated board, allowing direct comparison between component placement and the underlying routing, silkscreen, and keep-out zones.

Several design decisions become immediately apparent in a way that schematics alone cannot convey.

The LCD occupies the dominant visual area of the board, establishing it as the primary interface element. The ESP32 module is offset to preserve antenna clearance while maintaining unobstructed access to the USB port. The LED bar and its current-limiting resistors are grouped as a single functional block, reinforcing their shared role.

The push button is positioned at the board edge for direct access. This placement reduces accidental activation while remaining easy to reach during use.

Assembly guidance is printed directly on the PCB. Silkscreen arrows indicate correct orientation for both the LCD backpack and the ESP32 module, reducing ambiguity during assembly without requiring constant reference to documentation.

The copper keep-out zone beneath the ESP32 antenna is clearly defined in the layout. The region remains free of pours, traces, and components, making the constraint visible and verifiable by inspection.

The unpopulated board view highlights routing simplicity. Traces are widely spaced and easy to follow, with functional areas clearly separated. No routing is hidden beneath dense component clusters.

Taken together, the layout communicates how the board should be assembled, used, and understood. The PCB does not rely on implicit knowledge. Its structure is intended to be readable in the same way the firmware is readable.

13.7 Reproducibility and Repair

The board layout supports:

- manual assembly
- visual fault tracing
- component-level repair

- reproduction using basic tools

No specialised processes or proprietary techniques are required.

Inspection, modification, and repair are treated as normal outcomes rather than edge cases.

13.8 Included Files and Manufacturing

All design files are provided in standard formats.

There is no vendor lock-in. Any standard PCB manufacturer can produce the board.

The design is compatible with low-volume fabrication without special requirements.

13.9 Open Design Files and Ongoing Access

All design files for Spectrum One are provided openly.

Schematics, PCB layout files, and related assets are available through the project repository:

https://github.com/currenari

The PCB is designed in KiCad. Depending on revision state, the repository may include:

- native KiCad schematic and PCB files
- manufacturing outputs such as Gerber files
- supporting documentation related to revisions

Where Gerber files are provided, they allow direct manufacturing while preserving access to the full design intent through source files.

This approach supports learning, transparency, and independent reproduction. Future revisions build on an accessible reference rather than replacing it.

13.10 Assembly Notes

Assembly follows the same logical order as the breadboard build.

Pay particular attention to:

- ESP32 development board orientation
- LED polarity
- LCD backpack orientation
- Button placement

No calibration or tuning is required after assembly.

The device is intended to visualise relative activity and behaviour rather than provide calibrated or absolute measurements.

Power is supplied via the ESP32 development board USB port using a standard 5 V USB supply.

13.11 Closing the Reference Build

By completing this build, you now understand:

- how a breadboard design translates into a stable PCB
- why layout, spacing, and orientation matter in physical hardware
- how design choices affect assembly, repair, and longevity
- how hardware and firmware can be designed to teach rather than obscure

Spectrum One is not an endpoint. It is a stable reference.

13.12 Future Expansion and Design Continuity

The current PCB represents a reference implementation rather than a final form.

Its structure leaves room for future revisions that may expose additional ESP32 GPIOs or external interfaces for experimentation. That work is intentionally deferred.

Version v0.1.0 focuses on clarity, stability, and learning. Expansion is treated as a separate design direction that benefits from a reliable baseline.

Both the PCB layout and the firmware structure are designed with this progression in mind.

14 Final Distribution and Availability

Spectrum One is documented in this book as a complete reference system.

The hardware design, firmware behaviour, and physical layout described here represent the full implementation. There are no reduced editions, feature-limited variants, or alternate builds. What is presented in this book is the system as built and used.

Distribution exists to support direct interaction with the device itself, either through independent construction or by working with a completed reference build.

14.1 Reference Scope

This manuscript documents the definitive Spectrum One reference build.

It describes the hardware, firmware behaviour, assembly intent, and operational characteristics of the finished device. The information is presented as a technical reference, without abstraction, automation, or reliance on external services.

The system described here is intentionally finite and self contained.

14.2 Physical Builds

In addition to the documentation, physical builds may be offered.

All physical distribution is limited in scale and produced in small runs. Availability depends on production capacity and may vary over time. There is no mass production and no permanent stock.

Two physical formats may be made available.

14.3 Fully Assembled Reference Build

A fully assembled Spectrum One unit may be offered as a completed reference instrument.

This format provides a known good build identical to the system described throughout this book. The hardware layout and firmware behaviour match the reference exactly.

The fully assembled reference build is shown in Figures 13 and 14.

This option is intended for those who want a functioning reference system without performing assembly themselves.

14.4 Unassembled Hardware Set DIY Kit

An unassembled hardware set may also be offered for those who want full hands on involvement.

This format is intended for manual assembly, soldering, inspection of the physical design, and learning through direct construction.

The unassembled hardware set includes the custom Spectrum One PCB, all required electronic components, and an ESP32 development board supplied with the reference firmware pre installed.

The unassembled hardware set is shown in Figure 11.

Final assembly, soldering, and testing are performed by the builder at their own risk and responsibility.

14.5 Firmware Status

All ESP32 development boards supplied with physical builds are shipped with the Spectrum One reference firmware already installed.

After assembly and power up, the device is ready for operation without requiring initial flashing, toolchain setup, or external software.

The firmware is not locked. It may be modified, replaced, or repurposed using standard ESP32 tooling. The supplied firmware exists to provide a known working reference build.

14.6 Design Intent

Spectrum One is intentionally offline and self contained.

Once built, the device operates independently. It does not rely on external services, user accounts, mobile applications, operating system updates, or third party platforms. Its behaviour remains stable, observable, and under the control of the person using it.

This concludes the Spectrum One reference build.

14.7 Availability Notes

Information about physical builds or unassembled hardware sets, if offered,
is published at https://currenari.com

Distribution may occur through different channels over time. Availability is limited and may change without notice.

15 Beyond Spectrum One

Spectrum One ends as a complete instrument.

Nothing is missing. Nothing is required. It performs the task it was built for without qualification or dependency.

What follows is not an extension of that task, but a widening of perspective.

This chapter exists to show that the value of Spectrum One does not stop at WiFi, nor does it depend on it.

15.1 The Hardware Is the Instrument

Even with the WiFi firmware removed entirely, the device remains intact as a physical system.

