

MAYPHUS

Engineering notebook

Investigations, experiments
and the evidence between them

FIRST EDITION · 2026

10 October 2026

Three chapters selected from mayphus.org.
First published edition.

Contents

Follow the evidence	3
Make the model physical	7
Make progress recoverable	11

A note to the reader

This book connects three complete public records through editorial chapter introductions and small explanatory diagrams. The records retain their wording, measurements and qualifications; formatting is adapted for reading on paper. First-person statements belong to the original records. The investigations have not been rerun for this edition.

Human work. AI-assisted words.

Based on my real work, experiments and observations. AI helps turn them into published content. Claims should be judged by their evidence; corrections are welcome.

The chapter source links lead to the current web editions. The third chapter also links to its interactive model demo; a PDF cannot run that interface.

Follow the evidence

From silent UART to a working NanoPi R2S

This investigation separates two failures that produced the same symptom: a silent console. Its evidence moves from checking the instrument to identifying the boot image and observing a working system.

- 01 Check the serial path
- 02 Match the board and image
- 03 Verify the boot

Editorial schematic of the recorded investigation. Each step has a different evidence boundary.

August 19, 2026 · RK3328 · UART · Alpine Linux

Today I brought a NanoPi R2S from a completely silent serial console to a verified Alpine Linux boot. The board was healthy. Two independent problems had hidden that fact: an intermittent USB-TTL path and a 64 GB SD card flashed with an image for the wrong Rockchip family.

The result

With a replacement serial adapter and the correct official FriendlyELEC RK3328 image, the board spoke immediately at 1,500,000 baud. The log identified a NanoPi R2S, U-Boot 2017.09, Linux 6.6.134+, and Alpine Linux 3.23.4. I had left the UART VCC pin disconnected and used only ground plus crossed TX and RX, so the adapter observed the board without trying to power it.

Hardware

NanoPi R2S Rockchip RK3328 and 1 GB RAM

Serial proof

1,500,000 baud Dedicated 3.3 V debug UART

Booted system

Alpine 3.23.4 Linux 6.6.134+ after U-Boot 2017.09

Storage after first boot

58.9 GiB userdata Backup GPT repaired and userdata expanded

How the silence was isolated

- Start with the instrument. The first USB-TTL adapter and jumper path behaved intermittently in loopback, so silence at the board could not yet be trusted as evidence about the board.
- Replace one uncertain link. A replacement adapter passed repeated loopbacks at both 115200 and 1,500,000 baud. That made the serial path a known-good measuring tool.
- Inspect the boot medium. The original SD card contained a Rockchip image from the RK3588 family, not an R2S/RK3328 image. Its early boot code could never match this chip.

- Build and verify the right image. I assembled and flashed the official FriendlyELEC RK3328 Alpine image, then compared the boot-critical region byte-for-byte before putting the card into the board.
- Read the first real evidence. U-Boot appeared over UART, Linux continued, Alpine reached its login environment, and the system reported the correct board identity.

The important shift was treating every link as a separate hypothesis. A silent terminal did not mean a dead board; it meant only that the complete path had not yet produced readable bytes.

What actually boots

An SD card for this board is not just a Linux filesystem. The RK3328 begins with code built into the chip, then reads several board-specific stages from raw locations on the card before it can start a kernel.

- 01 Boot ROM Fixed inside the RK3328
- 02 IDB loader Raw early loader on the SD card
- 03 Trusted firmware Prepares the secure handoff
- 04 U-Boot Finds and starts the operating system
- 05 Linux + device tree Kernel plus the R2S hardware description
- 06 Alpine root filesystem The installed userspace
- 07 OpenRC services SSH and other configured services

Each stage only needs to load the next one. Seeing the U-Boot banner proves stages 1–4 before Linux or the root filesystem is involved.

What is on the SD card

The working card combines raw boot components, small support regions, a compact Alpine system, and a large data partition. On first boot the image repaired the backup GPT at the end of the larger card and grew userdata to use the remaining space.

Early boot raw area IDB loader, trusted firmware, U-Boot
Boot support small regions misc, DTBO, resources, kernel and boot files

Alpine rootfs about 448 MiB The compact operating system

userdata about 58.9 GiB Expanded on first boot

Conceptual rather than to scale: the tiny raw boot regions at the front matter just as much as the visible filesystems.

A quiet, healthy first boot

I kept the first health pass read-only. The board reported about 946 MB of total memory with about 851 MB available, a temperature near 44 °C, and low load. Configured services, including SSH, started. Both Ethernet ports correctly showed no carrier because no cables were attached. The hardware clock was old, and the remaining boot warnings were nonfatal defaults rather than failed hardware.

SSH being ready is useful, but it also sets the next safety boundary: the vendor-default root password must be changed before either Ethernet port is connected. I am deliberately not recording that password here.

The lesson I will reuse

A U-Boot banner over UART is already a strong milestone. It proves that the RK3328 Boot ROM found the raw loader, trusted firmware ran, and U-Boot started. A Linux kernel and root filesystem are not required to reach that point.

