CryNet

Infant cry detection for a two-cot ESP32 system — the complete project dossier

Everything that was built, measured, decided and shipped: the research that chose the corpora, the model that came out of it, the three firmwares, the radio protocol, the cry log and the hosted dashboard, the wiring, the decision record with all one hundred and eight entries, the forty-one faults and how each was diagnosed, every measured number with the method that produced it, and the exact procedure to rebuild the whole thing from an empty checkout.

Project	CryNet — two-cot infant cry monitor
Repository	`C:\Users\HP\.t3\projects\fp-two-node`
Remote	`origin` → `https://github.com/Amazo123/CryNet-TwoNode.git`
Head at time of writing	`de7bbe6`, on `main`, pushed to GitHub
System	2 × ESP32-C3 Pro Mini cot nodes → ESP32-S3 hub → ESP32-C3 SuperMini wearable
Dates	2026-10-28 → 2026-11-25
Status	Four boards verified on the bench; hosted dashboard live; battery runtime unmeasured
Audience	A reader who must defend every decision and rebuild the system unaided

What this document is

This is not a summary. It is the whole project, laid out in the order a defence would need it: what the system is, why it is built that way, how every number was obtained, what broke, how it was found, what was decided, and exactly what to type to reproduce any of it.

It is written to be *defensible*. Every claim carries its evidence — a file, a line, a log, a commit, a measurement — so that any statement here can be checked rather than believed. Where the project does not know something, this document says so; the open items are in Chapter 19, and they are the honest state of the work, not omissions.

How it was made

The body of this dossier was assembled from the repository itself: `DECISIONS.md` (every decision and every issue, with the reasoning), `PROJECT_HANDOFF.md` (the frozen contracts and the file map), `SYSTEM_UPDATE.md` (the living status log, rewritten after every change), `SYSTEM_WIRING.md` and `HARDWARE_COMPONENTS.md` (the hardware and its power chain), `DEMO_RUNBOOK.md` (the bench procedure), the two flashing guides (the offline route needs no toolchain at all), and the source of the three firmwares and the research pipeline. The full text of the first-party source is reproduced in Appendix B so that the claims can be checked against the code, not against this prose.

Conventions

- **D-n** — a decision, with its rationale. **I-n** — an issue: what it looked like, how it was diagnosed, what was done, what happened. **O-n** — an open item. These identifiers are the project's own, from `DECISIONS.md`.
- File references are `path:line`; quoted log lines are verbatim.
- Where a figure is a projection rather than a measurement, it says *projected*. That distinction is preserved throughout.

- Quirks in the record worth knowing before a defence: **D-51** and **D-52** were each used twice, and **D-70** and **D-95** were skipped, so one hundred and eight decision rows carry one hundred and six distinct numbers. These are in Chapter 19; neither reuse is a missing decision.

Generated from the repository at `de7bbe6` on `main`. Rebuild with `python docs/dossier/build.py`.

Contents

Part I — The system 9
- What CryNet is 9
- The three kinds of unit 9
- Two cots, enforced in the link layer 9
- The honest status 10
- What the system is not 11
- The system in one paragraph 11

Chapter 2 — Architecture and the end-to-end path 12
- Why the system is split the way it is 12
- The five hardware pieces 13
- The path a cry takes, step by step 13
- Timing budget 18
- What runs where, in memory 18

Chapter 3 — The data research: what was chosen, what was rejected, and why 20
- The question the research had to answer 20
- The primary corpus: ICD 20
- The supplementary corpora 21
- What was rejected, and on what evidence 22
- The level asymmetry — a finding that changed the firmware 22
- How the corpora are turned into training rows 23
- Getting the data 24

Chapter 4 — The signal contract: every constant and why it is that value 25
- What "frozen contract" means here 25
- The chain 25
- The constants 26
- Why each step is not optional 26
- The eight compile-time guards 28
- Where the working set lives, and the 3× speed-up 28
- How parity was proved 29

Chapter 5 — The model: architecture, training, mining, calibration, quantisation 30
- The constraint that designed the network 30
- The four candidates 30
- Training 32
- Hard-example mining — the second round 32
- Threshold calibration 33
- INT8 quantisation 33
- The deployed artefact 34
- Results 34
- Why the legacy model was replaced 36

Chapter 6 — The hub firmware 37
Layout 37
Boot, in order 37
Graceful degradation 39
The console 40
link_stage.c — the state machine 41
crynet_infer.cpp — the interpreter wrapper 43
Build configuration 43

Chapter 7 — The cot node firmware 46
What the node is responsible for 46
The audio path 46
The adaptive trigger — D-63 47
Freezing the right two seconds 48
Cry mode — D-69 49
Protecting a window that is in flight 49
Identity: this node must be 1 or 2 49
The node's own view of the link 50
The I^2S zero watchdog — D-79, committed, not yet flashed 50
The radio 51
Framing 53
The serial console 53
Build 53

Chapter 8 — The wearable firmware 54
The job 54
The rule that shapes the whole file: the radio must never block the human interface
The contract it is written against 54
The alert callback 55
The state callback 55
Reconnection, and the bug that made a live link deaf 56
The haptic drive 57
The display 59
Build 59

Chapter 9 — The radio link, and the airtime problem 60
Why ESP-NOW 60
The wire format 60
The airtime problem — the hardest issue in the project 61
Reducing the deafness itself (D-61) 63
The air counters 64
Two nodes on one channel 64
What was measured, finally 64

Chapter 10 — The announcement policy: turning a probability into an alarm 65
Why this layer exists at all 65
The constants, with their measured justification 65
The rule as implemented 66
The trade, measured in both directions 67
Clipping refusal 68
Hold, arbitration and hysteresis 68
Deafness is an alert, not a silence 68
The policy in one sentence 69

Chapter 11 — The Bluetooth contract and the caregiver's device 70
The frozen legacy contract 70
Why NimBLE and not Bluedroid 71
The advertising policy — the sharpest trade-off in the firmware 71
Publishing 72
The console shortcut that made this testable 72
The host-side test clients 73
Coexistence 73

Chapter 12 — The cry log, the network, and the dashboard 74
Why the hub needed a record at all 74
The cry log — crynet_log 74
The clock — crynet_clock 76
The network stage — crynet_net 76
The dashboard — two copies, one contract 78
The contract test — the one thing no compiler checks 79
The boot banner, and the stack that was too small 79
What this subsystem deliberately does not do 80

Chapter 13 — Hardware: wiring, bill of materials, power and the haptic drive 81
The four boards and their pins 81
The power chain, and its three rules (D-84) 83
Cell choice (D-81), and why it differs per unit 84
What was removed, and why (D-83) 84
Consolidated bill of materials 86
The haptic drive — what is proven, and the one thing that is not 87
Safety 88

Chapter 14 — The hub panel: driver, UI, and five debugging rounds 89
Why a panel at all 89
The driver 89
Five rounds of debugging, and what each one taught 89
The palette, and how it was chosen 90
Layout — two cards, one reading per row 91
The rounded corners, and a host-side regression check 94
What the panel is not 94

Chapter 15 — Every measured result, and how it was obtained 95
How to read this chapter 95
Model quality — offline, on the corpora 95
The front end — parity and speed 96
Against the deployed INT8 model, on real audio 96
The in-room confusion session — O-9, closed 96
The live chain — hub + node + a real BLE client 98
The two-cot bench session — 2026-11-05/06 98
Memory and build 100
The adaptive trigger, measured 100
Clipping, measured — and the session that made it a first-class finding 101
The dashboard phase, measured — 2026-11-24/25 103
What has not been measured 103

Chapter 16 — Bench verification and the demonstration procedure 105
The distinction that governs all bench work 105
Setting up a bench session 105
The demonstration procedure 105
Reading the log 107
The two-cot checks that are specific to this system 107
Troubleshooting, in the order to try 108
The demo kit and how it was built 108
Other verification procedures 109

Chapter 17 — Toolchain: what was installed, what broke, and why it is pinned 110
The starting point 110
The versions that are known to work 110
D-45 — why ESP-IDF is installed by hand 110
D-46 — why TFLM is vendored inside the project 111
D-47 — DIO flash mode 111
D-48 — the Arduino side 111
D-49 — every serial tool resynchronises on its own magic 112
D-50 — the notebook is a build artefact 112
The build commands 113
Where the evidence lives 113

Chapter 18 — The host tools: twenty-nine programs that made the firmware debuggable 114
Why this chapter exists 114
Flashing and installation 114
Listening and capture 115
Audio collection and labelling 115
Measurement and verification 116
The record and the dashboard 116
BLE 116
The research scripts 117
The offline images, and the check that guards them 118
The pattern worth defending 119

Chapter 19 — Limitations, open items, and what to do next 120
Why this chapter exists 120
1. The open items, and where each one stands today 120
2. The two-cot sub-items, and the items with no O-number at all 123
3. What was knowingly given up 128
4. What has not been measured 130
5. What to do next, in order 130
6. How to defend this 131

Chapter 20 — Replication: building the whole thing from an empty checkout 133
The rule that makes this chapter possible 133
0. What you need before you start 134
1. Get the code 134
2. The toolchain 135
3. Fetch the pinned firmware dependencies 136
4. Prove there is no artefact drift — before you build 136
5. Build the hub 137
6. Build the node and the wearable 137
7. Flash 138
8. Verify the chain, stage by stage 140
9. Reproduce the host-side checks 142
10. Bring up the dashboard 143
11. Reproduce the hardware documents 144
12. Reproduce the training — the optional path 144
13. The traps, and what each one looks like 147
14. The time and disk budget 148
15. The checklist 148

Chapter 21 — Glossary 150
A 150
B 150
C 150
D 151
E 152
F 152
G 150
H 152
I 152
J 153
K 153
L 153
M 153
N 154
O 154
P 155
Q 155
R 155
S 156
T 156
U 157
V 157
W 157
X 157
Y 158
Z 158
Acronyms at a glance 158
The project's own vocabulary, in one table 159

Appendix A — The project record, verbatim 160

Appendix B — Complete first-party source listings 281

Part I — The system

What CryNet is

CryNet is a monitor for an environment with more than one cot: a NICU bay, a maternity ward, a daycare, a twin nursery. Two microphone nodes sit with the cots, one at each. Each samples its own microphone continuously and decides for itself when something interesting has happened. When it does, it hands its two seconds of audio to a central hub over a short-range radio link. The hub reassembles the audio, runs a small neural network over it, and decides whether that sound is an infant cry. If it is, the hub names the cot on its display, raises an alert over Bluetooth — which a caregiver's wristband turns into a vibration — and writes the event to a record in its own flash, which a web dashboard reads so that a cry at three in the morning is still visible at nine.

The three design choices that everything else follows from:

1. **The nodes decide when to speak; the hub decides what it means.** A node that hears nothing interesting sends almost nothing. This is what makes the radio work at all with two of them sharing one channel, and what keeps a battery node alive for a whole shift.
2. **Only the hub runs the model.** One ESP32-S3 with 8 MB of PSRAM is a far better place for a neural network than two small battery-powered nodes, and it means the model can be retrained and re-flashed in one place instead of at every cot.
3. **The wearable is a dumb, reliable terminal.** It runs no model. It receives an alert and vibrates. Everything about it is about being trusted to say something the caregiver must not miss.

The three kinds of unit

Unit	Board	Role	Key numbers
Cot node ×2	ESP32-C3 **Pro Mini** + INMP441 I²S MEMS microphone	Samples audio at 16 kHz, tracks its own noise floor, freezes a two-second window when a cry crosses the trigger, sends it to the hub	GPIO4/5/6 for I²S; ESP-NOW channel 6; 286-fragment burst; 75 % flash, 50 % RAM of the C3's budget
Hub	ESP32-S3-DevKitC-1 (dual-core Xtensa LX7 @ 240 MHz, 512 KB SRAM, 8 MB PSRAM) + 2.8" ILI9341 TFT	Reassembles windows, runs the DSP front end and an INT8 CNN, arbitrates between cots, drives the panel, serves the BLE alert service, keeps the cry log, uploads it over Wi-Fi and posts presence	135 KB INT8 model; front end 41.7 ms, inference 101.8 ms per window; 14.3 % RAM, 77.2 % flash; 286/286 fragments, CRC-32 MATCH
Wearable	ESP32-C3 SuperMini + 0.96" SSD1306 OLED + coin ERM on a BC547 low-side switch	BLE client; shows the alert and which cot is crying; buzzes	GPIO7 → 2.2 kΩ → BC547 base; 22 Ω stall cap; 2.5 s haptic pulse train

Two cots, enforced in the link layer

The cot count is not a display setting. `NODE_LINK_MAX_NODES` is **2** and `HUB_DISPLAY_MAX_NODES` is **2**, and the hub's receive path drops any frame whose node id falls outside 1..2 *before* it can claim a slot:

```
/* This dedicated hub monitors only cot IDs 1 and 2.  Reject other
 * IDs before they can occupy a slot or influence alerts. */
if (node_id < 1 || node_id > NODE_LINK_MAX_NODES) {
    return;
}
```

That is `node_link.c`, at the top of the frame handler, ahead of the counter that says a frame was good. A third board powered up in the same room is not hidden from the panel — it is never heard, cannot occupy a slot, and cannot influence an alert. The nodes know the rule too: `CRYNET_COT_COUNT = 2` in the sketch, and a node whose id is outside 1..2 prints a warning at boot, because the hub will ignore every other number. Ids are set over the node's serial console with `id 1` and `id 2` and live in NVS, so they survive a reflash — which was checked on the bench, not assumed.

The reason to enforce it in the link layer rather than the panel is that a slot is a resource. Each one carries a 64 KB window buffer and a burst state machine. A hub that accepted a third id and then declined to draw it would still spend the memory, still arbitrate against it, and still let it hold the alert. Refusing the frame is the only version of "two cots" that is true everywhere.

The honest status

Three things must be said plainly before anything else, because they shape how every later number should be read.

The model has never heard a real infant cry. Every cry it has been tested against is a *recording played through a phone speaker* — compressed, band-limited and EQ'd by a small loudspeaker in a plastic case. The negatives are authentic (this room, this microphone), but the positives are a proxy. On that proxy the model is strong; on a real infant it is unproven. This is recorded as O-2 and it is the single largest accuracy risk in the project.

One unit has run from a cell; no runtime has ever been measured. The wearable ran on its **Li-Po pouch** during part of the 2026-11-20 bench session and kept working — so the battery path is not wholly untried. But the nodes and the hub have only ever been USB-powered, **no unit's battery life has been measured**, the boost module's own quiescent draw is unquantified, and the haptic buzz went missing while the wrist was on the pouch (a sagging 3.3 V rail is the live candidate). The runtime figures in the documents are *projections* from the regulator's behaviour and are labelled as such. This is O-1d, and Chapter 13 carries the arithmetic and the three rules the 5 V chain must obey (D-84).

Two smaller things belong in the same breath, because a defence should not be caught by them: the node's I²S zero watchdog is **committed but not flashed** (D-79 / O-1c — the boards are mid-solder), and the microphone has a **close-range blind spot** — a source loud enough to drive it to the rail produces a window the hub refuses outright, which was measured on 2026-11-20 and is recorded as O-1g.

The wrist, meanwhile, **names every crying cot** (D-86). The alert *packet* still names one — the loudest — because one cry must be one buzz; the *screen* reads the crying mask and lists them all. Two cots at once was staged on 2026-11-20 and the mask held at `0x03` for fifteen seconds.

Everything else described in this document was measured on real hardware.

What the system is *not*

- It is **not a certified infant monitor**. It is an academic prototype. Detection accuracy, radio delivery guarantees, battery life and alarm behaviour have not been established to any safety-critical standard, and it does not replace supervision.
- It is **not a cause classifier**. The model is binary: cry or no cry. It does not distinguish hunger from pain from tiredness. That is deliberate (D-1): the Bluetooth alert packet that the wearable was written against has one bit for it.
- It is **not a baby monitor that replaces a parent**. It is a second pair of eyes that wakes someone up.
- It has **no battery monitoring at all** (D-83). The battery sense and every status LED were designed, costed, and then withdrawn on purpose. Nothing in the system will tell a caregiver that a cell is nearly flat; the cells are watched by eye.

The system in one paragraph

Two ESP32-C3 cot nodes with INMP441 microphones sample a room at 16 kHz, each tracking its own running noise floor; when a sound crosses three times that floor the node freezes the two seconds centred on the crossing and streams them to an ESP32-S3 hub as a 286-fragment ESP-NOW burst with a per-fragment CRC-16 and a whole-window CRC-32. The hub reassembles the burst, refuses it outright if any sample saturated the microphone, computes a 64-band log-Mel spectrogram of the two seconds, standardises it, quantises it to INT8 and runs a 122,770-parameter CNN that returns P(cry); the window is announced only if the model is near-saturated, or if two consecutive windows agree and the second carries real energy and the right crest. A cry is held for eight seconds, the loudest cot keeps the alert unless another cot is clearly louder for a second and a half, and the whole system state is published over Bluetooth as a six-byte alert packet and an eight-byte state packet, which the wristband turns into a 2.5-second vibration and a face that names the cot. Every event — each cry episode, each deaf spell, each boot — is also written to a thirty-two-byte record in six megabytes of the hub's own flash, stamped with the wall clock where the hub knows it and marked untimed where it does not; when the hub is on a network it uploads those records to a dashboard — one offline copy, one hosted — which shows who is here, what happened, and when.

Chapter 2 — Architecture and the end-to-end path

Why the system is split the way it is

There are three plausible ways to build a multi-cot cry monitor on ESP32-class hardware, and the project chose the third. It is worth being able to argue all three, because the choice is the foundation everything else rests on.

Option 1 — every node runs the model. Each cot node captures, computes features, runs the network, and sends a one-byte verdict. Radio traffic is trivial. It fails for three reasons. The ESP32-C3 is a single-core RISC-V part at 160 MHz with 400 KB of SRAM and no PSRAM: the 81 × 64 feature plane, the tensor arena and the audio buffer together do not fit comfortably alongside a Wi-Fi radio and an I²S driver. Second, one copy of a model per cot is one copy per cot to keep in step: every retrain means reflashing battery units that are physically attached to cots. Third, arbitration between cots — *which* cot is crying, which one is louder — needs a place where both verdicts exist at once, and that place would have to be invented anyway.

Option 2 — every node streams continuously to the hub. The hub gets everything and decides everything. This is the simplest firmware and the worst system. Two nodes × 16,000 samples/s × 2 bytes = 64 KB/s of sustained radio traffic on a link whose measured practical throughput is roughly 31 KB/s for one node bursting alone. It does not fit by a factor of two, before contention — and one node streaming alone already uses the whole link. It also destroys battery life: the node's radio would be transmitting continuously.

Option 3 — the nodes decide *when*, the hub decides *what*. Each node runs a cheap energy detector against its own adaptive noise floor. Only when something crosses the floor does it freeze two seconds of audio and ship them. The hub — an ESP32-S3 with two cores at 240 MHz and 8 MB of PSRAM — does the expensive part once, for whichever cot asked. Radio traffic becomes bursty and rare, the model lives in exactly one place, and arbitration is natural because every verdict is computed in the same address space.

That is CryNet. The cost of the choice is that the node's trigger is a *gate on recall*: a sound the node never triggers on can never be classified, no matter how good the model is. This is not a hypothetical cost — it was measured, and it is the reason the adaptive trigger exists (D-63, Chapter 7).

The five hardware pieces

Piece	Part	Why this part
Cot node ×2	ESP32-C3 **Pro Mini**	Cheapest ESP32 with Wi-Fi (needed for ESP-NOW) and an I²S peripheral; 4 MB flash; native USB (`303A:1001`) on board. The owner's stock board, swapped in and verified live on 2026-11-02
Microphone ×2	INMP441 breakout	Digital I²S MEMS: no analogue path, no preamp, no ADC noise, no gain to calibrate. Fixed −26 dBFS sensitivity
Hub	ESP32-S3-DevKitC-1 N16R8	Dual core, vector instructions used by ESP-NN, 8 MB octal PSRAM for the tensor arena and audio, 16 MB flash
Panel	2.8" ILI9341, 320 × 240, SPI	Big enough for two full-width cot cards with a level bar and a signal bar each; a ubiquitous, well-documented controller
Wearable	ESP32-C3 **SuperMini** + SSD1306 128 × 64 OLED + coin ERM	The *same chip* as the cots, on the SuperMini board the cots used before the swap — so one toolchain and one FQBN still cover all three C3 units. BLE client, tiny display, a motor a caregiver cannot ignore

The path a cry takes, step by step

What follows is the whole system in sequence, with the real constants. Every stage named here has its own chapter.

1. Capture (node, continuous)

The INMP441 is clocked by the node's I²S peripheral at 16 kHz, 32-bit slots, left channel. Samples arrive in DMA-backed blocks and are pumped into a ring buffer of `int16` holding the last two seconds — 32,000 samples, 64,000 bytes. The top 16 bits of each 32-bit slot are the sample; the microphone produces 24-bit data left-justified in the slot, and the firmware keeps the top 16.

`capture_task` runs continuously. Nothing else on the node blocks it: this is the one job that must never miss a sample, because a gap in the audio is indistinguishable from a sound.

2. Level and floor (node, every 100 ms)

Every 100 ms the node computes the RMS of the most recent 100 ms of audio, `g_rms`. It maintains a noise floor `g_floor` that **falls immediately** when the room gets quieter and **rises slowly** when it gets louder — an exponential update, `g_floor += (g_rms - g_floor) * 0.002f`, a time constant of about 50 seconds. The trigger level is `g_level = 3.0 × g_floor`, clamped to `[0.005, 0.030]`.

The asymmetry is deliberate and is the single most important line in the node: if the floor rose as fast as it fell, a sustained cry would drag the floor up to its own level within a second or two and the node would stop triggering *in the middle of the cry*. A slow rise means a long cry keeps triggering; a fast fall means the system re-sensitises the instant a room goes quiet.

Two consecutive readings at or above `g_level` constitute a trigger. Two, not one, because a single 100 ms transient — a door catch, a chair leg — is not a cry.

3. Freeze (node)

A trigger does not freeze immediately. The node marks a **peak pending** and waits, so that the two seconds it eventually freezes are *centred* on the cry rather than ending at it:

- `PEAK_DELAY_MS` = 2000 ms — the cap for a cry that goes on;
- `PEAK_STOP_DELAY_MS` = 500 ms — if the loud streak breaks first (the cry stopped), freeze after only half a second, so the window holds the cry that just ended rather than the room behind it;
- `PEAK_EXPIRE_MS` = 4000 ms — the peak must still be inside the ring when the freeze happens.

`research/simulate_freeze.py` checked this stop-capped rule against the corpus: it catches as many cry episodes as the old peak-centred rule, and the hub announced no room window under any of the variants tested.

The freeze copies the ring into `g_frozen`, oldest sample first, and records `g_frozen_rms` (the window's own RMS with its mean removed — the same computation the hub's front end does), the CRC-32 of the exact bytes that will be uploaded, and a new burst sequence number.

Three timers then protect that window:

Timer	Value	Job
`FROZEN_HOLD_MS`	3000 ms	pure resend coverage after the upload; cut from 4000 when the burst got faster (D-71)
`TRANSFER_HOLD_MS`	4000 ms	protection while the burst is in the air, **refreshed on every fragment sent**
`FROZEN_EXPIRE_MS`	20000 ms	how long an unserved window may sit before it is dropped

If `g_frozen_rms >= 0.05` — five times the floor — the node enters **cry mode** (D-69): it will send one window per cry episode rather than one per trigger, exiting when `g_rms < 0.01` and the loud streak is zero. This stops a single long cry from producing a dozen redundant bursts.

4. Offer (node → hub, telemetry at 10 Hz)

The node broadcasts a 32-byte telemetry frame ten times a second on ESP-NOW channel 6. It carries the node's id, sequence number, `g_rms`, `g_floor`, `g_level` (as milli-RMS), the frozen window's own RMS and peak, and counters. Telemetry is the node's *advertisement*: "I have window *n*, and here is how loud it was."

Telemetry is **broadcast**; the 286-fragment burst is **unicast to the hub**, whose MAC the node learns from the first request it receives — a broadcast destination is filtered more aggressively by the receiver's radio than a unicast one. Broadcast telemetry at 10 Hz costs about 3.2 KB/s of airtime per node and is the heartbeat the hub's liveness display is built on.

5. Pull (hub → node)

The hub reads the telemetry. If the advertised window is new, and its RMS crosses the trigger level the node reported (falling back to `CRYNET_TRIGGER_RMS` = 0.030 for an older node), and the hub is not already busy, it sends a REQUEST naming that sequence number. This is the *pull* model: the hub, not the node, decides when the channel is used for a 64 KB transfer. That is what makes two nodes share one channel without a collision-avoidance protocol.

5b. Heartbeat and the two link readings (hub ↔ node, 1 Hz)

The hub broadcasts a **heartbeat once a second** — `node_link_send_heartbeat()`, a twelve-byte frame of type 4 that asks for nothing. It exists for one reason: a node cannot measure the strength of a link it never receives on. The node records the RSSI of the last frame *the hub* sent and prints its own four-bar view of the link on its serial console.

Only the hub's own frame types count towards that reading — a REQUEST (3), a HEARTBEAT (4) or a RESEND (5). Everything on channel 6 is broadcast, so a cot also hears the *other* cot's telemetry and window fragments; taking the RSSI of any frame would report the loudest cot in the room and call it the hub. That distinction is a bug that was caught before it shipped, and a host test now holds it (`research/test_heartbeat_framing.py`).

In the other direction the hub records the RSSI of each cot's telemetry (`slot->tel_rssi`), carries it through `link_stage` into the display state, and draws it as a four-bar glyph on that cot's card. Both ends therefore show a link reading, and they are readings of *different* directions — which is why they disagree by tens of dB on the bench and why that disagreement is not a fault.

6. Burst (node → hub, 286 fragments)

The node splits the 64,000-byte window into 286 fragments of 224 payload bytes each (the ESP-NOW maximum frame is 250 bytes; the protocol header and CRC-16 take the rest). Each fragment carries:

- the node id and window sequence number,
- the fragment index and total,
- a **CRC-16/CCITT-FALSE** over that fragment's payload,
- the payload bytes.

The last fragment also carries the **CRC-32 (IEEE 802.3)** of the whole 64,000-byte window — the same polynomial as `zlib.crc32`, which is what lets a host-side Python tool verify a capture without reimplementing anything.

Fragments are paced at roughly 4 ms. The transmit queue is 10 deep with at most 4 frames in flight; `on_send` releases a slot. The node's radio runs at **8.5 dBm**, not the 20 dBm default: at 20 dBm the transmitter, sitting centimetres from the microphone, put its own noise into the audio that was about to be judged (D-65).

7. Reassemble and repair (hub)

The hub allocates a 64,000-byte buffer per burst and a 286-bit received bitmap. Each arriving fragment is CRC-16 checked; a bad fragment is dropped and its bit left clear. When fragments stop arriving for `FRAG_STALL_MS` (100 ms, adaptive up to 1500 ms) the hub sends a **RESEND** frame — type 5, a 54-byte request carrying a 36-byte bitmap of exactly what is missing. It sends it three times, because the request itself is not acknowledged. The node replays only the missing fragments, and deduplicates a request only when it names **the same window and the same fragments** and arrives within `RESEND_DEDUPE_MS` = 250 ms. Up to `RESEND_MAX_ROUNDS` = 10 rounds. (That dedupe rule was once 800 ms and one-sided, and it silently destroyed every burst that lost a single fragment — see Chapter 7.)

When all 286 bits are set the hub computes the CRC-32 of the reassembled window and compares it with the value the node sent. `CRC32 MATCH` in the log is the proof that the two seconds of audio in the hub's memory are bit-for-bit the two seconds the microphone heard. **Measured: 286/286 with CRC-32 MATCH, repeatedly, including with a BLE client connected.**

If the node advertises a *newer* sequence number mid-burst, the hub abandons the old burst rather than chasing a window the node has already overwritten.

8. Refuse if clipped (hub)

Before any DSP: if any sample in the window has magnitude ≥ `CRYNET_CLIP_PEAK` = 31000 (of 32767, i.e. 0.946 of full scale), the window is **refused outright**. No features, no inference, no verdict.

The reason is physical. The INMP441 is a fixed-gain digital microphone: −26 dBFS ±3 dB, and its L/R pin selects the channel, not the gain. When the sound pressure exceeds what it can represent it flat-tops *inside the microphone*, before the 16-bit window ever sees it. A clipped window is not a loud cry — it is a distorted waveform whose harmonics are an artefact of the clipping, and the model has never been trained on that artefact. There is no software cure. The correct behaviour is to refuse the corrupted evidence and let the *other* cot — the bystander that heard the same cry from further away, unclipped — announce it. That is exactly what happens on the bench.

9. Front end (hub, 41.7 ms)

The frozen contract, D-11, executed in C:

1. `int16 pcm[32000]` → float, divided by 32768.
2. Subtract the mean, computed as an exact `int32` sum before division (so the DC removal is identical to Python's, not merely close).
3. Zero-pad 256 samples at the front and 512 at the rear — this reproduces librosa's `center=True` reflect-padding count for this configuration and gives exactly 81 frames.
4. 81 frames of 512 samples, hop 400, each multiplied by a **symmetric** Hann window.
5. Real FFT → 257 magnitude-squared bins.
6. 64 Mel bands, **slaney** normalisation, 20 Hz–8 kHz, 497 non-zero weights.
7. `10·log10(max(P, 1e-10))`, floored at −80 dB.
8. Per-window CMVN: subtract the mean of the whole 81 x 64 plane and divide by its standard deviation (plus 1e-6), so the plane arrives at the network with mean 0 and variance 1.
9. Quantise to INT8 with the model's input scale and zero-point.

Every one of those constants is a `_Static_assert` away from being wrong: `crynet_dsp.c` carries **eight** compile-time guards (lines 35–42) that fail the build if the deployed model's input or output scale, either zero-point, the silence gate, the sample count, the frame count or the mel band count drift from what the C code assumes.

Measured parity against the Python front end: maximum absolute difference 6.9×10^{-5} over four probe windows. That is float rounding, not a disagreement.

10. Inference (hub, 101.8 ms)

The INT8 model — 137,384 bytes, 122,770 parameters — runs on TensorFlow Lite Micro with ESP-NN's SIMD kernels, in a 262,144-byte tensor arena of which 108,328 bytes are actually used, allocated in PSRAM. Output: two INT8 values, dequantised and softmaxed to `[P(no_cry), P(cry)]`.

The output saturates at **P(cry) = 0.9961** for a clear cry. That number matters enormously for the alert policy: it means a threshold of 0.99 is *just* below saturation, and it means a threshold change cannot separate a confident true cry from a confident false one — they carry the identical value.

11. Announce (hub)

A verdict is not an alert. The announcement rule (D-64, D-74) is:

```
audible      = rms >= CRYNET_ANNOUNCE_MIN_RMS          (0.006)
unmistakable = audible && !clipped
               && (rms >= 0.080 ? p_cry >= 0.99 : p_cry >= 0.99)
               && (rms >= 0.020 || crest >= 3.5)
confirmed    = unmistakable || (cry_run >= 2 && rms >= 0.020)
```

In words: one window announces on its own only if the model is essentially saturated *and* the window carries real energy or a cry-like crest factor. Anything less is a *maybe*, and a maybe must be confirmed by a second consecutive window. A cry lasts seconds and its windows agree; a noise fluctuation that happens to look like a cry does not repeat.

This rule exists because the adaptive trigger, by design, asks the model about near-silent windows — and the model is not a reliable judge of those. The same quiet room produced verdicts of 0.0000 and 0.9766 for windows that differ by nothing a person could hear. The rule, not the model, is what makes that safe.

12. Hold and arbitrate (hub)

A confirmed cry is held for `CRY_HOLD_MS` = 8000 ms. The alert names the loudest crying cot, and it takes a **1.5× loudness margin sustained for 1500 ms** (`COT_SWITCH_MARGIN`, `COT_SWITCH_DWELL_MS`) before the panel will switch its primary label to a different cot. Without hysteresis, two cots hearing the same cry at similar levels make the label flicker between them — which is exactly the symptom the dwell timer removes. With two cots this is not an edge case: one baby, two microphones, one cry, and the two RMS readings are within a few dB of each other whenever the cots are a metre apart.

The alert therefore names **one** cot — the loudest — and that is deliberate: one cry must be one buzz. The state characteristic still carries the whole crying bitmask, so the information is not lost — and since **D-86** the wrist's alert face reads that mask and lists every crying cot, unioned with the cot the alert named. **The packet and the buzz stay one-cot; the screen does not.**

13. Publish (hub → wearable, BLE)

Two GATT characteristics on service `19b10000-e8f2-537e-4f6c-d104768a1214`:

- **Alert** `19b10001-…`, 6 bytes: `cot_id`, `alert_type` (1 = cry, 2 = stopped, 3 = deaf), `confidence_pct` (u16), `event_id` (u16).
- **State** `19b10002-…`, 8 bytes: `cots`, `crying_mask` (bit 0 = cot 1), `deaf_mask`, `confidence_pct`, `started_at_s` (u32).

The alert's `event_id` is now the cry log's own number for the episode — the same number the dashboard stores — so the buzz, the panel and the web record can be matched up after the fact (Chapter 12).

This contract is **frozen legacy**: the wearable was written against it before the rebuild, and it was kept byte-for-byte so the two halves could be developed independently. That is a real engineering decision with a real cost — `alert_type` has one value for "cry", which is why the model is binary (D-1).

14. Announce to the human (wearable)

The wearable's notify callback parses the packet, sets the alert face, and starts a 2.5-second haptic pulse train on GPIO7 through a BC547 low-side switch, with a 22 Ω stall cap in the motor's positive lead and a 1N5819 flyback diode across it (D-77, D-85). The alert face holds for 60 seconds and names the cot the alert carried — and the `event_id` it shows is the cry log's record number, so the buzz can be traced to the dashboard's entry for that exact cry.

15. Record and report (hub → flash → dashboard)

Independently of the alert, the episode is written to the hub's flash as a thirty-two-byte record — with a wall-clock stamp when the hub knows the time, and `epoch_s = 0` when it does not — and, when the hub is on a network, uploaded to the dashboard in batches, alongside a presence post every 30 seconds that says who is in the room right now. None of this is on the detection path: with no network, the only difference is that the records wait in flash. Chapter 12 is this subsystem's chapter.

Timing budget

Stage	Time	Source
Trigger decision	up to 200 ms (two 100 ms readings)	node design
Telemetry to hub	≤ 100 ms (10 Hz)	node design
Burst, 286 fragments	2.06–5.19 s, median 3.55 s	measured, D-61/D-62
Front end	41.7 ms	measured on chip
Inference	101.8 ms	measured on chip
Request → verdict, end to end	2.23–5.35 s, mean 4.03 s	measured, D-62
Panel shows the cry	≈ 2.5 s after the sound in the good case	measured, D-61

The burst dominates. It is not the pacing that sets its length — it is contention on channel 6, which is shared with every Wi-Fi network in the building. That is why the same firmware gives 2.06 s on a quiet channel and 5.19 s on a busy one, and why the honest figure is a range and not a number.

What runs where, in memory

Allocation	Size	Where	Why
Node audio ring	64,000 B	internal SRAM	DMA target, must be internal
Node frozen window	64,000 B	internal SRAM	held 4 s for resend
Hub burst buffer	64,000 B per active burst	PSRAM	large, not latency-critical
Hub tensor arena	262,144 B reserved / 108,328 B used	PSRAM	the single largest allocation
Hub feature plane	81 × 64 floats	PSRAM	intermediate
Hub framebuffer	320 × 240 × 2 = 153,600 B	PSRAM	full-frame redraw
Hub boot probe buffers	large, temporary	PSRAM, **freed after probes**	D-39/D-40

The boot-only buffers are the reason the hub's RAM figure went from **59.9 % (196,140 B) to 12.9 % (42,236 B)** — and since then the cry log, the clock and the Wi-Fi stage moved it again: the current build reports **14.3 % RAM (46,788 B), 77.2 % flash (1,619,615 B)**, the growth being the network stack, the certificate bundle and the log, none of it on the detection path. The boot probes exist to prove the DSP and the model agree with the host at startup; once proven they are dead weight, and freeing them is what leaves room for the radio, Bluetooth and Wi-Fi to coexist.

Initialisation order is **clock and log → radio → Wi-Fi uploader → Bluetooth → display → console**, and it is not arbitrary: each of those subsystems takes a large, permanent bite of *internal* heap, the display's framebuffer must go to PSRAM, and NimBLE's pools and the Wi-Fi driver's buffers come out of the *same* internal RAM — when Bluetooth claimed its memory first, the Wi-Fi driver ran out mid-init and the ESP-NOW link never started. So the radio and the uploader both take what they need while the heap is whole, and BLE lives on what is left. (The free-heap ladder through that sequence is printed on the boot banner at every start; the figures quoted in earlier editions predate the Wi-Fi stage joining the boot.) Getting the order wrong starves the last subsystem to start, which is how the BLE stack failed to initialise in an early build.

Chapter 3 — The data research: what was chosen, what was rejected, and why

The question the research had to answer

A cry detector is not a cry *recogniser*. The hard part is not recognising a cry — a cry is a loud, voiced, harmonic-rich sound with a characteristic rising-falling pitch contour, and almost any classifier can learn it. The hard part is **not alarming on everything else in a nursery**: an adult voice, a television, a radio, a cat, a door, a cough, a snore, another baby in the next cot.

So the corpus was chosen backwards from the confusion set. The question was never "where do I get baby cries" — it was "what does a nursery actually contain, and do I have training examples of every one of those things".

The primary corpus: ICD

`hoyyu1/infant-cry-detection` on Hugging Face. 28,478 clips, every one of them **16 kHz mono, exactly 5.00 seconds**, shipped in five folds. That uniformity is not a convenience — it is the reason this corpus was chosen over the alternatives. No resampling, no channel mixing, no length normalisation, no per-clip decisions. The signal contract can be defined once and applied to every clip identically.

	Clips
Positives (cry)	**13,330**
EnesBabyCries	5,026
CryCeleb2023	4,552
iFLYTEK	3,752
Negatives (non-cry)	**15,148**
VoxCeleb (adult speech)	4,963
DASEE (domestic noise)	4,506
Human non-speech (cough, laugh, sneeze…)	2,076
Cat meows	1,953
Silence	1,650
Total	**28,478**

Three independent cry corpora means hundreds of different infants recorded through three different chains. That matters: a model trained on one corpus of babies learns that corpus's microphone as much as it learns crying. Three chains break that shortcut.

The negative mix is the confusion set, deliberately: speech (the commonest sound in a room with adults), domestic noise (the room itself), human non-speech vocalisations (the sounds *closest* to a cry — a cough or a laugh is voiced and harmonic too), cat meows (the classic false alarm of every commercial baby monitor, and

acoustically a near-twin of an infant cry), and silence (so the model has an explicit "nothing here" class rather than an undefined region).

The supplementary corpora

Corpus	Clips	Role	Notes
Baby Cry 4-class	4,871	extra positives, different provenance	needs 3,103 / physical_pain 774 / scared 684 / burping 310. All four map to *cry*: the system is binary (D-1)
ESC-50	1,960 negatives + **40 held out**	environmental negatives; the 40 `crying_baby` clips are the *only* cross-corpus real-cry test	ESC-50 contains **no music and no speech** — a fact that later mattered enormously
Snoring	1,000	a sleeping adult, the other thing in a nursery at 3 a.m.	
GTZAN (added later, D-51)	1,000	**music**, ten genres	added *because a measurement demanded it*, not by foresight
Loud speech	recorded in this room	in-domain hard negatives	captured on the actual INMP441

The music story — the best argument in the project

This sequence is worth being able to tell in full, because it is the clearest demonstration that the project measures rather than assumes.

1. The first false-alarm map over ESC-50 (2026-10-31) showed the model rejecting speech, household noise and human non-speech well — **1.6 %**.
2. But ESC-50 contains **no music at all**, and neither did any training corpus. So "the model tells a cry from music" was not merely unmeasured — it was **untrained**.
3. A nursery has a radio, a television and a phone playing something. Music is a first-class confusion class here, not an afterthought.
4. GTZAN was fetched (1,000 clips, ten genres) and `research/eval_music.py` ran the *deployed* INT8 model over it. **79 of the 965 clips that reach the model alarmed — 8.19 %**, three times the rate for any other kind of sound.
5. The per-genre breakdown made the cause obvious: jazz **14.3 %**, blues **13.0 %**, pop **12.0 %**, disco **10.0 %**, rock **10.0 %**; reggae 2.0 %, classical 2.9 %, country 3.0 %, metal 4.0 %. A saxophone and a sung vowel are voiced, harmonic-rich sounds. In the log-Mel domain they *are* cries, as far as the features are concerned.
6. **A threshold change could not fix it.** The INT8 output saturates at P(cry) = 0.9961: **45 of those 81 music alarms and 34 of the 36 true cry detections carried exactly the same value.** No threshold separates them. This is the single most important number in the whole evaluation, and it is the reason the answer was a training round rather than a config change.
7. GTZAN went into the training mix (D-51), split by clip so that 25 % of it is held out as *music the model has never heard*. The notebook was re-run on Kaggle, the model redeployed and re-verified 4/4 on the board (D-58).

8. **Music false alarms: 8.19 % → 0.21 % (2 of 965).** ESC-50 non-cry alarms halved, 2.64 % → 1.18 %. ESC-50 crying-baby recall unchanged at 36/40. The quiet room still silent, 0/236.
9. **The cost was measured too, and it is real:** per-window recall on the phone-speaker cry sessions dropped — session 1 from 0.867 to 0.800, session 4 from 0.724 to 0.592. A sustained cry still trips many consecutive windows, so *episodes* are still caught; but the trade is stated, not hidden.

That is the whole loop: an unmeasured assumption, a measurement that broke it, a diagnosis that ruled out the cheap fix, a data change, a re-measurement, and an honest account of what got worse.

What was rejected, and on what evidence

Being able to say *why not* is worth more in a defence than being able to say why.

Rejected	Reason
`mennaahmed23/baby-cry*`	**Contaminated.** Its "silence" class measures RMS **0.22** with a constant **1225 Hz** tone. That is not silence; it is a synthesised signal. Training on it teaches the model that a pure tone is silence — the exact opposite of what is needed. Found by measuring the audio, not by reading the dataset card
`rahult25/five-class`	Its classes are **pitch-shifted derivatives** of the same source clips. Training and testing on it measures the model's ability to recognise a resampling artefact, not a cry. Inflated scores, zero transfer
ICSD	Gated access; not obtainable within the project's timeframe
AxonData	Paid
Baby Chillanto	Request-only, no guaranteed turnaround. A classic in the literature, but unusable on a deadline
AudioSet / FSD50K	Redundant with ESC-50 + DASEE for this confusion set, at ten times the download and processing cost. (One FSD50K speech archive was kept as a *single irreplaceable copy* — its GitHub release no longer carries the asset)
Raw DonateACry	8 kHz, heavily skewed in age and duration. Upsampling 8 kHz to 16 kHz invents the top octave the model would then learn to rely on

The pattern: **every rejection is a measurement or an access constraint, never a preference.** The contaminated corpus was found by computing RMS and looking at the spectrum. The derivative corpus was found by comparing waveforms. That is the standard the rest of the project holds to.

The level asymmetry — a finding that changed the firmware

While assembling the corpus, one measurement stood out: **ICD cries have RMS ≈ 0.006; the negatives have RMS 0.3–0.46.** The cries are, on average, *sixty times quieter* than the non-cries in the same dataset. That is an artefact of how the corpora were normalised by their authors, and it is a trap: a model trained on that data without level normalisation would learn "quiet ⇒ cry" and achieve excellent validation numbers for entirely the wrong reason.

Per-window CMVN makes the model level-invariant, which is why it is in the frozen contract and not an option. The mean and standard deviation of the whole 81 x 64 log-Mel plane are removed, per window, so absolute level carries no information into the network.

The same finding exposed a real bug in the legacy firmware. The old hub fired an audio pull only at **RMS ≥ 0.18**. Real cries in this data sit around 0.006. **A soft cry never reached any classifier at all** — the system was not misclassifying quiet cries, it was never being asked about them. The rebuild's trigger is 0.030 and, since

D-63, adaptive down to 0.005. Rejecting false triggers is exactly what the model was trained for; gating them out before the model was throwing away the model's whole purpose.

How the corpora are turned into training rows

A clip is 5.00 s; the contract's window is 2.00 s. Each clip therefore yields several windows, and the labelling rule is **"any cry present"**: a window is positive if cry audio is present anywhere in it. That is the right rule for an alarm — a caregiver needs to know a cry started, not that a majority of the last two seconds was crying.

Three properties of the assembly are load-bearing:

1. **Splits are by clip, never by window.** `split_by_clip()` shuffles clip identifiers and holds out whole clips. Splitting by window would put two overlapping windows of the same clip in train and test, and the resulting score would be meaningless.
2. **Augmented views exist only in TRAIN.** They are generated from training clips after the split, so an augmented copy of a test clip can never leak into training.
3. **Byte-identical windows are de-duplicated** by MD5 over the float16 feature bytes, with the earlier part winning. Silence clips in particular produce many identical windows; without de-duplication the model would be trained on thousands of copies of the same zero frame.

The split

Split	Contents
TRAIN	ICD folds 1–3 (clean + augmented) + 70 % of cry4 + 75 % of ESC-50 non-cry + 75 % of Snoring + 75 % of the music + 75 % of the loud speech
VAL	ICD fold 4
TEST	ICD fold 5
CROSS	the held-out 30 % of cry4, 25 % of ESC-50, 25 % of Snoring, 25 % of the music, 25 % of the speech, **plus all 40 ESC-50 `crying_baby` clips**

CROSS is the number to quote in a defence. It is *unseen corpora and unseen babies*, including real crying-baby recordings the model has never been trained on in any form. VAL and TEST are honest held-out folds of the primary corpus; CROSS is the honest estimate of what happens in a new room.

Augmentation

Two stages, in two different domains, for two different reasons.

Waveform domain (`augment_window` , applied when building augmented views):

```
y = x * 10 ** (uniform(-18.0, 6.0) / 20.0)     # random gain, -18..+6 dB
if peak > 0.98: y *= 0.98 / peak               # never clip
if random() < 0.35:                            # reverberation
    y = fftconvolve(y, random_RIR)[:N_SAMPLES]
if random() < 0.5:                             # real-noise mix
    snr_db = uniform(3.0, 25.0)
    a = sqrt(ps / (pn * 10 ** (snr_db / 10)))
    y = y + a * noise; renormalise if peak > 0.99
```

The gain range is wide (24 dB) on purpose: the microphone's sensitivity is fixed and the distance to the infant is not. The noise is **real recorded noise**, not synthesised white noise, because the thing the model must reject is a real room. The room impulse responses make the model tolerant of different rooms and cot positions.

Feature domain — SpecAugment, applied per batch on the GPU:

```
for _ in range(2):   # two frequency masks, width 1..8 Mel bins
for _ in range(2):   # two time masks, width 1..12 frames
```

Implemented in pure TensorFlow ops rather than `tf.py_function`, because a `py_function` erases the static tensor shape and Keras can then not infer the batch shape. That is a small detail with a large consequence, and it is the kind of thing a defence is made of.

The memory constraint that shaped the code

The training machine has **16 GB of RAM and no page file**. An allocation that does not fit is a hard failure, not a slowdown. The feature matrix is roughly 900 MB, and the first version of the split cell built it by concatenating copies — four live copies at once — and died with an allocation error before training ever started.

The rewrite is visible in the code and is worth knowing:

- one **preallocated** destination array, filled a chunk at a time (`fill_rows`, chunk = 2048);
- every source matrix released the moment it falls out of use;
- de-duplication done **per part** rather than over one giant concatenated array (`dedupe_parts`);
- features stored as **float16**, halving both the matrix and the hashing cost;
- a **bounded shuffle buffer of 8192** windows rather than a full-corpus shuffle — each buffered element is a whole 81 × 64 window, so a 32k buffer would hold ~340 MB on its own;
- hard examples **mixed in by weighted sampling** (`weights=[1.0, 0.014]`) rather than copied into a second matrix.

None of this changes what the model learns. All of it is the difference between a run that finishes and a run that does not.

Getting the data

The notebook downloads roughly **4.6 GB** of audio, resumable via `huggingface_hub`, into `research/_cache/`. Kaggle is the training host because it provides a GPU and because the download is fast there; the same notebook runs locally in `FAST` mode against a stub cache for pipeline validation. Chapter 20 gives the exact commands.

Chapter 4 — The signal contract: every constant and why it is that value

What "frozen contract" means here

The model is trained in Python on a workstation and executed in C on a microcontroller. Those two implementations must compute **the same function**, or the model on the chip is being fed something it was never trained on and its accuracy is a fiction. "Frozen" means the entire path from microphone sample to network input is written down once, with no free parameters, and both implementations are tested against each other.

This is D-11, and it is the most important decision in the project after the architecture itself.

The chain

```
int16 pcm[32000]                        2.000 s @ 16 kHz mono
  -> subtract mean                      DC removal (the INMP441 sits at ~+18000 counts)
  -> / 32768                            normalise to [-1, 1]
  -> zero-pad 256 front / 512 rear
  -> 81 frames, 512-pt symmetric Hann, hop 400
  -> |rfft|^2                           257 bins
  -> 64 Mel bands (librosa slaney, 20 Hz - 8 kHz)
  -> 10*log10(max(P, 1e-10))            floored at -80 dB
  -> per-window standardisation         (mean 0, std 1 over the whole plane)
  -> int8 quantise                      q = round(v/scale) + zero_point
```

The constants

Name	Value	Why
`SR`	16000	The INMP441's comfortable rate; every chosen corpus is already 16 kHz, so nothing is resampled. Nyquist 8 kHz covers a cry's fundamentals (300–600 Hz) and its first ten harmonics
`WIN_SEC`	2.0	Long enough to contain a full cry expiratory cycle and its pitch contour; short enough that the alert is not stale. A neonatal cry's expiratory phase is typically 0.5–1.5 s
`N_SAMPLES`	32000	2.0 × 16000
`N_FFT`	512	32 ms at 16 kHz. Frequency resolution 31.25 Hz — fine enough to resolve cry harmonics, coarse enough that a frame is quasi-stationary
`HOP`	400	25 ms, i.e. 112 samples (21.9 %) of overlap between consecutive 512-sample frames; gives exactly 81 frames with the chosen padding
`N_FRAMES`	81	(32000 + 256 + 512 − 512) / 400 + 1 = 81
`CENTER_PAD`	256	N_FFT / 2 — reproduces librosa's `center=True` framing
`TAIL_PAD`	512	makes the frame count land exactly on 81 rather than 80
`N_MELS`	64	Perceptual spacing; 64 bands over 20 Hz–8 kHz gives fine resolution where cry harmonics live and coarse resolution up top. 128 was the legacy value and doubles the input for no measured gain
`FMIN` / `FMAX`	20 / 8000	20 Hz discards rumble; 8 kHz is Nyquist
Mel scale	**slaney**	librosa's default: linear below 1 kHz, logarithmic above. **HTK** (the legacy firmware's choice) is logarithmic throughout. They are *different filterbanks* — using HTK weights against a slaney-trained model is a silent accuracy disaster
Mel weights	**497 non-zero**	stored sparsely as offsets + starts + ends, so the C filterbank loop touches 497 multiply-accumulates per frame, not 64 × 257 = 16,448
`LOG_EPS`	1e-10	floor inside the log, so a silent band gives −100 dB before the −80 dB clamp rather than −∞
`DB_FLOOR`	−80	clamps the dynamic range so silence does not dominate the standardisation
`CMVN`	True	per-window mean/variance removal — see below
`SILENCE_RMS_GATE`	0.0015	below this the window is declared silence and skipped entirely
`TARGET_RECALL`	0.95	the calibration target that picks the alarm threshold
`SEED`	1337	every RNG in the pipeline is seeded from this

Why each step is not optional

DC removal, and why the sum is an integer

The INMP441's output sits at roughly **+18000 counts**, not zero. Without DC removal the spectrum has an enormous zero-frequency component that leaks into every Mel band through the window's sidelobes.

The sum is accumulated in `int32_t`, and the comment in the source explains exactly why:

> A float32 accumulator loses the low bits badly at this magnitude: summing 32,000 samples of 18000 in float32 gives 575,726,660 instead of 576,000,000, so the "mean" comes out 8.5 counts low and 8.5 counts of DC survive. That is invisible in a normal signal but fatal for the DC-offset probe, where it leaves a constant residual that the per-window standardisation then blows up into a completely different feature vector.

`int32` holds the sum of 32,000 `int16` samples exactly (at most 1,048,576,000). It is also *faster*: integer adds are native, while the double-precision alternative would be emulated in software on the ESP32-S3.

This is a genuinely excellent defence answer. It is a case where the obvious implementation (`float sum`) is wrong, the wrongness is invisible on real audio, and it was caught by a *synthetic probe* designed to expose it.

The padding, and why there is no padded buffer

Frame `f` covers padded samples `[f·400, f·400 + 512)`; padded sample `p` holds PCM sample `p - 256`, with zeros outside the signal. The C code builds each frame **directly from the caller's PCM** with that index arithmetic, rather than materialising a padded copy. The copy was 128 KB of working set and bought nothing.

The window function must be *symmetric* Hann

`numpy.hanning` / `scipy.signal.windows.hann(N, sym=True)` and the *periodic* Hann used by some FFT libraries differ by one sample of phase. Over 81 frames that difference is small but systematic, and it is exactly the kind of small systematic difference that costs a few percent of accuracy for no visible reason. The contract names the symmetric one, and the C table is generated from it.

The FFT

An iterative radix-2 FFT, N = 512, in place, with the twiddle factors computed by **recurrence** (`cr, ci` rotated by `wr, wi` each butterfly) rather than by a lookup table or a `cosf` per bin. The recurrence is kept *identical to the reference implementation* so the two agree bit-for-bit on the same input — accumulated rounding in a twiddle recurrence is a real source of divergence between two otherwise-correct FFTs.

The Mel filterbank

librosa's **slaney** normalisation: each triangular filter is scaled by `2 / (f_hi - f_lo)`, so filters have equal *area*, not equal peak. HTK normalisation does not do this. The weights are precomputed in Python and emitted as three arrays — `CRYNET_MEL_OFFSET`, `CRYNET_MEL_START`, `CRYNET_MEL_END` — plus the 497 non-zero `CRYNET_MEL_WEIGHTS`. The C loop is then:

```
for (int m = 0; m < 64; m++) {
    float acc = 0.0f;
    const float *wts = &CRYNET_MEL_WEIGHTS[CRYNET_MEL_OFFSET[m]];
    for (int b = CRYNET_MEL_START[m]; b < CRYNET_MEL_END[m]; b++)
        acc += power[b] * wts[b - CRYNET_MEL_START[m]];
    mel[m] = acc;
}
```

Log, floor, and why −80 dB

`10·log10(max(P, 1e-10))`, then clamped at −80 dB. The clamp matters because of the *next* step: a band at −100 dB in an otherwise loud window would drag the window's standard deviation up and squash everything else

towards zero. The floor bounds the dynamic range the standardisation has to cope with.

Per-window standardisation — the level-invariance guarantee

```
mu = mean(mel_out[0 .. 81*64));
sd = sqrt(variance(...)) + 1e-6f;
mel_out[i] = (mel_out[i] - mu) / sd;
```

Note precisely: this is over the **whole 81 × 64 plane**, one mean and one standard deviation per window — not per Mel band. The `+1e-6` guards the all-silence case, where the variance is zero.

This is what makes the model level-invariant, and it is a direct response to a measured property of the data: ICD cries sit at RMS 0.006 and its negatives at 0.3–0.46. Without standardisation the network can learn "quiet ⇒ cry" and score beautifully for entirely the wrong reason.

Quantisation

```
v = nearbyintf(mel[i] / CRYNET_IN_SCALE) + CRYNET_IN_ZP;
clamp to [-128, 127];
```

`nearbyintf` — round-half-to-even under the default rounding mode — matches the converter's own quantisation, so the host parity test and the chip agree on ties as well as on everything else.

The eight compile-time guards

Because the generated header `crynet_tables.h` may only be included in **one** translation unit (file-scope `const` arrays have external linkage in C, so a second inclusion is a link error), the firmware mirrors a few generated constants under its own names. That duplication is a drift hazard, so it is fenced:

```
_Static_assert(CRYNET_INPUT_SCALE       == CRYNET_IN_SCALE,      "input scale drift");
_Static_assert(CRYNET_INPUT_ZERO_POINT  == CRYNET_IN_ZP,         "input zero-point drift");
_Static_assert(CRYNET_OUTPUT_SCALE      == CRYNET_OUT_SCALE,     "output scale drift");
_Static_assert(CRYNET_OUTPUT_ZERO_POINT == CRYNET_OUT_ZP,        "output zero-point drift");
_Static_assert(CRYNET_SILENCE_RMS_GATE  == CRYNET_SILENCE_RMS,   "silence gate drift");
_Static_assert(CRYNET_N_SAMPLES == 32000, "sample count drift");
_Static_assert(CRYNET_N_FRAMES  == 81,    "frame count drift");
_Static_assert(CRYNET_N_MELS    == 64,    "mel band count drift");
```

If a new set of Kaggle artefacts changes any of these values, the firmware fails to compile instead of silently classifying with stale numbers. That is the difference between a bug that ships and a bug that cannot.

Where the working set lives, and the 3× speed-up

The DSP context allocates only the spectrum scratch:

Buffer	Size
power spectrum	257 × 4 = 1,028 B
Mel band accumulator	64 × 4 = 256 B
FFT working array	512 × 8 = 4,096 B
Total	**≈ 5.4 KB**

It lives in **internal SRAM, not PSRAM**, and the reason is measured: PSRAM on the ESP32-S3 is external memory reached over SPI. The FFT's butterfly passes and the Mel filterbank's bin loop stream through these buffers 81 times per window. Leaving them in PSRAM made the front end cost **135 ms**; moving them to internal SRAM, which is plentiful once the 128 KB padded-signal copy is gone, brought it to **41.7 ms**.

The general rule the project follows: **large and cold goes to PSRAM; small and hot stays internal.** The tensor arena (262 KB, touched by the interpreter) and the framebuffer (153 KB, written once per redraw) are PSRAM; the FFT scratch (5.4 KB, touched 81 times per window) is internal.

How parity was proved

Two independent tests, both still runnable:

1. `firmware/tools/gen_probe_refs.py` regenerates `crynet_probe_refs.h` from the *notebook's own front end* — four probe windows with their expected 81 × 64 feature planes. The hub computes the same four at boot and compares. **Maximum absolute difference: 6.9×10^{-5}.**
2. `research/test_c_parity.py` runs the C front end on the host against the Python one on the same PCM, so a divergence can be debugged on a workstation rather than on a board.

The probes are not random audio. They include a **DC-offset case** specifically designed to expose the float-accumulator bug described above, and a near-silence case that exercises the `+1e-6` guard. A parity test on ordinary audio would have passed with the bug present.

Chapter 5 — The model: architecture, training, mining, calibration, quantisation

The constraint that designed the network

The network must run on **TensorFlow Lite Micro** on an ESP32-S3, in a fixed tensor arena, in about a tenth of a second. TFLM is not TensorFlow: it has a small, fixed set of kernels, no dynamic shapes, no control flow, and no allocator that can grow at runtime.

So the model zoo was restricted, before any training, to ops that TFLM demonstrably implements: `Conv2D`, `DepthwiseConv2D`, `BatchNormalization` (folded at conversion), `ReLU`, `MaxPool2D`, `GlobalAveragePooling2D` (which becomes `MEAN`), `Dense` (which becomes `FULLY_CONNECTED`), `Softmax`.

No attention. No RNN. No GRU. The legacy system used a 6.15 MB FP32 attention+GRU network called TinyCryNet. That is 45× larger than the deployed model and contains ops TFLM cannot execute at all. The rebuild's network is 122,770 parameters and **137,384 bytes** as INT8.

And this is not taken on trust — the export cell enumerates the converted model's operators and asserts every one has a TFLM kernel:

```
TFLM_KERNELS = {"CONV_2D", "DEPTHWISE_CONV_2D", "AVERAGE_POOL_2D", "MAX_POOL_2D",
                "MEAN", "FULLY_CONNECTED", "SOFTMAX", "LOGISTIC", "RELU", "TANH",
                "RESHAPE", "PAD", "SQUEEZE", "LEAKY_RELU"}
ops_for_tflm = [o for o in model_ops if o != "DELEGATE"]   # XNNPACK is host-only
missing = [o for o in ops_for_tflm if o not in TFLM_KERNELS]
```

The deployed model's operator set is `CONV_2D, FULLY_CONNECTED, MAX_POOL_2D, MEAN, SOFTMAX`. Five operators, all present. `DELEGATE` is filtered out because it appears only when the *host* interpreter runs XNNPACK; TFLM has no delegate.

The four candidates

Four architectures were defined, deliberately different in *shape* so that the comparison would be informative rather than a hyperparameter sweep.

`conv_cnn` — the one that won

```
inp = Input(shape=(81, 64, 1))
for f in (16, 32, 64, 64, 96):
    x = Conv2D(f, 3, padding="same", use_bias=False)(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    x = MaxPool2D((2, 2))(x)
x = GlobalAveragePooling2D()(x)
x = Dense(64, activation="relu")(x)
x = Dropout(0.3)(x)
out = Dense(2, activation="softmax", name="probs")(x)
```

122,770 parameters. A plain 2-D CNN over the log-Mel plane. Five conv blocks halve both axes each time: 81 × 64 → 40 × 32 → 20 × 16 → 10 × 8 → 5 × 4 → 2 × 2. Global average pooling then collapses the surviving 2 × 2 × 96 to a 96-vector, so the classifier head sees a *summary* of the whole two seconds rather than a flattened snapshot of one corner of it.

`use_bias=False` before every BatchNorm is not a style choice: BN's own β already provides the bias, so a preceding bias term is redundant parameters that quantisation then has to represent.

`sep_cnn` — the smallest

```
for f in (24, 48, 96, 128):
    x = DepthwiseConv2D(3, padding="same", use_bias=False)(x)
    x = BatchNormalization()(x); x = ReLU()(x)
    x = Conv2D(f, 1, use_bias=False)(x)          # pointwise
    x = BatchNormalization()(x); x = ReLU()(x)
    x = MaxPool2D((2, 2))(x)
```

29,839 parameters — 4.1× smaller. Depthwise-separable convolution factorises a 3 × 3 × C_in × C_out kernel into a per-channel 3 × 3 and a 1 × 1 mix, which is the standard MobileNet trick. It is the fastest on the ESP32 and the obvious choice if memory were the binding constraint. It was not.

`temporal_cnn` — 1-D over time

```
x = Conv2D(64, (5, 1), padding="same", use_bias=False)(x)   # time only
... MaxPool2D((2, 1))                                        # pool time, keep all 64 Mels
for f in (96, 128): Conv2D(f, (3, 1)) ... MaxPool2D((2, 1))
```

65,154 parameters. Treats Mel bins as channels and convolves only along time — the shape that wins keyword-spotting tasks. It was a serious candidate, not a straw man.

`bsc_cnn` — optional fourth

Four blocks (32, 64, 128, 128), a 96-unit head. Available via `CRYNET_KINDS` but not in the default three.

Training

Setting	Value	Note
Optimiser	Adam, lr **1e-3**	
Loss	sparse categorical crossentropy	two classes, integer labels
Batch	**128** (64 in FAST mode)	
Epochs	**35** (3 in FAST)	`CRYNET_EPOCHS` overrides
LR schedule	`ReduceLROnPlateau(monitor="val_loss", factor=0.5, patience=3, min_lr=1e-5)`	
Early stopping	`patience=5`, `restore_best_weights=True`	
Class weights	`len(y) / (2 · count_c)` for each class	balances the 13,330 / 15,148 split
Selection	**highest validation AUC**, not accuracy	AUC is threshold-free; the threshold is chosen afterwards
Seed	1337, `tf.keras.utils.set_random_seed` per variant	so the comparison is not a lottery

Selecting on **AUC** rather than accuracy is deliberate. Accuracy at threshold 0.5 conflates the model's ranking ability with an arbitrary cut point. The project needs the best *ranking*, and then picks the cut point separately against a recall target. Those are two different jobs and they are done in that order.

Hard-example mining — the second round

After the first round, the model's own mistakes are the most valuable training signal available. A cough it rates as a cry is exactly what turns a monitor into a nuisance; a cry it rates as noise is exactly what a caregiver needs to hear about.

```
p_train = score(X_train)                    # chunked, 4096 at a time
hard_neg = negatives sorted by -p_cry [:600] # the most cry-like non-cries
hard_pos = positives sorted by  p_cry [:600] # the least cry-like cries
ds_mine = sample_from_datasets([ds_train, ds_hard], weights=[1.0, 0.014])
model.compile(Adam(3e-4))                   # a third of the original rate
model.fit(ds_mine, epochs=6, EarlyStopping(patience=3, restore_best_weights=True))
```

Three details worth defending:

1. **Only training rows are mined.** The held-out folds are never scored for selection, so they stay honest. Mining on validation would be training on validation.
2. **The hard examples are mixed in by weighted sampling, not duplicated into a new matrix.** The old implementation duplicated each hard row, so 1.4 % of rows appeared twice; `weights=[1.0, 0.014]` reproduces that proportion exactly without allocating a second 900 MB copy on a machine with no page file.
3. **The learning rate drops to 3e-4.** Fine-tuning at the original 1e-3 on a set deliberately enriched in hard cases would overshoot and undo the first round.

This is the mechanism behind the README's `coughing` finding: ESC-50's coughing class produced a **57 % false-positive rate** in the first false-alarm map, and hard-example mining is the tool that class of failure is fed back into.

Threshold calibration

The threshold is not 0.5 and it is not chosen by taste. It is swept:

```
for t in np.arange(0.05, 0.96, 0.01):
    recall = (p_val[y_val == 1] >= t).mean()
    fpr    = (p_val[y_val == 0] >= t).mean()
ok = [t for t in rows if recall >= 0.95]
THRESHOLD = max(ok) if ok else 0.5
```

The highest threshold that still reaches 95 % recall on the validation fold. Highest, because every increment above the recall floor is free specificity — false alarms removed at no cost in missed cries. That single line is the whole alarm-policy philosophy of the project in miniature.

The sweep is printed every 0.05 so the recall/FPR trade-off is on the record, and the selected threshold is then used for *every* subsequent number: test fold, cross-corpus, per-corpus breakdown, per-subclass recall, and the deployed firmware.

INT8 quantisation

```
converter = TFLiteConverter.from_keras_model(BEST_MODEL)
converter.optimizations = [Optimize.DEFAULT]
converter.representative_dataset = representative_gen   # up to 512 seeded TRAIN windows
converter.target_spec.supported_ops = [OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type  = tf.int8
converter.inference_output_type = tf.int8
```

Full-integer quantisation, not the hybrid "weights INT8, activations float" default. That is required for two reasons: ESP-NN's SIMD kernels are integer kernels, and a float activation path would pull in the software floating-point reference kernels and blow the 100 ms budget.

The representative dataset is drawn from **training** windows with a fixed seed, one array per input tensor with the exact `(1, 81, 64, 1)` shape — the converter is strict about both.

A float32 TFLite model is exported alongside, as the parity reference.

The parity test, and the bug in its first version

```
n_check = min(4000, len(X_val))
_sel = np.random.default_rng(SEED + 21).permutation(len(X_val))[:n_check]
```

The comment above that line is a small masterpiece of experimental hygiene:

> `X_val` is ordered by the ICD manifest, which lists every positive clip before every negative one, so the first `n_check` windows of the validation fold would be ALL cry (the fold's 2,665 positive clips alone contribute 5,330 windows). A prefix is therefore not a sample: on the full run it produced a single-class slice, whose AUC is undefined, and the parity numbers were reported as NaN.

The first version took a prefix. It reported `NaN`. The fix is a seeded random subset. **A test that silently measures nothing is worse than no test**, and this one was caught.

Measured parity: ΔAUC −0.00137, decision agreement 99.956 % over 11,386 windows. Quantisation costs essentially nothing.

The deployed artefact

Property	Value
File	`crynet_int8.tflite`
Size	**137,384 bytes**
SHA-256	`bd592d54…dce43b9`
Parameters	122,770
Input	`int8[1, 81, 64, 1]`
Output	`int8[1, 2]` → dequantise → softmax → `[P(no_cry), P(cry)]`
Operators	`CONV_2D, FULLY_CONNECTED, MAX_POOL_2D, MEAN, SOFTMAX`
Arena	262,144 B reserved, **108,328 B used**, in PSRAM
Latency	**101.8 ms** per window on the ESP32-S3 at 240 MHz with ESP-NN

The artefact is embedded in the firmware as `crynet_model/include/model_data.h` — an 8,604-line generated C array — so there is no filesystem, no OTA partition and no model-loading failure mode. `firmware/tools/check_artifact_drift.py` verifies that `research/out/`, `deploy_config.json` and the staged firmware copies all describe the same bytes. It currently reports **PASS** — and it is the check that says which of two hashes in the project's prose is the live one.

Results

Split	AUC	What it means
ICD validation (fold 4)	**0.9994**	held-out fold of the primary corpus
ICD test (fold 5)	**0.9993**	never touched during selection
Cross-corpus	**0.9984**	unseen corpora, unseen babies: cry4 hold-out + ESC-50 + snoring + music hold-out + speech hold-out + all 40 ESC-50 `crying_baby`
INT8 vs float32, val	**0.9978 vs 0.9991**	the quantisation's own cost, measured inside the export

The cross-corpus figure is the one to quote. The other two are consistent with it, which is the useful observation: there is no large generalisation gap, so the model has learned crying rather than the corpus.

A trap worth walking into deliberately

These are the metrics of the **deployed** model — the one whose bytes are on the board. Its SHA-256 is `bd592d54…dce43b9`.

The project's own prose quotes a different hash. `DECISIONS.md` D-16 and `SYSTEM_UPDATE.md` both cite `4aef55d8…126bc11`, and D-16 quotes test AUC **0.9995** and cross-corpus AUC **0.9973**. Those are the numbers of the **previous** model — the one that was selected on 2026-10-29 and then superseded by the music retrain of 2026-11-01 (D-58). The history is recoverable from the tree:

	SHA-256	val	test	cross
Pre-retrain (2026-10-29, D-16's selection)	`4aef55d8…126bc11`	0.9994	0.9995	0.9973
Deployed (2026-11-01, D-58's retrain)	`bd592d54…dce43b9`	**0.9994**	**0.9993**	**0.9984**

The retrain *lowered* the in-domain test AUC by 0.0002 and *raised* the cross-corpus AUC by 0.0011. That is the trade it was made for: the whole point was to stop alarming on loud music (8.19 % → 0.21 %), and cross-corpus generalisation is what that buys. The pre-retrain model also recorded its INT8 parity as `NaN` — the comparison cell was broken on that run (I-1) — while the deployed model records a real `int8_val_auc 0.9978` against `float_val_auc 0.9991`.

`firmware/tools/check_artifact_drift.py` settles which artifact is which, and it is the right thing to run before quoting a hash:

```
OK  artifacts, firmware constants and deploy_config all agree
    model      : 137384 bytes  sha256 bd592d541027b9f4...
    threshold  : 0.95
    input quant: scale 0.116329655 zp -70
    staged     : 5 files identical to research/out/
```

A defence that quotes a hash must quote **that** one. The stale `4aef55d8…` in the decision log is recorded as an open documentary item in Chapter 19.

Against the deployed INT8 model, on real audio

Test	Before the music retrain	After (deployed)
ESC-50 `crying_baby` recall	36 / 40	**36 / 40** (unchanged)
ESC-50 non-cry false alarms	2.64 % (38 / 1,439)	**1.18 %**
GTZAN music false alarms	**8.19 %** (79 / 965)	**0.21 %** (2 / 965)
Quiet room, this INMP441	0 / 236 windows	**0 / 236**
Phone-played cry, per-window recall (session 1)	0.867	0.800
Phone-played cry, per-window recall (session 4)	0.724	0.592

The last two rows are the honest cost of the retrain and they are not buried. Per-window recall fell; **episode** recall did not, because a sustained cry trips many consecutive windows and the announce rule needs only one unmistakable window or two agreeing ones.

Why the legacy model was replaced

	Legacy	Rebuild
Architecture	TinyCryNet, attention + GRU	5-block plain CNN
Size	**6.15 MB, FP32**	**137,384 B, INT8**
TFLM-runnable	No (attention, GRU)	Yes, five kernels, all asserted
Mel bands	128, **HTK**	64, **slaney**
Normalisation	none	per-window CMVN
Trigger RMS	**0.18**	**0.030**, adaptive down to 0.005

Every row of that table is a defect the rebuild fixed, and the trigger row is the one that mattered most: at 0.18 a soft cry never reached any classifier at all.

Chapter 6 — The hub firmware

Layout

The hub is an ESP-IDF application built through PlatformIO, with ten first-party components. It is not an Arduino sketch: the BLE stack (NimBLE), the coexistence configuration and the PSRAM allocator all need `sdkconfig`, and Arduino's prebuilt ESP32 core does not expose it.

```
firmware/hub_infer/
  platformio.ini
  src/
    main.c            boot, probes, init order, then the main loop
    link_stage.c/.h   reassembly, verdicts, arbitration, liveness  (1,824 lines)
    console.c/.h      the serial command interface
    probes.c/.h       the four self-test windows
  components/
    crynet_dsp/       the frozen front end
    crynet_infer/     the TFLM interpreter wrapper (C++ in an extern "C" shell)
    crynet_model/     the embedded INT8 array + generated tables
    crynet_proto/     the wire format, shared with the node by construction
    node_link/        ESP-NOW receive/transmit
    ble_alert/        NimBLE GATT server
    hub_display/      ILI9341 driver + the panel UI
    crynet_log/       the cry log, on the hub's own flash   (Chapter 12)
    crynet_clock/     the wall clock, and honesty about not having it
    crynet_net/       Wi-Fi, the uploader, and the presence report
    esp-tflite-micro/ vendored third party
    esp-nn/           vendored third party, SIMD kernels
```

Vendoring TFLM and ESP-NN inside `components/` is deliberate: the PlatformIO registry versions did not match the ESP-IDF 6.1 API, and a build that fetches a moving dependency is not a build that can be reproduced in a year. `firmware/tools/fetch_deps.sh` and `patch_deps.py` record exactly what was fetched and what had to be patched.

Boot, in order

`app_main()` does eight things, and the order of six of them is load-bearing.

1. Report the platform

Chip revision and core count, PSRAM present/absent and its size, free internal heap, flash size, model size, the model's own threshold, the announce threshold, the silence gate, and the input quantisation scale and zero-point. Every one of these is a number a defence might be asked for, and the firmware prints them at every boot rather than leaving them in a document.

One ESP-IDF 6.x migration detail lives here: `esp_flash_get_size()` now takes a chip handle (`NULL` = default chip) and writes through an out-parameter, where 5.x returned it directly. And one honesty rule (D-106): the banner is the identity check, so the firmware version it prints is stamped at configure time — an earlier build inherited a stale stamp and described a new build as an old one, which is how a real stack overflow went undiagnosed for a night (see **The stacks**, below).

2. Initialise NVS

The Bluetooth stack keeps its bonding keys in NVS, so the store must exist before the host starts. The standard recovery is implemented: on `ESP_ERR_NVS_NO_FREE_PAGES` or `ESP_ERR_NVS_NEW_VERSION_FOUND`, erase and re-init.

3. Bring up the front end and the interpreter

`crynet_dsp_init()` allocates the 5.4 KB scratch in internal SRAM. `crynet_infer_create()` allocates the 262,144-byte arena in PSRAM and reports how much of it the model actually uses (108,328 B). Either failure aborts the boot with a specific error — there is no point running a monitor whose model cannot load.

4. Run the four probes — and gate everything on them

This is the design decision that most distinguishes this firmware. **The hub does not start its radio, its Bluetooth server, its display or its console unless the self-test passes.**

The probe buffers (90 KB in total: PCM, float features, INT8 features) are allocated in **PSRAM**, falling back to internal SRAM if PSRAM is absent. They are pointers, not file-scope arrays, precisely so they can be freed:

> They are only needed while the probes run, but as file-scope statics they would sit in internal RAM for the life of the program — and the Bluetooth stack needs that RAM far more than the probes do (NimBLE's store configuration was failing to allocate on a board whose radio worked perfectly).

For each probe:

1. Generate the PCM synthetically and check its **CRC-32 against the host's expected value**. If the generator itself has drifted, the test says so instead of comparing against the wrong reference.
2. Compute features; time the front end and the quantisation separately.
3. Compare the INT8 features to the host reference: **worst per-byte delta must be ≤ 1 LSB, and at most 1 % of the 5,184 features may differ at all.** Both conditions, not just the max — a single large error and a thousand small ones are different failures.
4. Run inference; time it; print `P(no_cry)` and `P(cry)`.
5. Apply **the same decision rule the live path uses** — silence gate plus `CRYNET_CLEAR_CRY`, not the model's own threshold. The comment records why: the boot banner used to say `ALERT` for the noise probe, a window the running hub would have refused, so the boot report and the system contradicted each other.

Then the buffers are freed and the free internal heap is printed again.

5. Memory, clock and log

Before any radio comes up, the hub opens its record: `crynet_clock_init()`, then `crynet_log_init()`, which mounts the six-megabyte SPIFFS partition that had been reserved from the first build and never used, and restores the clock from the last time the log saved. The hub's own start is written as a `boot` record — a gap in the

cry history means either a quiet night or a hub that was off, and only the boot records tell them apart. A flash that will not mount is a **warning, not a stop**: the monitoring is the product, the record is a benefit.

6. Radio, then Wi-Fi, then Bluetooth, then display

The order is stated in the source and it was learned the hard way:

> NimBLE's pools and the Wi-Fi driver's buffers come out of the same internal RAM, and when BLE claimed its memory first the Wi-Fi driver ran out mid-init and the ESP-NOW link never started. The radio takes what it needs while the heap is whole, and BLE lives on what is left.

Measured free internal heap through the sequence:

Point	Free internal heap
Boot	298,763 B
After the probes (buffers freed)	290,791 B
Radio up	215,155 B
Bluetooth up	**149,271 B**

(That ladder was measured before the Wi-Fi stage joined the boot; the current sequence — clock and log, radio, uploader, Bluetooth — prints its own free-heap lines on the banner at every start, and the Wi-Fi driver's buffers come out of the same internal RAM as NimBLE's pools, which is exactly why both of them initialise before Bluetooth.)

The uploader shares the radio with ESP-NOW, so `crynet_net_start()` runs here too, its interface created while the internal heap is still whole. With no network configured it starts nothing and costs nothing.

The display comes last of the subsystems because its 153,600-byte framebuffer goes to PSRAM and its driver needs only a small internal allocation.

7. Console last

`console_start()` is deliberately last, "so that every command it accepts can be carried out: the radio is up, the Bluetooth server is up, and a test alert has somewhere to go."

8. `link_stage_run()`

The main loop, which never returns.

Graceful degradation

Each of the five optional subsystems is started with a warning, not an abort:

```
if (!crynet_log_init())        ESP_LOGW(TAG, "the cry log could not be opened; "
                                        "the hub carries on without a record");
if (!link_stage_radio_start()) ESP_LOGW(TAG, "the radio could not be started; the hub carries on without
it");
if (!crynet_net_start())       ESP_LOGW(TAG, "the uploader could not be started; "
                                        "cries are still recorded in the hub's flash");
if (!ble_alert_start())        ESP_LOGW(TAG, "the Bluetooth alert server could not be started; ...");
if (!hub_display_start())      ESP_LOGW(TAG, "the display could not be started; ...");
```

The reasoning is stated in the code: *"a dead display does not stop the monitoring, it only stops the showing."* For a device whose job is to wake a caregiver, failing loudly and continuing is better than refusing to boot. The **probes** are the exception, because a hub whose model is wrong is worse than no hub.

The console

`console.c` is 444 lines: a 5,120-byte task at priority 3 reading lines from the same UART the log is written to. It grew with the record and the network, and its word list is the system's second control panel:

Command	Effect
`help`	list the commands
`status`	cots, liveness, crying mask, deaf mask, heap, counters
`alert <cot>`	inject a test alert for that cot, all the way to BLE
`clear`	clear the alert state
`history [n]`	the last n cry-log records, newest first
`log`	how much is recorded and how much is unsent
`log clear`	throw the log away (the record numbers carry on)
`net`	the network: state, batches, failures, last error
`net set <url>`	where the records are posted
`net ssid <s> <p>`	which Wi-Fi to join
`net name <n>`	this hub's name on the dashboard
`net token <t>`	the dashboard's shared secret (NVS, never compiled in)
`upload`	send the unsent records now
`logtest`	prove, on this chip, that out-of-order records are not lost

All of the configuration words write NVS, so they survive a reflash; nothing about the network is compiled into the image, and the token is not a tracked file (D-103).

`alert <cot>` is the reason the BLE path can be tested without making a noise. It exercises the real publish path — the same `ble_alert_publish()` the detection logic calls — so a successful `alert 1` proves the GATT server, the notification and the wearable's callback, independently of the model. That separation is what allowed the haptic fault to be localised to hardware (Chapter 14).

Two parsing rules were paid for in lost words: an exact match guards `log` because a prefix match would swallow `logtest` and print the log instead of running the proof (D-98); and `net` is matched with the same care, because the network's words all begin with it.

`link_stage.c` — the state machine

1,642 lines, the largest file in the project. Its constants are the system's timing behaviour:

Constant	Value	Meaning
`BURST_TIMEOUT_MS`	15000	abandon a burst that never completes
`BURST_FAIL_LIMIT`	3	consecutive failures before backing off a node
`BURST_BACKOFF_MS`	60000	how long to leave a failing node alone
`RECOOLDOWN_MS`	1500	minimum gap between verdicts on one node
`CRY_HOLD_MS`	8000	how long a confirmed cry stays on the panel
`FRAG_STALL_MS`	100	silence before asking for a resend
`FRAG_STALL_MS_MAX`	1500	adaptive upper bound on that silence
`FRAG_SETTLE_MS_MAX`	3000	how long to let a burst settle before judging it
`RESEND_MIN_FLIGHT_MS`	250	minimum gap between resend rounds
`RESEND_MAX_ROUNDS`	10	then give up on the window
`FRAG_PER_FRAGMENT_MS`	4	the pacing the hub asks the node for
`PREPARE_QUIET_MS`	3000	quiet period before a node is considered idle
`COT_SWITCH_MARGIN`	1.5	loudness ratio needed to steal the primary label
`COT_SWITCH_DWELL_MS`	1500	how long that ratio must hold
`NODE_LIVE_TIMEOUT_US`	2 s	open-radio silence before "offline"
`NODE_LIVE_MAX_US`	20 s	hard cap regardless of deafness discounts

Liveness, and why it is display-only

`node_live()` decides whether a cot's card says ONLINE or OFFLINE. It is **not** consulted anywhere in the detection or announcement path. A cot that is momentarily marked offline can still have its window classified and can still raise an alert.

The subtlety is `radio_deaf_now_us()`. While the hub is receiving a burst it is deliberately deaf to other traffic, and while it is advertising BLE it may miss frames. Counting that self-inflicted silence against a node would make every cot flicker offline whenever another cot speaks. So the liveness clock **discounts burst deafness** — `s_radio_deaf` is set only by `burst_started()`, not by BLE airtime — and is capped at 20 s so a genuinely dead node still goes offline eventually.

This is exactly the "node 1 went offline for a split second" observation from the bench: cosmetic, panel-only, and by design.

The arbitration

The hub keeps a per-cot record: last RMS, last verdict, `cry_run` (consecutive cry verdicts), the cry hold expiry, and the switch dwell timer. The primary label goes to the loudest crying cot; another cot takes it only if it is **1.5× louder for 1500 ms continuously**. Without that hysteresis the label oscillates between two cots hearing the same cry, which with two cots a metre apart is the normal case rather than an edge case.

The heartbeat, and the two RSSI readings

`link_stage_run()` sends a heartbeat once a second, on its own clock:

```
if ((probe_now - last_heartbeat_us) > 1000000) {
    last_heartbeat_us = probe_now;
    node_link_send_heartbeat(nl);
}
```

`node_link_send_heartbeat()` builds a **twelve-byte** frame of type 4 addressed to the broadcast MAC. It asks for nothing. Its entire purpose is to give each cot a frame whose RSSI it can measure, so the node's own link bar is a live reading rather than a stale one from the last pull.

The framing is the trap. `crynet_proto_build()` originally gave the twelve-byte payload-less layout only to a REQUEST; a HEARTBEAT fell through to the full fourteen-byte header and came out **sixteen bytes**. The parser on the far end treats a heartbeat like a request — any length other than `CRYNET_PROTO_REQUEST_BYTES` is a bad frame — so **every heartbeat would have been rejected as malformed**, and the node's link bar would have sat dark on a hub that was in fact answering. The fix is one condition:

```
if (type == CRYNET_PROTO_TYPE_REQUEST ||
    type == CRYNET_PROTO_TYPE_HEARTBEAT) {
```

A host test now holds both rules — `research/test_heartbeat_framing.py` compiles the real `crynet_proto.c` and checks that a heartbeat is exactly twelve bytes, that it round-trips as type 4, that the other frame types are unchanged, and that the two-cot id rule hears 1 and 2 while ignoring 0, 3 and 255. **The test was checked against the bug it was written for:** reverting the one-line fix makes it fail with `a heartbeat is 16 bytes, not 12`.

In the other direction, `slot->tel_rssi` — which the link layer had always measured and never exposed — now travels out through `node_link_poll_telemetry()` and `node_link_last_telemetry()` into `link_stage_cot_t.rssi`, and from there into the display state and the console. The console's `status` prints it beside each cot's other numbers:

```
cot 2: rms 0.0230  level 0.0090  window 0.0000 (filling)  confidence 0%  link -43 dBm
```

`link --` when there is no reading, because `0` is the sentinel for "never heard", not a dBm value.

Deafness

A node whose radio is transmitting but whose microphone produces nothing — or whose windows are all refused as clipped — is marked in `deaf_mask` and reported as `alert_type = 3`. A monitor that cannot hear is a different failure from a room that is quiet, and the system distinguishes them.

`crynet_infer.cpp` — the interpreter wrapper

247 lines of C++ behind an `extern "C"` interface, so the rest of the firmware stays C.