- An ESP32 microcontroller
- A sixteen by two character display
- A ten segment LED bar
- A single, deliberate input

This is not accidental.

This combination predates modern applications, dashboards, and touch interfaces. It is the same pattern found in test equipment, laboratory tools, industrial counters, and control panels built long before software became abstracted away from hardware.

Spectrum One inherits that lineage.

Because of this, the board is not bound to a single purpose. It is a general physical interface designed to make behaviour visible.

15.2 Making the Invisible Legible

At its core, the hardware does one thing well.

It reveals invisible processes in a readable, grounded way.

- The LED bar provides immediate, analogue style feedback
- The LCD provides slower, contextual information
- The single button enforces intention rather than interruption

Together, they form an interface that encourages observation instead of distraction.

This makes the platform useful wherever behaviour exists but is usually hidden, smoothed over, or ignored.

15.3 The LED Bar as a Visual Language

The ten segment LED bar is not a single indicator.

Each segment is driven independently. Each may represent its own threshold, state, or event.

This allows the bar to act as a visual language rather than a meter.

It may express:

- Intensity or direction
- Accumulation or decay
- Rhythm or change
- Progress or imbalance
- Uncertainty or transition

What matters is not the pattern itself, but that the pattern remains visible, persistent, and interpretable without explanation.

15.4 Other Directions the Platform Can Take

The following are not products, features, or roadmaps.

They are credible directions the same hardware may take if the firmware is replaced or extended.

They exist to demonstrate range, not obligation.

Counting and Events

The device can function as a physical counter.

- Manual event counting
- External pulse counting
- Rate and activity tracking over time

The LED bar reflects recent intensity. The display holds totals, averages, or session context.

This mirrors classic laboratory and industrial counters where trust is placed in what can be seen.

Time and Attention

With a stable display and expressive visual output, the board can operate as a time instrument.

- Focus intervals
- Session pacing
- Break reminders
- Long running timers

The LED bar expresses progress or urgency without noise. The display anchors the current state.

There are no notifications, no accounts, and no background services.

Environmental Observation

External sensors may be connected through analogue inputs, digital lines, or shared buses.

Examples include:

- Light
- Temperature
- Humidity
- Electrical behaviour
- Sound envelopes or vibration

In this role, the device becomes an observer rather than a precision instrument.

It shows trends, change, and presence without pretending to be laboratory grade.

Learning and Experimentation

The hardware is well suited to learning fundamental embedded concepts.

- State and transition
- Timing and drift
- Scaling and perception
- Filtering and averaging

Each LED segment can represent a state or threshold. The display provides names and context.

This makes the board useful not as a tutorial, but as a surface for experimentation.

Behavioural and Conceptual Systems

Not all systems measure physical quantities.

The same interface can be used to explore:

- Habits
- Pacing
- Discipline
- Deliberate slowness

In these cases, the value lies in interaction rather than data.

The device becomes a mirror rather than a sensor.

15.5 What Does Not Change

Regardless of purpose, the design philosophy remains the same.

- Measured output
- Limited interaction
- Predictable behaviour
- Independence from external services
- No required updates once built

The hardware encourages understanding rather than consumption.

15.6 A Finished Instrument, Not a Dead End

Spectrum One ships as a finished system. It does not ask to be improved.

At the same time, it is not a dead end.

If the firmware is replaced entirely, the board becomes a general purpose physical interface for ideas that benefit from being visible, slow, and tangible.

That openness is intentional.

15.7 Closing

Spectrum One tells one story through WiFi.

The hardware itself is capable of telling many others.

What matters is not the signal being observed, but the act of building, powering, and understanding a system from end to end.

What this becomes next is up to you.

16 Use of AI Tools

English is not my first language. It is the third language I use on a daily basis, including throughout this manuscript.

For this reason, AI tools were used for grammar checking, spelling correction, and sentence clarity. Their role was limited to language assistance.

All technical content, design decisions, explanations, and conclusions in this book are my own work. The concept of Spectrum One, its development from the first idea through breadboard prototypes to an assembled PCB, and the resulting reference build were produced independently by me.

All schematics, diagrams, and illustrations were created by me using Inkscape. All photographs included in this book are my own and were not post-processed or altered using AI-based tools.

AI was used as an assistive tool and did not replace authorship or responsibility. All responsibility and authorship for what is written, built, and documented here remain mine.

Jay J. Reszka

17 Figures

This section collects the reference figures used throughout the build.

Each figure is printed on its own page so it can be read at full size without being squeezed between paragraphs. That also makes it easier to flip back and forth while wiring, checking pinouts, or comparing layouts.

The text ends here on purpose. The pages that follow are single figures with their captions.