The three UART wires can stay fixed while I experiment with operating systems, but the software side is not interchangeable. Each image must enable the correct serial console and may need its own loader, firmware, U-Boot, kernel, and device tree.

Next experiments recorded on August 19, 2026

At that point, I wanted to try Ubuntu, Debian, OpenWrt and FreeBSD, using a separate SD card for each and preserving the working Alpine card as a recovery baseline.

Later records describe [booting FreeBSD from RAM](#) and an [experimental USB Ethernet workaround](#). Those pages retain their own results and limits; they are not outcomes of the August 19 Alpine session.

Make the model physical

A TinyStories model on an RP2040 board

The next record crosses from software into a physical device. It distinguishes training on a Mac, inference on the microcontroller, and the human observations needed to verify the screen and button behavior.

- 01 Train on the Mac
- 02 Quantize and check C inference
- 03 Observe the RP2040

Editorial schematic of the deployment path. Training and on-device inference are distinct.

Training · quantization · C inference · physical hardware

I trained a very small language model, converted it into an embedded C runtime, and ran it locally on my custom Raspberry Pi Pico display board. I did not solve the individual technical problems myself. I described what I wanted and what I could see; the AI agent translated that into code, measurements, firmware, and the next test.

The result

The final model is a 98,944-parameter byte-level Transformer with a 256-byte context. It was trained on short setup-and-conclusion stories, quantized to INT8 weights, checked against its Python reference through a portable C implementation, and then built into native RP2040 firmware.

Measurement	Recorded result
Model weights	98,944 bytes INT8 weights, plus 80 bytes of scales
Static SRAM	143,504 bytes within the RP2040's 264 KiB budget
Firmware image	145,980 bytes native RP2040 flash and code image
Physical generation	10.394 seconds sentence-aware output on the connected board

From intention to a physical result

- **Describe.** I said what I wanted: a TinyStories-style model made specifically for the RP2040, not a larger computer hidden behind it.
- **Inspect reality.** I connected the board and described what I saw on its screen, buttons, and USB connection. The agent inspected the software and measurements.
- **Implement.** The agent built the Python model, training path, quantization export, numerically checked C runtime, display driver, and firmware.
- **Verify.** I supplied the physical observations the computer could not infer: whether the display was blank, whether text appeared, and what happened after I pressed BOOTSEL.
- **Use.** The board now generates a fresh bounded micro-story locally at startup or after a button press.

My contribution was not pretending to be the firmware engineer. It was holding the intention, making the decisions, connecting the physical world, and describing reality accurately enough for the next step. The working loop was: describe, inspect reality, implement, verify, use.

What ran where

Training ran on my Apple M1 Mac, because training a Transformer is not the job of this microcontroller. The completed quantized model and inference runtime run on the RP2040 itself. Story generation does not call a cloud model or another computer.

The 256-byte model trained for 10,000 steps on 18,522 derived micro-stories. Its final validation loss was 0.7502 and perplexity was 2.1175. A scaled INT16 key-value cache made the longer context fit in SRAM, while the display driver draws glyphs directly over SPI instead of reserving a framebuffer.

Captured from the physical board

```
TINYSTORIES_RP2040_READY
model_state_bytes=138244
DISPLAY_READY
STORY_BEGIN
Once upon a time, there was a big treest of animals.
STORY_END
bytes=52 generation_ms=10394
```

The language is imperfect, and I am keeping that visible. This is a deployment proof for a genuinely tiny model, not a claim that a 99K-parameter storyteller matches a large language model.

What this proves

The physical board executed the quantized Transformer, initialized its ST7789 display, generated a learned story ending locally, and displayed the result. Earlier tests also verified that pressing and releasing BOOTSEL triggered a different generation without rebooting.

It does not prove that the model was trained on the Pico, that its writing is generally coherent, or that an AI can observe physical hardware without a person. The achievement is the complete collaboration: a person can direct a real embedded-AI project through intention and observation while an agent handles the technical path and preserves an evidence trail.

Make progress recoverable

GPT-124M benchmarking and restart recovery

The final record returns to measurement at a larger scale. The question is whether saved training state can survive a fresh session, with timing windows, intermediate milestones and final evaluation kept separate.

- 01 Measure a timed pilot
- 02 Save and replay full state
- 03 Resume in a fresh session

Editorial schematic of the tested recovery path. These tests do not establish recovery under every possible interruption.

SOURCE RECORD

mayphus.org/gpt124m-restartable-training/

I trained a small GPT model on free Kaggle GPUs. The work made training measurable and restartable: a timed single-GPU pilot, a checkpoint that preserves the training state, and continuation of a longer two-GPU run across fresh sessions.

This record was first published on October 5, 2026. The earlier session milestone was verified on October 6 in Asia/Shanghai (October 5 UTC). Final completion was verified at 01:46 UTC on October 6, 2026: the one-billion-token demo and its in-domain held-out evaluation are complete. The earlier measurements below remain as the run history.

Try the model

- [Try the public text-completion demo.](#)
- [Download the model and read its model card.](#)

Give it the beginning of a passage to continue. This is a base model, not an instruction-tuned chat assistant; its completions can be repetitive.