- The arena is allocated with `heap_caps_malloc(..., MALLOC_CAP_SPIRAM)`.
- `MicroMutableOpResolver<6>` registers **only six operators** — `CONV_2D`, `DEPTHWISE_CONV_2D`, `MAX_POOL_2D`, `MEAN`, `FULLY_CONNECTED`, `SOFTMAX` — rather than `AllOpsResolver`. Every unused kernel registered is flash the linker cannot drop. (The deployed `conv_cnn` uses five of the six; depthwise is registered so `sep_cnn` can be deployed without a firmware change.)
- ESP-NN provides the SIMD implementations of those kernels for the Xtensa LX7's vector unit; that is what turns a multi-second reference-kernel inference into **101.8 ms**.
- Output dequantisation: `(q − zero_point) × scale`, then softmax over the two logits.
- Errors are captured as a string retrievable with `crynet_infer_error()`, so a failure prints a reason rather than a status code.

Build configuration

Two files carry the configuration, and the split matters: `platformio.ini` holds the board and the build flags, `sdkconfig.defaults` holds everything ESP-IDF owns.

`platformio.ini` — `platform = espressif32 @ 8.0.0`, `board = esp32-s3-devkitc-1`, `framework = espidf`, `board_build.flash_mode = qio`, `board_upload.flash_size = 16MB`, `board_build.partitions = partitions_16mb.csv`, and the build flags `-DBOARD_HAS_PSRAM`, `-DCORE_DEBUG_LEVEL=3`, `-DCRYNET_LINK_PROMISCUOUS=1`.

Three comments in that file are hard-won knowledge and worth memorising:

- **Do not pin `framework-espidf` with a `platform_packages` URL.** PlatformIO 6.2 installs a URL-specified framework by treating the archive as an npm package: it reads the wrong `package.json` out of the tree and leaves `packages/framework-espidf` holding only a test-fixture package, so CMake then fails with *"include could not find requested file: .../tools/cmake/project.cmake"*. The framework is installed manually and pinned by `packages/framework-espidf/.piopm`.
- **`board_build.esp-idf.overwrite_managed_components = no`** — the IDF component manager downloads `espressif/esp-nn` into `managed_components`, and PlatformIO otherwise wipes it on every configure.
- **The optimisation level is set the ESP-IDF way**, in `sdkconfig.defaults` (`CONFIG_COMPILER_OPTIMIZATION_PERF`). The Arduino-style `-DCOMPILER_OPTIMIZATION_PERF` macro does nothing here.

The promiscuous flag is **off by default in the code** and currently **on in this ini**: it brings back the air counters, which count how many frames carrying the CryNet magic the radio hears against how many the ESP-NOW layer hands up. Those counters are what settle whether a fragment that never arrived was lost in the air or dropped by the hub's own receive path — the distinction that closed the airtime investigation (Chapter 9).

`sdkconfig.defaults` — the settings that make this board work:

Setting	Value	Why
`CONFIG_ESPTOOLPY_FLASHMODE`	**"dio"**	see below — the effective mode is DIO even though the QIO choice symbol is set
`CONFIG_SPIRAM_MODE_OCT` / `SPEED_80M`	y	the N16R8's **octal** PSRAM; quad mode would not see it
`CONFIG_SPIRAM_MALLOC_ALWAYSINTERNAL`	4096	allocations ≤ 4 KB stay internal automatically — the DSP scratch benefits
`CONFIG_SPIRAM_TRY_ALLOCATE_WIFI_LWIP`	y	Wi-Fi/LwIP buffers may go to PSRAM, relieving internal heap
`CONFIG_COMPILER_OPTIMIZATION_PERF`	y	**-O2**, with assertions still enabled. ESP-IDF defaults to `-Og`, and the FFT and the convnet are compute-bound
`CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ_240`	y	the platform default here is 160 MHz; the DevKitC-1's LX7 runs at 240
`CONFIG_FREERTOS_HZ`	1000	1 ms tick, so the burst timers mean what they say
`CONFIG_BT_NIMBLE_ENABLED`	y	NimBLE, **not** Bluedroid: the smaller stack, because the Wi-Fi radio and the internal heap are already busy
`CONFIG_BT_NIMBLE_ROLE_PERIPHERAL` / `BROADCASTER`	y	the hub is a server only
`CONFIG_BT_NIMBLE_ROLE_CENTRAL` / `OBSERVER`	n	compiled out — the hub never scans or connects
`CONFIG_BT_NIMBLE_MAX_CONNECTIONS`	1	one caregiver device
`CONFIG_BT_NIMBLE_ATT_PREFERRED_MTU`	247	comfortably above the 8-byte state packet
`CONFIG_ESP_COEX_SW_COEXIST_ENABLE`	**y**	software coexistence arbitrates airtime between ESP-NOW and BLE on the one radio
`CONFIG_ESP_WIFI_STATIC_RX_BUFFER_NUM` / `DYNAMIC`	32 / 64	a cot sends a 286-fragment burst as fast as the radio allows; the hub needs receive buffers that can hold one. Affordable now that the probe buffers live in PSRAM
`CONFIG_MBEDTLS_CERTIFICATE_BUNDLE` (+ `_DEFAULT_FULL`)	y	root certificates, so an `https://` dashboard can prove who it is; about 60 KB of flash, read only when a connection is actually made
`CONFIG_LWIP_SNTP_MAX_SERVERS`	3	lwIP's default room is one; two time servers are configured, so one being unreachable is not a clock lost
`CONFIG_ESP_WIFI_NVS_ENABLED`	**n**	the station keeps its config in RAM: the cot link owns the radio's channel, and nothing about the network should be written to flash on every boot

The stacks (D-107)

The last firmware reliability finding is not a setting anyone would guess at. The whole live system runs in ESP-IDF's `main` task — `app_main()` never returns — but that task's stack was left at the **3,584-byte default sized for an `app_main` that returns**, and the hub died mid-run with a stack overflow that presented as random restarts. The restarts had first been diagnosed as power-class resets (D-106) on the strength of a clean-power soak; the very next capture caught the real cause. The main task now has **16,384 bytes**, every task's stack is sized for what actually runs on it (console 5,120 · `crynet_net` 8,192 · display 4,096), and the same soak that once exonerated the power supply now runs silent: zero restarts, zero cot flap, presence posts on the exact 30-

second cadence. The lesson is recorded in the decision itself: *a soak proves the configuration that ran during the soak*.

The flash-mode subtlety (D-47)

`sdkconfig.esp32s3` carries `CONFIG_ESPTOOLPY_FLASHMODE_QIO=y` **and** `CONFIG_ESPTOOLPY_FLASHMODE="dio"`. The string is what esptool acts on, so **the images are written DIO** and both `bootloader.bin` and `firmware.bin` carry flash-mode byte `0x02`.

The reason: a combined image built with QIO makes the chip **watchdog-reset in a loop** (`rst:0x7 TG0WDT_SYS_RST` repeating, no application output). The ROM cannot read this board's flash in QIO before the bootloader has configured its own timing. **The bootloader switches the chip to QIO by itself once it is running**, so nothing is lost — DIO costs nothing at this workload.

`firmware/tools/flash_hub.py --merged` therefore passes `--flash-mode dio` explicitly when it combines the three images.

Build result: **RAM 14.3 % (46,788 B), flash 77.2 % (1,619,615 B)** of the 2 MB application slot. (The 13.0 % / 63.6 % quoted in earlier editions predates the cry log, the clock and the Wi-Fi stage; the 12.9 % / 61.9 % quoted in ones older than that predates the two-cot link fields, the RSSI plumbing and the heartbeat. Each figure was true when it was printed, which is why the banner, not this document, is the identity check.)

Chapter 7 — The cot node firmware

What the node is responsible for

`firmware/node/node_crynet/node_crynet.ino` — 1,808 lines of Arduino-framework C++ on the ESP32-C3. It has five jobs, in strict priority order:

1. **Never miss a sample.** The I²S capture task is the only thing on the node that must not be delayed.
2. **Decide when something is worth sending.** The adaptive trigger.
3. **Freeze the right two seconds.** Not the two seconds before the cry, not the two after.
4. **Deliver them intact.** 286 fragments, CRC-16 each, CRC-32 overall, resend on request.
5. **Report itself.** Telemetry at 10 Hz, a serial console, counters.

Everything else on the node is subordinate to those, and several of the hardest bugs in the project were cases where a lower-priority job interfered with a higher one.

The audio path

I²S configuration

```
g_i2s.setPins(PIN_SCK /*4*/, PIN_WS /*5*/, -1 /*no MCLK*/, PIN_DIN /*6*/);
g_i2s.setTimeout(500);
```

The INMP441 needs no master clock — it generates its own from SCK. It is configured **stereo with both slots**, because the peripheral must be given the whole 64-bit frame; the microphone drives only the slot selected by its L/R pin (tied to GND = left). The firmware then keeps the **top 16 bits** of the 32-bit slot. Narrowing it any other way would shift the audio by a byte — a mistake that produces plausible-looking noise rather than an obvious failure.

Sample rate 16 kHz, DMA-backed. `i2s_pump()` reads blocks and writes them into `g_window`, a 32,000-sample ring, advancing `g_write`.

`capture_task` is a bare loop:

```
for (;;) { i2s_pump(); }
```

with the comment *"The read itself paces this task: it returns when the samples have arrived."* No `vTaskDelay`, no yield — the blocking read is the scheduler.

Level measurement

Every 100 ms the main loop computes:

- `g_mean` — the running DC estimate;
- `g_rms` — `rms_of_last(mean, n)` over the most recent 100 ms, accumulated in **int64** so 32,000 squared 16-bit deviations cannot overflow;

- `buffer_rms()` — the same computation over a whole frozen window, used for `g_frozen_rms`.

Both remove the mean first, exactly as the hub's front end does, so the number the node reports and the number the hub computes describe the same thing.

The adaptive trigger — D-63

This is the node's most important logic and the one with the clearest empirical justification.

The problem it solved

The trigger used to be a fixed **0.030 RMS**, the same constant as the hub's. The in-domain confusion session measured what that meant: **a cry played one metre from the node has median RMS 0.0070, and not one of its 590 readings reached 0.030.** The system's reach was set by a constant, not by the classifier. The model never got the chance to be right.

The rule

```
if (g_floor < 0.0f)      g_floor = g_rms;                    // first reading
else if (g_rms < g_floor) g_floor = g_rms;                    // fall instantly
else                      g_floor += (g_rms - g_floor) * 0.002f;  // rise slowly

g_level = 3.0f * g_floor;
clamp(g_level, 0.005f, 0.030f);
```

Constant	Value	Reasoning
`TRIGGER_MULT`	**3.0**	chosen by `research/adaptive_trigger_study.py` over the recorded sessions, not by taste
`TRIGGER_MIN`	0.005	keeps the level above the microphone's own noise, where a trigger would mean nothing
`TRIGGER_MAX`	**0.030**	*exactly the fixed level this replaces* — so the rule is **never worse than before** in a loud room
`FLOOR_RISE`	0.002	0.2 % of the gap per reading; ten readings a second ⇒ **50 s time constant**

The asymmetry — instant fall, 50-second rise — is the whole design. If the floor rose as fast as it fell, a cry lasting a minute would drag the floor up to its own level and **switch the trigger off half way through the episode**. The slow rise means a long cry keeps triggering; the instant fall means the system re-sensitises the moment a room goes quiet.

The evidence for the multiple

From `adaptive_trigger_study.py` over the recorded sessions:

Condition	Fixed 0.030	3× floor
Cry at one metre	**0 trigger events/min**	**39**
Quiet room	~0	5
Loud sessions	capped at 0.030	capped at 0.030 (identical)

Measured on the boards afterwards: the level tracks the room (0.006–0.016 in a noisy room), the node's log shows it settling (`rms 0.0063 floor 0.0028 level 0.0083`), and the hub went from asking 1–2 times a minute to 5–11.

The consequence nobody wanted

A lower trigger asks the model about **near-silent windows**, and the model is not a reliable judge of those: the same quiet room produced verdicts of **0.0000 and 0.9766** for windows that differ by nothing a person could hear, and the second was a false alarm.

That is not a model bug to be fixed by retraining — it is a consequence of asking a question the model was never trained to answer. The fix belongs in the **announcement rule** (D-64, Chapter 10), not in the trigger. Recognising which layer owns a fault is the difference between a fix and a regression.

Two consecutive readings

`g_loud_streak` counts consecutive readings at or above `g_level`; a freeze needs **≥ 2**. One 100 ms transient — a door catch, a chair leg — is not a cry.

Freezing the right two seconds

A trigger sets `g_peak_pending` and records `g_peak_ms`. The freeze happens when:

```
cry_stopped = (g_loud_streak < 2);
if (age >= PEAK_DELAY_MS || (cry_stopped && age >= PEAK_STOP_DELAY_MS)) {
    if (age < PEAK_EXPIRE_MS) freeze_window();
    g_peak_pending = false;
}
```

Constant	Value	Meaning
`PEAK_DELAY_MS`	2000	the cap: two seconds *after* the peak, for a cry that goes on
`PEAK_STOP_DELAY_MS`	500	if the cry **stopped**, freeze after half a second
`PEAK_EXPIRE_MS`	4000	the peak must still be inside the ring

The source documents the design space explicitly: `PEAK_DELAY_MS = 0` gives the two seconds *before* the peak, `500` puts the peak in the middle, `2000` gives the two seconds *after*. The stop-capped rule takes the best of them: a short cry is captured whole and immediately; a long cry is captured after two seconds rather than making the system wait for it to end.

`research/simulate_freeze.py` verified that this catches as many of the corpus's cry episodes as the old peak-centred rule, and that the hub announced no room window under any of the three variants.

`freeze_window()` copies the ring oldest-first, computes `g_frozen_rms` and the **CRC-32 of the exact bytes that will be uploaded**, and increments `g_burst_seq`.

Cry mode — D-69

```
if (g_frozen_rms >= 0.05f) g_cry_mode = true;
...
if (g_cry_mode && g_rms < 0.01f && g_loud_streak == 0) g_cry_mode = false;
...
allow_freeze = !g_cry_mode || (g_frozen_rms > 0 && g_rms >= 3.0f * g_frozen_rms);
```

While in cry mode the node will not freeze another window unless a new moment is **three times louder** than the one it is holding. A sustained cry therefore produces one window, not six queued ahead of itself.

The entry threshold is absolute (0.05 RMS), not relative to the floor, and the reason is worth being able to give from memory: a quiet room has a floor near 0.001, and 5× that is 0.005 — which ordinary quiet windows exceed. A relative cry-mode threshold would put a silent room into cry mode permanently, and the node would then freeze one window and stop listening.

Protecting a window that is in flight

Three timers, and each exists because of a measured failure:

Timer	Value	The failure it fixed
`TRANSFER_HOLD_MS`	4000 ms, **refreshed on every fragment sent**	a louder moment used to be able to preempt a window mid-burst; the ring overwrote it and "*the burst died with a third of it on the floor*"
`FROZEN_HOLD_MS`	**3000 ms**	pure resend coverage after the upload. Was 8000, then 4000 (D-71), now 3000: at 8 s the three-minute session measured **6.5–7.0 s of hold-driven gap between every burst**
`FROZEN_EXPIRE_MS`	20000 ms	five seconds was not enough — the window expired while the hub was still waiting for the pieces it had asked for

The progression 8000 → 4000 → 3000 tracks the burst getting faster. That is the right way to tune a timer: measure what it is protecting, and shrink it as the thing it protects shrinks.

Identity: this node must be 1 or 2

The hub watches two cots and refuses every other id in its receive path, so a node's id is not a label — it is the difference between being monitored and being invisible.

`CRYNET_COT_COUNT = 2` in the sketch mirrors the hub's `NODE_LINK_MAX_NODES`. The id is set over the serial console (`id 1`, `id 2`) and stored in **NVS**, so it survives a reflash. A node left on a MAC-derived id outside 1..2 prints a warning at boot rather than silently failing to be heard.

That last sentence is the whole reason the warning exists. Without it the symptom is *"the hub does not see my cot"*, and the natural places to look are the radio, the channel, the antenna and the microphone — all of which are fine. On the bench both ids were checked **after** flashing, not before: COM31 came up with derived id 1 and COM37 with its saved id 2, which is the check that matters because the id lives in NVS and a flash could have cleared it.

The node's own view of the link

The node keeps `g_hub_rssi` and `g_hub_rssi_ms`: the RSSI of the last frame **the hub** sent, and when it arrived. The filter is the important part:

```
if (info != NULL && info->rx_ctrl != NULL &&
    (data[2] == LINK_TYPE_REQUEST || data[2] == LINK_TYPE_HEARTBEAT ||
     data[2] == LINK_TYPE_RESEND)) {
    g_hub_rssi = (int8_t)info->rx_ctrl->rssi;
    g_hub_rssi_ms = millis();
}
```

Everything on channel 6 is broadcast, so this node also hears the *other* cot's telemetry and its 286-fragment bursts. The first version of this code took the RSSI of any frame it heard, which would have reported the loudest transmitter in the room and labelled it "the hub" — a link bar that reads full while the hub is unplugged. Restricting it to the three hub-only frame types is what makes the reading mean what its label says.

The hub's heartbeat is what keeps the reading live between pulls: it arrives once a second whether or not any audio is wanted. `status` prints it as a bar on the same scale the panel uses:

```
link   [####] -12 dBm (hub frame 445 ms ago)
```

Bars	Threshold
4	≥ −60 dBm
3	≥ −70 dBm
2	≥ −80 dBm
1	≥ −90 dBm
0	below −90, or no reading (`0` is the sentinel for "never heard")

Both ends of the link now show a bar, and **they measure different directions**: the node's bar is the hub's transmitter as heard at the cot, the panel's bar is the cot's transmitter as heard at the hub. On the bench they read −12 dBm and −23 dBm for the same cot. That is not a contradiction; the two transmitters run at different powers (the node is deliberately at 8.5 dBm, D-65) and sit in different noise environments.

There is **no connection LED** (D-83). `SYSTEM_WIRING.md` §5.1 used to specify one on GPIO10, lit while a valid heartbeat had arrived inside a timeout. It worked and it was removed: the hub's panel already says which cots are answering, and a second, slower copy of the same answer on the cot itself was saying it in a place nobody was looking. The heartbeat and its reading stay — they are how the node knows the hub is there — and GPIO10 is free.

The I²S zero watchdog — D-79, committed, not yet flashed

A microphone that has stopped producing data does not produce an error. It produces **zeros**, and zeros look exactly like a very quiet room: RMS 0.0000, no trigger, no burst, no alert. The system fails silent, which for a monitor is the worst way to fail.

The watchdog counts *consecutive exact zeros*:

```
static const uint32_t I2S_ZERO_LIMIT = SAMPLE_RATE * 5u;   /* five seconds */
...
if (s == 0) { if (g_zero_samples < I2S_ZERO_LIMIT) g_zero_samples++; }
else if (g_zero_samples < I2S_ZERO_LIMIT) { g_zero_samples = 0; }
```

Five seconds of exact zeros is not a quiet room — a live INMP441 always has noise in its low bits. On reaching the limit the node tears down and restarts I²S from its owning task, marks the live ring damaged so no half-real window can be frozen, and increments `g_i2s_restarts`. **A frozen upload is deliberately left intact**: a window already in flight is real audio and must not be destroyed by a recovery. If the teardown or the warm-up fails, the node reboots rather than reusing handles the driver may have invalidated.

Two honest limits on this code, both recorded as O-1c:

- it detects **exact** zeros only. A microphone that fails to a stuck DC value or to low-level noise is not caught.
- the fault-injection test proved the code executes and refills a window; it did **not** prove it recovers the original physical fault, because the original fault has never been reproduced on demand.

It is committed but **not flashed** (D-79): the owner is soldering the final wiring, and a node should not be reflashed mid-solder. After it goes on, it needs a played-cry end-to-end run to confirm the node still streams and the hub still hears it.

The radio

Channel and power

`LINK_CHANNEL = 6`, set with `esp_wifi_set_channel()` **after** `esp_now_init()`. Both sides must agree; ESP-NOW does not negotiate.

TX power **8.5 dBm** (D-65), down from the 20 dBm default. The radio sits centimetres from the microphone and was putting its own noise into the audio it was about to be judged on. Five minutes of a noisy room after the change: **32 verdicts, 0 false alerts**.

`esp_wifi_set_ps(WIFI_PS_NONE)` — modem sleep disabled. A node that sleeps between beacons misses requests.

The peer rule — the bug that wedged the radio

```
static volatile bool g_hub_peer_pending = false;
```

> Set from the receive callback, acted on in the loop: the callback runs in the WiFi task, and `esp_now_add_peer()` waits for that very task, which wedged the radio — the send callback stopped firing and every `radio_send` burned its two-second cap. The hub's `node_link` carries the same rule: peers are added from a normal task.

This is a deadlock, not a race: the callback is *in* the task that `esp_now_add_peer()` blocks on. Both firmwares now defer peer registration to a normal task. It is the single most transferable lesson in the codebase for anyone writing ESP-NOW.

Broadcast and unicast

Telemetry is broadcast to `FF:FF:FF:FF:FF:FF`. The burst is **unicast to the hub**, whose MAC the node learns from the first request it receives — *"a broadcast destination is filtered more aggressively by the receiver's radio."*

The transmit queue

ESP-NOW's queue is **ten frames deep**. A burst is 286 fragments; handing them all over at once would lose every frame past the tenth, permanently.

The first implementation waited for each frame's own acknowledgement before handing over the next. That cost **~12 ms per fragment — 3.5 seconds for a burst**, which was most of the system's response time.

The current implementation keeps up to `TX_INFLIGHT_MAX = 4` frames in the radio at once:

```
g_tx_queued++;                    // handed over
on_send: g_tx_acks++;             // actually left
         if (status != OK) g_tx_failed++;
radio_wait_room(): while (inflight() >= 4) wait;
```

Four, not ten, for two reasons stated in the source: it leaves room for the telemetry and heartbeats that share the queue, and it keeps the far end's receive queue from overflowing — one fragment in three hundred was being lost on the way to the hub, and the far end's queue overflowing was the likeliest cause.

`g_tx_failed` counts frames the radio itself could not carry. **The callback's status used to be thrown away**, so this was always zero and the node reported a clean sheet even while the radio was dropping its burst. A counter that cannot fail is not a counter.

The resend dedupe — the most expensive line in the firmware

The hub sends each request three times because a broadcast is never acknowledged. The node must answer once. The dedupe window was **800 ms** — and the hub's own resend round comes about **670–770 ms** after the last ask. So the node threw away *every round after the first* as a repeat.

The consequence, quoted from the source:

> A burst that lost one fragment could never be completed. The hub asked for the missing fragment five times, the node answered none of the five requests, the burst sat at 285 of 286 until it timed out, and the whole two-second window — the one that held the cry — was thrown away. A burst that is one frame short is the *commonest* failure on a weak link, so this was the single most expensive line in the firmware.

The rule is now **two-sided**:

```
RESEND_DEDUPE_MS = 250;
repeat = (same seq) && (same bitmap) && (within 250 ms);
```

A round asking for a *different* fragment is a new round and is always answered. The bitmap comparison is what makes shortening the window safe.

Framing

Type	Value	Direction	Payload
`TELEMETRY`	1	node → hub	32 bytes: id, seq, rms, floor, level (milli-RMS), frozen rms/peak, counters
`FRAGMENT`	2	node → hub	index, total, **224 bytes**, CRC-16
`REQUEST`	3	hub → node	the sequence number wanted
`HEARTBEAT`	4	hub → node, 1 Hz	nothing — a 12-byte frame whose only job is to give the node an RSSI reading of the hub
`RESEND`	5	hub → node	seq + **36-byte bitmap** (286 bits), in a 54-byte frame

`FRAG_PAYLOAD = 224`, `FRAG_COUNT = ceil(64000 / 224) = 286`. `RESEND_BITMAP_BYTES = ceil(286 / 8) = 36`.

CRC-16/CCITT-FALSE per fragment (check vector `0x29B1` for `"123456789"`), CRC-32/IEEE 802.3 over the whole window — the same polynomial as `zlib.crc32`, which is what lets a host tool verify a capture with one line of Python.

Battery telemetry was removed with D-83: the payload went **34 → 32 bytes**.

The serial console

The node answers on the same USB serial it logs to. Commands include `status`, `stats`, `burst` (dump the frozen window in the radio's framing), `dump` (the raw window, for `firmware/tools/dump_node_window.py`), and the trigger controls.

`Serial.setTxTimeoutMs(0)` **is critical** (I-11). Without it, a `Serial.print` with no host attached blocks for the full USB CDC timeout — and on a node whose capture task must never stall, that is a dropped-audio bug that only appears when nobody is watching.

Build

Arduino CLI (bundled), ESP32 core **3.3.12**, FQBN `esp32:esp32:weact_studio_esp32c3` — the WeAct Studio ESP32-C3 board definition, which targets the **chip** rather than one module — it is the right entry for the **Pro Mini** on the cot and the SuperMini on the wrist alike. The current build compiles to **983,177 B (75 %) of flash, 166,404 B (50 %) of static RAM**, out of the 1,310,720 B app partition and 327,680 B of RAM.

> **A figure that was wrong in an earlier edition.** The stage-2b entry of 2026-10-30 records the node at **977,617 B (74 %) / 165,972 B (50 %)** — the size of the sketch on the day it was first compiled. That number was carried forward into the earlier dossier as if it were current. It is not: the sketch has been rebuilt many times since (the burst rework, the serial-set id that persists in flash, the I²S watchdog, D-83 taking the battery sense and the LEDs out). The figure above was measured on 2026-11-23 by compiling the tree as it stands, and it is unchanged by the board swap — a comment-only edit to the header compiles to the identical bytes of program storage.

`firmware/tools/flash_node.py` is the one-command flasher; `install_sketches.py` places the sketches where the Arduino CLI expects them.

Chapter 8 — The wearable firmware

The job

`firmware/wearable/wearable_crynet/wearable_crynet.ino` — 750 lines on an ESP32-C3 SuperMini. It runs no model, makes no decisions, and has exactly one purpose: **be trusted to tell a caregiver something they must not miss.** Every design choice in it follows from that.

Hardware: SSD1306 128 × 64 OLED on I²C (SDA 5, SCL 6), and a coin ERM vibration motor on GPIO7 through a BC547 low-side switch.

The rule that shapes the whole file: the radio must never block the human interface

Every Arduino BLE call blocks its caller. The two-argument `scan->start()` waits on a semaphore **forever**. `connect()` can hold for up to thirty seconds. A characteristic read waits for a peer that may never answer.

So the BLE work lives in its own FreeRTOS task, and `loop()` does only two things:

```
void loop() {
    haptic_tick();                                  // the motor, every 10 ms
    if (g_dirty || now - g_last_draw_ms >= 100) {   // the screen
        g_dirty = false; g_last_draw_ms = now; render();
    }
    delay(10);
}
```

```
xTaskCreate(ble_task, "ble", 6144, nullptr, 1, nullptr);
```

The comment states the intent exactly: *"Unplugging the hub can stall this task for a while; the screen, the buzz and the animation keep going."* A caregiver whose wristband freezes because the hub went out of range has been failed by the device, not informed by it.

The stack is 6,144 bytes — generous, because service discovery and the library's own bookkeeping run on it. Priority 1: above the idle task, below nothing the user sees.

The contract it is written against

```
HUB_NAME     = "INFANT_MONITOR_HUB";
SERVICE_UUID = "19b10000-e8f2-537e-4f6c-d104768a1214";
ALERT_UUID   = "19b10001-e8f2-537e-4f6c-d104768a1214";
STATE_UUID   = "19b10002-e8f2-537e-4f6c-d104768a1214";
```

This is the **frozen legacy contract**. The wearable existed before the hub was rebuilt, and the rebuilt hub was written to serve these exact UUIDs and packet layouts. That is why the two halves could be developed and tested independently, and it is why the model is binary — `alert_type` has one value for cry.

The alert callback

```
if (len < 6) { log("short alert"); return; }
prev_type = g_type;
g_cot      = data[0];
g_type     = data[1];
g_conf_pct = data[2] | (data[3] << 8);
g_event    = data[4] | (data[5] << 8);
g_have_alert = true; g_alert_ms = millis(); g_dirty = true;

if      (g_type == 1 && prev_type != 1) haptic_trigger();   // cry
else if (g_type == 3 && prev_type != 3) haptic_trigger();   // deaf
```

Three things are worth defending here.

Length is checked before anything is read. A malformed notification cannot read past the buffer.

One buzz per cry, not one per announcement. The hub re-issues the alert when a louder cot joins the cry — the display follows the baby from cot to cot, but the wrist buzzes once. The edge test `prev_type != 1` is what implements that. Without it, a cry heard by both cots would buzz twice and the caregiver would learn to ignore the buzz. **This was measured, not assumed:** on 2026-11-20 nine cry announcements across five episodes produced exactly five buzzes, and the four silent ones were handovers between cots.

Deafness buzzes too. `alert_type = 3` means the hub cannot get that cot's audio, so it cannot announce a cry from it. *"It is not being watched, and the caregiver has to hear that from the wrist, not only see it on the hub's panel."* A silent failure of a safety device is the worst failure it can have, and this is the line that prevents it.

The state callback

```
if (len < 8) return;
g_cots = data[0];
g_mask = data[1];        // bit 0 = cot 1
g_dirty = true;
```

The hub publishes state whenever it changes and **at least once a second** while a link is up, so the idle face's crying mask is never more than a second stale and nothing on the wearable ever blocks waiting for a read the hub may never answer. Push, not poll — the same reasoning as the BLE task.

`g_mask` drives the idle face's crying list — **and, since D-86, the alert face too.**

The alert face names every crying cot — D-86

The alert packet names **one** cot: the hub's loudest-crying-cot choice. That is deliberate and it stays, because one cry must be one buzz. But the *screen* used to draw its cot number from that single `cot_id` and then hold the

glass for `ALERT_HOLD_MS` = 60 s — so the one face a caregiver actually looks at during a cry hid the list the wrist was already receiving. Detection was never the gap: the nodes gate independently, the hub confirms each cry on its own evidence, and `crying_mask` has carried every crying cot since D-24.

The face now builds its label from **`crying_mask` unioned with the named cot**:

```
const uint8_t limit = g_cots ? g_cots : 8;
uint8_t mask = g_mask;
if (g_cot >= 1 && g_cot <= 8) mask |= (uint8_t)(1u << (g_cot - 1));
```

Why the union and not the mask alone. The alert is instant; the mask is published on change plus a 1 s heartbeat. So the mask can be a second behind the alert, or already clear while this face still holds. Without the union the screen could say `CRY` with nobody named — a safety device showing an alarm with no subject. The named cot is therefore always in the list.

Why the text shrinks. One cot renders exactly as before: `COT n` at size 3. Both crying render **`COTS 1 2`**, which is 8 characters — 144 px at size 3 on a 128 px glass, so it drops to size 2 (96 px). The loop still walks all eight mask bits and steps the size down again rather than let `snprintf` cut a cot off the end, because the state characteristic's mask byte is eight bits wide even though this hub only ever sets two. **A deaf cot keeps the mask out** — deafness is one cot by definition.

What it deliberately does not do: no second buzz when a second cot joins, no change to detection, to the serialized loudest-first pulls, to the 1.5×/1.5 s handover dwell, or to BLE/Wi-Fi coexistence. No new BLE traffic, and the hub image does not move — only the wrist is reflashed. The alternative, a seventh mask byte on the alert characteristic, was rejected: every consumer checks a *minimum* length so it would not have broken them, but it edits a frozen legacy contract and forces a hub reflash to buy about one second of freshness the wrist already has.

The confidence bar stays the loudest cry's figure — the only one the hub sends. It is not per-cot, and the code now says so.

Verified on the bench, 2026-11-20. One clip heard by both node microphones put the hub's mask at `0x03` for **fifteen consecutive seconds**, and the owner watched the glass read the two-cot label. Three separate two-cot events were captured. Over the whole session, **nine cry announcements across five episodes produced exactly five buzzes** — the four silent ones were handovers between cots, which is the one-cry-one-buzz rule working, not a miss.

Reconnection, and the bug that made a live link deaf

```
class HubClientCallbacks : public BLEClientCallbacks {
    void onDisconnect(BLEClient *) override {
        g_connected = false;
        g_alert = nullptr; g_state = nullptr;
        g_mask = 0; g_cots = 0;      // a stale mask names cots nobody is watching
        g_dirty = true;
    }
};
```

Clearing the mask on disconnect is deliberate: the alert face outlives a drop by up to a minute, and a stale mask would name cots that are no longer being watched. Cleared, the face falls back to the cot the alert itself named — which is still true.

And in `connect_to_hub()`:

> Fresh service discovery, every connection. The library keeps the previous connection's service handles across a disconnect — `m_haveServices` survives it — so without this the second connect hands back handles that point at a peer that no longer exists: **subscriptions go nowhere and the link looks alive but is deaf.**

That is a nasty class of fault: every indicator says connected, and no alert ever arrives. Forcing discovery on each connect is the cure, and the check `getServices()->empty()` fails the connection loudly rather than proceeding with nothing.

The scan runs in **two-second windows** with `setWindow(30)`, results cleared each round so the heap does not grow with every advertisement heard.

The haptic drive

```
HAPTIC_MS       = 2500;   // the whole buzz
HAPTIC_PULSE_MS = 250;    // one pulse

void haptic_trigger() {
    g_haptic_start = millis();
    g_haptic_until = g_haptic_start + HAPTIC_MS;
    Serial.printf("[haptic] alert %u — motor on GPIO7 for %u ms\n", g_event, HAPTIC_MS);
}

void haptic_tick() {
    bool on = false;
    if (millis() < g_haptic_until)
        on = (((millis() - g_haptic_start) / HAPTIC_PULSE_MS) % 2) == 0;
    digitalWrite(PIN_MOTOR, on ? HIGH : LOW);
}
```

250 ms on, 250 ms off, for 2.5 seconds — five pulses. The rhythm is the same as the alert band blinking across the top of the screen, *"so the wrist and the eye agree on what is happening."*

`haptic_tick()` is called from `loop()` every 10 ms and is non-blocking: no `delay()` inside an alert, ever. GPIO7 is driven **low before anything else happens** in `setup()`, so the motor cannot spin during boot.

How the motor was proved, and what is still unproven

The motor is a real coin ERM on a BC547 low-side switch, not the DRV2605L breakout the earliest planning documents named (D-77). Getting to a buzz took four separate proofs, and it is worth being able to tell them apart in a defence — because the fourth one is only half a proof, and saying so is the difference between an honest account and a sales pitch.

Proof 1 — the electrical path, with no hub in the room. A dedicated bench sketch, `firmware/wearable/haptic_test/haptic_test.ino`, drives GPIO7 with exactly the same pulse train the alert path uses and nothing else — no BLE, no OLED, no hub. It is flashed with one command:

```
python firmware/tools/flash_node.py --port COM30 --sketch haptic_test
```

The motor was seen and felt to toggle. That isolates the transistor, the base resistor, the flyback diode, the stall cap and the motor itself from every line of application code. The sketch stays in the repository, with the restore command in its header comment, so a fresh board can be proved without hunting for the procedure.

Proof 2 — the whole chain. On 2026-11-06 three alert cycles in a row came through complete: `ALERT cot 1 type 1 conf 100%` on the wrist, the haptic fired, and `clear` came back as `type 2`. The wearable also recovered from a hub reset mid-link — `hub lost — rescanning`, found again, resubscribed to *both* characteristics.

Proof 3 — the fitted cap. On 2026-11-11 the motor was replaced (the first one's coil was open, which showed up as the full **3.296 V** appearing across the motor with GPIO7 switching — no current means no drop across the cap) and the new motor vibrates correctly with the **22 Ω** that is actually fitted (D-85).

Proof 4 — on a cell, and this is where it gets interesting. The wearable **has run from its Li-Po pouch**: during the third round of the 2026-11-20 session it was on the pouch, not USB — which is precisely why COM30 was absent and no wrist log could be captured that round. So the unit works on its cell. **But the buzz went missing in that round**, and that is an open fault, not a closed proof.

The firmware is exonerated: D-86 touches no line matching `haptic|PIN_MOTOR|digitalWrite|prev_type`, so the drive is byte-identical to the build proven on 2026-11-11, and `[haptic] … motor on GPIO7 for 2500 ms` fired on all four alerts of the round before. **That log proves the command, not the current.** A sagging 3.3 V rail on the pouch is a live candidate — the OLED, the C3's radio peaks and the motor all share the board's one linear regulator.

The decisive test is a meter across the motor while an alert is live, and the three readings mean three different things:

Reading	Meaning
≈ **3.3 V**	no current — open coil or a dry joint
≈ **2.1 V**	current flowing; a weak motor or a sagging rail
≈ **0 V**	the BC547 is not switching

What is still unproved: the rail's behaviour during a buzz on a cell, and **runtime** — no unit's battery life has ever been measured. A coin motor's inrush is the largest transient in the system, and a pouch through a boost and a linear regulator is a stiffer test than a USB port. That is O-1d.

One trap worth knowing. `Adafruit_SSD1306` 2.5.17's `begin()` **returns** `true` **unconditionally** — the only failure path is the framebuffer allocation — and the library discards the I²C `endTransmission()` return value. So `[oled] up` does not mean a display is attached, and the `SSD1306 not found at 0x3C` line can never fire. The firmware's `g_oled_ok` is true on a board with no glass, writing a kilobyte per frame into a bus that NACKs it. Harmless, but it means **the OLED must be judged by eye, never by that log line.**

The display

The OLED is monochrome, so it cannot copy the hub panel's colour. It copies its *structure*: the waveform along the bottom edge, the wordmark, the accent band across the top of the title card, the boot progress bar. The accent is drawn **inverted**, and an alert turns the whole screen upside down for as long as the wrist is buzzing — *"the loudest thing one bit can say."*

Face	Duration	Content
Boot	`WEAR_BOOT_MS` 2400 ms	wordmark + a progress bar that fills as the device comes up
Title	`WEAR_TITLE_MS` 6400 ms	`AMAZING'S` / `CRYNET` on two lines, the owner's name and id, accent band, waveform
Dashboard	while idle	link state, cot count, crying mask, animated waveform
Alert	`ALERT_HOLD_MS` **60000 ms**	inverted screen; `COT n` at text size 3 for one cot, **`COTS 1 2`** at size 2 for both (D-86); confidence on a bar, blinking band

`centre_x()` is arithmetic rather than measurement because the classic Adafruit font is six pixels wide per cell at any text size — so centring is `(128 - 6·size·strlen(s)) / 2`, not a `getTextBounds` call. On a 1 KB-RAM display driver that matters.

The renderer redraws on a steady 100 ms beat *and* immediately whenever `g_dirty` is set, so an alert appears at once and also leaves on its own when its minute is up. No face is ever entered or left by a blocking wait.

Build

Same toolchain as the node: Arduino CLI, ESP32 core 3.3.12, FQBN `esp32:esp32:weact_studio_esp32c3`. One board type across three of the four units — both cots and the wrist — means one toolchain, one spare, one flashing procedure.

Chapter 9 — The radio link, and the airtime problem

Why ESP-NOW

The nodes must reach the hub without a Wi-Fi access point, without an IP stack, without a DHCP lease, and without a connection that can drop and need re-establishing. ESP-NOW is Espressif's connectionless 802.11 vendor-frame protocol: a MAC address and a channel, and frames go out. No association, no handshake, no reconnection state machine.

Its constraints are the design's constraints:

Constraint	Value	Consequence
Max payload per frame	250 bytes	a 64,000-byte window must be fragmented
Delivery	**unacknowledged for broadcast**	the protocol must supply its own reliability
Queue depth	10 frames	the sender must throttle itself
Channel	fixed, both ends	must be set explicitly, after `esp_now_init()`
Radio	shared with BLE on the hub	**the central problem of this chapter**

Channel **6** on both sides, set with `esp_wifi_set_channel()` after `esp_now_init()`.

The wire format

`crynet_proto.h` on the hub and the `LINK_TYPE_*` block in the node sketch were **written from the same table**. There is no code generation and no shared header between an ESP-IDF component and an Arduino sketch — so the table is duplicated deliberately and documented as such, and both sides' constants are asserted against the same numbers.

Type	Value	Direction	Payload
`TELEMETRY`	1	node → hub	32 B
`FRAGMENT`	2	node → hub	index, total, 224 B, CRC-16
`REQUEST`	3	hub → node	sequence wanted
`HEARTBEAT`	4	both	liveness
`RESEND`	5	hub → node	seq + 36-byte bitmap, 54 B total

`FRAG_PAYLOAD = 224`, `FRAG_COUNT = 286`, `RESEND_BITMAP_BYTES = 36`. `CRYNET_PROTO_MAX_PAYLOAD = MAX_FRAME - HEADER_BYTES - 2` — the 2 being the CRC-16.

Two checksums, and why two

CRC-16/CCITT-FALSE per fragment. Check vector `0x29B1` over `"123456789"`. This catches a corrupted fragment *at arrival*, so a bad fragment's bit is never set in the received bitmap and the resend logic asks for it.

Without a per-fragment check, a corrupted fragment would be marked received and the window would be silently wrong.

CRC-32/IEEE 802.3 over the whole 64,000-byte window. Computed by the node over the exact bytes it uploads, advertised in telemetry, and recomputed by the hub over what it reassembled. `MATCH` is the end-to-end proof. It is the same polynomial as `zlib.crc32`, chosen so that a host-side Python tool can verify a captured window in one line without reimplementing a CRC.

Two levels because they answer different questions: *is this piece intact?* and *is the whole thing the thing that was sent?*

The airtime problem — the hardest issue in the project

This is worth being able to tell in full, because **the reasoning was wrong twice before it was right**, and the way it was finally settled is a model of how to debug a system.

The symptom (I-24)

The burst never completed. **267–274 of 286 fragments arrived**, then the burst timed out. Every time, roughly the same fraction missing.

Hypothesis (a): the Wi-Fi channel — wrong

Both ends were moved to channel 6 and the node was heard. But the loss fraction did not change. If interference from other networks on the channel were the cause, changing channel should have changed the loss rate. It did not. **The loss was independent of the channel** — a fact that would later be the decisive clue.

Hypothesis (b): the node is dropping frames — wrong

The node's own counters: **7,667 frames sent, 0 lost, 2.3 ms average send wait.** The node was not dropping anything. (Getting to trust that number required first fixing `g_tx_failed`, which had been hard-wired to zero because the send callback's status was thrown away — a counter that cannot fail is not evidence.)

Hypothesis (c): the hub's own Bluetooth — right

The decisive experiment was **subtractive**: build the hub **without BLE at all** and run the same burst.

> **286/286 fragments in 3.75 s, four complete bursts in a row.** With BLE on, the same build lost ~6 % of every burst.

One variable removed, symptom gone. That is a controlled experiment, not a guess.

The mechanism

BLE advertises on channels **37, 38 and 39**. The hub has **one radio**. To send an advertisement it must *retune* to each of those channels in turn — and while it is retuned, **the Wi-Fi receiver is deaf**.

NimBLE's default advertising interval is 30–60 ms, and each advertisement is three events (one per channel). That is roughly **100 transmissions a second** and about **9 % deafness**, independent of the Wi-Fi channel and independent of the frame rate.

That single mechanism explains every observation, including the one that had ruled out the first two hypotheses: **why the loss was independent of the channel.**

> The root cause was confirmed. It is not a bug in either end's code; it is the physical reality of one radio carrying two protocols.

This is the sentence to be able to say in a defence. Not everything that breaks is a bug.

The four fixes

Issue	Fix	Result
I-25 — no way to recover lost frames	Implement `RESEND` (D-26): a 36-byte bitmap of what is missing in a 54-byte request; the node replays exactly those from the same frozen window with the same sequence number	**286/286 with BLE on**, CRC-32 MATCH, and the first end-to-end cry alert
I-26 — a resend request can itself be lost	Send the request **three times** (D-28). Broadcast is never acknowledged, and the request goes out at the moment the radio has been busiest	A lost request no longer costs a round
I-27 — the node served the same request 2–3 times	Dedupe repeats within 800 ms (D-28). With 128 fragments missing, three servings meant 384 fragments of air instead of 128, and a 9.6 s burst instead of 5	One serving per round; the burst settles in ~5–7 s
I-28 — "*no frozen window for it*"	The node **holds** an uploaded window (D-29), and the hub **abandons** a burst the node can no longer complete, asking for the newer window at once (D-30)	The resend is served; a hopeless burst recovers in ~3 s instead of 15
The heartbeat came out 16 bytes, not 12	`crynet_proto_build()` gave the payload-less layout only to a REQUEST; a HEARTBEAT fell through to the full 14-byte header. The far end parses a heartbeat like a request, so **every heartbeat would have been rejected as malformed** and the node's link bar would have sat dark on a hub that was answering. One condition fixes it; `research/test_heartbeat_framing.py` now holds it, and was checked by reverting the fix	The node's link bar is live: `link [####] -12 dBm (hub frame 445 ms ago)`

The fix that made I-27's cure into a new bug

The 800 ms dedupe window was **longer than the hub's own resend round**, which comes about **670–770 ms** after the last ask. So the node classified *every round after the first* as a repeat and answered none of them.

> A burst that lost one fragment could never be completed. The hub asked for the missing fragment five times, the node answered none of the five requests, the burst sat at 285 of 286 until it timed out, and the whole two-second window — the one that held the cry — was thrown away.

The rule became **two-sided**: a repeat must name the same window **and the same fragments** and arrive within **250 ms**. A round asking for a different fragment is always answered. Comparing the bitmap is what makes the shorter window safe.

The lesson generalises: *a dedupe window must be shorter than the interval it is deduplicating against, and identity must be content, not just timing*.

The live test that closed it

Hub + node + a real BLE client connected, cry recording played through speakers:

```
W (37022) link: node 1: the burst stalled with 268 of 286 fragments; asking for the 18 that are missing
W (38562) link: node 1: the burst stalled with 285 of 286 fragments; asking for the 1 that are missing
I (39830) link: node 1: 286/286 fragments in 6.58 s
I (39830) link: node 1: window crc32 0x336F318E, the node advertised 0x336F318E -> MATCH
...
I (60681) link: node 1: window crc32 0x6230F1C4, the node advertised 0x6230F1C4 -> MATCH
I (60842) link: node 1: ALERT — cry
```

and on the phone:

```
ALERT {'cot': 1, 'alert': 'cry', 'confidence_pct': 100, 'event_id': 1}
STATE {'cots': 1, 'crying_mask': 1, 'confidence_pct': 100, 'started_at_s': 0}
ALERT {'cot': 1, 'alert': 'cleared', 'confidence_pct': 100, 'event_id': 2}
```

Note what that log shows: the mechanism **working as designed**, not a link with no losses. Eighteen fragments lost, asked for, delivered; then one lost, asked for, delivered; then MATCH. The reliability is in the repair, not in the absence of loss.

Reducing the deafness itself (D-61)

Repair made the link correct; it did not make it fast. Three changes made it fast:

1. **The hub advertises only when it must** — not at all while a caregiver's device is connected, because nobody needs to *find* a hub that is already connected and the notifications travel over the connection.
2. **When it must, at 100 ms** (`ADV_INTERVAL 0x00A0`) instead of NimBLE's fastest default (`0x0020` = 20 ms, about 150 radio transmissions a second across the three BLE channels); and at **320 ms** (`ADV_INTERVAL_SLOW 0x0200`) while a burst is in the air.
3. **The node paces fragments at ~4 ms** instead of ~10 ms.

Measured: 5.54–6.93 s before → 2.06–2.11 s after, 286/286 fragments every time, with no resend rounds. Request-to-verdict latency 2.23–3.57 s.

The burst-time policy is *slow*, not *silent*, and the reason is a measurement from the bench: in the busiest session recorded the radio was busy **89 % of the time** — 230 s of burst airtime in a 260 s session — so a wearable waiting for a gap could take **a minute to attach**. (That session is the upper bound on airtime, not the two-cot steady state; it has not been re-measured with exactly two cots transmitting, and Chapter 19 says so.) An advertisement every 320 ms costs the burst a frame or two, and the resend path recovers them; silence cost the caregiver the device altogether. That is the trade stated plainly, and it is the right way round — a slightly slower burst is a nuisance, an unattachable wearable is a failure.

Later sessions gave a wider range — **2.06–5.19 s, median 3.55 s** — because the per-fragment cost is the *channel's* contention on shared channel 6, not the pacing. The honest figure is a range, and the reason the range

exists is understood.

The air counters

`-DCRYNET_LINK_PROMISCUOUS=1` enables promiscuous-mode counting: how many frames carrying the CryNet magic the radio *hears*, against how many the ESP-NOW layer *hands up*. That distinction is what separates "lost in the air" from "dropped by the hub's own receive path" — the two possibilities that hypothesis (b) could not distinguish from the node's side alone.

It is off by default in the code and currently enabled in `platformio.ini`, because it is cheap and the question it answers is the one that keeps recurring.

Two nodes on one channel

There is no collision-avoidance protocol, no token, no time slots. Two nodes share channel 6 — with every Wi-Fi network in the building — by three mechanisms:

1. **Telemetry is tiny and rare in bytes** — 32 bytes at 10 Hz ≈ 3.2 KB/s per node.
2. **Bursts are pulled, not pushed.** The hub requests one window at a time, so at most one node is ever transferring 64 KB. That is the whole arbitration scheme, and it needs no coordination between nodes at all.
3. **Cry mode** stops one node from queuing six windows ahead of itself.

The hub is the scheduler, and it schedules by choosing whom to ask. This is why the pull model was chosen over a push model, and it is a genuinely elegant answer to a problem that looks like it needs a MAC protocol.

What was measured, finally

Claim	Number
Bad frames / bad payloads	**0 / 0** over thousands of frames (hub's own counters)
Burst completion with a BLE client connected	**286/286, CRC-32 MATCH**
Burst duration	2.06–5.19 s, median 3.55 s
Request → verdict	2.23–5.35 s, mean 4.03 s
Live demo session	45 bursts, **45/45 complete**, no stall, timeout, CRC error or resend round

Chapter 10 — The announcement policy: turning a probability into an alarm

Why this layer exists at all

A classifier returns a number. An alarm wakes a person. Those are not the same thing, and the gap between them is where most of this project's engineering judgement lives.

Three measured facts forced the policy into existence:

1. **The model's output saturates at P(cry) = 0.9961.** A clear cry, a saxophone solo and a loud conversation can all land on the same value. No threshold separates them.
2. **The model is not a reliable judge of near-silence.** The same quiet room produced verdicts of **0.0000 and 0.9766** for windows that differ by nothing a person could hear. The adaptive trigger (D-63) deliberately asks about such windows, so this is a consequence of a good decision, not a defect.
3. **A conversation in the next room reaches 0.9727.** One step louder and an earlier, more permissive rule would have announced it as a cry at three in the morning.

So the hub does not announce on the model's calibrated threshold (0.95). It announces on **saturation plus physical plausibility**, and it keeps the weaker evidence as a counter rather than as a licence.

The constants, with their measured justification

All in `firmware/hub_infer/src/link_stage.h`.

Constant	Value	What it means
`CRYNET_TRIGGER_RMS`	0.030	fallback pull level for a node that does not report its own
`CRYNET_CLEAR_CRY`	**0.9900**	the saturation bar a quiet window must reach
`CRYNET_ANNOUNCE_MIN_RMS`	0.006	below this a window is not audible at all
`CRYNET_CRY_RUN`	2	consecutive cry verdicts before a run may confirm
`CRYNET_MAYBE_MIN_RMS`	**0.020**	the level a window needs to be counted or confirmed
`CRYNET_PULL_CRY_RMS`	0.050	a frozen window this loud is pulled immediately
`CRYNET_LOUD_ANNOUNCE_RMS`	0.080	above this the window is "loud"
`CRYNET_LOUD_ANNOUNCE_P`	**0.9900**	the confidence a loud window needs
`CRYNET_CONFIRM_P`	**0.9900**	the confidence the second window of a run needs
`CRYNET_CLIP_PEAK`	**31000**	of 32767 — 0.946 of full scale; refuse outright
`CRYNET_QUIET_CREST`	**3.5**	peak/RMS a quiet window needs to look like a cry

Where 0.050 came from

Measured on the bench, 2026-11-04: the loudest window the nodes froze out of **ordinary room sound** was **0.036 RMS** (node 2); every window a **phone cry at 15 cm** froze was **0.177 and 0.184**. 0.050 sits cleanly between them. Quieter frozen windows are still pulled the moment the *live* reading is loud, so a cry at one metre is not lost by this — it is pulled while it is happening.

Where 3.5 came from

The crest factor (peak ÷ RMS) separates an impulsive sound from a steady one:

Sound	Crest
Room hiss that scored P(cry) = 0.9961 but was not a cry	**2.2 – 2.9**
A real quiet cry, P(cry) = 0.9961 at 0.0103 RMS	**5.6**
Phone cries at 15 cm	**6.6 – 6.9**

Room hiss is Gaussian: no impulses, so peak and RMS stay close. A cry is a sequence of voiced bursts, so its peak runs well above its RMS. 3.5 sits in the empty gap between 2.9 and 5.6.

A loud bump has a high crest too — but the model rejects it, which is the model's job. The crest test is only applied to *quiet* windows, where the model's word alone is not trustworthy.

The rule as implemented

```
const bool saturated = (p_cry >= CRYNET_CLEAR_CRY);

const bool unmistakable = audible && !clipped &&
    ((rms >= CRYNET_LOUD_ANNOUNCE_RMS)
         ? (p_cry >= CRYNET_LOUD_ANNOUNCE_P)   /* loud: the model's word */
         : (p_cry >= CRYNET_CLEAR_CRY)) &&     /* quiet: it must saturate */
    ((rms >= CRYNET_MAYBE_MIN_RMS) || (crest >= CRYNET_QUIET_CREST));

if (says_cry) {
    if (rms >= CRYNET_MAYBE_MIN_RMS && st->cry_seq != st->request_seq) {
        st->cry_seq = st->request_seq;
        st->cry_run++;
    }
    confirmed = unmistakable ||
        (st->cry_run >= CRYNET_CRY_RUN && rms >= CRYNET_MAYBE_MIN_RMS &&
         p_cry >= CRYNET_CONFIRM_P);
} else {
    st->cry_run = 0;
}
```

Read it in words:

- **Audible** — RMS ≥ 0.006. A window with no sound in it cannot be a cry.

- **Not clipped** — peak < 31000. Corrupted evidence is refused, not interpreted.
- **Saturated** — P(cry) ≥ 0.99, on both the loud and the quiet path today.
- **Plausible** — either the window carries real energy (RMS ≥ 0.020) or, if quiet, it has a cry's impulsive shape (crest ≥ 3.5).

`cry_seq != request_seq` ensures one window is counted **once**, even if it is scored twice.

Why the run no longer announces anything

Both `CRYNET_LOUD_ANNOUNCE_P` and `CRYNET_CONFIRM_P` are **0.9900** — the same saturation the unmistakable path already requires. So a window that satisfies the run clause already satisfies the unmistakable clause. **In practice the run announces nothing.**

That is deliberate, and the header says so:

> The run is kept because it says how close the model came, and because setting this to 0.9500 together with `CRYNET_LOUD_ANNOUNCE_P` brings back the whole old rule — the loud window's own threshold and the two-window confirmation.

The earlier rule let a *loud* window announce at 0.95 without waiting for saturation. That was a band-aid for a clipped recording, and it opened the way for a conversation to be announced as a cry. It was removed, and the two constants were left in place at the saturation value so the whole old behaviour is **one edit away** if a future model earns it.

This is good engineering practice worth defending: *keep the knob, set it to the safe value, and write down exactly what turning it costs.*

The trade, measured in both directions

The header records both sides, which is what makes the choice defensible rather than arbitrary.

The cost of requiring saturation: on the corpus, **7 of the 53 cry episodes** the old rule caught peaked between **0.9688 and 0.9883** and are now missed. All seven are *strong* cries — RMS 0.09–0.22, crest 4.3–7.9. (An earlier count in `link_stage.c` records the same trade as 8 of 57 over a slightly different episode set.) On the bench, **three genuine cries out of three scored 0.9961 and all three still announce.**

The cost of the old rule: in twenty minutes of a room with people talking, the model scored **four windows between 0.9688 and 0.9805 that were not cries**, and the hub's own boot probe scores **0.9805 on synthetic noise**. None of the four was loud enough (0.080 RMS) for the old rule to let it through — *but that is the one thing about them that was not measured*: **how the model scores a loud conversation is unknown**, and the old rule would have announced it.

So the decision was made under a known unknown, and the unknown is written down:

> This number is the saturation, and it is the only thing between a loud conversation and a false alarm at three in the morning. Set it to 0.9500 to get the old rule back — 7 more episodes on the corpus, and the way open again.

That is the honest form of a safety trade-off: state what you gain, state what you lose, and state what you have not measured.

Clipping refusal

`CRYNET_CLIP_PEAK = 31000` of 32767 — **0.946 of full scale**. A window that reaches it is refused before any DSP runs.

The physics: the INMP441 is a **fixed-gain** digital microphone, −26 dBFS ±3 dB. Its L/R pin selects the channel, not the gain. When the sound pressure exceeds what it can represent, it flat-tops *inside the microphone*, before the 16-bit window ever sees it. The resulting harmonics are an artefact of the clipping, not of the sound, and the model has never been trained on that artefact.

There is no software cure. Turning down a digital gain after the fact does not restore information that was destroyed in the transducer. The correct behaviour is to refuse the corrupted evidence — and the system's redundancy handles the rest: the **bystander cot**, further from the baby and therefore unclipped, announces the cry instead. That is exactly what was observed on the bench.

Hold, arbitration and hysteresis

Mechanism	Constant	Behaviour
Cry hold	`CRY_HOLD_MS` 8000	the announcement outlives the window that confirmed it, so the display does not flicker while the cry goes on
Re-cooldown	`RECOOLDOWN_MS` 1500	minimum gap between verdicts on one node
Primary label	loudest crying cot	the panel names one cot as primary
Switch margin	`COT_SWITCH_MARGIN` **1.5×**	another cot must be half again as loud
Switch dwell	`COT_SWITCH_DWELL_MS` **1500 ms**	…and stay that loud for a second and a half

The hysteresis is not decoration, and with **two** cots it is not an edge case either. One baby, two microphones a metre apart, one cry: the two RMS readings land within a few dB of each other, so without the margin and the dwell the primary label oscillates — "first cot 1, then cot 1 and 2, then cot 2" — while nothing about the room has changed.

The alert names **one** cot for the same reason it holds for eight seconds: one cry must be one buzz. The `crying_mask` still carries every crying cot, so nothing is thrown away — and since **D-86** the wrist's alert face reads that mask and names **every** crying cot, while the packet and the buzz stay one-cot. The screen and the buzz are deliberately different: the caregiver needs the whole list to look at, and one pulse to act on. Two cots crying at once was staged on 2026-11-20 and the mask sat at `0x03` for fifteen seconds; Chapter 8 has the capture.

Deafness is an alert, not a silence

`alert_type = 3` is raised when a cot's audio cannot be obtained or every window it sends is refused. The wearable **buzzes** for it, with the same edge-test as a cry so it buzzes once.

The reasoning, from the wearable's own source: *"It is not being watched, and the caregiver has to hear that from the wrist, not only see it on the hub's panel."* A monitor that fails silently is worse than no monitor, because it produces false confidence.

The policy in one sentence

Announce only when the model is saturated and the window is physically plausible; refuse corrupted evidence outright; hold the announcement so it does not flicker; require sustained dominance before naming a different cot; and buzz the caregiver when the system cannot hear.

Chapter 11 — The Bluetooth contract and the caregiver's device

The frozen legacy contract

The wearable existed before the hub was rebuilt. Rather than change both ends at once, the project **froze the BLE contract** and wrote the new hub to serve it byte-for-byte. That is why the two halves could be developed, flashed and tested independently — and it is the reason for several constraints that would otherwise look arbitrary.

Item	Value
Device name	`INFANT_MONITOR_HUB`
Service UUID	`19b10000-e8f2-537e-4f6c-d104768a1214`
Alert characteristic	`19b10001-e8f2-537e-4f6c-d104768a1214`
State characteristic	`19b10002-e8f2-537e-4f6c-d104768a1214`
Stack	**NimBLE**, peripheral + broadcaster only
Max connections	**1**
Preferred MTU	247

The alert packet — 6 bytes

Offset	Field	Type	Meaning
0	`cot_id`	u8	which cot
1	`alert_type`	u8	**1 = cry, 2 = stopped/cleared, 3 = deaf**
2–3	`confidence_pct`	u16 LE	0–100
4–5	`event_id`	u16 LE	the cry log's record number for this event, truncated to 16 bits — the same number the dashboard stores; 0 means the log was unavailable and the hub counted the event itself

The state packet — 8 bytes

Offset	Field	Type	Meaning
0	`cots`	u8	how many cots the hub knows
1	`crying_mask`	u8	**bit 0 = cot 1**, bit 1 = cot 2; bits 2–7 always 0 on this hub
2	`deaf_mask`	u8	same bit order
3	`confidence_pct`	u8	the loudest cry's confidence
4–7	`started_at_s`	u32 LE	when the current cry episode began

`alert_type` **has one value for "cry".** That single byte is why the model is binary (D-1). A four-class cause classifier would have needed a new packet layout, a new wearable, and a new validation of the whole path — for a distinction the project explicitly decided not to make.

`crying_mask` is a bitmask rather than a single id so that the wrist can know about *both* cots at once. Only two bits can ever be set here — the hub refuses ids outside 1..2 in its link layer — but the byte is eight bits wide because the contract is frozen legacy and was not narrowed when the cot count was fixed. The deployed wearable uses the mask for its idle face **and, since D-86, for its alert face too** — unioned with the cot the alert named, so the screen can never say `CRY` with nobody named. The six-byte alert still carries a single `cot_id`, and that is what gates the buzz.

Why NimBLE and not Bluedroid

From `sdkconfig.defaults`: *"the hub is already using the Wi-Fi radio for ESP-NOW and the internal RAM for the front end's working buffers, and NimBLE is the smaller stack."*

The roles are trimmed to exactly what a server needs:

```
CONFIG_BT_NIMBLE_ROLE_PERIPHERAL=y
CONFIG_BT_NIMBLE_ROLE_BROADCASTER=y
CONFIG_BT_NIMBLE_ROLE_CENTRAL=n
CONFIG_BT_NIMBLE_ROLE_OBSERVER=n
CONFIG_BT_NIMBLE_MAX_CONNECTIONS=1
```

Central and observer are compiled out — the hub never scans and never connects. Every role removed is RAM and flash the coexistence budget gets back.

The advertising policy — the sharpest trade-off in the firmware

One radio carries both protocols. While the BLE stack transmits an advertisement the Wi-Fi receiver is deaf, and the hub's window arrives as 286 ESP-NOW fragments it must not miss. NimBLE's fastest default interval is `0x0020` = **20 ms**, and each advertisement goes out on all three BLE channels — about **150 radio transmissions a second**, every one of them a moment when the burst can lose a frame.

The policy, quoted from the source:

Situation	Behaviour	Reason
A caregiver's device is **connected**	**not at all**	nobody needs to *find* a hub that is already connected; notifications travel over the connection, not over advertising
A burst is **in the air**	**slowly, 320 ms** (`ADV_INTERVAL_SLOW 0x0200`)	the wearable is the thing that has to be found
Otherwise	**100 ms** (`ADV_INTERVAL 0x00A0`)	the phone still finds it in well under a second

The middle row is the interesting one, and it reversed an earlier decision. Advertising used to be **stopped** during a burst. Then the bench measured a session in which the radio was busy **89 % of the time** (230 s of burst airtime in 260 s), so the wearable was left waiting for a gap and **could take a minute to attach**.

> An advertisement every 320 ms costs the burst a frame or two, and the resend path recovers them; silence cost the caregiver the device altogether.

That is the correct ordering of costs: a slightly slower burst is a nuisance the protocol already repairs; an unattachable wearable is a failure of the device's whole purpose.

Implementation detail

NimBLE takes the advertising interval **when advertising starts**, so changing the interval means stopping and restarting:

```
static void update_advertising(void) {
    const bool wanted = !s_connected;
    const uint16_t interval = (s_radio_busy || s_preparing) ? ADV_INTERVAL_SLOW : ADV_INTERVAL;
    if (wanted == s_advertising && interval == s_adv_interval) return;   // nothing to do
    ...
}
```

Two flags drive it: `s_radio_busy` (the radio needs the air now) and `s_preparing` (it will need it shortly). The early return matters — the function is called on every burst start and stop, and an unconditional log line printed *"advertising is off (a phone is connected)"* seven times in a row before the guard was added.

Publishing

```
void ble_alert_publish(uint8_t cot_id, bool crying, uint16_t confidence_pct,
                       uint32_t record);
void ble_alert_publish_deaf(uint8_t cot_id, uint32_t record);
void ble_alert_publish_state(const ble_state_packet_t *state);
```

`record` is the cry log's number for the event, handed out when the episode *starts* so the wrist can show it while the cry is still running. It is what distinguishes a new event from a repeated notification of the same one, and it is the join between the three surfaces: the buzz, the panel and the dashboard's record all name the same number (Chapter 12). The packet field is sixteen bits, so the number wraps at 65,536 — about three years of nights — while the dashboard keeps the whole thirty-two. When the log could not be opened, the hub passes 0 and counts its own events, so the wrist still shows a number that rises. Notifications are sent only when a client is connected and subscribed; the value is also readable, so a client that attaches mid-cry gets the current state rather than waiting for the next edge.

State is published **whenever it changes and at least once a second** while a link is up. That guarantee is what lets the wearable be entirely push-driven — it never issues a blocking read.

The console shortcut that made this testable

`alert <cot>` on the hub's serial console calls the same `ble_alert_publish()` the detection logic uses. A successful `alert 1` therefore proves the GATT server, the notification, the wearable's callback and the display — **independently of the model and the radio**.

That separation is what allowed the haptic fault to be localised: the alert arrived, the callback fired, the log printed, and the motor did not move. Without a way to inject an alert, that diagnosis would have required making a noise and hoping.

The host-side test clients

Tool	What it does
`firmware/tools/ble_alert_test.py`	connects as a BLE client, subscribes to both characteristics, prints every packet — this is what produced the `ALERT {'cot': 1, ...}` lines quoted throughout
`firmware/tools/serve_alert_page.py`	a local web page showing the live alert state, for a demonstration without the wearable

Coexistence

`CONFIG_ESP_COEX_SW_COEXIST_ENABLE=y` — software coexistence arbitrates airtime between ESP-NOW and BLE on the one radio. Without it the two stacks fight and the burst loses. With it, and with the advertising policy above, the measured result is **286/286 fragments with CRC-32 MATCH while a BLE client is connected** — which is the exact configuration that failed before this work.

Chapter 12 — The cry log, the network, and the dashboard

Why the hub needed a record at all

Until this chapter's subsystems existed, everything the system knew about a cry lived in three places: the hub's panel (until the power cut), the wrist's buzz (until it faded), and the hub's serial console (until someone scrolled past it). A caregiver who slept through a buzz had nothing to look at in the morning. The dashboard is the answer to that, and it is four pieces that were built in this order:

Piece	Where	What it is
The cry log	`crynet_log`, on the hub	every event written to the hub's own flash, network or no network
The clock	`crynet_clock`, on the hub	real timestamps for those records, and honesty about when it has none
The network stage	`crynet_net`, on the hub	Wi-Fi, SNTP, the uploader, and the presence report
The dashboard	`dashboard/`, off the hub	a service and a page, in two copies: one offline, one hosted

One rule governs all four, and it is written at the top of `crynet_net.h`: **the hub must work exactly as it does now with no internet at all.** Nothing in this chapter is on the path between a cot's microphone and the caregiver's wrist. The log is written by the audio loop whether or not the network stage ever runs; the network stage only reads the log afterwards and posts it somewhere. If the network is absent, misconfigured, or broken, the only thing that changes is that the records stay on the hub — six megabytes of them — and go up when it returns.

The cry log — `crynet_log`

Where it lives

The partition table has always reserved six megabytes for a `storage` partition:

```
storage,  data, spiffs, 0x210000, 0x600000,
```

The space was there from the first build and nothing used it — the component that mounts it did not exist (D-96). `crynet_log` mounts it and appends to one file. Six megabytes at thirty-two bytes a record is 196,608 records; at fifty cries a night that is a decade of nights, and the log keeps a rolling window of *sent* records long before then.

The record

One fixed-size record per event, thirty-two bytes, so that 128 of them fill a flash page exactly:

Field	Type	Meaning
`seq`	u32	the hub's own number for this record — the number the wrist shows
`boot_id`	u32	which run of the hub it belongs to
`uptime_ms`	u32	when it happened, from the hub's start
`epoch_s`	u32	when it happened in the world, **if the clock knew**
`duration_ms`	u32	how long the episode lasted
`peak_rms_milli`	u16	the loudest moment, in thousandths
`mean_rms_milli`	u16	the average over the episode
`kind`	u8	1 cry · 2 deaf · 3 heard · 4 boot · 5 console test
`cot`	u8	which cot, 1-based
`peak_conf`	u8	the model's highest confidence during the episode, percent
`windows`	u8	how many windows the model judged
`reserved`	u32	room for the next field

A cry record is written **once, when the episode ends**, and never changed afterwards. But its *number* is spent when the episode **starts** — `crynet_log_reserve()` hands it out so the wrist can show "event #1042" while the episode is still running — and the record is written when it ends. That one asymmetry broke the first uploader, as the next section describes. The other kinds (a cot going deaf, a deaf cot coming back, the hub starting) are written immediately; they are the system's own health, and they belong beside the cry data.

The number is the join between the three surfaces: the wrist shows it in the alert, the panel's footer counts it, and the dashboard stores it as the record's key. A caregiver who saw an alert at three in the morning can find that exact cry afterwards. The BLE packet's field is sixteen bits, so on the wrist the number wraps at 65,536 (about three years of nights); the dashboard keeps the whole thirty-two. If the log could not be opened, the hub passes 0 and counts its own events as it always did — the wrist still shows a number that rises.

The ordering rule (D-94)

The uploader sends the records that have not been sent yet, oldest first, in batches, and after each batch it may claim *a number* as sent: everything up to it has gone, nothing above it has. Getting that number wrong loses cries in exactly the way nobody notices — the hub says everything has gone, and the dashboard simply never shows one.

Two facts make it subtle. First, **the file is not sorted by number**: a record's number is spent when the episode starts and written when it ends, so two cots crying at once can end in the opposite order they started, and the file can hold `…, 47, 49, 48, …`. The original binary search over the file was invalid in exactly that case — asked for the first record above a number it could land *past* a record with a lower number, and that record was then never sent. Second, the watermark cannot simply be the highest number in the batch: an episode that has started but not yet ended holds a spent number that has no record yet, and claiming it would say "sent up to #50" while #48 is still being measured.

The rule that survived: the batch is the **lowest unsent numbers** (a linear scan, not file order — taking file order could re-send the same high-numbered record forever while a lower one sits behind it), and the

watermark is **clamped below the lowest number that did not fit in the batch and the lowest number an open episode still holds**.

This property is proved twice, on purpose:

- `firmware/tools/test_log_order.py` models the algorithm on the host — six tests including a ten-thousand-case soak, plus a companion test that runs the *old* algorithm and shows it losing records, so the guard cannot silently stop guarding;
- the hub's `logtest` console command runs the same scenario against the real component on the real chip, on its own scratch file, touching nothing real: two cots crying at once, the first cry to start ending last, the uploader running in between — every record must still be sent.

What happens when it fills

`crynet_log_trim()` keeps the log to a rolling window, but only over records that have been sent. Unsent records are never dropped unless the log has genuinely run out of room, and when that happens the count is kept in `crynet_log_dropped()` and said out loud — a full log is not a silent loss, it is an alarm in the console. Clearing the log does not reset the numbering, so the dashboard never confuses new records with old ones.

The clock — `crynet_clock`

A record without a time is not much of a record — the point of the dashboard is seeing that the baby was up at three in the morning, in that cot, most nights this week. But the hub has no real-time clock and no battery-backed one. It gets the time from an NTP server when it is on the network, and two rules make that trustworthy anyway:

- **The hub's monotonic uptime is the authority for *how long*.** The wall clock is stored as a pair — *at this uptime, the time was that* — so a correction from the network moves the wall clock without moving the uptime, and no record is ever stamped twice.
- **The last known time survives a restart.** It is written to NVS as the hub runs and read back at boot, so the clock starts from where it was and runs on the uptime again. It is wrong by however long the hub was switched off — the price of not having a battery — and the network corrects it within a minute or two of connecting. Two NTP servers are configured, so one being unreachable does not leave the hub without a clock.

And the honesty rule: before the first correction of a boot, the clock reads as *unknown*, and the records written in that window carry `epoch_s = 0`. The dashboard **counts untimed cries but never places them on a timeline** — the contract test holds this as a first-class property, because an untimed cry placed on the wrong day would be a lie the page tells in pastel colours.

The network stage — `crynet_net`

One radio, two jobs, one non-negotiable rule

The cot nodes and the hub talk ESP-NOW on channel 6, fixed (Chapter 9). Joining a Wi-Fi network moves the radio to the router's channel — and if that is not 6, the cot link dies while the dashboard looks perfectly healthy. That failure mode is the one real hazard of this whole subsystem, and the firmware defends against it outright: **the moment the station associates, `crynet_net` checks the channel, and if it is not 6 it drops the connection, puts the radio back on 6, and says what to do about it.** The router's 2.4 GHz channel has

to be set to 6. That is a one-time change in the router's admin page, and it is the price of one radio doing two jobs.

Two build-level facts belong beside it:

- the station interface is created **before** the radio starts, not after — the original order failed in the field and became D-98, along with refusing uploads reported by HTTP status rather than as a silent retry;
- BLE and Wi-Fi share the one radio, so software coexistence (`CONFIG_ESP_COEX_SW_COEXIST_ENABLE`) arbitrates airtime between the ESP-NOW link and the BLE connection — and the burst measurements of Chapter 9 were re-checked with a Wi-Fi network joined.

Configuration lives in NVS, and the token lives off Git

Nothing about the network is compiled into the image. The hub's serial console carries the words:

```
net status                      what the network stage knows right now
net set <url>                   where records go, e.g. http://192.168.1.50:8787/api/v1/events
net ssid <ssid> <password>      which network to join
net token <token>               the shared secret, sent as X-CryNet-Token
name <name>                     the name the caregiver sees on the dashboard
log / log clear                 read or clear the cry log
logtest                         the ordering proof, on the chip
```

All of it survives a reflash. The token deserves its own sentence: it was once a string in a tracked file, and it was removed from Git deliberately (D-103) — it now exists only in the service's secret store and in the hub's NVS. A credential that authenticates uploads does not belong in a repository anyone can read.

The uploader

The uploader posts batches of 32 records, up to 32 batches per turn, oldest first, and **marks records sent only when the dashboard acknowledges them** — a dropped connection costs a retry, not a record. It is the watermark rule of D-94, running. `https://` targets are supported with the full certificate bundle (about 60 KB of flash, read only when a connection is made), so the hub can verify who it is talking to.

The presence report

The uploader answered one question — *what happened?* — but the dashboard also needs the other one: *is the system even on?* A dashboard that only learns of a hub when a cry happens cannot tell a quiet baby from a dead hub. So once every 30 seconds (D-105), whether or not there is anything to record, the hub posts who is here: itself, its firmware, its name, and for each cot whether it is answering, whether it has gone deaf, and how strong its radio is.

The cadence was picked twice. The first cut (D-104) was 15 seconds — a hub switched off reads as offline in about a minute (15 s report, 45 s staleness, 10 s page poll). That was too eager: the report wanted the radio while a cot's 64 KB burst was in flight. The final rule is that **the presence yields to the cots**: the hub reports every 30 s, a report may wait up to 20 s for a burst to finish (the link stage registers the guard that knows; a stuck guard can delay a report but never stop the reports), and the service calls a hub gone after 90 s. Nothing costs the cots anything, and a hub that is switched off still reads as gone in about two minutes.

The panel carries its own view of the same fact: a wireless mark drawn **only while the hub is on the network** — green when it is, nothing at all when it is not (D-100, redrawn in D-103). A dim grey "off" glyph was tried first and rejected: on a device that is *usually* offline by design, a permanent badge trains the eye to ignore it.

The freshness rule (D-108)

The first bench nights produced the one lie a monitor must not tell: the hub had been switched off for a while, the panel said hub offline — and both cots *online*. The cause was structural. The presence store keeps one row per cot, and that row is the hub's last word about that cot — but the row carried the hub's verdict (`online: true`) with no freshness check of its own, so the post the hub made just before a power cut held its cots "online" for ever.

The rule now: **a cot is online only while the hub's word about it is fresh.** The row is stamped when it is written, and on the same clock the hub itself is judged by — 90 seconds — a row the hub has stopped refreshing reads offline. A row with no stamp reads offline too; the default is closed. The hub's own row already had this discipline; the cots' rows were missing it, in both copies of the service. The regression test that proves it *fails on the old code* — aged store, old server, "cots [1, 2] read online long after the hub that sees them stopped speaking" — and that failure-on-old-code check is part of how the fix was accepted, not an afterthought.

The dashboard — two copies, one contract

There are two copies of the service, and they must agree field for field:

- **The offline copy** — `dashboard/server.py`, Python's standard library only, SQLite, one file. It runs on anything: a laptop on the nursery Wi-Fi, a Raspberry Pi on a shelf. No dependencies, no account, no internet.
- **The hosted copy** — a Cloudflare Worker on D1 (D-95), live at `https://amazings-crynet.amazingadelowo.workers.dev` (D-97). It is the copy that works when the caregiver is not on the nursery's Wi-Fi, which is the whole point of hosting it.

Both serve the same API, documented in `dashboard/API.md`:

Endpoint	Direction	What it does
`POST /api/v1/events`	hub → service	a batch of cry-log records; idempotent on `seq`
`POST /api/v1/presence`	hub → service	who is here, every 30 s
`GET /api/v1/presence`	page ← service	the hub and its cots, with freshness
`GET /api/v1/summary`	page ← service	KPIs, per-cot, per-day, per-hour
`GET /api/v1/analytics`	page ← service	duration histogram, patterns, insights
`GET /api/v1/events`	page ← service	the record list, filterable
`GET /api/v1/devices`	page ← service	the hubs known to the service
`GET /api/v1/health`	both	liveness, for tests and monitors

Authentication has two doors. The hub authenticates with `X-CryNet-Token` — the secret from NVS. The browser authenticates with HTTP Basic auth — a username and password the caregiver types once. The two secrets are separate on purpose: leaking the upload token lets someone *write* to the record; leaking the password only lets

someone *read* it. Both live in the platform's secret store and in one git-ignored local file; neither is in the repository, and every commit is scanned for them before it is made.

The page itself (`dashboard/web/` — one HTML file, one stylesheet, one script, no framework, no build step) reads like an instrument and not like a paragraph (D-102): presence cards with per-cot signal bars, a smooth daily curve, the duration histogram, the record list with the log's own numbers. It is named for the system and wears the hub's teal, with a dark and a light palette that both passed a contrast validator (D-99). It polls presence every 10 s and renders exactly what the service says — including, after D-108, the freshness rule.

The contract test — the one thing no compiler checks

The hub builds its upload bodies by hand with `snprintf`, because ESP-IDF ships no JSON library. The page is JavaScript, the service is Python. There is no type system, interface, or schema that joins them: a field renamed on either side would fail *silently* — the hub would report success and the dashboard would show nothing.

`firmware/tools/test_dashboard_contract.py` is the check. It builds the **same body, byte for byte**, from the same fields the hub's `build_batch()` uses — formatting, rounding, field order and all — posts it to a running dashboard, and checks that what comes back out of the API is what went in. Then the presence body likewise. It covers the properties the system has already been wrong about once each:

- a repeat batch (what a retry after a dropped connection looks like) must add **nothing**;
- an untimed record must be **counted but never placed** on the day or hour axes;
- a hub that just posted reads online, with its cots online **with it**;
- a store aged past the staleness window reads offline — hub *and* cots (D-108).

It starts its own throwaway server on a free port and writes nowhere inside the repository, so it runs anywhere, any time; `--port` points it at an already-running local dashboard instead. (The hosted copy is verified live in the same terms — a real login, the real presence rows — but the test itself stays local, so it can age the store to prove the staleness rule.) It is part of the pre-merge battery, and it failed on the old server the day D-108 was fixed, which is the strongest statement that can be made about a test.

The boot banner, and the stack that was too small

Two reliability findings closed this phase, and they are a pair because the second overturned the first.

The banner was lying (D-106). The hub's boot banner is the identity check — it is how the owner knows what is running. But it reported a firmware version inherited from a stale configure, so a fresh build could describe itself as an old one; and the "restarts" the banner counted were diagnosed as power-class resets, not crashes, on the strength of a clean-power soak: ten minutes, zero restarts, zero cot flap, live telemetry on both cots, and 15 presence posts at the exact 30-second cadence.

It was a stack overflow (D-107). The very next capture caught the hub dying mid-run: a stack overflow in task `main`. The whole live system ran on one 3,584-byte stack that ESP-IDF had sized for an `app_main` that *returns* — and nothing in this system's `app_main` ever returns. The restarts were never the power supply. The stacks are now sized for what actually runs on them, and the same clean-power soak that exonerated the power now runs silent: zero restarts, presence posts on the dot. The cot's "offline for a split second", reported alongside, is a 0.1-second staleness blip that loses nothing.

The lesson recorded in D-107's row is worth repeating here: a soak proves the configuration that ran during the soak. The banner fix and the power diagnosis were real, but the stack overflow sat underneath both, waiting for a load pattern the soak had not produced. The connectivity feature stayed in the system because after both fixes the soak was repeated and held.

What this subsystem deliberately does not do

- **No audio leaves the room.** The records are metadata — numbers, durations, confidences. There is no microphone feed on the network, no recording of the child, nothing to leak.
- **No remote control.** The dashboard reads. It cannot silence the wrist, change a threshold, or reconfigure the hub — every one of those is a physical act on the unit, on purpose.
- **No cloud dependency for detection.** With the network unplugged, detection, alerting, logging and display are exactly what Chapter 6 describes; the dashboard empties when it can.
- **No router gymnastics beyond one setting.** The one deployment requirement is that the router's 2.4 GHz channel is 6. Everything else about the network is optional.

Chapter 13 — Hardware: wiring, bill of materials, power and the haptic drive

The four boards and their pins

Cot node — ESP32-C3 Pro Mini + INMP441

INMP441 pin	Node pin	Note
VDD	**3V3**	3.3 V only
GND	**GND**	
L/R	**GND**	selects the **left** channel — this is *channel select*, not gain
SCK	**GPIO4**	I²S bit clock
WS	**GPIO5**	I²S word select
SD	**GPIO6**	I²S data in

No MCLK — the INMP441 generates its own clock from SCK.

The old specification PDF assigns GPIO5/6/7 and a status LED on GPIO4. Do not wire it. GPIO4 is now the microphone's bit clock. GPIO9 is a boot-strapping pin. The C3's strapping pins are **GPIO0/2/8/9** and the design avoids all of them.

Free pins since D-83: GPIO3 (was the battery ADC), GPIO7 (was the SOC LED), GPIO10 (was the link LED). Nothing drives them.

Microphone placement — a rule with a measurement behind it

The INMP441 is **fixed-gain**: no gain register, and its own output saturates before the node's 16-bit window does. Measured on the bench:

- node 2 clipped at **peak 31,801 of 32,767** on ordinary room sound;
- a phone held **10 cm** from the microphone clipped the cry it was playing — that window scored **0.9805 instead of 0.9961**, and **the alert named the wrong cot for 42 seconds**.

There is no software cure, so the assembly rules are:

- Mount on the cot frame or a wall bracket, on a short stiff lead, **at least 30 cm** from where the baby's head will be, and out of the mattress's own sound path.
- **Never leave it loose on the mattress.** A soft surface damps the high frequencies the model reads, and the cot frame moves.
- Keep the acoustic opening unobstructed — an enclosure with a sound port, not a sealed box.

Hub — ESP32-S3-DevKitC-1 N16R8 + 2.8" ILI9341

TFT pin	Hub pin
CS	**GPIO10**
MOSI	**GPIO11**
SCK	**GPIO12**
DC	**GPIO13**
RESET	**GPIO14**
LED / backlight	**GPIO4** (LEDC PWM, 55 % default)
MISO	**not connected** — the panel is write-only
VCC / GND	3V3 / GND

SPI2 at 40 MHz. **Do not add the old document's 100 Ω backlight series resistor** and do not apply 5 V to an unidentified TFT pin: module backlight circuits differ, and GPIO4 is a *control* signal, not an instruction to drive an unknown backlight directly.

Free pins since D-83: GPIO1, GPIO2, GPIO17 (were BLE-status and SOC LEDs).

Wearable — ESP32-C3 SuperMini + SSD1306 + coin ERM

Function	Pin
OLED SDA	**GPIO5**
OLED SCL	**GPIO6**
Motor drive	**GPIO7** → 2.2 kΩ → BC547 base

The haptic circuit, in full:

```
GPIO7 ──[2.2 kΩ]── B  BC547  C ── motor(-)
                   E ── GND
        100 kΩ from base to emitter (pulldown)

3V3 ──[22 Ω]── motor(+)
1N5819 across the motor, CATHODE UP (to motor +)
```

Part	Value	Why this value
Base resistor	**2.2 kΩ**	(3.3 − 0.7) / 2200 ≈ **1.2 mA** of base current; a BC547A (hFE ≥ 110) saturates and sinks the motor's running current
Base–emitter pulldown	**100 kΩ**	the motor stays off while the C3 boots with GPIO7 floating
Stall cap, in the motor's **+ lead**	**22 Ω**	3.3 / (37 + 22) ≈ **56 mA** stall, inside the BC547's 100 mA rating
Flyback diode	**1N5819**, cathode up	clamps the inductive kick when the transistor turns off

The 22 Ω is the value that is fitted, not the 10 Ω an earlier list named. The owner had a 22 Ω and no 10 Ω, and the bench proved the drive works with it on **2026-11-11**. It is also the safer value: 10 Ω would allow ≈70 mA. The cost is that the motor sees about **2.1 V** of the 3.3 V rail instead of about 2.6 V, so **the buzz is softer**. That trade is recorded, not hidden.

If a different motor is fitted, **measure its stall current with a bench supply and a meter in series before soldering — do not guess.** For anything lower-impedance than 37 Ω, fit a 2N2222A instead of the BC547.