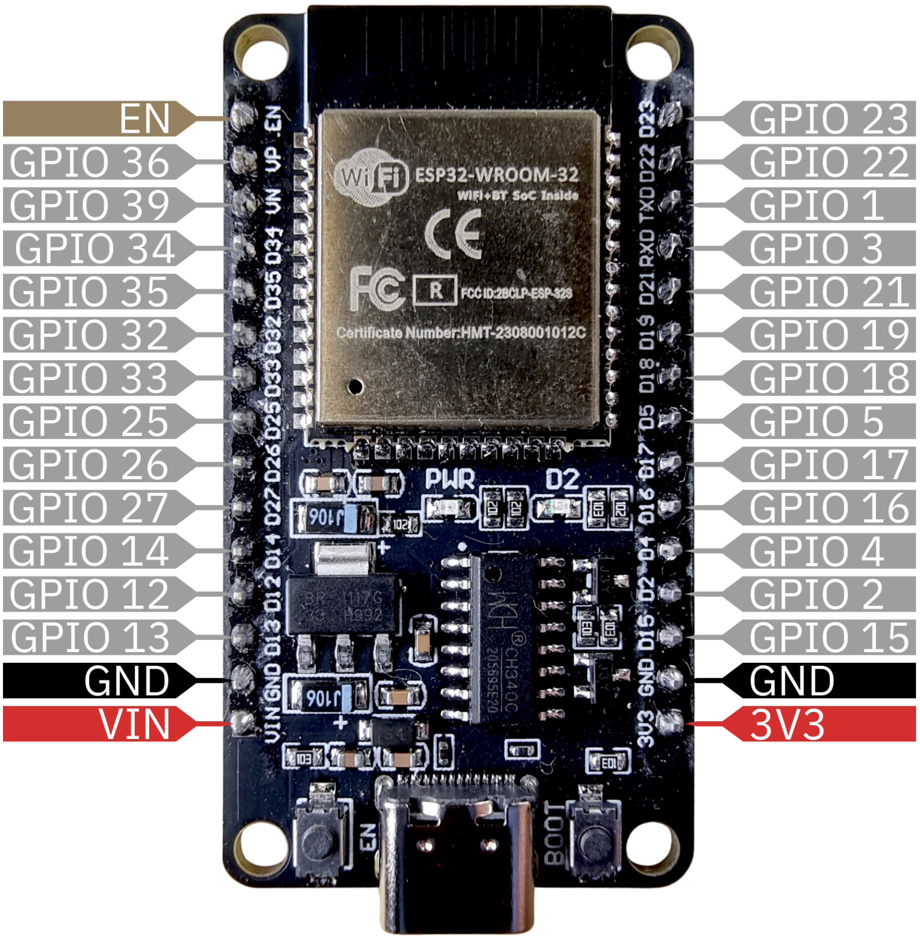
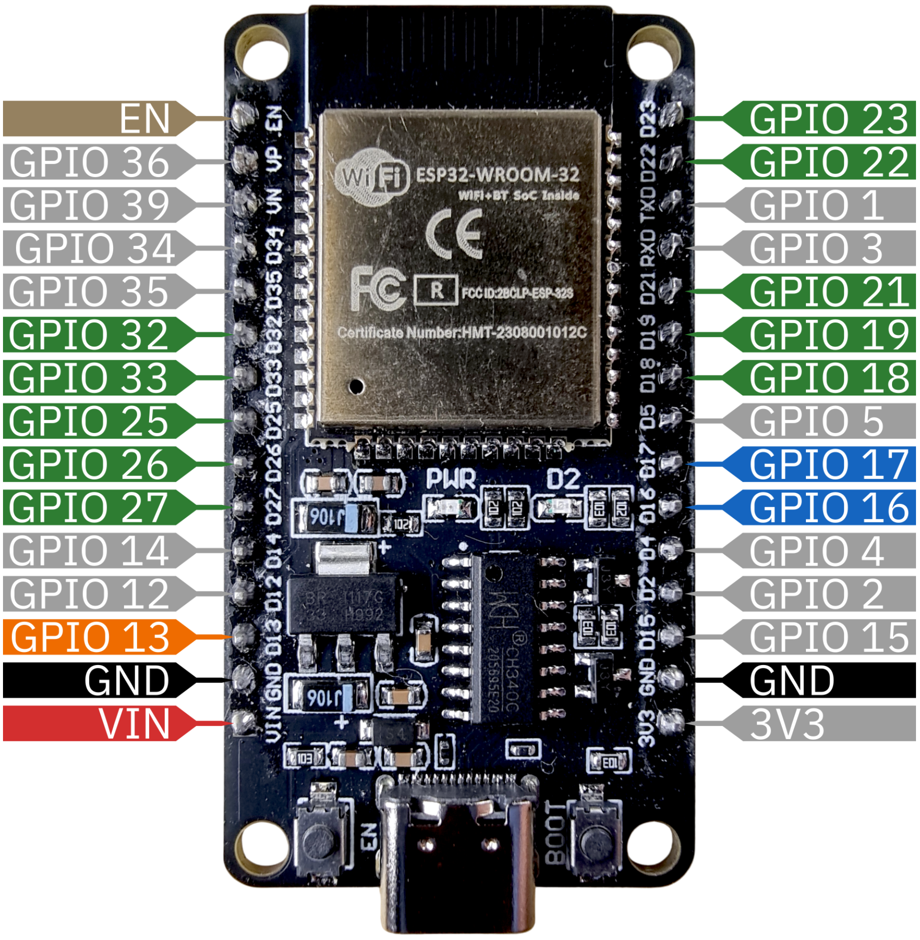


Figure 1: ESP32 Dev Kit V1 (30-pin) board pinout.



🟥 Power
⬛ Ground
🟫 EN / Control
⬜ Unused / Reference only

LCD I2C GPIO 🟦
LED Bar Array GPIO 🟩
Menu Switch GPIO 🟧

Figure 2: Spectrum One GPIO pin assignments.