What starts from scratch

The model has 124,439,808 randomly initialized parameters. Its GPT-2 architecture uses 12 layers, 12 attention heads, a width of 768, and a 1,024-token context. No pretrained model weights are loaded.

I use the existing GPT-2 tokenizer and public FineWeb-Edu data. Starting the model weights from scratch does not mean inventing a new tokenizer or collecting a new corpus.

A pilot with an explicit timing boundary

The single-T4 pilot processed 6,881,280 tokens in total. Its timed portion covered 6,750,208 tokens in 904.14 seconds, including checkpoint work. That gives approximately 7,465.9 tokens per second. Reported peak GPU memory was 4.46 GiB.

Single-T4 measurement	Result
Total pilot tokens	6,881,280
Tokens inside the timing window	6,750,208
Timed duration, including checkpoint work	904.14 seconds
Throughput inside that window	7,465.9 tokens/second
Reported peak GPU memory	4.46 GiB

The timing window matters. Dividing the total pilot token count by a duration that measured only part of the run would produce a different and misleading rate.

On a held-out set of 65,536 tokens, loss fell from approximately 10.99 to 8.41. Generated samples were still repetitive. These are early diagnostics, not a capability evaluation or evidence that the model is ready for practical use.

A checkpoint must preserve the next step

Saving model weights alone is not enough to resume the same training trajectory. The optimizer, learning-rate schedule, mixed-precision scaler, random-number generators, and data position all influence what happens next.

The full-state replay test restored the model, optimizer, scheduler, scaler, RNG state, and data offset. The tested continuation reproduced the expected result exactly.

I then tested a separate failure boundary: a completely fresh Kaggle session loaded the saved checkpoint and resumed successfully. That matters because a checkpoint that works only while the original process is alive has not demonstrated useful recovery.

These results establish the tested replay and saved-session recovery cases. They do not establish identical behavior across every hardware configuration or every possible interruption.

The longer two-T4 run

A separate training token store contains 1.1 billion token positions. A document-hash holdout separates the selected evaluation documents from training. No additional near-duplicate

decontamination has been established, so this split should not be described as a fully decontaminated benchmark.

An initial two-T4 distributed-data-parallel test reached approximately 14,787 tokens per second across the two GPUs. That is an initial measurement, not an established sustained rate for the longer run.

A completed session and a verified continuation

At the verified October 5 UTC session boundary, the first production session had safely finished approximately 10.5 hours of two-T4 training and saved its full state. The saved checkpoint represented 548,405,248 processed tokens: 54.84% of the one-billion-token target. This is a fixed checkpoint milestone, not a live progress counter.

Two-T4 production-session measurement	Result
Tokens at the saved checkpoint	548,405,248
Share of the one-billion-token target	54.84%
Reported sustained throughput	Approximately 14,533 tokens/second
In-domain held-out loss	4.0352
In-domain held-out perplexity	56.55

The full-state checkpoint's hash was verified. In a fresh successor session, the distributed replay check passed, and training then advanced beyond the saved checkpoint. This tests a stronger boundary than keeping one process alive: saved state was recovered in a new two-GPU session and used to continue the run. These production figures are separate from the single-T4 pilot above. The held-out loss and perplexity describe the selected in-domain holdout, with the same limits on decontamination already noted; they are not a general capability score.

The one-billion-token run was not complete at that intermediate verification point. The final result below follows the recovered continuation.

Completed one-billion-token demo

Final verification at 01:46 UTC on October 6 confirmed 1,000,341,504 predicted training tokens. The slight overshoot of one

billion comes from finishing a fixed global batch. The model still has 124,439,808 parameters initialized from scratch; no pretrained weights were introduced.

Final two-T4 result	Measurement
Predicted training tokens	1,000,341,504
In-domain held-out loss	3.5953
In-domain held-out perplexity	36.43
Held-out token predictions evaluated	65,536
Final training-batch loss	3.6821
Sustained continuation throughput	Approximately 15,683 tokens/second

The final training-batch loss is distinct from the held-out evaluation loss. The continuation throughput describes the resumed session, not the single-T4 pilot or the earlier two-T4 session.

The final full-state checkpoint is approximately 1.65 GB. Its SHA-256 checksum was verified, and its saved token count and data offset both match 1,000,341,504. Together with the earlier fresh-session numerical replay, this establishes the tested recovery path and a verified final saved state.

The run used free Kaggle resources only. This completes the one-billion-token demonstration; it does not announce or commit to further training.

What completion does—and does not—mean

The output remains repetitive. This is a next-token pretrained base model, not an instruction-tuned conversational assistant. The held-out figures come from the in-domain FineWeb-Edu document-hash split described above, not a general capability benchmark or a fully decontaminated evaluation.

The completed result is a small model trained from random initialization, measured through a single-GPU pilot and two-GPU sessions, recovered across a fresh session, and saved with verified final counters and checksum. Better loss and a completed token budget do not by themselves establish usefulness or reliable reasoning.