Both the OLED and the motor run from the SuperMini's 3V3 output, not from 5 V. The OLED's I²C pull-ups ride on its VCC rail, so 5 V there would push 5 V levels into non-5 V-tolerant C3 GPIOs.

The DRV2605L breakout and the IRLML2502/AO3400 MOSFET with its 100 Ω / 10 kΩ gate resistors belonged to earlier alternatives and are **not in this design**.

The power chain, and its three rules (D-84)

Every unit is powered the same way:

```
cell → LX-LCBST-TC-001 boost to 5.1 V → VO+/VO− → switch → board's 5 V pin
     → the board's own linear regulator → 3.3 V
```

The LX-LCBST-TC-001 pad map

Confirmed from the owner's photograph of the actual board:

Pads	Printed	Meaning	Connection
B+ / B− (front and back)	电池	**battery**	the cell
VO+ / VO− (two pairs, back)	输出	**output**	boost output → the power switch
IN+ / IN− (back)	输入	input	alternate charging input, 4.2–6.5 V — **leave unconnected**
USB-C socket	—	charging	charging runs from here

The switch goes on **VO+ → board 5 V**, never on the battery side, so B+/B− stay connected for charging while the unit is off.

Rule 1 — set the trimpot before any ESP32 is connected

Battery attached, meter on VO+/VO−, adjust to **≈ 5.1 V**. The module can ship set anywhere in its **4.2–28 V** range; connecting a board before setting it **can destroy the board**. Re-check after the first full charge cycle and occasionally thereafter.

Rule 2 — never connect the boost and USB at the same time

The boost output goes straight to the board's 5 V pin, and that pin is hard-wired to the USB 5 V rail through the board's own protection diode. With both connected the two supplies fight: 5.1 V and USB 5 V are the same net, and **current can be pushed back up the USB cable into the host** — a fault on the laptop's side, not just the board's.

To log while running on battery, use a USB cable with the 5 V (VBUS) wire cut. The data lines still carry the serial log and nothing can be pushed back into the host.

Rule 3 — the linear regulator is the weak point, and it gets hot

A linear regulator passes *current*, not power. The difference between 5.1 V in and 3.3 V out is burned as heat inside the board: **(5.1 − 3.3) × I**. At the C3's radio peaks of a few hundred mA that is **0.5–0.7 W** in a package with no heat sink.

Two consequences:

- The cell must supply about **1.8× the 3.3 V load current** (D-82). A 100 mA load at 3.3 V means ≈160 mA from the cell.
- **If a unit resets when the radio transmits, check the LDO's heat and the cell's sag first — not the radio.**

Runtime projection (not a measurement): a 1000 mAh pouch ≈ **six hours**; a 3000 mAh 18650 rather less than a day at the hub's heavier load. *No runtime has been measured on this system.*

The largest single lever on runtime is to **skip the 5 V → 3.3 V linear stage**: feed the board's **3V3 pin** from a 3.3 V buck-boost, which removes the dissipation entirely and roughly doubles runtime. It needs a separate module, because the LX-LCBST cannot be set below its 4.2 V minimum. Recorded as an option to evaluate at the bench (D-82), **not** something to rewire before the current build is finished.

Cell choice (D-81), and why it differs per unit

Unit	Cell	Reason
Nodes ×2	**protected 18650 Li-ion, 3000 mAh**	a cot frame can puncture a pouch; the holder must survive a rail being knocked and the cell must not move
Hub	**protected 18650 Li-ion, 3000 mAh**	the heaviest load; the cell most worth sizing carefully
Wearable	**protected Li-Po pouch, 1000 mAh**	an 18650 is 18 × 65 mm and 48 g — it will not sit on a wrist. 500 mAh was too small for a night shift

Never LiFePO4 anywhere in this system. The module charges to **4.2 V**; an LFP cell tops out at 3.65 V. Same chemistry otherwise: 3.7 V nominal, 4.2 V full.

The module's ~1 A charge current is **too high for the 1000 mAh pouch** — it must come down to **~400–500 mA**. Confirm during bench validation.

And the invariant that survives every other change: **the battery voltage must never reach an ADC pin directly.** The C3's IO is 3.3 V only and a full cell is 4.2 V.

What was removed, and why (D-83)

The battery sense and **every status LED** were designed, costed, and then **withdrawn before they were fitted**:

- the fixed **220 kΩ / 100 kΩ** divider (×3.2 ratio) and its **100 nF** filter;

- the node's divider read and its serial `battery` line;
- the node's link and SOC LEDs, the hub's BLE-status and SOC LEDs, the wearable's divider;
- the telemetry field — the payload went **34 → 32 bytes**.

The reasoning: the hub's panel already shows which cots are answering, and the battery figure was always an estimate — the reading sagged **100–200 mV** under the boost and radio load, so a linear voltage map would have been misleading anyway.

The cost is stated plainly: nothing in the system will warn the caregiver that a cell is nearly flat. The cells are watched by eye.

If a readout is ever wanted back, the honest part is a **MAX17048** fuel gauge on I²C — model-based SOC, no calibration table, indifferent to terminal sag under load — not a divider.

Do **not** buy the divider resistors, the divider capacitor or the indicator LEDs.

Consolidated bill of materials

Component	Total	Note
ESP32-C3 development boards	**3**	2 nodes + 1 wearable
INMP441 microphone breakouts	**2**	one per cot
ESP32-S3-DevKitC-1 with PSRAM (N16R8)	**1**	16 MB flash, 8 MB PSRAM
2.8" ILI9341 SPI TFT	**1**	320 × 240
0.96" SSD1306 I²C OLED	**1**	128 × 64
BC547	**1**	haptic switch
1N5819	**1**	flyback, cathode up
22 Ω	**1**	stall cap, motor + lead
2.2 kΩ	**1**	base
100 kΩ	**1**	base–emitter pulldown
Coin ERM vibration motor	**1**	~37 Ω; measure the stall current
LX-LCBST-TC-001 charger+boost	**3 + 1 candidate**	bench-validate **one** before buying the rest
Protected 18650, 3000 mAh	**3**	2 nodes + hub
18650 holders with leads	**3**	
Protected Li-Po pouch, 1000 mAh	**1**	wearable
On/off switches	**4**	≥1 A, on VO+
100 µF electrolytic, ≥10 V	**3 + 1**	5 V rail bulk
100 nF ceramic (104)	up to **3**	optional decoupling; check what the modules already carry
4.7 kΩ	up to **2**	I²C pull-ups **only if** the OLED module's are inadequate
Battery connector pairs	**4**	verify polarity physically, not by colour
Enclosures	2 / 1 / 1	node / hub / wearable + strap
Multimeter	**1**	verify supply, continuity, polarity and shorts before powering

The board swap is done, and it was verified live. The two cots run **ESP32-C3 Pro Mini** boards from the owner's stock; the original SuperMini prototype moved to the wearable. On **2026-11-02** the same node firmware was flashed onto a Pro Mini *unchanged* — native USB (`303A:1001`), 4 MB flash — and the bench gave link-up at once, **286/286 fragments** served, a quiet room refused, and **two cry plays producing two alerts**. The two boards on the bench are `90:DA:72:85:C5:44` (id 1, COM31) and `90:DA:72:87:CB:98` (id 2, COM37).

Why no firmware changed: the FQBN `esp32:esp32:weact_studio_esp32c3` selects the **ESP32-C3 chip** and its flash and USB settings, not one physical module. Both boards are the same chip with the same 4 MB flash and the same native USB, so the image is identical. What differs is the silkscreen: **confirm the pin labels against GPIO4/5/6 on the actual Pro Mini before soldering**, because the electrical assignment does not change but the printed names may sit in different places. This is also why one toolchain and one spare part still cover all three C3 units.

The haptic drive — what is proven, and the one thing that is not

The motor **works**. It is worth being precise about how that was established, because "the buzzer buzzes" is the kind of claim a defence gets pushed on.

Proof 1 — the electrical path alone. `firmware/wearable/haptic_test/haptic_test.ino` drives GPIO7 with the same 250 ms pulse train the alert path uses and nothing else: no BLE, no OLED, no hub. One command flashes it:

```
python firmware/tools/flash_node.py --port COM30 --sketch haptic_test
```

The motor was seen to toggle. That isolates the BC547, the 2.2 kΩ base resistor, the 100 kΩ pulldown, the 1N5819, the 22 Ω cap and the motor from every line of application code. The sketch stays in the repository with its restore command in the header, so a fresh board can be proved without hunting for the procedure.

Proof 2 — the whole chain, on 2026-11-06. Three alert cycles in a row came through complete: `ALERT cot 1 type 1 conf 100%` on the wrist, the haptic fired, `clear` returned `type 2`. The wearable also survived a hub reset mid-link and resubscribed to both characteristics.

Proof 3 — the fitted cap value, on 2026-11-11. The bench proved the drive works with **22 Ω** in the motor's positive lead, which is the value that is actually soldered (D-85 revises D-77's 10 Ω).

Proof 4, and it is only half a proof — the cell. The wearable **has run on its Li-Po pouch**: it was on the pouch, not USB, for the third round of the 2026-11-20 session, and it kept working. But **the buzz went missing in that round.** The firmware is exonerated — D-86 touches no motor line, so the drive is byte-identical to the build proved on 2026-11-11 — which makes a **sagging 3.3 V rail** a live candidate: the OLED, the C3's radio peaks and the motor all share the board's one linear regulator. A coin motor's inrush is the largest transient in the system, and a pouch through a boost and a linear regulator is a stiffer supply than a USB port. That is O-1d, and the measurement is short:

Reading	What it settles
Multimeter **across the motor during an alert**	voltage present and no motion → the motor or its joint; no voltage → the drive path
Pouch voltage during a buzz	a collapse there implicates the supply, not the switch
Cell voltage at the board's 5 V pin under a radio peak	whether the LDO is dropping out (Rule 3)

One inconsistency in the record, stated rather than smoothed over. The bill of materials describes the fitted actuator as a *3 V ERM coin motor, 85 mA*, while the stall arithmetic in D-85 assumes a **37 Ω** coil — and 3.3 V across 37 Ω is 89 mA, which is a *stall* figure, not a running one. The two numbers cannot both be the motor's running current. The practical consequence is that the 56 mA "stall" figure is a calculation from an assumed coil resistance, not a measurement of the motor on the bench. **Measure the actual stall current with a bench supply and a meter in series** before relying on the headroom argument; if it exceeds the BC547's 100 mA rating, fit a 2N2222A.

Safety

This is an **academic prototype**, not a certified medical device and not a substitute for an adult caregiver. Detection, range, radio delivery, battery life and alarms have not been established to any safety-critical standard. Keep electronics, cords, batteries and small parts out of an infant's reach.

Chapter 14 — The hub panel: driver, UI, and five debugging rounds

Why a panel at all

The wearable is the alerting device, but it is one device on one wrist. The panel is the system's **shared truth**: which cots are linked, which are crying, whether the caregiver's device is attached, and whether an alert actually got delivered. It is also the only place the system shows anything at all, since D-83 removed every LED.

Hardware: 2.8" ILI9341, 320 × 240, SPI2 at 40 MHz, write-only (MISO unconnected).

The driver

`ili9341.c` (413 lines) is a hand-written driver, not a library. Three reasons: the panel needed a specific rotation and colour order, the framebuffer had to live in PSRAM, and the backlight needed LEDC PWM control. A generic library would have fought all three.

- **Framebuffer:** 320 × 240 × 2 = **153,600 bytes of RGB565 in PSRAM**. The whole frame is composed in memory and pushed in one SPI transaction, so there is no tearing and no partial state on the glass.
- **Rotation:** `ROTATION_LANDSCAPE 2`.
- **Backlight:** LEDC PWM, `BACKLIGHT_DEFAULT 55` %, via `hub_display_set_brightness()`.
- **BOOT button** (`PIN_BUTTON 0`, active low) cycles **three faces**: `s_face = (s_face + 1) % 3` — the dashboard, the help page, and the colour check. The colour check is a grid of the palette's swatches, so the panel's own colours can be judged **without a camera in the loop**; the help page explains what every number on a cot card means.

Five rounds of debugging, and what each one taught

This sequence is worth telling in order, because each fault was invisible until the previous one was fixed.

Round 1 — static noise

The first flash produced **static**. Three bugs in the driver, all found by reading the ILI9341 command sequence against the datasheet:

1. the address window was written **little-endian** (the panel wants big-endian);
2. `CASET` **was sent twice** where the second should have been `PASET`;
3. **CS was released between a command and its parameters** — the panel treats each CS assertion as a new transaction, so the parameters were interpreted as commands.

Round 2 — a quarter turn out

The image appeared but rotated. This panel is **landscape-first** and wants `MADCTL_MY`, not the `MADCTL_MV` the legacy firmware used. A legacy setting is not evidence about a different module.

Round 3 — hearts and diamonds instead of letters

The new UI drew, but the text was symbols. **The font generator was reading the table 32 characters out of place** — an ASCII offset bug in `research/build_font.py`, not in the driver. The glyphs were correct; the indexing was not.

Round 4 — red and blue exchanged

The first photograph of the real glass confirmed the layout, text and orientation were all correct: `AMAZING'S CRYNET`, `S3 HUB`, `RADIO`, `PHONE`, `LISTENING`, `COT 1 / ONLINE`, `COT 2`, `2 cots linked`, `up 0:39`.

It also exposed a colour fault: **every colour had red and blue exchanged.** The teal accent came out olive; the amber `PHONE` pill came out blue. Cause: the driver set the panel's `MADCTL_BGR` bit while the framebuffer packs standard **RGB565**. Fixed, flashed and logged (`madctl 0x80`). The second photograph confirmed it: the `PHONE` square warm orange, the accent underline teal-green, the `RADIO` square cyan-green — the same elements that had been blue, pink and olive.

Round 5 — the panel lifts the dark end

The second photograph also showed a **glare**: the panel was brighter than the room, the camera clipped it, and the white balance drifted so the whole screen read as saturated blue. The panel's side of that was fixed with the PWM backlight at 55 %.

Then the **colour check** — the swatch grid, judged by eye rather than through a camera — proved the hues were right and found the real fault: `BLACK` black, `WHITE` white, `TEAL` teal, `GREEN` green, `AMBER` amber, `ROSE` rose — but the old navy background came back as a **saturated blue**. **This panel lifts the dark end, and the blue channel hardest**: the amber's requested blue of **36 measured back at 145**.

That is a property of the glass, not a bug in the code, and it cannot be fixed by correcting a colour constant. The fix was to **move the whole neutral family down the ramp** until the lifted values land where they should.

The palette, and how it was chosen

The neutrals were moved one step darker **on the user's own pick**: shown the ramp, they chose its **third step,** `(4,8,8)`, as the teal the screen should be, and the whole neutral family moved down one step with it. Then a second pass from the chair (D-59) dropped the background another step, `(4,8,8)` → `(2,4,4)`, because the cards still read as the same colour as the ground on the glass.

Name	RGB	Use
`C_BG`	**(2, 4, 4)**	background
`C_SURFACE`	(7, 15, 16)	card fill
`C_SURFACE_HI`	(12, 26, 28)	raised surface
`C_HEADER`	**(7, 20, 15)**	header strip
`C_BORDER`	(78, 116, 112)	card outline
`C_TEXT`	(228, 240, 236)	primary text
`C_DIM`	(122, 150, 146)	secondary text
`C_ACCENT`	**(45, 212, 191)**	teal accent
`C_ACCENT_DIM`	(22, 124, 112)	inactive accent
`C_BLUE`	(96, 165, 250)	radio indicator
`C_GREEN`	(52, 211, 153)	online
`C_AMBER`	(251, 191, 36)	phone / attention
`C_ALERT`	(251, 113, 133)	rose, crying

The header strip got its own colour, `(7,20,15)`, with green pushed five steps above blue, because the backlight's LEDs run along the **top** of the glass and lift that strip blue. A header painted the same colour as the cards photographed blue while the cards photographed teal. That is a physical property of the module discovered by looking at it, and the palette compensates for it deliberately.

Layout — two cards, one reading per row

Element	Geometry
Header	`HEADER_H 30`
Banner	x 10, y 40, w 300, h 44
Cot cards	`CARD_X 10`, `CARD_W 300`, **`CARD_H 64`**, `CARD_Y0 88`, **`CARD_STEP 70`**
Footer	`FOOT_Y 230`
Splash	**`SPLASH_MS 5600`** = `BOOT_MS 2400` + title + `FADE_MS 600`, `SPLASH_TICK 100`

Two full-width cards, and the arithmetic is deliberate: `88 + 2 × 70` puts the second card's bottom edge at **222**, leaving the footer at 230 clear. The cards used to be 44 tall on a 47 pitch, which fitted three of them and put three readings on one line. With two cots there is vertical room to give each reading its own row, and that is what the redesign spends it on:

One cot card, read row by row:

Row	What it shows	Example
Top	the cot's name and its state	`COT 1` `ONLINE`
Middle	the **level bar** with the **notch** where that cot's trigger sits, and the RMS beside it	`RMS 0.012`
Bottom	the **signal bar** and the RSSI of the link to that cot	`LINK -58 dBm`

The RMS used to share its line with the link, which put two readings on one row and made the card hard to read in a dark room. **There is no battery line** — the cots do not measure their cells (D-83).

The level bar, its scale, and its notch

`LEVEL_BAR_FULL` is **0.25**, not 1.0, and the scale is **square-rooted**:

```
const float frac = sqrtf(v / LEVEL_BAR_FULL);
```

Both choices come from measurement. An RMS of 1.0 is a clipping signal, and a cry does not clip: the loudest genuine cry measured on the bench carried **0.18**, the quietest one the system must catch **0.010**. Against a full scale of 1.0 even a scream filled a third of the bar, so the bar said nothing and only the number beside it was worth reading. At 0.25 a scream fills most of the bar, a cry at one metre about two thirds, and a quiet room a sliver. The square root is there because **the interesting part of this bar is the bottom fifth of the scale, where the trigger sits** — a linear scale squashes exactly the region the caregiver needs to see.

The **notch** is a one-pixel vertical line at the cot's own trigger level, drawn through the bar:

```
const int nx = x + 1 + (int)(sqrtf(t / LEVEL_BAR_FULL) * (float)(w - 2) + 0.5f);
fb_fill(nx, y, 1, h, C_TEXT);
```

It is the same `sqrtf` mapping as the fill, so the notch and the fill agree by construction. The bar's colour carries the same fact: **green** while the cot is below its trigger (only being listened to), **amber** once it crosses it (audio is going to the model), **rose** while the hub is announcing a cry from it. That is what turns the bar from "a number in colour" into a picture of the decision the cot is about to make.

The field is called `rms`, and the *threshold* is called `trigger`. This distinction is load-bearing in the project's vocabulary: **"level" means the threshold, never the loudness.** The banner's bare number is labelled `RMS 0.012` for the same reason — a bare number in a corner reads as a clock.

The signal bar

Four bars, 3 pixels wide on a 5-pixel pitch, heights 4/7/10/13, unfilled bars in `C_SURFACE_HI`:

```
const uint16_t col = (bars >= 3) ? C_GREEN : C_AMBER;
```

Bars	Threshold	Colour
4	≥ −60 dBm	green
3	≥ −70 dBm	green
2	≥ −80 dBm	amber
1	≥ −90 dBm	amber
0	below −90, or no reading	amber, all four empty

The dBm is printed beside it (`LINK -43 dBm`), and `LINK --` when there is no reading at all — the sentinel is `0`, which is not a plausible dBm value. The number and the bars are the same reading, and the help page says so explicitly, because a caregiver who sees four bars and −43 dBm should not have to wonder whether they are two different measurements.

This reading is the RSSI of the cot's **telemetry as heard at the hub**. The cot prints its own bar on its serial console from the RSSI of the hub's **heartbeat as heard at the cot**. Two directions, two numbers, and they are not expected to match.

The help page

The BOOT button's second face exists because of an obvious question: the panel says `RMS 0.023` next to a bar with a notch in it, and `LINK -43 dBm`, and none of those words mean anything to a caregiver at three in the morning who cannot go and look up what an RMS is. Twenty lines of small text, ten pixels apart, explain the four readings in plain language — a quiet room near 0.005, a cry at one metre about 0.03 to 0.05, the notch as "the level that cot sends audio over", −60 and better as strong, −90 and worse as barely there. The footer reads `press BOOT to go back`.

The splash, and the project's identity

The panel boots through **two** faces before the dashboard: a boot face (the waveform, the breathing wordmark, the boot steps, a filling progress bar) and a title card. The title card carries `AMAZING'S CRYNET` in accent teal, the strapline, the full project title across three lines, **AMAZING AYOMIKUN ADELOWO**, the id **22/05/04/0029**, and along the bottom:

```
ESP32-S3 HUB  -  TWO COT NODES
```

The waveform moved up and shrank to make room for the stacked name (`WAVE_Y` 84 → **46**, `WAVE_MAX` 26 → **12**), and nothing collides: the wave ends at 59, the name runs 68 to 116, the strapline at 134, the boot step at 190, the bar at 206, the footer at 222.

The footer counts cots, not cards

`snprintf(foot, …, "%u cot%s linked", online, …)` — the count is the number of cots **answering**, not the number of cards drawn (D-54). The bug this fixed is worth remembering: the panel said `3 cots linked` above cards that all read `NO SIGNAL`, because it was counting slots. With two fixed slots that bug would have been permanent and invisible.

Online versus offline treatment

Swapped on the user's request, then amended:

- a **live** cot carries the **full-strength vibrant teal outline** (rose when crying) and the *denser, more active* border-washed fill;
- an **offline** cot recedes — outline and empty bar washed towards the background, the faint accent wash for a fill — and its word is **OFFLINE**.

The original treatment blended live cards at 45 %, which made the *active* cards the faint ones. That is backwards: the eye should be drawn to what is alive.

The caregiver's device on the panel

Round 4 added the `PHONE` pill and, under a confirmed cry with nobody attached, the words `ALERT NOT DELIVERED`. A cry the system detected but could not hand to a human is not a success, and the panel says so.

The wireless mark, drawn only when it means something

The header carries a wireless mark — two arcs over a dot, the shape every phone shows for a connection — and it is drawn **only while the hub is on the network** (D-100, redrawn in D-103): green when it is, and *nothing at all* when it is not. A first version drew both states, green and dim grey, and that was rejected on a principle the rest of the panel already follows: a mark that is always there has to be read to be understood. With the mark present exactly when it means something, the panel has one rule — *a mark here means the hub is on the network* — and no second state to learn. Where the hub is offline, the dashboard says so instead; there the question has room to be answered.

Two implementation details were not obvious. The arcs are **walked by angle, not by column** — a column-by-column walk of a shallow arc leaves gaps at the sides, and the sides are where the eye looks first. And the flag that draws it is **the same one the uploader consults** before sending a record, so the glass cannot disagree with the network: the panel shows a mark only when the records are actually on their way (Chapter 12).

The rounded corners, and a host-side regression check

Every rounded corner had been drawing its arc on the **inside** of the shape — only the bottom-right one looked right — because the corner loop walked its offsets inward from each disc's centre. Each disc is now walked **towards its own corner**.

The fix is protected by `research/check_corners.py`, a host-side check that requires the four corners to be **exact mirror images of each other**. That is the right shape for a graphics regression test: not "does it look right" but a symmetry property that must hold, checkable without a display.

What the panel is not

It is not a touch interface, not a configuration surface, and not required for the system to work. `hub_display_start()` failing produces a warning and the hub carries on: *"a dead display does not stop the monitoring, it only stops the showing."*

Chapter 15 — Every measured result, and how it was obtained

How to read this chapter

Every number here was produced by a run that is still reproducible, or is quoted from a log still in the repository. Where a figure is a **projection** rather than a measurement, it says so. Where a figure changed after a retrain, **both** values are given — the older number is not erased, because the difference between them is itself a result.

The single most important caveat, stated once here and applicable to every positive-case number below: **the model has never heard a real infant cry.** Every cry it has been tested against is a recording played through a phone speaker — compressed, band-limited and EQ'd by a small loudspeaker in a plastic case. The negatives are authentic (this room, this microphone); the positives are a proxy. This is O-2 and it is the largest accuracy risk in the project.

Model quality — offline, on the corpora

Claim	Number	How it was measured
ICD validation AUC	**0.9994**	full Kaggle run, fold 4
ICD test AUC	**0.9993**	full Kaggle run, fold 5
Cross-corpus AUC	**0.9984**	cry4 hold-out + ESC-50 + snoring + music + speech, unseen corpora and unseen babies
INT8 vs float32	**ΔAUC −0.00137**, decision agreement **99.956 %**	11,386 windows, host
Deployed artefact	**137,384 bytes** INT8	SHA-256 `bd592d54…dce43b9`
INT8 vs float32, inside the export	val AUC **0.9978 vs 0.9991**	`deploy_config.json`
Parameters	122,770	`model.count_params()`

The cross-corpus figure is the one to quote in a defence: it is the honest estimate of what happens in a room the model has never seen.

The front end — parity and speed

Claim	Number	Method
C front end vs Python	max **6.9×10^{-5}**	4 probes, real run on the board
On-chip feature parity	worst **≤ 1 LSB**, **≤ 1 %** of 5,184 features differing	boot probes, both conditions required
On-chip front end	**41.7 ms** per 2 s window	ESP32-S3, 240 MHz, `-O2`
On-chip inference	**101.8 ms** per window	ESP-NN SIMD kernels
Front end before the SRAM move	135 ms	scratch buffers in PSRAM — the measurement that justified moving them
Tensor arena	262,144 B reserved, **108,328 B used**	`ArenaUsedBytes()`

Against the deployed INT8 model, on real audio

Test	Before the music retrain	**After (deployed)**	Tool
ESC-50 `crying_baby` recall	36 / 40	**36 / 40**	`research/eval_esc50.py`
ESC-50 non-cry false alarms	2.64 % (38 / 1,439)	**1.18 %**	`research/eval_esc50.py`
GTZAN music false alarms	**8.19 %** (79 / 965)	**0.21 %** (2 / 965)	`research/eval_music.py`
Quiet room, this INMP441	0 / 236 windows	**0 / 236**	`research/eval_indomain.py`
Phone-cry per-window recall, session 1	0.867	0.800	`research/score_demo_clips.py`
Phone-cry per-window recall, session 4	0.724	0.592	`research/score_demo_clips.py`

The last two rows are the **cost** of the retrain. Per-window recall fell; episode recall did not, because a sustained cry trips many consecutive windows.

The music breakdown that caused the retrain

Before: jazz **14.3 %**, blues **13.0 %**, pop **12.0 %**, disco **10.0 %**, rock **10.0 %**; reggae 2.0 %, classical 2.9 %, country 3.0 %, metal 4.0 %.

And the number that ruled out a threshold fix: **45 of the 81 music alarms and 34 of the 36 true cry detections carried exactly the same value, P(cry) = 0.9961.**

The in-room confusion session — O-9, closed

Nine one-minute takes recorded with `node_capture` on the real INMP441, cut into 5 s clips with a 2.5 s stride (22 clips, 44 windows each), labelled with the **ground truth of the session** rather than by loudness (`label_session.py --truth`). Procedure: `research/CONFUSION_SESSION.md`. Result: `research/in_domain/SESSION_REPORT.md`.

#	Take	Class	Median RMS	Peak RMS	Records ≥ 0.030	Clips reaching the model	**False alarms**	Recall
1	session5	quiet room	0.0020	0.0100	0 / 590	0 / 22	**0**	—
2	session6	speech, desk distance	0.0040	0.0530	3 / 590	0 / 22	**0**	—
2b	session6b	speech, 30 cm	0.0080	0.6670	62 / 590	6 / 22	**0**	—
3	session7	music, 1 m	0.0100	0.0380	1 / 590	0 / 22	**0**	—
4	session8	movie, 1 m	0.0070	0.0190	0 / 590	0 / 22	**0**	—
4b	session8b	movie, 30 cm	0.0260	0.2180	269 / 590	12 / 22	**0**	—
4c	session7b	music, 30 cm	0.0470	0.1410	493 / 590	20 / 22	**0**	—
5	session9	**the cry, 15 cm**	0.0370	0.3870	329 / 590	20 / 22	0	**38/44 = 86.4 %**
6	session10	**the cry, 1 m**	0.0070	0.0160	**0 / 590**	0 / 22	0	*18/44 = 40.9 % if asked*

Zero false alarms in 308 windows of non-cry audio.

The two loud takes are the ones that count. The trigger fired on **46 %** of the loud movie's records and **84 %** of the loud music's — the model was asked 44 times in each and **refused every one**. On GTZAN the music false-alarm rate had been 8.19 %; on the loudest music this room can produce it is **0/44**. That is the retrain working in the room it was supposed to work in.

The finding that changed the design

The cry at one metre never crosses the trigger. Median RMS 0.0070, peak 0.0160, **0 of 590 records** reach 0.030. The system is blind at one metre *by trigger*, not by discrimination — if the model were asked anyway it would still catch 18 of 44 windows.

That single row is why the adaptive trigger (D-63) exists. It is the clearest example in the project of a measurement redirecting the engineering: the temptation was to retrain the model, and the data said the model was never being asked.

The live chain — hub + node + a real BLE client

Claim	Number
Radio integrity	**0 bad frames, 0 bad payloads** over thousands of frames (the hub's own counters)
Burst completion	**286/286 with CRC-32 MATCH**, with a BLE client connected
Burst duration	2.06–5.19 s, **median 3.55 s**
Request → verdict	2.23–5.35 s, **mean 4.03 s**
Before the airtime fixes	5.54–6.93 s per burst
After the first airtime fix	**2.06–2.11 s**, no resend rounds
Demo session	**45 bursts, 45/45 complete**, no stall, timeout, CRC error or resend round
Verdicts in that session	**42**: 26 "ALERT — cry" at P(cry) **0.9922–0.9961**, 16 "no cry" at **0.0078–0.2148**
Quiet room + a loud non-cry sound	**P(cry) = 0.0000** on every window; the trigger fired (RMS 0.035–0.085) and the model rejected it
End-to-end alert delivered	`ALERT {'cot': 1, 'alert': 'cry', 'confidence_pct': 100}`

Note the gap in that verdict distribution: nothing between 0.2148 and 0.9922. On real audio in this room the model is not ambivalent — it is either near-zero or saturated. That is exactly the property the announcement rule is built on.

The two-cot bench session — 2026-11-05/06

This is the session that says the *two-cot* system works, as opposed to the components working. The bench map was read from each board's own boot banner with `who_is_who.py` before anything was flashed, because four different targets on four ports is four chances to flash the wrong firmware.

Unit	Port	Identity
Hub	**COM32**	CH343 USB-serial
Wearable	**COM30**	the C3's own USB, `303a:1001`
Cot 1	**COM31**	derived id 1
Cot 2	**COM37**	saved id 2, from NVS

Both ids survived the flash, which is the check that matters — the id lives in NVS and a flash could have cleared it.

Claim	Evidence
Both cots heard	`node 1 heard for the first time`, `node 2 heard for the first time`, both `is online`
Frame integrity	**993 frames off the air, 993 parsed, 0 bad, 0 bad payload**
Panel driver	`the panel is up: 320 x 240, rotation 2`
A full burst crossed and was judged	node 2: **286/286 fragments in 0.99 s**
The audio arrived unchanged	`window crc32 0xA8FCEBD5, the node advertised 0xA8FCEBD5 -> MATCH`
The model ran on it	`P(no_cry)=0.0508 P(cry)=0.9492` — no cry, room noise, **correctly refused**
Cot 1's own link bar	`link [####] -12 dBm (hub frame 445 ms ago)`
Cot 2's own link bar	`link [####] -31 dBm`
The hub's view of cot 1	`cot 1: … link -23 dBm`
The hub's view of cot 2	`cot 2: rms 0.0230 … link -43 dBm`
Alert path, exercised without a sound	`alert 2` → `ble_alert: a cry on cot 2 at 100%` → wrist `[ble] ALERT cot 2 type 1 conf 100% event #1`
Cancel path	`clear` → the matching `type 2` on the wrist
Wearable reattachment	`hub found`, `hub connected`, `subscribed to alerts` — unaided

Note the fourth row from the bottom and the two above it: **−12 dBm at the cot and −23 dBm at the hub for the same link.** Those are two different directions measured by two different receivers, and the node transmits at 8.5 dBm by design (D-65). A defence should not be caught calling that a discrepancy.

The bug this session found

The hub could plainly hear two cots, and the wrist read `0 cot(s), mask 0x00`. The state characteristic was published only when a cry started or stopped, so a cot *joining* the system never reached the caregiver, and the wearable read the packet it was given at boot for ever.

The fix was on the publishing side, not the wearable: the eight-byte packet is rebuilt every cycle and published whenever anything in it has moved — the cry mask, the deaf mask, or **the number of cots the hub can hear** — and on a slow beat besides. The sentinels were changed so the first cycle always publishes (`0xFF` for a cot count that cannot happen, `-1000` for the timestamp).

Verified on the hardware: the wrist reads `state: 2 cot(s)`, a console test alert moved the mask `0x00 → 0x02` and back `0x02 → 0x00` on `clear`, with the wearable printing the matching cancel.

Memory and build

Claim	Number
Hub RAM before the PSRAM move	59.9 % (196,140 B)
Hub RAM after	**12.9 % (42,236 B)**
Hub RAM, current build (log + clock + Wi-Fi in)	**14.3 % (46,788 B)**
Hub flash, before the two-cot fields	61.9 % (1,297,303 B of 2 MB)
Hub flash, two-cot build	63.6 % (1,335,248 B)
Hub flash, current build	**77.2 % (1,619,615 B)** — the growth is the network stack, the certificate bundle and the cry log
Free internal heap at boot	298,763 B
…after the probes	290,791 B
…with the radio up	215,155 B
…with Bluetooth up	**149,271 B**
Node flash / RAM (compiled 2026-11-23)	**983,177 B (75 %) / 166,404 B (50 %)** — the stage-2b figure of 977,617 B / 165,972 B is the day-one build, not this one
Wearable flash / RAM, before D-86	685,143 B (51 %) / 18,684 B (5 %)
Wearable flash / RAM, with D-86	**685,589 B (52 %) / 18,684 B (5 %)** — +446 B of flash, **no extra RAM**
Continuous watch on USB power	**25 minutes, no brownout or reset**; 153 windows scored, 0 alerts
Ambient room with people talking	**20.5 minutes, 0 false alarms** (`research/out/ambient_25min/summary.md`)

The adaptive trigger, measured

Condition	Fixed 0.030	3× floor
Cry at one metre	**0 trigger events/min**	**39**
Quiet room	~0	5
Loud sessions	capped at 0.030	capped at 0.030

On the boards afterwards: level tracks the room (0.006–0.016 in a noisy room), the node's log shows it settling (`rms 0.0063 floor 0.0028 level 0.0083`), and the hub asks **5–11 times a minute** instead of 1–2.

And the cost, measured: the same quiet room produced verdicts of **0.0000 and 0.9766** for indistinguishable windows. Five minutes of a noisy room after the announce-rule change and the TX power reduction: **32 verdicts, 0 false alerts**.

Clipping, measured — and the session that made it a first-class finding

Observation	Number
Node 2, ordinary room sound	peak **31,801** of 32,767 — clipped
Phone at 10 cm	clipped the cry; that window scored **0.9805 instead of 0.9961**
Consequence	**the alert named the wrong cot for 42 seconds**
Refusal threshold	`CRYNET_CLIP_PEAK` **31000** (0.946 of full scale)

2026-11-20 — "the cries aren't being detected", and it was the microphone

The owner reported two symptoms in one session: cries sometimes not detected at all, and a cry played at **cot 2** announced as **cot 1**. Both turned out to be one fault, and the fault is not in the firmware. Two node-side peak probes ran alongside a wrist capture for 300 s while a cry was played once at each cot.

Cot	Window RMS	Peak (of 32767)	Samples at rail	Crest	Verdict
1	0.1362	**32767**	29	7.3	refused
2	0.1421	**32637**	16	7.0	refused
2	0.1627	**32736**	58	6.1	refused
1 (bystander)	0.0703	24697	0	—	clean
1 (bystander)	0.0681	18090	0	—	clean

All three loud windows are ≥ `CRYNET_CLIP_PEAK`, so all three were refused at `link_stage.c:471` **before the model's score could matter**. Cot 1 also produced two *clean* windows because it heard the same cry from across the table at a lower level — and that is why the alert named cot 1: the true source's own evidence was discarded and the bystander's clean window announced first. The wrist log confirms it: exactly two alerts, both `cot 1`, and a mask histogram of **283 ×** `0x00` **and 16 ×** `0x01` **— never** `0x02`**, never** `0x03`. The delay the owner noticed is the same mechanism: the hub waits several 2 s windows until a quieter portion of the clip yields a clean one.

The correction to our own operator guidance. All three refused windows sat *inside* the 0.10–0.20 RMS band the runbook calls the target. **Window RMS is not a safe proxy for clipping.** The 2 s window includes the clip's quiet lead-in and tail, which drags RMS down while the peak stays at the rail — measured *window* crest here was 6.1–7.3, against ~2.9–3.1 for the loudest 100 ms blocks of the same clips. At crest 7 this clip's ceiling is window RMS ≈ **0.135**, not 0.20. **The instrument that matters is peak.** The fix is distance, not volume: peak falls roughly as 1/d, so 1.5–2× the phone-to-mic gap puts these windows near 0.5–0.6, clean with margin.

The fix was verified the same day. With the phone moved back, four plays across two rounds named the right cot every time — `cot 1 #10`, `cot 2 #12`, `cot 1 #14`, `cot 2 #16` — all `type 1 conf 100%`, each with its 2500 ms pulse and each followed exactly 8 s later by `type 2`. The mask histogram became symmetric: **388 ×** `0x00`**, 16 ×** `0x01`**, 16 ×** `0x02`. Cot 2 naming itself is the case that had failed. Its peaks were 0.4253/0.6859 and 0.7309/0.7614, zero samples at the rail. **Cot 1 remains the marginal microphone** — it clipped once at peak 32764 (79 samples at rail), and its clean window peaked 0.8604, only 10 % under the refusal threshold.

A design limitation this exposes, recorded and not fixed (O-1g). The clip gate is a bare peak test, so **one sample out of 32,000 discards a whole 2 s window** — in a later round seq 82 was refused on a single

sample at 32042 while its RMS 0.1901 and crest 5.1 looked otherwise usable. A count- or fraction-based gate would tolerate an isolated transient. And **a real infant crying directly at the microphone lands in the same dead zone**. The refusal is correct — the INMP441 is fixed-gain and flat-tops before the node's 16-bit window does, so the harmonics the model reads are already gone — but it is a genuine close-range blind spot that only placement or a gain change can address.

2026-11-20 — two cots at once, and the one-buzz rule (D-86)

The supervisor's question — *what does the alert look like when both cots detect a cry at the same time?* — was answered on the bench. With one clip heard by both microphones, the hub's crying mask sat at `0x03` **for fifteen consecutive seconds** (12:36:41–12:36:55), and the owner watched the wrist's alert face read the two-cot label where before it named only one cot. **Three separate two-cot events were captured** (15 s, 3 s and 7 s).

Measurement	Result
Simultaneous-cry mask	`0x03`, 15 s continuous — plus two more events
Cry announcements, whole session	**9**, across **5** episodes
Haptic pulses	**5** — one per episode
Silent announcements	**4**, all handovers between cots
Attach time	**0.4 s** (`hub found` → `subscribed to alerts + state`)
Single-cot path	unchanged: `0x01` → `COT 1`, `0x02` → `COT 2`, one buzz each

The two handovers inside the two-cot episode (`ALERT cot 2` then `ALERT cot 1`, as their levels crossed) produced **no buzz at all** — exactly as `g_type == 1 && prev_type != 1` says they should. This is the one-cry-one-buzz rule surviving the case it was written for.

A later round showed the mask self-correcting. Cot 2 clipped 3 of 6 windows while bystander cot 1 stayed clean, and the glass went **cot 1 alone** → `COTS 1 2` → **cot 2 alone**. No forced handover was involved: cot 1 RMS 0.1317 against cot 2's 0.1508 is 1.145×, under `COT_SWITCH_MARGIN` 1.5 — so the *mask* did the work, not the loudest-first switch. The system ended on the correct cot by itself.

And one cosmetic flicker, traced rather than guessed. The owner saw cot 1 go offline for a split second. `node_live()` is consulted **only** for whether to keep probing, which cot the panel names, and the card's online/offline text. It appears **nowhere** in the announce path: the crying mask uses `published_cry`, the wrist's cot count uses the latched `heard`, and the deaf mask uses the node's own microphone self-report. **A liveness flicker cannot drop a cry, misattribute one, change the wrist's cot count, or raise a spurious buzz.** Its one real consequence is cosmetic — the panel can momentarily name a different cot than the wrist. The duration was *not* measured (the hub's UART was absent), so the cause is reasoned from the code, not observed in a log.

The dashboard phase, measured — 2026-11-24/25

Claim	Number	Source
Presence cadence, clean-power soak	**15 posts at the exact 30 s cadence, zero cot flap, zero restarts in 10 min**	D-106
The restarts' real cause	**stack overflow in task `main`** — a 3,584-byte stack sized for an `app_main` that returns, running a system whose `app_main` never returns; now 16,384 B	D-107
The same soak, after the stack fix	**zero restarts, telemetry live on both cots**	D-107
D-108's live proof	production D1 rows with `at` **2,251 s old** reading "online" against a 90 s window; after the fix the same rows read offline	deploy `d6deaeda`
The owner's end-to-end cry test	**5 cries, seq 111–115, both cots, two episodes one second apart** — all present on the hosted dashboard with presence fresh	live API
The contract test	full pass, including the D-108 section, which **fails on the pre-fix server**	`test_dashboard_contract.py`
Upload idempotence	a repeated batch adds **0** records	contract test

Two of these deserve their sentences. The stack overflow is the phase's honest headline: the "restarts were power" diagnosis of D-106 was made in good faith from a real soak and was still wrong, and only the next capture caught it — the decision record keeps both, in order. And the D-108 fix is proved the strongest way a dashboard fix can be: the regression test was made to fail against the old server first, then the fix was deployed and the same live rows that lied now tell the truth.

What has *not* been measured

Listed here so the boundary of the evidence is explicit:

- **Runtime on any battery-powered unit.** The ~6 h pouch / <1 day 18650 figures are *projections* from the regulator's behaviour (D-82), not measurements.
- **A real infant cry.** Every positive is a phone-speaker proxy (O-2).
- **How the model scores a *loud* conversation.** Four non-cry windows scored 0.9688–0.9805 in twenty minutes of talking, but none was loud enough (0.080 RMS) to test the loud path. This unknown is the stated reason the loud path requires saturation.
- **The haptic's rail on a cell.** The motor was proved three ways on USB — the standalone bench sketch, three complete alert cycles on 2026-11-06, and the fitted 22 Ω cap on 2026-11-11. The wearable **has since run on its Li-Po pouch** (2026-11-20) and kept working, but **the buzz went missing in that round** and the rail's behaviour during a buzz is still unmeasured. The 56 mA stall figure is arithmetic from an assumed 37 Ω coil, not a measurement of the fitted motor. One meter across the motor during a live alert settles it (O-1d).
- **This repository's own D-86 image on hardware.** The two-cot behaviour above was verified from the sibling repository's build. The port here compiles to the identical figures — 685,589 B / 18,684 B — but no board was attached when it was ported, so `firmware/prebuilt/wearable/app.bin` in this repo has never been written to a device (O-1f).

- **The 89 % radio-busy figure with exactly two cots.** That number comes from the busiest session recorded (230 s of burst airtime in 260 s), which had more transmitters on the channel than the two-cot system does. It is an upper bound on airtime and has not been re-measured at two.
- **The I²S zero watchdog recovering a real fault.** The code is committed and fault-injection proved it executes and refills a window; it is not flashed on either cot (D-79, O-1c), and it detects exact zeros only.

Chapter 16 — Bench verification and the demonstration procedure

The distinction that governs all bench work

The log is the evidence. The wrist is the output.

The wrist buzzes once per cry and lists every cot the hub says is crying. "It buzzed" therefore proves much less than the log line does: it proves an alert was published, not that the *right* cot was named, not that the model was confident, and not that the window was clean. Every verification procedure in this project reads the hub's serial log.

Setting up a bench session

1. **Flash all units** (Chapter 20). One command per unit.
2. **Open the hub's log** — `firmware/tools/listen_hub.py`, or any serial terminal at 115200.
3. **Open the node's log** alongside it — `firmware/tools/listen_node.py`. The node's log carries the floor, the level and every frozen window's own RMS; without it you cannot tell a trigger problem from a model problem.
4. **Attach a BLE client** — the wearable, or `firmware/tools/ble_alert_test.py` on a laptop.
5. `firmware/tools/capture_multi.py` captures several ports at once with timestamps, which is what makes a two-board timing measurement possible.

The demonstration procedure

From `DEMO_RUNBOOK.md`, in order.

Once, before starting

Set the phone's distance, not just its volume, and do not touch either during the demonstration. The runbook's stated target is a hub reading of **0.10–0.20 RMS**. **That target is not safe on its own**, and this is the single most important operational lesson in the project.

> **Measured on 2026-11-20, and it contradicts the runbook.** Three windows were refused for clipping at peaks **32767, 32637 and 32736** — and *all three sat inside the 0.10–0.20 RMS band the runbook recommends*. The quantity that decides refusal is the **peak** (`peak >= 31000`), not an RMS bound. The 2 s window includes the clip's quiet lead-in and tail, which drags RMS down while the peak stays at the rail: measured **window crest was 6.1–7.3**, against ~2.9–3.1 for the loudest 100 ms blocks of the same clips. At crest 7 the ceiling is window RMS ≈ **0.135**, not 0.20. **What went wrong on the bench:** cries were sometimes not detected at all, and a cry played at **cot 2 was announced as cot 1** — because cot 2's own saturated windows were refused and the bystander cot's clean window, hearing the same cry across the table, announced first. **The fix is distance, not volume.** Peak falls roughly as 1/d, so **1.5–2× the phone-to-mic gap** puts these windows near peak 0.5–0.6, clean with margin. Aim to keep the **peak under about 0.75 of full scale**; treat the RMS number as a rough sanity check only. With the phone moved back, four plays named the right cot four times.

Leave about 10 s between plays. `CRY_HOLD_MS` = 8000 keeps a confirmed cot crying for eight seconds, during which a challenger needs 1.5× the RMS sustained for 1.5 s to take the name — and the wrist buzzes only on a fresh quiet→cry transition. A second play inside the hold can therefore be silent and can keep the first cot's name. **That is the one-cry-one-buzz rule working, not a miss.** Ten seconds makes each play an independent test.

The clip: `session9_clip_0005.wav` from `research/out/demo_kit/` — the most forgiving of the kit; it announces at every volume and never clips. Backups, in order: `session4_clip_0012.wav`, `session1_clip_0005.wav`.

Do not play from the laptop's speakers. They reshape the cry enough that the model refuses it — **P 0.03–0.15 on a window the model scores 0.9961 from the same recording**. The phone is not a nicety; it is the only source shown to work (D-61).

The room: quiet if you can get it. The system holds **0 false alarms in 20.5 minutes** of a room with people talking, but the demonstration is easier in a quiet room.

Per cot — about 15 seconds each

The phone goes 15 cm from the microphone. Not 10 cm (that saturates the microphone) and not 1 m (the cry never crosses the trigger). **15 cm.**

1. Hold the phone 15 cm from the cot's microphone.
2. Press play. **Let the whole 5-second clip run out.** Do not press twice; do not stop it early.
3. **Wait 5 seconds.**
4. **Read the hub's log:**

``` `node 1: front end 48.2 ms, inference 103.5 ms, P(no_cry)=0.0039 P(cry)=0.9961 node 1: ALERT — cry` ```

That second line, on the node you played at, is the pass condition.

5. **Wait 10 more seconds.** The alert is held 8 s and then clears. You must let it clear before the next cot, or the hub keeps naming the previous cot.

Repeat for each cot. Afterwards, wait 10 seconds of quiet and check: **one `ALERT — cry` line per cot, and nothing after the last one.** A quiet window must never announce.

Total: about 45 seconds of playing, 75 seconds with the waits.

Why one play, and not several

The node freezes **one** window per cry (cry mode, D-69). Once it has, it will not freeze again until the room goes quiet. A second play of the same clip therefore lands on the tail of the clip — a pause between cries — and the model scores that around **0.11**. That "no cry" is the *clip* being wrong, not the system being wrong.

> Playing once and reading the log tells you what happened. Playing three times and watching the wrist tells you almost nothing.
```

Reading the log

The log says	What it means	What to do
`node N: ALERT — cry`	**pass** — the right cot named	next cot
`the window is CLIPPED`	the phone is too close; the microphone saturated	move it **back** to 15 cm; do **not** turn the volume down
`no cry` with P(cry) high	the window caught a pause in the clip	play once more, same place
`cry (too quiet to announce)`	below the 0.020 RMS floor	phone closer, or volume up
`cry (unconfirmed)`	a maybe: P(cry) high but the window was not clean	play once more
`ALERT — cry` on the **wrong** cot	another cot heard it cleanly	wait out the 8 s hold, then play again
nothing at all	the cry never crossed the trigger	phone closer

The "clipped → move back, not down" row is counter-intuitive and correct: turning the volume down changes the *recording's* dynamics as well as its level, and the kit was characterised at a fixed volume. Distance changes only the level.

The two-cot checks that are specific to this system

Before any acoustic test, four things are worth confirming, because each has a failure mode that looks like an acoustic problem and is not one.

1. The bench map. Four targets, four ports, and the wrong port means the wrong firmware. Read each board's own boot banner rather than trusting a remembered port number:

```
python firmware/tools/who_is_who.py
```

The map on the last verified session was COM32 hub, COM30 wearable, COM31 cot 1, COM37 cot 2.

2. The ids. `id 1` and `id 2` over each cot's console, then confirm they survived the flash. A cot on any other number is not quiet — it is *invisible*, because the hub drops its frames before they reach a slot. The symptom of a wrong id is identical to the symptom of a dead radio, which is why this check comes second and not last.

3. Both link bars, in both directions. On the cot's console, `status` prints its view of the hub (`link [####] -12 dBm (hub frame 445 ms ago)`). On the hub's console, `status` prints its view of each cot. If the cot's bar is empty, the hub's heartbeat is not arriving — check that the hub is running the firmware that sends one, because a hub built before that change answers requests perfectly and still leaves the cot's bar dark.

4. The wrist's cot count. The wearable must read `state: 2 cot(s)`. If it reads `0 cot(s)` while the hub can hear both, the hub is running firmware that only publishes state on a cry transition — the bug the 2026-11-05 session found. It is a publishing-side fault, not a wearable fault.

Then the alert path, without any sound at all:

```
alert 2      → hub: ble_alert: a cry on cot 2 at 100%
             → wrist: [ble] ALERT cot 2 type 1 conf 100% event #1   (and a buzz)
clear        → wrist: the matching type 2 cancel
```

That exercises the whole downstream half — arbitration, BLE, the wrist's parsing, the haptic — with none of the acoustic variables in play. If it fails, the problem is not the microphone.

Troubleshooting, in the order to try

1. **Move the phone back to 15 cm.** Clipping is the commonest failure and the fix is distance.
2. **Play once more**, same place, and read the log.
3. **Check the hub is alive** — the console answers `status`:

``` I (12345) console: << status I (12346) console: cot 1: rms 0.0154 level 0.0092 window 0.0121 (whole) confidence 0% link -23 dBm I (12347) console: cot 2: rms 0.0148 level 0.0088 window 0.0119 (whole) confidence 0% link -43 dBm ```

No answer means the hub has crashed or lost power.

4. **Check the node is alive** — its console answers `status` and `stats`.
5. **The link**, if a window never arrives: the hub logs `the burst stalled with N of 286 fragments` and asks for the missing ones. The resend rounds recover nearly all of it; if a burst dies outright, move the cot away from whatever is on the same channel.

The demo kit and how it was built

`research/build_demo_kit.py` cut the recorded sessions into 5-second clips; `research/screen_clips.py` and `research/score_demo_clips.py` scored every clip against the deployed model and the announcement rule.

Fact	Number	Source
Clips that announce under the rule on the boards	**27 of 37**	`research/out/demo_rehearsal.md`
Clips that clip if turned up	**17**	same
Freeze lands after the clip starts	2.0–5.4 s, **median 2.9 s**	`research/out/demo_rehearsal.csv`
Continuous cry the model needs in the window	**~1.5–1.75 s**	`research/out/DEMO_KIT.md`
Ambient room with people talking	**20.5 min, 0 false alarms**	`research/out/ambient_25min/summary.md`
Both cots on one hub, continuous	**25 min, no brownout or reset**	`SYSTEM_UPDATE.md`

The kit is not a lucky selection: it is every clip from the recorded sessions, scored, with the failures listed alongside the successes. That is what makes "27 of 37" a defensible number rather than a demonstration chosen to work.
```

Other verification procedures

The confusion session (O-9)

`research/CONFUSION_SESSION.md` gives the full procedure: nine one-minute takes with `node_capture` on the real INMP441, cut into 5 s clips with a 2.5 s stride, labelled with the session's ground truth via `label_session.py --truth` — **not** by loudness. Labelling by loudness would have made the evaluation circular. Results in Chapter 15.

Burst integrity

`firmware/tools/measure_burst.py` and `verify_burst.py` capture a burst and check every fragment's CRC-16 and the whole window's CRC-32 against the node's advertised value, using `zlib.crc32` on the host. `dump_node_window.py` writes the node's frozen window out over its USB console so the exact bytes the node holds can be compared with what the hub reassembled.

Latency

`firmware/tools/measure_latency.py` timestamps request → verdict across the two logs.

Artefact drift

`firmware/tools/check_artifact_drift.py` verifies `research/out/`, `deploy_config.json` and the staged firmware copies all describe the same model bytes. Currently **PASS**.

The dashboard contract

`python firmware/tools/test_dashboard_contract.py` starts a throwaway dashboard, builds the hub's upload and presence bodies *byte for byte* from the same fields, posts them, and checks what comes back out of the API. It covers the batch idempotence, the untimed-record rule, the presence round trip and the staleness rule (D-108). It needs no hardware and is part of the pre-merge battery; Chapter 12 explains why this one test exists.

The cry log's ordering rule

Two proofs, deliberately: `python firmware/tools/test_log_order.py` models the uploader's lowest-unsent / clamped-watermark algorithm on the host (six tests including a ten-thousand-case soak, plus a companion test that shows the *old* algorithm losing records); and `logtest` on the hub's console runs the same scenario against the real component on the real chip, on its own scratch file, touching nothing real.

Presence, on the bench

With the hub on a network, the console's `net` word reports batches, failures and the last error; the dashboard's own page answers who is online within about two minutes of the hub being switched off (30 s report, 90 s staleness, 10 s poll — D-105). The panel's wireless mark is drawn only while the hub is actually on the network (D-103). A clean-power soak — ten minutes, `status` and `net` at the ends — is the check that the connectivity stage added no restarts (D-106/D-107).

The wearable, without a noise

`alert <cot>` on the hub console injects a real alert through the real publish path. This is how the BLE server, the notification and the wearable's callback are tested independently of the model — and how the haptic fault was localised to hardware.

Chapter 17 — Toolchain: what was installed, what broke, and why it is pinned

The starting point

The machine had **no ESP-IDF, no PlatformIO, no Arduino IDE and no Xtensa compiler**. Everything the build needs is now reproducible **from the repository** rather than from the state of one machine (D-48). That is the standard the toolchain work was held to, and it is why `dependencies.lock`, `fetch_deps.sh` and `patch_deps.py` are tracked.

The versions that are known to work

Layer	Version
Python	**3.11.9**
TensorFlow (CPU)	**2.21.0**
librosa	**0.11.0**
soundfile	**0.14.0**
numpy	**2.4.6**
scipy	**1.17.1**
ziglang (used as the host C compiler for the parity test)	**21.1.0**
PlatformIO	**6.2.0**
platform-espressif32	**8.0.0**
ESP-IDF	**6.1.0** (bundled as `framework-espidf 4.60100.0`)
Arduino ESP32 core	**3.3.12**
Node/wearable FQBN	`esp32:esp32:weact_studio_esp32c3`

D-45 — why ESP-IDF is installed by hand

PlatformIO 6.2 installs a **URL-specified** `platform_packages` entry by treating the archive as an **npm package**. It picks the wrong `package.json` out of the tree, writes a bogus version into `.piopm`, and leaves `packages/framework-espidf` holding only a **test-fixture package**. CMake then fails with:

```
include could not find requested file: .../tools/cmake/project.cmake
```

That is I-6, and it cost real time. IDF 5.3.2 could not be installed through PlatformIO at all; 6.1.0 was already on disk and satisfies the TFLM port's `idf >= 5.1`.

The fix is to install the framework manually and pin it with `packages/framework-espidf/.piopm`, and never to put a `platform_packages` URL in `platformio.ini`. The ini carries a comment saying exactly that, so

the next person does not repeat it.

D-46 — why TFLM is vendored inside the project

PlatformIO re-compiles IDF components with SCons using the **source basename** for object names. The TFLM port has **twelve basename collisions**, so the build died with:

```
Multiple ways to build the same target
```

Moving the vendored components **inside** `firmware/hub_infer/components/` makes the paths relative and distinct. `esp-nn` is wired as the port's own dependency with the registry copy removed, and `board_build.esp-idf.overwrite_managed_components = no` stops PlatformIO wiping `managed_components` on every configure.

`firmware/tools/patch_deps.py` keeps the build **offline and pinned**; `fetch_deps.sh` re-fetches exactly what was fetched.

This is the general lesson: a build that downloads a moving dependency is not a build that can be reproduced in a year, and vendoring is not untidiness — it is the reproducibility mechanism.

D-47 — DIO flash mode

The effective configuration is `CONFIG_ESPTOOLPY_FLASHMODE="dio"`, and both `bootloader.bin` and `firmware.bin` carry flash-mode byte `0x02`.

A combined image built with QIO makes the chip **watchdog-reset in a loop** — `rst:0x7 TG0WDT_SYS_RST` repeating, with no application output at all. The ROM cannot read this board's flash in QIO before the bootloader has configured its own timing. **The bootloader switches the chip to QIO by itself once it is running**, so DIO costs nothing at this workload.

`flash_hub.py --merged` passes `--flash-mode dio` explicitly when combining the three images at `0x0` (bootloader), `0x8000` (partition table) and `0x10000` (application).

Note the trap: `sdkconfig.esp32s3` still shows `CONFIG_ESPTOOLPY_FLASHMODE_QIO=y` as the *choice symbol*, while the *string* that esptool acts on says `dio`. Reading only the choice symbol gives the wrong answer.

D-48 — the Arduino side

The node and wearable sketches are compiled with the **Arduino IDE's own** `arduino-cli` against core **3.3.12**, FQBN `esp32:esp32:weact_studio_esp32c3` — the WeAct Studio ESP32-C3 definition, which selects the **chip** and its flash/USB settings rather than one physical module, which is the exact module on the bench. **USB CDC on boot** must be enabled, or the serial console does not appear.

`firmware/tools/install_sketches.py` places the sketches where the CLI expects them; `flash_node.py` is the one-command flasher.

The fourth toolchain: none at all

The three units above need a compiler. The **offline path** needs none: `firmware/prebuilt/` holds the built images, and `flash_unit.py` writes them with `esptool` alone. That is a deliberate fourth option, not a

convenience — it exists so a board can be brought up on a machine that has never seen this project, with no Arduino core, no PlatformIO, no ESP-IDF and no network.

What stops it going stale is `BUILD_INFO.json`, which records the SHA-256 of every source file that went into each image, and `check_prebuilt.py`, which recomputes them. Chapter 18 has the details; Chapter 20 has the commands.

Two firmwares, two toolchains — ESP-IDF for the hub (because NimBLE, coexistence and PSRAM need `sdkconfig`) and Arduino for the two C3 units (because they are simple and the core is already proven). That split is deliberate, not accidental.

D-49 — every serial tool resynchronises on its own magic

A USB CDC port can enumerate **after** the sketch has already printed its banner. And a single dropped byte on the USB link would otherwise **shift every later sample and quietly ruin the dataset**.

So every host tool that reads a serial stream:

- **resynchronises on its own magic** rather than waiting for a fixed block;
- treats capture records as CRC-16-protected, and **drops any record that fails**;
- is written as a **testable parser class**, checked against synthetic streams.

That last point is the one that makes it trustworthy: a stream parser tested only against a live device is a parser tested only on the happy path.

D-50 — the notebook is a build artefact

`notebooks/crynet_training.ipynb` is **generated** from `research/build_notebook.py` and **never hand-edited**. Editing it directly makes it impossible to regenerate and to diff. It is validated with `nbformat.validate()` after every regeneration.

```
cd research && python build_notebook.py
```

The generator is 2,293 lines and emits the notebook cell by cell, with the reasoning as markdown cells inside the notebook itself — so a person running it on Kaggle reads the *why* next to the code, not in a separate document.

The build commands

```
# hub — about a minute incremental
cd firmware/hub_infer && python -m platformio run

# node
arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3 firmware/node/node_crynet

# wearable
arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3 firmware/wearable/wearable_crynet

# re-fetch the pinned dependencies
bash firmware/tools/fetch_deps.sh
```

Where the evidence lives

Every run's log is under `research/toolchain/` and every run's conclusion is in `SYSTEM_UPDATE.md`**'s change log**. That pairing — raw log plus a written conclusion — is what makes the project's claims checkable rather than remembered.

Chapter 18 — The host tools: twenty-nine programs that made the firmware debuggable

Why this chapter exists

None of these tools ships in the product. All of them are why the product works. A microcontroller system with no host-side instrumentation is a system whose faults can only be guessed at, and almost every diagnosis in this dossier was made by a tool in this list.

They live in `firmware/tools/` (**29 Python programs**, plus `fetch_deps.sh`, the PowerShell helpers and `flash.bat`) and `research/` (the evaluation and analysis scripts). Every one is reproduced in full in Appendix B.

Flashing and installation

Tool	What it does
`flash_hub.py`	Flashes the ESP32-S3 and prints its boot log. `--merged` combines bootloader (0x0), partition table (0x8000) and application (0x10000) into one file **with** `--flash-mode dio`, which is what a graphical flasher expects. `--flash-only`, `--log-only --seconds N`, `--port COM32`. Everything it needs is already on the machine — nothing to download.
`flash_node.py`	Builds the node sketch and flashes it over USB with `arduino-cli`. `--sketch` also selects `wearable` or `haptic_test`, so the motor's bench proof is one command.
`flash_unit.py`	**The offline path.** Flashes any of the three roles from `firmware/prebuilt/` using `esptool` alone — no Arduino core, no PlatformIO, no compiler, no network. Writes region by region and **never writes NVS**, so a cot keeps its id. `--check` verifies every image against `SHA256SUMS`; `--list`, `--backup`, `--log N`.
`make_prebuilt.py`	Compiles all three units from source, copies the images into `firmware/prebuilt/`, rewrites `SHA256SUMS` and writes `BUILD_INFO.json` — the SHA-256 of **every source file** that went into each image.
`check_prebuilt.py`	Recomputes those source fingerprints and compares. **This is the check that refuses to let a stale image be flashed**: an edited sketch with old `.bin` files is caught before anyone writes it to a board.
`prebuilt_lib.py`	The shared fingerprinting and manifest logic behind the two above, so the builder and the checker cannot disagree about what counts as a source file.
`serial_monitor.py`	A plain serial reader for when no parser is wanted — the raw log, nothing interpreted.
`install_sketches.py`	Copies the sketches into the Arduino sketchbook, because the IDE builds a sketch from its own folder while the sources live in the repo.
`who_is_who.py`	Says which node is on which port. Node id lives in **NVS**, so it survives a reflash — but only if the flash tool does not erase that partition. This script resolves the ambiguity instead of leaving it to guesswork.
`patch_deps.py`	Makes the vendored TFLite Micro port build against the vendored `esp-nn`, overriding the port's `idf_component.yml` declaration. Keeps the build offline and pinned.
`fetch_deps.sh`	Re-fetches exactly the pinned dependencies.

Listening and capture

Tool	What it does
`listen_hub.py`	Reads the hub's log and prints the chain. **It plays nothing** — it is a passive observer, so a sound can be made by any means and the hub's reaction recorded.
`listen_node.py`	Reads the node's log: the noise floor, the trigger level, and *why* it froze each window. Since D-63 the node owns the trigger, so this log is the only place that question can be answered.
`capture_serial.py`	Reads the hub's serial log, optionally resetting the board first.
`capture_multi.py`	Watches **several boards at once** with timestamps. The bench has the hub, both cots and the wearable on one machine, so the questions that matter need every board's log aligned in time. This is what made the two-board latency measurements possible — and what settled the resend-bitmap question by putting both ends of one exchange in one file.
`capture_hub_wait.py`	Waits for a specific event and writes `research/out/hub_node1_fullchain.log` with `[t]` timestamped lines. Exit 0 on capture, 1 on timeout — so it can be used in a script.
`node_console.py`	Sends one-line commands to the node's console at 115200 and prints its state.

Every one of these implements D-49: **resynchronise on the stream's own magic, never wait for a fixed block, and drop any record whose CRC-16 fails.** A USB CDC port can enumerate after the sketch has already printed its banner, and one dropped byte would otherwise shift every later sample.

Audio collection and labelling

Tool	What it does
`collect_clips.py`	Records in-domain audio from the node over USB and cuts it into fixed-length clips. The deployed model was trained on corpora this room's microphone has never produced; this is how that gap is closed.
`label_session.py`	Turns a captured session into **labelled** 5-second clips. A fixed cut is not a label — `--truth` applies the session's ground truth, so a loud non-cry take is labelled non-cry rather than being labelled by its loudness. Without this the evaluation would be circular.
`dump_node_window.py`	Dumps the node's frozen window over its serial console using the node's `burst` command, which writes it in exactly the radio's framing. This is how the *exact bytes the node holds* are compared with what the hub reassembled.

Measurement and verification

Tool	What it does
`measure_burst.py`	Measures the hub's burst time on the real boards. This is the tool that produced the 5.54–6.93 s → 2.06–2.11 s comparison.
`measure_latency.py`	Measures end-to-end latency and whether the hub calls the cry, from the log.
`verify_burst.py`	Checks the node's audio path end to end: dumps the frozen window and verifies every fragment CRC-16 and the whole-window CRC-32, using `zlib.crc32` on the host. Can write the window as a WAV.
`check_artifact_drift.py`	Verifies that `research/out/`, `deploy_config.json` and the staged firmware copies all describe the same model bytes. Run after dropping new Kaggle outputs in and **before** building. Currently **PASS**.
`gen_probe_refs.py`	Regenerates `crynet_probe_refs.h` by running the **notebook's own** log-Mel front end, extracted from the notebook. The on-device probes therefore compare against the training pipeline itself, not against a reimplementation of it.

The record and the dashboard

Tool	What it does
`hub_console.py`	A serial console session for the hub's own word list (`status`, `history`, `log`, `net`, `logtest` ...) — the network and record configuration of Chapter 12, driven from a PC instead of a bare UART.
`test_log_order.py`	The host model of the uploader's ordering rule: the lowest unsent numbers, the clamped watermark, and a companion test that shows the *old* algorithm losing records so the guard cannot silently stop guarding.
`test_dashboard_contract.py`	Builds the hub's upload and presence bodies **byte for byte** and proves a running dashboard round-trips them — idempotence, the untimed-record rule, presence freshness. The one contract in the system no compiler checks (Chapter 12).

BLE

Tool	What it does
`ble_alert_test.py`	A PC BLE client for the hub's GATT server. Subscribes to both characteristics and prints every packet — this produced the `ALERT {'cot': 1, ...}` lines quoted throughout the dossier.
`serve_alert_page.py`	Serves the alert page over **HTTPS**, because Web Bluetooth only works in a secure context and on a phone that means HTTPS. Lets a phone be the caregiver's device without the wearable.

The research scripts

Script	What it does
`build_notebook.py`	Generates the training notebook (2,293 lines). The notebook is never hand-edited (D-50).
`make_stub.py` / `run_stub.sh`	A synthetic corpus and a FAST-mode run: validates the whole pipeline end to end in minutes without downloading 4.6 GB. This is how the music corpus was validated before the Kaggle run.
`run_real_fast.sh`	The real corpus, subsampled, on CPU.
`test_c_parity.py`	Compiles the C front end on the host (with `ziglang` as the compiler) and compares it with Python on the same PCM. A divergence can be debugged on a workstation rather than on a board.
`verify_int8_parity.py`	INT8 against float32 on the host.
`eval_indomain.py`	The deployed model on this room's audio.
`eval_esc50.py`	ESC-50: the 40 `crying_baby` clips and the 1,439 non-cry clips.
`eval_music.py`	GTZAN, ten genres — the script that found the 8.19 % music failure.
`eval_speech.py`	Speech negatives.
`adaptive_trigger_study.py`	Sweeps trigger multiples over the recorded sessions. **This is what chose 3.0**, not taste.
`analyze_gates.py`	What each gate in the announcement rule rejects, and what it costs.
`simulate_freeze.py`	Replays the freeze rule against the corpus: the stop-capped rule catches as many episodes as the peak-centred rule.
`analyze_window.py`	Says what is in a window and what the model makes of it.
`screen_clips.py` / `score_demo_clips.py` / `build_demo_kit.py`	Build and score the demonstration kit — every clip scored, failures listed alongside successes.
`check_corners.py`	Host-side graphics regression: the panel's four rounded corners must be exact mirror images.
`test_heartbeat_framing.py`	Compiles the real `crynet_proto.c` with `ziglang` and holds the two-cot protocol rules: a heartbeat is exactly **12 bytes** and round-trips as type 4, the other frame types are unchanged, and the id rule hears 1 and 2 while ignoring 0, 3 and 255. **Checked against the bug it was written for** — reverting the framing fix makes it fail.
`ui_mock.py`	Renders the panel to a PNG on the host, so a layout can be judged before it is flashed. The two-card redesign was rendered this way first (`research/out/two_node_display_mock.png`).
`simulate_demo.py`	Replays the demo kit against the announcement rule without any hardware.
`build_font.py`	Generates the panel's font table (and once had the 32-character offset bug).
`rssi_walk_test.py`, `analyze_starvation.py`, `cry_audio_check.py`,	The specialist instruments: radio range walks, queue starvation analysis, corpus audio sanity (this is the class of check that found the contaminated "silence" class), and panel photograph measurement.

Script	What it does
`measure_photo.py`, `compare_photo.py`	

The offline images, and the check that guards them

`firmware/prebuilt/` holds the built firmware for all three roles, so a board can be brought up on a laptop that has never seen this project. It needs exactly one executable: `esptool`.

Folder	Chip	Files and offsets
`hub/`	ESP32-S3	`bootloader.bin` 0x0, `partitions.bin` 0x8000, `firmware.bin` 0x10000
`node/`	ESP32-C3	`bootloader.bin` 0x0, `partitions.bin` 0x8000, `boot_app0.bin` 0xe000, `app.bin` 0x10000
`wearable/`	ESP32-C3	the same four

Two rules are load-bearing, and both are easy to get wrong:

1. **Write the regions, not one merged file.** A merged image fills the gaps between regions with `0xFF`, and writing that would **erase NVS** — which is where a cot keeps its id. The offsets above skip NVS on purpose (0x9000–0xE000 on the C3 boards, 0x9000–0xF000 on the hub). A merged flash would silently turn a working cot into an invisible one, because its id would be gone.
2. **The hub images are DIO.** A QIO image makes the S3 watchdog-reset in a loop right after power-on (D-47). The bootloader header in `hub/bootloader.bin` says DIO.

`BUILD_INFO.json` is the interesting part. It records, per role, the SHA-256 of **every source file** that went into that image — 52 files for the hub, one for each sketch — with an explicit rule set for what does *not* count (build directories, `sdkconfig*`, the vendored third-party trees, editor artefacts). `check_prebuilt.py` recomputes those fingerprints and compares.

Verified at the time of writing:

```
== the offline images vs the source ==
  images: 11 files, all match SHA256SUMS
  hub       source fingerprint 7ebe61d46bec5b4b... over 52 files - up to date
  node      source fingerprint a715bd57edc46a8a... over 1 files - up to date
  wearable  source fingerprint 7071ab878821afe3... over 1 files - up to date

OK  the offline images are up to date with the source
```

`flash.bat` runs the same check in PowerShell — so it needs no Python — before it writes anything, and refuses a stale or damaged image. A GitHub workflow (`.github/workflows/prebuilt-check.yml`) rebuilds all three on every push and fails if the result differs from what is committed.

The reason this is worth the machinery: the failure it prevents is *silent*. A stale image flashes successfully, boots, and behaves like the old firmware, and the person holding the board has no way to tell. A fingerprint check turns that into a refusal with a message.

The pattern worth defending

Three properties recur across all of these, and they are the reason the project's numbers can be trusted:

1. **Passive observation before intervention.** `listen_hub.py` plays nothing. The measurement is separated from the stimulus, so a result cannot be an artefact of the tool.
2. **Host-side reimplementation of the check, not of the thing being checked.** `verify_burst.py` uses `zlib.crc32`; `gen_probe_refs.py` uses the notebook's own front end. The reference is independent of the implementation under test.
3. **Every parser is a testable class checked against synthetic streams.** A parser tested only against a live device is tested only on the happy path.

Chapter 19 — Limitations, open items, and what to do next

Why this chapter exists

A dossier that lists only what works is a brochure. The value of this document is that it can be defended, and a defence is only credible if it knows where the ground stops. This chapter is the boundary of the evidence: what has **not** been measured, what is **open**, what the project **knowingly gave up**, and what the next person should do about each.

Three kinds of thing appear here, and they are different:

1. **Open items (O-n).** Work that was identified and deferred. Each has a reason it is still open.
2. **Unmeasured things.** Work that was never attempted, or attempted and abandoned without a result. These are the honest holes.
3. **Knowingly-given-up things.** Things the project decided *not* to do, and accepted the cost of. These are not open — they are closed decisions with a price attached.

The record's O-list was written before the system existed in its present form, so it is out of date in places. Where that has happened, both the original wording and the current status are given. The two-cot work also added five sub-items of its own - **O-1a … O-1e** - which are part of the repository's record and are covered here alongside the original nine.

---

1. The open items, and where each one stands today

O-1 — "the wearable does not exist yet" — CLOSED

Original wording: "The wearable (Stage 2d) does not exist yet."

Status: closed, and recorded as O-1a. `firmware/wearable/wearable_crynet/` was written, flashed and verified. It is a BLE client against the hub's GATT server with an SSD1306 OLED (SDA 5, SCL 6) and a BC547 low-side haptic drive on GPIO7 (D-77). On **2026-11-06** three alert cycles in a row came through complete, the haptic fired, and the unit recovered from a hub reset mid-link. Chapter 8 is the full description.

What O-1 gated was the wearable as a bench artefact, and that is done. What it did **not** gate is battery *runtime*: the unit has run on its pouch, but nobody has timed it (O-1d). The alert face now names **every** crying cot (D-86, which closed O-1b); the alert packet and the buzz still name one.

O-2 — the model has never heard a real infant cry — STILL OPEN, and it is the largest one

Original wording: "The only cry source available in this room is a *recording played through a phone speaker* — compressed, band-limited and EQ'd by the speaker."

Status: still open. This is the single most important caveat in the project.

Everything the model has been measured against is one of:

Positive class	What it physically is
ICD (13,330 clips)	16 kHz recordings of infants, from three source corpora, at their original levels
cry4 (4,871 clips)	16 kHz, LUFS-normalised, VAD-trimmed real infant cries
ESC-50 `crying_baby` (40 clips)	16 kHz, held out entirely as a cross-domain test
The room sessions (session 9, 10)	**a phone loudspeaker playing one of the above**, at 15 cm and at 1 m

The **negatives** are authentic: this room, this INMP441, this microphone, this gain, this noise floor. The **positives** in the *room* are a proxy. A phone speaker in a plastic case applies a band-limited, compressed, EQ'd response, and it removes exactly the low-frequency detail that distinguishes a cry from other vocal sounds. A model trained on clean 16 kHz corpus audio and tested on speaker-played audio is being tested on a corrupted version of its training distribution; the fact that it still scores 0.9961 on that corruption says the signal it keys on survives the corruption, but it does not say what happens on a real infant in a real cot.

What would close it. Fifty to a hundred real clips: ten infants × several cries, recorded with `node_capture` on the actual INMP441 in the actual room, plus room negatives in the same takes. The tooling already exists — `firmware/tools/collect_clips.py` records, `firmware/tools/label_session.py --truth` labels by ground truth rather than by loudness, and `research/CONFUSION_SESSION.md` documents the procedure. Those clips then go into the notebook as a custom corpus, and the whole notebook is re-run.

Why it has not been done. It needs a baby and a quiet room, and it needs consent. It is not a software problem.

O-3 — one photograph of the panel with the room lit — STILL OPEN, and it is now cosmetic

Original wording: "What is still wanted is one photograph **with the room lit**, so the ramp yields the panel's own tone curve instead of a colour choice."

Status: still open, but the consequence has shrunk. The panel's palette was set from the colour-check page judged by eye (D-55) and then from a dark-ramp photograph the user read (D-57, D-59). The measurement shows this panel *lifts the dark end and lifts the blue channel hardest*, and the palette compensates for that deliberately (Chapter 14, Round 5).

What a lit-room photograph would add is the panel's tone curve — whether the shipped palette is correct in a lit room, not only on a dark one. It is a one-minute job with a camera and it is the last thing that would settle the display's colours. It does not affect detection, alerting or any number in this dossier.

O-4 — the tensor arena is still in PSRAM — STILL OPEN, and deliberately so

Original wording: "Moving it into internal SRAM is the obvious next optimisation. At 101.8 ms of inference per two-second window it is not urgent for a triggered monitor."

Status: still open. The arena is 262,144 bytes reserved, 108,328 used, and it lives in PSRAM. The front end's scratch buffers were moved *out* of PSRAM into internal SRAM for exactly this reason — it took the front end from 135 ms to 41.7 ms — so the same treatment has not yet been applied to the arena.

The reason it was left alone is the memory arithmetic. Hub RAM after the PSRAM move is **12.9 % (42,236 B)** of 512 KB, and the free internal heap with Bluetooth up is **149,271 B**. A 262 KB arena does not fit in 149 KB. Options are (a) shrink the arena to what is actually used — 108 KB fits, but the allocator's peak is not the reserved size and the difference is not free), (b) use `AllocateTensors()` failure as a fallback to PSRAM, or (c)

leave it. At 101.8 ms for a window that is only analysed after an event has already been triggered, (c) is defensible. It is listed here so that nobody later mistakes it for an oversight.

O-5 — the in-domain audio is speaker-played, not real — STILL OPEN, and it is O-2 from the other side

Original wording: "A real cry in this room, recorded with `node_capture`, is the honest test of whether the system works in the real room."

Status: still open. This is the same hole as O-2 seen from the measurement side rather than the training side. O-2 is "the model has never trained on a real cry"; O-5 is "the system has never been evaluated on a real cry". They close together and with the same 50–100 clips.

O-6 — the distance curve — STILL OPEN

Original wording: "The phase cues did not line up in the earlier attempt, so the far-side measurement has to be repeated."

Status: still open. The system has two known points on the distance curve and one hard limit between them:

Distance	What happens	Evidence
15 cm (the realistic case — the mic is *in* the cot)	trigger fires, model scores 0.9961, alert raised	D-60, D-61, live sessions
1 m	**0 of 590 records cross the trigger** — the model is never asked	D-60, session 10

The 1 m result is not a model failure: asked anyway, the model would have caught 18 of 44 windows. It is a *reach* limit set by `CRYNET_TRIGGER_RMS` (0.030) and the INMP441's fixed −26 dBFS ±3 dB sensitivity with no gain strap. The node's adaptive trigger (D-63, 3× the running floor) partly addresses it, and on the boards afterwards the hub asked 5–11 times a minute instead of 1–2 — but a *continuous* distance curve has never been measured. The earlier attempt failed because the phase cues did not line up between the phone and the node, so the two clocks could not be aligned; the fix is to trigger on an acoustic event visible in both streams (the phone's own playback start) rather than on a phase relationship.

O-7 — the resend mechanism has not been exercised under a genuinely hostile channel — STILL OPEN

Original wording: "It is verified against the hub's own Bluetooth advertising, which is the real interference in this design, but not against a busy Wi-Fi environment with a second router on the same channel."

Status: still open. The resend path is real and exercised: the hub logs "the burst stalled with N of 286 fragments", asks for the missing ones, and the resend rounds recover nearly all of it (D-26, D-27). The hub's own Bluetooth advertising is a genuine and *severe* interferer — it is the reason the advertising interval is throttled to 320 ms during a burst (Chapter 11) and the reason the burst duration dropped from 5.54–6.93 s to 2.06–2.11 s (Chapter 9).

What is untested is a **second Wi-Fi router on channel 6** — a hospital or a school where the nursery's AP is on the same channel as the ESP-NOW network. ESP-NOW and Wi-Fi share the radio, so a busy channel does not merely drop frames, it prevents the node from transmitting at all, and the resend rounds then fight a channel that never opens. The prediction to test: the resend rounds recover most of it (they recovered against

the BLE interferer), but the *burst* can exceed `BURST_TIMEOUT_MS` (15 s) and be abandoned, with `BURST_FAIL_LIMIT` (3) consecutive failures putting the node into a 60-second backoff.

O-8 — the music false-alarm rate — CLOSED

Original wording: "The deployed model false-alarms on 8.19 % of loud music, and the fix is a training round that has not been run yet."

Status: closed by D-58. GTZAN was added to the training corpus, the notebook was patched and regenerated, the second full Kaggle run happened, and the retrained model is deployed and verified on the board.

	Before	After
GTZAN music false alarms	79 / 965 = **8.19 %**	2 / 965 = **0.21 %**
ESC-50 non-cry false alarms	38 / 1,439 = **2.64 %**	17 / 1,439 = **1.18 %**
ESC-50 `crying_baby` recall	36 / 40	**36 / 40** (unchanged)
Quiet room	0 / 236	**0 / 236** (unchanged)

The cost is real and was accepted: per-window INT8 recall on the phone-speaker sessions fell (session 1: 0.867 → 0.800; session 4: 0.724 → 0.592). Episode recall did not fall, because a cry is sustained and trips many consecutive windows.

O-9 — the confusion session — CLOSED

Original wording: "The confusion session — the in-room measurement of speech, music, television and the phone-played cry — has not been recorded."

Status: closed by D-60. Nine one-minute takes on the real INMP441, labelled by the session's ground truth rather than by loudness, cut into 5 s clips with a 2.5 s stride. Full table in Chapter 15.

Result	Number
False alarms on speech, music and film	**0 in 308 windows**
Loudest music the room can produce	trigger fired on 84 % of records, **0 of 44** windows alarmed
Loudest film	trigger fired on 46 % of records, **0 of 44** windows alarmed
Per-window recall, cry at 15 cm	**86.4 %** (38/44)

The session also produced the finding that redirected the engineering: **the cry at one metre never crosses the trigger**, so the system's reach is set by software, not by the microphone.

---

2. The two-cot sub-items, and the items with no O-number at all

O-1a — the wearable exists and has run on its cell; the runtime does not — PARTLY CLOSED

The wearable is on the bench, proven end to end, and **has run from its Li-Po pouch** (2026-11-20). What remains under this item is *how long* it lasts, and the buzz that went missing while it was on the cell — both O-1d

below. The battery/SOC readout that used to be listed here is **cancelled**, not deferred: there is no divider and no SOC text on the OLED (D-83).

O-1b — two cots crying at the same instant — CLOSED BY D-86

This was open when the dossier was first written, and it is now measured. On **2026-11-20** one clip was heard by both node microphones and the hub's crying mask sat at `0x03` **for fifteen consecutive seconds**; the owner watched the wrist's alert face read the two-cot label. **Three separate two-cot events** were captured (15 s, 3 s, 7 s).

The one-buzz rule held across all of them. Over the whole session, nine cry announcements across five episodes produced **exactly five buzzes**; the four silent ones were handovers between cots. The single-cot path is unchanged.

What the test did not show, and this matters for the defence: there is **no second buzz when a second cot joins**. That was never implemented, and D-86 deliberately does not add it — one cry is one buzz. The earlier note on this item described a branch that would buzz on a second cot; that branch (`feature/wearable-multi-cot`) turned out to be an **earlier and different** attempt with no `COTS` label at all, and it is superseded, not merged.

What remains genuinely untested: two *independent* cries — two sources, two babies — rather than one clip heard by two microphones. The mask, the arbitration and the buzz rule were exercised by the shared-source case; genuinely colliding bursts from two separate cries have still not been staged.

O-1f — this repository's D-86 image is compile-verified, not flashed — OPEN

The behaviour above was proved from the **sibling repository's** build. The change has been ported here and compiles to the identical figures — **685,589 B (52 % of flash) / 18,684 B (5 % of RAM)** — and `check_prebuilt.py` passes. But **no board was attached when it was ported**, so this repo's `firmware/prebuilt/wearable/app.bin` has never been written to a device.

How it closes: `python firmware/tools/flash_unit.py --role wearable --port COM30`, then re-run the two-cot play and watch the glass. Until then, treat the port as verified-by-equivalence, not verified-on-hardware.

O-1c — the I²S zero watchdog is committed but not flashed — OPEN

The watchdog exists because a microphone that stops producing data produces **zeros**, and zeros look exactly like a very quiet room: no trigger, no burst, no alert. The system fails silent, which for a monitor is the worst way to fail. Chapter 7 has the code.

Two honest limits: it detects **exact** zeros only, so a microphone stuck at a DC value or at low-level noise is not caught; and the fault-injection test proved the code executes and refills a window, not that it recovers the original physical fault, which has never been reproduced on demand.

It is not flashed because the owner is soldering the final wiring and a node should not be reflashed mid-solder (D-79). **After it goes on it needs a played-cry end-to-end run** to confirm the node still streams and the hub still hears it.

O-1d — no *runtime* has ever been measured; one unit has run on a cell — OPEN, and it is the largest hardware gap

A correction to this dossier's own first edition. It said *"nothing has ever run from a cell."* That was too strong, and the bench record disproves it: during the third round of the **2026-11-20** session the wearable was on its **Li-Po pouch**, not USB — which is exactly why COM30 was absent and no wrist log could be captured that round. **A unit has run from a cell and kept working.** The nodes and the hub have still never been battery-powered.