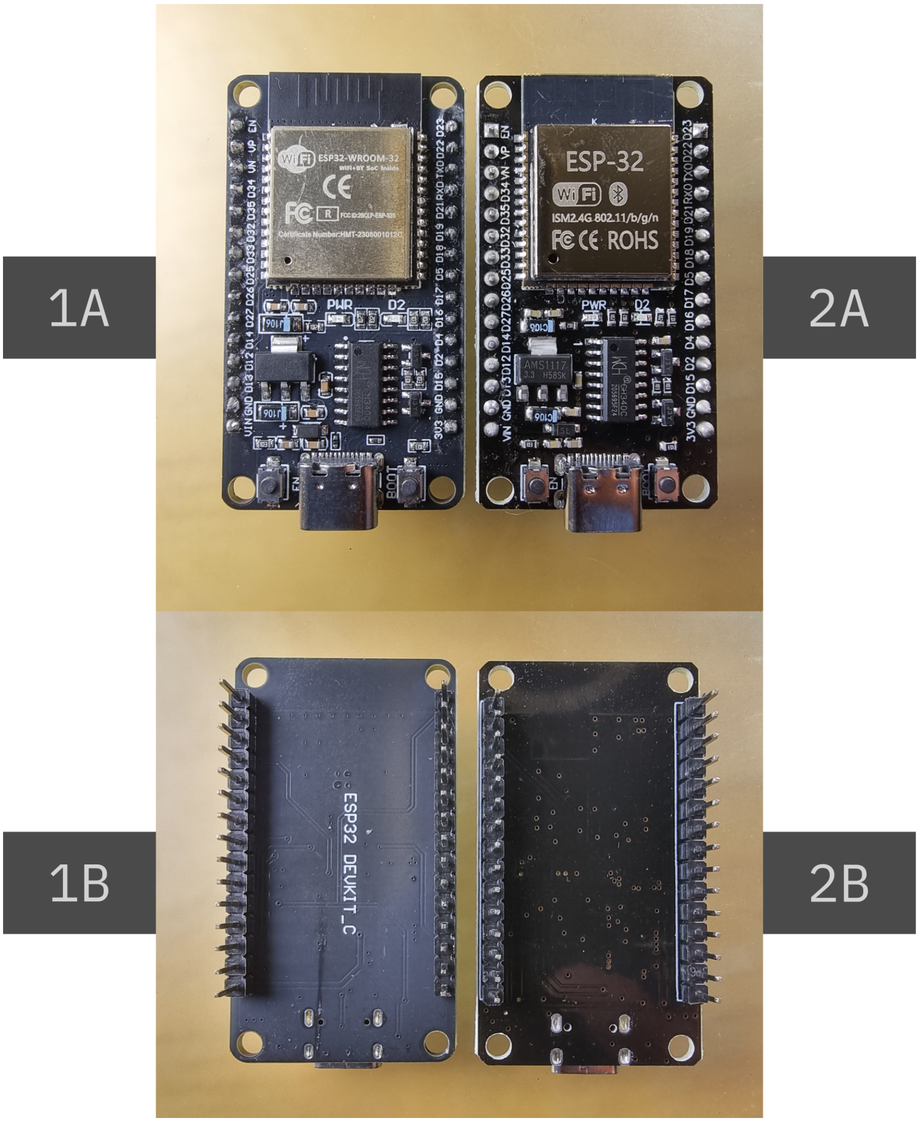


Figure 3: Two ESP32 development boards populated with ESP32-WROOM-32 modules, showing differences in silkscreen and layout without GPIO behaviour changes.

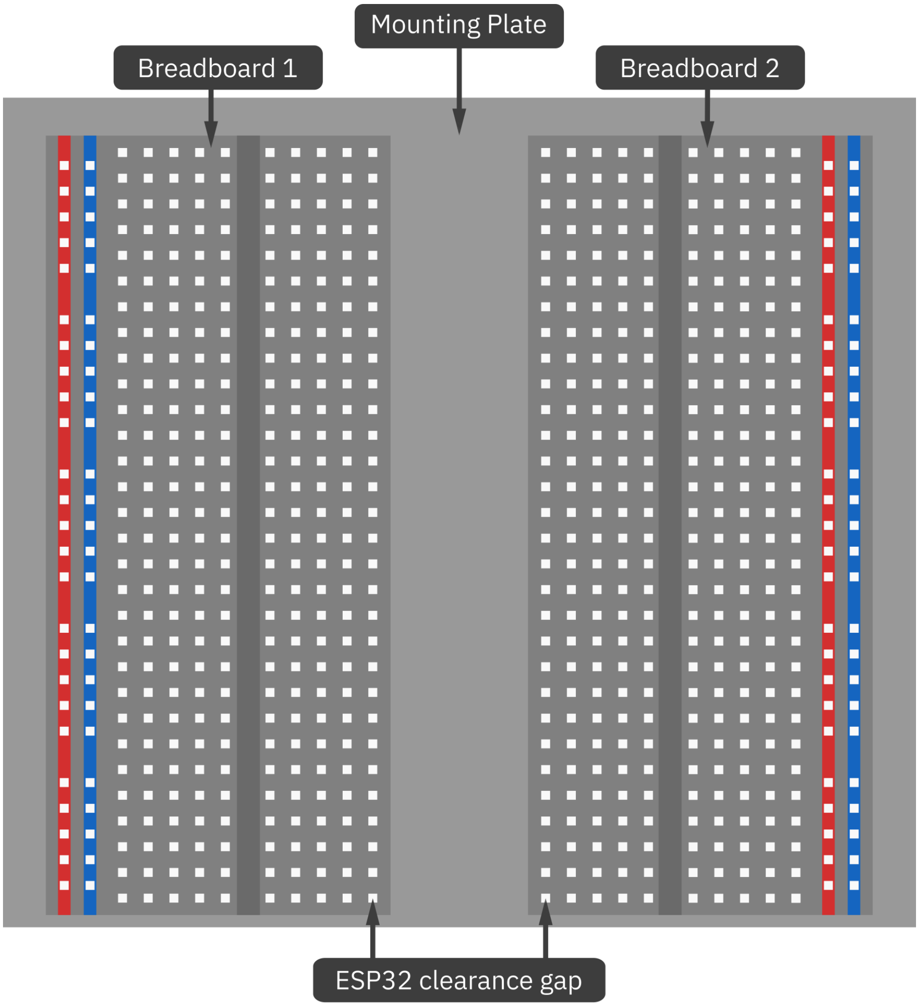


Figure 4: Dual breadboard base with inner power rails removed to create a central clearance gap.

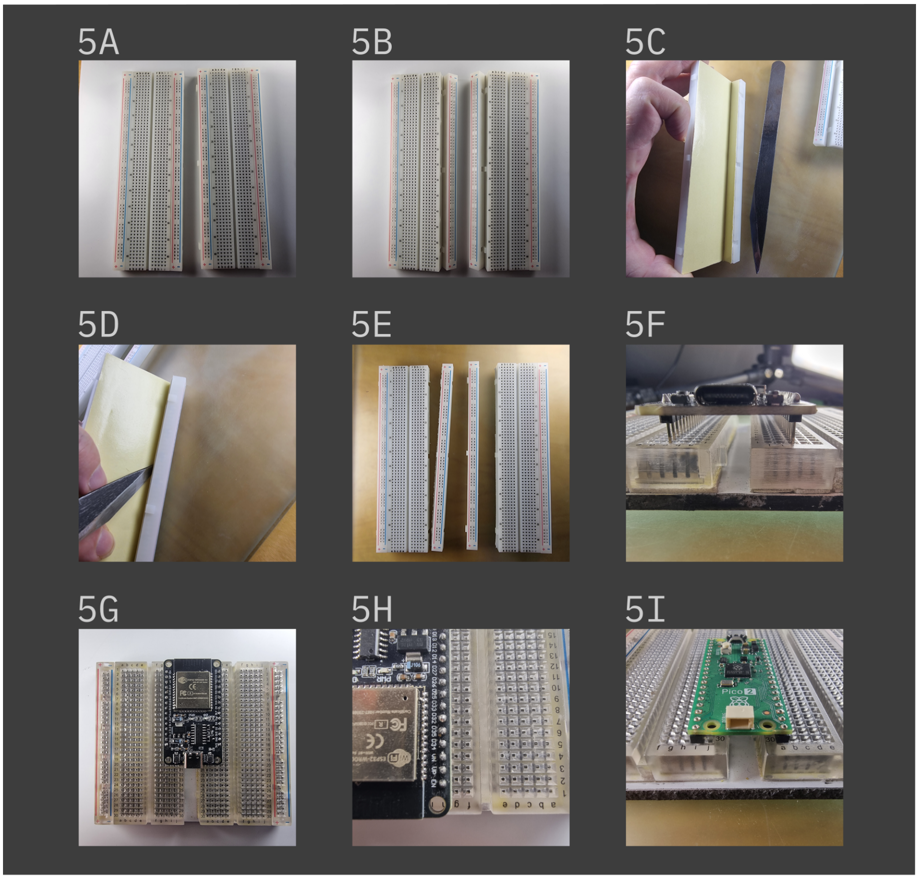


Figure 5: Breadboard modification stages and resulting layout.