What is still open is how long anything lasts, and one live fault came out of that round: **the haptic buzz went missing while the wrist was on the pouch.** The firmware is exonerated — D-86 touches no motor line, and the drive is byte-identical to the build proved on 2026-11-11 — so a sagging 3.3 V rail is a real candidate, since the OLED, the C3's radio peaks and the motor share one linear regulator. The decisive test is a meter across the motor during a live alert: ≈3.3 V means no current (open coil or dry joint), ≈2.1 V means current flowing but a weak motor or a sagging rail, ≈0 V means the BC547 is not switching.

Every power *number* the project has is still **arithmetic from the regulator's behaviour**, not a bench reading. The ~6-hour pouch figure and the <1-day 18650 figure are **projections** (D-82, D-84), and this document labels them as such everywhere it quotes them.

Unknowns only a cell can answer: the boost module's quiescent current, its switch-off drain, the cell's voltage curve under a radio-burst load, the actual charge current, and the thermal behaviour of the board's own linear regulator - which dissipates (5.1 − 3.3) V × I ≈ **0.5–0.7 W** at the radio peaks. **The regulator is the weak point in the chain, not the radio.** A module that draws a milliamp in standby would dominate a wearable's budget, and nobody has measured it.

The procedure is in `SYSTEM_WIRING.md` §6: build **one** power assembly, validate it, then buy the rest. The three rules of D-84 are not optional - set the trimpot to 5.1 V before any board is attached, never run the boost and USB into the same board, and use a VBUS-cut cable to log on battery.

O-1e — the divider's source impedance — CLOSED BY D-83

This item was opened because the proposed 220 k / 100 k divider has a source impedance of **68.75 kΩ** (220 kΩ ‖ 100 kΩ), well above the ~10 k an ESP32 ADC wants, and the 100 nF on the sense node was what made the reading accurate - load-bearing, not a nicety. A fallback was worked out (22 kΩ / 10 kΩ keeps the ×3.2 ratio exactly and drops the source impedance to 6.9 kΩ for about 118 µA more drain).

None of it is needed. The whole feature was withdrawn before it was fitted: no divider, no capacitor, no ADC channel. The item is closed by deletion, which is a legitimate way to close an item and worth naming as such.

O-10 — how the model scores a *loud* conversation — OPEN, no O-number in the record

In twenty minutes of people talking, four non-cry windows scored **0.9688-0.9805**. None was loud enough (0.080 RMS) to exercise the loud path. This unknown is the stated reason the loud path requires saturation: `CRYNET_LOUD_ANNOUNCE_RMS` 0.080 **and** `CRYNET_LOUD_ANNOUNCE_P` 0.9900. If a loud conversation can reach 0.99, the loud path announces it.

How it closes: play speech at a level that reaches 0.080 RMS at the microphone without clipping, and read P(cry). One session.

O-11 — the runbook's RMS target is *unsafe* advice, not merely imprecise — OPEN, and now measured

`DEMO_RUNBOOK.md` says to aim the phone at a hub reading of **0.10-0.20 rms**. The quantity that actually decides refusal is the **peak**: the clip gate is `peak >= 31000` (0.946 of full scale), not an RMS bound.

This was upgraded from "imprecise" to "unsafe" by measurement on 2026-11-20. All three windows refused in that session — peaks 32767, 32637 and 32736 — sat **inside** the 0.10–0.20 band the runbook recommends. The 2 s window includes the clip's quiet lead-in and tail, which drags RMS down while the peak stays at the rail: measured *window* crest was **6.1–7.3**, against ~2.9–3.1 for the loudest 100 ms blocks of the same clips. At crest 7 the clip's ceiling is window RMS ≈ **0.135**, not 0.20. An operator following the runbook exactly can therefore saturate the microphone and see cries missed, or named to the wrong cot, with the hub's own number telling them they are in range.

The correct instruction is a peak target (keep the peak under about 0.75 of full scale) **and distance rather than volume** — peak falls roughly as 1/d, so 1.5–2× the phone-to-mic gap puts these windows near 0.5–0.6 with margin. The same file's example `status` output still shows a third cot. Both are documentation defects rather than system defects, and both are offered rather than applied — but this one now has a measured failure behind it, and it is the first documentary item worth fixing.

O-1g — the microphone's close-range blind spot, and a clip gate that discards on one sample — OPEN, design

This is the system limitation the 2026-11-20 session exposed, and it is separate from the runbook wording above.

The blind spot. A source close enough to drive the INMP441 to the rail produces a window the hub refuses outright — correctly, because the microphone is fixed-gain and flat-tops *before* the node's 16-bit window does, so the harmonics the model reads are already gone. **A real infant crying directly at the microphone lands in the same dead zone.** No firmware change fixes this; only placement or a gain change does. It is a genuine boundary of the system and should be stated as one in a defence.

The gate's granularity. `CRYNET_CLIP_PEAK` is a bare peak test over all 32,000 samples, so **one saturated sample discards a whole 2 s window**. In the measured round, seq 82 was refused on a *single* sample at 32042 while its RMS 0.1901 and crest 5.1 looked otherwise usable. A count- or fraction-based gate — refuse on sustained saturation, tolerate an isolated transient — would have passed it and named the correct cot sooner.

Why it is recorded and not implemented: the model's score on lightly-clipped windows has not been measured, so relaxing the gate would be a guess about accuracy. The order is measure first, then branch, bench-verify and reflash the hub.

O-12 — the record's own numbering and identity blocks are inconsistent — OPEN, documentary

Seven separate things, all in the documents rather than the code:

What	Where	The fact
D-51 and D-52 are each used twice	`DECISIONS.md` §1.2 and §1.5	D-51 is both "music joins the corpus" and "the font table is indexed by raw ASCII"; D-52 is both the zip-index fallback and the `PHONE` pill. 87 rows, 85 distinct numbers
D-70 was skipped	`DECISIONS.md`	The cry-mode entry threshold's reasoning is in the code and in D-69's neighbourhood, but no D-70 row exists
D-95 was skipped	`DECISIONS.md`	The hosted Cloudflare copy's decision is described in D-97 (the URL) and in the D-91..D-94, D-96..D-99 rows around it, but no row carries the number 95 — the commit that introduced the copy used it, the record never did. 108 rows now carry 106 distinct numbers
D-3 still says "3-node → hub → wearable"	`DECISIONS.md` §1.1	The topology it describes is this system's, with two nodes. D-75…D-81 and §0 correct it, but the row itself was not edited
O-1's struck-through text still names a DRV2605L	`DECISIONS.md` §4	The fitted drive is a BC547 (D-77). The resolution note beside it is correct; the original wording is not
D-16 and `SYSTEM_UPDATE.md` quote the *previous* model's SHA-256	`DECISIONS.md` §1.2, `SYSTEM_UPDATE.md`	`4aef55d8…126bc11` is the pre-retrain artefact of 2026-10-29. The **deployed** model is `bd592d54…dce43b9`, and `deploy_config.json` and `check_artifact_drift.py` both carry the live one. D-16's quoted test AUC (0.9995) and cross-corpus AUC (0.9973) belong to the same superseded run; the deployed model's are 0.9993 and 0.9984. **Chapter 5 has the full comparison**
The headline board tables said SuperMini for the cots	`PROJECT_HANDOFF.md`, `TWO_COT_README.md`, `firmware/node/README.md`, `HARDWARE_COMPONENTS.md`, `SYSTEM_WIRING.md`, and the header comment of `node_crynet.ino` itself	The swap to **Pro Mini** nodes was made and verified live on **2026-11-02**, but those five documents still described it as planned or named the old board, and the sketch's own header named the SuperMini in two lines. **Corrected in this edition** — and the first edition of this dossier copied them, which is how the error reached page 8
The node's build figures were the day-one build	this dossier, chapters 7, 14 and 19	The earlier edition printed **977,617 B (74 %) / 165,972 B (50 %)**, which is the stage-2b compile of **2026-10-30**. The tree as it stands compiles to **983,177 B (75 %) / 166,404 B (50 %)**, measured 2026-11-23; the D-83 entry of 2026-11-07 already said "75 % flash, 50 % RAM". **Corrected in this edition**

None of these changes what the system does. All of them are the kind of thing a supervisor finds in two minutes, so they are listed here rather than left to be discovered.

The lesson worth keeping. This dossier was faithful to the repository, and the repository was behind the bench: three sessions were run on this two-cot hardware but logged in the sibling repo. A document built from prose inherits the prose's staleness. Only the source, the artefacts and the dated bench entries are authoritative — which is why this edition was checked against all three.

O-13 — one stale comment in the node sketch — CLOSED in this edition

`node_crynet.ino` described the hub's heartbeat as arriving "every 100 ms", in **two** places (the `g_hub_rssi` block and the peer-learning block). The hub sends it **every 1000 ms** (`link_stage.c:1495`, `(probe_now - last_heartbeat_us) > 1000000`). The code was always right and the comments were wrong; the node's link-bar freshness print is computed from the actual timestamp, so nothing misbehaved. It mattered because a reader sizing a timeout from that comment would size it ten times too tight — and the second comment also claimed the hub's address is learned "within a tenth of a second of boot", which is really about a second.

Both comments now say 1000 ms, and the peer-learning comment says "within about a second of boot". Comment-only: the sketch compiles to the identical **983,177 B (75 %) / 166,404 B (50 %)**, and `make_prebuilt.py --role node` was re-run so `BUILD_INFO.json`'s source fingerprint matches the edited file. `check_prebuilt.py` passes — 11 images match, all three roles up to date.

---

3. What was knowingly given up

These are not open items. They are closed decisions with a price, and the price is stated so that nobody later discovers it as a surprise.

#	What was given up	Why	The price
1	**Battery state of charge**	The panel already says which cots are answering; a second copy of that answer on a cot is redundant, and the figure was always an estimate from a resting-voltage curve (D-83)	**Nothing in the system warns a caregiver that a cell is nearly flat.** The cells are watched by eye. The telemetry payload shrank 34 → 32 bytes
2	**Every status LED**	Same decision (D-83)	A unit that is off looks identical to a unit that is asleep. The panel is the only status surface
3	**Cause classification** (hunger / pain / tired)	The BLE alert packet has one bit for it (D-1)	The caregiver gets "a cot is crying", not "why"
4	**A certified infant monitor**	Out of scope for an academic prototype	Detection accuracy, radio delivery guarantees, battery life and alarm behaviour are not established to any safety-critical standard, and the system does not replace supervision
5	**A soft trigger** — the hub never hears a cry below 0.030 RMS	The old system's 0.18 trigger was a real bug, but 0.030 is the floor that keeps a quiet room quiet	The system is blind at 1 m. The node's adaptive trigger partly recovers this; the hub's fixed gate does not
6	**Announcing on a two-window run without energy**	Fourteen false alarms in a quiet evening (D-72, D-73)	The run clause currently announces **nothing** — see Chapter 10 — and a real cry that only reaches two windows is not announced
7	**Per-window alerting**	A recording of cries has gaps; per-window alerting flaps (D-24)	A cry shorter than the 8-second hold is announced late
8	**A 3.3 V buck-boost feeding the board's 3V3 pin**	The LX-LCBST cannot be set below 4.2 V, so it needs a separate module (D-84)	The board's linear regulator burns (5.1 − 3.3) V × I ≈ 0.5–0.7 W as heat. Runtime is roughly half what a direct 3.3 V supply would give
9	**The DRV2605L haptic breakout**	Not needed; the BC547 low-side switch with a 22 Ω stall cap is simpler and cheaper (D-85)	The stall cap drops ~1.2 V of the rail, so the motor runs slower and quieter than on the originally-specified 10 Ω
10	**LiFePO4 cells**	Charge-current and thermal behaviour under the project's charging scheme is not established	18650 for the nodes and hub, a Li-Po pouch for the wearable
11	**Per-cot confidence in the BLE alert**	The packet is a frozen six-byte legacy format (D-42)	The confidence bar on the wrist is the loudest cry's figure, not a per-cot figure
12	**A second buzz on a cot handover**	The hub always sends `alert_type` 1 for a cry and the wrist buzzes only on the transition into 1	When the loudest cot changes hands, the wrist stays silent — by design, because the caregiver has already been alerted
13	**One radio doing two jobs**	ESP-NOW and Wi-Fi share the hub's radio, and ESP-NOW's channel is fixed at 6 (D-98, Chapter 12)	**The router's 2.4 GHz channel must be set to 6.** On any other channel the hub refuses the network and the dashboard goes dark — and the cots never notice, which is the point of the rule
14	**Audio never leaves the room**	The dashboard's records are metadata — numbers, durations, confidences; there is no microphone feed on any network	No remote listening-in, ever, at any price; the record cannot leak what it does not hold

4. What has *not* been measured

The full list is in Chapter 15; this is the same list arranged by what it blocks.

Unmeasured	What it blocks	How it would be closed
A real infant cry, end to end	Any claim about real-world recall	50–100 recorded clips on the real INMP441, then a notebook re-run and a bench session
Battery runtime on any unit	Any runtime claim	One unit, one cell, a log, and a stopwatch. The current figures are projections
The loud-conversation score	The loud path's safety	Play speech at a level that reaches 0.080 RMS and read the P(cry)
Two *independent* cries at once	The arbitration rule under genuinely colliding bursts	Two phones, two cots, one hub, at once. One clip heard by both mics is already verified (D-86)
This repo's D-86 image on hardware	Treating the port as bench-verified here	Flash the wearable from this repo and re-run the two-cot play (O-1f)
The haptic's rail on a cell, and the motor's real stall current	The alert's last mile under battery power — the buzz went missing on the pouch	One multimeter across the motor during a buzz, and a series ammeter on a bench supply (O-1d)
The I2S watchdog recovering a real fault	The silent-failure path	Flash it, then a played-cry end-to-end run (O-1c)
The 89 % airtime figure at two cots	The advertising policy's justification	Re-measure the duty cycle with exactly two cots transmitting
The distance curve beyond 15 cm and 1 m	The reach of the system	An acoustic alignment on a playback-start event, not on phase
A busy Wi-Fi channel (O-7)	The resend guarantee	A second router on channel 6
The tensor arena in internal SRAM	Nothing — it is an optimisation	Try it, measure, keep it if it fits
A lit-room panel photograph	The panel's tone curve	One photograph
The charge current and quiescent drain	The power budget	A meter and a cell

5. What to do next, in order

The ordering is by **value per unit of effort**, and it is deliberately not by what is most interesting.

First: meter the motor during an alert on the pouch (O-1d)

The wearable already runs on its pouch, so start there — with a meter, not a stopwatch. Put it on the cell, trigger an alert, and read the voltage across the motor while it should be buzzing: ≈3.3 V means no current (open coil or dry joint), ≈2.1 V means current flowing but a weak motor or a sagging rail, ≈0 V means the BC547 is not switching. That single reading resolves the one fault the pouch run exposed, and it costs five minutes.

Then the stopwatch. Runtime is still the largest gap between what the documents claim and what has been observed, and it is the item every other power question waits on: the charge current, the boost's quiescent drain, the regulator's heat. Build **one** node power assembly, validate it against the three rules of D-84, log it against the USB baseline, and only then buy the rest.

Second: record the real in-domain corpus (O-2, O-5)

Fifty to a hundred clips: ten infants × several cries, on the real INMP441, in the real room, with room negatives in the same takes. The tooling is written and proven — `collect_clips.py`, `label_session.py --truth`, `CONFUSION_SESSION.md`. This single action closes the project's largest accuracy risk, and it is the only thing that changes the headline number from "strong on a proxy" to "strong on the real thing".

Then, and only then, re-run the notebook with the new corpus included and re-deploy. The rule about the model being deployed only after its own evals have been run on the repo's own scripts (D-58) applies to this round exactly as it applied to the last one.

Third: measure the distance curve (O-6)

Because the system's reach is set by a software threshold and the microphone has no gain control, the distance curve is the honest statement of what the product can do. It is also a measurement that a sceptic will ask for.

Fourth: run the hub on a cell for one shift (O-1d)

One unit, one 18650, one log. It converts every runtime figure in the document from *projected* to *measured*, which is a categorical change in how defensible the power section is.

Fifth: close the documentary items (O-11, O-12, O-13)

Update `DEMO_RUNBOOK.md` to the peak-based target and to a two-cot `status` example; fix the stale heartbeat comment in the node sketch; and bring `DECISIONS.md`'s numbering and identity blocks into line. Cheap, and it removes three places where a reader could catch the project out.

Not on the list, and deliberately so

- **Re-tuning the model on the phone-speaker sessions.** The music problem was structural and the fix was a training round; the phone-speaker sessions are a *proxy for a proxy* and tuning on them would fit the artefact rather than the problem.
- **Lowering `CRYNET_TRIGGER_RMS` on the strength of 308 windows from one afternoon.** The adaptive trigger was the evidence-driven change; a second threshold change needs a second afternoon of evidence.
- **Moving the arena to internal SRAM.** At 101.8 ms per analysed window, for an event-driven system, this buys nothing a user can feel.
- **Adding a cause classifier.** The wire format has one bit. It is D-1, and it was deliberate.

---

6. How to defend this

If the question is *"what is the weakest part of this?"*, the answer is three sentences long and it does not require hedging:

> The model has never heard a real infant cry — every positive is either a clean corpus recording or a recording played through a phone speaker, while every negative is the authentic sound of this room through this microphone. The system's reach at one metre is set by a software threshold rather than by the microphone, which has no gain control — and a source close enough to drive that microphone to the rail is refused outright, so there is a close-range blind spot. And although the wearable has run on its pouch, **no unit's runtime has ever been measured**, which makes every runtime figure in the document a projection — one power assembly and a stopwatch would settle it.

Each of the three is (a) recorded in the project's own decision log, (b) stated in the same terms in the overview, the results chapter and this one, and (c) accompanied by the procedure that would close it and an estimate of what it would cost. Nothing about the system's accuracy is claimed beyond what was measured, and nothing about its limits is hidden.

That is what it means for a project to be defensible: not that there are no weaknesses, but that every weakness is known, named, measured as far as it can be, and priced.

Chapter 20 — Replication: building the whole thing from an empty checkout

The rule that makes this chapter possible

You do not need a Kaggle account, a GPU, or 4.6 GB of downloads to build and flash the system.

The trained artifacts are **in the repository**:

Artifact	Path	Committed?
The INT8 model	`research/out/crynet_int8.tflite`	**yes**
The model as a C array	`research/out/model_data.h`	**yes**
The Mel filterbank, Hann window, quantisation constants	`research/out/crynet_tables.h`	**yes**
The portable reference front end	`research/out/dsp_crynet.c` , `.h`	**yes**
The deployment manifest	`research/out/deploy_config.json`	**yes**
The metrics	`research/out/metrics.json`	**yes**
The staged copies the firmware actually compiles	`firmware/hub_infer/components/crynet_model/include/`	**yes**

`research/out/` holds **176 tracked files** — the artifacts *and* the logs that produced every number in this dossier. So the repository alone is enough to rebuild the firmware, flash it, and reproduce every bench measurement.

And the repository alone is enough to flash it **without a compiler at all**: `firmware/prebuilt/` holds the built images for all three roles, with a SHA-256 manifest and a source-fingerprint record. See §7.4.

What the repository is **not** enough for is **retraining**. The training run needs a GPU and the corpora, and that is a separate, optional path — given in §12 below.

The corollary matters for the defence: **the model is a committed, hash-identified artifact**, not a result someone has to trust. `deploy_config.json` carries the SHA-256, and `check_artifact_drift.py` proves that the bytes the firmware compiles are the bytes the evaluation scripts scored.

---

0. What you need before you start

Thing	Why	Where it comes from
A Windows or Linux machine with git, Python and a C compiler	everything	free
Python 3.11.9	the research pipeline	python.org
PlatformIO **6.2.0**	builds the hub	`pip install platformio`
`platform-espressif32` **8.0.0**	the board support package	PlatformIO, on first build
ESP-IDF 6.1.0 (`framework-espidf 4.60100.0`)	the hub's framework	**installed by hand — see §2.2. This is the single biggest trap in the project**
Arduino CLI + core **3.3.12**	builds the node and the wearable	`arduino-cli` + the ESP32 board package
`ziglang` **21.1.0**	the host C compiler for the parity test	`pip install ziglang` (there is no gcc on the reference machine)
An ESP32-S3-DevKitC-1 N16R8	the hub	the bench
2 × ESP32-C3 **Pro Mini** + INMP441	the cot nodes	the bench
1 × ESP32-C3 SuperMini + SSD1306 + coin ERM	the wearable	the bench

Versions that are known to work together (Chapter 17):

```
Python 3.11.9 · TensorFlow 2.21.0 (CPU) · librosa 0.11.0 · soundfile 0.14.0
numpy 2.4.6 · scipy 1.17.1 · ziglang 21.1.0 · PlatformIO 6.2.0
platform-espressif32 8.0.0 · ESP-IDF 6.1.0 · Arduino ESP32 core 3.3.12
FQBN: esp32:esp32:weact_studio_esp32c3
```

TensorFlow is only needed for the research pipeline, not for the firmware. If you only want to build and flash, you can skip it entirely.

---

1. Get the code

```
git clone https://github.com/Amazo123/CryNet-TwoNode.git
cd CryNet-TwoNode
git log --oneline -1        # 0a82711  at the time of writing
```

Check that the artifacts are present:

```
ls research/out/crynet_int8.tflite research/out/model_data.h research/out/crynet_tables.h
ls firmware/hub_infer/components/crynet_model/include/
```

If `research/out/` is missing the model files, the clone is incomplete — they are tracked, so `git checkout -- research/out` will restore them.

---

2. The toolchain

2.1 PlatformIO

```
pip install platformio==6.2.0
pio --version            # PlatformIO Core, version 6.2.0
```

The first build downloads `platform-espressif32 8.0.0` and the Xtensa toolchain. That download is several hundred megabytes and takes a while. It happens once.

2.2 ESP-IDF 6.1.0 — install it by hand

This is D-45 and it is the trap that cost the project real time. **Do not put a `platform_packages` URL for the framework in `platformio.ini`.** PlatformIO 6.2 installs a URL-specified package by treating the archive as an **npm package**: it picks the wrong `package.json` out of the tree, writes a bogus version into `.piopm`, and leaves `packages/framework-espidf` holding only a **test-fixture package**. CMake then fails with:

```
include could not find requested file: .../tools/cmake/project.cmake
```

The fix that worked, and it is idempotent:

1. `pio pkg install` once (or let the first build fail) so the platform package is present.
2. Find the platform's own framework archive — it ships with `platform-espressif32 8.0.0` as `framework-espidf 4.60100.0` (= ESP-IDF **6.1.0**). It is normally in `~/.platformio/.cache/downloads/` or the platform's own `packages/` directory.
3. Delete `~/.platformio/packages/framework-espidf` and **move the fully extracted tree into that place**. A partially extracted tree (missing `tools/`, `main/`, `examples/`, `version.txt`) is the signature of an install that was interrupted — delete it and start again.
4. Write `~/.platformio/packages/framework-espidf/.piopm` **by hand**:

```
{"type": "tool", "name": "framework-espidf", "version": "4.60100.0", "spec": {"owner": "platformio", "id":
null, "name": "framework-espidf", "requirements": "~4.60100.0", "uri": null}}
```

5. Confirm: `cat ~/.platformio/packages/framework-espidf/version.txt` → `6.1.0`.

Why 6.1.0 and not 5.3.2. 5.3.2 was the original target, but PlatformIO's registry only offers `~4.60100.0` (6.1.0) or `~3.50505.0` (5.5.0) for this platform, and the 5.3.2 archive could not be installed through PlatformIO at all. 6.1.0 satisfies the TFLM port's `idf >= 5.1` requirement.

Fallback if the port misbehaves under IDF 6.1 / GCC 14: install `platform-espressif32 6.9.0` (framework `~3.50301.0` = IDF 5.3.1, toolchain 13.2.0) and let PlatformIO fetch both.

2.3 Arduino CLI, for the node and the wearable

```
pip install arduino-cli          # or download the binary from arduino.cc
arduino-cli config init
arduino-cli config add board_manager.additional_urls https://espressif.github.io/arduino-
esp32/package_esp32_index.json
arduino-cli core update-index
arduino-cli core install esp32:esp32@3.3.12
arduino-cli board listall | findstr -i weact
```

The board is **WeAct Studio ESP32C3** — FQBN `esp32:esp32:weact_studio_esp32c3`. It is the exact module on the bench, and its definition is what the node's USB CDC configuration depends on.

> **USB CDC on boot must be enabled** in the board options, or the serial console never appears. `firmware/tools/flash_node.py` uses the FQBN as-is; if the console is missing, check this first.

---

3. Fetch the pinned firmware dependencies

```
bash firmware/tools/fetch_deps.sh
```

This clones exactly two trees, at exactly two revisions:

Dependency	Repository	Revision
esp-tflite-micro	`github.com/espressif/esp-tflite-micro`	`dd4085950a119d9e33a6de1d7be1fc7fe29e0cfc`
esp-nn	`github.com/espressif/esp-nn`	`744d48deae6aa4a4d7b93b6a80f548d5298813a4`

into `firmware/hub_infer/components/`, and then runs `patch_deps.py`, which makes the vendored TFLM port build against the vendored `esp-nn` instead of the component registry. **This is what keeps the build offline and pinned.** The script is idempotent — re-running it on an existing checkout does not re-clone.

> **Why the vendoring is not untidiness.** PlatformIO re-compiles IDF components with SCons using the *source basename* for object names, and the TFLM port has **twelve basename collisions** — the build dies with *"Multiple ways to build the same target were specified for .../circular_buffer.cc.o"*. Moving the components **inside** `firmware/hub_infer/components/` makes the paths relative and distinct. (D-46)

---

4. Prove there is no artefact drift — before you build

```
python firmware/tools/check_artifact_drift.py
```

This checks five things and must print **PASS**:

1. `research/out/crynet_int8.tflite`'s SHA-256 equals `deploy_config.json`'s `model_sha256`;
2. `model_data.h` embeds exactly `model_bytes` bytes;

3. `crynet_tables.h` defines the same macros as the ones mirrored into `crynet_model.h`;
4. `deploy_config.json` agrees with `crynet_tables.h` on scale, zero-point and threshold;
5. the staged copies under `firmware/hub_infer/components/` are **byte-identical** to the ones in `research/out/`.

This check exists because the notebook **re-derives the threshold and the quantisation scales on every run**. Dropping a new set of Kaggle outputs into `research/out/` without restaging them leaves the firmware compiling against the *old* constants with the *new* weights — a silent mismatch that no compiler would catch.

If it fails, the message says what moved. Restage the artifacts and re-run.

---

5. Build the hub

```
cd firmware/hub_infer
python -m platformio run
```

About a minute incremental; several minutes the first time (the toolchain is being unpacked).

Expected at the end:

Figure	Value
`firmware.bin`	~1.3 MB
Flash used	61.9 % of 2 MB (the partition table is `partitions_16mb.csv`, so 16 MB flash)
RAM used	12.9 % (42,236 B) of 512 KB
Free internal heap at boot	298,763 B

The build prints `ESP-NN enabled` and the octal PSRAM at 80 MHz.

> **The flash-mode subtlety (D-47).** `sdkconfig.esp32s3` still shows `CONFIG_ESPTOOLPY_FLASHMODE_QIO=y` as the *choice symbol*, while the *string* that esptool actually acts on says **`dio`**. Reading only the choice symbol gives the wrong answer. The effective configuration is `CONFIG_ESPTOOLPY_FLASHMODE="dio"`, and both `bootloader.bin` and `firmware.bin` carry flash-mode byte **`0x02`**. The bootloader switches the chip to QIO by itself once it is running, so DIO costs nothing at this workload — and a **QIO image makes this board watchdog-reset in a loop** (`rst:0x7 TG0WDT_SYS_RST`, no application output at all), because the ROM cannot read the flash in QIO before the bootloader has configured its own timing.

---

6. Build the node and the wearable

The Arduino IDE builds a sketch from **its own folder**, and the sources live in the repository, so they are copied into the sketchbook first:

```
python firmware/tools/install_sketches.py
```

That copies `node_crynet`, `node_capture` and `wearable_crynet` into `~/Documents/Arduino/`.

Then either use the Arduino IDE (board **WeAct Studio ESP32C3**, USB CDC on boot enabled), or build from the command line:

```
arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3 firmware/node/node_crynet
arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3 firmware/node/node_capture
arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3 firmware/wearable/wearable_crynet
```

Or, in one command per unit, which builds *and* flashes:

```
python firmware/tools/flash_node.py --port COM30 --sketch node
```

Figure	Value
Node flash / RAM (current tree, compiled 2026-11-23)	**983,177 B (75 %) / 166,404 B (50 %)**
App partition / RAM budget on the C3	1,310,720 B / 327,680 B

> **I-11.** The Arduino core's serial driver can block on a full TX buffer. The sketches call `Serial.setTxTimeoutMs(0)` so a write never stalls the audio loop. If you edit the sketches and see the trigger stop working, check that line survived.

---

7. Flash

7.1 The hub

```
python firmware/tools/flash_hub.py --port COM32
```

This writes the three images at the offsets the partition table expects — `bootloader.bin` at `0x0`, `partitions.bin` at `0x8000`, `firmware.bin` at `0x10000` — and then prints the boot log.

For a graphical flasher (the Arduino IDE's "ESP32 Flash Tools" extension), make one combined file:

```
python firmware/tools/flash_hub.py --merged
```

> **The merged image MUST be built with `--flash-mode dio`.** `flash_hub.py` passes it explicitly. A merged image built with QIO produces the watchdog-reset loop described in §5.

Other flags: `--flash-only` (flash, no log), `--log-only --seconds N` (just watch).

7.2 The nodes and the wearable

```
python firmware/tools/flash_node.py --port COM31 --sketch node      # cot node 1
python firmware/tools/flash_node.py --port COM37 --sketch node      # cot node 2
python firmware/tools/flash_node.py --port COM30 --sketch wearable # the wristband
```

The port map above is the one read from the last verified bench session with `who_is_who.py`; your ports will differ, and the script is how you find out which is which.

> **I-40 — the wrong-port trap.** A flash aimed at the wrong serial port parks the cot node in its **download mode**, and the radio then goes silent (**0 frames for two minutes**). The hub is on a CH343 bridge (COM32 on the reference bench); the C3 units enumerate on their own native USB. **Check `who_is_who.py` before you flash anything.**

7.3 Which unit is which

```
python firmware/tools/who_is_who.py
```

The node id lives in **NVS**, so it survives a reflash — but only if the flash tool does not erase that partition. This script resolves the ambiguity rather than leaving it to guesswork.

On a two-cot system the id is not a label — it is the difference between being monitored and being invisible. The hub's receive path drops any frame whose node id is outside 1..2 before it can claim a slot, so a cot left on a MAC-derived id is simply never heard. A node outside 1..2 prints a warning at boot, but the definitive check is that the hub's console lists it:

```
node 1 heard for the first time
node 2 heard for the first time
```

Set the ids over each cot's own serial console, then **re-check them after the flash** — the id lives in NVS, so a flash that erased that partition would leave a cot invisible with no other symptom:

```
id 1     # on COM31
id 2     # on COM37
```

```
python firmware/tools/node_console.py --port COM31 --command "id 1"
```

7.4 The offline path — no compiler, no network, no toolchain

This is the route that exists because the owner flashes boards himself, offline, on a machine that may never have seen this project. `firmware/prebuilt/` holds the built images for all three roles. It needs exactly one executable on the machine: `esptool`.

```
python firmware/tools/flash_unit.py --check                              # verify the images
python firmware/tools/flash_unit.py --list                               # what is on this machine

python firmware/tools/flash_unit.py --role hub     --port COM32 --log 25
python firmware/tools/flash_unit.py --role node    --port COM31 --log 15
python firmware/tools/flash_unit.py --role wearable --port COM30 --log 15
```

`--backup` writes the board's current flash to a file first; `--log N` prints N seconds of the boot log after the write.

Two rules the tool enforces, and both matter.

1. **The regions are written, not one merged file.** The offsets are `bootloader.bin` 0x0, `partitions.bin` 0x8000, `boot_app0.bin` 0xe000, `app.bin` 0x10000. A merged image fills the gaps between regions with `0xFF`, and writing that would **erase NVS** — which is where a cot keeps its id. The skipped gap (0x9000–0xE000 on the C3, 0x9000–0xF000 on the hub) is deliberate. A merged flash would silently turn a working cot into an invisible one.
2. **The hub images are DIO.** A QIO image makes the S3 watchdog-reset in a loop right after power-on (D-47). The bootloader header in `hub/bootloader.bin` says DIO.

The staleness check is the important part. `BUILD_INFO.json` records, per role, the SHA-256 of **every source file** that went into that image — 52 files for the hub, one for each sketch — with an explicit rule set for what does not count (build directories, `sdkconfig*`, the vendored third-party trees, editor artefacts). `check_prebuilt.py` recomputes those fingerprints and compares.

```
python firmware/tools/check_prebuilt.py     # are the images current with the source?
```

At the time of writing it reports:

```
== the offline images vs the source ==
  images: 11 files, all match SHA256SUMS
  hub       source fingerprint 7ebe61d46bec5b4b... over 52 files - up to date
  node      source fingerprint a715bd57edc46a8a... over 1 files - up to date
  wearable  source fingerprint 7071ab878821afe3... over 1 files - up to date

OK  the offline images are up to date with the source
```

`flash.bat` runs the same check in PowerShell — so it needs no Python — before it writes anything, and refuses a stale or damaged image. A GitHub workflow (`.github/workflows/prebuilt-check.yml`) rebuilds all three on every push and fails if the result differs from what is committed.

The reason this machinery is worth having: **the failure it prevents is silent.** A stale image flashes successfully, boots, and behaves like the old firmware, and the person holding the board has no way to tell. A fingerprint check turns that into a refusal with a message.

To rebuild the images from source (this needs the full toolchain of §2):

```
python firmware/tools/make_prebuilt.py
```

which compiles all three, copies the images in, rewrites `SHA256SUMS` and regenerates `BUILD_INFO.json`. **Never hand-edit either file.**

8. Verify the chain, stage by stage

Do these in order. Each one isolates a different layer, so a failure tells you where to look.

8.1 The hub's own boot self-check

```
python firmware/tools/flash_hub.py --port COM32 --log-only --seconds 30
```

The hub runs its **probe** before anything else. It computes log-Mel features on known probe audio and compares them with references generated by the **notebook's own** front end (`gen_probe_refs.py`), and it checks the PCM CRC.

Expected:

```
RESULT: PASS — 4 probes, 5184/5184 features exact on every probe
```

plus `front end 48.2 ms, inference 103.5 ms` and a clean radio counter line.

If the probes fail, nothing else starts. The hub deliberately refuses to bring up the radio, Bluetooth or the display if the probes fail — a wrong answer is worse than no answer. The two static-assert failures you will meet here are described in §13.

8.2 The node's audio path

```
python firmware/tools/verify_burst.py --port COM30 --out research/out/node_window.wav
```

This asks the node to dump its frozen window over its serial console in exactly the radio's framing, checks every fragment's CRC-16 and the whole window's CRC-32 with `zlib.crc32` on the host, and reports the mean, the RMS and the peak.

What the numbers tell you: the INMP441's DC offset is around **+18,000 counts**, so the mean should be near zero *after* the front end removes it, the RMS should be small (0.002–0.016 in a room), and the peak tells you whether the microphone is clipping (the refusal threshold is 31,000 of 32,767).

8.3 The radio burst

```
python firmware/tools/measure_burst.py --port COM32 --seconds 45 --sound-seconds 14
```

Play something loud during the `--sound-seconds` window. **Expected: 286/286 fragments, CRC-32 MATCH, burst duration 2.06–2.11 s** with the current airtime fixes (it was 5.54–6.93 s before them).

8.4 The end-to-end verdict and the BLE alert

```
python firmware/tools/listen_hub.py --port COM32 --seconds 90     # the hub's log
python firmware/tools/listen_node.py --port COM30 --seconds 90    # the node's log, alongside it
python firmware/tools/ble_alert_test.py --listen --seconds 90    # the BLE client
```

Play a cry clip from a phone at **15 cm** from the microphone. **Expected on the hub's log:**

```
node 1: front end 48.2 ms, inference 103.5 ms, P(no_cry)=0.0039 P(cry)=0.9961
node 1: ALERT — cry
```

and on the BLE client:

```
ALERT {'cot': 1, 'alert': 'cry', 'confidence_pct': 100}
```

That second line, on the node you played at, is the pass condition.

8.5 The full demonstration

`DEMO_RUNBOOK.md` is the procedure, and Chapter 16 explains it in full: the phone 15 cm from the mic, **one** play, let the whole 5-second clip run out, wait 5 s, read the log, wait 10 s so the 8-second hold clears before the next cot. After the last cot, wait 10 seconds of quiet and check that **nothing announces**.

---

9. Reproduce the host-side checks

These need no hardware and are the fastest way to convince a sceptic.

```
cd research

python make_stub.py && bash run_stub.sh        # synthetic end-to-end smoke test, minutes
bash run_real_fast.sh                          # the real corpus, subsampled, on CPU
pip install ziglang && python test_c_parity.py # C front end vs Python on the same PCM
python verify_int8_parity.py                   # INT8 vs float32
python eval_indomain.py                        # the deployed model on this room's audio
python eval_esc50.py                           # ESC-50: 40 crying_baby + 1,439 non-cry
python eval_music.py                           # GTZAN, ten genres
python eval_speech.py                          # speech negatives
```

Expected, with the deployed model:

Check	Result
C vs Python front end	max deviation **6.9×10^{-5}**
INT8 vs float32	ΔAUC **−0.00137**, decision agreement **99.956 %** over 11,386 windows
ESC-50 `crying_baby` recall	**36 / 40**
ESC-50 non-cry false alarms	**17 / 1,439 = 1.18 %**
GTZAN music false alarms	**2 / 965 = 0.21 %**
Quiet room	**0 / 236 windows**

The trigger study and the gate analysis

```
python adaptive_trigger_study.py     # this is what chose 3.0 as the trigger multiple
python analyze_gates.py              # what each gate in the announcement rule rejects
python simulate_freeze.py            # the stop-capped freeze rule against the peak-centred one
```

The confusion session — the in-room measurement

The full procedure is `research/CONFUSION_SESSION.md`. In outline:

```
python firmware/tools/collect_clips.py --port COM30 --out research/in_domain/session9 --seconds 60
python firmware/tools/label_session.py --session research/in_domain/session9 --truth cry
```

Nine one-minute takes (quiet room; speech at desk distance and at 30 cm; music at 1 m and at 30 cm; film at 1 m and at 30 cm; the phone-played cry at 15 cm and at 1 m), labelled with the session's **ground truth** rather than by loudness.

> `--truth` **is not optional.** The old auto-label called a clip a cry if it was *loud*, which for the music and film takes would have labelled the loudest non-cry clips as cries — and the evaluation would have been circular.

Expected: 0 false alarms in 308 windows of non-cry audio; 86.4 % per-window recall on the cry at 15 cm; and 0 of 590 records crossing the trigger on the cry at 1 m.

The demo kit

```
python build_demo_kit.py
python screen_clips.py
python score_demo_clips.py
```

Expected: 27 of 37 clips announce under the rule; 17 clip if turned up; the freeze lands 2.0–5.4 s after the clip starts, median 2.9 s.

The host-side graphics regression

```
python check_corners.py        # the panel's four rounded corners must be exact mirror images
python build_font.py           # regenerates the panel's font table
```

---

10. Bring up the dashboard

The dashboard is two copies of one service. Either is optional — the hub records whether or not either exists — but a full reproduction includes one of them.

The offline copy (no account, no dependencies)

```
python dashboard/server.py                      # http://localhost:8787
python dashboard/server.py --port 8787 --token SECRET   # require the hub's token
```

Python's standard library and SQLite; the database is one file. Open the page, and point the hub at it (below).

The hosted copy (Cloudflare Workers + D1)

The full procedure is `dashboard/worker/README.md`. In outline:

```
cd dashboard
npx wrangler d1 execute crynet --remote --file schema.sql   # the tables, once
npx wrangler secret put UPLOAD_TOKEN                        # the hub's X-CryNet-Token
npx wrangler secret put DASHBOARD_PASSWORD                  # the caregiver's login
npx wrangler deploy
```

Both secrets live only in Cloudflare's store (and, locally, in a git-ignored `.secrets`); neither is in the repository, and this dossier does not print them.

Introduce the hub to it

The router's **2.4 GHz channel must be 6** — the cot link's channel is fixed, and the hub drops a network that would move the radio off it. Then, on the hub's serial console:

```
net ssid <ssid> <password>
net set <url>          e.g. http://192.168.1.50:8787/api/v1/events, or the Worker's /api/v1/events
net token <token>      the same secret the service holds
upload                 send what has already been recorded
```

Expected: `net` reports batches and no failures; the page shows the hub online within ~30 s and its cots with it; a cot unplugged reads offline in about two minutes (30 s report, 90 s staleness, 10 s poll — D-105); a hub unplugged takes its cots offline **with it** (D-108).

Prove the contract

```
python firmware/tools/test_dashboard_contract.py            # throwaway server, no hardware
python firmware/tools/test_log_order.py                     # the uploader's ordering rule, on the host
```

and, on the hub itself, `logtest` — the same ordering proof against the real component on the chip.

---

11. Reproduce the hardware documents

```
cd tools/wiring
python build.py
```

This regenerates `SYSTEM_WIRING_DIAGRAMS.pdf` — the hub, node, wearable and overview pages, plus the LX-LCBST pad map (电池 / 输出 / 输入). The pages are generated from the same data as `SYSTEM_WIRING.md` and `HARDWARE_COMPONENTS.md`, so a change to the wiring is a change to the generator, not to the PDF.

---

12. Reproduce the *training* — the optional path

This is the only part that needs a GPU and the corpora. It is **not** needed to build or flash the system; it is needed to produce a *different* model.

11.1 Regenerate the notebook

The notebook is a **build artefact** and is never hand-edited (D-50):

```
cd research && python build_notebook.py
```

This emits `notebooks/crynet_training.ipynb` cell by cell (2,293 lines of generator) and validates it with `nbformat.validate()`. The reasoning lives inside the notebook as markdown cells, so a person running it reads the *why* next to the code.

11.2 Run it on Kaggle

1. New notebook → *Add Input* → attach nothing (everything downloads).
2. Accelerator **GPU T4**, Internet **On**.
3. Upload `notebooks/crynet_training.ipynb` → *Run All*.

Expected: ~10 min download (4.6 GB, resumable via `huggingface_hub`) + ~5 min feature extraction + ~20–40 min training for the variants + mining + INT8 export. Artifacts land in `/kaggle/working/out/`.

11.3 Or run it locally

```
pip install numpy librosa soundfile soxr scipy scikit-learn tensorflow huggingface_hub
CRYNET_FAST=1 python research/run_notebook_local.py     # smoke test, minutes
```

Environment variable	Effect
`CRYNET_FAST=1`	subsample the corpora and train 3 epochs
`CRYNET_OFFLINE=1`	never download; require the archives to be in the cache
`CRYNET_KINDS`	which variants to train (`conv_cnn,sep_cnn,temporal_cnn`)
`CRYNET_EPOCHS`	training epochs (default 35)
`CRYNET_MINING_ROUNDS`	hard-example mining rounds (default 1; 0 disables)
`CRYNET_CACHE` / `CRYNET_WORK` / `CRYNET_OUT`	cache, scratch and output directories

Caches: the HF archives at `research/_cache/audio/positive.zip`, `research/_cache/audio/negative.zip`, `research/_cache/metadata/all28479.csv`; the Kaggle archives at `research/_cache/<key>__<owner>__<slug>.zip`.

> `research/_cache/speech__fsd50k_speech.zip` **must not be deleted.** Its GitHub release has no asset on it, so it cannot be re-fetched.

11.4 Stage the new artifacts — in this order

1. Drop the notebook's `out/` into `research/out/`.
2. **Back up the old model**: `research/out/deployed_model_backup/`.
3. `python firmware/tools/check_artifact_drift.py` → it will now FAIL, because the staged copies in `firmware/hub_infer/components/` are the old ones. That is the check working.

4. Copy the new `crynet_tables.h`, `model_data.h` and `deploy_config.json` into `firmware/hub_infer/components/crynet_model/include/`, and the new `dsp_crynet.c` / `.h` into `firmware/hub_infer/components/crynet_dsp/reference/`.
5. `python firmware/tools/check_artifact_drift.py` → **PASS**.
6. `python firmware/tools/gen_probe_refs.py` → regenerate `crynet_probe_refs.h` by running the **notebook's own** log-Mel front end, extracted from the notebook. The on-device probes then compare against the training pipeline itself, not against a reimplementation of it.
7. Rebuild, flash, and re-run the evaluations **before** believing the new model: `eval_esc50.py`, `eval_music.py`, `eval_indomain.py`, `verify_int8_parity.py`, `test_c_parity.py`.
8. Only then flash the board and check the boot probe.

That is exactly the order D-58 followed, and the reason the music fix could be defended with numbers instead of hope.

---

13. The traps, and what each one looks like

Symptom	Cause	Fix
`include could not find requested file: .../tools/cmake/project.cmake`	PlatformIO installed the framework as an **npm package** (D-45, I-6)	Install ESP-IDF 6.1.0 by hand and write `.piopm` — §2.2
`Multiple ways to build the same target were specified for …/circular_buffer.cc.o`	PlatformIO's SCons object naming collides on basenames (D-46)	The components must live **inside** `firmware/hub_infer/components/` — §3
The chip watchdog-resets in a loop: `rst:0x7 TG0WDT_SYS_RST`, no output	A **QIO** image on this board (D-47)	The effective mode is **DIO**; `--merged` passes `--flash-mode dio`
The build succeeds but the boot probe fails: `max delta 87, 0/5184 exact -> FAIL`	The weights moved but the front end did not (I-41)	The static asserts in `crynet_dsp.c` are the drift guards; fix the constants or regenerate the probe references
The radio is silent: 0 frames for two minutes	The node was flashed at the wrong port and is parked in **download mode** (I-40)	`python firmware/tools/who_is_who.py` before flashing anything
The serial console never appears	**USB CDC on boot** is off in the board options	Enable it, or the sketch's logs go nowhere
The node's trigger stops working after an edit	The Arduino TX timeout (I-11)	Keep `Serial.setTxTimeoutMs(0)`
A stale model compiles into the firmware	The notebook re-derives the threshold and the scales every run	`check_artifact_drift.py` **before** the build — §4
The panel shows static / is rotated / has red and blue swapped / has hearts for letters	I-30, I-31, I-35, I-33	Chapter 14, rounds 1–4
The rounded corners are wrong	I-39	`python check_corners.py`
The hub asks for audio 5–11 times a minute in a quiet room	The adaptive trigger's 3× floor (D-63)	Expected; the hub then answers "no cry" on every window. Measured: **32 verdicts, 0 false alerts**
The hub joins nothing; `net` says the channel is wrong	The router is not on **channel 6** and the hub refused it on purpose	Set the router's 2.4 GHz channel to 6; the hub reconnects on its own
The dashboard shows the hub offline but the cots online	The old staleness bug (D-108)	Fixed in both copies of the service; the contract test holds the rule
Uploads answer 401	The hub's token and the service's `UPLOAD_TOKEN` disagree	`net token <t>` on the hub, or `npx wrangler secret put UPLOAD_TOKEN`; both ends must hold the same value

14. The time and disk budget

Stage	Wall time	Notes
PlatformIO's first-time downloads	15–40 min	once, ~2 GB
Hub build (incremental)	~1 min	~4 min from clean
Node / wearable build	~1 min each	Arduino CLI
Flash + boot log	~1 min	
Host-side checks (§9)	10–30 min	`run_real_fast.sh` is the slow one
Dashboard bring-up	~5 min	§10; the offline copy is one command
The full Kaggle training run	45–90 min	GPU; optional
The confusion session	~30 min	nine takes plus the labelling
This dossier	~1 min	`python docs/dossier/build.py`

Disk: the toolchain is ~2 GB, the corpora are 4.6 GB (cacheable), the repository with its tracked `research/out/` is well under 500 MB.

---

15. The checklist

Print this. Tick it. You have reproduced the project.

- [ ] `git clone` → the artifacts are in `research/out/`
- [ ] PlatformIO 6.2.0 installed, ESP-IDF 6.1.0 installed **by hand**, `.piopm` written
- [ ] `bash firmware/tools/fetch_deps.sh` → two trees at two pinned revisions
- [ ] `python firmware/tools/check_artifact_drift.py` → **PASS**
- [ ] `python firmware/tools/check_prebuilt.py` → the offline images are **up to date with the source**
- [ ] `python -m platformio run` → the hub builds
- [ ] `python firmware/tools/test_dashboard_contract.py` → **OK** (Chapter 12's contract holds)
- [ ] `python firmware/tools/test_log_order.py` → all six tests **pass**
- [ ] `arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3 …` → the node and the wearable build
- [ ] The hub flashes, boots, and prints `RESULT: PASS — 4 probes, 5184/5184 features exact`
- [ ] Each cot's id is set (`id 1`, `id 2`) and **still set after the flash**
- [ ] The hub's console shows `node 1 heard for the first time` and `node 2 heard for the first time`
- [ ] `verify_burst.py` → the node's window, CRC-32 MATCH
- [ ] `measure_burst.py` → 286/286 fragments, 2.06–2.11 s
- [ ] `listen_hub.py` + `ble_alert_test.py` → `P(cry)=0.9961` and `ALERT — cry` on the right cot
- [ ] `test_c_parity.py` → max deviation 6.9×10^{-5}
- [ ] `verify_int8_parity.py` → ΔAUC −0.00137, agreement 99.956 %
- [ ] `eval_music.py` → 2 / 965 = 0.21 %; `eval_esc50.py` → 36 / 40 and 17 / 1,439
- [ ] The confusion session → 0 false alarms in 308 non-cry windows, 86.4 % recall at 15 cm

- [ ] Both cots hear one clip → the hub's mask reaches `0x03` and the wrist reads `COTS 1 2` (D-86)
- [ ] Five cry episodes → **five** buzzes, with handovers silent
- [ ] The phone is far enough back that no window's peak reaches **31000** — check peak, not RMS
- [ ] `python docs/dossier/build.py` → this document, in PDF

Chapter 21 — Glossary

Every term a defender might be asked about, defined once. Where a term has a project-specific meaning that differs from its ordinary one, that is stated.

A

ADC / ID scale / `IN_SCALE`, `IN_ZP` — the affine parameters that map a float into the model's INT8 input. `q = round(v / scale) + zero_point`, clamped to [−128, 127]. The deployed model's `IN_SCALE` is 0.116329655 and `IN_ZP` is −70. These are re-derived on every training run, which is why the artefact-drift check exists.

Adaptive trigger (D-63) — the node's trigger, defined as a **multiple of the node's own running noise floor** rather than a fixed level. `TRIGGER_MULT 3.0`, clamped to [0.005, 0.030]. The floor rises exponentially (`g_floor += (g_rms - g_floor) * 0.002f`, a ~50 s time constant) and falls instantly. Two consecutive readings above the level are required.

Announcement rule — the hub's policy for turning a window's model output into an alert. Six gates, in the order given in Chapter 10. **The log is the evidence; the wrist is the output** — an alert proves a notification was published, not that the right cot was named.

Arduino CLI (`arduino-cli`) — the Arduino IDE's command-line compiler, used to build the node and the wearable against core **3.3.12**.

Arena (tensor arena) — the single scratch buffer TFLite Micro allocates for the model. 262,144 B reserved, 108,328 B used, and it lives in **PSRAM** (open item O-4).

B

Bandpass / crest factor — the ratio of a signal's peak to its RMS. A cry in this system has a crest of 2.2–2.9; the loud music that used to false-alarm had 5.6; the loudest film had 6.6–6.9. `CRYNET_QUIET_CREST 3.5f` is the gate that uses this.

BC547 — the small-signal NPN transistor that switches the haptic motor's current. Used here as a **low-side switch**: the motor's + lead goes to 3.3 V through a 22 Ω stall cap, and the transistor pulls the motor's − lead to ground.

Basic auth — how the caregiver's browser authenticates to the dashboard: a username and password, separate from the hub's upload token on purpose — a leaked password only reads the record; a leaked token writes to it (Chapter 12).

BLE — Bluetooth Low Energy. The hub is a **peripheral and broadcaster** (never a central); the wearable is the client.

Burst — one complete window of audio moving from a node to the hub: **286 fragments** of 224 payload bytes each, ESP-NOW, unicast, with a CRC-16 per fragment and a CRC-32 over the whole window.

C

CMVN — **cepstral mean and variance normalisation**, applied over the *whole* 81 × 64 feature plane of a window: one mean and one standard deviation, with `+1e-6f` in the denominator. This is what makes the model **level-invariant** — ICD cries sit at RMS ≈ 0.006 while negatives sit at 0.3–0.46, and without CMVN the model would be a level meter.

`CRYNET_TRIGGER_RMS` — 0.030. The hub's fixed audio-pull threshold. *Not* the node's trigger, which is adaptive.

CRC-16/CCITT-FALSE — the per-fragment checksum (polynomial 0x1021, init 0xFFFF, no reflection). **CRC-32/IEEE** — the whole-window checksum, computed on the host with `zlib.crc32`.

CRC-32 MATCH — the log line that means the window the hub reassembled is bit-identical to the window the node froze. It is the proof that the audio crossed the radio unchanged.

COTS 1 2 — the wearable's alert-face label when **both** cots are crying (D-86). Built from the state characteristic's `crying_mask` unioned with the cot the alert packet named. Rendered at text size 2 (96 px) because size 3 would be 144 px on a 128 px glass. One cot still reads `COT n` at size 3.

Cry mode (D-69) — the node's behaviour after it has frozen a window: it will not freeze again until the room goes quiet (`g_rms < 0.01f && g_loud_streak == 0`), unless a much louder sound arrives (`g_rms >= 3.0f * g_frozen_rms`).

Cry log — the hub's record of every event, thirty-two bytes per record in six megabytes of SPIFFS reserved for it from the first build. Written whether or not there is a network; uploaded in batches when there is. Its record numbers are what the wrist's alert shows and what the dashboard stores (Chapter 12).

D

D-n — a **decision** in `DECISIONS.md`: what was decided, why, and what it cost. **108 decision rows** numbered D-1…D-108, carrying **106 distinct numbers**: D-51 and D-52 were each used twice for different decisions, and D-70 and D-95 were skipped. All 108 are in this repository's file and reproduced in Appendix A.

D1 (Cloudflare) — the SQLite database behind the hosted dashboard copy. One Worker, one D1 database, two secrets; see Chapter 12 and `dashboard/worker/README.md`.

DIO / QIO — the two flash access modes. This board **must be booted with DIO**; a QIO image watchdog-resets in a loop before the bootloader has configured its own timing. The bootloader switches to QIO by itself once running, so DIO costs nothing. The trap: `sdkconfig` shows the *choice symbol* `CONFIG_ESPTOOLPY_FLASHMODE_QIO=y` while the *string* esptool acts on says `dio`.

DFT / probe — the hub's **boot self-check**: it computes log-Mel features on known probe audio and compares them with references generated by the notebook's own front end. Pass is **5184/5184 features exact on every probe**, plus a PCM CRC check. The hub refuses to bring up the radio, Bluetooth or the display if the probes fail.

Dropout / fade — the mic's DC offset. The INMP441 sits at about **+18,000 counts**, so the front end subtracts the mean in exact **int32** arithmetic before anything else.

E

ESC-50 — a 2,000-clip environmental sound corpus. 40 of its clips are `crying_baby` and are held out entirely as a cross-domain test; the other 1,960 are negatives.

ESP-NOW — Espressif's Wi-Fi-derivative protocol: no router, no association, direct point-to-point frames on a chosen channel (**channel 6** here).

ESP-NN — Espressif's optimised kernels for INT8 neural networks. On an S3 it is the difference between ~40 ms and ~2.3 s per convnet invoke. It is vendored and pinned alongside the TFLM port.

ESP-IDF — Espressif's IoT framework, and the hub's. **6.1.0** in this project, installed **by hand** (D-45).

F

FIR / filterbank / `slaney` — the Mel filterbank, generated by `librosa` with the **slaney** normalisation (not HTK), 64 bands from 20 Hz to 8 kHz, with **497 non-zero weights**. The same filterbank is emitted as a C table, so the firmware and the training pipeline compute *identical* features.

`FROZEN_HOLD_MS` — 3000. How long the node keeps a frozen window before discarding it.

FLOOR (noise floor) — the node's running estimate of the room's background level. In the reference room it settled at **0.0028** with a level of 0.0083.

G

GAP (global average pooling) — the model reduces each 16-channel feature map to one number per channel before the dense layers. It is what keeps the parameter count at 122,770.

GTZAN — the 1,000-clip, ten-genre music corpus. It was **not** in the training mix until D-51, which is why the model used to false-alarm on **8.19 %** of it. Now **0.21 %**.

H

Hann window — the 512-point analysis window, **symmetric** (not periodic). Symmetric matters: the two differ by one sample in the endpoints, which is enough to break the parity test.

Hop — 400 samples (25 ms). 81 frames fit in 2.0 s after 256 samples of front padding and 512 of rear padding.

I

I-n — an **issue** in `DECISIONS.md`: what it looked like, how it was diagnosed, what was done, what happened. 41 numbered, I-1…I-41.

ILI9341 — the 2.8" 320 × 240 TFT controller on the hub. Driven by a hand-written driver (`ili9341.c`, 413 lines) over SPI2 at 40 MHz, write-only.

INT8 — 8-bit integer inference. The deployed model is **137,384 bytes**. Measured against float32: ΔAUC **−0.00137**, decision agreement **99.956 %** over 11,386 windows.

IRQ / ISR — an interrupt. The hub's radio receive path runs in a Wi-Fi callback, and one specific lesson (**D-2x, the peer deadlock**) is that adding a peer from inside that callback deadlocks: the peer must be added **deferred**, out of the callback.

J

Jetson / JSON / JPEG — not used.

K

Keyed / key-value — not used.

L

Librosa — the Python audio library whose `slaney` Mel filterbank and log scaling the C front end reproduces exactly.

`log_mel` / `log10(max(P, 1e-10))` — the power spectrum is converted to decibels and floored at **−80 dB** before CMVN.

LX-LCBST — the adjustable boost module that takes a 1-cell Li-ion to 4.2–28 V. Its trimpot must be set to **5.1 V before any board is attached** (D-84).

M

Mel band — one of the 64 triangular filters of the Mel filterbank. 64 of them, over 20 Hz–8 kHz.

Model data (`model_data.h`) — the INT8 model as a C byte array. 8,604 lines of generated data; it is the reason the appendix reproduces the source but not this file.

MSB / LSB — the on-device feature-parity criterion is **worst ≤ 1 LSB** and **≤ 1 % of the 5,184 features differing**; the boot probes require **exact** agreement.

Link bar / signal bar — the four-bar glyph that shows the strength of a radio link, on the hub's panel for each cot and on each cot's console for the hub. Four bars above −60 dBm, three above −70, two above −80, one above −90. **The two ends measure different directions** — the hub reads the cot's telemetry, the cot reads the hub's heartbeat — so they are not expected to agree.

N

NVS — the ESP-IDF **non-volatile storage** partition. The node's id lives there, so it survives a reflash — provided the flash tool does not erase that partition. That proviso is why the offline flasher writes the flash *regions* rather than one merged file: a merged image fills the NVS gap with `0xFF` and would silently turn a working cot into an invisible one.

Node id — the cot's number, **1 or 2**. It is not a label: the hub's receive path drops any frame whose id falls outside 1..2 before it can claim a slot, so a cot on any other number is never heard. `NODE_LINK_MAX_NODES` is **2** and `HUB_DISPLAY_MAX_NODES` is **2**; the sketch carries `CRYNET_COT_COUNT = 2` to match. The id is set over the serial console with `id 1` / `id 2`.

`node_capture` — the second node sketch: it records raw audio over USB and makes no decisions. It is how the in-domain corpus was collected.

Node (cot node) — an ESP32-C3 **Pro Mini** with an INMP441. It samples at 16 kHz, tracks its own noise floor, freezes a two-second window when something crosses its trigger, and sends that window to the hub as a 286-fragment ESP-NOW burst.

Heartbeat — a twelve-byte frame the hub broadcasts **once a second** asking for nothing. Its only job is to give each cot an RSSI reading of the hub, from which the cot builds its own link bar. It shares the REQUEST's layout exactly; a heartbeat built with the full header would be sixteen bytes and every one would be rejected as malformed. `research/test_heartbeat_framing.py` holds the rule.

Notch — the one-pixel line on a cot card's level bar marking **that cot's own trigger level**: the level at which it ships audio to the model rather than only listening. The bar is square-rooted and scaled to `LEVEL_BAR_FULL 0.25`, so the notch and the fill agree by construction. The bar turns **amber** once the cot crosses it and **rose** while the hub is announcing a cry from it.

O

O-n — an **open item**: work identified and deferred. **O-1…O-9** in `DECISIONS.md`, plus the two-cot sub-items **O-1a…O-1g** and **O-10…O-13** discussed in Chapter 19. **O-2** (the model has never heard a real infant cry) is the largest, and **O-1d** (no *runtime* has ever been measured, though the wearable has run on its pouch) is the largest hardware gap. **O-1b is closed** by D-86; **O-1f** is this repo's unflashed D-86 image and **O-1g** the microphone's close-range blind spot.

OdB (one-decibel-below) — not used.

Prebuilt / offline image — the built firmware for all three roles, committed in `firmware/prebuilt/` with a `SHA256SUMS` manifest and a `BUILD_INFO.json` recording the SHA-256 of every source file that went into each image. `flash_unit.py` writes it with `esptool` alone — no compiler, no toolchain, no network — and `check_prebuilt.py` refuses to flash an image that is stale relative to its source.

P

Presence report — the hub's every-30-second post that says who is in the room: itself, and for each cot whether it is answering, deaf, and how strong its radio is. A report yields to a cot's burst (up to 20 s), the service calls a hub gone after 90 s without one (D-105), and a cot is online only while the hub's word about it is fresh (D-108).

Pro Mini (ESP32-C3 Pro Mini) — the board the **two cot nodes** use. Same ESP32-C3 chip, 4 MB flash and native USB (`303A:1001`) as the SuperMini, so the firmware image is identical; the swap was made and verified live on 2026-11-02. The two on the bench are `90:DA:72:85:C5:44` (id 1) and `90:DA:72:87:CB:98` (id 2). See **SuperMini** for the board that moved to the wrist.

PSRAM — the hub's 8 MB of octal external RAM. The framebuffer (153,600 B) lives here, and so does the tensor arena. The DSP's scratch buffers **do not** — putting them in PSRAM took the front end from 41.7 ms to 135 ms (D-46's neighbour, the memory move).

P(cry) — the model's output probability of a cry. On real audio in this room the model is not ambivalent: measured values are either near-zero or saturated, with **nothing between 0.2148 and 0.9922** across 45 verdicts.

Partial window — a burst with holes in it. **A burst is analysed only when every one of its 286 fragments is present** (D-23): the link tolerates loss by *recovering* it, not by pretending it away.

Peaks / `CRYNET_CLIP_PEAK` — 31000 (0.946 of full scale). A window whose peak reaches this is **refused**, not clipped and analysed.

Promiscuous mode (`CRYNET_LINK_PROMISCUOUS`) — a build flag that counts the radio frames carrying the CryNet magic against the frames the ESP-NOW layer hands up. It is the instrument that separates "lost in the air" from "dropped by the hub's receive path".

Prometheus / pytest — not used.

Q

QIO — see DIO.

Quantisation — see INT8 and `IN_SCALE` / `IN_ZP`.

Quiet room — the reference measurement: **0 false alarms in 236 windows** on this INMP441, and **0 in 20.5 minutes** of a room with people talking.

R

Radio busy — the fraction of time the radio is occupied. **89 %** in the busiest recorded session — which is why the BLE advertising interval is throttled during a burst.

Reassembly / resend — the hub reassembles fragments into the window; if a fragment is missing it logs "the burst stalled with N of 286 fragments", asks the node for the missing ones, and the resend rounds recover nearly all of it. `RESEND_DEDUPE_MS 250` on **both** sides (same sequence number **and** same bitmap).

RMS — root mean square, the project's usual level measure. Note the caveat: the quantity that actually decides refusal is the **peak**, not the RMS (see O-14).

Rotation (`MADCTL_MY`) — the panel is **landscape-first** and wants `MADCTL_MY`, not the `MADCTL_MV` the legacy firmware used. A legacy setting is not evidence about a different module.

S

SuperMini (ESP32-C3 SuperMini) — the board the **wearable** uses. It was the original cot-node prototype; when the cots moved to Pro Mini stock on 2026-11-02 this board was reassigned to the wrist. Same chip and same FQBN as the Pro Mini, which is why one toolchain covers all three C3 units.

SPIFFS — the flash file system that holds the cry log, on the six-megabyte `storage` partition the partition table reserved from the first build (D-96). Append-only from the log's point of view: records are written once and never changed.

SNTP — how the hub learns the time when it is on a network. Two servers configured; the wall clock is anchored to the hub's monotonic uptime and persisted in NVS, so a restart loses only the duration of the power cut (Chapter 12).

S3 / ESP32-S3 — the hub's MCU: dual-core Xtensa LX7 at 240 MHz, 512 KB SRAM, 8 MB PSRAM, 16 MB flash.

Sample rate — 16 kHz, mono, everywhere.

Silane / slaney — the two Mel-filterbank normalisations; this project uses **slaney**.

Silence gate (`SILENCE_RMS_GATE`) — 0.0015. Windows below it are not asked about.

SMA / SNR / softmax — the model's final layer is a two-way softmax over `[no_cry, cry]`.

Splash — the startup sequence on the panel (`SPLASH_MS 5600` = `BOOT_MS 2400` + the title card + `FADE_MS 600`). Two faces: a boot face with the wordmark and a progress bar, and a title card with the project's name, the owner's name and id, and `ESP32-S3 HUB - TWO COT NODES`.

Static assert — a compile-time check. There are **eight** in `crynet_dsp.c`, guarding the input scale and zero-point, the output scale and zero-point, the silence gate, `N_SAMPLES 32000`, `N_FRAMES 81` and `N_MELS 64`. They are the reason a mis-staged model fails at build time rather than on a cot.

T

TFLM — TensorFlow Lite Micro, the microcontroller inference runtime.

TFLite — the `.tflite` file itself; `crynet_int8.tflite` is the deployable artifact.

Threshold — 0.95 on the INT8 output, chosen as **the largest threshold in `arange(0.05, 0.96, 0.01)` whose recall is still ≥ 0.95**. It is calibrated on the validation fold, not chosen.

Trigger — the level at which audio is interesting enough to be sent. Three of them, and they are different things:

Trigger	Where	Value
The node's adaptive trigger	the node	**3 × the node's own floor**, clamped to [0.005, 0.030]
The hub's fixed pull	the hub	`CRYNET_TRIGGER_RMS` **0.030**
The announcement rule's gates	the hub	`CRYNET_ANNOUNCE_MIN_RMS` 0.006, `CRYNET_MAYBE_MIN_RMS` 0.020, `CRYNET_PULL_CRY_RMS` 0.050, `CRYNET_LOUD_ANNOUNCE_RMS` 0.080

U

UMD / USB CDC — the C3's serial console. **USB CDC on boot must be enabled** or the console never appears.

Uptime / `up 0:39` — the panel's footer shows the hub's uptime and the number of cots linked.

V

VCC / VBUS / VO+ / VO− — the power-chain vocabulary. `VO+` / `VO−` are the LX-LCBST's output terminals; `VBUS` is the USB 5 V rail, and **it is the same net as the board's 5 V pin** through the board's protection diode. That is why the boost and USB must never be connected at the same time (D-84).

`VERBOSE` / `vs` / `volatile` — not used in this sense.

W

Wearables / wearable — the caregiver's wristband: an ESP32-C3 SuperMini, an SSD1306 OLED and a coin ERM on a BC547 low-side drive. It runs **no model**. It receives a six-byte alert packet and vibrates.

Window — two seconds of audio: 32,000 int16 samples at 16 kHz. The unit the model classifies and the unit the radio carries.

Wi-Fi coexistence — ESP-NOW and BLE share one radio. The hub's Bluetooth is brought up **after** the radio, and the advertising interval is throttled during a burst (Chapter 11).

Watermark — the number the uploader may claim as sent after a batch: everything up to it has gone, nothing above it has. Clamped below the lowest number that did not fit in the batch and the lowest number an open episode still holds, so the hub never says "sent" while a record is still being measured (D-94).

WDT / watchdog — `rst:0x7 TG0WDT_SYS_RST` is the reset reason that identifies a QIO image looping on this board.

X

XNNPACK — a host-side delegate for TFLite. It is **filtered out** of the INT8 conversion, because it has no TFLM kernel; the notebook asserts the operator list at export time and records it in `deploy_config.json`.

Y

Y-axis / `MADCTL_MY` — see Rotation.

Z

Zeropoint / zero-point — see `IN_ZP`, `OUT_ZP`.

Zlib — the host's CRC-32 and DEFLATE library. `zlib.crc32` is the *host-side reimplementation of the check*, used by `verify_burst.py` so that the reference is independent of the implementation under test.

Acronyms at a glance

ADC	analog-to-digital converter	**LM**	linear model
AP	access point	**LRU**	least recently used
AUC	area under the ROC curve	**ML**	machine learning
BLE	Bluetooth Low Energy	**MP3/WAV**	audio container formats
BN	batch normalisation	**NR**	new radio (5G)
BW	bandwidth	**PSRAM**	pseudo-static RAM
CMVN	cepstral mean-variance normalisation	**PWM**	pulse-width modulation
CNN	convolutional neural network	**RIR**	room impulse response
CRC	cyclic redundancy check	**RMS**	root mean square
CS	chip select	**RTC/RTT**	real-time clock / round-trip time
DC	direct current (offset)	**SNR**	signal-to-noise ratio
DFT	discrete Fourier transform	**SoC**	system on a chip
DMA	direct memory access	**SPI**	Serial Peripheral Interface
DSP	digital signal processing	**SRAM/DRAM**	static / dynamic RAM
FFT	fast Fourier transform	**TFLM/TFLite**	TensorFlow Lite (Micro)
GAP	global average pooling	**TX/RX**	transmit / receive
GPIO	general-purpose I/O	**UART**	serial port
I²S	Inter-IC Sound (mic data bus)	**USB**	Universal Serial Bus
IDF	Espressif IoT Development Framework	**WAV**	audio file format
INT8	8-bit integer	**WDT**	watchdog timer

The project's own vocabulary, in one table

Term	Meaning
D-n	a decision — what, why, what it cost
I-n	an issue — the symptom, the diagnosis, the fix, the outcome
O-n	an open item — deferred work, with the reason it is still open
The log is the evidence	an alert proves a notification was published; only the hub's serial log proves the right cot was named, at what confidence, on what window
The wrist is the output	the wearable is what the caregiver feels; it is not the evidence
The model has never heard a real infant cry	every positive in the room is a phone-speaker proxy; the negatives are authentic
Frozen contract	the signal contract (D-11): the firmware, the exported tables and the weights change together or not at all
One artifact per checkpoint	the project's working discipline: deliver one thing, verify it, then move on
Passive observation before intervention	the host tools measure without stimulating, so a result cannot be an artefact of the tool
Host-side reimplementation of the check	verify the thing on the host with an independent implementation (`zlib.crc32` , the notebook's own front end), not with the code under test
Measured vs projected	every runtime figure is labelled *projected* until it is read off a cell. None is, yet (O-11)

Appendix A — The project record, verbatim

The repository's own documents, reproduced in full and unedited. Every D-n decision, every I-n issue, the wiring, the bill of materials, the bench runbook, the flashing guides and the handoff briefing are here exactly as they are in the tree, so that the narrative chapters can be checked against the primary record rather than taken on trust. Where a document is out of date, Chapter 18 says which one and why.

18 files reproduced in full.

The decision and issue record

DECISIONS.md

444 lines · 172,568 bytes

```
# CryNet — decision and issue log

> **What this file is.** The project's report and documentation record: every
> decision that was made, why it was made, every issue that was hit, how it
> was diagnosed, what was done about it, and what the result was.
>
> **Repo:** `C:\Users\HP\.t3\projects\fp-two-node` (branch `main`, remote
> `origin` → `https://github.com/Amazo123/CryNet-TwoNode.git`)
> **Companion files:** `SYSTEM_UPDATE.md` (the living status log and recovery
> layer — rewritten after every change) and `PROJECT_HANDOFF.md` (the deep
> briefing: the frozen contracts and the file map).
> **Sibling repository:** `C:\Users\HP\.t3\projects\fp-2` (branch `main`,
> remote `crynet` → `https://github.com/Amazo123/CryNet.git`) — the **three-cot**
> system. The two repositories share their history up to commit `99bb257` and
> then diverge; the two-node system is the one on the bench.
> **Last updated:** 2026-11-11
>
> **Conventions.** Each decision is written as **D-n** and each issue as
> **I-n**. A decision that was later changed carries a "revised by" note
> rather than being deleted, so the reasoning history survives. Every claim
> here was measured on the real hardware or in a real run; nothing is
> asserted from theory alone.

---

## 0. The system in one paragraph

**Two** ESP32-C3 cot nodes (INMP441 microphone, I2S at 16 kHz) → ESP-NOW
radio → an **ESP32-S3-DevKitC-1 hub** that reassembles each node's two-second
window, runs a TFLite Micro INT8 binary cry/no-cry classifier on it, and
shows the result on an ILI9341 TFT and publishes it over BLE GATT → a wearable
(ESP32-C3 + SSD1306 OLED + **BC547 haptic motor**) on the caregiver's wrist.
The classifier, the model and the signal contract are shared with the
three-cot sibling repository `fp-2`; what differs here is the node count, the
haptic circuit, and the wearable's role. The previous firmware attempt failed
after flashing, so the project was rebuilt from model training upwards.

---

## 1. The decisions

### 1.1 Scope and architecture

| # | Decision | Why |
|---|---|---|
| **D-1** | **Binary cry / no-cry**, not a multi-class head. | The BLE packet the wearable firmware was written against carries `alert_type 0x01` for a cry. There is no wire format for "cough" or "tired" without changing the wearable. Binary keeps the protocol honest and the alert unambiguous. |
| **D-2** | **Rebuild the firmware from scratch in ESP-IDF** rather than port the abandoned Arduino code. | The legacy hub was one 46 KB `.ino` with global state and a `loop()`/FreeRTOS mix. Its DSP implemented the *old* feature contract (128 mel bands, HTK scale, no normalisation), which is incompatible with the new model, so a "port" would have meant rewriting the signal path anyway. Rewriting gave a component tree, a single explicit state machine, and a wire format with per-fragment integrity. |
| **D-3** | **Keep the hardware facts, re-derive the software.** Physical pin assignments, the 3-node → hub → wearable topology and the legacy BLE UUIDs were inherited unchanged; the model, features, trigger, threshold, transport and application architecture were re-derived. | The boards and the wiring are facts about the user's real hardware. The parts that failed were the ones that had been guessed. Section 6 of `SYSTEM_UPDATE.md` carries the line-by-line comparison against the legacy repo. |
```

| **D-4** | **Model training first, firmware second.** The model is retrained and exported before any firmware is written. | The legacy system's trigger (`RMS ≥ 0.18`) meant a soft cry (the ICD corpus sits at RMS ≈ 0.006) never reached its classifier at all, so the old model was never fairly tested. Training first means the firmware is built against a contract that is known to work on real audio. |
| **D-5** | **One artifact per checkpoint; the agent updates `SYSTEM_UPDATE.md` after every edit and never pastes large logs into chat.** | Sessions were repeatedly lost to "400 Context Limit Exceeded". The status log is the recovery layer, and the checkpoints mean a crash costs one stage rather than the project. |

1.2 Data

#	Decision	Why
D-6	**Every candidate corpus was profiled at audio level** (sample rate, duration, RMS, autocorrelation F0, spectral centroid and flatness) before any of it was used.	Sample rate and level decide whether a corpus can train a model that will run behind a fixed signal contract. A corpus that is 8 kHz or whose "silence" is loud cannot be mixed into a dataset that has none.
D-7	**Primary corpus: ICD (`hoyyu1/infant-cry-detection`)**, 28,478 clips, all 16 kHz mono 5.00 s, five shipped folds. Added: **Baby Cry 4-class** (cry diversity and a cross-corpus test), **ESC-50** (negatives and 40 held-out `crying_baby`), **Snoring** (hard negatives).	ICD is the only large, clean, 16 kHz cry corpus available without gating. The others widen the negative class and prove the model is not tuned to one recorder.
D-8	**Rejected corpora and the reason for each:** `mennaahmed23/baby-cry*` family — **contaminated** (fake "silence" at RMS 0.22, renamed ESC-50, synthetic derivatives); `rahult25/five-class…` — pitch-shifted derivatives of the same material; ICSD — gated on Hugging Face; AxonData — paid; Baby Chillanto — request-only; AudioSet / FSD50K — redundant; raw DonateACry — 8 kHz and skewed, and already merged into cry4.	Contaminated or derivative data does not add information and can teach the model the contamination. Gated or paid corpora cannot be re-fetched by the notebook, so the run would not be reproducible.
D-9	**Negatives deliberately include cat meows, speech and snoring.**	The deployed negative is a room, a baby and a television, not silence. A model that has only ever seen silence as a negative will fire on a fan.
D-10	**Label policy: "any cry present" in the 2 s window** (approximately three or more active frames).	The window is the unit the firmware classifies. A soft label (the loudest frame) would disagree with the label the hub's trigger effectively uses.
D-51	**Music joins the training corpus: GTZAN (Kaggle `carlthome/gtzan-genre-collection`, ten genres, 1,000 real recordings) as an all-negative corpus, with random windows per clip and the same 75 / 25 train / cross-corpus split as the other supplementary corpora.**	The false-alarm map of 2026-10-31 found that **no corpus in the training mix contained any music** — not ICD, not ESC-50, not the cry set, not the snoring set — so the model had never been taught to reject a song, and ESC-50 could not be used to measure it either, because ESC-50 has no music in it. A nursery has a radio, a television and a mobile playing something, so music is a first-class confusion class, not an afterthought. Random windows rather than a single centred one, because a thirty-second track is not one sound: the part that plays in the cot matters, not the average of the track. The 25 % held out keeps the model measurable on music it has never seen.
D-52	**`ZipReader` gained `index_by_path()` and `build_corpus` falls back to it.**	GTZAN names every clip `blues.00000.au` … `blues.00099.au`, once per genre — the same file name in ten folders. A stem index keeps only the first of them, so four fifths of the corpus would have vanished silently and the run would have looked normal.

1.3 The signal contract (frozen — D-11)

Nothing in the firmware, the exported tables or the weights may change without changing all of them at once.

```
int16 pcm[32000] (2.0 s @ 16 kHz mono)
  → /32768 → subtract mean → zero-pad 256 front / 512 rear
  → 81 frames, 512-pt symmetric Hann, hop 400
  → |rfft|² (257 bins)
  → 64 Mel bands (librosa slaney, 20 Hz–8 kHz, 497 non-zero weights)
  → 10·log10(max(P, 1e-10)) floored at −80 dB
  → per-window standardisation (CMVN)
  → int8 quantise → TFLM → [P(no_cry), P(cry)]
```

Constant	Value	Constant	Value
SR	16000	N_FFT	512
WIN_SEC	2.0	HOP	400
N_SAMPLES	32000	N_MELS	64
N_FRAMES	81	FMIN/FMAX	20 / 8000
CENTER_PAD	256	TAIL_PAD	512
DB_FLOOR	−80 dB	LOG_EPS	1e-10
CMVN	True	SILENCE_RMS_GATE	0.0015
TARGET_RECALL	0.95	SEED	1337

#	Decision	Why
D-11	**Freeze the signal contract, and export the Mel filterbank from Python into C** (`crynet_tables.h`).	The firmware cannot drift from the model if the filterbank *is* the model's filterbank: the weights are copied, not reimplemented. A hand-written Mel in C would drift silently and only show up as a small accuracy loss on the board.
D-12	**Per-window CMVN, not absolute dB.**	The legacy system used `power_to_db(top_db=80)` — absolute dB with no normalisation — which made the model level-dependent, and the deployed levels are wildly different from the training levels (ICD cries sit at RMS 0.006, the user's INMP441 room sits at 0.002–0.016 and a cry at 0.057–0.082). CMVN makes the model level-invariant, which is what lets one threshold work in both places.
D-13	**64 Mel bands, librosa slaney scale, fmin 20 Hz** (not the legacy 128 HTK bands).	A consequence of D-11: the features are defined by the notebook that trained the model. Changing them changes the meaning of the weights.
D-14	**The DC-removal mean is an exact `int32` sum, and the 100 ms RMS reported by the node is computed with the mean of the whole two-second window removed.**	Two reasons. First, a float accumulator loses counts on 32,000 samples and the residual DC is what the per-window standardisation then amplifies (see I-1). Second, the number the node reports must be *the same quantity* the hub's front end computes for that window — otherwise the hub's 0.03 trigger means nothing.

1.4 The model and its verification

| # | Decision | Why |

|---|---|---|
D-15	**Three plain CNN variants, no attention and no RNN** (`conv_cnn`, `sep_cnn`, `temporal_cnn`), and **every op is asserted to have a TFLite Micro kernel at export time**.	The legacy `TinyCryNet` (channel attention + GRU) needed ONNX and a bespoke 6.0 MB `app0` partition. Constraining the architecture to ops TFLM implements means the model is deployable on the target at all, and the assertion makes that a build-time fact rather than a hope.
D-16	**`conv_cnn` selected**: 122,770 parameters, val AUC **0.9994**, cross-corpus AUC **0.9973**; INT8 artefact **137,384 bytes**, SHA-256 `4aef55d8…126bc11`.	Best validation and cross-corpus AUC of the three, and 45× smaller than the legacy 6.15 MB FP32 model, so it fits a stock partition with room to spare.
D-17	**Calibrated alarm threshold = 0.95**, re-derived from the full run (an earlier subsample's 0.23 no longer applies).	The threshold is a *calibration result*, not a guess: it is the point on the recall curve that holds ≥ 0.95 recall, and the whole recall/FPR curve is reported with it.
D-18	**Verification is automated and numerical at every layer**: C front end vs Python, INT8 vs float32, TFLM kernel check, and the same four probes run *on the chip*.	The legacy repo had a parity script but it was a simulation. Four of the five Stage 2a defects were only findable by running the code on the silicon (I-1 through I-5).
D-19	**The tensor arena is reserved in PSRAM** (262,144 B reserved, 108,328 B used).	The 256 KB arena does not fit in the S3's internal SRAM alongside everything else. PSRAM is slower but the arena is touched linearly. Moving it into internal SRAM is the obvious next optimisation and was not needed for a triggered monitor.

Measured results (authoritative, from the full Kaggle run):

Split	AUC	F1	Recall	Precision
ICD val (fold 4)	**0.9994**	0.9905	0.9844	0.9966
ICD test (fold 5)	**0.9995**	0.9905	—	—
cross-corpus (cry4 + ESC-50 + snoring, never trained on)	**0.9973**	0.9801	—	—

Check	Result
C front end vs Python (real run)	max deviation **6.905×10^{-5}**, mean 2.79×10^{-6}
INT8 vs float32 (11,386 windows, host-measured)	ΔAUC **−0.001373**, decision agreement **99.956 %**, recall @ 0.95 0.9780 vs 0.9782, FPR 0.0036 vs 0.0036
TFLM kernel check	`CONV_2D, FULLY_CONNECTED, MAX_POOL_2D, MEAN, SOFTMAX` — all present
On-device probes (ESP32-S3)	**4/4 PASS** — features match the host exactly on three probes, within 1 LSB on 3 of 5,184 values on the fourth

1.5 Decision logic

#	Decision	Why
D-20	**The trigger is RMS ≥ 0.030, not the legacy 0.18.**	The legacy value was a real bug: soft cries never reached a classifier. 0.030 sits an order of magnitude above the measured quiet room (0.002–0.016) and below a real cry (0.057–0.082), and it is the *same* number on the node and the hub.
D-21	**The trigger requires the level to have been sustained for two telemetry readings** (~200 ms), and the node's own event level is 0.030 as well.	The INMP441's slow DC drift produces one lone reading above the trigger on its own. In the quiet room the old trigger fired on the drift and pulled bursts continuously; the sustained rule stays quiet.
D-22	**The node freezes the window *before* it announces it**, and advertises the window's sequence number and CRC-32 in the telemetry.	The hub triggers on the node's 100 ms RMS but is served a *two-second* window. Before this change the hub was handed the live window, whose average is mostly the gaps between cries (trigger 0.0710 → window 0.0182 → "no cry"). Freezing first also gives the hub something it can *check*: the CRC-32 of the window it reassembles must equal the one the node froze, which is the proof that the audio crossed the link unchanged.
D-23	**A burst is analysed only when every one of its 286 fragments is present.**	A hole in the window is a hole in the audio. The alternative — analysing a partial window — would feed the classifier a signal the model never saw. The link tolerates loss by *recovering* it (D-26), not by pretending it away.
D-24	**The alert is published on state *change*, not per window, and only after the cry has been held for `CRY_HOLD_MS` = 8000 ms.**	A recording of cries has gaps; a per-window alert would flap on and off. One notification when a cot starts crying and one when it stops is what a caregiver needs.
D-25	**A silence gate of 0.0015 sits below the trigger.**	Below the gate the window is silence and no classification is meaningful. Above it, the calibrated 0.95 threshold decides.