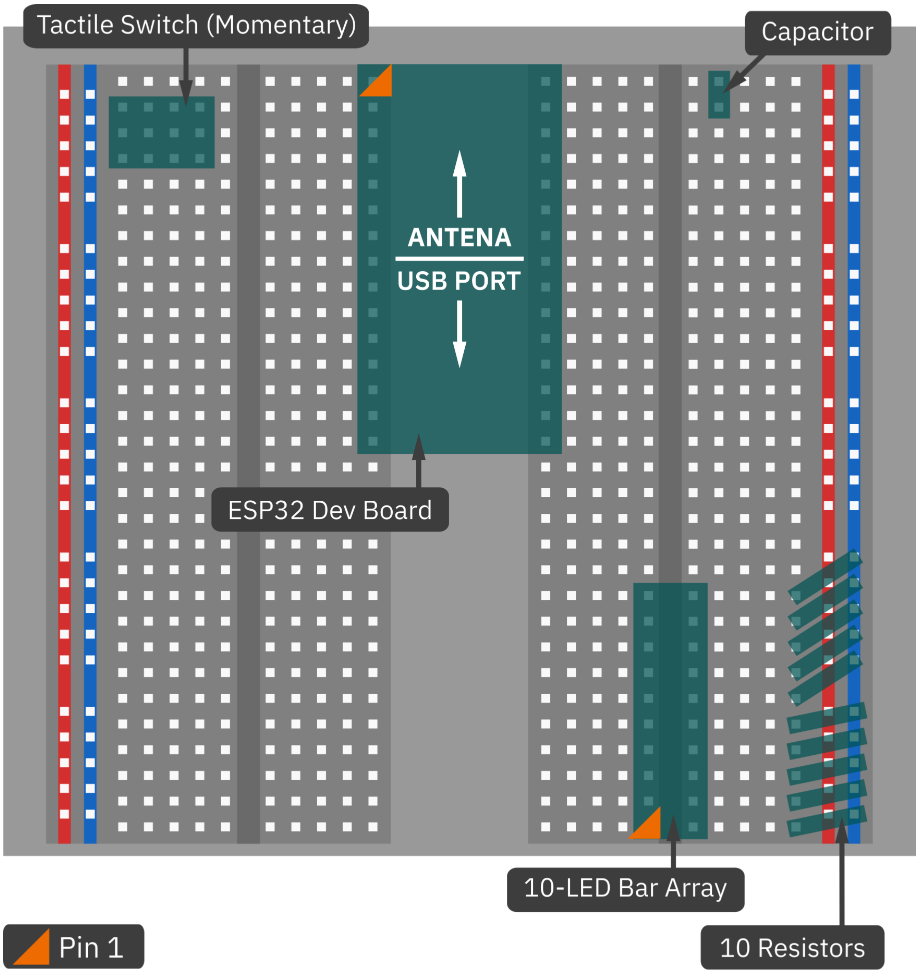


Figure 6: Component footprint on the breadboard during the Spectrum One prototyping phase.

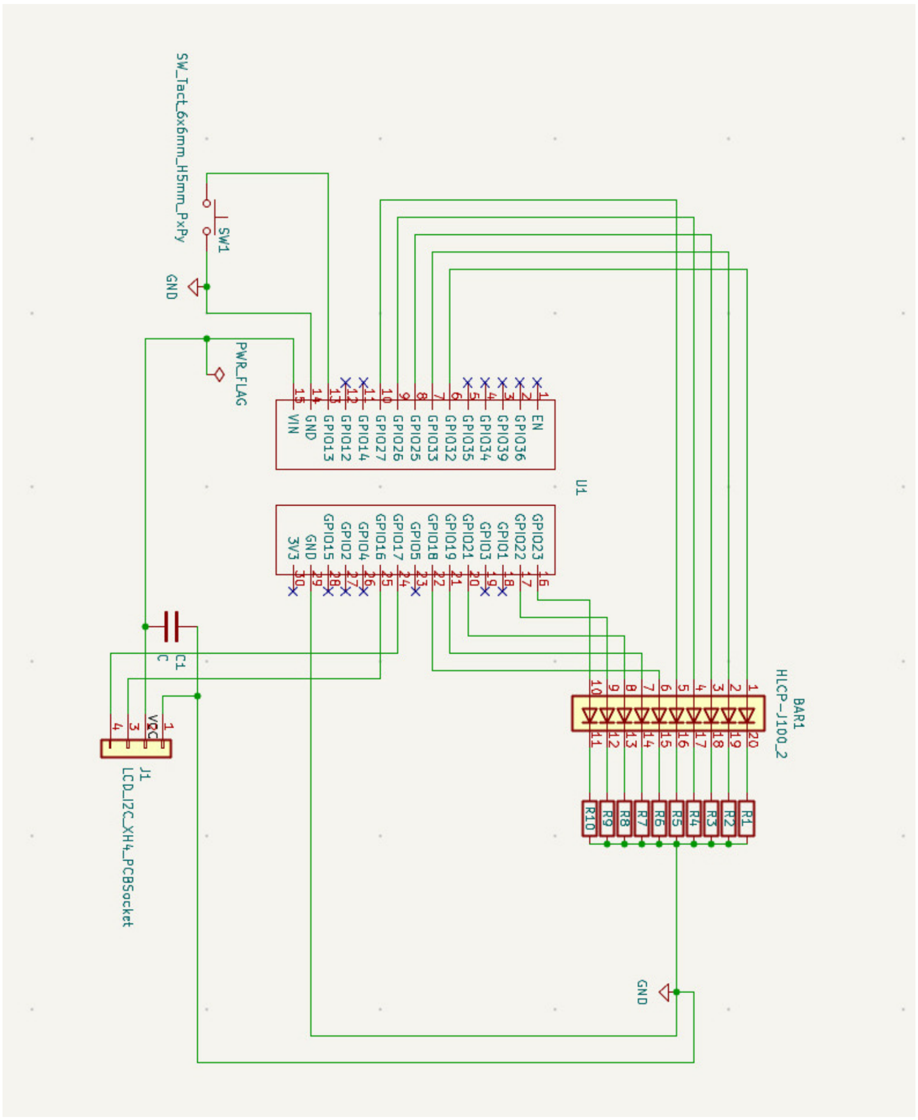


Figure 7: Complete schematic showing all signal and power connections.