1.6 The radio link (node ⇔ hub)

#	Decision	Why
D-26	**The hub may ask the node to re-send the exact fragments that went missing** — a new `RESEND` frame carrying a 36-byte bitmap, one bit per fragment. This is the mechanism that makes the burst complete despite airtime loss.	The hub's radio is deaf whenever it is retuned to a Bluetooth channel, so a burst of 286 frames sent flat out always loses a few of them, scattered through the middle. The link must tolerate that by *recovery*, and a bitmap of the missing fragments is the cheapest way to ask: one 54-byte request covers any scattered pattern, and the node re-sends only what is missing. See I-24 to I-28.
D-27	**The stall detector asks for the missing fragments as soon as the burst stops growing**, and the wait before it declares a stall grows with the number of fragments still missing.	Asking early keeps the latency down; asking proportionally avoids spending a round while the node is still sending a hundred fragments that take a second and a half to arrive.
D-28	**The resend request is sent three times, and the node answers a repeat of the request it is already serving only once.**	ESP-NOW broadcast is never acknowledged: the sender is told the frame left the queue, not that it arrived. A resend request sent at exactly the moment the burst has just gone quiet is at the moment the radio has been busiest. Repeating the request covers a lost one; the node's dedupe stops the repeats turning into two or three redundant sends of a hundred fragments.
D-29	**The node holds an uploaded window for four seconds before freezing the next one.**	The upload is not the end of the conversation about that window: the hub asks for the fragments it missed, and only the node can send them again. The node used to freeze the next window about three seconds after the upload, and the hub's first request for the missing fragments often arrived just after that — by which time the node had thrown the old window away and refused the request.
D-30	**The hub abandons a burst the node can no longer complete** — if the node's telemetry reports a newer window sequence while a burst is in flight, the burst is dropped and the newer window is asked for at once.	The node no longer holds the fragments of the old window, so the burst cannot be completed and waiting out the 15-second timeout only delays the next attempt.
D-31	**ESP-NOW channel 6, set explicitly on both sides with `esp_wifi_set_channel()`** — not `WiFi.channel()`.	The Arduino core has no channel *setter*: `WiFi.channel(x)` is the scan result's channel (`WiFiScanClass::channel(uint8_t i)`), so the call compiled, returned an int, and changed nothing. The node sat on channel 1 while the hub sat on 6 and the sends failed with "peer

channel is not equal to the home channel". ESP-NOW does not roam. The channel is also set *after* `esp_now_init()`, because starting the driver puts the interface back on channel 1. |
D-32	**Broadcast at 10 Hz; unicast was tried and rolled back.**	Unicast to the learned hub address wedged the ESP-NOW send path: the frame was accepted, never transmitted, the send callback never fired, and every send burned its wait cap until the loop starved. Broadcast is reliable now that the node's loop runs in real time, and it disproved the earlier "the hub radio sleeps" theory — which is why the 100 ms heartbeat was removed (`WIFI_PS_NONE` plus constant node traffic is why the hub hears everything).
D-33	**The node waits for the previous frame's send callback before queueing the next one**, and paces the fragments about 13 ms apart.	ESP-NOW's transmit queue is ten frames deep; a 286-fragment burst sent back to back overflows it and every frame past the tenth is lost for good. The pacing spreads the burst over about four seconds, which the hub's timeout has plenty of room for.
D-34	**Promiscuous Wi-Fi mode is off by default** (build with `-DCRYNET_LINK_PROMISCUOUS=1` to bring it back).	It hands the host *every* frame on the channel. With the laptop's own Wi-Fi on channel 1 that flooded the receive path and cost the hub about fifteen fragments out of every burst.
D-35	**Every frame carries a CRC-16/CCITT-FALSE; the frozen window additionally carries a CRC-32 (the IEEE 802.3 one, the same function `zlib.crc32` computes).**	The CRC-16 covers each frame's header and payload; the CRC-32 covers the whole two-second window and is what proves the reassembled audio is byte-for-byte what the node froze. The wire format was validated numerically *before* any hardware was involved: CRC-16 reproduces the standard check vector `0x29B1`, and the C frame builder and the Python parser agree on every header offset and both CRCs.
D-36	**A fragment belonging to a window the hub did not ask for is dropped, not counted.**	The radio delivers frames in the order it wants to; one straggler from the last burst must not wipe the burst being assembled now.
D-37	**The node's serial console can never cost the node loop time** (`Serial.setTxTimeoutMs(0)`).	See I-12 — this was the root cause of a whole week of "the microphone is dead" symptoms.
D-38	**The node's radio is set up on channel 6 and the peer is added from `loop()`, never from a receive callback.**	`esp_now_add_peer()` waits for the Wi-Fi task to service it. Called from inside that task's own callback it blocks the very task it is waiting for, and the radio stops delivering frames until something else nudges it loose — the hub was receiving one frame in sixty. The hub's own `node_link.c` documents this trap.

1.7 The hub system: memory, Bluetooth, display

#	Decision	Why
D-39	**Boot-only buffers live in PSRAM and are freed after the probes run.**	154 KB of the hub's static RAM (the probe PCM, the Mel and quantise buffers, and a 64 KB CRC buffer) was needed only while the boot probes ran, but as statics it held internal RAM for the life of the program — which is exactly the RAM NimBLE's store configuration wanted. Moving them to PSRAM took the hub's RAM from **59.9 % to 12.9 %** (196,140 B → 42,236 B) and made the whole stack fit: free internal heap 298,763 B at boot → 290,791 B after the probes → **215,155 B with the radio up → 149,271 B with Bluetooth up**.
D-40	**Init order: radio → Bluetooth → display.**	NimBLE's pools and the Wi-Fi driver's buffers come out of the same internal RAM, and when Bluetooth claimed its memory first the Wi-Fi driver ran out mid-init (`esf_buf_setup_static: alloc eb fail(1)`) and the ESP-NOW link never started. The radio takes what it needs while the heap is whole; Bluetooth lives on what is left. The panel is optional — a dead display must not stop the monitoring.
D-41	**`CONFIG_ESP_COEX_SW_COEXIST_ENABLE=y`** so the two stacks share airtime.	The one radio has to carry both. Coexistence is the software arbitrator; without it the two stacks fight.
D-42	**The BLE server serves the legacy UUIDs**, so the wearable firmware written against them keeps working: service `19b10000-e8f2-537e-4f6c-d104768a1214`, alert characteristic `19b10001-…` (read/notify/indicate; the legacy six-byte packet — cot id, alert type 1 = cry / 2 = cancel, confidence %, event id), plus a **new** state characteristic `19b10002-…` carrying the whole system (`cots`, `crying_mask`, loudest confidence, how long the cry has run).	Reusing the UUIDs means the wearable is a drop-in. The extra state characteristic is for a phone that opens late: it still sees the current state without waiting for the next change.
D-43	**The display is a state snapshot behind a spinlock, redrawn by a low-priority task pinned to core 1**, and each line is erased to the right edge so a shorter string cannot leave the tail of a longer one on the panel.	A 30 ms SPI burst must never delay the link loop or the inference. The level bar is sqrt-scaled so a quiet room still shows a sliver and the 0.030 trigger lands about a third of the way up.
D-44	**The Wi-Fi receive buffers were raised to 32 static / 64 dynamic** once the RAM was free.	The node sends a 286-fragment burst as fast as the radio allows; the hub needs receive buffers that can hold one.
D-51	**The 5x7 font table is indexed by the raw ASCII code — `font[c * 5 + i]`, the way `Adafruit_GFX::drawChar` reads it — and `research/build_font.py` now *detects* that base instead of assuming it.**	Adafruit's table is 1,280 bytes, and the first 32 entries are decorative icons (hearts and diamonds): the printable set starts at ASCII 0x20, so a character lives at byte `code * 5`, **not** `(code - 0x20) * 5`. The generator now finds the classic exclamation mark (`00 00 5F 00 00`) in the table, derives the base from where it lands, and refuses to write the header unless it finds it in exactly one place — so a font with a different layout fails loudly instead of producing a screen of nonsense (I-33).
D-52	**The panel shows the caregiver's device: a `PHONE` pill — green when a client is connected and subscribed, amber when the Bluetooth stack is up but nobody has attached, dim when Bluetooth is down — and a confirmed cry with nobody attached reads `ALERT NOT DELIVERED`.**	The user asked for the old system's arrangement, "the status of all nodes and the wearable". The wearable and the caregiver's phone connect to the *same* legacy GATT service, so the hub cannot tell them apart and does not pretend to: the pill says that a caregiver device is attached. A cry that reaches a hub with no listener has gone nowhere, and the caregiver has to be able to see that on the panel instead of discovering it by waiting for an alert that never comes.
D-53	**The framebuffer is RGB565 and the panel is told so: `MADCTL_BGR` is not set, and the inversion state is pinned explicitly with `INVOFF`.**	The ILI9341's `BGR` bit in `MADCTL` tells the controller that the *top* bits of each 16-bit pixel are blue. The framebuffer puts **red** there (`DISPLAY_RGB(r,g,b)` is `(r & 0xF8) << 8 | (g & 0xFC) << 3 | (b >> 3)` — standard RGB565), so leaving `BGR` set made every colour arrive with red and blue exchanged (I-35). The bit was inherited from Adafruit's rotation-2 setting, which this panel's legacy firmware used; whatever pairing that library keeps internally, **this** driver writes plain RGB565, and the measurement in I-35 says the panel was being told the other order. `INVOFF` (0x20) is now sent explicitly even though the chip's reset default is already "inversion off": the picture should not depend on what the last user of this glass left in the controller, and one explicit command removes the variable from every future photograph.
D-54	**The footer counts the cots that are answering, not the cards on the screen.** The count came from `st->node_count` - the number of cot slots the hub has - so a card for a cot that had gone quiet was still counted as a linked cot, and the panel read '3 cots linked' above two cards that both said NO SIGNAL.	The panel is the one surface the caregiver is certain to look, and a number on it that contradicts the cards directly above it teaches the caregiver not to trust the screen. A cot that has gone quiet is a cot that is not linked, whatever slot it occupies. The footer now counts ONLINE cards and says 'no cots linked' when none are answering.
D-55	**The backlight is a brightness, not an on/off switch: the pin is a LEDC PWM output at 5 kHz, on at 55 % by default (`hub_display_set_brightness()`), and the BOOT button opens a *colour check* - a grid of the palette's swatches.**	The second photograph of the panel showed the R/B swap fixed (the `PHONE` square is warm orange where it had been blue, the accent underline teal-green where it had been pink) but the whole screen came out saturated blue. The panel was not doing that on its own: the earlier photograph, taken in a dark room and properly exposed, shows a near-black background rendering as a light pink-grey, and the new one is a close-up whose highlights are **clipped** (the `RADIO` square measures (144,255,255) - fully blown out). A photograph of a

glowing panel in a dark room is not a colour measurement: the camera opens up for the screen, the highlights clip and the white balance drifts with what it sees. So the fault was on the panel's side of the wire, and the lever there is the backlight - a TFT at 100 % in a dark room glares, and a glare is what makes a colour look like the wrong colour. 55 % keeps the dark navy background dark and lets the teal read as teal. The **colour check** is the instrument: with a photograph of the swatches, the panel's own colours can be read off directly - black and white at the ends of the first row, the accent in the middle, the alert colour last, where a red that is really a swapped blue cannot hide - so the palette can be *compensated* if the panel's own black level really is lifted. |
166 | **D-56** | **The neutrals are darker and teal-tinted: the background is `(7,15,16)`, not `(12,20,32)`, and the surfaces, border and dim text follow.** | The colour-check page (D-55) measured this panel's own behaviour for the first time. Its hues are all correct - black is black, white is white, teal is teal, amber is amber, rose is rose - so the R/B swap is behind us (I-35). But the **dark end is lifted**: the old navy `(12,20,32)` came back as a saturated blue while the `(0,0,0)` black beside it stayed black, and the lift is worst in the blue channel (the amber swatch's requested blue of 36 measured back at 145). So the rule this panel obeys is **the less blue the screen asks for, the less blue it gets**. The neutrals were changed accordingly, and they were changed to *teal*-tinted rather than merely dark, because the user asked for the screen to read as teal: the theme now says teal in its darks as well as in its accent. What the screen *says* is unchanged - teal for the system, green for a cot that is fine, amber for a loud one, rose for a confirmed cry - and every contrast was re-checked afterwards (text 16.6:1 on the background, dim 6.1:1, accent 10.4:1, card border 3.4:1). |
167 | **D-57** | **The level of the teal was picked off the panel itself: shown the dark ramp, the user chose its third step, `(4,8,8)`, as the teal the screen should be - one step darker than the background it replaced - and the whole neutral family moved down one step of the same ramp with it.** | The fourth step `(7,15,16)` was the background this replaces, and it still read as bright blue on the glass. A photograph cannot measure absolute colour (I-38), but a ramp step is a colour the firmware *asked for* - so a choice made on the ramp is exact no matter what the camera does, which is what makes the ramp the right instrument for this decision. The surfaces follow the background one step apart so the cards stay a shade above it: background `(4,8,8)`, surface `(7,15,16)`, surface-hi `(12,26,28)`. The border and the dim text keep their levels: contrast only improves as the background darkens, and the re-check confirms it (text 16.5:1 on a card, dim 6.3:1, accent 8.5:1, alert 5.8:1, card border 3.72:1 - every threshold it has to clear it clears with room to spare). |
168 | **D-58** | **The retrained model — the second full Kaggle run, with the GTZAN music corpus trained in — is deployed to the hub and verified 4/4 on the board.** Same architecture (122,770 params, cross-AUC 0.9987), new weights, and a recalibrated input: `IN_SCALE` 0.060125612 → 0.116329655, `IN_ZP` −4 → −70. Threshold 0.95, the silence gate, the output calibration and the geometry are unchanged. | The 8.19 % music false-alarm rate (O-8) was the project's sharpest measured failure and it was structural — music had never been in the training corpus — so the training round was the only fix, and it was measured before deployment on the repo's own evals: **music alarms 79/965 = 8.19 % → 2/965 = 0.21 %**; ESC-50 non-cry alarms 38/1439 = 2.64 % → 17/1439 = 1.18 %; ESC-50 `crying_baby` recall **36/40 → 36/40, identical**; quiet room 0/236 → 0/236, identical. The cost is honest and accepted: per-window INT8 recall on the phone-speaker cry sessions dropped (session 1: 0.867 → 0.800; session 4: 0.724 → 0.592, while that session's AUC *rose* 0.928 → 0.941). A cry is sustained: at 0.592 per-window recall an eight-window cry episode is missed entirely with probability $(1-0.592)^8 \approx 0.0006$ if windows were independent, so episodes still trip the alert. And the phone-speaker recordings are the weakest cry proxy in the building (O-2). On-device: `c_parity.json` max abs diff 6.9e-5; after regenerating the probe references (I-41) the boot self-check reads **RESULT: PASS — 4 probes, 5184/5184 features exact on every probe**, inference 102 ms per window, the noise probe alerts at P(cry) 0.9805, the radio is clean (339 frames, 0 bad). The old model is backed up at `research/out/deployed_model_backup/`. |
169 |
170 | **D-59** | **The second look from the chair: the background dropped another step of the same dark ramp — `(4,8,8)` → `(2,4,4)` — the header strip got its own green-biased colour `(7,20,15)`, a live cot's outline is now the full-strength vibrant teal (rose, when it cries), and a no-signal card recedes.** | One photograph of the new palette said four things. The cards and the ground read as the same colour: the surface was one ramp step above the background, and this panel lifts the dark end until a two-step gap reads as one — so the background went down another step of the same ramp and the cards keep `(7,15,16)`. The header photographed blue while the cards photographed teal even though the firmware paints both `C_SURFACE`: the backlight's LEDs run along the top edge of the glass, and that strip lifts blue hardest — the same panel behaviour D-56 measured — so the header asks for five steps of green *above* blue, which the lift spends on its way in. And the online/offline treatments were backwards in effect, exactly as the user saw it: the live cards' outline was a 45 % blend toward the border while the no-signal cards carried the plain border, so the dead cards read as the livelier ones; the user asked for the swap. Live cards now get `C_ACCENT` at full strength (rose when crying); dead cards get the border washed towards the background — and their empty level bars too, because an empty bar in a full-strength frame also read as "active". Contrast re-checked (text 16.47:1 on a card, dim 6.46:1 on the darker ground, accent 8.47:1, alert 5.84:1, card border 3.72:1); built and flashed; the boot self-check reads PASS, 4 probes, 5184/5184 exact, and the radio is clean. **Amended within the hour, on the user's second look:** the *fills* were swapped by hand - the accent wash read faint and the border wash read as the denser, more active surface, the opposite of what the accent wash was meant to say - so a live card now wears the border-washed fill and a dead card the faint accent wash (the outlines, fixed in the same round, stay); and the dead card's word became **OFFLINE**, the network term for the state the card is in, at the user's suggestion. |
171 | **D-60** | **The in-room confusion session is recorded (O-9 closed), and the honest in-room number is **0 false alarms in 308 windows of non-cry audio**, **86.4 % per-window recall on a cry at 15 cm** - and the limit on how far the hub can hear is the **trigger**, not the model. | Nine one-minute takes with `node_capture` on the real INMP441: quiet room, speech at desk distance, speech at 30 cm, music at 1 m, music at 30 cm, a film at 1 m, a film at 30 cm, the phone-played cry at 15 cm, the phone-played cry at 1 m. **The model raised no alarm on a single non-cry window: 308 windows of speech, music and film, 0 false alarms** - including the two loud takes, where the trigger fired on 269/590 and 493/590 records, so the model was asked 44 times in each and refused every time. On the retrained model this is the room-level version of D-58's GTZAN result (8.19 % -> 0.21 %): on the loudest music this room can produce, 0/44. The cry at 15 cm - the realistic case, the mic is *in* the cot - is caught in 38 of 44 windows (86.4 %). **The cry at 1 m reaches a median RMS of 0.0070 and 0 of 590 records cross the 0.030 trigger, so it is never offered to the model at all** (asked anyway, the model would have caught 18 of 44): the system's reach is set by software, not by the microphone: the INMP441 has **no gain setting** - its L/R pin is a left/right *channel select*, not a gain strap, and the part is fixed at -26 dBFS +/-3 dB sensitivity - so the levers are the hub's trigger threshold and a digital gain in the telemetry path, and the better version of the trigger is an *adaptive* one (a multiple of the node's own running noise floor, which here is 0.0020, so 3x = 0.006 would pass the one-metre phone). Lowering the trigger to ~0.012 is the same lever from the other end and the evidence supports trying it in the next training round, but it is **not** changed on the strength of 308 windows from one afternoon. The labelling tool grew a `--truth` flag for this: the old auto-label called a clip 'cry' if it was *loud*, which for the music and film takes would have labelled the loudest non-cry clips as cries and measured nothing. | Full table, method and caveats: `research/in_domain/SESSION_REPORT.md`. |
172 | **D-61** | **The burst is 2.06 s instead of 5.5-6.9 s, and the panel now shows a cry about two and a half seconds after it happened instead of eight.** The cause of the slow burst was the hub's own Bluetooth advertising: the ESP32-S3 has one radio and it carries both Bluetooth and Wi-Fi, so every advertisement is a moment when the Wi-Fi receiver cannot hear the fragments of the burst. NimBLE's default advertising interval for an undirected advertisement is 30-60 ms (the `BLE_GAP_ADV_FAST_INTERVAL1_MIN/MAX` constants) and each advertisement goes out on all three BLE channels - about a hundred radio transmissions a second. **Three changes, measured on the real boards with `firmware/tools/measure_burst.py`:** (1) *the Bluetooth stack now advertises only when it must*: not at all while a caregiver's phone is connected (nobody has to find a hub that is already connected, and the notifications travel over the connection) and not at all while a burst is in the air - the link stage tells it with `ble_alert_set_radio_busy()`, and the log shows advertising stopping 1 ms after the window is requested and returning 181 ms after the burst lands; (2) *when it does advertise, it advertises slowly*: 100 ms (`itvl_min`/`itvl_max` = 0x00A0) instead of NimBLE's 30-60 ms; (3) *the node paces the fragments every 4 ms instead of every 10 ms* - the ten-millisecond gap was written when the burst really did lose frames to the Bluetooth deafening, and

once that was fixed the gap was the whole of the burst's length. **The measurements:** before, 5.54 s and 6.93 s with resend rounds; after the Bluetooth policy, 3.76 s and 3.81 s with **no missing fragments and no resends at all** (the same with a phone connected, because then the hub does not advertise either); after the node's pacing, **2.06 s, 2.06 s, 2.11 s**, 286/286 fragments every time. The request-to-verdict latency is now **2.23-3.57 s** (mean 2.68 s), of which 0.16 s is the model's own inference. The remaining floor is the node's radio: at 4 ms the transmit queue sometimes backs up and the burst stretches to 3.40 s, which the hub repairs with one resend round. **And a finding that needs the user: played from the *laptop's* speakers the cry is not detected** - the model answers P(cry) 0.38-0.95, just under the 0.95 threshold, and raising the laptop's volume to 100 % made it *worse* (P(cry) 0.38), while the node's RMS stayed at 0.045. So it is not the level: the laptop's speakers produce something the model does not read as a cry, and that is very likely the speakers' own frequency response (a small laptop speaker rolls off the low end, and a baby cry is low-end heavy). The demo's source is the *phone*, whose audio the model detects in 38 of 44 windows offline (D-60), and **the phone path on the live hub is verified in D-62**. | `firmware/hub_infer/components/ble_alert/ble_alert.c`, `firmware/hub_infer/components/ble_alert/include/ble_alert.h`, `firmware/hub_infer/src/link_stage.c`, `firmware/node/node_crynet/node_crynet.ino`, `firmware/tools/measure_burst.py` (new), `firmware/tools/measure_latency.py` (new) |
173	**D-62**	**The demo's sound path is verified on the live boards: a cry played from a phone at the node's microphone is called a cry by the hub, and the whole chain runs in 2.2-5.4 s.** The user played the cry from a phone 10-15 cm from the INMP441 while the hub's log was read live (`firmware/tools/listen_hub.py`, new). **42 verdicts over a four-minute listen: 26 "ALERT - cry" at P(cry) 0.9922-0.9961, and 16 "no cry" at P(cry) 0.0078-0.2148.** The model's INT8 output falls into two clean clusters with **nothing anywhere near the 0.95 threshold**, so not one verdict was a coin toss. The link was perfect throughout: **45 bursts, 45/45 with all 286 fragments, and not one stall, timeout, CRC error or resend round.** The request-to-verdict latency was min 2.23 s, median 3.72 s, max 5.35 s (mean 4.03 s), of which 0.15 s is the front end and the inference together. **The burst's length is no longer the fixed 2.06 s of D-61 but a range - min 2.06 s, median 3.55 s, max 5.19 s** - the node paces the fragments at 4 ms, but the hub and the node share WiFi channel 6, and the per-fragment transmit cost is the *channel's* contention rather than the pacing: with the phone's WiFi on the same band the air is busier and each frame waits longer, so the burst stretches. That is the same channel-contention term that made the burst 5.5-6.9 s before D-61, now visible as a range instead of a constant. **The 16 "no cry" verdicts are the architecture showing itself rather than a classifier failure:** the node freezes the *loudest* two seconds it has heard since the last freeze and the hub judges that one window, so a window that contains the cry but is not the loudest is never offered to the model at all. *(Amended 2026-11-01, late: the first half of that sentence was wrong, and the firmware's own comment repeated the error - **the node does not freeze the loudest two seconds**, it freezes the two seconds *centred on the first crossing of the level*: the ring holds two seconds, the crossing arms a pending peak, and one second later the ring is copied as it stands, which is `[crossing - 1 s, crossing + 1 s]`. The comment in `link_stage.c` now says so. The reading of the sixteen no-cry verdicts does not change - a window that is not the loudest is still the only one that is judged - but the window that is judged is the one *around the crossing*, and for a cry that is only beginning, that window is the weakest two seconds of it rather than the strongest.)* The no-cry verdicts fall in blocks (sequences 6, 8, 9, 16, 17, 22) and the cry verdicts in blocks (7, 10-15, 18-20) - which is what a loudest-window rule does when the loudest thing in the room is not always the cry. The demo must use the phone (D-61's laptop-speaker finding stands), and the demo's own claim is the one that matters: **a cry at the cot is called a cry.**	`firmware/tools/listen_hub.py` (new), `research/live_phone_cry_2026-11-01.log`
174			
175	**D-63**	**The trigger is adaptive, and the *node* owns the floor and the level: it tracks its own noise floor, freezes its window at three times that floor, and tells the hub the level in its telemetry. The hub triggers on the number the node reports.**	The reach of the system was set by a constant, not by the classifier: the cry a metre from the node has a median RMS of 0.0070 and not one of its 590 readings reached the fixed 0.030, so it was never offered to the model at all (D-60). **The multiple was chosen from the recorded sessions, not by taste** (`research/adaptive_trigger_study.py`, which simulates the exact rule over each session's 590 readings): at three times the floor the one-metre cry goes from 0 trigger events a minute to 39, the cry at 15 cm gives 35, the quiet room gives 5, and the loud sessions are capped by the old 0.030 - so the rule is never worse than before in a loud room. At four times the floor the one-metre cry drops to 20 and at five to 7, so three is where the reach is bought without the trigger running away. **Why the node owns it:** the node is the one that freezes the window, so the level that matters is the one it used; a hub-side trigger would be a second guess about the same audio. **The floor falls at once when the room gets quieter and rises with a 50 s time constant** (0.002 of the way per 100 ms reading), so a cry that lasts a minute cannot drag the floor up to its own level and switch the trigger off in the middle of the episode. **The floor is tracked as a minimum statistic**, which is the standard noise-floor estimator and the only one that keeps the one-metre cry reachable: in that session the floor sits near 0.003 while the cry sits at 0.007. **The level travels in the telemetry** (payload 7, milli-RMS) beside the frozen window's own level and the floor, and the hub falls back to `CRYNET_TRIGGER_RMS` for a node that reports none. The node also now arms its freeze on **two** consecutive readings at or above the level - the rule the hub already used and the one the study assumed - so a single drifting reading cannot freeze a window. **Measured live:** the level tracks the room (0.006-0.016 in a noisy room), the node's own log reads `rms 0.0063 floor 0.0028 level 0.0083`, and the hub asks 5-11 times a minute instead of 1-2.	`firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/components/crynet_proto/{include/crynet_proto.h,crynet_proto.c}`, `firmware/hub_infer/components/node_link/{include/node_link.h,node_link.c}`, `firmware/hub_infer/src/{link_stage.c,link_stage.h}`, `research/adaptive_trigger_study.py`
176	**D-64**	**A cry is announced when one window is unmistakable (P(cry) >= 0.99), or when two windows in a row say cry (>= 0.95). A window between 0.95 and 0.99 is a *maybe* and is not announced until the next window agrees.**	**The model is not a reliable judge of a near-silent window.** Measured on the real boards, the same quiet room produced verdicts of 0.0000 and of 0.9766 for windows that differ by nothing a person could hear, and the second of those was a false alarm; across the watches the marginal verdicts came out at 0.6172, 0.7188, 0.7852, 0.8398, 0.8672, 0.8906 and 0.9766. This is the cost of D-63: a lower trigger asks the model about windows that are very nearly silent, and near silence is where the model is unsure. **The rule uses two facts:** the model's INT8 output **saturates at 0.9961 for a clear cry** (D-58) - so a window at 0.99 or above has nothing left to be uncertain about and is announced at once, which keeps D-61's headline (the panel shows the cry about two and a half seconds after it happened) - and **a cry lasts for seconds, so its windows agree with each other**, while a noise fluctuation that happens to look like a cry is not repeated in the next window. The same window is judged more than once when the hub asks for it again, and it always gives the same answer, so it is counted once: the run counts distinct window sequences. **Measured after:** five minutes of a noisy room, 32 verdicts, **0 false alerts** - where before, one window in eight or so of the near-silent windows was a false alarm.	`firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/src/link_stage.h`
177	**D-65**	**The node's transmitter runs at 8.5 dBm instead of the default 20 dBm.**	The Wi-Fi antenna and the INMP441 are centimetres apart on the same board, and the radio's own transmission reaches the microphone. The evidence: the windows the node froze with the radio running carried roughly twice the energy of the same room's windows recorded with the radio off (`node_capture`, which has no radio at all, and the same I2S setup), and the extra energy had the shape the model reads as a cry - the false alarms of D-64's opening. With the power at 8.5 dBm - 11 dB less into the room - the marginal verdicts fell from 0.89 to 0.72 and the count of near-silent windows above 0.5 fell from 3 in 12 to 1 in 9. Eight and a half dBm is ample for a link of a couple of metres, which is what a cot node is, and **on a battery node it is also simply the right thing to be doing**: the radio is the dominant power draw.	`firmware/node/node_crynet/node_crynet.ino`
178	**D-66**	**The hub no longer asks for a frozen window whose sequence it has already received and judged.**	The first verification session (phone at 15 cm, then at 1 m) exposed a latency regression: the first alert at 15 cm took ~21 s instead of the ~2.5 s of D-62. **The cause:** the hub re-requested the *same* frozen window up to four times in a row (seq 100 was asked for four times), and each serve reset the node's 4 s hold timer (`g_frozen_served = true; g_frozen_hold_ms = millis() + FROZEN_HOLD_MS`), so the node could not freeze the new loud window until the hub stopped asking - the new cry sat unfrozen for ~15 s. **The re-request never helped anything:** re-judging the same window gives the same answer, and the run rule (D-64) counts distinct window sequences,

which a repeat never advances - `st->cry_seq != st->request_seq` is false on a repeat. **The fix:** the hub records the seq of every window it actually receives and judges (`last_completed_seq`), set in the burst-complete branch, and the request condition now requires `st->seq != st->last_completed_seq`. The hub waits for the node to advertise a new seq. Side effect: the "resend refused: no frozen window for it" fragility (four times in the session) disappears, because the node is no longer asked to re-serve a window whose hold has expired while it has moved on. Expected first-alert latency at 15 cm: ~3 s after the cry starts (1 s peak delay + ~2 s burst). **Verified in the second session: the re-request loop is gone** (no seq was asked for twice), but the latency was still ~27 s - the node still froze a new window every ~4 s while the hub emptied one every ~4 s, so the queue never drained. See D-67. | `firmware/hub_infer/src/link_stage.c` |
D-67	**The node freezes at most one window every eight seconds, not every four.**	D-66 removed the re-request loop but the second verification session still showed a **~27 s delay** to the first cry alert. The node was still freezing a new window every ~4 s (the hold was 4 s, and during a continuous cry the loud streak never drops, so the node freezes the moment the hold expires), while the hub processes one window every ~4 s. The two rates matched exactly and the queue never drained: the first cry window (seq 17, frozen at 87.7 s) waited behind seqs 13, 14, 15 and 16 and was not judged until 115.4 s. **The fix:** `FROZEN_HOLD_MS` goes from 4000 to 8000. The node now freezes at most one window every 8 s while the hub empties one every 4 s, so the hub stays ahead and the queue drains. Eight seconds still covers the hub's three resend rounds, which all land inside the first seven seconds. **Expected first-alert latency at 15 cm: ~5 s** (1 s peak delay + ~4 s burst) instead of 27 s. **Verified in the third session: the queue is still the bottleneck** (seq 27, frozen at 34.26 s, was not judged until 70.89 s) - see D-68.	`firmware/node/node_crynet/node_crynet.ino`
D-68	**A loud cry preempts the node's hold, and the hub does not wait for a streak to request it.**	D-67 reduced the freeze rate but the third verification session still showed a **~36 s delay**: the cry (peak rms 0.31 at 30-35 s) froze seq 27 at 34.26 s, but the hub did not judge it until 70.89 s because it was still processing seqs 19-26 (quiet windows frozen before and during the cry). **Two changes:** (1) the node, when a crossing at three times the trigger level happens while a previous window is still being served, abandons the hold and freezes the new (louder) window immediately - the hub's abandon branch then asks for the loud window instead of the stale one; (2) the hub's request condition treats a window as loud when the live RMS is at least three times the trigger level, even if the two-reading streak has not built up, saving ~5 s per window. **Combined expected first-alert latency at 15 cm: ~4 s.** Built and flashed; boot self-check passes; **verified in the sixth session: preemption did not help** because telemetry is lost during bursts - the hub did not learn that a loud window had been frozen until after the current burst completed, by which time the node had frozen several more. See D-69.	`firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/src/link_stage.c`
D-69	**Cry mode: the node freezes exactly one window per cry.**	D-68's preemption did not work because telemetry is lost during bursts: the hub does not learn about a new frozen window until after the current burst completes, by which time the node has frozen several more. **The fix:** once a loud window is frozen (win rms >= 5x the floor), the node enters cry mode and does not freeze again until the room goes quiet (rms < level for one reading). A sustained cry now freezes exactly one window, which the hub processes in ~5 s. The 1 m cry (rms ~0.007, 2.3x the floor) does not trigger cry mode, so it is detected normally when the queue is empty. Combined with D-68 (hub bypasses the streak for loud windows), the expected first-alert latency at 15 cm is ~5-8 s. **Built and flashed; not yet verified on the live boards.**	`firmware/node/node_crynet/node_crynet.ino`
D-71	**The frozen-window hold returns to 4 seconds, and the demo's phone must play quieter than the microphone can clip.**	The three-minute correlated session (phone cry ~20 cm from the mic) **detected the cry three times at P(cry)=0.9961** and vindicated the chain end to end: the mic heard RMS 0.26-0.53 (the panel's 0.265 was real - an earlier "live" listener had stopped before the user played and had been measuring silence), the node froze cry-level windows one per cry segment with the cry-mode marker, the link ran 286/286 with CRC MATCH, and there were zero false alarms before and after. **The hold's cost was measured directly: 6.5-7.0 s of hold-driven gap between every burst.** With cry mode capping freezes at one per cry segment, the hold no longer has the queue duty it was raised to 8 s for (D-67) - it is pure resend coverage, and 4 s covers the burst and the first resend round. `FROZEN_HOLD_MS` 8000 -> 4000. **The session's remaining detection problem is clipping:** the phone now plays so loud that the INMP441 pegs (peaks 32694-33100 of 32767), and clipped cries are a lottery for the model - only 3 of 13 loud windows scored >= 0.9961, where session 1's unclipped windows (peaks at 45-75 % of full scale) scored 0.9961 on every one. The demo fix is to play the phone quieter, targeting the panel's level reading of ~0.10-0.20 (session 1's range).	`firmware/node/node_crynet/node_crynet.ino`
D-72	**A near-silent window can neither announce nor confirm a cry - two rms gates close the quiet-room false alarms.**	The user saw **fourteen "cry detected" flashes in a quiet evening, none over 98 %**, with panel levels of 0.002-0.012 and no cry playing. The mechanism: in a quiet room the adaptive trigger pins at its minimum (TRIGGER_MIN 0.005), so the hub judges near-silent window after near-silent window; the model's verdict on those is documented noise (the same room has produced 0.0000 and 0.9766 for windows that differ by nothing audible); and the run rule assumed "a noise fluctuation is not repeated in the next window" - a quiet room repeats it constantly, so two consecutive scattered verdicts at 0.95-0.98 confirmed each other, each held 8 s by the cry hold. **Fix (hub only): `CRYNET_ANNOUNCE_MIN_RMS 0.006` - no window under 0.006 rms may announce at all, even at P(cry) >= 0.99** (it sits under the quietest genuine detection, the one-metre cry at win rms 0.0103, with margin); **`CRYNET_MAYBE_MIN_RMS 0.020` - a maybe (0.95-0.99) only counts toward the confirming run when its window carries real energy** (every measured false maybe sat at <= 0.012; every measured genuine cry announced saturated, and the unmistakable path stays open at >= 0.006 rms). A refused quiet maybe now logs `cry (too quiet to announce)`. **Verified live within 90 s of flashing:** the room produced exactly the failure shape - `window rms 0.003917  P(cry)=0.9531` - and the new gate refused it.	`firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/src/link_stage.h`
D-73	**A quiet window must have a cry's shape, not just its confidence - the crest gate.**	The session after D-72 caught four more quiet-room false alarms in nine minutes (rms 0.0067-0.0088, P(cry) 0.9922-0.9961, peaks 621-789): **the model saturates on pure INMP441 self-noise roughly every two minutes in tonight's very quiet room.** The 0.006 rms gate (D-72) sat under the flukes, and no confidence threshold can sit above saturation - P(cry)=0.9961 is the same value a real cry produces. The discriminator is physical: **crest factor (peak/rms)**. Room hiss is Gaussian - every quiet window the model has ever scored highly on that was not a cry has a crest of 2.2-2.9; the quietest genuine cry (the one-metre phone cry, P=0.9961 at 0.0103 rms) has a crest of 5.6, the 15 cm cries 6.6-6.9. **`CRYNET_QUIET_CREST 3.5`: a window under 0.020 rms announcing on saturation alone must also have crest >= 3.5**; loud windows announce on saturation alone (clipped cries can flatten the crest to 1.9-3.9, so the shape test would be wrong there). A loud bump has a high crest but the model refuses bumps on its own (measured 0.0977), so the gate only does work where the model is least trustworthy. The hub now prints `crest` on every window line. The four session flukes (crest 2.3-2.9) would all be refused.	`firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/src/link_stage.h`
D-74	**The run confirms only a window that itself carries energy - no announcing on the run's momentum.**	The volume-down verification session (play at rms 0.07-0.19, peaks 57-69 % of full scale - no clipping) detected the cry on every loud window (P=0.9922-0.9961, five ALERTs). But 12 s after the last cry window, a quiet hiss window (rms 0.0096, crest 2.5, P=0.9883) printed ALERT. Cause: the cry's own loud windows advanced the confirming run (they say cry and carry energy), reaching run=2; the quiet window then says cry (0.9883 >= 0.95) so it does not reset the run, and the run-of-two confirmed it. With tonight's flake factory (quiet hiss scoring >= 0.95 constantly) a post-cry run could be refreshed by hiss windows indefinitely. **Fix: `confirmed = unmistakable || (cry_run >= 2 && rms >= CRYNET_MAYBE_MIN_RMS)`** - a quiet window may keep the run alive (a real cry has quiet gaps and the next loud window should not start over) but may not announce; the announcement is refreshed only by windows with real sound behind them.	`firmware/hub_infer/src/link_stage.c`

1.8 Toolchain and process

#	Decision	Why
D-45	**PlatformIO 6.2.0 with `platform-espressif32` 8.0.0 and ESP-IDF 6.1.0**, installed by hand rather than pinned by

URL. | PlatformIO 6.2 installs a URL-specified package by treating the archive as an **npm package**: it picks the wrong `package.json` out of the tree, writes a bogus version into `.piopm`, and leaves `packages/framework-espidf` holding only a test-fixture package (I-6). IDF 5.3.2 could not be installed through PlatformIO at all; 6.1.0 was already on disk and satisfies the port's `idf >= 5.1`. |
D-46	**The vendored TFLite Micro port lives inside `firmware/hub_infer/components/`**, and `esp-nn` is wired as the port's own dependency with the registry copy removed.	PlatformIO re-compiles IDF components with SCons using the *source basename* for object names, and the TFLM port has twelve basename collisions, so the build died with "Multiple ways to build the same target". Moving the vendored components inside the project makes the paths relative and distinct. `firmware/tools/patch_deps.py` keeps the build offline and pinned.
D-47	**The flash mode is DIO throughout** — the effective config is `CONFIG_ESPTOOLPY_FLASHMODE "dio"` and both `bootloader.bin` and `firmware.bin` carry flash-mode byte `0x02`.	A combined image built with QIO makes the chip watchdog-reset in a loop (`rst:0x7 TG0WDT_SYS_RST` repeating, no application output). The ROM cannot read this board's flash in QIO before the bootloader has configured its own timing. DIO costs nothing at this workload.
D-48	**The node sketches are compiled with the Arduino IDE's own `arduino-cli` against core 3.3.12**, and the toolchain state is kept in the repo (`dependencies.lock`, `firmware/tools/fetch_deps.sh`, `patch_deps.py`).	The user's machine originally had no ESP-IDF, no PlatformIO, no Arduino IDE and no Xtensa compiler. Everything the build needs is now reproducible from the repo rather than from the state of one machine.
D-49	**Every host tool that reads a serial stream resynchronises on its own magic** and never waits for a fixed block; capture records carry a CRC-16 and the collector drops any record that fails it.	A USB CDC port can enumerate *after* the sketch has already printed its banner, and a single dropped byte on the USB link would otherwise shift every later sample and quietly ruin the dataset. Both stream parsers are testable classes, checked against synthetic streams.
D-50	**The notebook is generated from `research/build_notebook.py` and never hand-edited.**	The notebook is a build artefact. Editing it directly makes it impossible to regenerate and to diff. It is validated with `nbformat.validate()` after every regeneration.

1.9 The two-cot system (this repository)

The decisions below are the ones where this repository deliberately differs from the three-cot sibling repository `fp-2`. Everything above this line (the model, the signal contract, the trigger rules, the radio link) is shared.

#	Decision	Why
D-75	**The two-cot system lives in its own repository** (`fp-two-node`, remote `https://github.com/Amazo123/CryNet-TwoNode.git`), sharing history with `fp-2` up to commit `99bb257` and then diverging.	The two systems are different products with different firmware, different wiring and different documentation, and the owner works on both. Sharing the history keeps the model and the signal contract traceable in one place without letting the two firmware trees drift into each other again - which is exactly what happened to the wearable sketch before the split (the `fp-2` copy was still blinking an LED while the bench was driving a motor). `fp-2` keeps its own remote (`crynet` → `https://github.com/Amazo123/CryNet.git`) and is untouched by this decision.
D-76	**The wearable has no status LEDs; its OLED dashboard carries the state** (LINK or SCAN), and the battery/SOC readout is marked as needing firmware that does not exist yet.	A newer revision of the wiring diagram gave the wearable a blue link LED on GPIO1 and a red SOC LED on GPIO4, which contradicted `fp-2`'s `SYSTEM_WIRING.md` §7 (then titled "SOC display and charge prompt") and its own committed PDF (page 4). Neither LED has firmware: wiring them produces two dark LEDs and a caregiver who reads a dark LED as a fault. The OLED is already on the wrist and already says what the LEDs would have said.
D-77	**The haptic is a real motor on a BC547 low-side switch** (GPIO7 → 2.2 kΩ → base, 100 kΩ base-emitter pulldown, **22 Ω** in the motor's + lead *(revised by D-85)*, 1N5819 with the cathode up, collector → motor −, emitter → GND), **not the DRV2605L breakout that the planning documents still describe.**	The bench hardware is the BC547 circuit the owner drew, and the motor is now driven by the firmware itself (`HAPTIC_MS 2500`, `HAPTIC_PULSE_MS 250`) rather than by an LED stand-in. The DRV2605L description survives in `SYSTEM_WIRING.md` §2.3 and contradicts §5.3 in the same file; the firmware and the bench win, and the documents are what need correcting.
D-78	**The hub's TFT backlight stays on 3V3 and is permanently on; there is no sleep mode.**	The hub's firmware already treats GPIO4 as the backlight (`PIN_BL 4`, `ili9341_backlight_set` with LEDC PWM), so a 2N7002 on GPIO4 would make the panel dimmable and switchable. The owner chose against it: the panel's LED pin is tied to the panel's own 3.3 V net, the glass cannot be blanked while the board is powered, and the system is always plugged in or charging. The MOSFET was removed from the wiring diagram **before anything was soldered**, so nothing had to be unbuilt. GPIO4 is therefore free.
D-79	**The node's I2S zero watchdog is committed but not flashed; it goes onto a node after the soldering is finished.**	The watchdog exists because node 2 was once found reading rms 0.0000 on every mic read - the INMP441 delivering nothing - and the fault cleared on a hard reset with no reflash, so its cause was never established. A fault-injection test proved the code executes and refills a window; it did not prove recovery from the original physical fault, and it detects *exact* zeros only, so a mic that goes quiet-but-not-zero is still not covered. The owner is soldering now, and a node should not be reflashed mid-solder, so the watchdog waits until the hardware is settled and can then be flashed and played-cry tested on the bench.
D-80	**SUPERSEDED BY D-86: the two-cot case has now been tested, and the alert face is no longer parked.** ~~**The multi-cot wearable behaviour is held on its own branch** (`feature/wearable-multi-cot`), not on `main`.~~	Two cots crying at the same time had never been tested when this was written, so the multi-detection alert face was parked until that test could be run. **The test was run on 2026-11-20** on this bench's two Pro Mini cots and the wrist: the hub's crying mask sat at `0x03` for fifteen consecutive seconds, and the owner watched the alert face read the two-cot label. D-86 records the result and the change. The parked branch `feature/wearable-multi-cot` is an **earlier and different** attempt — it has no `COTS` label and it predates the current tree — so it is not the implementation D-86 shipped, and it should not be merged.
D-81	**SUPERSEDED IN PART BY D-83: the divider half of this decision was withdrawn before it was ever fitted. The cell half stands.** ~~**The battery sense is a fixed 220 kΩ / 100 kΩ divider (ratio ×3.2) — there is no variable resistor in the battery path — and SOC is read from a voltage curve, not a linear map.**~~ **The cell is an 18650 for the nodes and the hub and a 1000 mAh Li-Po pouch for the wearable, and LiFePO4 cells are excluded from the whole system.**	The owner was advised to put a *variable resistor* between the cell and the ADC pin, read the maximum (4.2 V) as 100 % and the minimum (3.0 V) as 0 %. That is wrong in three ways. **(1) A two-terminal pot — a rheostat between the cell and the pin — cannot divide at all.** The ADC pin's only path to ground is its own high-impedance input, so turning the pot down barely moves the voltage: the pin still sits near the cell's 4.2 V, past the C3 IO's 3.6 V absolute maximum. Dividing needs a second leg to ground, and that is what the fixed pair provides. **(2) A three-terminal pot used as a genuine divider — one end to B+, one end to B−, the wiper to the ADC — does work, and the firmware does *not* have to read the wiper**: the ratio can be measured once at assembly and handed to the firmware as a constant. *An earlier version of this decision said the chip 'would not know the division ratio'. That was overstated and is corrected here.* What the pot costs instead is **(a) per-unit calibration** — the ratio is a build-time measurement, not a datasheet value, so every unit needs a meter reading and its own constant; **(b) drift** — the wiper is a mechanical setting that moves with temperature, vibration, wear and the knocks a cot gives the node, and the failure is **silent**, because the reading keeps looking plausible while the ratio underneath it has moved; and **(c) magnitude** — `BAT_CURVE` moves 100 mV = 20 percentage points around 3.8 V, so **1 % of ratio error is already about 10 points of SOC error**. A pot's tolerance-plus-drift is 1–5 % and unrepeatable from unit to unit; the fixed 1 % pair is ±1 %

on the ratio, known at the design stage, on every unit, with no per-unit assembly step. `analogReadMilliVolts()` already applies the chip's factory-calibrated ADC reference, so the residual error is a few per cent — which the existing banding absorbs. **(3) The 3.0–4.2 V linear map is chemically wrong:** a lithium cell is roughly 4.20 V = 100 %, 3.85 ≈ 55 %, 3.70 ≈ 25 %, 3.60 ≈ 10 %, 3.40 ≈ 0 %, so a linear map would call 3.70 V "58 %" when the cell is really about a quarter full — and it errs optimistic across exactly the range the cell spends most of its life. `hub_display.c` already holds this as a table (`BAT_CURVE`) and prints it as `BAT ~80 %` with a tilde. The bottom is anchored at ~3.3 V rather than 3.0 V, because 3.0 V is roughly where a protected cell's protection circuit cuts out — that is end-of-life, not "empty". On the **cell format**, the two units have opposite requirements: a node sits on a cot frame within a baby's reach, so it gets an 18650 in a holder — more energy per unit volume, and a steel can that cannot be punctured — while the wearable goes on a wrist, where an 18650's 18 × 65 mm and 48 g are impossible and a flat pouch cell is the only sensible form. The pouch is 1000 mAh rather than the original 500 mAh, and the module's ~1 A charger must be reduced to ~400–500 mA for it, since 1 A into 1000 mAh is 1 C. **LiFePO4 is excluded everywhere** because the module charges to 4.2 V and an LFP cell tops out at 3.65 V — it would be destroyed. If a true SOC percentage is ever wanted instead of an estimate, the part for it is a **MAX17048** fuel gauge on I²C (the hub and wearable have a spare bus; the node's microphone occupies I²S, so nodes keep the divider). **One measured-and-recorded caveat on the divider itself (O-1e):** the pair's source impedance is **68.75 kΩ** (220 kΩ ‖ 100 kΩ), which is above the ~10 kΩ an ESP32 ADC likes; the **100 nF on the sense node is what makes that workable and is required, not optional** — the ADC takes its charge from the capacitor rather than from the divider. |

214 | **D-82** | **The 5 V→3.3 V linear regulator stage in the power chain is recorded as the largest lever on battery runtime, but is not changed now.** | The chain is **cell → boost to 5.1 V → the board's 5 V pin → the board's onboard linear regulator → 3.3 V**. A linear regulator passes *current*, not power, so the cell ends up supplying roughly **1.8× the 3.3 V load current** while the difference is burned as heat; on a 100 mA load that is about 160 mA from the cell, which projects to roughly six hours for the wearable on a 1000 mAh pouch and rather less than a day for a node on a 3000 mAh 18650. *These are projections from the regulator's behaviour, not measurements — no runtime has been measured on this system.* Feeding the board's **3V3 pin** from a 3.3 V buck-boost would bring the cell current close to the load current and roughly double the runtime. That is a change to the power architecture, and the owner is mid-solder on the design as drawn, so it is deferred to a bench evaluation rather than applied to the wiring now. The other lever is the wearable's OLED (~25 mA), which is always on. |

215 | **D-83** | **The battery SOC feature and every status LED are removed from the system: no battery-sense divider, no cell-voltage telemetry, no battery text on the hub's panel, no link LED on the node, and no BLE-status or SOC LED on the hub.** The battery *power* is untouched — the cells, the LX-LCBST modules, the switches and the charge currents are exactly as D-81 settled them. | The owner asked for it directly: the hub's panel already shows which cots are answering, so a second copy of that answer on the cot itself is redundant, and the battery figure was always an estimate — a resting-voltage curve printed with a tilde — that the caregiver was not going to act on. **The removal is total rather than partial on purpose:** a divider that is still on the bill of materials but never read is a part that gets bought, soldered and trusted, and a LED that is wired but never driven is a dark LED that reads as a fault. **What is given up: nothing in the system will warn the caregiver that a cell is nearly flat.** The cells must be watched by eye and charged before a unit goes back on a cot. **What is kept:** the heartbeat and its ACK, because that is how a node knows the hub is there — only the LED that displayed the answer is gone, and the node prints the same fact on its serial console (`status`) while the hub shows it on the panel. **The wire format changes with it:** the telemetry's battery tier is removed and the radio counters move down two bytes, from offset 18 to 16. The hub still reads the old 34-byte layout, so a node that has not been reflashed yet does not turn the counters into noise while the units are being reflashed one at a time. The freed pins — GPIO3, GPIO7 and GPIO10 on the node, GPIO1, GPIO2 and GPIO17 on the hub — are free and need no boot-strapping verification. |

216 | **D-84** | **The 5 V supply chain has three rules: set the boost trimpot to 5.1 V before any board is attached, never run the boost and USB into the same board at once, and treat the board's own linear regulator as the thing that gets hot.** | **These are consequences of the chain D-82 describes — cell → boost to 5.1 V → VO+/VO− → switch → the board's 5 V pin → the board's onboard linear regulator → 3.3 V — and none of them was written down until the owner asked whether the boards could be powered through their 5 V pins.** **(1) The trimpot is set before anything is attached.** The LX-LCBST ships with its boost output anywhere in 4.2–28 V, and a board connected before the trimpot is set sees whatever the module happens to be producing. Set it to ≈5.1 V with a meter first, every time the cell or the module is changed. **(2) The boost and USB must never be connected at the same time.** The boost's output goes to the board's 5 V pin, and that pin is tied to the USB 5 V rail through the board's own protection diode, so the two supplies are on one net. With both connected the 5.1 V rail and the host's 5 V rail fight each other and current can be pushed back up the USB cable into the laptop — a fault on the host's side, not only the board's. **Unplug USB before the battery goes on**; when a unit must be watched on its console while running from its cell, use a **USB cable with the 5 V (VBUS) wire cut** — the data lines still carry the serial log and nothing can be pushed back into the host. **(3) The linear regulator is the weak point in the chain and it runs hot.** A linear regulator passes *current*, not power, so the difference between 5.1 V in and 3.3 V out is burned as heat inside the board: **(5.1 − 3.3) V × I**, which is roughly **0.5–0.7 W** at the C3's radio peaks of a few hundred mA, in a package with no heat sink. That is also why the cell supplies about 1.8× the 3.3 V load current (D-82), and it is the first thing to check when a unit browns out or resets as the radio transmits — **not the radio**. Feeding the **3V3 pin** from a 3.3 V buck-boost removes the dissipation and roughly doubles the runtime, but it needs a **separate** 3.3 V module, because the LX-LCBST cannot be set below its 4.2 V minimum. **What does not change:** the OLED and the INMP441 stay on 3.3 V in every arrangement, because the OLED's I²C pull-ups ride on its VCC rail and 5 V there would put 5 V levels into non-5-V-tolerant C3 GPIOs. **Still unmeasured:** none of this has been run on a cell yet — the charge current, the boost's quiescent drain and the switch-off drain are all still unknown (O-1d), and the numbers above are arithmetic from the regulator's behaviour, not bench readings. Evidence: `SYSTEM_WIRING.md` §6 |

217 | **D-85** | **The stall cap fitted in the haptic motor's + lead is 22 Ω, not the 10 Ω the documents named.** (This revises D-77, which named a 10 Ω.) | The owner had a 22 Ω and no 10 Ω, so 22 Ω is what went on the board, and on 2026-11-11 the bench proved the drive works with it: the motor was replaced (the first one was faulty — its coil was open, which showed up as the full 3.296 V appearing across the motor with the GPIO7 pin switching, because no current means no drop across the cap) and the new motor vibrates correctly. **22 Ω is the safer of the two values, as well as the fitted one:** the stall current is capped at 3.3 V ÷ (37 Ω + 22 Ω) ≈ **56 mA** for a 37 Ω coin ERM, against ≈70 mA for a 10 Ω, so it sits further inside the BC547's 100 mA rating. **The cost is on the buzz:** the cap drops about 1.2 V of the 3.3 V rail, so the motor sees about **2.1 V** instead of about 2.6 V and runs a little slower and quieter. That is the trade the owner accepted by using the resistor he had, and it is a better trade than it looks — a lower-impedance motor on a 10 Ω would have needed a 2N2222A. **What changed in the repository:** the BOM row, `SYSTEM_WIRING.md` §5.3, this decision, and page 4 of `SYSTEM_WIRING_DIAGRAMS.pdf` now say 22 Ω, and the stall-current figures in each of them were recomputed (56 mA, not 69 mA). The documents were not merely tidied — a 10 Ω in the BOM is a part the owner would have bought and soldered, and the figure that goes with it was wrong. Evidence: `SYSTEM_WIRING.md` §5.3, `HARDWARE_COMPONENTS.md`, `tools/wiring/page_wearable.py` |

218 | **D-86** | **The wearable's alert face names every cot the hub says is crying, while the alert packet still names one cot.** The change is on the wrist only — the six-byte alert, the hub's alert policy, detection and the buzz rule are untouched. Ported from the sibling repository's verified `c35ead1`. | Two cots crying at once was already detected correctly: the nodes gate independently, the hub confirms each, and the state characteristic's `crying_mask` has carried every crying cot since D-24. The wrist was already receiving that mask and already listing both cots on its **idle** face. The gap was one screen: the **alert** face drew its cot number from the alert packet's single `cot_id` and held the glass for `ALERT_HOLD_MS` = 60 s, so the one face the caregiver looks at during a cry hid the list the wrist already had. The face now builds its label from `crying_mask` **unioned with the named cot** — the alert is instant while the mask is published on change plus a 1 s heartbeat, so the mask alone can lag it or already be clear, and without the union the screen could say `CRY` with nobody named. One cot renders exactly as before (`COT n`, text size 3); both crying render **`COTS 1 2`** at size 2, because that label is 144 px at size 3 on a 128 px glass and 96 px at size 2. The loop still walks all eight mask bits and steps the size down again rather than let `snprintf` cut a cot off the end, because the mask byte is eight bits wide even though this hub only ever sets two. **A deaf cot keeps the mask out** — deafness is one cot by definition. **`g_mask` and `g_cots` are now cleared on disconnect**, which they were not before: the idle face was guarded by `g_connected` but the alert face

outlives a drop by up to a minute, so a stale mask would have named cots nobody is watching. **What this deliberately does not do:** no second buzz (the hub always sends `alert_type` 1 for a cry and the wrist buzzes only on the transition into 1, so a handover between cots stays silent), no change to detection, the serialized loudest-first pulls, the 1.5×/1.5 s handover dwell or BLE/Wi-Fi coexistence, and no new BLE traffic. The alternative — a seventh mask byte on the alert characteristic — was rejected: every consumer checks a *minimum* length so it would not have broken them, but it edits a frozen legacy contract and forces a hub reflash to buy about one second of freshness the wrist already has. The confidence bar remains the loudest cry's figure, the only one the hub sends; it is not per-cot. **Verified on the bench 2026-11-20, on this system's two Pro Mini cots:** mask `0x03` for fifteen consecutive seconds (12:36:41–12:36:55), the owner watched the glass read the two-cot label, and over the whole captured session **nine cry announcements across five episodes produced exactly five buzzes** — the four silent ones were handovers. Three separate two-cot events were captured in all. **In this repository the port is compile-verified only** — `arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3` gives **685,589 B (52 % of flash) / 18,684 B (5 % of RAM)**, the same figures as the build that was flashed and watched, and `check_prebuilt.py` passes — but **no board was attached when it was ported, so this repo's own image has not been flashed** (O-1f). Evidence: `firmware/wearable/wearable_crynet/wearable_crynet.ino`, `SYSTEM_UPDATE.md` 2026-11-20 |

1.10 The record and the dashboard

The system worked, but it kept nothing. A caregiver who was woken four times in a night had no way to see whether that was getting better or worse, or whether one cot's baby cried more than the other's. The owner asked for a dashboard, and asked that the hub record locally always and upload only when it has internet — *"the system should be able to run perfectly offline as it is now"*. That sentence is the constraint the whole design answers to.

#	Decision	Why
D-87	**Every cry episode is recorded in the hub's own flash, whether or not there is a network. The uploader is a separate, best-effort reader of that record.** Three new hub components: `crynet_log` (the record), `crynet_clock` (the time), `crynet_net` (the network). Nothing any of them does is on the path between a cot's microphone and the caregiver's wrist.	The partition table had reserved six megabytes for storage since the beginning (`storage, data, spiffs, 0x210000, 0x600000`) and **nothing had ever mounted it** — the space was there and no code used it. That is why the hub kept nothing. The record is written by the audio loop itself, so a hub with no Wi-Fi, a wrong password, or a dead router still records exactly the same file. Every log call is best-effort: if the flash will not mount, `crynet_log_reserve` returns 0, the episode helpers do nothing, and the wrist is told precisely what it is told today. The uploader never touches the audio path — it reads the finished records afterwards.
D-88	**The router's 2.4 GHz channel must be set to 6, and the hub enforces it.** On associating, the hub reads the radio's channel; if it is not 6 it disconnects at once, puts the radio back on 6, says what to do about it, and retries in five minutes.	The cot nodes and the hub talk ESP-NOW on **channel 6, fixed** (D-13, `node_link.c`). The hub has one radio. Joining an access point moves that radio to the router's channel, and if the router is on 1 or 11 the cot link dies — **the wrist goes silent while the dashboard looks healthy**, which is the worst failure this system can have. Letting the association stand would trade the monitor for the record. The guard makes the record yield to the monitor, every time, and turns a silent failure into a printed instruction. The cost is one setting in the router's admin page.
D-89	**One record per cry *episode*, not per window: 32 bytes, appended once and never changed.** It carries the episode's duration, its peak and mean loudness, the model's peak confidence, and how many windows were judged. The number on the caregiver's wrist ("event #1042") is the same number the dashboard holds.	A baby who cries for a minute produces twenty-odd two-second windows and one episode, and it is the episode a caregiver wants: when it started, how long it went on, how loud. Per-window records would be twenty times the flash wear and would need the dashboard to reassemble episodes the hub already knows the boundaries of. Fixed-size append-only records make a power cut cheap: a half-written tail record is truncated at mount, and the record number is spent when the episode *opens* (so a cut leaves a hole in the numbering, never a reused number). The log holds 8192 records, trims to 2000 after a successful upload, and compacts through a temp file and a rename in 128-record chunks so it never needs a large allocation. `event_id` on the wrist is sixteen bits, so it wraps at 65536 — about three years of nights; the dashboard keeps the whole number.
D-90	**The hub has no battery-backed clock, and the dashboard is honest about that.** Until it reaches a time server the hub records `start: 0`, meaning *we know this happened, not when*. Those records are counted in totals and never placed on a time axis. The last known time is kept in the log's own meta file and restored at boot, then corrected by SNTP.	An RTC cell is a part the owner would have to buy and wire, and D-83 removed every optional part that was not doing a job. The honest alternative costs nothing: a record with no time is still evidence that a cry happened, and the dashboard says how many such records there are rather than inventing a date for them. The base-pair clock (`epoch_ms` at `uptime_ms`) drifts with the crystal, so it is a bridge across a reboot, not a timekeeper — SNTP corrects it whenever the network is there.
D-91	**The dashboard is Python's standard library and SQLite, with no dependencies and no build step.** It serves three static files and a small JSON API; it runs on whatever machine the owner already has.	The owner flashes and runs this system himself, offline, without asking (D-70). A dashboard that needs `npm install`, a Node version, a cloud account or a subscription is a dashboard that stops working the day one of those changes. `python dashboard/server.py` is the whole installation. SQLite in WAL mode is a single file he can copy off the machine as the record. The API contract is written down in `dashboard/API.md` so the firmware and the page can be changed independently.
D-92	**The Wi-Fi name and password are compiled in as defaults, overridable at runtime into NVS.** The dashboard's address is *not* compiled in — it is set from the hub's serial console with `net set <url>`.	The owner gave the credentials for this purpose and wants a hub that joins his network on first boot with no setup. NVS survives a reflash, so `net ssid` and `net set` are typed once and kept. **The password is plaintext in the source** (`components/crynet_net/crynet_net.c`) and in any image built from it; that is the deliberate trade for a hub that works out of the box, and it is written down here so nobody mistakes it for a secret. The dashboard address cannot be known at build time — it depends on which machine runs the server and what address that machine has — so the server prints the exact console line to type.
D-93	**The flash write that makes a record number durable stays synchronous, on the audio path, and its cost is measured on the board rather than estimated.** The episode helpers keep it because the wrist's number and the dashboard's number have to be the same number; the commit's second page write was removed, and the counters file is now rewritten in place instead of being deleted and rebuilt.	This was checked properly rather than assumed. Fragment reassembly happens in the ESP-NOW receive callback (`handle_fragment`), not in the main loop, so a flash write in `analyse()` **cannot cost a fragment** — the fragments are already in the buffer before the write starts. What it does is delay the *next* pull, by one page erase and program (tens of milliseconds; the hub prints the measured figure, so it is a number and not a guess). The node holds a frozen window for three seconds and drops it after twenty, so a delay of that size is comfortably inside what the link tolerates; the deaf logic discounts the hub's own deafness, so the extra time makes a cot look *more* alive, never less. The durable reservation is kept for one reason only: if the number were handed out from RAM, a power cut in the middle of a cry would let the next cry be given the same number, and the caregiver's wrist would name a record the dashboard holds for a different cry. That is the one guarantee worth a page of flash. Everything else the log does was moved off the audio path or made conditional: the commit only rewrites the counters when the clock has actually moved (the clock is set once by SNTP and then runs on its own), and the uploader's own path writes them at most once a minute.

| **D-94** | **The uploader picks the records to send by *number*, not by position in the file, and the number it claims as sent stops below every number that is not finished.** The file is walked in order with a linear scan; the batch is the lowest unsent numbers; the watermark is clamped below the lowest number that did not fit in the batch and the lowest number an open episode still holds. | A record's number is spent when the episode *starts* and written when it *ends* (D-89), so two cots crying at once can end in the opposite order — the file is **not sorted by number**, and the old binary search over it was invalid in exactly the case that matters: asked for the first record above a number, it could land *past* a record with a lower number, and that record was then **never sent** — the hub said everything had gone, and the dashboard simply never showed that cry. The fix has two halves. Taking the lowest unsent numbers (not the first ones in the file) is what stops a livelock: taking file order could re-send the same high-numbered record forever while a lower one sits behind it. Clamping the watermark is what stops a false claim: the hub must never say "sent up to #50" while #48 is still being measured. The property is proved twice — `firmware/tools/test_log_order.py` models the algorithm on the host (six tests including a ten-thousand-case soak, and a companion test that shows the *old* algorithm losing records, so the guard cannot silently stop guarding), and the hub's `logtest` console command runs the same scenario against the real component on the real chip, on its own scratch file, touching nothing real. |

---

2. The issues, in the order they were found

Each entry: what it looked like → how it was diagnosed → what was done → the result.

2.1 Model and data

#	Issue	Diagnosis	Measure	Result
I-1	**The INT8-vs-float32 parity cell reported `NaN` for both AUCs.**	The cell took a *prefix* of the validation fold. The ICD manifest lists every positive clip before every negative one, so the fold's positive clips alone contribute 5,330 windows — more than the 4,000-window cap — so that slice was 100 % cry and its AUC was undefined.	The notebook cell now draws a seeded random permutation subset and **warns loudly if a slice is single-class**. The gap was closed independently on the host by `research/verify_int8_parity.py`.	The model was never affected — every headline metric is computed over whole splits — but the quantisation delta had never actually been measured. Now: ΔAUC **−0.001373**, decision agreement **99.956 %**, and the independently computed float AUC (0.999299) corroborates the Kaggle figure (0.999419) from a different set of windows.
I-2	**Contaminated corpora were nearly used for training.**	Every corpus was profiled at the audio level, which is what exposed them.	Rejected at the source with the reason recorded in D-8.	The `mennaahmed23/baby-cry*` family was never trained on. Its "silence" sits at RMS 0.22 — louder than a real cry — and would have taught the model that loud means safe.
I-29	**The deployed model raises a false alarm on 8.19 % of loud music clips — a class it had never seen.** The user's requirement is that the system tells a cry from background noise, speech and music. Speech and noise are handled (1.6 % each); **music was not handled at all**, and the cause was structural, not a tuning slip: **no corpus in the training mix contained any music**, and ESC-50 — the corpus used for the false-alarm map — has none either, so neither the training nor the measurement ever touched it.	Two measurements. First `research/eval_esc50.py` over ESC-50's 2,000 real clips through the deployed INT8 model: 36 of the 40 real `crying_baby` clips alarm (90 %) and 38 of the 1,439 loud non-cry clips do (2.64 %) — urban 4.3 %, animal 3.7 %, domestic 1.6 %, human non-speech 1.6 %, nature 0.8 %. Then the corpus was checked for what it does *not* contain, which is what exposed the gap.	GTZAN was downloaded and `research/eval_music.py` ran the same deployed model over its 1,000 real music clips: **79 of the 965 clips that reach the model alarm — 8.19 %**, three times the rate for any other kind of sound. Worst genres: jazz 14.3 %, blues 13.0 %, pop 12.0 %, disco 10.0 %, rock 10.0 %; best: reggae 2.0 %, classical 2.9 %, country 3.0 %, metal 4.0 %. The mechanism is obvious in hindsight: a cry is a voiced, harmonic-rich sound, and a saxophone, a trumpet or a sung vowel in the log-Mel domain looks very much like one. GTZAN was added to the training corpus (D-51), the notebook was regenerated and validated locally on the stub cache, and the Kaggle re-run is the user's to start.	**Not fixed yet — the deployed model still false-alarms on music, and the fix is a training round, not a code change.** The deployed threshold cannot rescue it either: the model's INT8 output saturates at P(cry) = 0.9961, so 45 of the 81 music alarms and 34 of the 36 true cry detections sit at exactly the same value, and raising the threshold from 0.95 to 0.99 would remove 27 music alarms at the price of two real cry detections. A "two consecutive windows" confirmation was considered and deferred: music is continuous, so it would not help against music, and it would cost the demo a short single cry clip.

2.2 Stage 2a — the hub's inference core

#	Issue	Diagnosis	Measure	Result
I-3	**The DC-removal mean was accumulated in float32.**	Summing 32,000 samples of 18,000 in float32 gives 575,726,660 instead of 576,000,000, so the mean came out 8.5 counts low and 8.5 counts of DC survived. The per-window standardisation then blows that residual up into a completely different feature vector — the DC-offset probe failed with a max delta of 131.	The sum is now an exact `int32`.	Probe 2 passes; the on-device feature vectors match the host exactly.
I-4	**The build was at `-Og`, not optimised.**	`build_flags = -DCOMPILER_OPTIMIZATION_PERF` is an *Arduino* macro that means nothing to PlatformIO's ESP-IDF builder, so ESP-IDF's debug default applied.	Set the optimisation the ESP-IDF way: `CONFIG_COMPILER_OPTIMIZATION_PERF=y`.	Part of the front-end speed-up below.
I-5	**The CPU ran at 160 MHz, not 240.**	Read from the platform summary on the device.	`CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ_240=y`.	The board now reports 240 MHz.
I-6	**The front end kept a 128 KB padded copy of the signal in PSRAM.**	PSRAM on the S3 is uncached external memory, and the FFT and the Mel filterbank stream through the working set 81 times per window.	The padded copy is gone: each frame is built straight from the caller's int16 PCM. The working set is now **5,380 bytes in internal SRAM**.	Front end **135 ms → 41.7 ms**.
I-7	**A combined image built with QIO flash mode made the chip watchdog-reset in a loop.**	`rst:0x7 TG0WDT_SYS_RST` repeating with no application output; the ROM cannot read this board's flash in QIO before the bootloader has configured its own timing.	The combined image must use **DIO**. The whole build is DIO — the effective config is `CONFIG_ESPTOOLPY_FLASHMODE "dio"` and both images carry flash-mode byte `0x02`.	The board boots and runs. DIO costs nothing at this workload.
I-8	**The FlatBuffer alignment, the arena fit, the feature agreement and the latency were all unverified.**	Only a build proves nothing about the chip.	The same four probes the host parity test uses are run **on the target MCU**, and the device reports its own arena usage and timings.	**RESULT: PASS — 4 probes.** Arena 262,144 B reserved / 108,328 B used; front end **41.7 ms**; inference **101.8 ms** (ESP-NN SIMD kernels linked in: `esp_nn_conv_s8_esp32s3`, `esp_nn_conv_s8_1x1`, `esp_nn_conv_s8_3x3_opt`).

2.3 Stage 2b — the node firmware

#	Issue	Diagnosis	Measure	Result
I-9	**The INMP441's data line read exactly 0.00 in steady state.** The captured window was flat at 0–1 count with 533 single-sample spikes.	A new bring-up sketch (`node_diag`) read both I2S slots and scanned every exposed GPIO. The SCK scan read 0.00 on **every** pin, which is the clearest statement of the fault: with the clock on the wrong pin the microphone is never clocked, and

with the clock on the right pin its data line still stayed flat. A repeat measurement of the identical configuration read 0.00/0.00 where the first read 3961/2320 — the apparent "signal" was the peripheral's **start-up transient**. | The fault was in the **wiring**, not the firmware: the INMP441 only drives SD when it is powered, clocked and given a valid word-select. The user rewired the board. | `verify_burst.py` then reports **PASS, 286/286 fragments, 0 bad CRC, exactly 64,000 bytes**, written out as a 2.00 s WAV: 1,943 distinct values, RMS 349 counts (0.011 of full scale), peak ±1,150 — a quiet room, no clipping. |
272 | **I-10** | **The two sketches disagreed about the I2S format** (one `MONO`+`SLOT_LEFT`, the other `STEREO`+`SLOT_LEFT`). | The diagnostic settles it: **STEREO with both slots, taking every other word**. MONO halves the I2S frame to 32 bits, which breaks the INMP441's framing — the same microphone read the same way gives −3500 counts of DC with the full frame and −620 with the halved one. Asking for stereo also makes the Arduino library force the both-slots mask whatever mask is passed, so the mask argument never does what it appears to do. | Both sketches now do the same thing. | The INMP441 puts 24 bits into a 64-bit frame, so the I2S setup uses **32-bit slots**; the narrowing to int16 is done in software by keeping the top 16 bits. The ESP32-C3 is I2S hardware v2, so the left-channel slot is selected in hardware, with no software decimation. |
273 | **I-11** | **A closed serial port starved the node's loop — the root cause of the week.** The mic appeared dead; the "INMP441 power-on transient" seen in hub captures was stale ring-buffer data. | ESP32 Arduino's USB-CDC (HWCDC) write path waits `tx_timeout_ms` (default 100) × up to 20 consecutive timeouts = **~2 s per print** when no host reads the port, and `usb_serial_jtag_is_connected()` stays true whenever the cable is plugged in, so `if (Serial)` gates did *not* help. The node printed its RMS every 100 ms, so an unattended node crawled to ~3 loops/s: the mic starved, telemetry fell to 0.35 Hz, and the window never filled. | **`Serial.setTxTimeoutMs(0)`** in `setup()` — prints drop instead of block. | Telemetry hit 10 Hz within one flash. (The earlier sessions only worked because the Arduino IDE serial monitor held the port open.) |
274 | **I-12** | **The INMP441's power-on transient** — the first 100 ms record read 0.656 of full scale, decaying 656 → 79 → 41 → 22 → 12 milli over ~500 ms. | It is the microphone's power-on burst, and it mattered twice: clips are cut from t=0, so `clip_0000` of every session contained it, and on the deployed node it would freeze a garbage window the moment it crossed the event level and hand the hub a spurious upload after every boot. | Both sketches read and discard 500 ms (8,000 stereo frames) right after the I2S starts. | A fresh capture session reads **7 milli on its very first record** — room level from sample 0, versus 656 before. `node_crynet` boots with a first RMS of 0.047 and never crosses EVENT. |
275 | **I-13** | **The node's window mean averaged the stale tail with the fresh head.** | After a burst the live window is dropped and refills; until it is full the newest samples are not at the start of the array, so a mean summed from index 0 was meaningless for exactly as long as the window was invalid. | The valid region is now walked **backwards from the write cursor**; the cursor is no longer reset, and a request arriving while the window is incomplete is refused rather than answered with a window whose order is unknown. | Both sketches recompiled clean. |
276 | **I-14** | **The Arduino IDE build failed with `cc1plus.exe: out of memory allocating 4194296 bytes`.** | The machine's **page file is disabled**, so the commit limit was 16,190 MB against 15,109 MB committed. | `AutomaticManagedPagefile` was set to `True` (the Windows default), which takes effect at the next boot; meanwhile the sketches are built and flashed with `arduino-cli` outside the IDE. | Both sketches build with **zero warnings**: `node_crynet` 979,375 B (74 %) flash / 166,308 B (50 %) RAM; `node_capture` 311,120 B / 18,592 B. |
277
278 ### 2.4 Stage 2c — the radio, the panel and Bluetooth
279
280 | # | Issue | Diagnosis | Measure | Result |
281 |---|---|---|---|---|
282 | **I-15** | **`handle_telemetry` memset a new node's slot to zero, wiping the 64 KB window pointer `node_link_create` had pre-allocated.** | Every fragment was rejected on `window == NULL` and **not one burst ever reassembled**. The stats line decoded it: 702 raw, 702 parsed OK, 560 bad — parse passed, `handle_fragment` rejected; the only silent reject was the window pointer. | Save and restore the pointer across the memset; a slot with no window buffer now logs an error instead of being dropped silently. | Bursts reassemble. |
283 | **I-16** | **The hub's 2-second probe had become a 50 ms flood**, and the node wiped its whole live window after serving a one-fragment probe. | The heartbeat edit had deleted `last_probe_us = probe_now`. | Restore the timestamp; the node only wipes the live window after a real burst. | The probe flood is gone. |
284 | **I-17** | **`esp_now_add_peer` ran from the node's WiFi-task callback.** | The hub's own `node_link.c` documents this exact trap (see D-38). | Peer-add moved into `loop()`. | The hub receives every frame the node sends. |
285 | **I-18** | **Unicast to the learned hub address wedged the ESP-NOW send path.** | `esp_now_send` returned OK, frames never left, the send callback never fired, every send burned its wait cap and the loop starved. | **Rolled back to broadcast** at 10 Hz, which is reliable now that the loop runs in real time. | Broadcast works. This also disproved the "the hub radio sleeps" theory, so the 100 ms heartbeat was removed. |
286 | **I-19** | **The node on the board was running stale firmware.** | The "freeze before you announce" change had been written to `node_crynet.ino` but never reflashed, so the node had no auto-freeze at all: `g_peak_pending` appears 0 times in the last committed node build and 7 times in the current source. | Rebuilt and reflashed. | The chain runs for the first time end to end. |
287 | **I-20** | **The node's freeze was cancelled in the same tick that created it.** | `if (g_frozen_valid && (now - g_frozen_ms) > FROZEN_EXPIRE_MS)` used a `now` captured *before* `freeze_window()` ran, and `g_frozen_ms` is stamped *inside* it — so the subtraction underflowed to ~4 billion and the window expired the instant it was frozen. The symptom was unmistakable: the frozen sequence advanced (1, 2, 3, 4, 5 …) while the flags stayed `0x06` and the CRC stayed `0x00000000`, so the node never once advertised a burst as ready. | Measure the age from a fresh `millis()`. | The telemetry advertises `flags 0x07` and the window's CRC. |
288 | **I-21** | **The hub was still in promiscuous Wi-Fi mode** — a leftover diagnostic. | It hands the host every frame on the channel; with the laptop's own Wi-Fi on channel 1 that flooded the receive path. | Promiscuous mode off by default (D-34); `ESP_WIFI_STATIC_RX_BUFFER_NUM` 16 → 32 and `DYNAMIC` 32 → 64; 3 ms of pacing between the node's fragments. | The fragments are lost between the two radios, not in either end's code — the node's own numbers were healthy: 7,667 frames sent, **0 lost**, 2.3 ms average send wait, and the hub's parse was clean (0 bad frames, 0 bad payloads). |
289 | **I-22** | **Bluetooth and the ESP-NOW radio did not coexist.** With BLE up, `esp_wifi_init` failed (`esf_buf_setup_static: alloc eb fail(1)`) and the radio never started; in a later build NimBLE **asserted** on `Failed to allocate memory for ble_store_config_vars` and reset the chip in a loop. | The first fix attempt (radio before BLE) moved the failure to the other side, which is what showed the problem was not ordering. The real cause was in the ELF: **154 KB of the hub's static RAM was boot-only buffers that never got handed back**. | Move them to PSRAM (borrowed and returned) and free the probes' CRC buffer after each use (D-39). | RAM **59.9 % → 12.9 %**. The whole stack fits: free internal heap 215,155 B with the radio up, 149,271 B with Bluetooth up. Verified live: the panel comes up, the radio hears the node, the BLE server advertises, and a real client connects and subscribes while the radio keeps running — frames counted 21 → 308 with **0 bad, 0 bad payloads, 0 failed requests**, no resets. |
290 | **I-23** | **Three IDF 6 facts the build had to be taught.** | (1) The SPI and GPIO drivers live in `esp_driver_spi` / `esp_driver_gpio`, not `driver`. (2) A stale CMake configure ignores a `REQUIRES` change until the build directory's CMake state is deleted. (3) `font5x7.h`'s rows need their own braces or `-Werror=missing-braces` stops the build. | All three fixed. | The panel driver builds and runs. |
291 | **I-31** | **The panel's picture was a quarter turn out.** The first photograph showed the whole screen rotated 90°: the title ran down the side of the glass instead of across it. | The panel is a **"landscape-first" ILI9341** — a variant whose glass is 320x240 in the controller's *portrait* mode. The legacy firmware drove exactly this panel with **Adafruit's rotation 2 (`MADCTL_MY | BGR`) and a 320x240 logical framebuffer**; its `HubDisplay` class exists for no other reason than to override Adafruit's 240 px clipping. My driver used rotation 1 (`MADCTL_MV`), which is the *normal* chip's landscape setting and transposes the image on this glass. The proof was in the legacy source, not in theory. | The driver now uses the legacy's setting, and it logs what it chose (`rotation 2 (madctl

0x88)`) so the next photograph settles it in one look. | *The general lesson, and it is the same lesson as the pin assignments: **a panel's orientation is a fact about the hardware, not about the chip.** When two systems disagree about a physical device, the one that was photographed working wins.* |
292 | **I-32** | **The Kaggle run died with `TypeError: ufunc 'invert' not supported for the input types` — and the real fault was that the music corpus was empty.** The crash was in `X_gtzan[~gtzan_hold]`: `split_by_clip` returned `np.array([])` for an empty corpus, which numpy types as **float64**, and `~` (bitwise invert) has no float implementation. The empty corpus was the real problem, and the numpy error hid it. | GTZAN ships as Sun/NeXT **`.au`** (1,000 files, `genres/<genre>/<genre>.00000.au`), and the notebook's reader filtered the archive by extension and decoded with libsndfile. **The notebook had been validated on a synthetic stub whose file names and format were different from the real archive**, which is exactly why the gap survived; the local run against the real archive produces 2,000 windows with 0 unreadable, so the *logic* is right and Kaggle's environment is what failed. | Three changes: (1) `split_by_clip` always returns a **bool** array and an empty corpus can no longer crash a split; (2) the corpus reader is **no longer filtered by extension** — anything inside a genre folder is offered to the decoder, and the genre is read from the folder above the file, with a fallback for a flat archive; (3) **`decode()` can now read a `.au` without libsndfile** (the format is simple: magic, offset, size, encoding, rate, channels — and the standard library's `sunau` is gone in Python 3.13), so a build of libsndfile without AU support cannot silently empty the corpus. On top of that, `build_corpus` now **counts why items were dropped** (`no_member`, `unreadable`) and the notebook **refuses to go on to train if GTZAN is empty**, printing the archive's size, member count and first three names. | *A training round that quietly loses the music corpus produces a model that false-alarms on music while looking like it was fixed. Failing loudly is worth more here than finishing.* The user's next Kaggle run will either work, or print exactly which of the three possible failures it was. |
293 | **I-30** | **The TFT panel showed static noise — and the driver reported "the panel is up" the whole time.** The user wired the panel and plugged both boards in; the chain ran end to end on real audio (nine windows at P(cry) = 0.9961, one at 0.9883, CRC-32 MATCH on every burst, and P(cry) = 0.0000 on the quiet room and on a loud non-cry sound), but the panel was a field of coloured noise. **Three bugs, all in my own driver, all of which the driver was structurally unable to detect.** | **(1) The address window was little-endian.** `CASET` and `PASET` take four bytes big-endian; the code cast a `uint16_t` to `uint8_t *`, so 319 (0x013F) went out as 0x3F 0x01 and the panel read column 16129. **(2) `CASET` was sent twice.** The window is one command for the columns and one for the rows; sending the column start and then the column *end* as a second `CASET` re-sets the start instead of the end, collapsing the window to one pixel. **(3) CS was released between a command and its parameters.** The ILI9341 resets its serial interface when CS goes high, so every parameter byte was arriving as a *command* — the panel was never initialised at all, and what the user was looking at was uninitialised display memory. | The driver now **drives D/C in the SPI pre-transmission callback** (the only moment the level can safely change), sends **a command and its parameters as one transfer** with `SPI_TRANS_CS_KEEP_ACTIVE` on an acquired bus, and builds the address window big-endian, one command per axis. The text renderer was rewritten to draw **horizontal runs** instead of single pixels — about sixty times less SPI traffic, which is what makes a redraw cheap enough to run every second. A **blue 12×12 marker in the panel's top-right corner** settles the orientation from one photograph. | **Fixed and reflashed** (`research/toolchain/pio_build_display.log`, RAM 12.9 %, flash 61.9 %). The lesson is recorded rather than the fix: **a display driver that cannot fail is a display driver that lies** — the ILI9341's ID register can be read, but MISO is not wired, so it cannot be, and the driver's "up" message was only ever reporting that the SPI transactions returned without error. |
294 | **I-33** | **The panel showed hearts and diamonds where the letters should have been.** Every glyph was 32 characters out of place, so the new dashboard was unreadable — and the host-side preview of the same screen (`research/out/ui_dashboard.png`) showed the same nonsense, which is how the fault was visible before the hardware ever was. | `Adafruit_GFX::drawChar` reads the classic font as `font[c * 5 + i]` — **the raw ASCII code**, with no `- 0x20`. The generator assumed the old layout and wrote `off = (code - FIRST) * 5`, so ASCII 0x20 took the table's first entry, ASCII 0x41 took the entry for 0x21, and every character drew the glyph of the character 32 codes earlier: 'A' drew '@', and a space drew the heart at index 0x01. **The font file was never corrupt** — the local `glcdfont.c`, the upstream copy and the cached copy are byte-identical, which is what finally killed the "the library's font is damaged" theory. | The generator now **detects** the table's base from the position of the classic '!' and refuses to write the header unless the layout is what it expects (D-51). The renderer reads bit 0 as the top row, the way Adafruit's own `drawChar` does (`for (j = 0; j < 8; j++, line >>= 1) if (line & 1)`); the host-side preview had the opposite convention and was drawing every glyph upside down, which would have hidden the fix behind a second bug. The preview now mirrors the firmware's layout page for page, so the two cannot drift. | The glyphs render as letters — checked by decoding the generated header back to ASCII art ('A', '0', 'g', '!', 'R' all read correctly) — and the dashboard reads `CRYNET`, `LISTENING`, `COT 1`, `ONLINE`, `NO SIGNAL`. |
295 | **I-34** | **The display source was saved in Windows-1252, and a later edit double-encoded it.** | The em-dashes in the comments were single 0x97 bytes, so a UTF-8 read of the file failed outright, and the next edit turned them into `â€”` — mojibake in the source of the file the user was reading. | The file is now valid UTF-8 (seven em-dashes, no mojibake), decoded byte by byte so that no character is guessed at, and **no non-ASCII byte remains inside a string literal** — the panel can only draw ASCII in any case. | The source compiles and the comments read as written. |
296 | **I-35** | **Every colour on the panel arrived with red and blue exchanged — the teal theme was showing as olive and salmon.** The dashboard's teal accent came out olive, the amber "no phone attached" pill came out **blue**, and the user read the whole thing as a different, garish theme ("the colour scheme also thrash"). The picture was right — layout, text and orientation were all correct — so the fault was in the colour path alone, and the driver reported the panel as "up" throughout. | The photo was measured rather than guessed at. The screen was located in the photograph, the picture mapped onto it with a bilinear transform, and the *saturated* pixels sampled — text and the 6×6 pill squares, which the palette pins down exactly: `CRYNET` (teal 45,212,191) measured `(210,210,179)` olive; `LISTENING` measured `(212,218,184)` olive; the `RADIO` square (green 52,211,153) measured `(174,180,146)` olive; the `PHONE` square (amber 251,191,36) measured `(5,175,247)` **blue**. Every hue is present and correct *except* that red and blue are in the wrong places. The cause is the `MADCTL_BGR` bit in the driver's rotation setting: the panel was told that the top bits of each RGB565 word are blue, while `DISPLAY_RGB` puts red there. | The `BGR` bit is cleared from every rotation setting, so the panel is told the order the framebuffer actually uses (D-53), and `INVOFF` is sent explicitly so the inversion state is pinned rather than inherited. The driver logs the control byte it actually wrote (`madctl 0x80`), so the next photograph can be checked against the log rather than against a belief. | Built (RAM 12.9 %, flash 62.0 %) and flashed: the panel comes up at 320×240 rotation 2 and the log now reads **`madctl 0x80`** — the `BGR` bit is gone — with the radio healthy (110 frames off the air, 0 bad, 0 bad payload). **One more photograph is needed to confirm the colours on the glass.** |
297 | **I-36** | **The panel claimed more cots were linked than were answering.** The footer read '3 cots linked' while `COT 2` and `COT 3` both read NO SIGNAL - visible in the user's photograph. | The footer's count came from `st->node_count`, the number of cot *slots* the hub holds, so every card drawn was counted as a cot linked, whether or not that cot had been heard from recently. | The footer now counts the cots whose cards read ONLINE, and says 'no cots linked' when none are answering; the host-side preview was changed the same way so it cannot drift back (D-54). | The footer agrees with the cards above it: '1 cot linked' beside one ONLINE cot and two NO SIGNAL cards. |
298 | **I-37** | **The Kaggle download cell failed with `FileNotFoundError: '/kaggle/working/out'`.** The user ran the training notebook, it finished, and the cell that was supposed to hand over the artifacts stopped on the first line that touched the disk. | The cell assumed a fixed path, but the notebook's own output directory is `OUT = Path(os.environ.get("CRYNET_OUT", "./out")).resolve()` - **relative to whatever the working directory was when that cell ran**. The session that ran the notebook and the session that looked for the result were not the same one, so the directory was not where the cell expected it. The same failure mode is invisible to a notebook that is run end to end in one go, which is why the local validation never caught it. | The cell now **asks the notebook first** - the variable `OUT` survives in the kernel if the export cell ran in that session - and otherwise **searches `/kaggle/working`, the working directory, the home directory and `/tmp` for the artifacts by name**, wherever they are. If it finds nothing it prints what *is* under `/kaggle/working`, so the next attempt starts from facts instead of another guess. It still writes `/kaggle/working/crynet_out.zip` and offers it with `FileLink`, keeping the `out/` folder at the top of the archive. | The

```
artifacts are still outstanding: until the zip is unpacked, the deployed model false-alarms on 8.19 % of loud music (I-29), and the retrained model cannot be compared with the one on the board. |
 299  | **I-38** | **A photograph of the panel cannot measure its colours - the same colour reads (9,51,210) in one place and (51,120,235) in another.** The swatch page made the problem measurable for the first time: the navy swatch and the page background are *the same colour*, 80 screen pixels apart, and they came back 147 blue levels apart. | The phone is doing it, not the panel: in a dark room the camera opens up for a glowing screen, the highlights clip (the `RADIO` square in the dashboard photograph measured (144,255,255) - fully blown out) and the white balance drifts with whatever it sees. Measured along one horizontal strip of the swatch page, **one single colour** reads (40,116,238) at the left and (25,75,204) at the right: a spread of 40 levels per channel for a colour that does not change. Any conclusion drawn from a difference smaller than that is noise. | The colour-check page is the way out: it says what the panel was *asked* to draw, so a photograph can be compared with it rather than believed. The page has grown a **dark ramp** - six steps of one hue from black up - so the panel's own tone curve is read off in one row. And the one photograph that would settle it is the one to take **with the room lit**, where the camera does not open up for the screen. |
 300  | **I-39** | **Every rounded corner on the panel drew its arc on the *inside* of the shape - only the bottom-right corner looked right.** The user saw it before any tool did: 'the corners are off, only the bottom right corners are okay, the other are sort of flipped'. | The corner loop walked its offsets only inward from each disc's centre: for a top-left corner centred at `(x+r, y+r)` it drew the quadrant to the right and *below* the centre - the quadrant that lies inside the rectangle. The fill version painted over pixels the straight bands had already filled (harmless, invisible), and the outline version floated a stray quarter-arc inside each corner while the real corner was left square-cut. The bottom-right corner only looked correct because its inward quadrant happens to end exactly at the shape's own edge. | Each disc is now walked **towards its own corner** (a per-corner sign pair `sx`/`sy`, offsets `0..r`), so the quarter-disc covers the corner quadrant and the outline's ring hugs the corner arc. A host-side regression check (`research/check_corners.py`) requires the four corners to be exact mirror images of each other and the fill's arc to hug a circle within 1 px, for every radius the UI uses - the old loop fails it, the new one passes. |
 301  | **I-40** | **A flash aimed at the wrong serial port parked the cot node in its download mode, and the radio went silent (0 frames for two minutes).** | Two boards were attached: the hub on COM32 (the CH343 bridge) and the node on COM30 (the C3's own USB) - exactly what the ports table in SYSTEM_UPDATE.md section 0.1 says. The hub's upload was attempted with auto-detect and then with COM30 named explicitly; both times esptool connected to the **node**, put it into download mode, and then stopped on a chip mismatch - leaving the node sitting in the bootloader instead of running its firmware. The hub then probed an empty room: `0 frames off the air`. The display change under test touches nothing in the radio path, which is what made 'the radio is suddenly dead' diagnosable as an environment fault rather than a code fault. | `esptool --chip esp32c3 --port COM30 run` told the node to leave the bootloader; the next hub log reads `410 frames off the air, 410 parsed, 0 bad, 0 bad payload` with the node answering at rms 0.004-0.014. The rule going forward: **read the ports table before flashing**, and a failed esptool session leaves its target in the bootloader - `esptool run` brings it back. |
 302  | **I-41** | **With the new weights in, the build stopped on a static assert - and after that was fixed, the first boot failed its on-device feature check (`max delta 87, 0/5184 exact -> FAIL`).** | Both are the drift guards working as designed. `crynet_dsp.c` static-asserts the firmware's pinned input calibration against the generated tables, and the retrain recalibrated the input (`IN_SCALE` 0.0601 → 0.1163, `IN_ZP` −4 → −70), so the stale mirrored macros in `crynet_model.h` stopped the build instead of letting a wrong-quantised model ship. With the macros updated, the boot probes then compared the device's quantised features against `crynet_probe_refs.h` - reference vectors **baked from the old quantisation**. The float front end is identical, but every int8 feature moved with the scale (worst delta 87, zero exact). | `firmware/tools/gen_probe_refs.py` re-runs the notebook's front end over the four closed-form probes and rewrites the reference header; after it, the self-check reads **5184/5184 exact on every probe, RESULT: PASS**. The rule: a new model means four files change together - `model_data.h`, `crynet_tables.h`, the macros in `crynet_model.h`, and the probe references - and `check_artifact_drift.py` verifies the first three against `deploy_config.json`. |
 303
 304  ### 2.5 The airtime problem — the last and hardest issue
 305
 306  This is the one that kept the burst from ever completing, and it is worth recording in full because the reasoning was wrong twice before it was right.
 307
 308  | # | Issue | Diagnosis | Measure | Result |
 309  |---|---|---|---|---|
 310  | **I-24** | **The burst never completed: 267–274 of 286 fragments arrived, then the burst timed out.** | Three hypotheses were tested in turn and two were wrong. *(a) The Wi-Fi channel* — both sides were moved to channel 6 and the node was heard. *(b) The node's own counters* — 7,667 frames sent, **0 lost**, 2.3 ms average send wait, so the node was not dropping anything. *(c) **The hub's own Bluetooth advertising*** — the decisive test: the hub was built **without BLE** and produced **286/286 fragments in 3.75 s, four complete bursts in a row**. With BLE on, the same build lost ~6 % of every burst. | BLE advertises on channels 37/38/39, and the hub's single radio must *retune* to reach them — during which the Wi-Fi receiver is deaf. Three BLE events per advertisement at ~30 ms intervals ≈ 9 % deafness, independent of the Wi-Fi channel and the frame rate. That fits every observation, including why it was independent of the channel. | The root cause was confirmed. It is not a bug in either end's code; it is the physical reality of one radio carrying two protocols. |
 311  | **I-25** | **The burst had no way to recover the frames it lost.** | The hub could see *that* fragments were missing but had no way to ask for them, and the node had no way to send them again. | Implement the `RESEND` frame (D-26): a 36-byte bitmap of the missing fragments in a 54-byte request; the node re-sends exactly those from the same frozen window with the same sequence number. | **The burst completed 286/286 with BLE on**, CRC-32 MATCH, and the first end-to-end cry alert. |
 312  | **I-26** | **A resend request could be lost entirely.** | ESP-NOW broadcast is never acknowledged. The resend goes out at exactly the moment the burst has just gone quiet — the moment the radio has been busiest. One lost request means the round is wasted and the burst times out with the fragments still missing. | The request is sent **three times** (D-28). | A lost request no longer costs a round. |
 313  | **I-27** | **The node re-sent the same fragments two or three times.** | The hub's three request copies each set the node's single pending flag, so the node served the whole request again. With 128 fragments missing that meant 256 or 384 fragments of air instead of 128, and the burst took 9.6 s instead of 5. | The node answers a repeat of the request it is already serving only once, within an 800 ms window (D-28). | One serving per round; the burst settles in about 5–7 s. |
 314  | **I-28** | **The node refused the resend: "no frozen window for it."** | The node froze the next window about three seconds after the upload, and the hub's request for the missing fragments often arrived just after that — by which time the old snapshot had been replaced. | The node **holds** an uploaded window for four seconds (D-29), and the hub **abandons** a burst the node can no longer complete and asks for the newer window at once (D-30). | The resend is served, and a hopeless burst is recovered in about three seconds instead of fifteen. |
 315
 316  **The live test that closed it** — hub + node + a real BLE client connected, with the cry recording played through the PC's speakers:
 317
 318  ```
 319  W (37022) link: node 1: the burst stalled with 268 of 286 fragments; asking for the 18 that are missing
 320  W (38562) link: node 1: the burst stalled with 285 of 286 fragments; asking for the 1 that are missing
 321  I (39830) link: node 1: 286/286 fragments in 6.58 s
 322  I (39830) link: node 1: window crc32 0x336F318E, the node advertised 0x336F318E -> MATCH
```

```
...
I (60681) link: node 1: window crc32 0x6230F1C4, the node advertised 0x6230F1C4 -> MATCH
I (60842) link: node 1: ALERT — cry
```

and on the phone:

```
ALERT {'cot': 1, 'alert': 'cry', 'confidence_pct': 100, 'event_id': 1}
STATE {'cots': 1, 'crying_mask': 1, 'confidence_pct': 100, 'started_at_s': 0}
ALERT {'cot': 1, 'alert': 'cleared', 'confidence_pct': 100, 'event_id': 2}
```

---

3. Results so far

Claim	Number	How it was measured
ICD validation AUC	**0.9994**	full Kaggle run, fold 4
ICD test AUC	**0.9995**	full Kaggle run, fold 5
Cross-corpus AUC (never trained on)	**0.9973**	cry4 + ESC-50 + snoring
Deployed model	**137,384 bytes** INT8	SHA-256 `4aef55d8…126bc11`
C front end vs Python	max **6.9 × 10⁻⁵**	4 probes, real run
INT8 vs float32	ΔAUC **−0.00137**, agreement **99.956 %**	11,386 windows, host
On-chip front end	**41.7 ms** per 2 s window	ESP32-S3, 240 MHz, −O2
On-chip inference	**101.8 ms** per window	ESP-NN SIMD kernels
Hub RAM	**12.9 %** (42,236 B)	after the PSRAM move
Hub flash	61.9 %	1,297,303 B of 2 MB
Hub internal heap with radio + Bluetooth	**149,271 B** free	at boot
Node flash / RAM	979,375 B (74 %) / 166,308 B (50 %)	`weact_studio_esp32c3`
Radio link	**0 bad frames, 0 bad payloads** over thousands of frames	hub's own counters
Burst completion	**286/286 with CRC-32 MATCH**	live test, BLE client connected
End-to-end alert	**`ALERT {'cot': 1, 'alert': 'cry', 'confidence_pct': 100}`**	delivered to a real BLE client
Real baby cries detected	**36 of 40** ESC-50 `crying_baby` clips alarm	deployed INT8 model, `research/eval_esc50.py`
False alarms, loud non-cry sounds	**2.64 %** (38 of 1,439)	ESC-50, environmental + human non-speech
False alarms, **loud music**	**8.19 %** (79 of 965)	GTZAN, ten genres — `research/eval_music.py`
False alarms, quiet room	**0 in 236 windows**	this room's INMP441, `research/eval_indomain.py`
Live chain, phone-played cry	**P(cry) = 0.9961** on nine windows and 0.9883 on a tenth, every burst **286/286 with CRC-32 MATCH**	hub + node + a real BLE client, `research/hub_live.log`
Live chain, quiet room and a loud non-cry sound	**P(cry) = 0.0000** on every window, no alert raised	same session — the trigger fired (RMS 0.035-0.085) and the model rejected it

---

4. Open items

#	Item	Why it is still open
O-1	~~**The wearable (Stage 2d)** does not exist yet.~~ **RESOLVED (O-1a):** the wearable exists, is flashed and is on the bench.	It is a BLE notification client with an SSD1306 OLED and a DRV2605L haptic driver. The hub's side of the contract (the legacy UUIDs and the six-byte alert packet) is done and verified against a real client, so the wearable can be written directly against it.
O-1a	**The wearable exists and is on the bench** — OLED on 3V3 / SDA GPIO5 / SCL GPIO6, and a BC547 low-side switch driving a real haptic motor on GPIO7 (D-77). **What remains: the battery itself.** The battery/SOC readout that used to be listed here is **cancelled** — there is no divider and no SOC text on the OLED (D-83).	The wearable was flashed and proved end to end on 2026-11-06: three alert cycles in a row came through complete (`ALERT cot 1 type 1 conf 100%` on the wrist, the haptic fired, `clear` came back as `type 2`), and it recovered from a hub reset mid-link (`hub lost — rescanning` → found again → subscribed to alerts *and* state). The OLED shows the link state and the alerts, and nothing about the cell. The cell choice is settled (D-81: a 1000 mAh pouch, with the charger's current reduced to ~400–500 mA); what is left on this unit is the battery and its power path, not an indicator.
O-1d	**No battery runtime has ever been *measured*, and the charge current, the boost converter's quiescent drain and the switch-off drain of the LX-LCBST module are all still unmeasured. The wearable has now RUN on its cell — that part is no longer a proposal.**	**Corrected 2026-11-23 from the 2026-11-20 bench session:** the wearable was on its **Li-Po pouch**, not USB, for the third round of that session (which is why COM30 was absent and no wrist log could be captured). So a unit has run from a cell and kept working — the earlier absolute claim that *nothing* had ever run from a cell was wrong. **What is still unmeasured is how long it lasts**, and one live suspicion came out of that round: the haptic buzz was missing while on the pouch, and a sagging 3.3 V rail is a candidate (`SYSTEM_WIRING.md` §5.3 anticipates exactly this). The firmware is exonerated — D-86 touches no motor line — so the decisive test is D-85's meter-across-the-motor reading during an alert: ≈3.3 V means no current (open coil or dry joint), ≈2.1 V means current flowing but a weak motor or a sagging rail, ≈0 V means the BC547 is not switching. The nodes and the hub have still never run from a cell. The battery-powered versions of those units are still proposals. The runtime projections in D-82 come from the linear regulator's behaviour, not from a bench measurement, and the module's own current draw is unquantified — a module that draws a milliamp in standby would dominate a wearable's budget. Until one board is built and measured, the numbers in the documents are estimates and are labelled as such. The bench procedure is in `SYSTEM_WIRING.md` §6: build **one** power assembly, measure charge current, set the trimpot to 5.1 V, confirm switch-off behaviour and off-state drain, then run the working node firmware from it and compare against the USB baseline.
O-1e	**CLOSED BY D-83 — the divider was withdrawn before it was fitted, so there is no source impedance to measure and no 100 nF to keep.** The item was opened because the pair's source impedance is 68.75 kΩ (220 kΩ ∥ 100 kΩ), above the ~10 kΩ an ESP32 ADC likes, and the 100 nF on the sense node was what made the reading accurate — load-bearing, not a nicety. The fallback had been worked out (a 22 kΩ / 10 kΩ pair keeps the ×3.2 ratio exactly and drops the source impedance to 6.9 kΩ, for about 118 µA more drain), and it was left as a bench decision because the owner was mid-solder. **None of it is needed now.**	The whole feature is gone: no divider, no 100 nF, no ADC read, no telemetry field, no panel text. The firmware no longer has a battery reading to be noisy or compressed (D-83), and the BOM carries no divider parts. **Nothing was ever measured** — no cell was ever read through this divider, so the open question this item carried has simply ceased to exist rather than been answered. The reasoning behind the divider is kept in D-81 for the record, and the fallback pair is recorded there too in case the feature is ever revived; if it is, the honest part is a MAX17048 fuel gauge rather than a divider.

374 | **O-1b** | **CLOSED BY D-86 — two cots crying at the same time has been tested, on the bench, and the wrist names both.** | On 2026-11-20 one clip was heard by both node microphones and the hub's crying mask sat at `0x03` for **fifteen consecutive seconds**; the owner watched the alert face read the two-cot label. Three separate two-cot events were captured. **The one-buzz rule held across all of them:** nine cry announcements across five episodes produced exactly five buzzes, and the four silent ones were handovers between cots, exactly as the code says they should be. What the test did **not** show is a second buzz when a second cot joins — that was never implemented and D-86 deliberately does not add it: one cry is one buzz. The parked branch `feature/wearable-multi-cot` (D-80) is an earlier, different attempt and is superseded. |
375 | **O-1f** | **This repository's D-86 wearable image is compile-verified but not flashed.** | The behaviour was proved on the bench on 2026-11-20 from the sibling repository's build, and the port here compiles to the identical figures (685,589 B / 18,684 B), but **no board was attached when it was ported**, so `firmware/prebuilt/wearable/app.bin` in this repo has never been written to a device. Flash it with `python firmware/tools/flash_unit.py --role wearable --port COM30` (or `flash_node.py --sketch wearable`) and re-run the two-cot play before treating it as bench-verified here. |
376 | **O-1g** | **The microphone has a close-range blind spot, and the runbook's level advice does not protect against it.** | Measured 2026-11-20: a phone held close to the INMP441 drove the window to the rail (cot 1 peak **32767 of 32767**, 29 samples saturated; cot 2 **32637** and **32736**), and every such window is refused at `link_stage.c:471` before the model's score can matter. The consequence was observed, not theorised: a cry played at **cot 2** was announced as **cot 1**, because cot 2's own windows were refused and the bystander cot's clean window — hearing the same cry across the table — announced first. **The runbook's "aim for 0.10–0.20 rms" is unsafe advice**, because all three refused windows sat inside that band: the 2 s window includes the clip's quiet lead-in and tail, which drags rms down while the peak stays at the rail (measured window crest 6.1–7.3). **The instrument that matters is peak, not rms**, and the fix is distance — peak falls roughly as 1/d, so 1.5–2× the phone-to-mic gap puts these windows near 0.5–0.6 with margin. **A real infant crying directly at the microphone lands in the same dead zone.** The refusal is correct (the INMP441 is fixed-gain and flat-tops before the node's 16-bit window does, so the harmonics the model reads are already gone), but it is a genuine limitation only placement or a gain change can address. **Recorded, not fixed:** the clip gate is a bare peak test, so **one sample out of 32,000 discards a whole 2 s window** — a count- or fraction-based gate would tolerate an isolated transient. Any such change needs the model scored on lightly-clipped windows first, then a branch, a bench test and a hub reflash. |
377 | **O-1c** | **The node's I2S watchdog is committed but not flashed** — it goes onto a node once the soldering is finished (D-79). | The owner is soldering the final wiring, and a node should not be reflashed mid-solder. The fault-injection test proved the code executes and refills a window, but not that it recovers the original physical fault, and it detects *exact* zeros only. After it is flashed it needs a played-cry end-to-end run to confirm the node still streams and the hub still hears it. |
378 | **O-2** | **The model has never heard a real cry.** | The only cry source available in this room is a *recording played through a phone speaker* — compressed, band-limited and EQ'd by the speaker. It is labelled as such. On that audio the model's AUC is 0.9283 with recall 0.724 and FPR 0.000; the misses are two-second windows landing in the gaps between cries. The negatives (this room, this microphone) are authentic. ~50 self-recorded real clips remain the single biggest accuracy win. |
379 | **O-3** | **The screen has been seen three times; one photograph is still worth taking, and it has to be taken with the room lit.** | The first photograph confirmed the layout, the text and the orientation (I-31, I-35); the second confirmed the R/B fix; the third was of the colour-check page, and it confirmed every hue (D-56); the fourth was of the ramp, taken in the dark again, and it could not measure the steps - the red channel even jumps the wrong way between adjacent steps - which confirms I-38. But it did its job as a *chooser*: the user picked the third ramp step off it as the teal the screen should be (D-57). What is still wanted is one photograph **with the room lit**, so the ramp yields the panel's own tone curve instead of a colour choice. |
380 | **O-4** | **The tensor arena is still in PSRAM.** | Moving it into internal SRAM is the obvious next optimisation. At 101.8 ms of inference per two-second window it is not urgent for a triggered monitor. |
381 | **O-5** | **The in-domain audio is speaker-played, not real.** | A real cry in this room, recorded with `node_capture`, is the honest test of whether the system works in the real room — and the dataset the model should be fine-tuned on. |
382 | **O-6** | **The distance curve (how far the phone can be) is still open.** | The phase cues did not line up in the earlier attempt, so the far-side measurement has to be repeated. |
383 | **O-7** | **The resend mechanism has not been exercised under a genuinely hostile channel.** | It is verified against the hub's own Bluetooth advertising, which is the real interference in this design, but not against a busy Wi-Fi environment with a second router on the same channel. |
384 | **O-8** | **The deployed model false-alarms on 8.19 % of loud music, and the fix is a training round that has not been run yet.** | GTZAN is now in the training corpus and the notebook is patched, regenerated and validated locally (D-51, I-29), but the re-run happens on the user's Kaggle account. **RESOLVED 2026-11-01 (D-58):** the re-run happened and the retrained model is deployed and verified on the board; music false alarms are 2/965 = 0.21 %. The remaining cost is the phone-speaker per-window recall (D-58), and the honest in-room number is still O-9's to produce. |
385 | **O-9** | **The confusion session — the in-room measurement of speech, music, television and the phone-played cry — has not been recorded.** **RESOLVED 2026-11-01 (D-60):** it has been recorded — nine one-minute takes on the real INMP441, and the honest in-room number is **0 false alarms in 308 windows of non-cry audio** (speech, music and film, including a loud film and loud music where the trigger fired on 46 % and 84 % of the records) and **86.4 % per-window recall on the phone-played cry at 15 cm**. The one thing the session also found is that the cry at one metre never reaches the model's trigger at all, so the system's reach is limited by `CRYNET_TRIGGER_RMS` and `CRYNET_TRIGGER_RMS` and the microphone's fixed sensitivity rather than by the classifier. | `research/CONFUSION_SESSION.md` has the exact commands; the measured table and the caveats are in `research/in_domain/SESSION_REPORT.md`, the per-session numbers in `research/out/eval_session*.json`. |
386 |
387 | **D-96** | **The hub's cry log is opened, read and written in the way SPIFFS actually supports, and the numbers in the file are the truth when the counters and the file disagree.** The log is created with `O_CREAT \| O_EXCL` and falls back to a plain open when it is already there; a read at an offset is a seek and a read, and a write at an offset is a seek and a write; and at boot the highest number in the tail of the file sets the next number to hand out. | Three facts about the flash were found the hard way, on the board, and each of them had been quietly wrong since the log was first written. **(1) ESP-IDF's VFS drops a bare `O_CREAT`:** `spiffs_mode_conv` only honours it when it arrives with `O_EXCL` or `O_TRUNC`, so `open(path, O_RDWR \| O_CREAT)` on a file that is not there fails with *No such file or directory* — on a **freshly formatted board, which is every brand-new unit flashed from scratch**, the hub could not create its log and went on running with **no record at all**, saying nothing. **(2) SPIFFS has neither `pread` nor `pwrite`:** the VFS answers both with *function not implemented* (`CHECK_AND_CALL` returns `ENOSYS` and there is no emulation), so the counters could not be written (`the counters did not write: Function not implemented`) and **no record could ever be read back** — four separate sites, so the hub could write a record and never read it, which no test on the host could ever have caught because the host has a real filesystem. **(3) The counters file and the records file could fall out of step** when a power cut landed between the two writes, and the next number handed out would be handed out twice; the file's own numbers are the truth, so the tail of the file now sets the next number at boot. All three are proven on the board: `boot 16: 16 records kept, 16 still to send, next number 17` across a reboot, with the record surviving. |
388 |
389 | **D-97** | **The hosted dashboard is `https://amazings-crynet.amazingadelowo.workers.dev` — the Worker is named for the system, and it sits on the account's own `amazingadelowo` subdomain.** | The owner's Cloudflare account carries the subdomain `amazingadelowo`, and the Worker's name is the one the owner asked for, *Amazing's Crynet*, which becomes the label in the address. **The two secrets are separate and neither is in this repository:** the caregiver logs in to the page with HTTP Basic (`caregiver` plus the dashboard password), and the hub posts with `X-CryNet-Token` plus the upload token. **The live edge certificate was checked against the hub's own root bundle** — Cloudflare's `workers.dev` edge presents a Let's Encrypt chain that ends at **ISRG Root X2**, which the hub's 145-root bundle already carries, so `net set https://…` needs no firmware change and no new certificate; the handshake

```
negotiates TLS 1.3. |
390
391  | **D-98** | **The station interface is created before the radio starts; the console's `log` word no longer swallows `logtest`; and a refused upload is reported with its HTTP status.** | Three faults found on the bench, in one session, all after the dashboard went live. **(1) The hub joined the Wi-Fi and never got an address** — `dhcp client start failed` — and an lwIP-debug build gave the real reason: *Cannot use this netif with DHCP: MTU is too small*. The station netif is only added to lwIP when the driver reports that the station has started, so a netif created after `esp_wifi_start()` never joins the stack and keeps MTU 0, which fails DHCP's first check (`mtu < 576`, `ERR_MEM`). `node_link` now calls `esp_netif_create_default_wifi_sta()` **before** `esp_wifi_start()`, and `crynet_net` looks the interface up by its if-key (`WIFI_STA_DEF`) instead of making a second one — verified on the board: offer and ACK from the router, `sta ip: 172.20.10.2`, the clock set, **68 records sent in 3 batches, 0 failures**, and later 2 more on the clean image. **(2) `logtest` had never actually run** — the console matched `log` as a three-letter prefix, so `logtest` printed the log state and returned; the prefix is now the whole word or the word plus a space, and the self-test finally says `logtest: PASS -- a cry that finishes out of order is still sent`, on the chip. **(3) A 401 from the dashboard arrived as `ESP_ERR_NOT_SUPPORTED`**, because the HTTP client answers a challenge it has nothing to answer with; the uploader now reports `HTTP 401 (ESP_ERR_NOT_SUPPORTED)` so the status itself is visible. `hub_console.py` was rewritten around the two facts the board taught: opening the port can reset it, and the ready banner only prints at boot — it now waits for signs of life, confirms every command by its echo, and retries what is not echoed. |
392
393  | **D-99** | **The dashboard is named for the system and wears the hub's teal; a toggle switches the glass between dark and light, and both palettes passed the validator.** | The owner asked for three things: the name on the hub's title card (*Amazing's CryNet*, not just CryNet), the teal scheme the hub display has always used, and a dark/light switch. The dark theme's surfaces are now the hub panel's own RGB values from `hub_display.c` (C_BG `#020404`, C_SURFACE `#070f10`, C_SURFACE_HI `#0c1a1c`, C_TEXT `#e4f0ec`, C_DIM `#7a9692`), and the hub accent `#2dd4bf` is the chrome accent — while the **data** marks obey the validated band: the hub accent fails the dark data band (OKLCH L 0.785 > 0.67), so dark slot 1 is `#23a893`, the nearest passing step of the same hue, and both modes re-ran the palette validator against their true surfaces — light PASS (worst adjacent CVD 9.1, normal-vision 19.6, contrast WARN on aqua/yellow/magenta covered by the shipped relief channel: table twins and written "Cot n" labels), dark all PASS. The switch is a moon/sun pill stamped on `<html data-theme>` before first paint from `localStorage['crynet-theme']` (falling back to the OS setting, which the page follows live until a choice is made), with charts repainted from CSS custom properties and the heatmap's number ink flipping direction with the ramp. Verified in the browser with seeded data and deployed (worker version `39b339f5`); the auth realm now greets the browser with the system's name too. |
394
395  | **D-100** | **The hub's panel carries a wireless mark, and it is drawn only while the hub is on the network — green when it is, and nothing at all when it is not. (Superseded in part by D-103: the mark was first drawn in both states, green and dim grey.)** | The owner asked for "a sort of wifi icon to show on the hub display when it is connected to the wifi", and the panel now has one between `S3 HUB` and the `NODE` pill — the two things the hub depends on, side by side. It is **drawn only when the link is up**: the mark means one thing, and that thing is *the hub is on the network* — the same thing the uploader's own gate means, so the glass cannot disagree with the network. (D-103 reverses the earlier decision to draw it in both states; the geometry below is unchanged.) There is no glyph for it in the panel's 5x7 font and no room for a picture, so it is drawn from the panel's own pixels: a three-pixel dot for the router and two two-pixel arcs over it, walked **by angle** rather than by column — a column-by-column walk of a shallow arc leaves gaps at the sides, and the sides are where the eye looks first. The arcs were checked as ASCII on the host before being committed, at a size that leaves the mark 151–169 px across in a 320-px header. The state comes from `crynet_net_status().wifi_up`, the same flag the uploader uses, so the mark cannot disagree with the network. |
396
397  | **D-101** | **The hub says who is here, once a minute, whether or not there is anything to record — so the dashboard can tell a quiet baby from a dead hub.** | The owner asked to see the online/offline state of the hub and the nodes on the dashboard. A cry record cannot answer that: it only says what was heard, so a dashboard that learns of a hub *when a baby cries* cannot say whether a quiet week means a quiet baby or a hub that has stopped working — and that is the question a caregiver actually opens the dashboard to ask. The cots have no network of their own (they talk to the hub over ESP-NOW on channel 6), so the hub is the only thing that can answer for them, and it now posts a presence — `{device, hub, nodes:[{cot, online, deaf, rssi}]}` — to `POST /api/v1/presence` **every minute, and immediately whenever a cot's state changes**, so a cot that has just dropped off the air is not a minute stale. The service stores one row per cot (replaced each time) and answers `GET /api/v1/presence` with the hub list, each hub's age, and each cot's state; a hub is taken to be there if it has spoken in the last three minutes — three missed chances, so one dropped request does not make the caregiver think the hub has died. **The owner still configures one address**: the presence endpoint is derived from the events URL by replacing its last path segment, so `net set https://…/api/v1/events` needs no second setting. The page shows it as a strip of pills under the numbers ("Here · ● Nursery hub online · ● Cot 1 online −47 dBm"), with the state carried by a word as well as by the colour. The demo data seeds a presence for the demo hub, and the demo hub is held online because it is a made-up hub with nothing posting it — a demo that showed the panel working when it is not would be the one thing a demo must not do. |
398
399  | **D-102** | **The page reads like an instrument and not like a paragraph: the copy is cut to the facts, the daily curve is drawn smooth, the bars carry an edge, and `hidden` now means hidden.** | The owner asked for the dashboard to be cleaned up — "I don't want it looking AI generated, I want that thought through and built feel to it" — and for two changes to the charts. **The copy.** The strapline "the cry record" is gone from the title and the header, and every subtitle, note and empty state was cut back to the fact it carries: the trailing reassurance ("A quiet stretch after that does not mean a quiet baby — that cot was not being watched.") is now the shorter true statement ("that cot was not being watched"), and the notes that restated what the chart already showed were removed. Nothing was removed that the reader needs: the untimed-record rule, the night boundary and the deaf-cot warning all stay. **The curve.** The sparkline of daily cries was a polyline, and a corner between two counts says something that is not there — that the count jumped and stayed put. It is now a cubic spline whose tangents are chosen the **Fritsch–Carlson** way, so the curve turns at a point and never past it: a plain Catmull-Rom spline dips *below* the baseline between two quiet days, and a chart of counts that draws a negative count is lying. **The bars.** They are now paths with a rounded top and a square foot (the foot stands on the axis, so it may not lift off it) and a one-pixel edge in the axis colour, so a bar on a pale grid still has a boundary the eye can find. **A real bug found on the way.** `[hidden]` did nothing on any element with a `display` set by a class — the browser's own `[hidden] { display: none }` is outranked by an author rule like `.control { display: flex }` — so the **Hub** dropdown was on the page with a single hub, when the code had always meant to hide it. One line, `[hidden] { display: none !important; }`, makes `hidden` mean hidden everywhere; it also stops the new presence strip from being unable to hide itself. |
400
401  | **D-103** | **The wireless mark is drawn only while the hub is on the network, and it was drawn big enough to read at a glance; the upload token lives off Git.** | Two corrections to D-100, one of them the owner's. **The mark now has one meaning.** D-100 drew it in both states — green on the network, dim grey off it — on the reasoning that a mark which appears and disappears cannot say whether the network is down or the panel is asleep. The owner answered that directly: *"I'd rather have the wifi icon show only when connected."* He is right for this panel, and the disagreement with the earlier reasoning was the reason to think about it again: on a 320×240 screen of a 5×7 font there is no room for a second legend, and a mark that is always present has to be **read** to be understood — grey means "not connected", which is a second thing to learn beside the two pills. Drawn only when it means something, the rule is one line long — *a mark here means the hub is on the network* — and the flag is the same one the uploader uses to decide whether a record can leave the hub, so the glass cannot disagree with the network. Where the hub is not on the network, the dashboard says so instead, and there the question has room to be asked and answered. **The mark was also too small to see.** It was drawn at
```

radii 6 and 10 over a 35°–145° span; it is now radii 8 and 13 over 40°–140°, with the router dot and the arcs two pixels thick — 21 px wide and 14 px tall in a 320-px header, between `S3 HUB` and the `NODE` pill. The geometry was measured against the panel's own constraints (the free space between "S3 HUB", which ends at x≈126, and the `NODE` pill at x=196) rather than chosen by eye, and the host mock was proved **pixel-identical** to the firmware by emulating C's float32 `cosf`/`sinf` in Python — no host C compiler was available, so the equivalence was proved rather than assumed. **And the mark was not visible on the bench for a third reason, which was not the firmware's:** the board was running the pre-branch image, in which the glyph did not exist at all. **The token stays off Git.** This repository is public, so the upload token is in exactly three places and no more: Cloudflare's secret store (`UPLOAD_TOKEN`), the hub's own NVS (`net token`), and a local, **git-ignored** `dashboard/.secrets` (`CRYNET_TOKEN`, `CRYNET_PASSWORD`). The local file is what lets the offline dashboard be started against the same token without typing it, and `dashboard/worker/README.md` now says where it is and how to change it. D-97's rule — *the two secrets are separate and neither is in this repository* — holds. |

| **D-104** | **The hub's connection is checked as fast as the three links in the chain allow: the hub reports every 15 s, the service calls a hub gone after 45 s, the page asks every 10 s — so a hub that is switched off reads as offline in about a minute.** | The owner switched the hub off and the dashboard went on saying *online*, and asked for the hub's connection to be checked "as often as possible". **When the dashboard decides a hub has gone is the sum of three delays, and no one of them can be shortened alone:** how often the hub speaks (was 60 s), how long the service waits before calling a hub gone (was 180 s), and how often the page asks (was 30 s) — **46 s to 270 s from the plug being pulled, measured on the old numbers, and the owner was standing there watching it say *online*.** All three were shortened, keeping the rule the presence was built on — *three chances missed before the dashboard calls a hub gone* — just three times faster. The hub now reports every **15 s** (5,760 requests a day against the free tier's 100,000; the real cost is a TLS handshake each time, a few hundred milliseconds of radio a day, on the same channel the cots use, which the budget affords). The service now waits **45 s** (`PRESENCE_STALE_S`, in both `server.py` and the Worker). The page now asks for the presence **every 10 s on its own clock**, without dragging the nine reads of the record along with it — cries do not arrive every second, and *is the watcher alive* is the question the owner actually opens the dashboard to ask — and asks again the moment the tab comes back, because a browser may have throttled the timers in the background. It also refuses an answer older than the one it has, so a fast read that returns late cannot say the hub is back. **Two faults were found while measuring, and both were real.** (1) **The presence strip was inside the overview body, and the overview body is hidden when there is nothing to show** — so the one element whose whole job is *is the watcher alive* vanished precisely when the record was empty, which is exactly when the question is asked. It is now outside the body, at the top of the overview, and drawn from `render()` unconditionally. (2) **A request that never came back froze the page on its last answer for ever.** `refresh()` refuses to run while `state.busy` is set, and nothing cleared that flag if a fetch hung — so one half-open connection meant the page never polled again, and a hub that had gone offline would read as online indefinitely. Every read now has a 15-second deadline and the guard has a 60-second one. **Measured with the new numbers:** the service's deadline passed at `1795575496` and the page said *offline* at `1795575498`, on a 10-second poll — so the whole chain now lands in **roughly a minute**, where the old chain took up to four and a half. The Worker was redeployed (`ef3ec161`) and the hub on COM32 was flashed with the new image. |

| **D-105** | **The presence yields to a cot's burst, and its rate is set by the cots' needs rather than the dashboard's: the hub reports every 30 s, a report may wait up to 20 s for a burst to finish, and the dashboard calls a hub gone after 90 s — so nothing costs the cots anything, and a hub that is switched off still reads as gone in about two minutes.** | The owner, having asked for the connection to be checked "as often as possible", added the constraint that actually decides it: *"if this is going to affect the cots or any system performance, you can scrap it or increase the interval or whatever, i don't want anything affecting the system."* **The physics is the link stage's own finding, written down when a cot's "offline" clock had to be taught to run on the time the radio was open: while a burst is on the air the Wi-Fi receiver hears nothing at all — not that burst's own fragments, and nothing else.** A presence report is a TLS handshake, which takes the radio for a moment; at 15 s it would land inside a burst roughly one time in three, each time costing fragments and a round of resends. So the 15 s figure was withdrawn and three things were done instead. **(1) The hub reports every 30 s**, not 15 — 2,880 handshakes a day, twice the original rate, half the rate of yesterday's. **(2) The presence *yields*.** The link stage hands the network stage one function — `crynet_net_set_radio_guard()` — which answers whether a burst is in flight (`st->requested`), and the report waits for it to finish rather than take the air from under it. The wait is **capped at 20 s**, because a hub that has been switched off must still be reported gone: a report may be twenty seconds late, it may not be missing, and the worst gap between reports — 50 s — stays inside the dashboard's 90 s idea of how long a hub may go without speaking. **(3) The dashboard's window went to 90 s** (three reports missed, the same rule of three), in `server.py` and the Worker alike. **The cap is the guard's safety, proved by a host model of the scheduler:** radio never busy → a report every 30 s; one long burst → the report waits and posts at the cap (50 s); near-continuous bursts → 34 s; and **a guard stuck saying "busy" for ever still posts every 50 s** — so the new code cannot make the dashboard think the hub has gone away. The record upload was left alone: it has always contended with bursts and the whole chain was verified with it running. The page keeps its own 10-second clock — it asks the *service*, so it costs the cots nothing. The numbers: a hub switched off now reads as gone in **about 1.7–2.2 minutes**, against 2–4.5 minutes before this. The knob is one line (`PRESENCE_EVERY_S`) if the day's end-to-end tests say otherwise; what to watch is the link stage's own counters — `resends_sent`, `gap_ewma_ms`, `burst_failures` — with the presence posting and with `crynet_net_set_radio_guard(0)`. The Worker was redeployed (`c4f78eb4`) and the hub on COM32 was flashed with the new image. |

| **D-106** | **A unit's boot banner is the identity check — and the banner was lying, because the build inherited its version stamp from a stale configure; the reported restarts were power-class resets, not crashes; and on a clean-power boot the hub ran ten minutes with zero restarts, zero cot flap, live telemetry on both cots and 15 presence posts at the exact 30 s cadence, so the connectivity feature stays.** | The owner reported three things at once: *"the hub keeps restarting"*, *"cot 1 going offline for a split second"*, and *"cot 2 rms freezing on one value longer than normal"* — and restated the standing constraint in its strongest form: *"if this connectivity status is going to cause an issue, you can scrap it or make it in a way it doesn't affect system performance at all."* The diagnosis, in the order it fell out. **(1) The banner lied.** The hub's boot banner read `bd3d6fb-dirty`, built Nov 24 06:31:16 — three features behind the branch tip — yet its behaviour was the current code: presence posts marched past at the exact 30-second D-105 cadence. **ESP-IDF bakes the app version and the compile time at configure time, and scons had reused a Nov 24 configure for every build since**, so fresh code wore a stale label. Flashing the committed prebuilt reproduced the identical banner byte-for-byte — and catching that lie disproved the stale-image scare: the hub had been running the D-105 code all along. **(2) The restarts were power-class, not crashes.** No panic text was captured anywhere; every reset observed reads power-on (`rst:0x1`). The CH343 USB bridge resets the board whenever a serial monitor opens the port, and the boot counter's four bumps during the diagnosis (97 → 101) are exactly the two capture-opens and the flash session — the instruments' resets, not the hub's. The morning's cot-1 flapping and cot-2 frozen RMS came on a boot during the owner's own unplug/replug tests — the same power fiddling behind *"it still says online"* — and did not survive a clean-power boot. **(3) The watch that settles it: ten minutes, one radio, both jobs.** 600 s of serial: **zero restarts, zero `is offline` flaps, node 1's window advancing (seq 42 → 63), node 2's telemetry live (seq 40 → 45, RMS moving), and 15 presence posts at exactly 30 s** — the presence feature and the cots' link coexisting on one radio, which is the answer to the constraint: nothing needs scrapping. **(4) The fix is to the build, not the firmware:** a release build now forces a reconfigure so the banner is regenerated with the source (an incremental build alone keeps the old stamp), `firmware/prebuilt/` was repackaged — the banner reads `3acaf99`, built Nov 25 — and the hub on COM32 was reflashed from it. **The rule for every future check: when a unit's banner disagrees with `git log`, suspect the build cache, not the flash** — and if a restart ever recurs, its reset reason now names itself against a banner that tells the truth. All checks pass. Nothing pushed, nothing merged. |

| **D-107** | **D-106's "the restarts were power-class, not crashes" was wrong: the very next capture caught the hub dying mid-run with a stack overflow in task main — the whole live system runs on one 3,584-byte stack that ESP-IDF sized for an `app_main` that**

returns; the stacks are now sized for what actually runs on them; and the cot's "offline for a split second" is a 0.1-second staleness blip that loses nothing.** | The corrected image from D-106 was on the hub and the 600-second watch was still running when it caught what every earlier window had missed: at 502.84 s, `***ERROR*** A stack overflow in task main has been detected`, backtrace corrupted, `rst:0xc (RTC_SW_CPU_RST)`, and the hub booted itself again — banner, version and all, honest. The ELF SHA in the panic (`72d2f7266`) matches the flashed image, so this is the current code, and the crash log is kept at `research/out/hub_stack_overflow.log`. **What D-106 got wrong, and why:** its conclusion rested on absence — no crash text had been *captured*, and the boot counter's bumps had been attributed to the instruments. The attribution was a plausible fit, not a proof, and at least one of those restarts was the hub dying on its own. The rule that replaces it: **absence in a capture window is not absence, and a boot-counter argument needs the instruments' resets counted and the rest owned as suspicious.** **The root cause:** `app_main` never returns in this firmware — `link_stage_run()`'s `for(;;)` runs the telemetry, burst handling, front end, inference, logging, BLE notify and display *on the main task*, whose 3,584-byte stack was ESP-IDF's default for an `app_main` that spawns workers and returns. The sdkconfig comment even said "larger task stack for the inference task" — and no config line stood under it. **The corrupted backtrace cannot name the deepest frame**, so the fix is not tuned to one call path: `CONFIG_ESP_MAIN_TASK_STACK_SIZE=16384` (12.8 KB more internal RAM against ~296 KB free at boot, and nothing at runtime), plus the net task 6,144 → 8,192 for headroom of its own — its HTTPS posts run a TLS handshake on that stack, and the owner's standing rule is that the connectivity feature may not cost the system anything, so its marginal stack is bought out too. **What the overflow is not:** the crash came 0.64 s after a presence post, which invites blaming the connectivity feature — but the post runs on the `crynet` net task with its own stack, so it did not eat main's; 16 of the capture's 17 posts landed at exactly 30 s and one took 33 s because the D-105 radio guard made it *wait* for a burst, which is the design working. Presence stays. **The cot symptoms, explained at last:** "offline for a split second" is the link layer's own offline/online pair 110 ms apart when the staleness tick lands just before a frame that was already in the air (twice in 600 s, zero frames bad across either); "rms freezing" is telemetry-less frames carrying the last frozen value between telemetry updates. Both cosmetic; no firmware change. Rebuilt, repackaged, reflashed (banner `dabb4e0-dirty`, boot 108), and a fresh 600-second soak ran clean on the new image: zero resets, zero offline lines, **a 286-fragment burst — the size that preceded the crash — completing 286/286 in 0.98 s at 451.9 s**, 20 presence posts on cadence, node 1 seq 217 and node 2 seq 231 live at 600 s, and 11,974 frames off the air with 0 bad and 0 failed. One clean soak cannot name the frame the corrupted backtrace hid; it shows the workload that killed the old stack running with an order of magnitude of headroom. Nothing pushed, nothing merged. |
409 | **D-108** | **The presence panel could hold a dead hub's cots "online" for ever: the hub's own staleness clock was never applied to the cot rows the hub leaves behind, so the last post before a power cut outlived the hub that made it — a cot is online now only while the hub's word about it is fresh, in both the Worker and the offline copy, with the contract test proving the fix against the old behaviour.** | The owner powered the whole system down and found the dashboard saying the hub was offline while both cots read online — a contradiction, because the cots have no network of their own: the hub is the only thing that can answer for them. The cause was one missing rule. In `build_presence()` (both `dashboard/worker/index.js` and `dashboard/server.py`), the **hub** is judged by the D-105 staleness window (90 s since its last word), but a **cot** row's `online` flag was served verbatim from whatever the hub last posted — no freshness check on the row's own `at` stamp. So the hub's final presence post, made seconds before the power was pulled, said `online:1` for both cots, and that answer was served for ever: the panel whose one job is *"was the baby quiet, or did the watcher stop?"* answered **watched** while nothing was watching. **The fix is one rule, applied in both copies so they still agree field for field: a cot is online only if the hub said so *and* the saying is fresh** — `at` inside the same 90 s window the hub itself is judged by; a row with no stamp reads offline (default closed); the demo hub keeps its by-design exemption, because a made-up hub has nothing that can go offline. The page needed no change: it renders what the service says, and the service now tells the truth. **Proved twice.** `test_dashboard_contract.py` now posts the firmware's exact presence body (`build_presence()` in `crynet_net.c`, byte for byte, as `hub_body()` mirrors `build_batch()`), reads both cots online, then moves the store's clock ten minutes past the window — the only way a test can pull a plug — and demands the cots read gone with their hub: **against the old code it fails with precisely the owner's report** (`cots [1, 2] read online long after the hub that sees them stopped speaking`), against the new code it passes. And the Worker was redeployed (version `d6deaeda`) with the live store confirming the diagnosis: both cot rows still carried the hub's last words with `at` **2,251 s old against the 90 s window** — exactly the rows that produced the lie, now aged out, so the next 10 s poll of an already-open page shows all three offline with no reload. (The local, git-ignored `dashboard/.secrets` records the upload token but its password entry is empty, so the live endpoint was verified through the store itself rather than a browser login; if the owner wants the offline copy to authenticate against the hosted one, `CRYNET_PASSWORD=` wants filling in.) No firmware change, so `firmware/prebuilt/` is untouched and nothing was flashed — every unit is off. Nothing pushed, nothing merged. |
410
411 ---
412
413 ## 5. How to reproduce the results
414
415 ```bash
416 # 1. the model, in the cloud
417 #   open notebooks/crynet_training.ipynb and run it on a Kaggle GPU
418 #   (regenerate it with: cd research && python build_notebook.py)
419
420 # 2. the host-side checks
421 cd research
422 python make_stub.py && bash run_stub.sh   # synthetic smoke test
423 bash run_real_fast.sh   # real corpus, subsampled, CPU
424 pip install ziglang && python test_c_parity.py # C <-> Python parity
425 python verify_int8_parity.py   # INT8 <-> float32 parity
426 python eval_indomain.py   # the model on this room's audio
427
428 # 3. the firmware
429 cd firmware/hub_infer && python -m platformio run   # ~1 min incremental
430 python firmware/tools/flash_hub.py --port COM32   # flash and read the log
431 bash firmware/tools/fetch_deps.sh   # re-fetch the pinned deps
432
433 # 4. the node (Arduino IDE, board WeAct Studio ESP32C3, USB CDC on boot)
434 arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3 firmware/node/node_crynet
435 python firmware/tools/verify_burst.py --port COM30   # the node's window, as a WAV
436 python firmware/tools/collect_clips.py --port COM30 --out research/in_domain/session1
437
438 # 5. the live end-to-end test
439 python firmware/tools/flash_hub.py --port COM32 --log-only --seconds 150
440 python firmware/tools/ble_alert_test.py --listen --seconds 140   # the phone
441 ```
442
443 Every run's log is under `research/toolchain/` and every run's conclusion is
444 in `SYSTEM_UPDATE.md`'s change log.

The handoff briefing (frozen contracts, file map, versions)

PROJECT_HANDOFF.md

322 lines · 24,915 bytes

```
# CryNet — project handoff (paste this into a new chat)

> **Purpose of this document.** It is a complete, self-contained briefing for a fresh chat that has no memory of this project. Everything needed to continue — hardware, decisions, dataset research, the exact signal contract, validation evidence, file map, how to run, known risks, and the next checkpoint — is below.
>
> **Repo:** `C:\Users\HP\.t3\projects\fp-two-node` (git, branch `main`, remote `origin` → `https://github.com/Amazo123/CryNet-TwoNode.git`) — the **two-cot** system.
> **Sibling repository:** `C:\Users\HP\.t3\projects\fp-2` (remote `crynet` → `https://github.com/Amazo123/CryNet.git`) — the **three-cot** system; the two share history up to commit `99bb257` and then diverge.
>
> **Status at handoff:** the two-cot system is on the bench and proven end to end (both cots detected, named and buzzed on the wrist, then cleared). The owner has started soldering the final wiring. The node's I2S watchdog is committed but **not flashed** — it waits until the soldering is finished. The multi-cot wearable behaviour is held on branch `feature/wearable-multi-cot` until two cots can cry at the same time.

---

## 1. The project

A **multi-node infant cry monitor** for multi-crib environments (NICU, maternity ward, daycare, twin nursery).

### Hardware (fixed — this is a final-year project with real boards)

| Subsystem | Board | Role |
|---|---|---|
| **Cot nodes ×2** | ESP32-C3 **Pro Mini** + INMP441 MEMS mic | I2S capture at 16 kHz, DC-coupled RMS telemetry over ESP-NOW, on-demand 64 KB audio burst back to the hub |
| **Central hub** | **ESP32-S3-DevKitC-1** (dual-core Xtensa LX7 @ 240 MHz, 512 KB SRAM, **8 MB octal PSRAM**) | ESP-NOW receiver, DSP front end, **TFLite Micro inference**, TFT display (ILI9341 320×240), BLE GATT server |
| **Wearable** | ESP32-C3 SuperMini + SSD1306 OLED + BC547 haptic motor | Caregiver alert on the wrist via BLE notification; the OLED carries the state (LINK/SCAN), there are no status LEDs |

**The hub is the target device for everything below.** It has 512 KB internal SRAM and 8 MB PSRAM; the model must fit in flash and the tensor arena in RAM/PSRAM.

### Why the rebuild happened

The previous build (documented in the original 2.0.0 spec attached at the start of the project) **did not work after the firmware was uploaded** — the project stalled after flashing to the S3 and one C3 node. The decision was to restart **from model training**, discarding the old model's mechanism but keeping the hardware and the system architecture (3 cot nodes → hub → wearable).

### The three architectural decisions (made with the user, now frozen)

1. **Binary cry / no-cry head.** Not cry-cause classification. The existing BLE alert packet (`wearable_alert_packet_t`: `cot_id`, `alert_type` 0x01 = genuine cry, `confidence_pct`, `event_id`, `checksum`) only supports a binary decision.
2. **Label policy = "any cry present".** A 2-second window counts as positive if it contains *any* real cry (≈ ≥ 3 active frames), not only sustained crying. Rationale: the hub fires on an RMS spike and then grabs a 2 s buffer that usually begins just before the cry.
3. **Training runs on Kaggle GPU** (the user runs it); the notebook is validated locally first by the agent.

---

## 2. The dataset research (this was the core of checkpoint 1)

The approach was **not** to trust dataset descriptions. Every candidate was downloaded and profiled at the audio level: sample rate, duration, RMS, autocorrelation F0, spectral centroid, spectral flatness — enough to separate a real cry from a synthetic one or from renamed filler.

### Used

| Corpus | Role | Evidence |
|---|---|---|
| **ICD — `hoyyu1/infant-cry-detection`** (Hugging Face) | **PRIMARY** | 28,478 clips, *all* 16 kHz mono / 5.00 s, 5 pre-assigned balanced folds. Cries: EnesBabyCries 5,026 · CryCeleb2023 4,552 · iFLYTEK 2020 "xunfei" 3,752. Negatives: VoxCeleb speech 4,963 · DASEE domestic 4,506 · human non-speech 2,076 · **cat meows** 1,953 · silence 1,650. Manifest ↔ archives verified 1:1 (13,330 positives, 15,148 negatives, zero missing, zero extra). |
| **Baby Cry 4-class — `karimeletriby/infant-cry-classification-4-classes`** (Kaggle) | cry diversity + cross-corpus test | 4,871 real cries, 16 kHz mono WAV, −23 LUFS normalised, high-pass 50 Hz, VAD-trimmed, **cross-dataset deduplicated** (1,297 near-dupes removed). Classes: `needs` 3,103 / `physical_pain` 774 / `scared` 684 / `burping` 310. |
| **ESC-50** (Kaggle mirror `akaliutau/esc-50`) | negatives + cross-corpus test | 2,000 clips / 50 classes @ 44.1 kHz. 1,960 used as negatives (`crying_baby` excluded); the **40 `crying_baby` clips are held out entirely** as an unseen cross-domain cry test. |
```

| **Snoring — `tareqkhanemu/snoring`** (Kaggle) | hard negatives | 1,000 adult-vocal recordings — the confusion class a nursery monitor hits constantly. |

Rejected, with reasons

* **`mennaahmed23/baby-cry*` (five Kaggle mirrors) — CONTAMINATED.** Its `silence` class is not silence (measured RMS 0.22, peak 1.0, a constant 1225 Hz component); its `noise` class is renamed ESC-50 files (e.g. `1-100032-A.ogg`); hundreds of files are synthetic `generated3_*` derivatives. Training on it teaches dataset artifacts.
* **`rahult25/five-class-infant-cry-audio-datase`** — same base corpus plus pitch-shifted derivatives (`uncomfortable_10 … _72`).
* **ICSD (`QingyuLiu1/ICSD`)** — the best-curated corpus in existence (3,641 weakly + 807 real strongly-labelled clips with event timestamps) but **gated on Hugging Face**, needs a token and manual approval.
* **`AxonData/infant-cry-detection`** — 50 h of expertly labelled domestic recordings, but the Kaggle upload is an 18-file teaser and the full set is paid.
* **Baby Chillanto (INAOE)** — 2,267 high-quality 16 kHz newborn cries, but no clean public mirror (request-only from the lab).
* **AudioSet / FSD50K** — need YouTube extraction or a ~30 GB pull for a handful of clips ICD already covers.
* **DonateACry** — used *only* indirectly, because it is already merged (and deduplicated) inside the Baby Cry 4-class corpus. Raw DonateACry is 8 kHz with a badly skewed class distribution (382 of 457 files are `hungry`).

Two findings that shaped the design

1. **Level asymmetry.** ICD cries sit at **RMS ≈ 0.006** while ICD negatives sit at **RMS 0.3–0.46**. A model could learn "quiet = cry" and then collapse in the real room. → the front end applies **per-window standardisation (CMVN)**, making the representation level-invariant. (Also why the silence gate had to be lowered to 0.0015.)
2. **The old trigger threshold was a real bug.** The previous hub fired an audio pull at **RMS ≥ 0.18**, so a *soft* cry never reached any classifier at all. The new design feeds the model anything above ~0.03 RMS and lets it reject false triggers — rejecting speech, cat meows and household noise is exactly what it is trained for. **This requires a firmware change in checkpoint 2.**

---

3. Technical design

Signal contract (frozen — the firmware, the C code and the weights all depend on it)

```
int16_t pcm[32000]                2.000 s @ 16 kHz mono
  -> float x = pcm / 32768.0
  -> subtract mean                DC removal (INMP441 sits at ~+18000 counts)
  -> zero-pad 256 front / 512 rear
  -> 81 frames, 512-pt symmetric Hann window, hop 400
  -> |rfft|^2                     257 bins
  -> 64 Mel bands                 librosa slaney mel, 20 Hz – 8 kHz (497 non-zero weights)
  -> 10*log10(max(P, 1e-10))      floored at −80 dB
  -> per-window standardisation   (mean 0, std 1)
  -> int8 quantise                q = round(v/scale) + zero_point
  -> TFLite Micro int8 inference  -> [P(no_cry), P(cry)]
```

Constant	Value	Constant	Value
`SR`	16000 Hz	`N_FFT`	512
`WIN_SEC`	2.0 s	`HOP`	400
`N_SAMPLES`	32000	`N_MELS`	64
`N_FRAMES`	81	`FMIN/FMAX`	20 / 8000 Hz
`CENTER_PAD`	256	`TAIL_PAD`	512
`DB_FLOOR`	−80 dB	`LOG_EPS`	1e-10
`CMVN`	True	`SILENCE_RMS_GATE`	0.0015 (~−56 dBFS)
`TARGET_RECALL`	0.95	`SEED`	1337

Mel-band weights are **exported from Python into C** (`crynet_tables.h`, sparse: start/end/offset + 497 floats), so the firmware physically cannot drift from the model.

Dataset assembly

* 2 s windows; clips longer than 2 s are cropped (random for train, centre for eval); shorter clips are zero-padded.
* **Splits are by source clip, never by window** — no recording can appear in both train and test.
  * ICD: folds 1–3 → train, fold 4 → val, fold 5 → test (the corpus ships its own folds).
  * cry4: 70 % train / 30 % cross-corpus test.
  * ESC-50: 75 % train / 25 % cross-corpus test, **plus all 40 `crying_baby` clips held out**.
  * Snoring: 75 % train / 25 % cross-corpus test.
* Exact-duplicate windows are removed by MD5 of the feature bytes.
* Augmentation (train only, waveform domain): random gain (−18…+6 dB) · real non-cry noise mixed at SNR 3–25 dB drawn from the ICD negative pool · optional room impulse response (ICD ships `rir.zip`, applied with p = 0.35) · SpecAugment in the feature domain (2 frequency bands ≤ 8 bins, 2 time spans ≤ 12 frames, drawn per sample).

Models (all TFLite-Micro friendly — no attention, no RNN)

Name	Params	Shape
`conv_cnn`	122,770	5 × [Conv2D + BN + ReLU + MaxPool2D] → GAP → Dense(64) → Dropout(0.3) → Dense(2)
`sep_cnn`	29,839	4 × [DepthwiseConv2D + pointwise Conv2D + BN + ReLU + MaxPool2D] → GAP → Dense(64) → Dropout → Dense(2)
`temporal_cnn`	65,154	Conv2D over time (Mel bins as channels) + MaxPool((2,1)) → GAP → Dense(64) → Dropout → Dense(2)

`bsc_cnn` exists in the code as an optional fourth candidate. The best variant is chosen on **validation AUC**.

Hard-example mining (added after the first real-data run)

After the main training, the notebook scores its own training set, takes the **600 most cry-like non-cry windows** and the **600 least cry-like cry windows**, duplicates them, and fine-tunes for 6 epochs at LR 3e-4 with early stopping. Only training rows are mined, so the held-out folds stay honest.

Motivation (measured): `coughing` produced a **57 % false-alarm rate** (car horn / church bells ~33 %) — acoustically a cough is a vocal expulsion, like a cry.

Threshold calibration

The alarm threshold is **not** hard-coded to 0.65. The notebook sweeps thresholds on the ICD validation fold and picks the largest threshold that still reaches ≥ 95 % recall on the cry class, then reports the whole recall/FPR curve so the trade-off is visible. (On the small real-data run this chose 0.23 → recall 95.7 %, FPR 13.6 %; at 0.50 it would be recall 91.9 %, FPR 4.2 %.)

INT8 quantization

Full INT8 post-training quantization (`TFLITE_BUILTINS_INT8`, int8 input and output) with a 512-window representative dataset drawn from the training set. The notebook then runs the **actual int8 TFLite interpreter** over the validation set and reports float-vs-int8 parity, and asserts every operator in the model has a TFLite Micro kernel.

---

4. Validation evidence

Check	Result
Whole notebook end-to-end (data → features → training → mining → INT8 → export)	✅ all 24 cells, on synthetic stand-in archives
Same pipeline on the **real** corpus	✅ cells 1–15 on all 28,478 real clips (feature extraction 128 s for 56,956 windows)
C front end vs. the Python training front end	✅ compiled and compared on 4 probes (sine+harmonic, noise, DC-offset cry with the INMP441 bias, silence): **max deviation 7.06×10^{-5}**, mean 2.8×10^{-6}
Every op has a TFLite Micro kernel	✅ `CONV_2D`, `DEPTHWISE_CONV_2D`, `MAX_POOL_2D`, `MEAN`, `FULLY_CONNECTED`, `SOFTMAX` — asserted at export time, recorded in `deploy_config.json`
INT8 vs float32	✅ AUC delta **−0.00007**, 99.70 % decision agreement
Corpus integrity	✅ ICD manifest ⇔ archives exact 1:1; all 16 kHz mono / 5.00 s
Exported model size	✅ **135 KB INT8** (122,770 params)
`nbformat.validate()`	✅ notebook is valid

Measured on the real corpus

From a deliberately small local CPU run — **3 epochs on an 8,490-window subsample** (the full Kaggle run is 74 k windows / 35 epochs / 3 models), so these are a **floor, not final numbers**:

Split	AUC	F1	Recall	FPR
ICD val (fold 4)	0.9807	0.9145	0.9504	0.1240
ICD test (fold 5)	0.9839	0.9131	0.9534	0.1058
cross-corpus (cry4 + ESC-50 + snoring — never trained on)	**0.9887**	0.9413	0.9544	0.0789

Cry recall on the **unseen** cry4 corpus: `needs` 98.8 % · `physical_pain` 96.9 % · `scared` 90.5 % · `burping` 82.0 %. On ESC-50's `crying_baby` (also unseen): 85 % recall.

Per-category false-alarm rates (worst): `coughing` 57 % · `car_horn` 33 % · `church_bells` 33 % · `siren` 25 % · `washing_machine` 23 %.

Mining effect measured on the stand-in run: cross-corpus FPR **0.109 → 0.004**.

Bugs found and fixed during validation

All of these would have surfaced **on the device, not in training**:

1. `tf.py_function` erased the batch shape → Keras raised `as_list() is not defined on an unknown TensorShape`. Fixed by rewriting SpecAugment in pure TF ops (also faster on GPU).
2. The TFLite converter rejected the representative dataset: it must yield a **list** of arrays, not a bare array.
3. The front-end output tensor was missing its channel axis → `X[i:i+1]` instead of `X[i:i+1, :, :, None]`.
4. A runaway silence gate (0.0035) would have discarded the quiet ICD cries (RMS 0.006) → lowered to 0.0015.
5. The augmentation loop iterated over *windows* instead of *clips*, doubling the training set and memory → fixed.
6. Holding every intermediate array caused a **native segfault** during `model.fit` on Windows (17 GB box, 10.6 GB free) → added explicit `del` + `gc.collect()` and a bounded 32,768 shuffle buffer.
7. `GlobalAveragePooling2D` lowers to `MEAN`; TFLM registers it in `kernels/reduce.cc` (**not** `mean.cc`, so a naive filename search wrongly suggests it is unsupported) → confirmed and now asserted automatically.

---

5. File map

```
C:\Users\HP\.t3\projects\fp-2\
├── README.md                    ← project overview, dataset decision, contract
├── PROJECT_HANDOFF.md           ← static deep briefing (paste into a new chat)
├── SYSTEM_UPDATE.md             ← LIVING status log, updated after every edit
                                 ← recovery layer if the chat/context is lost
├── .gitignore
├── notebooks/
```

```
│   └── crynet_training.ipynb    ← THE DELIVERABLE (24 cells, Kaggle-ready)
└── research/                    ← the investigation; not needed to run the notebook
    ├── README.md
    ├── build_notebook.py        ← generates the .ipynb (patch this, then rebuild)
    ├── run_notebook_local.py    ← executes the notebook cell-by-cell locally
    ├── run_stub.sh              ← end-to-end test on synthetic stand-in archives
    ├── run_real.sh              ← full-corpus local run (CPU; slow)
    ├── run_real_fast.sh         ← full-corpus, subsampled, CPU-friendly
    ├── make_stub.py             ← builds the stand-in archives
    ├── test_c_parity.py         ← compiles dsp_crynet.c, compares against Python
    ├── scan_kaggle.py           ← Kaggle dataset search/metadata API (no auth)
    ├── inspect_zips.py          ← archive structure without extracting
    ├── analyze_audio.py         ← audio forensics (SR, duration, F0, flatness)
    ├── _cache/                  ← 5.5 GB of downloaded corpora (gitignored)
    └── _stub_out/, _realfast_out/  ← artifacts from local runs (gitignored)
```

The notebook is generated. To change it, edit `research/build_notebook.py` and run `python build_notebook.py` — do not hand-edit the `.ipynb`.

Artifacts the notebook writes to `out/`

File	Contents
`crynet_int8.tflite`	the deployable model
`crynet_float.tflite`	float32 reference
`crynet.keras`	the trained Keras model, so INT8 export can be repeated without retraining
`model_data.h`	the model as a C byte array (`crynet_model[]`, `crynet_model_len`)
`crynet_tables.h`	Mel filterbank (start/end/offset + 497 weights), Hann window, all constants, quantisation scales, threshold, silence gate
`dsp_crynet.c` / `.h`	portable C99 front end — `crynet_log_mel()`, `crynet_quantise()`, `crynet_dequantise()`, `crynet_window_rms()`
`crynet_infer_example.c`	TFLite Micro glue: PCM → log-Mel → quantise → `Invoke()` → `P(cry)` → `crynet_is_cry()`
`deploy_config.json`	input shape, quantisation params, operator list, TFLM kernel check, threshold, metrics, model SHA-256
`metrics.json`	per-variant AUC / F1 / recall
`c_parity.json`	measured C-vs-Python front-end deviation

---

6. How to run

Kaggle (the real run)

1. New notebook → *Add Input* → attach nothing (everything downloads) or pre-attach for speed.
2. Accelerator **GPU T4**, Internet **On**.
3. Upload `notebooks/crynet_training.ipynb` → *Run All*.
4. Expected: ~10 min download (4.6 GB, resumable via `huggingface_hub`) + ~5 min feature extraction + ~20–40 min training for the 3 variants + mining + INT8 export. Artifacts land in `/kaggle/working/out/`.

Locally

```bash
pip install numpy librosa soundfile soxr scipy scikit-learn tensorflow huggingface_hub
# end-to-end smoke test on synthetic stand-ins (minutes)
cd research && python make_stub.py && bash run_stub.sh
# full corpus, subsampled, CPU (tens of minutes)
bash run_real_fast.sh
# C <-> Python parity
pip install ziglang && python test_c_parity.py
```

Environment switches (env vars read by the notebook)

Variable	Effect
`CRYNET_FAST=1`	subsample the corpora and train 3 epochs — smoke test
`CRYNET_OFFLINE=1`	never download; require the archives to already be in the cache
`CRYNET_KINDS`	which variants to train, e.g. `temporal_cnn` (default `conv_cnn,sep_cnn,temporal_cnn`)
`CRYNET_EPOCHS`	training epochs (default 35)
`CRYNET_MINING_ROUNDS`	hard-example mining rounds (default 1; 0 disables)
`CRYNET_CACHE` / `CRYNET_WORK` / `CRYNET_OUT`	cache, scratch and output directories

The HF archives are cached at `CRYNET_CACHE/audio/positive.zip`, `.../audio/negative.zip`, `.../metadata/all28479.csv`; the Kaggle ones at `CRYNET_CACHE/<key>__<owner>__<slug>.zip`. A file attached under `/kaggle/input` is used as-is.

Verified versions (local Windows box)

Python 3.11.9 · TensorFlow 2.21.0 (CPU-only; no GPU on Windows TF ≥ 2.11) · librosa 0.11.0 · soundfile 0.14.0 · numpy 2.4.6 · scipy 1.17.1 · scikit-learn · ziglang 21.1.0 (used as a C compiler for the parity test, since no gcc is installed).

---

7. Known risks and open questions

1. **The full Kaggle run has not happened yet.** Every number above comes from a small local run. Expect better, but verify.

2. **`coughing` remains the top confuser** (57 % FPR before mining). If it survives mining, the next lever is a dedicated cough corpus (the FSDKaggle2018 cough set would add several hundred clips).
3. **No in-domain data.** The model has never heard your INMP441, your room or your gain. **Recording ~50 real clips from your own hardware (cries + room noise + speech + TV) and adding them as a custom corpus is the single biggest remaining accuracy win.**
4. **TFLM latency is unmeasured.** Estimate: 135 KB INT8 CNN over 81×64 on a 240 MHz LX7 ≈ 100–300 ms per 2 s window — acceptable for an event, but it must be measured on the real board. The old spec claimed 38–54 ms for a smaller 58 k-param model.
5. **Tensor arena size** is a hint only (120 KB). Verify `AllocateTensors()` succeeds; put the arena in PSRAM if it does not fit in internal SRAM.
6. **The ble/display policy** is unchanged from the old spec and still assumes binary output — fine.
7. The old firmware's other gotchas still apply: INMP441 DC offset, ESP-NOW TX queue overflow (needs callback flow control), Arduino-3.x callback signature change, and the ILI9341 240 px clipping (the `HubDisplay` subclass).

---

8. Next checkpoint — the ESP32-S3 firmware

Concrete plan, in order:

1. **Port `dsp_crynet.c` to ESP-IDF.** Swap the reference radix-2 FFT for ESP-DSP `dsps_fft2r` + `dsps_cmplx_mag` (~10× faster). Copy `crynet_tables.h` and `model_data.h` into the firmware tree.
2. **Wire TFLite Micro.** INT8 input `[1, 81, 64, 1]`; `MicroMutableOpResolver` with `AddConv2D`, `AddDepthwiseConv2D`, `AddMaxPool2D`, `AddMean`, `AddFullyConnected`, `AddSoftmax`; arena in PSRAM if needed. `crynet_infer_example.c` is the template.
3. **Fix the trigger threshold** in the hub: drop the RMS ≥ 0.18 audio-pull trigger to ~0.03 so soft cries reach the model, and let the calibrated threshold do the discriminating. Add the 0.0015 silence gate.
4. **Forward the model output** to the existing BLE packet: `alert_type = 0x01` when `P(cry) >= threshold`, `confidence_pct = round(P(cry) * 100)`.
5. **Measure** inference latency and arena usage on the real board, and log both.
6. **Collect ~50 in-domain clips** and add them as a custom corpus to the notebook, then retrain.

---

9. Working preferences (important)

* **Work in checkpoints.** Deliver one artifact per checkpoint; confirm before starting the next stage.
* **Run very few agents in parallel.** A previous session launched ~18 agents at once and exhausted the usage limit within minutes; several sessions then died with "400 Context Limit Exceeded". Prefer direct tool calls (WebSearch / WebFetch / curl / scripts) over subagents. Two sessions died mid-run during this checkpoint — long foreground commands are risky; **prefer background runs writing to a persistent log file in the repo** (e.g. `research/stub_run.log`) so a session break does not lose the work.
* The user is happy for the agent to make and document judgement calls (they answered three structured questions: binary head, "any cry" label policy, Kaggle execution).

---

10. Source references

- ICD corpus — https://huggingface.co/datasets/hoyyu1/infant-cry-detection (paper: arXiv 2508.19308, "Infant Cry Detection In Noisy Environment Using Blueprint Separable Convolutions and Time-Frequency Recurrent Neural Network")
- ICSD (gated, rejected) — https://arxiv.org/html/2408.10561v1
- Baby Cry 4-class — https://www.kaggle.com/datasets/karimeletriby/infant-cry-classification-4-classes
- ESC-50 — https://github.com/karolpiczak/esc-50
- Snoring — https://www.kaggle.com/datasets/tareqkhanemu/snoring
- Survey of infant-cry datasets — https://www.computer.org/csdl/journal/ta/2026/01/11267068/2bUqPTUno40
- TinyML infant-cry classification (79,461 params, 96.26 % Baby Chillanto) — https://ieeexplore.ieee.org/iel8/6287639/10820123/11132295.pdf
- TFLite Micro kernel registry — https://github.com/tensorflow/tflite-micro (`tensorflow/lite/micro/kernels/reduce.cc` registers MEAN; `micro_mutable_op_resolver.h` has `AddMean`)
- MicroNets / BC-ResNet (tiny audio backbones) — https://arxiv.org/abs/2010.08691

One-paragraph summary for a lazy reader

Checkpoint 1 of the infant-cry detector is done: the best public corpora were identified by audio-level forensics (ICD as primary — 28,478 real 16 kHz clips from CryCeleb2023/EnesBabyCries/iFLYTEK — plus Baby-Cry-4, ESC-50 and Snoring; the `mennaahmed23` family was rejected as contaminated), and a 24-cell Kaggle notebook was written that trains a ~135 KB INT8 CNN on 2 s 81×64 log-Mel windows, calibrates the alarm threshold for ≥95 % recall, verifies the C front end against Python to 7 × 10^{-5}, and exports `crynet_int8.tflite`, `model_data.h`, `crynet_tables.h`, `dsp_crynet.c` and `crynet_infer_example.c`. The notebook is written and validated end-to-end (24/24 cells) on stand-in archives, cells 1–15 on the real 28,478-clip corpus, with an indicative real-data result of **val AUC 0.981 / cross-corpus AUC 0.989 and an INT8-vs-float AUC delta of −0.00007**. The user's next action is to run the notebook on Kaggle (T4, ~40 min) and download `out/`; the following checkpoint is the ESP32-S3 firmware port, where the trigger threshold must drop from RMS 0.18 to ~0.03.

The living status log

SYSTEM_UPDATE.md

651 lines · 461,970 bytes

```
# CryNet — SYSTEM_UPDATE.md (living status log)

> **This file is the recovery layer.** It is rewritten/appended by the agent **after every project update or edit** (code, notebook, config, decision, or a completed run). If the chat dies, is compacted into garbage, or hits "400 Context Limit Exceeded", a fresh chat reads this file first and is immediately back up to speed.
>
> **Repo:** `C:\Users\HP\.t3\projects\fp-two-node` (branch `main`, remote `origin` → `https://github.com/Amazo123/CryNet-TwoNode.git`) — the **two-cot** system.
> **Sibling repository:** `C:\Users\HP\.t3\projects\fp-2` (branch `main`, remote `crynet` → `https://github.com/Amazo123/CryNet.git`) — the **three-cot** system.
> Shared history up to `99bb257`, then separate. **The system on the bench is this one.**
> **Deep briefing (static, rarely changes):** `PROJECT_HANDOFF.md`
> **Decision and issue log (for the report):** `DECISIONS.md`
> **Last updated:** 2026-11-25 (**the branch is merged: `main` now carries the whole dashboard + presence line, on the owner's explicit call -- not pushed yet.**) | After the D-108 fix, the D-107 stack soak, and the owner's own end-to-end cry test (five cries, both cots, presence all online), the owner said: *"i think you can merge now, as everything works as they should."* **`main` was fast-forwarded `dd7a693` -> `dd85282`** -- the D-100-to-D-108 line lands together: the dashboard (`server.py`, web, Cloudflare Worker, API docs), the hub's cry log / net / clock components and uploader, the presence feature with its burst-yielding 30 s cadence, the boot-banner build fix, the task-main stack fix, and the presence staleness fix. `check_prebuilt.py`, `check_artifact_drift.py` and `test_dashboard_contract.py` (including the D-108 presence-staleness section) **all pass on the merged `main`**. The branch `feature/hub-wifi-and-dashboard-presence` is retained. **`origin/main` is still at `dd7a693` -- the push is deliberately not done**, because the repository is public and the owner's call covered the merge; pushing is its own decision to confirm. > **Earlier:** 2026-11-25 (**the presence panel could hold a dead hub's cots "online" for ever: a cot is online now only while the hub's word about it is fresh -- D-108.**) | The owner powered the whole system down and the dashboard said hub offline, both cots online -- a contradiction, because the cots have no network of their own: the hub is the only thing that can answer for them. The cause was one missing rule in `build_presence()` (both the Worker and `server.py`): the hub is judged by the D-105 window (90 s since its last word), but a cot row's `online` flag was served verbatim from whatever the hub last posted, with no freshness check on the row's own `at` stamp -- so the hub's final post before the power was pulled held both cots "online" for ever, and the panel whose one job is "quiet baby or dead hub?" answered **watched** while nothing was watching. **The fix is one rule, in both copies so they still agree field for field: a cot is online only if the hub said so and the saying is fresh** -- `at` inside the same 90 s window the hub itself is judged by; a row with no stamp reads offline; the demo hub keeps its by-design exemption. The page needed no change. **Proved twice:** `test_dashboard_contract.py` posts the firmware's exact presence body, reads both cots online, moves the store's clock ten minutes past the window (the only way a test can pull a plug) and demands the cots go down with their hub -- against the old code it fails with precisely the owner's report, against the new it passes. The Worker was redeployed (`d6deaeda`) and the live store confirmed the diagnosis: both cot rows still carried the hub's last words with `at` 2,251 s old against the 90 s window -- the very rows that produced the lie, now aged out, so the next 10 s poll of an already-open page shows all three offline with no reload. (The git-ignored `dashboard/.secrets` records the token but its password entry is empty, so the live endpoint was verified through the store itself, not a browser login.) No firmware change, so `firmware/prebuilt/` is untouched and nothing was flashed -- every unit is off. **Nothing pushed, nothing merged into `main`.** > **Earlier:** 2026-11-25 (**the "restarts" were a real crash after all: a stack overflow in task main, whose 3,584-byte ESP-IDF default carries the entire live system because app_main never returns -- the stacks are sized for what runs on them now -- D-107, correcting D-106.**) | The corrected image from D-106 was on the hub and the 600-second watch was still running when it caught what every earlier window had missed: at 502.84 s, `***ERROR*** A stack overflow in task main has been detected`, backtrace corrupted, `rst:0xc (RTC_SW_CPU_RST)`, and the hub booted itself again. The ELF SHA in the panic (`72d2f7266`) matches the flashed image; the crash log is kept at `research/out/hub_stack_overflow.log`. **D-106's "power-class, not crashes" rested on absence** -- no crash text had been captured, and the boot counter's bumps had been attributed to the instruments -- a plausible fit, not a proof. The rule that replaces it: **absence in a capture window is not absence.** The root cause: `app_main` never returns in this firmware -- `link_stage_run()`'s `for(;;)` runs the telemetry, burst handling, front end, inference, logging, BLE notify and display on the main task, whose 3,584 bytes were ESP-IDF's default for an `app_main` that spawns workers and returns; the sdkconfig comment even said "larger task stack for the inference task", and no config line stood under it. The corrupted backtrace cannot name the deepest frame, so the fix is headroom, not a tuned path: **`CONFIG_ESP_MAIN_TASK_STACK_SIZE=16384`** (12.8 KB more internal RAM against ~296 KB free at boot, nothing at runtime) and **the net task 6,144 -> 8,192** for headroom of its own -- its HTTPS posts run a TLS handshake on that stack, and the standing rule is that the connectivity feature may not cost the system anything. What the crash is not: it landed 0.64 s after a presence post, but the post runs on the `crynet` net task with its own stack, so it did not eat main's; 16 of the capture's 17 posts kept the exact 30 s cadence and one took 33 s because the D-105 guard made it wait for a burst -- the design working. **Presence stays.** The cot symptoms: "offline for a split second" is the link layer's offline/online pair 110 ms apart at a staleness tick boundary (twice in 600 s, zero frames bad across either); "rms freezing" is telemetry-less frames carrying the last frozen value between updates. Cosmetic both. **The fix soak, on the new image (banner `dabb4e0-dirty`, boot 108): 600 s clean** -- zero resets, zero offline lines, a 286-fragment burst (the size that preceded the crash) completing 286/286 in 0.98 s at 451.9 s, 20 presence posts on cadence, node 1 seq 217 and node 2 seq 231 live at 600 s, 11,974 frames off the air, 0 bad, 0 failed. One clean soak cannot name the frame the corrupted backtrace hid; it shows the workload that killed the old stack running with an order of magnitude of headroom. `check_prebuilt.py`, `check_artifact_drift.py` and `test_dashboard_contract.py` all pass. **Nothing pushed, nothing merged into `main`.** > **Earlier:** 2026-11-25 (**the banner was lying about which build the hub runs -- D-106; its "the restarts were power-class resets" conclusion was corrected by D-107 hours later, but the banner fix and the ten-minute coexistence watch stand.**) | The owner reported the hub restarting, cot 1 flapping offline for a split second, and cot 2's RMS frozen, and restated the constraint in its strongest form: *"if this connectivity status is going to cause an issue, you can scrap it or make it in a way it doesn't affect system performance at all."* **The feature is not the cause, and it stays.** Three findings. **(1) The hub's boot banner read `bd3d6fb-dirty` (built Nov 24 06:31) -- three features behind the branch tip -- while its behaviour was the current code: presence posts marched past at the exact 30-second D-105 cadence. ESP-IDF bakes the app version and the compile time at configure time, and scons had reused a Nov 24 configure for every build since** -- fresh code wore a stale label, and the committed prebuilt carried the same one, which a flash reproduced byte-for-byte: that is how the lie was caught. **(2) The resets were power-class, not crashes.** No panic text was captured anywhere; every reset observed reads power-on. The CH343 USB bridge resets the board whenever a serial monitor opens the port, and the boot counter's four bumps during the diagnosis (97 -> 101) are exactly the two capture-opens and the flash session -- the instruments' resets, not the hub's. The morning's cot-1 flap and cot-2 frozen RMS came on a boot during the owner's own unplug/replug tests and did not survive a clean-power boot. **(3) The settling watch: 600 s of serial, zero restarts, zero `is offline` flaps, node 1's window advancing (seq 42 -> 63), node 2's telemetry live (seq 40 -> 45, RMS moving), and 15 presence posts at exactly 30 s** -- one radio, both jobs. **The fix is to the build, not the firmware:** `firmware/tools/make_prebuilt.py` now forces a reconfigure before the hub build and refuses an image whose banner does not carry the current `git describe`; `firmware/prebuilt/` was repackaged and the hub on COM32 reflashed. The rule written into D-106: **when a unit's banner disagrees with `git log`, suspect the build cache, not the flash** -- and if a restart ever recurs, its reset reason now names itself against a banner that tells the truth. `check_prebuilt.py`, `check_artifact_drift.py` and `test_dashboard_contract.py` all pass. **Nothing pushed, nothing merged into `main`.** > **Earlier:** 2026-11-25 (**the presence now yields to a cot's burst, and its rate was set back by the owner's constraint that nothing may cost the cots anything -- D-105.**) | The owner, having asked for the connection to be checked "as often as possible", added the constraint that actually decides it: *"if this is going to affect the cots
```

or any system performance, you can scrap it or increase the interval or whatever, i don't want anything affecting the system."* **The physics is the link stage's own finding: while a burst is on the air the Wi-Fi receiver hears nothing at all -- not that burst's own fragments, and nothing else** -- so a 15-second presence report would land inside a burst roughly one time in three and cost fragments and resends each time. Three things were done. **(1) The hub reports every 30 s**, not 15 -- 2,880 TLS handshakes a day, twice the original rate, half yesterday's. **(2) The presence yields**: the link stage hands the network stage `crynet_net_set_radio_guard()`, which answers whether a burst is in flight (`st->requested`), and the report waits for it -- capped at **20 s**, because a hub that has been switched off must still be reported gone. The worst gap between reports is 50 s, inside the dashboard's 90 s idea of how long a hub may go silent, and a **host model of the scheduler proves the cap**: a guard stuck saying "busy" for ever still posts every 50 s, so the new code cannot make the dashboard think the hub has gone away. **(3) The dashboard's window went to 90 s** (three reports missed, the same rule of three), in both `server.py` and the Worker. The record upload was left alone -- it has always contended with bursts and the chain was verified with it running; the page keeps its own 10-second clock, which asks the service and costs the cots nothing. **A hub switched off now reads as gone in about 1.7-2.2 minutes** (was 2-4.5 before any of this). The knob is one line (`PRESENCE_EVERY_S`) if the day's end-to-end tests say otherwise; what to watch is the link stage's counters -- `resends_sent`, `gap_ewma_ms`, `burst_failures`. The Worker was redeployed (`c4f78eb4`) and the hub on COM32 was flashed with the new image. Builds clean, `firmware/prebuilt/` rebuilt, `check_prebuilt.py`, `check_artifact_drift.py` and `test_dashboard_contract.py` all pass. **Nothing pushed, nothing merged into `main`.** > **Earlier:** 2026-11-25 (**the hub's connection is now checked in about a minute -- the hub reports every 15 s, the service calls a hub gone after 45 s, the page asks every 10 s -- D-104.**) | The owner switched the hub off and the dashboard went on saying *online*, and asked for the connection to be checked "as often as possible". **When the dashboard decides a hub has gone is the sum of three delays** -- how often the hub speaks (was 60 s), how long the service waits (was 180 s), and how often the page asks (was 30 s) -- 46 to 270 seconds from the plug being pulled, and the owner was watching it say *online*. All three were shortened, keeping the rule of three chances missed before calling a hub gone, three times faster. The presence has **its own clock on the page** (every 10 s, without dragging the nine reads of the record along), and is asked again the moment the tab comes back. **Two real faults fell out of the measuring:** (1) **the presence strip was inside the overview body, which is hidden when there is nothing to show** -- so the one element whose job is *is the watcher alive* vanished exactly when the question is asked; it is now outside the body and drawn whatever the record holds. (2) **a request that never came back froze the page on its last answer for ever** -- `refresh()` refuses to run while `state.busy` is set and nothing cleared it, so one hung fetch meant the page never polled again; every read now has a 15 s deadline and the guard a 60 s one. **Measured:** the service's deadline passed at 1795575496 and the page said *offline* at 1795575498; and on the live Worker the hub's new image posts every 15 s (`at` 1795575661 -> 1795575676 -> ...) and reads `online:false` at age 77. The Worker was redeployed (`ef3ec161`) and the hub on COM32 was flashed with the new image. Builds clean (RAM 14.3 %, flash 77.2 %), `firmware/prebuilt/` rebuilt, `check_prebuilt.py`, `check_artifact_drift.py` and `test_dashboard_contract.py` all pass. **Nothing pushed, nothing merged into `main`.** > **Earlier:** 2026-11-25 (**the wireless mark now means exactly one thing, it is big enough to read, and the presence chain is closed end to end from the board -- D-103.**) | The owner gave the upload token and asked for two things: **the Wi-Fi mark should show only when connected**, and *"as it is now, i can't see anything like it on the hub display"*. Both are done, and the second one had three separate causes. **(1) The mark is inverted.** D-100 drew it in both states -- green on the network, dim grey off it -- on the reasoning that a mark which appears and disappears cannot say whether the network is down or the panel is asleep. The owner overrode that, and the override is the better design on a 5x7-font panel: a mark that is always there has to be **read** to be understood, and *grey means not connected* is a second thing to learn beside the two pills. Drawn only when it means something, the rule is one line long -- *a mark here means the hub is on the network* -- and the flag is the same one the uploader uses to decide whether a record can leave the hub, so the glass cannot disagree with the network. Where the hub is off the network the dashboard says so instead, and there the question has room to be asked and answered. The `C_DIM` off-state no longer exists in the firmware. **(2) The mark was too small.** Radii 6 and 10 over a 35-145 degree span became radii 8 and 13 over 40-140 degrees, the router dot and both arcs two pixels thick -- 21 px wide and 14 px tall in a 320-px header, sitting in the free space between `S3 HUB` (which ends at x~126) and the `NODE` pill (x=196). The geometry was measured against the panel's own constraints rather than chosen by eye, and the host mock was proved **pixel-identical** to the firmware by emulating C's float32 `cosf`/`sinf` in Python -- no host C compiler was available, so the equivalence was proved rather than assumed. **(3) It was not on the glass for a third reason, which was not the firmware's:** the board was running the pre-branch image, in which the glyph did not exist at all. **(4) The token was proven against the live Worker.** `POST /api/v1/presence` with the token answers 200 and without it 401; `GET /api/v1/presence` returns `hub-f9172f80` with `"online":true`. The link D-101 could not close is now closed **from the board**: the flashed hub's own log shows `crynet: presence posted: 0 cot(s)` and `crynet: sent 1 records, up to #78 ({"accepted":1,"received":1})`, and the live Worker answers with the hub present. (`nodes: []` is correct, not a bug -- the cot nodes are off, so the hub has nobody to report; the hub itself is genuinely present.) **(5) The token is off Git.** The repository is public, so the token lives in exactly three places: Cloudflare's secret store (`UPLOAD_TOKEN`), the hub's own NVS (`net token`), and a local, **git-ignored** `dashboard/.secrets`. `.gitignore` now lists `dashboard/.secrets` and `dashboard/worker/.dev.vars`, and `dashboard/worker/README.md` says where the file is, how to use it and how to change the secret. D-97's rule holds. **(6) One thing to know about Cloudflare and Python:** the edge **rejects `Python-urllib` with error 1010** before the request reaches the Worker, so a Python script posting to the hosted copy needs its own `User-Agent` (`CryNet-Tools/1.0`); the hub is unaffected -- it sends `ESP32 HTTP Client/1.0`, which the edge passes. **The hub on COM32 has been flashed with the new image** (via `firmware/tools/flash_unit.py --role hub --port COM32`; the images were hash-verified first; NVS was never written, so the board's id and its token survived). The firmware builds clean (RAM 14.3 %, flash 77.2 %), `firmware/prebuilt/` was rebuilt from it, and `check_prebuilt.py`, `check_artifact_drift.py` and `test_dashboard_contract.py` all pass. **Nothing has been pushed and nothing has been merged into `main`.** > **Earlier:** 2026-11-25 (**the dashboard now says who is here — the hub's panel carries a wireless mark, the hub reports in once a minute whether or not there is anything to record, and the page reads like an instrument rather than a paragraph — D-100 to D-102.**) | Three changes the owner asked for together. **(1) The panel has a wireless mark.** A Wi-Fi glyph drawn from the panel's own pixels (a dot for the router, two arcs over it, walked by angle rather than by column so the sides have no gaps), between `S3 HUB` and the `NODE` pill. It is drawn **whether or not the hub is on the network** — green when it has an address, dim grey when it does not — because a mark that only appears when the link is up cannot say whether the network is down or the panel is asleep. The glyph was checked as ASCII on the host before it was committed. **(2) The hub reports who is here.** New `POST /api/v1/presence` and `GET /api/v1/presence`, in both `dashboard/server.py` and the Cloudflare Worker, with a `presence` table in each. The hub posts it **every minute and again the moment a cot's state changes**, so a quiet week on the dashboard is legible: was the baby quiet, or did the hub stop? A cry record cannot answer that — it only says what was heard. The cots have no network of their own, so the hub answers for them. **The owner still configures one address**: the presence endpoint is derived from the events URL by replacing its last path segment. The page shows it as a strip of pills under the numbers. **(3) The page was cleaned up.** The strapline "the cry record" is gone from the title and the header; every subtitle, note and empty state was cut back to the fact it carries. The daily-cry sparkline is now a Fritsch–Carlson monotone cubic — a plain Catmull-Rom spline dips below the baseline between two quiet days, and a chart of counts that draws a negative count is lying — and the bars are paths with a rounded top, a square foot and a one-pixel edge. **A real bug fell out of it:** `[hidden]` did nothing on any element a class gave a `display` to, so the **Hub** dropdown sat on the page with a single hub; `[hidden] { display: none !important; }` fixes it everywhere. The hub firmware builds clean (flash 77.2 %, RAM 14.3 %) and `firmware/prebuilt/` was rebuilt from it — `check_prebuilt.py` and `check_artifact_drift.py` both pass. **Nothing had been flashed at this point**: the owner flashes the units himself (the hub on COM32 was flashed later the same day -- see the entry above). The hosted Worker was redeployed with the new endpoint, the new table and the new page. Branch `feature/hub-wifi-and-dashboard-presence`.) > **Earlier:** 2026-11-24 (**a cry could be lost for ever — the log's file is not in number order and the uploader searched it as if it were — D-87 to D-94; built and compile-verified, committed on the branch, not flashed**) — three new hub components (`crynet_log`, `crynet_clock`, `crynet_net`) mount the six-megabyte storage partition that had always been reserved and never used, write one 32-byte record per cry episode whether or not there is a network, and upload the stored records when there is. The wrist's "event #N" is now the dashboard's record #N. **The router's 2.4 GHz channel must be 6** — the cots talk ESP-NOW on channel

6, the hub has one radio, and the uploader deliberately drops any association on another channel rather than let the wrist go silent. The dashboard is `dashboard/server.py` (Python stdlib + SQLite, no dependencies); start it and it prints the exact `net set` line to type on the hub's console. Owner's guide: `dashboard/README.md`. **A cry could be lost for ever, and it was (D-94):** a record's number is spent when the episode *starts* and the record is written when it *ends*, so two cots crying at once can end in the opposite order and the file is **not sorted by number** — the uploader's binary search over it could land past a record that was never sent, and that cry would never reach the dashboard while the hub said everything had gone. The uploader now takes the **lowest** unsent numbers by a linear walk, and the number it claims as sent stops below the lowest number that did not fit and below any episode still being measured. **Proved twice:** `firmware/tools/test_log_order.py` (host model, six tests, a ten-thousand-case soak, and a companion test that shows the *old* code losing records) and the hub's new **`logtest`** console command, which runs the same scenario against the real component on the real chip on its own scratch file. **The flash write on the audio path was checked against the link code rather than assumed: fragment reassembly runs in the ESP-NOW receive callback, so a record write cannot cost a fragment — it only delays the next pull (D-93). The write was trimmed to one page per episode and the hub now prints what it actually costs.** The web client was driven in a real browser against seeded and empty databases: both tabs render, the console is clean, and the empty state says so rather than showing a broken chart. Branch `feature/cry-dashboard-and-log`; everything committed in five commits (`44cbfb9`, `c5da57d`, `8c43fa2`, `7cd0816`, `614a629`), **nothing pushed and nothing merged into `main`**, nothing flashed, and the offline proof has not been run on the bench yet.