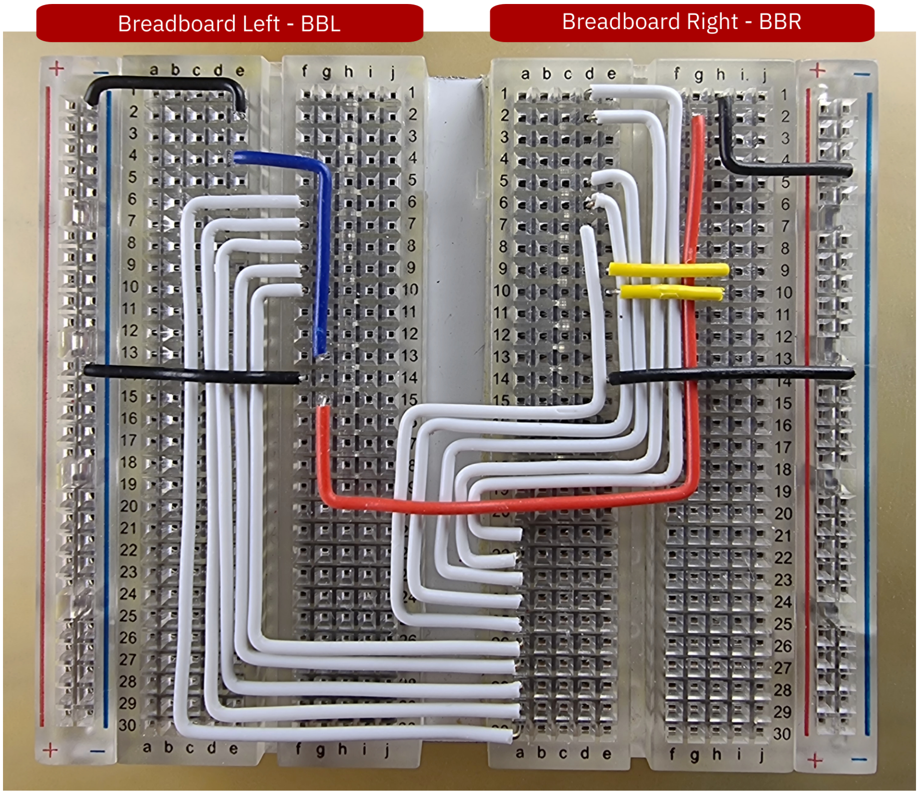


Figure 8: Top view of the dual breadboard showing completed wiring before LCD installation.

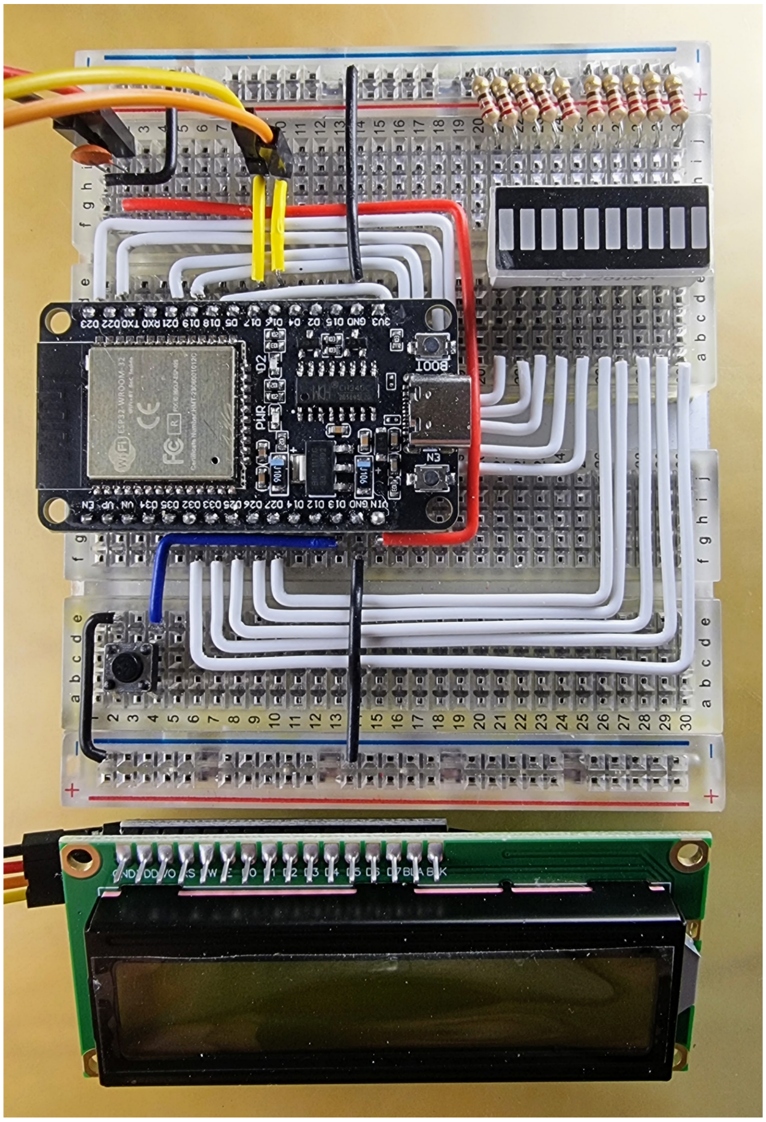


Figure 9: Fully populated breadboard showing the complete Spectrum One reference build.