> **Earlier:** 2026-11-11 (**the documents now say 22 Ω where the board has 22 Ω, and the 5 V supply chain has three written rules** -- the stall cap fitted in the haptic motor's + lead is 22 Ω, not the 10 Ω the documents named (D-85: it is the safer value, capping the stall current at about 56 mA instead of 70 mA, at the cost of a slightly softer buzz); the BOM, SYSTEM_WIRING.md, D-77 and page 4 of the wiring diagram were corrected and the PDF rebuilt. **New D-84** writes down the three rules for powering a board through its 5 V pin: set the boost trimpot to 5.1 V with a meter before attaching any board, never run the boost and USB into the same board at once (they are one net through the board's protection diode, so current can be pushed back into the laptop -- use a VBUS-cut cable to watch a unit on its console), and remember that the board's own linear regulator burns (5.1-3.3) V x I, roughly 0.5-0.7 W at radio peaks. The sibling repository got the same corrections.) > **Last updated:** 2026-11-11 (**the real wearable firmware is back on the board, and the board carries exactly the committed offline image** -- flashed on COM30 with `python firmware/tools/flash_node.py --port COM30 --sketch wearable`, 685,296 B written, hash verified, and the boot log in `research/out/wearable_boot_20261111.log` shows the real firmware running: `CryNet wearable — BLE alert client (stage 2d)`. A fresh build of the same source and `firmware/prebuilt/wearable/app.bin` differ in **74 bytes out of 685,296** -- the build stamp and the two hashes that follow from it -- so the flash is the shipped image in behaviour; `check_prebuilt.py` and `check_artifact_drift.py` both still pass. **One caveat in that log:** it says `[oled] up` although **no display is attached** -- the SSD1306 library's `begin()` returns true unconditionally and never checks the ACK, so that line proves nothing about the panel; judge the screen by eye when it is attached. **The motor is idle by design in this firmware:** the buzz waits for an alert over BLE from the hub.) **the wearable's haptic motor was driven from a bench test sketch on COM30 and proved to be toggling** -- new `firmware/wearable/haptic_test/haptic_test.ino`, flashed with `python firmware/tools/flash_node.py --port COM30 --sketch haptic_test`; the real wearable firmware has to go back on the board afterwards. The battery path is still unmeasured and the documents still say 10 ohm where the fitted cap is 22 ohm.) **the D-83 firmware was flashed onto all four units and verified live** — the hub, both cot nodes and the wearable now run the firmware with no battery sense and no status LEDs; the hub's boot self-check passed, the panel is up, both cots are online and the wearable is connected over BLE. **Both nodes read exact digital zero, and that is correct: the owner confirmed only the boards were connected, with no INMP441 attached at all.** D-83, D-79/O-1c.) **The battery SOC and every status LED were removed from the system — D-83.** The owner took the battery readout out altogether, along with the node's connection LED, the node's SOC LED, the hub's BLE-status LED and the hub's SOC LED, because the hub's display already shows the connection. The firmware, the wiring documents, the bill of materials and the diagrams were all changed to match; the battery *power* — cells, modules, switches, charge currents — is untouched. **Nothing has been flashed yet**: the firmware is built and compile-verified, and the units are waiting on USB. D-81 is marked superseded in part, O-1e is closed, O-1a drops the wearable SOC readout)

---

0. Protocol the agent follows (non-negotiable)

1. **Update this file after every edit.** Every change to the repo, every decision, every completed run gets an entry in §5 *Change log* (newest first) and, if it moves the needle, a refresh of §3 *Live status*.
2. **Keep chat context small.** Never paste large files/diffs/logs into chat. Write them to disk (e.g. `research/*.log`) and reference the path. Prefer background runs with a persistent log over long foreground commands. If context is getting long, say so and ask the user to run `/compact` — the agent then reloads state from this file.
3. **One artifact per checkpoint.** Confirm before starting the next stage.
4. **Minimal agent fan-out.** Use direct tool calls (WebSearch/WebFetch/curl/python) over subagents. At most 1–2 subagents when genuinely needed. A previous 18-agent fan-out exhausted the usage quota and several sessions died with 400 errors.
5. **Never hand-edit `notebooks/crynet_training.ipynb`.** Edit `research/build_notebook.py`, then regenerate with `python build_notebook.py`.
6. **Long runs go to the background** with output to `research/<name>.log`, so a session break never loses the work.

Recovery procedure (for a fresh chat or lost context)

1. Read this file (top to bottom).
2. Read `PROJECT_HANDOFF.md` for the frozen contracts (signal contract, dataset decisions, file map).
3. `git status` + `git log --oneline -5` to see what is committed vs. uncommitted.
4. Continue from **§4 Next action**.

0.1 Live background work (as of 2026-10-31, afternoon)

What	Where	How to check
A local stub run of the patched training notebook (FAST mode, the stub cache) — it validates the new music corpus end to end.	`research/stub_music_run.log`	`tail -30 research/stub_music_run.log`
The GTZAN music corpus downloading from Kaggle (`carlthome/gtzan-genre-collection`, ~1.2 GB).	`research/_cache/gtzan__carlthome_gtzan-genre-collection.zip`	`ls -la research/_cache/gtzan__*`
A hub firmware build with the display fix (the loudest crying cot).	`research/toolchain/pio_build_music.log`	`tail -5 research/toolchain/pio_build_music.log`
The ESP32-S3 firmware **compiles, flashes and passes its on-device verification** — see §3.	—	`tail -60 research/toolchain/pio_build.log` for the last build
Rebuild from scratch	`cd firmware/hub_infer && python -m platformio run`	~8 min from clean; ~1 min incrementally
Flash + read the log	`python firmware/tools/flash_hub.py`	finds the board, writes it, prints the report

Re-fetch the vendored deps	`bash firmware/tools/fetch_deps.sh`	clones the pinned commits and re-applies `patch_deps.py`
Install the node sketches into the Arduino IDE	`python firmware/tools/install_sketches.py`	copies `firmware/node/*` into `~/Documents/Arduino`
Build a node sketch without the IDE	`arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3 firmware/node/node_crynet`	`arduino-cli.exe` ships inside the Arduino IDE
Build and flash a node in one step	`python firmware/tools/flash_node.py --port COM38`	finds `arduino-cli` inside the Arduino IDE, builds the node sketch and uploads it; falls back to the sketchbook copy when the machine has no checkout
Ports	the hub on **COM32** (CH343), the node on **COM30** (`303a:1001`, the C3's own USB)	`python firmware/tools/collect_clips.py --list`

A background build survives `/compact` and even a lost session — it writes to the log file on disk, not to the chat. Compaction only rewrites conversation history.

What is NOT yet done — do not claim otherwise:

- Stage 2a is complete *for the inference core*: the front end and the model are verified against the host reference on real hardware.
- Stage 2b is **done and verified on the board**: the rewired INMP441 produces real audio (room RMS 6–11 milli, 9,237 distinct values), the node streams telemetry and uploads bursts, and both sketches discard the INMP441's power-on burst.
- **Stage 2c's radio is DONE and verified end to end on the real boards** (2026-10-30, late): the hub pulls a 2 s window over ESP-NOW and the CRC-32 it computes over the reassembled audio **matches the CRC-32 the node froze** (`286/286 fragments in 1.15 s`, `0x23C2EE17 -> MATCH`, then inference on the reassembled window). Getting there required four real fixes — a closed serial port starved the node's loop (USB-CDC blocks ~2 s per print; cure `Serial.setTxTimeoutMs(0)`), a memset in `handle_telemetry` wiped every slot's pre-allocated window buffer (no burst ever reassembled), the hub's probe flood (50 ms) kept resetting the node's window, and `esp_now_add_peer` ran from the node's WiFi-task callback. Unicast was tried and rolled back — broadcast at 10 Hz is reliable. **The TFT display and the BLE GATT server are now written and running on the hub together (2026-10-31): the panel comes up (320×240), the radio hears the node, and a BLE client connects and subscribes while the radio keeps counting frames with 0 bad frames — the conflict that stopped both was the hub's own 154 KB of boot-only static buffers, now in PSRAM (RAM 59.9 % → 12.9 %).** *Caveat added later the same day: "the panel comes up" only ever meant the driver returned success — the panel was in fact showing noise, because of three bugs in the display driver (little-endian address window, `CASET` sent twice, CS released between a command and its parameters). All three are fixed and reflashed; what the panel actually shows still needs one photograph.*
- The inference is 101.8 ms per window. That is 5 % of the 2 s window and fine for a triggered monitor, but the tensor arena is still in PSRAM; moving it into internal SRAM is the obvious next optimisation.
- **The chain works end to end (2026-10-31, late).** The hub pulls the node's two-second window over ESP-NOW (286/286 fragments, CRC-32 MATCH), runs the front end and the INT8 model on it, and announces the cry — the panel, the radio and the BLE server all run together on the hub, and a real phone received `ALERT {'cot': 1, 'alert': 'cry', 'confidence_pct': 100}`. The last blocker was the hub's own Bluetooth advertising deafening the Wi-Fi receiver; the fix is a per-fragment resend (`RESEND` frame, type 5) plus three supporting changes. See `DECISIONS.md` section 2.5.
- **In-domain audio exists** — four sessions were recorded with the real INMP441 (`research/in_domain/session1..4`): a quiet-room baseline, a phone-speaker cry session, a smoke test and a rescued partial. **And now the full confusion session (2026-11-01, D-60): nine more takes (`session5..10`, with `6b`, `7b` and `8b` for the close-and-loud versions) measured in this room — 0 false alarms in 308 non-cry windows, 86.4 % recall on a cry at 15 cm; the report is `research/in_domain/SESSION_REPORT.md`.** The deployed INT8 model scores **0 false alarms in 236 quiet-room windows** on the baseline, and AUC 0.9283 / recall 0.724 / FPR 0.000 on the phone-cry session.
- **The false-alarm map exists (2026-10-31, new).** `research/eval_esc50.py` ran the *deployed* INT8 model over ESC-50's 2,000 real-world clips: **36 of the 40 real `crying_baby` clips alarm (90 %)**, and **38 of the 1,439 loud non-cry clips do too (2.64 %)**. Worst groups: urban 4.3 % (`car_horn` 6 of 38 reached, `hand_saw` 4 of 37, `church_bells`), animal 3.7 % (`crow` 4 of 32, `sheep`, `frog`), domestic and human non-speech 1.6 %, nature 0.8 %. Report: `research/out/eval_esc50.json`.
- **There is no music anywhere in the training corpus.** ICD, ESC-50, the 4-class cry set and the snoring set hold no music at all — and ESC-50, the corpus the map above uses, contains **no music and no speech** either (it is environmental sound plus human non-speech vocalisations). So "the model tells a cry from music" was **untrained and unmeasured**. Speech *is* in the training set (VoxCeleb, 4,963 clips); music is not.
- **Music was measured, and the model as deployed then failed it — the retrain fixed it (2026-11-01, D-58).** `research/eval_music.py` had run the deployed INT8 model over GTZAN's 1,000 real music clips (ten genres): **79 of the 965 clips that reach the model alarm — 8.19 %**, three times the rate for any other kind of sound. Worst: jazz 14.3 %, blues 13.0 %, pop 12.0 %, disco 10.0 %, rock 10.0 %; best: reggae 2.0 %, classical 2.9 %, country 3.0 %, metal 4.0 %. The cause is plain once seen — a cry is a voiced, harmonic-rich sound and a saxophone or a sung vowel looks like one in the log-Mel domain. **A threshold change cannot fix it:** the INT8 output saturates at P(cry) = 0.9961, so 45 of those 81 music alarms and 34 of the 36 true cry detections carry exactly the same value. The fix is a training round, and the corpus is now in the notebook.

Next concrete steps:

1. **DONE (2026-11-01, evening): the Kaggle re-run happened and the retrained model is deployed and verified on the board.** The user ran the notebook, `crynet_out.zip` came back, and the retrained model was measured, deployed, and verified 4/4 on-device (D-58). **Music false alarms are now 0.21 % (2/965), down from 8.19 %**; ESC-50 non-cry alarms halved (2.64 % → 1.18 %); ESC-50 crying-baby recall is unchanged (36/40); the quiet room is still silent (0/236). The known cost: per-window recall on the phone-speaker cry sessions dropped (session 1: 0.867 → 0.800; session 4: 0.724 → 0.592) — a sustained cry still trips many windows in a row, so episodes are still caught. What is *not* yet re-measured is O-9, the in-room confusion session.
2. **Look at the TFT panel - it is wired, flashed, and the text now renders as letters.** The panel is on CS 10, MOSI 11, SCK 12, DC 13, RST 14, backlight 4, 3V3, GND; SPI2 at 40 MHz. The first flash came up as **static noise** (three bugs in my driver: little-endian address window, `CASET` sent twice, CS released between a command and its parameters), the second came up **a quarter turn out** (this panel is landscape-first and wants `MADCTL_MY`, not `MADCTL_MV` - the legacy firmware's setting), the third had the **new UI** but drew **hearts and diamonds instead of letters** (the font generator was reading the table 32 characters out of place), the fourth added the **caregiver's device on the panel** (`PHONE` pill, and `ALERT NOT DELIVERED` under a confirmed cry when nobody is attached), and the **fifth - the first photograph of the real glass - confirmed the layout, the text and the orientation are all correct**: `CRYNET`, `S3 HUB`, `RADIO`, `PHONE`, `LISTENING`, `COT 1 / ONLINE`, `COT 2` and `COT 3 / NO SIGNAL`, `3 cots linked`, `up 0:39`. **The same photograph exposed a colour fault: every colour had red and blue exchanged** (the teal accent came out olive, the amber `PHONE` pill came out blue) because the driver set the panel's `MADCTL_BGR` bit while the framebuffer packs standard RGB565. That is fixed, flashed and logged (`madctl 0x80`). **The second photograph confirmed the fix**: the `PHONE` square is now warm orange, the accent underline is teal-green, the `RADIO` square is cyan-green - the same elements the previous photograph showed as blue, pink and olive. **What it also showed is a glare**: the panel is brighter than the room, so the camera clipped it and the white balance drifted, and the whole screen read as saturated blue. The panel's side of that is now fixed - **the backlight is a LEDC PWM output at 55 % by default** (`hub_display_set_brightness()`), and the **BOOT button opens a colour check**: a grid of the palette's swatches, which is how the panel's own colours can be judged without a camera in the loop. **The colour check then proved the hues are right and found the real fault**: `BLACK` came back black, `WHITE` white, `TEAL` teal, `GREEN` green, `AMBER` amber, `ROSE` rose - but the old navy background came back as a **saturated blue**, because this panel **lifts the dark end, and the blue channel hardest** (the amber's requested blue of 36 measured back at 145). **The neutrals then moved one step darker again - on the user's own pick**: shown

the ramp, they chose its **third step, `(4,8,8)`, as the teal the screen should be**, and the whole neutral family moved down one step of the same ramp with it (background `(4,8,8)`, surfaces `(7,15,16)` and `(12,26,28)`; border and dim text kept, every contrast re-checked). **The same round fixed the rounded corners**: every corner had been drawing its arc on the *inside* of the shape - only the bottom-right one looked right - because the corner loop walked its offsets inward from each disc's centre; each disc is now walked *towards its own corner*, and a host-side regression check (`research/check_corners.py`) requires the four corners to be exact mirror images of each other. **The repo is now pushed to GitHub after every change** (remote `crynet`; eight commits had piled up locally before the first push). **The panel then had its second look from the chair (D-59):** the background dropped another step of the same ramp — `(4,8,8)` → `(2,4,4)` — because the cards still read as the same colour as the ground on the glass; the header strip got its own colour `(7,20,15)`, green pushed five steps above blue, because the backlight's LEDs run along the top of the glass and lift that strip blue (a header painted the same colour as the cards photographed blue while the cards photographed teal); and the online/offline treatments were swapped as the user asked — a live cot now carries the **full-strength vibrant teal outline** (rose when crying) while a no-signal card recedes, its outline and empty bar washed towards the background, the reverse of the 45 % blend the live cards had before. **One more pass from the chair (D-59, amended):** the *fills* are swapped - a live card wears the border-washed fill the offline card had (the denser, more active one) and an offline card the faint accent wash, outlines unchanged - and the dead card's word became **OFFLINE**. **What is still wanted is one photograph of the colour check with the room lit**, so the ramp yields the panel's own tone curve: in the dark the camera's veil is stronger than the steps it is supposed to measure (the fourth photograph proved that too - its red channel jumps the wrong way between adjacent ramp steps).

69 3. **DONE (2026-11-01, later at night): the burst is 2.06 s and the panel shows a cry about two and a half seconds after it happened.** The cause was the hub's own BLE advertising deafening the Wi-Fi receiver (NimBLE's default interval is 30-60 ms on all three channels, ~100 transmissions a second). Three changes: the Bluetooth stack advertises only when it must (never with a phone connected, never while a burst is in the air), at 100 ms instead of 30-60 ms when it must; and the node paces its fragments every 4 ms instead of every 10 ms. **Measured: 5.54-6.93 s before, 2.06-2.11 s after, 286/286 fragments every time, with no resend rounds.** The request-to-verdict latency is 2.23-3.57 s (D-61).

70 4. **DONE (2026-11-01, night): the confusion session is recorded — O-9 is closed.** Nine one-minute takes with `node_capture` on the real INMP441 (the procedure is in `research/CONFUSION_SESSION.md`; the result is in `research/in_domain/SESSION_REPORT.md`). **0 false alarms in 308 windows of non-cry audio** (speech, music and film — including a loud film and loud music where the trigger fired on 46 % and 84 % of the records, so the model was asked 44 times in each and refused every time), and **86.4 % per-window recall on the phone-played cry at 15 cm** (the realistic case: the mic is in the cot). The one finding that changes the design: **the cry at one metre never crosses the 0.030 trigger** (median RMS 0.0070, 0/590 records), so the system's reach is limited by `CRYNET_TRIGGER_RMS` and the microphone's fixed sensitivity, not by the classifier. The clips are now the hard negatives for the next training round.

71 5. **DONE (2026-11-01, last thing tonight): the demo's sound path is verified on the live hub.** The user played the cry from a phone 10-15 cm from the node's mic while the hub's log was read live: **42 verdicts, 26 "ALERT — cry" at P(cry) 0.9922-0.9961, 16 "no cry" at P(cry) 0.0078-0.2148**, nothing near the 0.95 threshold, and the link never dropped a fragment (45 bursts, 45/45 complete, no stall, timeout, CRC error or resend round). Request-to-verdict 2.23-5.35 s (mean 4.03 s). The burst's length is now a range (2.06-5.19 s, median 3.55 s) because the per-fragment cost is the *channel's* contention on the shared channel 6, not the pacing. **The demo must use the phone** — the laptop's speakers are not detected (D-61). Details: D-62.

72 6. **DONE (2026-11-01, late): the adaptive trigger is built on both boards and it works — and it exposed a fault in the model's judgement of near-silent windows, which the announcement rule now fixes.** The multiple was chosen from the recorded sessions, not by taste (`research/adaptive_trigger_study.py`): at **three times the node's own noise floor** the cry a metre away goes from 0 trigger events a minute to 39, the quiet room gives 5, and the loud sessions are capped by the old fixed 0.030 — so the rule is never worse than before. **The node owns the floor and the level** (it freezes the window, so its level is the one that matters); the level travels in the telemetry (payload 7, milli-RMS) beside the frozen window's own level and the floor, and the hub triggers on the number the node reports, falling back to `CRYNET_TRIGGER_RMS` for an older node. The floor falls at once when the room gets quieter and rises with a 50 s time constant, so a long cry cannot drag the floor up to its own level and switch the trigger off in the middle of an episode. **Measured on the boards:** the level tracks the room (0.006–0.016 in a noisy room), the node's own log shows it settling (`rms 0.0063 floor 0.0028 level 0.0083`), and the hub now asks 5–11 times a minute instead of 1–2. **But a lower trigger asks the model about near-silent windows, and the model is not a reliable judge of those:** the same quiet room produced verdicts of 0.0000 and of 0.9766 for windows that differ by nothing a person could hear, and the second of those was a false alarm. **So the announcement rule changed (D-64):** one window announces a cry only when it is unmistakable — P(cry) ≥ 0.99, where the model's output saturates at 0.9961 for a clear cry, so the panel still shows the cry two and a half seconds after it happened — and anything between 0.95 and 0.99 is a *maybe* that has to be confirmed by the next window. A cry lasts for seconds and its windows agree; a noise fluctuation that happens to look like a cry is not repeated. **And the node's transmitter was turned down to 8.5 dBm (D-65):** the radio sits centimetres from the microphone and was putting its own noise into the audio it was about to be judged on. Five minutes of a noisy room afterwards: 32 verdicts, **0 false alerts**.

73 7. **NEXT: the user's verification session, all of it in one go.** The cry from the phone at **15 cm** (the demo case — it must announce at once, at the old two-and-a-half-second speed) and at **one metre** (the case the adaptive trigger was built for — it must now reach the model at all), with the **node's own serial log captured alongside the hub's** so the floor, the level and every frozen window's own level are on the record. The tool that made this possible is new: `firmware/tools/dump_node_window.py` writes the node's frozen window out over its USB console, and `research/analyze_window.py` says what is in a window and what the model makes of it.

74 8. **Two nodes in one room.** The per-node path already exists — per-node inference, a per-node BLE alert and a `crying_mask` in the state characteristic — and the panel now names the **loudest** crying cot, but it has never been run with two nodes, and the demo has to show *which* cot the cry came from.

75 9. Then Stage 2d (the wearable).

76

77 **Tools built so far (all runnable on the host):**

78 - `firmware/tools/check_artifact_drift.py` — verifies `research/out/` against `deploy_config.json` and the staged firmware copies. Currently **PASS**.

79 - `firmware/tools/gen_probe_refs.py` — regenerates `crynet_probe_refs.h` (expected on-device features) from the notebook's own front end.

80

81 ---

82

83 ## 1. Project in one paragraph

84

85 Multi-node infant cry monitor. 3× ESP32-C3 cot nodes (INMP441, I2S 16 kHz) → ESP-NOW → **ESP32-S3-DevKitC-1 hub** (TFLite Micro INT8 inference, ILI9341 TFT, BLE GATT) → wearable (ESP32-C3 + OLED + DRV2605L haptic). Model is a **binary cry / no-cry** classifier on 2 s log-Mel windows. Previous firmware attempt failed after flashing, so the project was rebuilt from model training.

86

87 ---

88

89 ## 2. Frozen decisions (do not re-litigate)

90

91 | # | Decision |

92 |---|---|

93 | 1 | Binary cry / no-cry head (BLE packet `alert_type 0x01` only supports binary). |

```
  94  | 2 | Label policy = "any cry present" in the 2 s window (≈ ≥ 3 active frames). |
  95  | 3 | Training runs on Kaggle GPU (user executes); notebook validated locally first by the agent. |
  96  | 4 | Signal contract (below) is frozen — firmware, C code and weights all depend on it. |
  97  | 5 | Trigger threshold must drop from RMS ≥ 0.18 → ~0.03 in the hub (old value was a real bug: soft cries never reached a classifier). |
  98
  99  **Signal contract (frozen):**
 100  `int16 pcm[32000] (2.0 s @ 16 kHz mono)` → `/32768` → subtract mean → zero-pad 256 front / 512 rear → 81 frames, 512-pt symmetric Hann, hop 400 → `|rfft|²` (257 bins) → 64 Mel bands (librosa slaney, 20 Hz–8 kHz, 497 non-zero weights) → `10*log10(max(P,1e-10))` floored at −80 dB → per-window standardisation (CMVN) → int8 quantise → TFLM → `[P(no_cry), P(cry)]`.
 101
 102  | Constant | Value | Constant | Value |
 103  |---|---|---|---|
 104  | 2026-11-01 (night, final) | **The shipped firmware verified on a one-minute play: 7 live ALERTs, every one P(cry)=0.9961, first ~8 s after the first cry-level freeze; 26/26 bursts complete; zero false detections before, during, or after.** Windows straddling the recording's pauses scored 0.80-0.94 and were refused cleanly; two impulsive non-cry ticks (crest 10-18) were refused by the model itself (P=0.03-0.09). The momentum fix (D-74) passed its first live test - the post-play quiet windows never announced. This closes the demo verification: play at panel level ~0.10-0.20, alerts at saturation, silent otherwise. | (no code changes - verification only) | *Logs: %TEMP%/hub_v13.log, node_v13.log.* |
 105  | 2026-11-01 (night, latest) | **The volume-down play verified the whole chain: five for five cry windows announced at 0.9922-0.9961 with no clipping (peaks 57-69 % of full scale), and the quiet watch held zero live ALERTs while refusing 31+ flukes - including two full saturations (0.9961 at crest 2.7 and 2.8) that the crest gate stopped. One residual defect found and fixed (D-74):** 12 s after the cry ended, a quiet hiss window (rms 0.0096, crest 2.5, P=0.9883) announced on the confirming run's momentum - the cry's own loud windows had advanced the run to 2, and quiet windows that score >= 0.95 keep it alive without resetting it. The run now confirms only windows with real energy (rms >= 0.020): quiet windows keep the run alive but cannot announce. Flashed; final quiet check running. | `firmware/hub_infer/src/link_stage.c` | *Session verdicts: quiet watch ~7 min, 0 live ALERTs; the play: first cry window (seq 183, loud onset) scored 0.9688 (a maybe - unconfirmed), every subsequent cry window 0.9922-0.9961 (ALERT), post-play flukes all refused.* |
 106  | 2026-11-01 (night, later) | **Four more quiet-room false alarms in nine minutes; the model saturates on pure mic self-noise; the crest gate closes it (D-73).** The quiet-watch session after D-72 caught the flukes live: rms 0.0067-0.0088, peaks 621-789 (crest 2.3-2.9 - flat Gaussian hiss), scoring P(cry)=0.9922-0.9961 - the same value as a real cry - roughly every two minutes. No confidence threshold can fix this (saturation is saturation); the discriminator is the **crest factor**: every falsely-high quiet window ever measured has crest 2.2-2.9, the quietest genuine cry (one metre, P=0.9961) has 5.6. `CRYNET_QUIET_CREST 3.5`: a quiet saturated window announces only with a cry's impulsive shape; loud windows (>= 0.020 rms) announce on saturation alone (clipping flattens the crest on loud cries, 1.9-3.9, so the shape test must not apply there). Flash verified: boot self-check PASS. **Verification in progress** - six quiet minutes expected to produce zero ALERTs (the room flukes ~every 2 min, now refused), then the volume-down play (panel 0.10-0.20). | `firmware/hub_infer/src/{link_stage.c,link_stage.h}` | *The build also prints crest on every window line, so future sessions measure it directly.* |
 107  | 2026-11-01 (night) | **Fourteen quiet-room false alarms, found and closed (D-72).** The user saw ~14 "cry detected" flashes at 95-98 % with no cry playing (panel levels 0.002-0.012). Mechanism: in a quiet room the adaptive trigger pins at its minimum (0.005), the hub judges near-silent windows continuously, the model's verdict on those is documented noise, and two consecutive scattered maybes (0.95-0.98) confirmed each other under the run-of-two rule - the rule's "a noise fluctuation is not repeated" assumption fails when the room repeats the fluctuation every few seconds. **Fix (hub only):** `CRYNET_ANNOUNCE_MIN_RMS 0.006` (no window under 0.006 rms announces at all, even at >= 0.99; the one-metre cry at 0.0103 stays above it) and `CRYNET_MAYBE_MIN_RMS 0.020` (a maybe counts toward the confirming run only with real energy behind it). **Verified live:** within 90 s of flashing, the room produced `rms 0.003917  P(cry)=0.9531` - the exact failure shape - and it logged `cry (too quiet to announce)` instead of alerting. Real-cry paths unchanged: saturated verdicts announce alone at >= 0.006 rms; the demo-volume re-test (phone quieter, panel ~0.10-0.20) is still pending. | `firmware/hub_infer/src/{link_stage.c,link_stage.h}` | *Hub re-flashed (COM32), boot self-check pass; the 90 s capture also confirmed the flood is real - the hub judged ~8 windows, all refused or refused-by-model.* |
 108  | 2026-11-01 (later at night) | **The burst is 2.06 s instead of 5.5-6.9 s, and the panel shows a cry about two and a half seconds after it happened instead of eight.** The cause was the hub's own Bluetooth advertising: one radio carries both Bluetooth and Wi-Fi, so every advertisement is a moment when the Wi-Fi receiver cannot hear the burst's fragments - and NimBLE's default advertising interval is 30-60 ms on all three channels, about a hundred transmissions a second. **Three changes:** the Bluetooth stack now advertises only when it must (never while a caregiver's phone is connected, never while a burst is in the air - the link stage tells it, and the log shows it stopping 1 ms after the request and returning 181 ms after the burst lands); when it does advertise, it advertises at 100 ms instead of 30-60 ms; and the node paces its fragments every 4 ms instead of every 10 ms, because that ten-millisecond gap was only ever needed to survive the deafening that is now gone. **Measured on the boards** (`firmware/tools/measure_burst.py`): before 5.54 s and 6.93 s with resend rounds; after the Bluetooth policy 3.76 s and 3.81 s with no missing fragments and no resends at all; after the node's pacing **2.06 s, 2.06 s, 2.11 s**, 286/286 every time. The request-to-verdict latency is now 2.23-3.57 s (mean 2.68 s), of which 0.16 s is the model's own inference. **A finding to settle with the user:** played from the *laptop's* speakers the cry is *not* detected - P(cry) 0.38-0.95, just under the threshold, and turning the volume to 100 % made it worse while the node's RMS stayed at 0.045 - so it is the speakers' frequency response, not the level; the demo's source is the phone, and the phone path on the live hub is being checked. | `firmware/hub_infer/components/ble_alert/ble_alert.c`, `firmware/hub_infer/components/ble_alert/include/ble_alert.h`, `firmware/hub_infer/src/link_stage.c`, `firmware/node/node_crynet/node_crynet.ino`, `firmware/tools/measure_burst.py` (new), `firmware/tools/measure_latency.py` (new), `DECISIONS.md` (D-61) | The boot self-check still reads PASS (4 probes, 5184/5184 exact), a BLE client still finds the hub, connects and subscribes to both characteristics, and the burst is now bounded by the node's radio rather than by the hub's Bluetooth. || 2026-11-01 (night) | **The in-room confusion session is recorded — O-9 closed. The honest in-room number: **0 false alarms in 308 windows of non-cry audio**, and **86.4 % per-window recall on the phone-played cry at 15 cm**; the limit on how far the hub can hear is the trigger, not the model.** Nine one-minute takes with `node_capture` on the real INMP441 (quiet room; speech at desk distance and at 30 cm; music and a film at 1 m and at 30 cm; the phone-played cry at 15 cm and at 1 m). **The model raised no alarm on any non-cry window** — 308 windows of speech, music and film, 0 false alarms — and the two loud takes are the ones that count: the trigger fired on 269/590 records of the loud film and 493/590 of the loud music, so the model was asked 44 times in each and refused every time. That is the retrain (D-58) working in the room it was meant to work in: on GTZAN the music false-alarm rate was 8.19 %, and on the loudest music this room can produce it is 0/44. **The cry at 15 cm — the realistic case, the mic is in the cot — is caught in 38 of 44 windows.** **The cry at 1 m (phone at maximum volume) reaches a median RMS of 0.0070 and 0 of 590 records cross the 0.030 trigger, so it never reaches the model at all** (asked anyway, the model would have caught 18/44): the system's reach is set by software, not by the microphone: **the INMP441 has no gain setting** - its L/R pin is a left/right *channel select*, not a gain strap, and the part is fixed at -26 dBFS +/-3 dB sensitivity - so the levers are the hub's trigger threshold and a digital gain in the telemetry path, and the better version of the trigger is an *adaptive* one (a multiple of the node's own running noise floor, 0.0020 here, so 3x = 0.006 would pass the one-metre phone). Lowering the trigger to ~0.012 is the same lever from the other end and the evidence supports trying it in the next training round — it is **not** changed on 308 windows from one afternoon. The labelling tool grew a `--truth` flag, because the old auto-label called a clip 'cry' if it was *loud*, which for the music and film takes would have labelled the loudest non-cry clips as cries and measured nothing. The node was flashed back to `node_crynet` afterwards, so the live chain is running again. | `research/in_domain/session{5,6,6b,7,7b,8,8b,9,10}/` (audio, gitignored), `research/in_domain/SESSION_REPORT.md` (new),
```

`research/out/eval_session*.json`, `firmware/tools/label_session.py` (`--truth`), `DECISIONS.md` (D-60, O-9 closed) | **The one measurement that uses this room's microphone, this room's gain and this room's acoustics.** The honest caveat is in the report: the cry is a phone speaker 15 cm from a mic in a cot, and a real baby in a real cot is a different sound. || SR | 16000 | N_FFT | 512 |
WIN_SEC	2.0	HOP	400
N_SAMPLES	32000	N_MELS	64
N_FRAMES	81	FMIN/FMAX	20 / 8000
CENTER_PAD	256	TAIL_PAD	512
DB_FLOOR	−80 dB	LOG_EPS	1e-10
CMVN	True	SILENCE_RMS_GATE	0.0015
TARGET_RECALL	0.95	SEED	1337

Mel weights are **exported from Python into C** (`crynet_tables.h`), so firmware cannot drift from the model.

---

3. Live status (as of 2026-10-29)

Checkpoint 1 = DONE, including the full Kaggle run. Checkpoint 2 (ESP32-S3 firmware) = Stage 2a in progress.

Done
- Dataset forensics: every candidate corpus downloaded and profiled at audio level (SR, duration, RMS, autocorr F0, spectral centroid/flatness).
  - **Used:** ICD `hoyyu1/infant-cry-detection` (PRIMARY, 28,478 clips, all 16 kHz mono 5.00 s, 5 shipped folds), Baby Cry 4-class (cry diversity + cross-corpus test), ESC-50 (negatives + 40 held-out `crying_baby`), Snoring (hard negatives).
  - **Rejected:** `mennaahmed23/baby-cry*` family (CONTAMINATED — fake "silence" at RMS 0.22, renamed ESC-50, synthetic derivatives), `rahult25/five-class...` (pitch-shifted derivatives), ICSD (gated HF), `AxonData` (paid), Baby Chillanto (request-only), AudioSet/FSD50K (redundant), raw DonateACry (8 kHz, skewed; already merged in cry4).
- 24-cell Kaggle notebook written, generated from `research/build_notebook.py`.
- **Validated end-to-end** on synthetic stand-ins (24/24 cells) and cells 1–15 on the real 28,478-clip corpus.
- **C front end == Python front end:** max deviation **7.06×10^{-5}**, mean 2.8×10^{-6}, over 4 probes (sine+harmonic, noise, DC-offset cry with INMP441 bias, silence).
- **Every op has a TFLM kernel** — asserted at export: `CONV_2D, FULLY_CONNECTED, MAX_POOL_2D, MEAN, SOFTMAX`.
- **INT8 vs float32:** AUC delta **−0.00007**, 99.70 % decision agreement.
- Notebook validated with `nbformat.validate()`.

Current numbers — **the full Kaggle run** (authoritative)

The user ran `notebooks/crynet_training.ipynb` on Kaggle (GPU T4) and the artifacts are in `research/out/`. These numbers supersede the earlier subsample figures.

Split	AUC	F1	Recall	Precision
ICD val (fold 4)	**0.9994**	0.9905	0.9844	0.9966
ICD test (fold 5)	**0.9995**	0.9905	—	—
cross-corpus (cry4 + ESC-50 + snoring, never trained on)	**0.9973**	0.9801	—	—

Variant	Params	val AUC	cross AUC
`conv_cnn` (selected)	122,770	**0.9994**	0.9973
`sep_cnn`	29,839	0.9974	0.9928
`temporal_cnn`	65,154	0.9805	0.9714

- Deployed model: `crynet_int8.tflite`, **137,384 bytes**, SHA-256 `4aef55d8cfc0fb75aa944e73945ef5b832f12d71479c48f82aff58563126bc11`.
- **Calibrated alarm threshold = 0.95** (re-derived by this run; the earlier subsample's 0.23 no longer applies).
- Input quantisation: scale **0.060125612**, zero_point **−4**. Output: `[1,2]` int8, scale 0.00390625, zp −128, classes `["no_cry","cry"]`.
- Tensor arena hint **120 KB** (the firmware reserves 256 KB and places it in PSRAM).
- C front end vs Python on the real run: max deviation **6.905×10^{-5}**, mean 2.79×10^{-6}.
- TFLM kernel check: **ok** — ops `CONV_2D, FULLY_CONNECTED, MAX_POOL_2D, MEAN, SOFTMAX`.
- **INT8 vs float32 — CLOSED.** The full run reported both AUCs as `NaN` because the parity cell took a *prefix* of the validation fold; the ICD manifest lists every positive clip before every negative one, and the fold's positive clips alone contribute 5,330 windows — more than the 4,000-window cap — so that slice was 100 % cry and its AUC was undefined. The model was never affected (every headline metric is computed over whole splits), but the quantisation delta for these weights was never measured.
  Re-measured independently on the host by `research/verify_int8_parity.py`, which rebuilds fold 4 from the cached archives with the notebook's own front end and runs **both** the exported INT8 model and the float32 Keras model over 11,386 windows (5,693 clips, centre-cropped — independent of the notebook's random crops):

	INT8 (as deployed)	float32 reference
val AUC	**0.997926**	**0.999299**
recall @ 0.95	0.9780	0.9782
FPR @ 0.95	0.0036	0.0036

  ΔAUC **−0.001373**; max $|\Delta p|$ 0.182; mean $|\Delta p|$ 0.0029; **decision agreement 99.956 %**. The independent float AUC (0.999299) corroborates the Kaggle figure (0.999419) from a different set of windows. Report: `research/out/verify_int8_parity.json`.
- **Notebook fixed:** the parity cell now draws a seeded random permutation subset and warns loudly if a slice is single-class. Notebook regenerated and `nbformat.validate()`-passed.

- **On-device verification — CLOSED (2026-10-30).** The Stage 2a firmware runs on the user's ESP32-S3-DevKitC-1 (COM32, CH343 USB-serial) and **all four probes pass**: the device's quantised features match the host reference exactly for silence, DC bias and the two-tone signal, and to within 1 LSB on 3 of 5,184 values for the noise probe. `RESULT: PASS — 4 probes, front end and model agree with the host`.

```
   | | value |
   |---|---|
   | PSRAM | 8,388,608 bytes, present |
   | flash | 16,777,216 bytes |
   | model | 137,384 bytes int8 |
   | front-end working set | 5,380 bytes, internal SRAM |
   | tensor arena | 262,144 reserved, **108,328 used** |
   | front end | **41.7 ms** per 2 s window (was 135 ms before optimisation) |
   | inference | **101.8 ms** per window (ESP-NN SIMD kernels linked in) |
   | CPU | 240 MHz, `-O2` |

   Four real bugs were found *only* by running it on the chip:
   1. **The DC-removal mean was accumulated in float32.** Summing 32,000 samples of
      18,000 in float32 gives 575,726,660 instead of 576,000,000, so the mean came
      out 8.5 counts low and 8.5 counts of DC survived. On the DC-offset probe that
      leaves a constant residual which the per-window standardisation blows up into
      a completely different feature vector — the probe failed with a max delta of
      131. The sum is now accumulated in `int32`, which holds it exactly.
   2. **The build was at `-Og`.** `build_flags = -DCOMPILER_OPTIMIZATION_PERF` is an
      *Arduino* macro that means nothing to PlatformIO's ESP-IDF builder, so
      ESP-IDF's debug default applied. The optimisation level is now set the
      ESP-IDF way, `CONFIG_COMPILER_OPTIMIZATION_PERF=y`.
   3. **The CPU ran at 160 MHz**, not 240. `CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ_240=y`.
   4. **The front end kept a 128 KB padded copy of the signal in PSRAM.** PSRAM on
      the S3 is uncached external memory, and the FFT and the Mel filterbank stream
      through the working set 81 times per window. The padded copy is gone — each
      frame is now built straight from the caller's int16 PCM — and the working set
      is 5.4 KB in internal SRAM. Front end 135 ms → 41.7 ms.

   A fifth trap is not a firmware bug but a flashing one: **a combined image built
   with QIO flash mode makes the chip watchdog-reset in a loop** (`rst:0x7
   TG0WDT_SYS_RST` repeating, no application output). The ROM cannot read this
   board's flash in QIO before the bootloader has configured its own timing. The
   combined image must use **DIO**. In fact the whole build is DIO — the effective
   config is `CONFIG_ESPTOOLPY_FLASHMODE "dio"` and both `bootloader.bin` and
   `firmware.bin` carry flash-mode byte `0x02` — so the application runs in DIO as
   well, which costs nothing at this workload. See `firmware/README.md`.

 - **Stage 2b — the node firmware is written and compiles (2026-10-30), but has
   not run on the board.** Two Arduino sketches for the ESP32-C3, built with the
   Arduino IDE's own `arduino-cli` against core 3.3.12, **zero warnings**:

   | Sketch | Flash | Static RAM | What it does |
   |---|---|---|---|
   | `firmware/node/node_crynet/` | 977,617 B (74 %) | 165,972 B (50 %) | the node: I2S capture, a 2 s rolling window, 100 ms RMS telemetry over ESP-NOW, the 64 KB burst in 286 CRC-framed fragments |
   | `firmware/node/node_capture/` | 311,120 B (23 %) | 18,592 B (5 %) | streams the microphone to the PC so in-domain audio can be recorded |

   The I2S setup is Espressif's own recipe for this microphone — **32-bit slots**
   (the INMP441 puts 24 bits into a 64-bit frame, so 16-bit slots would misalign
   the data), `I2S_SLOT_MODE_MONO` with `I2S_STD_SLOT_LEFT`, and the narrowing to
   int16 done in software by keeping the top 16 bits. The ESP32-C3 is I2S
   hardware v2, so the left-channel slot is selected in hardware, with no
   software decimation.

   The node repeats the hub's DC lesson: the mean is an exact **integer** sum,
   and the 100 ms RMS is computed with the mean of the **whole 2 s window**
   removed, so the number the node reports is the same quantity the hub's front
   end will compute for that window — otherwise the hub's 0.03 trigger threshold
   means nothing.

   Two host tools came with it: `firmware/tools/verify_burst.py` asks the node
   for its window over the serial console, checks every frame's CRC and writes
   the result as a WAV; `firmware/tools/collect_clips.py` records the session and
   cuts it into 5 s clips. `firmware/tools/install_sketches.py` copies the
   sketches into the Arduino sketchbook.

   **Not yet proven:** that the wiring is right, that the INMP441's actual level
   and DC offset land in a sensible range, and that the effective sample rate is
   within tolerance of 16 kHz. All three are measured by `verify_burst.py` the
   first time the node runs.

   The **wire format was validated numerically** before any hardware was
   involved: CRC-16/CCITT-FALSE reproduces the standard check vector `0x29B1`,
   and the C frame builder and the Python parser agree on every header offset
   and on both CRCs (the frame's CRC covers its header *and* its payload; a
   request's CRC covers its first 10 bytes).

 - **Firmware build — CLOSED (2026-10-30).** `firmware/hub_infer` now **compiles and links** for `esp32s3`:
   - toolchain: PlatformIO 6.2.0, `platform-espressif32` 8.0.0, **ESP-IDF 6.1.0**, `toolchain-xtensa-esp-elf` 14.2.0 (GCC 14.2);
   - images: `firmware.bin` **468,800 B**, `bootloader.bin` 22,560 B, `partitions.bin` 3,072 B;
   - **RAM 52.1 %** (170,668 B of 327,680 B) and **flash 22.3 %** (468,373 B of the 2 MB factory partition);
   - effective config: `esp32s3`, 16 MB flash, **octal PSRAM at 80 MHz**, custom 16 MB partition table, and **`CONFIG_ESP_TFLITE_MICRO_USE_ESP_NN=y`** — the ESP-NN SIMD kernels are compiled in, which is the difference between roughly 40 ms
```

and roughly 2.3 s per convnet invoke on an S3;
   - the whole TFLite Micro port and ESP-NN compile cleanly under GCC 14 with the port's own `-Werror`;
   - `firmware/tools/check_artifact_drift.py` still **PASSes** (model SHA-256, threshold, quantisation, staged copies);
   - `firmware/hub_infer/dependencies.lock` records the exact IDF version and is committed.
   Still unverified: anything that only happens on the chip — the FlatBuffer alignment, the arena fitting, the probe feature agreement and the latency.

Artifacts on disk
`research/out/` — **the authoritative full-run artifacts**: `crynet_int8.tflite`, `crynet_float.tflite`, `crynet.keras`, `model_data.h`, `crynet_tables.h`, `dsp_crynet.c/.h`, `crynet_infer_example.c`, `deploy_config.json`, `metrics.json`, `c_parity.json`.
`research/_realfast_out/` — the earlier subsample run, kept for comparison.
Logs: `research/real_fast.log`, `research/stub_run.log`.

Git state and remotes
Local branch `main`, tracking `crynet/main`. **The node produces real audio on the real hardware, and both sketches discard the INMP441's power-on burst** (see §3, Stage 2b). All source committed; a few line-ending-only diffs may remain from the editor.

Remote	Points at	Purpose
`crynet`	`github.com/Amazo123/CryNet.git`	**The live project.** Every change is committed here and pushed. Created 2026-10-29, **public** since 2026-10-29.
`legacy`	`github.com/Amazo123/Multi-Node-Infant-Cry-Detection.git`	The **old** project (Arduino firmware, BOM, wiring guides, old `model_data.h`). Read-only reference — do not push to it. Its hub `.ino`, `firmware/common/protocol.h`, `firmware/central_hub_s3/pins.h` and `dsp_mel_extractor.cpp` are the starting point for checkpoint 2.

Backup rule: after every meaningful change → update this file → `git add -A && git commit && git push crynet main`. Never commit `research/_cache/` (5.5 GB of corpora) or the `_*_work/` scratch dirs; they are gitignored and the notebook re-downloads the corpora itself.

What is *not* in GitHub: the raw datasets (5.5 GB — re-downloaded automatically by the notebook) and the `_work`/`_parity` build scratch. Everything that is source, model, exported artifact, doc or log **is** in the repo.

Open risks
1. **Unverified on hardware.** A successful build says nothing about the chip: the FlatBuffer alignment, whether the 256 KB arena actually fits, whether the DSP reproduces the Python features, and the real latency are all still unknown. This is the top risk, and the reason Stage 2a exists.
2. `coughing` is the top confuser; if it survives mining, add the FSDKaggle2018 cough set.
3. **No in-domain data** — the model has never heard the user's INMP441, room or gain. ~50 self-recorded clips (cries + room noise + speech + TV) is the single biggest remaining accuracy win.
4. TFLM latency unmeasured. With ESP-NN enabled the expectation is tens of milliseconds per 2 s window on a 240 MHz LX7, not the 100–300 ms the un-optimised kernels would cost; it must be measured on the real board.
5. The tensor arena is a 256 KB reservation placed in PSRAM; `AllocateTensors()` has never run. The notebook's hint was 120 KB, so it should fit — but the *actual* high-water mark is unmeasured.
6. BLE/display policy unchanged from the old spec; it already assumes binary output.
7. Old firmware gotchas still apply: INMP441 DC offset (~+18000 counts), ESP-NOW TX queue overflow (needs callback flow control), Arduino-3.x callback signature change, ILI9341 240 px clipping (the `HubDisplay` subclass).
---

4. Next action (checkpoint 2 — ESP32-S3 firmware)

> **Live pointer (2026-11-04, evening):** the plan below is the original stage plan and is *behind* the work. The current state is the newest entries in §5 *Change log* — read those first. **Where the project stands now:** the whole bench is on one laptop — the three cot nodes on a USB hub, the hub and the wearable straight in — and `firmware/tools/capture_multi.py` watches all five boards at once. The branch `feature/stop-capped-freeze` is **merged into `main`** (main = `bfd65c4`, pushed to GitHub) — it carries the hybrid freeze rule (now flashed to **all three nodes**) and the flapping fix, and the merge was made after the bench verification: **three cries, three buzzes, the right cot each time, the name never changed hands** (`research/out/demo_run4/`). On top of that this evening the hub gained: **the saturation-only announcement rule** (a conversation in the room scores 0.9727 and the old loud path would have announced it), **the refusal of clipped windows** with a loud warning, **a burst stall noticed in 100 ms instead of 1.5 s** (median window fetch 2.48 s → 1.88 s), and **a console** (`help`, `status`, `alert <cot>`, `clear`) that proved the BLE path end to end — the wearable printed `ALERT cot 2 type 1 conf 100%` with no sound at all. **The open items, in order:** (1) **the model on loud speech** — the corpus has no loud-speech negatives, and **the user has no one around to record a conversation**, so the speech is to be **sourced online** (real conversation recordings), **measured against the deployed model first**, and a training round run only if that measurement shows the margin is thin; (2) **the live clip test** — the three-cot attribution test is **DONE** (2026-11-04, `research/out/demo_run4/`: the cry played at each cot in turn, each named correctly, one buzz per cry), but the demo-kit clip has still to be played once at every cot in the kit's shortlist to prove *those* clips announce from the distance the demo will use. **Closed since the last entry: cot 2's microphone.** It is not a hardware fault — the "clipping and DC offset" was the phone at ~10 cm from the mic (the INMP441 is fixed-gain and saturates before the node's 16-bit window does) and a small DC offset that is normal for the part and removed by the per-window CMVN. At the kit's 15–20 cm, node 2's window was clean (rms 0.1058, peak 16918, crest 4.9) and it announced at 0.9961 with the wrist naming cot 2.

4.0 Kaggle outputs — received and staged
The user placed the full-run `out/` at `research/out/`. `firmware/tools/check_artifact_drift.py` verifies the SHA-256, model size, quantisation, thresholds and the staged copies. **Current result: PASS** (137,384 bytes, SHA-256 `4aef55d8…126bc11`, threshold 0.95, input scale 0.060125612 / zp −4).

File	Used by	Why it matters
`crynet_int8.tflite`	reference	the deployable model; `model_data.h` is the same bytes as a C array
`model_data.h`	hub	`crynet_model[]` + `crynet_model_len` — compiled into flash
`crynet_tables.h`	hub, node	Mel filterbank (start/end/offset + weights), Hann window, all constants, quant scales, **calibrated threshold**, silence gate
`deploy_config.json`	hub	input shape, quantisation params, operator list, TFLM kernel check, threshold, metrics, model SHA-256
`metrics.json`	docs	per-variant AUC / F1 / recall
`c_parity.json`	docs	measured C-vs-Python front-end deviation

| `crynet_float.tflite`, `crynet.keras`, `dsp_crynet.c/.h`, `crynet_infer_example.c` | reference | float reference, retrainable model, the portable C front end, TFLM glue template |

`research/_realfast_out/` stays as the *subsample* run for comparison; `research/out/` becomes the **authoritative** one.

4.1 The firmware is written from scratch — what that means concretely

The firmware is **not** a port of the abandoned Arduino code. ESP-IDF, new module layout, new transport, new application state machine. Explicitly:

Re-derived (legacy code is not reused):

1. **Application architecture.** The legacy hub was one 46 KB `.ino` with global state and a `loop()`/FreeRTOS mix. New: ESP-IDF with separate components (`crynet_dsp`, `crynet_model`, `crynet_infer`, `node_link`), a dedicated inference task, and an explicit event-driven state machine.
2. **The node↔hub transport.** The legacy telemetry had no ack, no retry, no per-fragment integrity check — which is exactly why it lost audio packets. New: my own packet layout with session IDs, per-fragment sequence numbers, CRC, and retransmission; the hub tolerates loss by design.
3. **The DSP front end.** The legacy `dsp_mel_extractor.cpp` implements the *old* contract (128 mel, HTK scale, no normalisation) and is incompatible with the new model anyway. New: the notebook's `dsp_crynet.c` (this rebuild's own code, already validated against Python to 7.06 × 10^{-5}), ported to ESP-DSP `dsps_fft2r` + `dsps_cmplx_mag`.
4. **Trigger and decision logic.** Legacy: hard-coded `RMS ≥ 0.18` + `CONFIRMATION_PROB 0.65`. New: `RMS ≥ 0.03` trigger, 0.0015 silence gate, and the **calibrated** threshold from `crynet_tables.h`.

Kept deliberately (hardware facts, not design decisions):

1. The model's input contract (2 s, 16 kHz, 81 × 64 log-Mel, CMVN) — the trained weights require it exactly.
2. Physical pin assignments and the board wiring — facts about the user's real hardware, verified against the wiring docs, with the driver code written fresh.
3. The 3-node → hub → wearable topology and the binary alert semantics (`alert_type 0x01`).

4.2 Staged plan — one artifact per stage

Stage 2a — hub inference core. *(the risk-reduction stage: no radio, no display, no BLE)*
Minimal ESP-IDF app that links `model_data.h` + `crynet_tables.h`, runs the DSP front end and TFLM INT8 on a 2 s buffer, and prints `P(cry)`, inference latency and tensor-arena usage. Runs the **same four probes** as the Python parity test *on the target MCU*, so the DSP is verified on real hardware, not just on the host.
Exit criteria: compiles; runs on the S3; on-board front-end deviation vs Python < 1 × 10^{-3}; latency and arena measured and logged; arena moved to PSRAM if it does not fit in internal SRAM.

Stage 2b — node firmware (ESP32-C3). I2S INMP441 capture at 16 kHz, DC-offset removal, 100 ms RMS telemetry, on-demand 64 KB audio burst with fragmentation + CRC + retry.
*Status (2026-10-30): **done and verified on the board.** The rewired INMP441 produces real audio, the node streams telemetry and serves burst requests, and both sketches discard the microphone's power-on burst.*

Stage 2c — hub system. ESP-NOW receive, RMS trigger, audio assembly across 286 fragments, inference, TFT display, BLE GATT server, alert packet.
*Status (2026-10-30, late): the radio half is **verified end to end on the real boards** — the hub requests a window at RMS >= 0.03, reassembles 286/286 fragments in ~1.15 s, and the CRC-32 it computes over the reassembled audio matches the one the node froze (`0x23C2EE17 -> MATCH`); the front end and INT8 model then run on the window on the hub. Four bugs were found and fixed on the way (closed-serial-port loop starvation on the node; the telemetry memset wiping the slot's window buffer; a 50 ms probe flood; peer-add from the WiFi-task callback). **The TFT display and the BLE GATT server were finished and verified on 2026-10-31** — the panel comes up at 320×240, and a BLE client connects and subscribes while the ESP-NOW radio keeps running with 0 bad frames. Making both fit at once meant finding the hub's real memory problem: 154 KB of boot-only static buffers (the probe PCM and feature buffers) were holding internal RAM for the life of the program; they now live in PSRAM, and the hub's RAM went from 59.9 % to 12.9 %.*

Stage 2d — wearable (ESP32-C3). BLE notification client, SSD1306 OLED, DRV2605L haptic.

4.3 Toolchain status

This machine originally had **no ESP-IDF, no PlatformIO, no Arduino IDE and no Xtensa compiler**. PlatformIO **6.2.0** is installed (`research/toolchain/pio_install.log`) so the firmware can actually be **compiled and checked** here rather than written blind. Flashing still requires the user's physical boards.

What is installed now

Package	Version	Notes
`platform-espressif32`	**8.0.0**	pinned in `platformio.ini`
`framework-espidf`	**ESP-IDF 6.1.0** (`4.60100.0`)	installed **by hand** — see below
`toolchain-xtensa-esp-elf`	14.2.0	GCC 14.2
`toolchain-riscv32-esp`, `toolchain-esp32ulp`, gdb packages	—	for the C3 nodes and debugging
`arduino:esp32` (Arduino core)	**3.3.12**	the node sketches. `ESP_I2S`, `ESP_NOW` and `BLE` are all present in it
`arduino-cli`	**1.5.1**	ships *inside* the Arduino IDE (`resources/app/lib/backend/resources/arduino-cli.exe`) — it is how the node sketches are compiled and checked here, without opening the IDE
PlatformIO IDE (VS Code extension)	3.3.4	pointed at the **pip-installed** Core 6.2.0 (see below), so it drives the same toolchain every build in this repo came from and needs no network

The PlatformIO IDE in VS Code. The extension is installed and works with
the toolchain that is already here. `.vscode/settings.json` (at the repo
root and in `firmware/hub_infer/`) sets:

```json
"platformio-ide.useBuiltinPython": false,
"platformio-ide.useBuiltinPIOCore": true,
"platformio-ide.customPATH": "C:\\Users\\HP\\.platformio\\penv\\Scripts"
```

`useBuiltinPIOCore` must be `true`. The extension's compatibility

check is `get-platformio.py check core --version-spec ">=6.1.6"`, and it looks for a Core in `~/.platformio/penv/Scripts` — nowhere else. With the setting off it reports *"Could not find compatible PlatformIO Core. Please enable `platformio-ide.useBuiltinPIOCore`"*. The Core was installed there with PlatformIO's own bootstrap script:

```bash
python ~/.platformio/.cache/tmp/get-platformio-1.2.2.py
```

It is **6.2.0**, the same version the command line uses, and it shares `~/.platformio`, so ESP-IDF 6.1.0, the Xtensa toolchain and esptool are the ones already installed and **nothing is re-downloaded**. The hub builds through it in **99.3 s**. Open `firmware/hub_infer` as the folder and use the toolbar: **✓ build**, **→ upload**, **plug serial monitor**.

Do not pin the framework with a `platform_packages` URL. PlatformIO 6.2 installs a URL-specified package by treating the archive as an **npm package**: it picks the wrong `package.json` out of the tree, writes a bogus version into `.piopm`, and leaves `packages/framework-espidf` holding only a test-fixture package. CMake then fails with:

```
include could not find requested file:
  .../packages/framework-espidf/tools/cmake/project.cmake
```

The fix that worked: delete `packages/framework-espidf`, move the fully extracted tree into its place, and write `.piopm` by hand:

```json
{"type": "tool", "name": "framework-espidf", "version": "4.60100.0",
 "spec": {"owner": "platformio", "id": null, "name": "framework-espidf",
          "requirements": "~4.60100.0", "uri": null}}
```

Why ESP-IDF 6.1.0 and not 5.3.2. 5.3.2 was the original target, but PlatformIO's registry only offers the framework as `~4.60100.0` (6.1.0) with `~3.50505.0` (5.5.0) as the alternative for this platform, and the 5.3.2 archive could not be installed through PlatformIO at all. 6.1.0 was already on disk, satisfies the port's `idf >= 5.1` requirement, and needed no further download. **Fallback if the port misbehaves under IDF 6.1/GCC 14:** install `platform-espressif32` 6.9.0 (framework `~3.50301.0` = IDF 5.3.1, toolchain 13.2.0) and let PlatformIO fetch both.

4.4 User-side action in parallel

Done: the user ran the full Kaggle notebook and dropped `out/` into `research/out/`; the user connected the ESP32-S3 and the Stage 2a on-device verification **passed**.

Now, in order:

1. Upload `node_crynet` to the C3 — Arduino IDE → board **WeAct Studio ESP32C3** → port **COM30** — with the INMP441 wired as in `firmware/node/README.md`, then run `python firmware/tools/verify_burst.py --port COM30`. It writes the node's two-second window out as a WAV and reports the mean, the RMS and the peak — that is what tells us the wiring is right and what the microphone's DC offset and level actually are.
2. Upload `node_capture` and run `python firmware/tools/collect_clips.py --port COM30 --out research/in_domain/session1` to record the room, then ~50 real cry clips.
3. The hub's radio (Stage 2c) can then be written against a node that is known to be producing correct audio.

Both boards are already connected: the ESP32-S3 on COM32, the ESP32-C3 SuperMini (with the INMP441 on the breadboard) on COM30. The C3's USB is its own native USB, so *"USB CDC On Boot"* must be **Enabled** — the WeAct board entry has it on by default.

---

5. Change log (newest first)

- 2026-11-25 (**the branch is merged: `main` fast-forwarded `dd7a693` -> `dd85282` on the owner's explicit call -- not pushed yet.**) | The owner, after the D-108 fix, the D-107 soak and their own end-to-end cry test: *"i think you can merge now, as everything works as they should."* Fast-forward only -- `main` sat exactly at the branch's base, nothing to reconcile. The merged `main` carries the dashboard (offline + Worker), the cry log/net/clock components and uploader, the presence feature (30 s, burst-yielding), the banner fix, the 16 KB main / 8 KB net stacks, and the presence staleness fix. All three checks pass on the merged `main`; the branch is retained; **`origin/main` remains at `dd7a693` until the owner says push**.
- 2026-11-25 (**the owner's end-to-end cry test verified the D-108 fix live, and the hosted copy was finally read through its own front door -- D-108 follow-up.**) | The owner powered everything back on and ran cry tests end to end, and the hosted dashboard was checked with the caregiver login (the password now lives in the git-ignored `dashboard/.secrets`; a wrong password reads 401, the real one 200, and the hub's own token still answers `ok:true`): **presence all online and fresh** -- hub age 17 s, cot 1 -59 dBm, cot 2 -62 dBm, both words 17 s old -- which is the first live exercise of the fix's online side, after the offline side was proven on the stale rows. **Five cries landed in order, seq 111-115**, 06:43-06:47 local, both cots, 2-8 s, conf 100 % -- with cot 2 at 06:46:20 and cot 1 at 06:46:21, one second apart: the two-cot case recorded cleanly in the wild. Seven boot records (05:06-05:37) are this morning's power-ups, including two same-second doubles (05:06:24, 05:35:39) -- quick bounces during plug-in, worth remembering D-84's rule about boost and USB on one board. No deaf spells. Day total: 5 cries, 16 s, cots [1, 2]. A probe device row the token check had created was deleted; the devices table holds only `hub-f9172f80`. Nothing pushed, nothing merged into `main`.
- 2026-11-25 (**the presence panel could hold a dead hub's cots "online" for ever -- a cot is online now only while the hub's word about it is fresh -- D-108.**) | `build_presence()` in both `dashboard/worker/index.js` and `dashboard/server.py` judged the hub by the 90 s staleness window but served each cot row's `online` flag verbatim from the hub's last post -- so the post made just before a power cut held both cots "online" while the hub itself correctly read offline (the owner saw exactly that with everything powered down). The fix: a cot is online only if the hub said so **and** the row's `at` stamp is inside the same 90 s window; no stamp reads offline; demo exempt. Both copies kept identical; the page needed no change. `test_dashboard_contract.py` gained the presence section -- post the firmware's exact body, read back, age the store ten minutes, demand the cots read gone with their hub -- which fails on the old code with the owner's exact symptom and passes on the new. Worker redeployed (`d6deaeda`); the live store confirmed both cot rows carried `at` 2,251 s old, now aged out. No firmware change: prebuilt untouched, nothing flashed. Nothing pushed, nothing merged

into `main`.
413 - 2026-11-25 (**the hub really was crashing: a stack overflow in task main on the 3,584-byte default -- the stacks now fit what actually runs on them -- D-107.**) | D-106 said the resets were power-class; hours later its own still-running watch caught the crash in the act at 502.84 s (`***ERROR*** A stack overflow in task main has been detected`, `rst:0xc`, log kept at `research/out/hub_stack_overflow.log`). **The whole live system runs on task main** -- `app_main` never returns, and `link_stage_run()`'s `for(;;)` is the telemetry, burst handling, front end, inference, logging and BLE notify -- while 3,584 bytes was ESP-IDF's default for an `app_main` that spawns workers and returns; the sdkconfig comment even promised a larger stack, and no config line stood under it. The corrupted backtrace names no frame, so the fix is headroom, not a tuned path: `CONFIG_ESP_MAIN_TASK_STACK_SIZE=16384` in `sdkconfig.defaults`, and the net task 6,144 -> 8,192 because its HTTPS posts run a TLS handshake on that stack (the owner's rule: the connectivity feature may not cost the system anything). **What the crash is not:** it followed a presence post by 0.64 s, but posts run on the `crynet` net task's own stack -- 16 of 17 posts kept the exact 30 s cadence and one waited 33 s on the D-105 radio guard; presence stays. **The cot symptoms are cosmetic:** "offline for a split second" is the link layer's offline/online pair 110 ms apart at a staleness tick boundary (twice in 600 s, zero frames bad across either); "rms freezing" is telemetry-less frames carrying the last value between updates. **The soak on the fixed image (banner `dabb4e0-dirty`, boot 108): 600 s, zero resets, a 286-fragment burst completing clean in 0.98 s, 20 presence posts on cadence, node 1 seq 217 and node 2 seq 231 live, 11,974 frames, 0 bad, 0 failed.** `check_prebuilt.py`, `check_artifact_drift.py` and `test_dashboard_contract.py` all pass. Nothing pushed, nothing merged into `main`.
414 - 2026-11-25 (**the boot banner told a three-day-old story, the reported restarts turned out to be power-class resets, and the presence feature was watched for ten minutes doing its job without costing the cots a thing -- D-106.**) | The owner reported the hub restarting, cot 1 flapping offline for a second at a time and cot 2's RMS frozen on one value, and offered to scrap the connectivity status if it was the cause. It was not, and it stays. **The hub's banner said `bd3d6fb-dirty` (Nov 24 06:31) while its behaviour was the current code -- ESP-IDF bakes the version and compile time at configure time and scons had reused a Nov 24 configure ever since; the committed prebuilt wore the same stale label.** The resets were power-class (`rst:0x1`, no panic text anywhere; the boot counter's four bumps during diagnosis are exactly the two capture-opens and the flash -- the CH343 resets the board when a monitor opens the port), and the morning's flap/freeze happened during the owner's own unplug/replug tests and did not survive clean power: **600 s of serial -- zero restarts, zero `is offline`, node 1 seq 42 -> 63, node 2 seq 40 -> 45 with live RMS, 15 presence posts at exactly 30 s.** `firmware/tools/make_prebuilt.py` now touches the project's CMakeLists before the hub build (forcing the reconfigure that regenerates the stamp) and refuses an image whose banner does not carry the current `git describe`; `firmware/prebuilt/` was repackaged and the hub on COM32 reflashed. The rule: **a unit's boot banner is the identity check -- when it disagrees with `git log`, suspect the build cache, not the flash.** All checks pass. Nothing pushed, nothing merged into `main`.
415
416 - 2026-11-25 (**the presence now yields to a cot's burst, and the owner's constraint -- nothing may cost the cots anything -- set its rate -- D-105.**) | The owner, having asked for the connection to be checked "as often as possible", added the constraint that actually decides it: *"if this is going to affect the cots or any system performance, you can scrap it or increase the interval or whatever, i don't want anything affecting the system."* **The physics is the link stage's own finding, written down when a cot's "offline" clock had to be taught to run on the time the radio was open: while a burst is on the air the Wi-Fi receiver hears nothing at all -- not that burst's own fragments, and nothing else.** A presence report is a TLS handshake, which takes the radio for a moment; at 15 s it would land inside a burst roughly one time in three, each time costing fragments and a round of resends. So the 15-second figure was withdrawn and three things were done. **(1) The hub reports every 30 s**, not 15 -- 2,880 handshakes a day, twice the original rate, half yesterday's. **(2) The presence yields.** The link stage hands the network stage one function -- `crynet_net_set_radio_guard()` -- which answers whether a burst is in flight (`st->requested`), and the report waits for it to finish rather than take the air from under it. The wait is **capped at 20 s**, because a hub that has been switched off must still be reported gone: a report may be twenty seconds late, it may not be missing. **(3) The dashboard's window went to 90 s** (three reports missed, the same rule of three), in `server.py` and the Worker alike; the Worker was redeployed (`c4f78eb4`). **The cap is the guard's safety, and a host model of the scheduler proves it:** radio never busy -> a report every 30 s; one long burst -> the report waits and posts at the cap (50 s); near-continuous bursts -> 34 s; and **a guard stuck saying "busy" for ever still posts every 50 s** -- so the new code cannot make the dashboard think the hub has gone away. The record upload was left alone: it has always contended with bursts and the whole chain was verified with it running. The page keeps its own 10-second clock -- it asks the *service*, so it costs the cots nothing. **The numbers:** a hub switched off now reads as gone in **about 1.7-2.2 minutes**, against 2-4.5 minutes before any of this. The knob is one line (`PRESENCE_EVERY_S`) if the day's end-to-end tests say otherwise; what to watch is the link stage's own counters -- `resends_sent`, `gap_ewma_ms`, `burst_failures` -- with the presence posting and with `crynet_net_set_radio_guard(0)`. The hub on COM32 was flashed with the new image. Builds clean, `firmware/prebuilt/` rebuilt from it, and `check_prebuilt.py`, `check_artifact_drift.py` and `test_dashboard_contract.py` all pass. **Nothing pushed, nothing merged into `main`.**
417
418 - 2026-11-25 (**the hub's connection is now checked in about a minute -- the hub reports every 15 s, the service calls a hub gone after 45 s, the page asks every 10 s -- D-104.**) | The owner switched the hub off and the dashboard went on saying *online*, and asked for the connection to be checked "as often as possible". **When the dashboard decides a hub has gone is the sum of three delays, and no one of them can be shortened alone:** how often the hub speaks (was 60 s), how long the service waits before calling a hub gone (was 180 s), and how often the page asks (was 30 s) -- 46 to 270 seconds from the plug being pulled, and the owner was standing there watching it say *online*. All three were shortened, keeping the rule the presence was built on -- three chances missed before calling a hub gone -- three times faster. **The hub now reports every 15 s** (`PRESENCE_EVERY_S` in `crynet_net.c`): 5,760 requests a day against the free tier's 100,000, and the real cost is a TLS handshake each time -- a few hundred milliseconds of radio a day, on the same channel the cots use, which the budget affords. **The service now waits 45 s** (`PRESENCE_STALE_S`, changed in both `server.py` and the Worker). **The page now asks for the presence every 10 s on its own clock**, without dragging the nine reads of the record along -- cries do not arrive every second, and *is the watcher alive* is the question the owner actually opens the dashboard to ask -- and it asks again the moment the tab comes back, because a browser may have throttled the timers while the tab was in the background; and it refuses an answer older than the one it already has, so a fast read that returns late cannot say the hub is back. **Two real faults fell out of the measuring.** (1) **The presence strip was inside the overview body, and the overview body is hidden when there is nothing to show** -- so the one element whose whole job is *is the watcher alive* vanished precisely when the record was empty, which is exactly when the question is asked. It is now outside the body, at the top of the overview, and drawn from `render()` unconditionally, whatever the record holds. (2) **A request that never came back froze the page on its last answer for ever.** `refresh()` refuses to run while `state.busy` is set, and nothing cleared that flag if a fetch hung -- one half-open connection meant the page never polled again, and a hub that had gone offline would read as online indefinitely. Every read now has a 15-second deadline (`AbortController`) and the busy guard a 60-second one. **Measured with the new numbers:** the service's deadline passed at 1795575496 and the page said *offline* at 1795575498, on a 10-second poll -- the whole chain lands in **roughly a minute**, where the old chain took up to four and a half. **And it is live:** the Worker was redeployed (`ef3ec161`) and the hub on COM32 was flashed with the new image, whose own posts march past the live endpoint every fifteen seconds (`at` 1795575661 -> 1795575676 -> 1795575692) and read `online:false` at age 77 against the 45-second deadline. Builds clean (RAM 14.3 %, flash 77.2 %), `firmware/prebuilt/` rebuilt from the new image, and `check_prebuilt.py`, `check_artifact_drift.py` and `test_dashboard_contract.py` all pass. **Nothing pushed, nothing merged into `main`.**
419
420 - 2026-11-25 (**the wireless mark means one thing now, it is big enough to read, and the presence chain is closed end to end from the board -- D-103.**) | The owner gave the upload token and asked for two things: **the Wi-Fi mark should show only when connected**, and *"as it is now, i can't see anything like it on the hub display"*. Both are done, and the second had three separate causes. **The mark is inverted.** D-100 drew it in both states -- green on the network, dim grey off it -- on the reasoning that a

mark which appears and disappears cannot say whether the network is down or the panel is asleep. The owner overrode that and the override is the better design on a 5x7-font panel: a mark that is always there has to be **read** to be understood, and *grey means not connected* is a second thing to learn beside the two pills. Drawn only when it means something, the rule is one line long -- *a mark here means the hub is on the network* -- and the flag is the same one the uploader uses to decide whether a record can leave the hub, so the glass cannot disagree with the network. Where the hub is off the network the dashboard says so instead, and there the question has room to be asked and answered. The `C_DIM` off-state no longer exists in the firmware. **The mark was too small.** Radii 6 and 10 over a 35-145 degree span became radii 8 and 13 over 40-140 degrees, the router dot and both arcs two pixels thick -- 21 px wide, 14 px tall, in the free space between `S3 HUB` (ends at x~126) and the `NODE` pill (x=196). The geometry was measured against the panel's own constraints, not chosen by eye; the host mock was proved **pixel-identical** to the firmware by emulating C's float32 `cosf`/`sinf` in Python (no host C compiler exists here, so the equivalence was proved rather than assumed). **And it was not on the glass for a third reason, which was not the firmware's:** the board was running the pre-branch image, in which the glyph did not exist at all. **The token was proven against the live Worker.** `POST /api/v1/presence` with the token answers 200 and without it 401; `GET /api/v1/presence` returns `hub-f9172f80` with `"online":true`. The link D-101 could not close is now closed **from the board**: the flashed hub's own log shows `crynet: presence posted: 0 cot(s)` and `crynet: sent 1 records, up to #78 ({"accepted":1,"received":1})`, and the live Worker answers with the hub present. (`nodes: []` is correct, not a bug -- the cot nodes are off, so the hub has nobody to report; the hub itself is genuinely present.) **The token is off Git.** The repository is public, so the token lives in exactly three places: Cloudflare's secret store (`UPLOAD_TOKEN`), the hub's own NVS (`net token`), and a local, **git-ignored** `dashboard/.secrets`. `.gitignore` now lists `dashboard/.secrets` and `dashboard/worker/.dev.vars`, and `dashboard/worker/README.md` says where the file is, how to use it and how to change the secret. D-97's rule -- *the two secrets are separate and neither is in this repository* -- holds. **One thing to know about Cloudflare and Python:** the edge **rejects `Python-urllib` with error 1010** before the request reaches the Worker, so a Python script posting to the hosted copy needs its own `User-Agent` (`CryNet-Tools/1.0`); the hub is unaffected -- it sends `ESP32 HTTP Client/1.0`, which the edge passes. **The hub on COM32 was flashed with the new image** (via `firmware/tools/flash_unit.py --role hub --port COM32`; the images were hash-verified first; NVS was never written, so the board's id and its token survived). Its boot log: joined *Amazyn* on channel 6, `sta ip: 172.20.10.2`, `Certificate validated`, `crynet: presence posted: 0 cot(s)`, `crynet: sent 1 records, up to #78`. The live D1 was cleaned of every test row afterwards -- only the real hub remains. The firmware builds clean (RAM 14.3 %, flash 77.2 %), `firmware/prebuilt/` was rebuilt from it, and `check_prebuilt.py`, `check_artifact_drift.py` and `test_dashboard_contract.py` all pass. The mock PNGs were regenerated (`research/out/ui_dashboard*.png`, plus a 4x header crop with and without the mark). **Nothing has been pushed and nothing has been merged into `main`.**

421

422 - 2026-11-25 (**the dashboard says who is here — the panel has a wireless mark, the hub reports in once a minute, and the page reads like an instrument — D-100 to D-102.**) | The owner asked for four things and they are all done. **(1) The panel's wireless mark.** `hub_display.c` gained `draw_wifi_glyph()`: a three-pixel dot for the router and two two-pixel arcs over it, drawn between `S3 HUB` and the `NODE` pill. The arcs are walked **by angle**, not by column — a column-by-column walk of a shallow arc leaves gaps at the sides, and the sides are where the eye goes first. The mark was checked as ASCII on the host before it was committed, and it is drawn **whether or not the hub is on the network**: green (`C_GREEN`) with an address, dim grey (`C_DIM`) without, so the panel can say "the records are in my flash, not on the dashboard" without the caregiver asking a phone. The flag is `crynet_net_status().wifi_up`, the same one the uploader uses, so the glass cannot disagree with the network. **(2) Presence.** The hub now posts `{device, hub, nodes:[{cot, online, deaf, rssi}]}` to `POST /api/v1/presence` **every minute and again the instant a cot's state changes**; `link_stage.c` hands the picture over every two seconds and `crynet_net` decides whether anything the caregiver would notice has changed. The service (`server.py` and the Worker) keeps one row per cot, replaced each time, and answers `GET /api/v1/presence`; a hub is *there* if it has spoken in the last three minutes — three missed chances, so one dropped request does not make the caregiver think the hub has died. **The owner still configures one address**: the presence endpoint is the events URL with its last path segment replaced, so `net set https://…/api/v1/events` is all it takes. The page shows it as a strip of pills under the numbers — "Here · ● Nursery hub online · ● Cot 1 online −47 dBm" — and the header line carries the summary. The demo data seeds a presence for the demo hub, and the demo hub is held online because it is a made-up hub with nothing posting it: a demo that showed the panel working when it is not would be the one thing a demo must not do. **(3) The copy.** "the cry record" is gone from the title and the header; every subtitle, note and empty state was cut back to the fact it carries, and the trailing reassurance went with it. Nothing the reader needs was removed — the untimed-record rule, the night boundary and the deaf-cot warning all stay. **(4) The charts.** The daily-cry sparkline was a polyline, and a corner between two counts says something that is not there: that the count jumped and stayed put. It is now a cubic spline whose tangents are chosen the **Fritsch–Carlson** way, so the curve turns at a point and never past it — a plain Catmull-Rom spline dips *below* the baseline between two quiet days, and a chart of counts that draws a negative count is lying. The bars are now paths with a rounded top, a square foot (the foot stands on the axis, so it may not lift off it) and a one-pixel edge in the axis colour, so a bar on a pale grid still has a boundary the eye can find. **A real bug fell out of the cleanup:** `[hidden]` did nothing on any element a class gave a `display` to — the browser's own `[hidden] { display: none }` is outranked by an author rule like `.control { display: flex }` — so the **Hub** dropdown sat on the page with a single hub, and the code had always meant to hide it. One line, `[hidden] { display: none !important; }`, makes `hidden` mean hidden everywhere. **Built and checked, not flashed:** the hub firmware compiles clean (flash 1 619 723 B = 77.2 %, RAM 46 784 B = 14.3 %), `firmware/prebuilt/` was rebuilt from that source, and `check_prebuilt.py` (11 files match) and `check_artifact_drift.py` both pass. The dashboard was driven in a real browser against the seeded demo — both themes, the presence strip, the smooth curve and the outlined bars all checked by eye, and the console is clean. The hosted Worker was redeployed with the new endpoint, the new table and the new page. **Nothing has been flashed to a board** — the owner flashes the units himself. Branch `feature/hub-wifi-and-dashboard-presence`. | `firmware/hub_infer/components/{hub_display, crynet_net}`, `firmware/hub_infer/src/link_stage.c`, `dashboard/{server.py, worker/index.js, worker/schema.sql, web/{index.html, app.js, style.css}}`, `dashboard/API.md` |