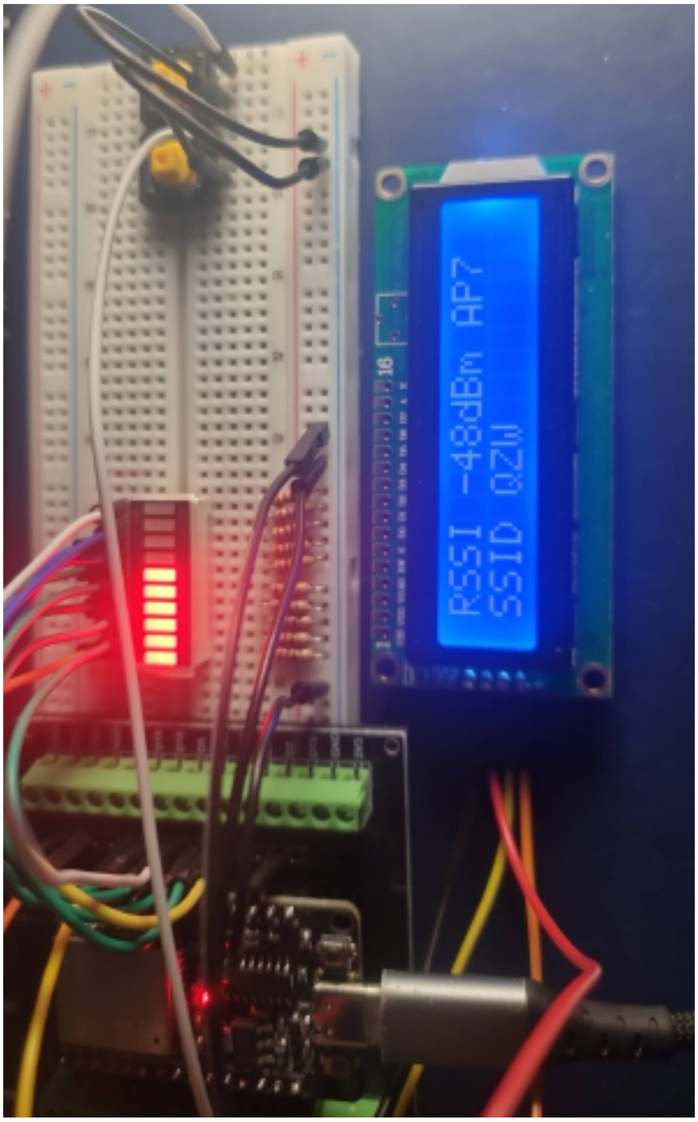


Figure 10: Early rapid prototyping stage using loose jumper wiring with LED bar and LCD output.

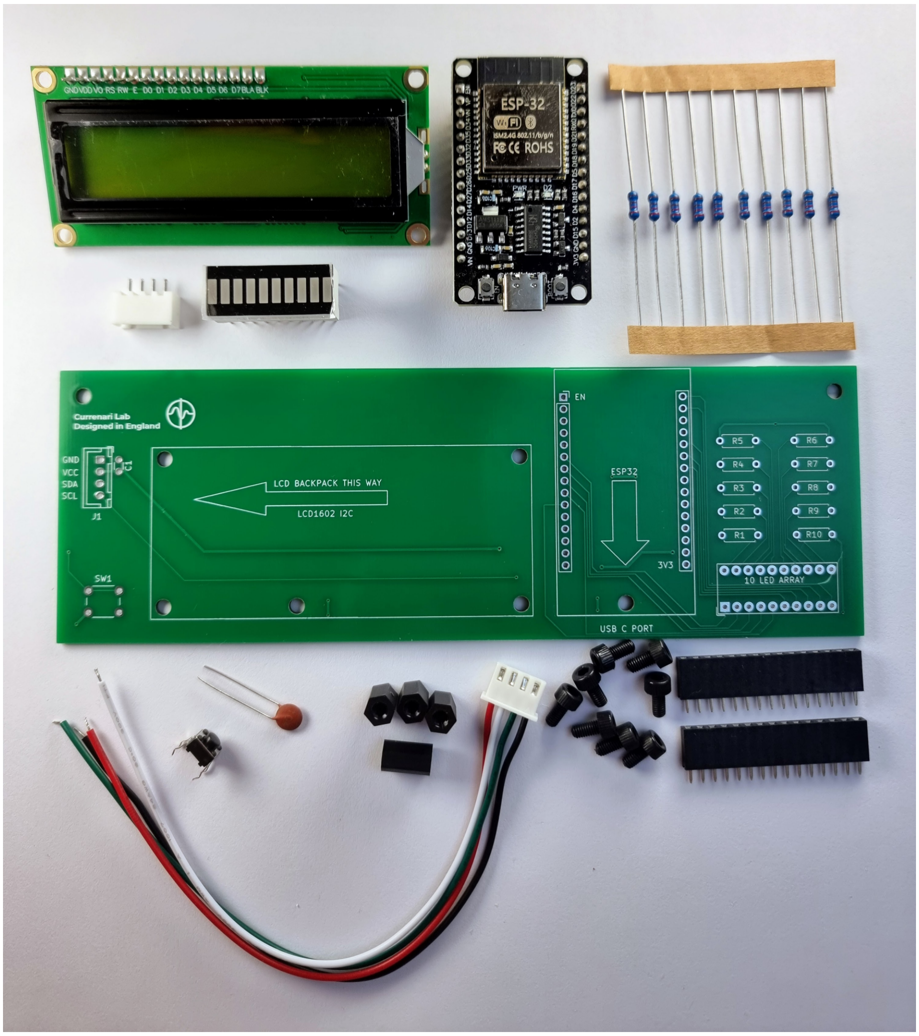


Figure 11: Complete Spectrum One DIY kit showing all required components.

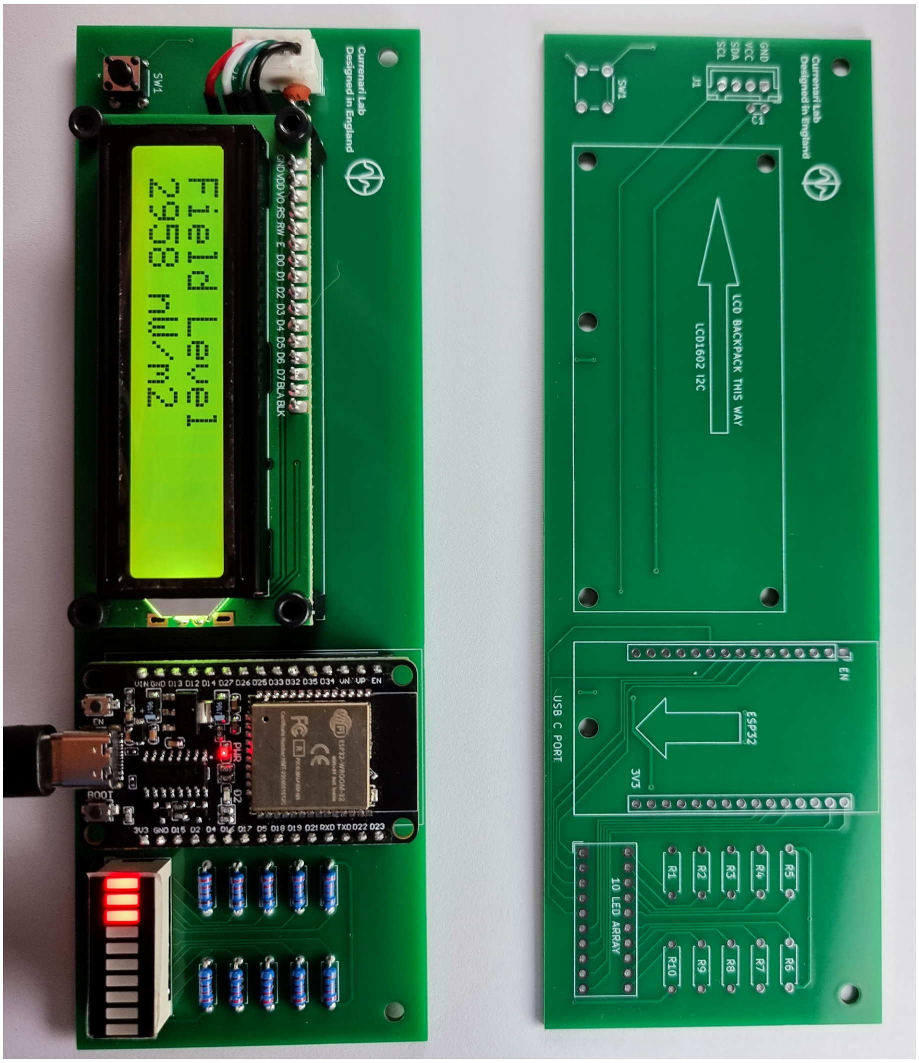


Figure 12: Assembled Spectrum One PCB showing the populated top side alongside an unpopulated board.

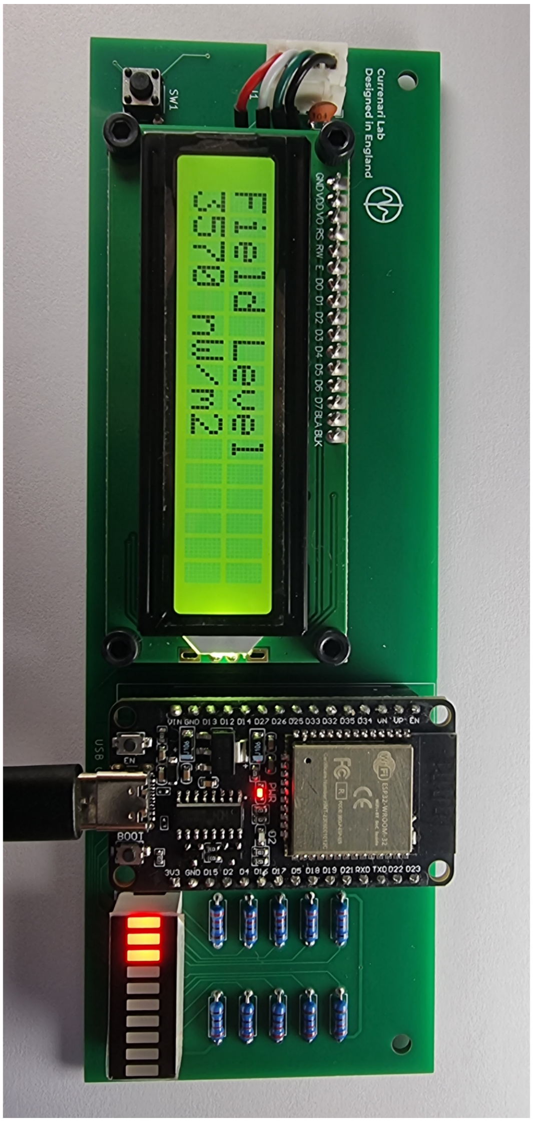


Figure 13: Assembled Spectrum One PCB in operation showing LCD output and active LED bar response.

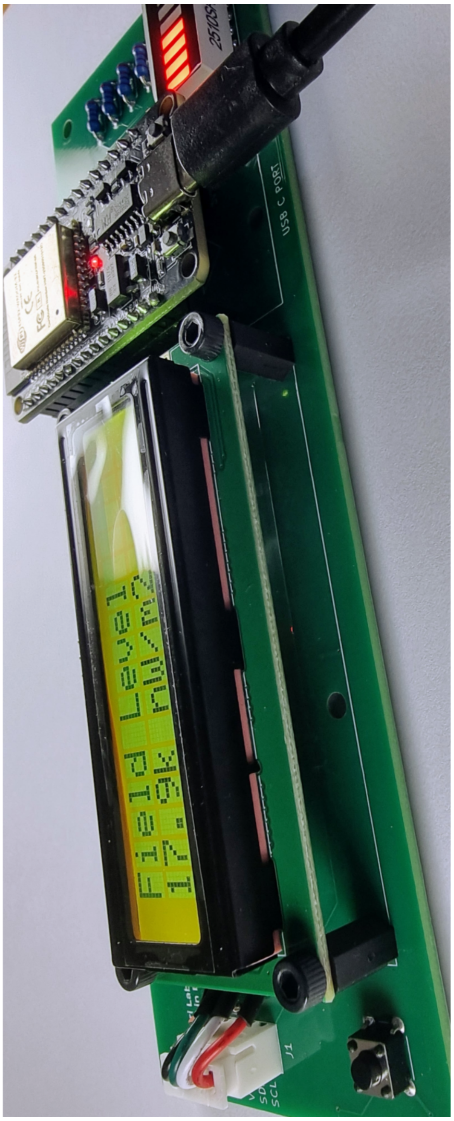


Figure 14: Angled side view of the assembled Spectrum One PCB in operation.