423

424 - 2026-11-25 (**the dashboard wears the system's name and the hub's own glass — "Amazing's CryNet", the teal from `hub_display.c`, and a dark/light toggle — D-99.**) | Three cosmetic changes to the caregiver dashboard, made together and deployed in one version. **(1) The name.** The page title, the header brand and the Worker's basic-auth realm all read **Amazing's CryNet** — the exact wording of the hub panel's title card (`fb_text_centre(22, "AMAZING'S CRYNET", …)`), so the web record and the glass say the same thing. **(2) The colour.** The reference blue is gone. The dark theme's surfaces are the hub panel's own values lifted verbatim from `hub_display.c` — C_BG `#020404`, C_SURFACE `#070f10`, C_SURFACE_HI `#0c1a1c`, C_TEXT `#e4f0ec`, C_DIM `#7a9692` — and the hub accent `#2dd4bf` (C_ACCENT) is the chrome accent in dark mode and a deeper teal `#0d9488` on the light surface. The validated categorical palette keeps its eight hues in fixed cot order; only slot 1 and the sequential ramp move to the hub's teal. **Both modes were re-run through the palette validator against their own surfaces:** light (`#fcfcfb`) PASS — worst adjacent CVD 9.1, normal-vision 19.6, with the known contrast WARN on aqua/yellow/magenta covered by the already-shipped relief channel (every chart has a table twin, every cot mark a written "Cot n" label); dark (`#070f10`) all PASS. The hub accent itself **fails the dark data band** (OKLCH L 0.785 against a 0.48–0.67 band), so the dark chart teal is **`#23a893`**, the nearest passing step of the same hue, and `#2dd4bf` stays where the band does not apply: brand mark, tab underline, focus ring. **(3) The toggle.** A moon/sun pill in the header; an inline script stamps the theme on `<html data-theme>` **before first paint** (saved choice in `localStorage['crynet-theme']`, else the OS setting, which the page keeps following until a choice is made), so there is no flash of the wrong theme; charts read CSS custom properties, so one `drawCharts()` repaints them, and the heatmap's cell-number ink flips with the ramp — white ink on the deep end of the light ramp, near-black on the bright end of the dark ramp, which runs the other way. **Verified in the browser with seeded data:** bars compute `rgb(13,148,136)` in light and `rgb(35,168,147)` in dark, the dark body background is `rgb(2,4,4)` = C_BG exactly, the choice survives a reload. **Deployed to the live worker** (`amazings-crynet`, version `39b339f5`, uploaded 3 files); the caregiver login is unchanged and now greets the browser with the system's name. Nothing here touches the firmware, the hub, or the monitor's

offline behaviour.
425
426 - 2026-11-24 (**the hub is online for real — the DHCP fault is fixed and diagnosed to its root, the records are on the live dashboard, and `logtest` has finally run — D-98.**) | Three faults found in one session, all after D-97 put the URL live, and the first one was the fault D-97 left open. **(1) The hub joined the Wi-Fi and never got an address.** `esp_netif_lwip: dhcp client start failed` with the association perfectly healthy (`rssi: -32`) — so the lwIP debug options went into the local (gitignored) sdkconfig and the board was asked again. lwIP answered with the real reason: **`Cannot use this netif with DHCP: MTU is too small`**. The mechanism, read out of the ESP-IDF 6.1 sources: `esp_netif_new` merely `calloc`s the netif (MTU 0); lwIP gets the interface only inside `esp_netif_start_api`, which the driver triggers from the **STA_START event**; and `dhcp_start()` refuses any netif whose MTU is under 576 (`DHCP_MAX_MSG_LEN_MIN_REQUIRED`), returning `ERR_MEM` — the same code an allocation failure returns, which is why static analysis of the three documented paths could never say which one it was. **`node_link` had been starting the radio before the station interface existed, so the start event fired with no netif to receive it and the interface never joined the stack.** The fix is an ordering rule, not a patch: `node_link_start()` calls `esp_netif_create_default_wifi_sta()` **before** `esp_wifi_start()`, and `crynet_net_start()` looks the interface up by its if-key (`WIFI_STA_DEF`) instead of creating a second one — because by the time the network component runs, the radio is already up, and a netif created then would repeat the same fault (it cannot even be started: `esp_netif_start` is not public API in IDF 6.1). **Verified on the board:** DHCP offer and ACK from the router, `sta ip: 172.20.10.2`, `asking the network what time it is` — and then the thing the whole chain was built for: **68 records uploaded to the live dashboard in 3 batches, 0 failures in a row**, `{"accepted":68,"received":68}` end to end, the device appearing as `hub-f9172f80` ("Nursery hub", fw `2.0-dashboard`). Two more uploads followed on the rebuilt clean image. **(2) `logtest` had never actually run — on any board, ever.** The console matched `log` as a three-letter prefix, so `logtest` fell into the log branch, printed the log state and returned; the PASS that had never been observed was never observed because the self-test was unreachable. The prefix is now the whole word or the word plus a space, and the board finally says **`logtest: PASS -- a cry that finishes out of order is still sent`** — record 2 written before record 1, both durably numbered, both read back. **(3) A refused upload arrived as `ESP_ERR_NOT_SUPPORTED`** — the HTTP client tries to meet the server's authentication challenge, and with nothing to answer it returns that instead of the status; the uploader now reports `HTTP 401 (ESP_ERR_NOT_SUPPORTED)` so the status itself is visible. **`hub_console.py` was rewritten** around the two facts the board taught last session: opening the port can reset the hub, and the ready banner only prints at boot — the tool now waits up to `--wait` for the banner, falls back to signs of life when the board was already running, confirms **every command by its echo** (`console: << net`), retries a command that is not echoed (safe on a still-booting board), and listens with a bounded quiet-wait so a chatty board cannot hang it. **The clean shipping image was rebuilt without the debug options** (flash 1618704 B = 77.2 %, RAM 46720 B = 14.3 %), flashed to COM32, and re-verified end to end: IP, clock, upload, `logtest: PASS`. **`firmware/prebuilt/` was rebuilt** from the same source (`SHA256SUMS`, `BUILD_INFO.json`) and all three checks pass — `check_prebuilt.py` (11 files match), `check_artifact_drift.py` (artifacts, constants and deploy config agree), `flash_unit.py --check` (hub images hash OK). **Nothing here changes the monitor's behaviour offline**, which is the standing constraint: the wrist still buzzes, the flash still holds the record, and the dashboard's absence still changes nothing.
427
428 - 2026-11-24 (**the hosted dashboard is LIVE and verified from the public internet — D-97.**) | The owner's Cloudflare account now carries the `amazingadelowo` subdomain, so the Worker that D-95 wrote could be deployed for real, and it was: **`https://amazings-crynet.amazingadelowo.workers.dev`** is the caregiver's dashboard, live on the public internet, with the D1 database `crynet` behind it. **The live copy was verified from outside**, not only locally: `GET /api/v1/health` answers **200** and is deliberately public (it says only `ok`, the time and how many devices exist), every *read* answers **401** without credentials — `summary` with no token is 401, the *page* itself with no login is 401, and an upload with the wrong token is 401 — while `summary` with the hub's `X-CryNet-Token` answers **200** and the page with the caregiver's Basic credentials answers **200**. The database is empty (0 devices, 0 events), so nothing fake has been put in the caregiver's record. **The hub's TLS trust was checked against the live edge before the hub was pointed at it:** Cloudflare's `workers.dev` edge presents a Let's Encrypt chain — `CN=amazingadelowo.workers.dev` → `CN=YE1` → `CN=Root YE` → **`ISRG Root X2`** — and **ISRG Root X2 is already in the hub's 145-root bundle**, so no certificate has to be added and no firmware changed; the handshake negotiates TLS 1.3 with `TLS_AES_256_GCM_SHA384`. The dashboard's own contract test was re-run and still prints **"OK the hub's upload body and the dashboard's API agree"**. **The one thing still standing between the hub and this URL is the address:** the hub joins the Wi-Fi and never gets one (the DHCP fault below), so nothing has been uploaded yet and the dashboard is, for the moment, an empty page waiting for its first record.
429
430 - 2026-11-24 (**the hub's cry log had never actually worked on real hardware — three separate SPIFFS faults, all found on the board and all fixed — D-96.**) | The D-95 work put the hub's flash behind the caregiver's dashboard, and this entry is the honest record of what happened when the flashed image was finally run against a real SPIFFS partition instead of the host's filesystem. **The log was broken in three ways, and each one had been wrong since the log was first written (commit `44cbfb9`) — no host test could have caught any of them, because the host has a real filesystem and SPIFFS does not.** **(1) The hub could not create its own log.** ESP-IDF's VFS drops a bare `O_CREAT`: `spiffs_mode_conv` in `esp_spiffs.c` only honours it when it arrives with `O_EXCL` or `O_TRUNC`, so `open(path, O_RDWR | O_CREAT)` on a file that is not there fails with *No such file or directory*. On a **freshly formatted partition — which is every brand-new board flashed from scratch — the hub ran with no record at all** and said nothing about it. Fixed by creating with `O_CREAT | O_EXCL` and falling back to a plain open when the file is already there. **(2) The hub could write a record and never read it back.** SPIFFS registers **no `pread` and no `pwrite`** operation, and the VFS answers both with *function not implemented* (`CHECK_AND_CALL` returns `ENOSYS`, with no emulation behind it). Four sites were affected: the counters write (`the counters did not write: Function not implemented`), the header read, the per-record read, the tail scan and the compaction copy. The hub could therefore **append records it was incapable of reading** — a record the uploader needs by number, and the number itself was never durable. Fixed by seeking and reading (and seeking and writing) at an offset, which is what SPIFFS does support. **(3) The counters file and the records file could fall out of step**, when a power cut landed between the two writes, and the next number handed out would be handed out **twice** — the same record number on two different cries, which is exactly what the wrist shows the caregiver. The numbers in the file are the truth, so at boot the highest number in the tail of the file now sets the next number to hand out, and the hub says so out loud when it disagrees with the counters. **Verified on the board (COM32):** the flashed hub boots clean, mounts the storage (**1506 of 5775761 bytes used**), and prints `boot 16: 16 records kept, 16 still to send, next number 17` — and the same numbers come back after a reboot, which is the proof that the records are durable *and* readable. The `log` console command agrees: `the log holds 17 record(s), 17 still to send, boot 16, device hub-f9172f80`. **New tool `firmware/tools/hub_console.py`:** the old `node_console.py` typed its command **half a second after the port opened**, which is *during the board's boot* — because opening the hub's port resets it (the USB-serial bridge holds its enable line) — so the command was typed into a board that was not listening yet, and the silence looked exactly like a command that had been ignored. The new tool **waits for the console's own ready banner** before it types anything, and it documents every hub command in its docstring. **One fault is still open, and it is not small:** the hub **joins the Wi-Fi and never gets an address** — `wifi:connected with Amazyn, aid = 2, channel 6, BW20, rssi: -32` is immediately followed by `esp_netif_lwip: dhcp client start failed`, so the hub has no IP, uploads nothing, and the SNTP clock never starts. lwIP's `dhcp_start()` has exactly three ways to fail, all of them `ERR_MEM`: the netif's MTU being under 576, the DHCP client's own allocation failing, and the shared DHCP UDP PCB failing to be created. The MTU is 1500 (the WLAN netif's `low_level_init` sets it), so the failure is an allocation — **and lwIP's own debug output is the only way to say which allocation**, so that is the next build. **Nothing here changes the monitor's behaviour offline**, which is the standing constraint: the wrist still buzzes, the flash still holds the record, and the dashboard's absence still changes nothing.
431
432 - 2026-11-24 (**the caregiver's record can live on the internet now, and it costs nothing — D-95.**) | The dashboard was a Python file on a laptop on the same network as the hub, which is exactly right for a bench and exactly wrong for a caregiver who wants

to look at last night's record from work. The constraint is unchanged and still absolute: **the monitor never depends on the dashboard being reachable** — the hub records every cry to its own flash first and uploads whenever it can, so hosting changes nothing about how the monitor behaves offline. The hosting was chosen from what the system *is*, not from a list of hosts: **Cloudflare Workers + D1 on the free tier** (no card, always on, no idle-reclaim, and the hub's ESP-IDF CA bundle already trusts the certificate Cloudflare's edge presents — GTS Root R4 is in the 145 roots the hub was built with, verified against the live chain, so `net set https://…` needs no firmware change and no new code on the board). D1 speaks SQLite, so the schema is the same one `server.py` has always written. **The free tier is not the interesting part; the two limits are, and the system was measured against them.** Workers allows 100,000 requests/day and this system uses about a hundred (uploads are batched 32 records at a time). D1 allows 5M rows read/day, hard-enforced since September 2026 — and the old summary read *the whole table* on every page load (`limit=100000`), so a busy household would have walked into the wall and the dashboard would simply have stopped answering. The window queries were therefore split into two index-friendly halves so the budget is spent on the window and not on the table. **Two real defects were found on the way out, both of which only bite once the service is public.** (1) **The read API was completely open.** `_authorised()` existed but was called only on `POST`; every GET — summary, analytics, events, devices — was world-readable, so publishing the URL would have published the baby's whole cry record to anyone who found it. The hosted copy now puts a login in front of every read and keeps `X-CryNet-Token` for the hub. (2) **The days, hours and nights were counted in the *server's* timezone.** On the caregiver's own machine the server and the browser were the same machine, so they agreed; hosted in UTC with the phone on W. Central Africa, "the night of the 14th" and the hour-of-day heatmap would have been computed an hour away from every timestamp the page displays — a wrong number, silently, in the one figure a caregiver acts on. The page now sends its zone with every request and both copies bucket in it. **The hosted copy is a Worker (`dashboard/worker/`), a port of `server.py` field for field, and `server.py` stays exactly as it was — it is still the whole offline dashboard, still one file, still no installs.** The port is deliberately kept honest by the existing contract test, and the numbers were checked by hand against the Python: untimed records counted but never placed on a time axis, per-cot sums, night naming, and the `start`/`end` names the hub actually sends. The hub was reflashed on COM32 in the same session and booted clean — display up, BLE advertising, console listening, NVS untouched.

- 2026-11-24 (**a cry could be lost for ever, and it was — the log's file is not in number order, and the uploader searched it as if it were — D-94.**) | Reviewing the new uploader found a bug that would have broken the whole point of the feature: **a cry the hub had recorded could never reach the dashboard, and the hub would say it had sent everything.** The cause is a fact about the design that was written down in D-89 but not carried through: **a record's number is spent when the episode starts and the record is written when the episode ends.** Two cots crying at once can therefore end in the *opposite* order to the one they started in — cot 1's long cry holds number 41 and is still running, cot 2's short one holds number 42 and finishes first, so **the file holds 42 before 41**. The uploader found the records still to send with a **binary search**, which is only valid on a sorted file. The file was not sorted. Asked for the first record above a number, the search could land *past* a record whose number was lower — and that record was then never sent, never uploaded, never seen. The dashboard simply never showed that cry, and nothing said so. **The fix has two halves, and both are necessary.** (1) The batch is now the **lowest** unsent numbers, found by a linear walk of the file — taking the first records *in file order* would have been just as wrong in the other direction: the same high-numbered record would be re-sent for ever while a lower one sat behind it. That livelock did not show up in my first attempt; the randomised soak caught it, and it is now the reason the batch is chosen by number. (2) The number the hub claims as sent is **clamped below two things**: the lowest number that did not fit in the batch, and the lowest number an episode still holds while it is being measured. Without that clamp the hub would claim "sent up to #42" while #41 was still running, and #41 would be skipped for ever. **The proof is in two places, because this machine has no C compiler and no board attached.** `firmware/tools/test_log_order.py` models the algorithm on the host: six tests, including a **ten-thousand-case soak** over random orderings, batch sizes and upload moments, plus a companion test that runs the soak against the **old** code and *requires it to lose records* — so the guard cannot quietly stop guarding. The hub's new **`logtest`** console command runs the same scenario against the **real component on the real chip**, on its own scratch file, with the counters file pointed at its own path so the real log is not touched even for a moment; it restores every piece of state afterwards and deletes what it made. **Verified:** the hub builds clean (`pio run` SUCCESS, flash **1 617 171 B** = 77.1 % of the 2 MB app slot, static RAM **46 720 B** = 14.3 %); `make_prebuilt.py --role hub` rebuilt the images and `check_prebuilt.py` passes (hub fingerprint `a39ef5eae2e040ba…` over **61** files), `check_artifact_drift.py` passes, `flash_unit.py --check` passes on every hash, `test_log_order.py` prints **"OK every record the log writes is eventually sent"**, and `test_dashboard_contract.py` still prints **"OK the hub's upload body and the dashboard's API agree"**. | `firmware/hub_infer/components/crynet_log/{crynet_log.c,include/crynet_log.h}`, `firmware/hub_infer/components/crynet_net/crynet_net.c`, `firmware/hub_infer/src/console.c`, `firmware/tools/test_log_order.py` (new), `DECISIONS.md` (D-94) | **Committed on `feature/cry-dashboard-and-log` in five commits (`44cbfb9`, `c5da57d`, `8c43fa2`, `7cd0816`, `614a629`); nothing pushed, nothing merged into `main`, and nothing flashed.** The proof has run on the host; the on-board half (`logtest`) is compiled but has **not** been run, because no board was attached — that is the owner's next minute with the hub on USB: type `logtest`, and it answers `PASS`. The dashboard's own data is untouched by this: the change is in how the hub picks what to send and what it claims to have sent.

- 2026-11-24 (**the flash write on the audio path was checked properly, trimmed, and measured on the board — D-93. The dashboard's web client was driven in a real browser.**) | The one thing left open from the dashboard work was a self-raised question: *does writing to SPIFFS in the middle of the audio path cost the cot link anything?* **The answer was established from the code rather than assumed.** Fragment reassembly happens in the **ESP-NOW receive callback** (`handle_fragment` in `node_link.c`), not in the main loop, and `analyse()` runs only **after** `burst.complete` — so by the time the flash write starts, every fragment is already in the buffer. **A flash write cannot cost a fragment.** What it does is delay the *next* pull, by one page erase and program; the node holds a frozen window for three seconds and drops it after twenty, and the deaf logic *discounts* the hub's own deafness, so the extra time makes a cot look **more** alive, never less. **Three changes came out of it.** (1) `save_meta_locked()` no longer truncates and rebuilds the counters file — it rewrites the same thirty-two bytes in place, which is fewer pages of flash on the path between a cot's microphone and the caregiver's wrist. (2) `crynet_log_commit()` no longer rewrites the counters on every episode: the clock is set once by SNTP and then runs on its own, so the counters are rewritten only when the clock has actually moved by more than a minute. **An episode now costs one page write instead of three.** (3) The hub **prints the measured cost** of both writes (`record 42: the number was made durable in 37 ms`, `record 42: written in 12 ms`), so the owner sees a number from the board in front of him instead of taking an estimate on trust — and those two lines are also the cheapest possible check that the record is being kept. **The durable reservation itself was kept, deliberately:** if the record number were handed out from RAM, a power cut in the middle of a cry would let the *next* cry be given the same number, and the wrist would name a record the dashboard holds for a different cry. That is the one guarantee worth a page of flash; everything else was moved off the path or made conditional. **The dashboard's web client was then driven in a real browser against a live server** (the agent's own headless Chrome, seeded with a week of invented cries and again with an empty database): the Overview and the Patterns tab both render, the duration histogram, the hour-by-cot heatmap and the per-night table all draw, the range buttons and the cot filter work, the 30-second poll matches the contract in `API.md`, **the console is clean**, and **the empty state says "Nothing has been uploaded yet" instead of a broken chart**. `app.js` passes `node --check`. **Verified:** the hub builds clean (`pio run` SUCCESS, flash **1 614 759 B** = 77 % of the 2 MB app slot, static RAM **45 424 B** = 13.9 %); `make_prebuilt.py --role hub` rebuilt the images, `check_prebuilt.py` passes (hub fingerprint `33621c97c42c5da3…` over **61** files), `check_artifact_drift.py` passes, `flash_unit.py --check` passes on every hash, and `test_dashboard_contract.py` prints **"OK the hub's upload body and the dashboard's API agree"** — it posts the hub's exact byte stream, and it found a real bug in the process (the dashboard was dropping the untimed records from its totals instead of counting them, which contradicted `API.md`; fixed and covered by the test). | `firmware/hub_infer/components/crynet_log/{crynet_log.c,CMakeLists.txt}`, `DECISIONS.md` (D-93), `SYSTEM_UPDATE.md` | **Nothing is committed and nothing is flashed.** The flash-cost figures are printed by the firmware but **not yet

measured on the bench** — that is the next thing the owner does with the hub on USB. The dashboard's own record (the SQLite file) is untouched by all of this; the change is in the hub and in the page's verification, not in the data.

- 2026-11-24 (**the hub now keeps a record of every cry, and there is a caregiver's dashboard — D-87 to D-92. Built, not yet flashed.**) | The owner asked why the wearable shows an "event #number" when the hub stores nothing (that number was a RAM counter, `s_event_id`, reset on every boot), and then asked for a dashboard: *"i want cry data to be always recorded locally and when the hub has access to the internet, it uploads the cry data that has been stored locally to the online dashboard … the system should be able to run perfectly offline as it is now"*. That last clause is the constraint the whole design answers to. **What was found:** the partition table had reserved six megabytes for storage since the beginning (`storage, data, spiffs, 0x210000, 0x600000`) and **nothing had ever mounted it**. The space was there; no code used it. **What was built:** three new hub components — `crynet_log` (32-byte append-only records on SPIFFS, one per cry *episode*, 8192-record cap, trims to 2000 after upload, power-cut safe), `crynet_clock` (a base-pair wall clock with no RTC; unknown time is recorded as `start: 0` and never placed on a time axis), and `crynet_net` (SNTP + batched HTTP upload, idempotent by device and record number). `link_stage.c` opens an episode record when a cry confirms, gathers peak/mean loudness and peak confidence window by window, and commits it when the hold expires; deaf/heard/test/boot are recorded too. The wrist's `event_id` now comes from the log's record number, so "event #1042" on the wrist is record #1042 on the dashboard. **The radio hazard, and the guard:** the cots talk ESP-NOW on channel 6 and the hub has one radio, so joining a router on any other channel would kill the cot link — the wrist silent while the dashboard looks healthy. `crynet_net` checks the channel the instant it associates, disconnects if it is not 6, restores channel 6, and retries in five minutes. **The router's 2.4 GHz channel must be set to 6**; that is written in three places. **The dashboard:** `dashboard/server.py`, Python stdlib + SQLite WAL, no dependencies and no build step; `dashboard/web/{index.html,style.css,app.js}`; the contract in `dashboard/API.md`; the owner's guide in `dashboard/README.md`. **Verified:** the hub builds clean (`pio run` SUCCESS, flash 1 613 939 B = 77 % of the 2 MB app slot, static RAM 45 424 B = 13.9 %); the only two warnings in `link_stage.c` (`saturated`, `stall_us`) are pre-existing in `HEAD`. `firmware/prebuilt/hub/` rebuilt with `make_prebuilt.py --role hub`, SHA256SUMS and BUILD_INFO.json refreshed. The server was smoke-tested on an empty database (health and summary both answer correctly with zero events) and its startup banner prints the exact `net set` line for the hub. **Not yet done:** nothing is committed, nothing is flashed, and the offline proof (router off → `alert 1` → wrist still buzzes) has not been run on the bench. Branch `feature/cry-dashboard-and-log`.

- 2026-11-23 (**the dossier was checked against the bench and found stale in five ways — D-86 is ported, the boards are corrected, the node's size figure was a day-one number, and two "never" claims were wrong**) | The owner opened the new PDF, saw **ESP32-C3 SuperMini** listed for the cot nodes on page 8, and asked whether the dossier was actually current before he studied it. It was not. **Five real gaps, all now closed or recorded:** **(1) The board naming was wrong, and the repo's own tables were the source.** The swap to **Pro Mini** nodes was made and **verified live on 2026-11-02** (see the entry below: same firmware unchanged, native USB, 286/286 bursts, two cry plays → two alerts), and both bench cots are Pro Minis — `90:DA:72:85:C5:44` (id 1, COM31) and `90:DA:72:87:CB:98` (id 2, COM37). But `PROJECT_HANDOFF.md`, `TWO_COT_README.md` and `firmware/node/README.md` still said SuperMini in their headline tables, `HARDWARE_COMPONENTS.md` called the swap "planned production", and `SYSTEM_WIRING.md` §8 said the owner "plans a board swap". **The dossier copied those tables.** All five documents are corrected here, and thirteen of the dossier's chapters follow. The firmware never needed changing: `weact_studio_esp32c3` targets the **chip**, and both boards are C3 / 4 MB / native USB. **(2) D-86 was missing.** The alert face that names every crying cot was merged in the sibling repo (`c35ead1`, 2026-11-20) and **bench-verified on this system's two cots** — mask `0x03` for fifteen seconds, the owner watching the glass read the two-cot label, nine announcements across five episodes giving exactly five buzzes. It existed here only on `feature/wearable-multi-cot`, which turned out to be an **earlier and different** attempt with no `COTS` label at all, predating the current tree. **The verified change is now ported here**: the two repos' wearable bases were byte-identical modulo CRLF, the 83-line diff was exactly D-86, and only the three-cot comments needed adapting (`COTS 1 2` is 144 px at size 3 on a 128 px glass, 96 px at size 2). **Compile-verified: 685,589 B (52 % flash) / 18,684 B (5 % RAM) — the identical figures to the build that was flashed and watched.** `make_prebuilt.py --role wearable` rebuilt the offline image and `check_prebuilt.py` passes (11 images match, all three roles up to date). **No board was attached, so this repo's image is not flashed — recorded as O-1f.** D-80 is marked superseded. **(3) The 2026-11-20 clipping session was absent.** It measured the close-range blind spot: cot 1 peaked **32767 of 32767** (29 samples at the rail), cot 2 **32637** and **32736**, all refused at `link_stage.c:471`, which is why a cry played at **cot 2 was announced as cot 1** — the bystander's clean window won. It also established that the runbook's "0.10–0.20 rms" target is **unsafe advice**, since all three refused windows sat inside that band (window crest 6.1–7.3): **peak is the instrument, not rms**, and the fix is distance. Recorded as **O-1g**, including the design limitation that one saturated sample out of 32,000 discards a whole window. **(4) Two absolute claims were false.** "Nothing has ever run from a cell" — the 2026-11-20 log says plainly that **the wearable was on its Li-Po pouch**, not USB, for the third round (which is why COM30 was absent). A unit has run from a cell; **runtime is still unmeasured**, and a sagging 3.3 V rail on the pouch is a live candidate for the buzz that went missing that round. O-1d is rewritten to say exactly that. And **O-1b is closed**: two cots crying at once has been tested. **(5) The build figures, the line counts and the decision count were earlier numbers presented as current.** The dossier said **977,617 B (74 %) / 165,972 B (50 %)**, which is the stage-2b build of **2026-10-30**, carried forward unchanged through every later rebuild. Compiling the tree as it stands on 2026-11-23 gives **983,177 B (75 %) / 166,404 B (50 %)** — and this repository's own D-83 entry of 2026-11-07 already recorded the node at "75 % flash, 50 % RAM", so the correct percentage was in the record all along. The node sketch's header comment also still named the SuperMini; it now names the **Pro Mini** and says plainly that the image is board-agnostic, and `make_prebuilt.py --role node` was re-run so `BUILD_INFO.json`'s source fingerprints match (the comment-only edit compiles to the identical program storage — verified by compiling the pre-edit version too). **The dossier's O-13 is closed in the same pass:** the sketch described the hub's heartbeat as arriving "every 100 ms" in **two** places, while `link_stage.c:1495` sends it every **1000 ms**; both comments now say 1000 ms, and the peer-learning comment's "within a tenth of a second of boot" now reads "within about a second of boot". Comment-only again, same compiled size. The same sweep caught **five stale line counts and a stale decision count**: `link_stage.c` is **1,642** lines (the dossier said 1,622), `console.c` **180** (said 170), `node_crynet.ino` **1,808** (said 1,697), `wearable_crynet.ino` **750** (said 748 — that is the sibling repo's count, before this port's comment), `build_notebook.py` **2,293** (said 2,332); and `DECISIONS.md` now carries **87 rows over 85 distinct numbers, D-1…D-86** (the dossier said 86 rows / 84 distinct / D-1…D-85, which was true before D-86). `crynet_infer.cpp` (247), `ili9341.c` (413) and `model_data.h` (8,604) were checked and **are** right. The hub's figures were checked the other way and **are** current: the shipped `firmware/prebuilt/hub/firmware.bin` is 1,335,248 B of the 2 MB partition = **63.6 %**, and RAM **13.0 %**, exactly as the D-83 entry records. **What was already correct and was re-checked, not assumed:** the deployed model's SHA-256 and AUCs, the signal contract and its eight guards, the node constants, the two-cot link-layer enforcement, the heartbeat/RSSI path, the panel geometry and the offline images. | `firmware/wearable/wearable_crynet/wearable_crynet.ino`, `firmware/node/node_crynet/node_crynet.ino` (header comment only), `firmware/prebuilt/{wearable/app.bin,node/app.bin,SHA256SUMS,BUILD_INFO.json}`, `DECISIONS.md` (D-86, D-80 superseded, O-1b closed, O-1d rewritten, O-1f and O-1g new), `PROJECT_HANDOFF.md`, `TWO_COT_README.md`, `firmware/node/README.md`, `HARDWARE_COMPONENTS.md`, `SYSTEM_WIRING.md`, `docs/dossier/chapters/`, `SYSTEM_UPDATE.md` | **Committed on `feature/wearable-multi-cot-d86` and fast-forwarded into `main`.** `main` was 17 commits behind the bench — it still built the pre-D-83 firmware, with the battery divider and the status LED, and it had no `firmware/prebuilt/` at all, so the offline one-command flash route did not exist on it. The merge therefore brings the repository up to the hardware, not only the dossier up to the bench. **Before this image is trusted on the wrist:** flash it (`flash_unit.py --role wearable --port COM30`) and re-run the two-cot play, because the port is compile-verified only. The dossier was then rebuilt and checked: **545 pages, 13.9 MB, 210 bookmarks (23 top level, 187 sub), 219 internal links (all in range), 219 HTML anchors with 0 duplicate ids and 0 anchors without a target, 103 code listings.** The cover is still exactly one page and page 2 opens the contents. **The lesson worth keeping:** the dossier was faithful to the repository and the repository was behind the bench — three sessions were run on this two-cot hardware but logged in `fp-2`. A document built from prose inherits the prose's staleness; only the source, the artefacts and the dated bench entries are authoritative. |

- 2026-11-22 (**the two-cot project dossier: a 532-page PDF that explains this system from the first commit to the last measurement**) | The owner asked for a complete, defensible breakdown of the project — not a summary — able to be reproduced unaided, and asked for it in the **two-cot** form, because two cots is what the system is and what is on the bench. `docs/dossier/` now holds `build.py`, `chapters/` (21 chapters of markdown) and the two outputs. **The PDF is `docs/dossier/CryNet-TwoNode-Dossier.pdf` — 532 pages, 13.7 MB.** Cover; a paginated contents with real page numbers; 21 narrative chapters; **Appendix A**, the project's own fifteen documents verbatim; **Appendix B**, **88 first-party source files** with line numbers (103 listings, none missing). 210 bookmarks (23 top level, 187 sub) and 218 internal links, every one resolving. Rebuild with `python docs/dossier/build.py` (no `-I`: pypdf and pymupdf live in user site-packages). **What the rewrite changed, and why it is more than a find-and-replace:** the two-cot system is not the three-cot system with a smaller number, and the dossier now says so from the first line. (1) **Two cots are enforced in the link layer, not drawn.** `NODE_LINK_MAX_NODES` and `HUB_DISPLAY_MAX_NODES` are 2 and `node_link.c` drops any frame whose id is outside 1..2 before it can claim a slot, so a third board is not hidden from the panel — it is never heard. Chapter 2 carries the code. (2) **The panel chapter was rewritten around the real layout**: two 64-tall cards on a 70 pitch, one reading per row, the `LEVEL_BAR_FULL 0.25` square-rooted bar with its trigger notch, the four-bar signal glyph and its dBm, the three BOOT faces (dashboard / help / colour check), and the title card that already reads `ESP32-S3 HUB - TWO COT NODES`. (3) **The RSSI and heartbeat path is documented**, including the 16-bytes-not-12 heartbeat framing bug that would have left every node's link bar dark, and the rule that only the hub's own frame types count towards the cot's reading of the hub. (4) **The node chapter gained** the id rule and its NVS survival, the I2S zero watchdog (D-79, committed and *not* flashed), and the fact that GPIO10 is free because the connection LED was withdrawn. (5) **The haptic chapter is now a proof, not a fault**: the standalone bench sketch, three complete alert cycles on 2026-11-06, and the fitted 22 Ω cap on 2026-11-11 — with the honest remainder, that none of it has ever run from a cell, and that the 56 mA stall figure is arithmetic from an assumed 37 Ω coil rather than a measurement of the fitted motor. (6) **The offline flashing route is now a chapter**: `firmware/prebuilt/`, `flash_unit.py`, the region offsets that keep NVS alive, and `BUILD_INFO.json`'s per-file source fingerprints — with the check **run and passing** at the time of writing. (7) **Chapter 18 was rewritten around this repository's own open items** (O-1a…O-1e) and the items that have no O-number, and the documentary defects are listed rather than hidden: D-51 and D-52 used twice, D-70 skipped, D-3 still saying three nodes, O-1 still naming a DRV2605L, the runbook's RMS target, and a stale heartbeat comment. (8) **A real error was found and corrected.** The dossier had inherited the *pre-retrain* model's SHA-256 (`4aef55d8…126bc11`) and its test/cross AUCs (0.9995 / 0.9973) from the project's own prose. The **deployed** artefact is `bd592d54…dce43b9`, 137,384 bytes, with test AUC **0.9993** and cross-corpus AUC **0.9984** — `deploy_config.json` and `check_artifact_drift.py` both carry the live one, and the history shows the retrain lowered the in-domain test AUC by 0.0002 to raise the cross-corpus AUC by 0.0011, which is the trade the music retrain was made for. Chapter 5 has the comparison; the stale hash in D-16 and `SYSTEM_UPDATE.md` is now recorded as an open documentary item. (9) Two smaller corrections: the DSP has **eight** compile-time guards, not five, and the node is 977,617 B / 165,972 B, not the three-cot figures. **Nothing in the firmware, the model, the research or the hardware was touched** — this is a document. | `docs/dossier/` (new), `SYSTEM_UPDATE.md` | The dossier is a new artefact and is **not committed** — it is untracked in `docs/`, on branch `docs/two-cot-dossier`, and the owner decides whether it joins the repository. The three-cot dossier in `fp-2` is untouched and remains valid for that system.

- 2026-11-11 (**the documents now say 22 Ω where the board has 22 Ω, and the 5 V supply chain has three written rules**) | Two things the owner asked about were answered from the bench and neither was in the documents. **(1) The stall cap.** The owner had a **22 Ω** and no 10 Ω, so that is what is fitted in the motor's + lead, and on 2026-11-11 the bench proved the drive works with it — the motor was replaced (the first one's coil was open: the full **3.296 V** appeared across the motor with GPIO7 switching, because no current means no drop across the cap) and the new motor vibrates correctly. **22 Ω is the safer of the two values as well as the fitted one:** the stall current is capped at 3.3 V ÷ (37 Ω + 22 Ω) ≈ **56 mA**, against ≈70 mA for a 10 Ω, so it sits further inside the BC547's 100 mA rating. The cost is the buzz: the cap drops about 1.2 V of the 3.3 V rail, so the motor sees ≈2.1 V instead of ≈2.6 V and runs slightly softer — which is also why the motor's burst current in the 3.3 V budget is ≈55 mA now, not 85 mA. **The documents were not merely tidied — a 10 Ω in the BOM is a part the owner would have bought and soldered, and the 69 mA figure that goes with it was wrong.** Corrected here: the **BOM** (the motor row in `HARDWARE_COMPONENTS.md`), **`SYSTEM_WIRING.md` §5.3** (the cap, the stall-current figure, the motor's burst current), **`D-77`** (revised to point at the new decision), and **page 4 of `SYSTEM_WIRING_DIAGRAMS.pdf`** — `tools/wiring/page_wearable.py` now draws `22 Ω` and the note gives the 56 mA figure, and the PDF was rebuilt with `python tools/wiring/build.py` after `--check` passed. **New `D-85`** records the decision. **(2) The 5 V supply chain.** The owner asked whether the ESP32-C3 SuperMini and the Pro Minis could be powered through their 5 V pins. The answer is yes, with three rules that were not written down anywhere, and they are consequences of the chain D-82 already describes (cell → boost 5.1 V → VO+/VO− → switch → the board's 5 V pin → the board's own linear regulator → 3.3 V): **(a) Set the trimpot to 5.1 V with a meter before any board is attached** — the module ships anywhere in its 4.2–28 V range, and a board connected first sees whatever it happens to be producing. **(b) Never connect the boost and USB at the same time.** The boost's output goes to the board's 5 V pin, and that pin is tied to the USB 5 V rail through the board's own protection diode, so the two supplies are on one net: they fight, and current can be pushed back up the USB cable into the laptop — **a fault on the host's side, not only the board's**. Unplug USB before the battery goes on; to watch a unit on its console while it runs from its cell, use a **USB cable with the 5 V (VBUS) wire cut**, so the data lines still carry the serial log and nothing can be pushed back. **(c) The board's linear regulator is the weak point and it gets hot.** It passes *current*, not power, so the difference between 5.1 V in and 3.3 V out is burned as heat inside the board: **(5.1 − 3.3) V × I**, roughly **0.5–0.7 W** at the C3's radio peaks of a few hundred mA, in a package with no heat sink. That is why the cell supplies ≈1.8× the 3.3 V load current (D-82), and it is the first thing to check when a unit browns out or resets as the radio transmits — **not the radio**. Feeding the **3V3 pin** from a 3.3 V buck-boost removes the dissipation and roughly doubles the runtime, but it needs a **separate** 3.3 V module, because the LX-LCBST cannot be set below its 4.2 V minimum. **What does not change:** the OLED and the INMP441 stay on 3.3 V in every arrangement, because the OLED's I²C pull-ups ride on its VCC rail and 5 V there would put 5 V levels into non-5-V-tolerant C3 GPIOs. **New `D-84`** records all three rules, and `SYSTEM_WIRING.md` §6 now carries them as a subsection of the LX-LCBST chapter, where the trimpot instruction already lived. **The numbers in (c) are arithmetic from the regulator's behaviour, not bench readings** — nothing has been run on a cell yet, so the charge current, the boost's quiescent drain and the switch-off drain are all still unknown (O-1d). **The sibling repository `fp-2` was given the same two corrections**, since the hardware is the same there: its BOM, `SYSTEM_WIRING.md` §2.3/§3.4/§5.3, its page-4 diagram and its decision log all now say 22 Ω and carry the same three supply rules.

- 2026-11-11 (**the real wearable firmware is back on the board, and the flash is provably the shipped image**) | The owner changed the vibration motor, confirmed it now vibrates perfectly, and asked for the original wearable firmware back on the board. **Flashed on COM30** (the SuperMini) with the one command the repo already had: `python firmware/tools/flash_node.py --port COM30 --sketch wearable` -- compiled from source with the WeAct FQBN `esp32:esp32:weact_studio_esp32c3`, **685,296 B written at 0x00010000 in 2.5 s, "Hash of data verified."**, hard reset over RTS, and the tool printed `wearable on COM30 is running the new firmware`. **The boot log is the proof it is the real firmware and not the bench test**, and it is saved at `research/out/wearable_boot_20261111.log`: the ESP-ROM banner, then `CryNet wearable — BLE alert client (stage 2d)`, then `[oled] up`. **The flashed image is behaviourally identical to the offline image in `firmware/prebuilt/wearable/`:** the source was rebuilt to a scratch directory and compared byte for byte with the committed `app.bin` -- **74 bytes differ out of 685,296**, in exactly three places (the app descriptor's build stamp, the app's own SHA-256, and the trailing checksum), which is the same 74-byte difference class the 2026-11-07 entry recorded for this unit; the code is the same firmware. **`check_prebuilt.py` still passes** (`wearable source fingerprint 7071ab87... over 1 files - up to date`, and the images match `SHA256SUMS`) and `check_artifact_drift.py` still passes, so nothing shipped went stale and nothing had to be rebuilt. **Two things the owner should know about the log, and one of them is a trap.** (1) **`[oled] up` does not mean a display is connected.** The owner had the wearable on a breadboard **with no display**, and the library still reports the panel as up: `Adafruit_SSD1306` 2.5.17's `begin()` **returns `true` unconditionally** (the only failure path is the framebuffer `malloc`) and **never checks that the panel acknowledged** -- `ssd1306_command1()` and `ssd1306_commandList()` discard `endTransmission()`'s return

value, so a bus with nothing on it is indistinguishable from a working panel. **The `[oled] SSD1306 not found at 0x3C` line can therefore never fire**, and the firmware's `g_oled_ok` is `true` on a board with no display: the screen code runs and writes 1 KB per frame into the void. That is harmless (no I2C timeout, the writes are simply NACKed) but it means **the OLED must be judged by eye once it is attached, never by this log line**, and a future "the display is dead" symptom will not be caught by the firmware either. **The custom pins survive** the library's own `wire->begin()` call, because the ESP32 core 3.3.12 refuses to re-init a bus that is already running ("Bus already started in Master Mode."), so SDA 5 / SCL 6 stay as `Wire.begin(PIN_SDA, PIN_SCL)` set them. (2) **There is no `[ble] hub found` line, and there should not be:** the hub is not plugged in on this bench (only COM30 is present; the hub's CH343 port is gone), so the wearable is scanning for a hub that is not there, which is the correct behaviour. **What the owner has to know to make the wrist buzz with the real firmware:** the buzz is driven by an **alert arriving over BLE from the hub**, not by anything local -- `haptic_trigger()` is called from the alert/state notification handlers, and the buzz is 2500 ms long with a 250 ms pulse on/off inside it. With no hub on the bench the motor will stay silent no matter what, so the next test is: hub powered, an alert sent from the hub console (`alert <cot>`), and the wrist buzzes once per cry. **The bench test sketch stays in the repository** (`firmware/wearable/haptic_test/haptic_test.ino`, one command: `--sketch haptic_test`) with its restore line in the header, so the motor can be proven again on a fresh board without hunting for the command. **Still open on this build (unchanged):** the fitted stall cap is the owner's **22 ohm**, not the 10 ohm the documents and the diagram still say, so the buzz will be weaker than the design assumes and the documents are out of date; and nothing has been measured on the battery path yet (O-1d) -- the boost trimpot still has to be set to ~5.1 V with a meter **before** any board is attached.

- 2026-11-11 (**the wearable's vibration motor was proved on the bench, on a breadboard, without the OLED**) | The owner has the wearable wired on a breadboard with the BC547 haptic switch and the real motor, no display, powered from the SuperMini over USB (not from the battery yet), and asked for a sketch that drives the motor so he can see that it vibrates. **New `firmware/wearable/haptic_test/haptic_test.ino`**: GPIO7 HIGH turns the BC547 on, LOW lets it stop, 3 s on / 2 s off, forever -- exactly the timing the owner asked for, and the same pin and the same switching sense the real wearable firmware uses (D-77). `firmware/tools/flash_node.py` gained a `--sketch haptic_test` choice, so it is one command: `python firmware/tools/flash_node.py --port COM30 --sketch haptic_test`. **Built and flashed on COM30** (the SuperMini, USB `303a:1001`) with the WeAct FQBN `esp32:esp32:weact_studio_esp32c3`: 274,880 B written, hash verified, and the serial log then showed the expected `motor ON` / `motor OFF` / `motor ON` ... cycle, so the sketch is running and toggling GPIO7. **The motor's own behaviour is still the owner's observation to make** -- this proves the firmware drives the pin, not that the motor turns. **This image replaces the wearable firmware on that board**; put the real one back with `python firmware/tools/flash_node.py --port COM30 --sketch wearable` (or the built image in `firmware/prebuilt/wearable/`), and `check_artifact_drift.py` still passes, so nothing prebuilt went stale. **Still open on this build (unchanged by today's test):** the fitted stall cap is the owner's **22 ohm**, not the 10 ohm the documents and the diagram still say, so the buzz will be weaker than the design assumes and the documents are out of date; and nothing has been measured on the battery path yet (O-1d) -- the boost trimpot still has to be set to ~5.1 V with a meter **before** any board is attached.

- 2026-11-07 (**the offline images now prove they are up to date, and `copy_sketches.bat` is no longer a step you have to remember**) | Two questions from the owner. **(1) *"must I run `copy_sketches.bat` when I want to upload to the node?"* -- No.** That step belongs to Way C (building in the Arduino IDE) and only matters when the node's **code has been changed**; flashing a node with the finished firmware is Way A (`flash.bat`), which never involves the IDE or the sketchbook. The guide had put the copy as step 1 of C4 with no "only if you changed the code" on it, so it read as a step in *uploading*. It now says so explicitly, at the top of the guide, in a table ("Flashing a node that already has the firmware -> No"; "Changing the node's code -> Yes"; "Just watching -> No"), and again in C3 and C4. **Checked the claim behind it rather than repeating it:** the copy exists because the IDE's file picker starts in the sketchbook, so editing the repository's `.ino` and opening *that* would build the stale copy. The IDE's own `selectSketch()` accepts any path, and `arduino-cli compile` of `firmware/node/node_crynet` **directly from the repository, outside the sketchbook, succeeds** (983,177 B, exit 0) -- so the constraint is the IDE's picker default, not a hard rule, and the copy remains the safe habit. **(2) *"make sure the files that are to be used to upload offline are always updated."*** Nothing had been forcing `firmware/prebuilt/` to match the source: a sketch could be edited and committed while the `.bin` files stayed old, and the next flash would put firmware on a board that is not in the repository. **Three guards now exist.** (a) **New `firmware/prebuilt/BUILD_INFO.json`** records, for each unit, the SHA-256 of **every source file that went into it** (52 for the hub, 1 each for the node and the wearable), plus the rules used to find those files. It holds nothing that varies by machine, so it is byte-identical everywhere -- **verified by rebuilding it at a different path with no git present and diffing: identical.** The volatile part (which commit, which tool versions, when) went to `firmware/prebuilt/.build-state.json`, which is git-ignored. (b) **New `firmware/tools/check_prebuilt.py`** recomputes those hashes and compares: it names the files that moved (`changed firmware/node/node_crynet/node_crynet.ino`) and exits 1. **Verified by drift probe** -- appending a line to the node sketch makes it fail, reverting makes it pass. `firmware/prebuilt/check_prebuilt.ps1` does the same in **PowerShell only** (no Python needed) and is what `flash.bat` runs before writing anything, so **a stale or damaged image is refused rather than flashed**. (c) **New `.github/workflows/prebuilt-check.yml`** rebuilds all three units on every push and fails if the result differs from what is committed. **The hashes are of content with line endings normalised**, so a clone that checked out CRLF gives the same answer as one that did not -- otherwise the CI check would fail for the wrong reason. **Both checkers also handle the case that matters for the offline path: `firmware/prebuilt/` copied on its own, with no source beside it.** There is nothing to compare against, so they say so plainly ("not checked -- the source is not in this folder") and carry on with the `SHA256SUMS` check, which still proves the images are intact; they do not claim a freshness they could not establish. **Verified by copying `firmware/prebuilt/` alone to a temp folder and running the PowerShell check there.** **Also verified the images really are current:** a full `make_prebuilt.py` rebuild of all three units came back **byte-identical** to what is committed (`git status` clean), with `firmware.bin` `f7f09a34...`, node `app.bin` `aabb406f...`, wearable `app.bin` `193cdaaf...` -- the same hashes as before, so the new record describes exactly the images that ship. **Docs updated:** a new section C7 in `GETTING_STARTED_FLASHING.md` ("Keeping the offline images up to date"), a matching section in `FLASHING.md` section 10, the `firmware/prebuilt/README.md`, and the top-level `README.md` table. **The two rules are unchanged:** write the regions, never one merged image (it would erase NVS, where a node keeps its number), and the hub images are DIO (a QIO image watchdog-resets the S3 at power-on).

- 2026-11-07 (**the flashing guide is now written for someone who has never flashed anything**, and the whole job can now be done by double-clicking a file) | The owner said it plainly: "you always been the one flashing, I never flashed with platformIO IDE or arduino IDE, and I don't even know how to use platformIO IDE, I don't know where the files to flash are -- that is what I want: a step-by-step guide on from start to finish." The earlier `FLASHING.md` was a **reference for someone who already knew the toolchain**: it opened with the PlatformIO package path, assumed what a `.bin` was, and never said where on the disk anything lived. **New `GETTING_STARTED_FLASHING.md`** is the walkthrough, in three ways to do the job, easiest first: **Way A** -- double-click `firmware/prebuilt/flash.bat`, press 1/2/3, type the port; **Way B** -- PlatformIO IDE for the hub (VS Code is already installed, the PlatformIO extension is already installed, and `firmware/hub_infer/` is already a PlatformIO project -- `platformio run` rebuilds the hub **offline in 15 s** and the three rebuilt images are **byte-identical** to the shipped `firmware/prebuilt/hub/` ones); **Way C** -- the Arduino IDE for the node and the wearable, which are ordinary `.ino` sketches. The guide names **every file by its full absolute path**, explains what a COM port is and why the MAC address is the board's real identity, gives the hub-first order and the reason for it, and lists the exact log lines that mean success. **New helpers, so nothing needs typing:** `firmware/prebuilt/flash.bat` (double-click flash: menu, port list, **refuses a port that is not plugged in** -- the I-40 lesson -- automatic backup to `CryNet_Backups` in the home folder, flash, then `run`), `firmware/prebuilt/watch.bat` (the Serial Monitor without the Arduino IDE), `firmware/prebuilt/list_ports.ps1` and `list_ports_verbose.ps1`, `firmware/tools/copy_sketches.bat` (copy the `.ino` files into the Arduino sketchbook -- the rule that catches everyone, because the IDE only builds what is inside its own sketchbook) and `firmware/tools/serial_monitor.py` (the same monitor as a script). **Verified on this machine:** the hub rebuilds offline and matches

the shipped image exactly (`f7f09a34...` for `firmware.bin`, `aefe21ad...` for `bootloader.bin`, `463793a6...` for `partitions.bin`); `flash.bat` was driven end to end against a real port -- menu, port list, the invalid-port guard ("There is no port called COM99 on this PC right now."), the backup, the backup-failure escape ("Flash anyway without a backup?") and the flash-failure path all behave; `copy_sketches.bat` brings the sketchbook copies back in sync with the repository (they had drifted). **The two rules are unchanged:** write the regions, never one merged image (it would erase NVS, where a node keeps its number), and the hub images are DIO (a QIO image watchdog-resets the S3 at power-on).

- 2026-11-07 (**the firmware can now be flashed offline, by the owner, with `esptool` alone**) | The owner said plainly what the flashing guide was *for*: *"in case I get new controllers, I want to be able to upload the firmware of the units — the hub, the nodes and the wearable — myself, offline, without having to come here and ask you."* The previous `FLASHING.md` did not serve that: it described a procedure that runs on **this** machine, with **this** machine's compilers. **A fresh clone of the repository could not flash a hub at all**, because `firmware/hub_infer/components/esp-tflite-micro` and `esp-nn` are deliberately untracked (`fetch_deps.sh` pins them by commit and clones them from GitHub, which needs network), and `.pio/` is gitignored — so a clone has neither the dependencies nor the built images. The fix is to ship the built images in the repository and make `esptool` the only tool the flash path needs. **New in `firmware/prebuilt/`:** the hub (`bootloader.bin` 22,560 B / `partitions.bin` 3,072 B / `firmware.bin` 1,335,248 B), the cot node and the wearable (`bootloader.bin` / `partitions.bin` / `boot_app0.bin` / `app.bin`), plus `SHA256SUMS` over every one of them. **New tools:** `firmware/tools/flash_unit.py` (flash one unit from those images, `--check` verifies the toolchain and every hash before anything is written, `--backup` reads the whole flash first, `--dry-run` prints the exact `esptool` command, `--list` shows the ports, and it **never guesses a port** — the I-40 lesson) and `firmware/tools/make_prebuilt.py` (rebuild the images from source and refresh the hashes). **`FLASHING.md` is rewritten around this**: §0 the one tool needed and how to get it on a new machine, §1 clone-or-copy, §2 `--check`, §3 identifying a board by MAC (with §3d for a board that has never been flashed: `esptool chip-id`), §4 the hub-first order and why, §5 one command per unit *and* the same thing by hand, §6 what a brand-new board additionally needs (the node id is MAC-derived — `(mac[5] & 0x0F) + 1`, so 1..16 — and the hub watches only cots 1 and 2), §7 verification, §8 the failures, §9 backup and rollback, §10 rebuilding from source, §11 what D-83 deliberately removed, §12 the one-line version. **Two facts established while doing it:** the hub build is **reproducible and needs no network** once the two vendored components are on disk — the rebuild of 2026-11-07 produced a `firmware.bin` **byte-identical** (md5 `9a79f34ee5d08384a12f051b5a817a34`) to the one flashed to the bench units; and the Arduino sketches must be compiled with **`-j 1`**, because the parallel build dies with `cc1plus.exe: out of memory` on this 16 GB machine (the node build takes ~4 min that way, the wearable longer — its BLE library is the cost). **The images are written region by region, never as one merged file**, because a merged image would write `0xFF` over the gaps and **erase NVS** — the node's id lives there, and `0x9000–0xE000` is skipped by the offsets on purpose. **A third check, the useful one:** the newly built wearable images were compared against `wearable_baseline_COM30.bin` (the whole-flash backup of the wearable from 2026-11-06) — the bootloader and the partition table are **byte-identical**, and the application differs in **74 bytes out of 685,296**: the app descriptor's build stamp (`18:03:56 Nov  7 2026` against `10:42:51 Oct 30 2026`) and the two hashes that follow from it. **So the wearable's firmware has not changed since the bench flash.** The same test on the node — building the D-83 node sketch a second time and comparing the two builds — came back with **69 differing bytes out of 983,328**, again only the app's SHA-256, the build stamp and the trailing checksum, which says two things at once: the Arduino build is **deterministic for unchanged source**, and the image shipped in `firmware/prebuilt/node/` is functionally the same firmware that was flashed to the bench units today. The node's app image is a different matter against the *pre-D-83* backup: recompiling after the D-83 change moves every address, so it differs there in 97 % of its bytes, as it must. No board was plugged in during this work, so **no live flash was performed**; what was verified is that the images build, that the hashes match, that the wearable image is the firmware already on the bench board, that the node build is deterministic, and that the `esptool` command line the tool produces is the one the 2026-11-07 bench flash used.

- 2026-11-07 (**FLASHING.md — the flashing procedure written down as a guide**) | The owner is mid-soldering and asked for a full step-by-step guide to flash every unit, so the procedure the 2026-11-07 bench flash actually used is now `FLASHING.md` and the README points at it. It covers: the tool locations (`esptool` from PlatformIO for the hub, `arduino-cli` from the Arduino IDE install for the C3 boards); **backing up each board's whole flash before touching anything**; identifying boards by MAC with `who_is_who.py` instead of trusting COM numbers; **the hub-first order** (the old hub cannot read the new 32-byte telemetry, the new hub reads both); the exact good-boot-log lines to look for (`RESULT: PASS — 4 probes`, no `bat` field, `Hash of data verified`); why `--merged` must not be used for the hub (QIO makes the S3 watchdog-loop); the zero-watchdog behaviour with no microphone fitted; unit-by-unit verification; the bootloader-recovery `run` command from I-40; and whole-flash rollback from the three `%TEMP%` backups. Every flag quoted in it was checked against the tools, and a superseded same-day draft (`FLASHING_GUIDE.md`) was deleted so only one guide exists.

- 2026-11-07 (**the D-83 firmware was flashed onto all four units and verified live**) | The owner plugged the units in and asked for the updated firmware to be uploaded. **The port map was established first, not assumed.** `Get-CimInstance Win32_PnPEntity` showed **COM32 = the hub** (CH343, VID 1A86 PID 55D3) and two Espressif native-USB boards on **COM30 and COM31**, and `firmware/tools/who_is_who.py` read the boot banners: **COM31 = node 1, MAC `90:DA:72:85:C5:44`**; **COM37 = node 2, MAC `90:DA:72:87:CB:98`** (it appeared when the last board was plugged in); **COM30 = the wearable**, confirmed by the `%TEMP%/wearable_baseline_COM30.bin` backup being named for that port — its firmware was a foreign "sensor = … output = …" sketch from the separate system test, as the owner had said. COM30's firmware was replaced with the wearable sketch and the other three with the CryNet firmware. **The hub was flashed first, deliberately:** the new hub reads both the old 34-byte and the new 32-byte telemetry, but the *old* hub could not read the new node format, so hub-first is the only order in which no node ever meets a hub that misreads it. **Verified live, not assumed:** the hub's boot self-check reads **`RESULT: PASS — 4 probes, front end and model agree with the host`**; **`ili9341: the panel is up: 320 x 240, rotation 2`** (the hub's TFT is connected, even though the owner thought the displays were off — the *wearable's* OLED is the one not attached); both cots are online (`node 1 is online`, `node 2 is online`) on the **new telemetry format, with no `bat` field in the log line** where the old build printed `bat %u mV`; the node consoles print **no battery line** and their ids are unchanged (node 1 on COM31, node 2 on COM37, both saving to NVS); and the wearable is connected over BLE and reports **`state: 2 cot(s), mask 0x00`**. **This flash also put the I2S zero-watchdog (D-79) onto the nodes for the first time** — it was committed but deliberately never flashed while the owner was soldering, and O-1c said it needed a played-cry end-to-end run afterwards. **That run has not happened yet, and the reason is now settled: the owner confirmed that only the boards were connected — the INMP441 microphones are not attached at all.** Both nodes reading `rms 0.0000` and `mean 0` — exact digital zero, every sample — is therefore the *correct* behaviour for a board with no microphone on it, not a fault. The I2S is producing frames (`pump calls=15 empty=0 frames=3840`) and every one of them is silence, because there is nothing on the data line to be anything but zero.

One thing this does teach, and it is worth writing down: the zero-watchdog cannot tell an *unattached* microphone from a microphone that has *died*, so on a board with no mic it will restart the I2S forever — node 1's watchdog restarted it **8 times** during the run, and one burst stalled at 274 of 286 fragments before the resend mechanism recovered it. With a real INMP441 fitted the trigger should stay quiet, because a live mic in a quiet room delivers noise, not zeros. **If the restarts continue once a microphone is attached, the watchdog is firing on the installed mic and needs a look** — that is exactly the distinction the commit message for 9b94d60 warned about ("it detects exact zeros only, so a mic that delivers quiet-but-nonzero noise is still not covered").

Next: attach the INMP441 modules to both node boards, then do the played-cry end-to-end run that D-79/O-1c is waiting for. The flash itself is finished and correct; nothing in it needs redoing. **Rollback remains possible for every board:** `%TEMP%/fullflash_31.bin`, `%TEMP%/fullflash_37.bin`, `%TEMP%/wearable_baseline_COM30.bin` (md5 `0ba262cd9dd6e6f255438e79dbeb729a`).

- 2026-11-07 (**a note was added about the printed wiring PDF, and two dangling cross-references were fixed**) | `SYSTEM_WIRING.pdf` in this repository is a **snapshot exported on 2 November 2026** and is out of date: it predates the battery design (D-81), the two-cot split and the removal of the battery sense and the status LEDs (D-83). There is **no markdown-to-PDF toolchain on this machine** — no pandoc, no Python markdown library — so it was **flagged rather than regenerated** with something crude, and `SYSTEM_WIRING.md` now says so at the top. **The live documents are `SYSTEM_WIRING.md` and `SYSTEM_WIRING_DIAGRAMS.pdf`.** Two cross-references left dangling by the D-83 rewrite were also fixed: §7 no longer describes a charge prompt, and D-76 points at what that section used to be called.

- 2026-11-07 (**the battery SOC and every status LED were removed from the system — D-83**) | The owner decided to take the battery readout out of the system altogether, together with the connectivity and SOC LEDs, because the hub's display already shows the connection. **Nothing was flashed and no hardware was touched** — the owner is mid-solder and the units were not on USB; the firmware is written and compile-verified, and the units are waiting to be plugged in. **What was removed, in the firmware:** the node's battery sense on GPIO3 (`PIN_BATTERY`, the 16-sample average, the 800–1500 mV trust gate and the ×3.2 scaling), the cell-voltage field in the telemetry, the `BAT_CURVE` table and `battery_percent()` in `hub_display.c`, the `HUB_BAT_*` bands and the hysteresis that moved them, the `BAT ~80 %` / `BAT LOW` / `BAT CRIT` text on the cot cards, and the node's **connection LED** on GPIO10 (`PIN_LINK_LED`, `link_led_tick()`). The node's `status` command no longer prints a cell voltage, and the hub's boot log no longer prints `bat %u mV`. The **heartbeat and its ACK stay** — that is how a node knows the hub is there; only the LED that displayed the answer is gone, and the node prints the same fact on its serial console instead. **The wire format changed with it:** the telemetry's battery tier is removed, so the radio counters move down two bytes, from offset 18 to 16, and the node's telemetry payload goes from 34 bytes to 32. The hub still reads the old 34-byte layout (counters at 18) when a node has not been reflashed yet, so the counters do not turn to noise while the units are being flashed one at a time. **In the documents:** `SYSTEM_WIRING.md` §3.3 is now "No battery sensing, and no status LEDs", §5.1/§5.2/§5.3 have "battery sense: none" and "status LEDs: none" rows, and §7 is now "There is no battery readout"; `HARDWARE_COMPONENTS.md` no longer lists the 220 kΩ, the 100 kΩ, the 100 nF divider capacitor or any LED; `TWO_COT_README.md` and the panel mock-up lose the `BAT ~92%` line; and the five wiring-diagram pages were regenerated with the divider and the LEDs drawn out and a note in their place saying what is there instead. **What did *not* change:** the battery power itself — the cells (18650 on the nodes and the hub, 1000 mAh pouch on the wearable, never LiFePO4), the LX-LCBST modules, the switches, the boost converters and the charge currents are exactly as D-81 settled them. **What is given up:** nothing in the system will now warn the caregiver that a cell is nearly flat; the cells have to be watched by eye and charged before a unit goes back on a cot. **Freed pins:** GPIO3, GPIO7 and GPIO10 on the node, GPIO1, GPIO2 and GPIO17 on the hub — all free, and none of them a boot-strapping pin on its board. **Verified:** both firmware targets build clean on this machine — the hub with PlatformIO/ESP-IDF (RAM 13.0 %, flash 63.6 %) and the node sketch with the Arduino IDE's `arduino-cli` on `esp32:esp32:weact_studio_esp32c3` (75 % flash, 50 % RAM) — and the diagram generator's own layout checker passes on all five pages. **Not verified:** none of it has run on hardware; the first live run will be the end-to-end test with a played cry. Recorded as D-83, with D-81 marked superseded in part, O-1e closed (there is no divider left to measure) and O-1a's wearable SOC readout cancelled.

- 2026-11-07 (**the D-81 rationale was corrected after the owner clarified how the variable resistor would actually be used, and a source-impedance caveat on the divider was found and recorded as O-1e**) | The owner asked why the variable-resistor advice had been rejected. **The clarification was that the pot would be set once, so the output lands inside the analog pin's range, and the reading then followed the battery down.** That is a fair description, and it **corrected an overstatement in the record**: D-81 had said that "a pot's wiper position cannot be read by the firmware, so the chip would not know the division ratio". As a blanket claim that was **wrong** — a **three-terminal** pot used as a genuine divider (one end to B+, one end to B−, the wiper to the ADC) does work, and the firmware does not have to read the wiper at all, because the ratio can be measured once at assembly and handed to the firmware as a constant. **The decision is unchanged, but the reasons are now the correct ones.** *(1) A **two-terminal** pot — a rheostat between the cell and the pin — cannot divide at all:* the ADC pin's only path to ground is its own high-impedance input, so turning the pot down barely moves the voltage and the pin still sits near 4.2 V, past the C3 IO's 3.6 V absolute maximum. Dividing needs a second leg to ground. *(2) A three-terminal pot works, but it costs **per-unit calibration** (a meter reading and a per-unit constant on every board, where the node firmware's ×3.2 is baked in), **silent drift** (a wiper is a mechanical setting that moves with heat, vibration, wear and the knocks a cot gives the node, and the reading keeps looking plausible while it moves), and **magnitude**: `BAT_CURVE` moves 100 mV = 20 percentage points around 3.8 V, so **1 % of ratio error is already ~10 points of SOC error**, against a pot's 1–5 % tolerance-plus-drift. A fixed 1 % pair is ±1 % on the ratio, known at the design stage, on every unit, with no per-unit step. *(3) The linear-map objection is untouched by any of this — it is about the **curve**, not the divider, and 3.0–4.2 V linear still calls a quarter-full cell 58 %.* **One genuine finding came out of checking the ADC's input network:** the divider's source impedance is **68.75 kΩ** (220 kΩ ‖ 100 kΩ), above the ~10 kΩ an ESP32 ADC likes. The **100 nF on the sense node is what makes that workable** — the ADC's sample-and-hold takes its charge from the capacitor rather than from the divider — so **the capacitor is load-bearing, not a nicety, and must not be omitted.** The fix is available without touching the firmware or the ratio: a **22 kΩ / 10 kΩ pair gives the same ×3.2, the same 1.31 V at a full cell, the same 800–1500 mV trust-gate window**, and drops the source impedance to 6.9 kΩ, for about 118 µA more drain (131 µA against 13 µA — under 0.1 % of the load). **That is recorded as O-1e and is a bench decision, not a change applied to wiring the owner is soldering now:** on the one battery-powered assembly, compare the reported cell voltage against a meter at full, mid and near-empty; if it tracks within a few per cent, the 220 kΩ / 100 kΩ pair stays and nothing changes. | `DECISIONS.md` (D-81, new O-1e), `HARDWARE_COMPONENTS.md`, `SYSTEM_WIRING.md`, `tools/wiring/diagram_lib.py`, `tools/wiring/page_hub.py`, `tools/wiring/page_wearable.py`, `SYSTEM_WIRING_DIAGRAMS.html`, `SYSTEM_WIRING_DIAGRAMS.pdf`, `SYSTEM_UPDATE.md` | **Nothing was flashed and no hardware was touched.** This is a **documentation correction** on the same branch as the battery work — it changes the *justification* on file, not the design, and **not one component value changed**: the divider is still 220 kΩ / 100 kΩ with a 100 nF across the 100 kΩ, and the firmware's ×3.2 is untouched. The five-page diagram was rebuilt and the layout checker passed with **zero `LayoutError`s on all five pages**. **Nothing is merged to `main`.** The parts list is unchanged from the previous entry, so anything already bought against it still stands. |

- 2026-11-07 (**the battery design was settled: a fixed divider rather than a variable resistor, a curve rather than a linear SOC map, an 18650 on the nodes and the hub, a pouch on the wrist, and the runtime cost of the 5 V→3.3 V linear stage named**) | The owner asked about two things: **battery SOC measurement** — they had been advised to put a *variable resistor* between the cell and the ADC pin, read the maximum (4.2 V) as 100 % and the minimum (3.0 V) as 0 % — and **which cell to use**, having been told lithium-ion cells are pricey. **That advice was checked against the design and rejected on three counts, and the answer is recorded as D-81 and D-82.** *(1) As described it would put 4.2 V on a 3.3 V pin.* The C3's IO is 3.3 V only, so a divider is mandatory whatever element is used to make it. *(2) The firmware cannot read a wiper position.* A pot's ratio would drift with heat, vibration and age, and the chip would have no idea what it was dividing by; a worn wiper is a real failure mode on a unit a cot bumps every night. **Two 1 % resistors (220 kΩ / 100 kΩ, ratio ×3.2) give a ratio the firmware knows exactly**, and `analogReadMilliVolts()` already applies the chip's factory-calibrated ADC reference — so the only residual error is the resistors' own tolerance, which the existing banding absorbs. *(3) The 3.0–4.2 V linear map is chemically wrong.* A lithium cell runs roughly 4.20 V ≈ 100 %, 3.85 ≈ 55 %, 3.70 ≈ 25 %, 3.60 ≈ 10 %, 3.40 ≈ 0 %, so a straight line would report a genuinely quarter-full cell as "58 %" — wrong across exactly the range the cell spends most of its life, and wrong in the direction that says "fine" when the answer is "not fine". **`hub_display.c` already held the correct table (`BAT_CURVE`) and already printed it as `BAT ~80 %` with a tilde**; the tilde is doing real work and must not be dropped. The bottom of the scale was moved from 3.0 V to **~3.3 V**, because 3.0 V is roughly where a protected cell's protection circuit cuts out — end-of-life, not "empty". The reading is also taken *under load*, so the cell sags 100–200 mV while the boost converter and radio draw current: the firmware averages 16 samples, filters through the 100 nF, and the hub bands with hysteresis. **A percentage read this way is a caregiver-facing estimate, not a fuel gauge**; if a true SOC is ever wanted, the part is a **MAX17048** on I²C (the hub and wearable have a spare bus; the node's microphone occupies I²S, so nodes keep the divider). **On the cell:** "Li-

ion" and "Li-Po" are the same chemistry — the decision is *format*, and it is **opposite for the two kinds of unit**. A node sits on a cot frame within a baby's reach, so it gets an **18650** (3.7 V nominal, 4.2 V full, 3000 mAh, protected, in a holder): more energy per unit volume, and a steel can that cannot be punctured. A wrist cannot take an 18650 — 18 × 65 mm and 48 g — so the **wearable keeps a flat 1000 mAh Li-Po pouch** (raised from the original 500 mAh, and the module's ~1 A charger must come down to **~400–500 mA**, since 1 A into 1000 mAh is 1 C). **LiFePO4 is excluded from the whole system**: the module charges to 4.2 V and an LFP cell tops out at 3.65 V, so it would be destroyed. **The runtime finding (D-82):** the chain is **cell → boost to 5.1 V → the board's 5 V pin → its onboard linear regulator → 3.3 V**, and a linear regulator passes *current*, not power, so the cell supplies roughly **1.8× the 3.3 V load current** while the difference is burned as heat — about 160 mA from the cell for a 100 mA load. That projects to **roughly six hours** for the wearable on a 1000 mAh pouch and rather less than a day for a node on a 3000 mAh 18650. *These are projections from the regulator's behaviour, not measurements — no runtime has ever been measured on this system.* The largest single lever is to **feed the board's 3V3 pin from a 3.3 V buck-boost** and skip the linear stage, which would roughly double runtime; that changes the power architecture, so it is recorded as an option to evaluate at the bench rather than applied to the wiring the owner is soldering. **A contradiction between the repository's own documents was found and fixed:** `HARDWARE_COMPONENTS.md` still described the **three**-cot system with a **DRV2605L** haptic and listed the divider as **100 kΩ + 100 kΩ giving 2.1 V** — which is the wrong pair for firmware that divides by 3.2, and had already been flagged in an earlier entry of this log. The BOM now describes the **two**-cot system, the **BC547** haptic that is actually fitted, and the **220 kΩ / 100 kΩ** divider the firmware expects; the consolidated purchasing count, the assembly-supplies counts and the cell quantities were corrected with it. `SYSTEM_WIRING.md` §2.3 (the DRV2605L connection list) is now marked **SUPERSEDED** and points at §5.3 rather than being left to contradict it, and §7 gained the voltage-to-percent curve and the reason the reading sags. The five-page diagram was rebuilt (HTML + PDF, 219,351 bytes), the layout checker passed with **zero `LayoutError`s on all five pages**, and pages 1, 2 and 4 were re-rendered to PNG and inspected. | `HARDWARE_COMPONENTS.md`, `SYSTEM_WIRING.md`, `DECISIONS.md` (D-81, D-82, O-1d), `tools/wiring/page_overview.py`, `tools/wiring/page_node.py`, `tools/wiring/page_hub.py`, `tools/wiring/page_wearable.py`, `tools/wiring/diagram_lib.py`, `SYSTEM_WIRING_DIAGRAMS.html`, `SYSTEM_WIRING_DIAGRAMS.pdf`, `SYSTEM_UPDATE.md` | **Nothing was flashed and no hardware was touched.** This is a **design and documentation change on branch `feature/battery-soc-and-cell-design`**, not on `main`; the branch is not merged. **The parts list changed, so anything already bought against the old BOM should be checked:** the divider resistors are **220 kΩ + 100 kΩ** (not two 100 kΩ), the LED current-limiters are **1 kΩ** per `SYSTEM_WIRING.md` §5.1, the haptic is a **BC547 + 1N5819 + 10 Ω** (no DRV2605L breakout to buy), and the **cell count is four** (2 nodes, 1 hub, 1 wearable) — three 18650s and one pouch, not five pouches. **New open item O-1d:** no runtime has been measured and the module's charge current, quiescent drain and switch-off drain are all still unmeasured — the bench procedure in `SYSTEM_WIRING.md` §6 (build ONE assembly, measure charge current, set the trimpot to 5.1 V, confirm switch-off behaviour, compare against the USB baseline) is how that gets closed. **The node's I2S watchdog (D-79) and the multi-cot wearable candidate (D-80) are untouched by this entry** and remain as they were.

466

467 - 2026-11-06 (**the two-cot system was given its own GitHub repository and remote, and its decisions were written down as D-75…D-80**) | The two-cot system had been living in `fp-two-node` with **no git remote at all** — nothing could be pushed, and the sibling three-cot repository `fp-2` had its own remote (`crynet` → `Amazo123/CryNet`). **A new public repository was created, `Amazo123/CryNet-TwoNode`**, and `fp-two-node` now has it as `origin`. **Published:** `main` (created from the working branch, which is the branch everything landed on today), plus `feature/wiring-diagrams-x2-corrected`, `feature/wearable-multi-cot`, `feature/two-node-display` and `fix/link-resend-bitmap-verification`. **The two repositories share their history up to `99bb257`** and then diverge, so the model, the signal contract and the whole decision record remain traceable in one place without the two firmware trees drifting into each other — which is exactly what happened to the wearable sketch before the split. **This repository's documents had been a stale copy of `fp-2`'s, and that is now fixed.** `DECISIONS.md` said `Repo: fp-2`, described **three** cot nodes and a **DRV2605L** haptic, and carried the three-cot system's open items — including O-1 "the wearable does not exist yet", which was true of `fp-2` and false here. **The identity blocks in all three documents now name this repository, its branch, its remote and its sibling.** `DECISIONS.md` §0 describes two cot nodes and the BC547 haptic; the hardware table in `PROJECT_HANDOFF.md` now reads `Cot nodes ×2` and `SSD1306 OLED + BC547 haptic motor`; `SYSTEM_UPDATE.md` and `PROJECT_HANDOFF.md` say the same. **New subsection `DECISIONS.md` §1.9 "The two-cot system (this repository)"** records the six decisions where this build deliberately differs from `fp-2`, so the two log sets can be read side by side: **D-75** its own repository; **D-76** the wearable has no status LEDs, the OLED carries the state; **D-77** the haptic is a BC547 low-side switch, not the DRV2605L the planning documents still describe; **D-78** the hub backlight stays on 3V3, permanently on, no sleep mode; **D-79** the node's I2S watchdog is committed but not flashed; **D-80** the multi-cot wearable behaviour is held on its own branch. **Three open items were added or corrected** (O-1 marked resolved, O-1a the wearable's missing battery readout, O-1b the untested dual-cry path, O-1c the watchdog waiting on the soldering). | `DECISIONS.md`, `SYSTEM_UPDATE.md`, `PROJECT_HANDOFF.md` | **The owner has started soldering the final wiring.** That is why the node's I2S watchdog (D-79) is committed but **not flashed**: a node should not be reflashed mid-solder. It goes on once the hardware is settled, and it then needs a played-cry end-to-end run to confirm the node still streams and the hub still hears it. **Nothing was flashed and no board was touched** — at the time of this entry no ESP32 was on USB (the only COM port present was `COM3`, the Intel management interface). **Rollback remains possible for every board:** `%TEMP%/fullflash_31.bin`, `%TEMP%/fullflash_37.bin`, `%TEMP%/wearable_baseline_COM30.bin` (md5 `0ba262cd9dd6e6f255438e79dbeb729a`). **The GitHub credential is already configured** (`credential.helper=manager`, account `Amazo123`), so no new authentication was needed; the token was read from the credential helper and used for the one API call that created the repository, and it was not written anywhere.

468

469 - 2026-11-06 (**the two candidates were committed, and the untested one was moved onto its own branch**) | The owner confirmed the present deployed firmware works and asked for the working changes to be committed, with the one that has never been tested held back. **Three commits landed on `feature/wiring-diagrams-x2-corrected`.** (1) `db416e6` - the `capture_multi.py` fix (a busy port no longer aborts the whole capture; the log opens on first use). (2) `9b94d60` - the node's I2S zero watchdog: five seconds of consecutive exact-zero samples triggers an I2S restart from the capture task, a failed teardown or warmup reboots rather than reusing broken handles, and the restart count is reported in the status line. **Its commit message is explicit that the fault-injection test proves the code executes, not that it recovers the original physical fault.** (3) `345665a` - the backlight-on-3V3 diagram change and the change-log entry for it. **The multi-cot wearable change was moved off this branch onto `feature/wearable-multi-cot` (`4e6049f`)**, because two cots crying at the same time has never been tested: the branch holds the multi-detection alert face, the haptic when a second cot joins, and the suppressed CLEARED, with the single-cot path unchanged in behaviour. It stays there until the dual-cry test can be run. **So `feature/wiring-diagrams-x2-corrected` now carries only work that is either proven on the bench or is a self-contained guard that changes no existing behaviour - the branch's firmware is the firmware the owner already has running, plus the watchdog that has not been flashed.** | `firmware/tools/capture_multi.py`, `firmware/node/node_crynet/node_crynet.ino`, `firmware/wearable/wearable_crynet/wearable_crynet.ino` (branch `feature/wearable-multi-cot`), `SYSTEM_UPDATE.md` | **Nothing was flashed and no board was touched: at the time of this commit no ESP32 was on USB** (the only COM port present was `COM3`, the Intel management interface), and this repository has **no git remote configured**, so there was nothing to push to either - `git push` is not possible from here until a remote is added. **Rollback remains possible for every board:** `%TEMP%/fullflash_31.bin`, `%TEMP%/fullflash_37.bin` and `%TEMP%/wearable_baseline_COM30.bin` (md5 `0ba262cd9dd6e6f255438e79dbeb729a`). **Before the watchdog is flashed to a node,** it needs a played-cry end-to-end run to confirm the node still streams and the hub still hears it, and the existing notes should be read: it detects *exact* zeros only, so a mic that delivers quiet-but-nonzero noise is still not covered. **Merging to `main` is still the owner's call** and has not been done.

470

471 - 2026-11-06 (**the hub backlight goes back on 3V3 and stays on: the owner chose no sleep mode, so the GPIO4 MOSFET came off the diagrams**) | Earlier today the hub page was redrawn with the TFT backlight switched by a 2N7002 on GPIO4 so the panel could be dimmed or turned off for a future sleep mode. **The owner has now decided against that: the LED stays on 3V3, the backlight is permanently

on, and there is no sleep mode.** Nothing had been soldered, so the change is on paper only. **Page 3 was redrawn.** The 2N7002, its 100 ohm gate resistor and its 100 k pull-down are gone, and the hub page shrank back from 575 to 490 units tall. The TFT box now carries the `LED` pin as a normal left-edge pin between `VCC` and `GND`, and the backlight anode branches off the same 3.3 V net that feeds the panel's VCC (a junction dot at the branch point), so there is one wire fewer and no new part. A note under the panel says `Backlight: LED pin on 3.3 V - always on, no dimming.`, and the verified caption is back to the six TFT signal pins with the LED pin explicitly described as tied to the panel's own 3.3 V. **GPIO4 is freed again.** The hub's bottom pin row is back to three pins (GPIO1 divider, GPIO17 SOC LED, GPIO2 BLE LED) with no GPIO4 label, and the firmware's `PIN_BL 4` / `ili9341_backlight_set` simply drives a pin with nothing attached, which does no harm - the existing backlight LEDC code is left in place. **Page 1's hub box** now reads `backlight on 3V3 - always on` instead of `backlight switched by GPIO4`, and its Li-Po line was rewrapped to two lines so nothing collides. The HTML and PDF were rebuilt (`SYSTEM_WIRING_DIAGRAMS.pdf`, 5 pages, 200,417 bytes) and the layout checker passed with zero problems; every page was re-rendered to PNG and inspected. | `tools/wiring/page_hub.py`, `tools/wiring/page_overview.py`, `SYSTEM_WIRING_DIAGRAMS.html`, `SYSTEM_WIRING_DIAGRAMS.pdf`, `SYSTEM_UPDATE.md` | **Nothing was flashed and no hardware was touched** - the live hub's panel LED was already tied to 3V3, so the bench matches the drawing again and the previous entry's `Still open: the backlight MOSFET is a wiring change that has not been built` is resolved by withdrawal, not by building it. The consequence the owner accepted: the glass cannot be dimmed or blanked while the board is powered, which on a system that is always plugged in or charging is the simpler and more reliable choice. **Still open, unchanged:** the hub has no ADC read and no LED firmware, the node's GPIO7 SOC LED has no firmware, and the wearable's LED question stays as `fp-2` decided it - no LEDs, the OLED carries the state.

- 2026-11-06 (**the wiring diagrams were corrected and regenerated for the two-cot system, with the hub TFT backlight moved onto GPIO4 behind a MOSFET**) | The generator was copied from `fp-2` into this repository as `tools/wiring/` and the five pages were corrected against the running firmware, then rebuilt into `SYSTEM_WIRING_DIAGRAMS.html` / `.pdf` (5 pages). **Six corrections.** (1) **Three cot nodes became two.** Page 1's unit box reads `Cot node x2`, page 2's title reads `(one per node, x2)`, and page 2's footer says `Repeat identically for node 2.` Page 5's `buy ONE module for validation; only then buy the remaining three` is unchanged and still correct - four modules are needed (node 1, node 2, hub, wearable). (2) **The hub TFT backlight moved off 3V3 onto GPIO4, switched by a 2N7002.** `hub_display.c` already treats GPIO4 as the backlight (`PIN_BL 4`, `ili9341_backlight_set` driving it with LEDC PWM, high = on), but GPIO4 had no load, so the panel's LED pin was still tied to 3V3 and could never be switched off. The page now draws the LED pin at the TFT's right edge and a real N-channel MOSFET (2N7002 TO-92 or AO3400 SOT-23): **3V3 to the drain, the source to the TFT `LED` pin** (the module's backlight cathode is internal to its GND, so switching the anode works), **the gate through 100 ohm to GPIO4 with a 100 k pull-down to ground** so the backlight is dark during reset and until the firmware brings it up. The MCU pin never sources the backlight current. The TFT's VCC stays on 3V3. The backlight is now PWM-dimmable by the existing firmware, and a future sleep mode only has to set the PWM to 0. (3) **The hub's bottom pin row was re-laid out and a pin/LED mix-up was fixed.** The bottom labels now sit at GPIO1 ~415, GPIO4 ~462, GPIO17 ~510 and GPIO2 ~558, inside the board box, and the wires that leave the bottom edge start at the matching x: the divider ADC at 420 (GPIO1), the gate wire at 462 (GPIO4), the amber SOC LED chain at 510 (GPIO17) and the blue BLE LED chain at 558 (GPIO2). **In the previous PDF the blue `BLE` wire and the orange `SOC` wire were swapped relative to the pin labels** - per `SYSTEM_WIRING.md`, GPIO2 is the BLE-status LED and GPIO17 is the SOC LED, and that is what the page now shows. The hub page grew from 480 to 575 units tall so the gate pull-down hangs below the board. (4) **The firmware-status captions are now true.** The old text said `Firmware for the divider and the LEDs does not exist yet`, which was wrong for the node: `node_crynet.ino` implements the divider (`PIN_BATTERY 3` = ADC1_CH3, a trust gate that only accepts 800-1500 mV, and a x3.2 scale - so the 220 k / 100 k values on the page are correct) and the **GPIO10 blue link LED** (`PIN_LINK_LED 10`). There is **no GPIO7 anywhere in the node firmware**, so the GPIO7 amber SOC LED is a dead wire until that firmware is written on a branch; the page now says exactly that, and notes that the bench node printing `battery none (USB, or the divider is not wired)` on USB power is the expected trust-gate behaviour, not a fault. The hub firmware has **no `analogReadMilliVolts` and no LED output at all**, so the hub's GPIO1 divider, GPIO2 BLE LED and GPIO17 SOC LED are all marked plan only - `BAT_CURVE` / `battery_band_advance` in `hub_display.c` describe the *nodes'* reported cell voltage, not the hub's own. The TFT signal pins (CS 10, MOSI 11, SCK 12, DC 13, RST 14, VCC 3V3, GND) and the GPIO4 backlight are marked verified on the live system. (5) **Mic placement was added to the node page.** The INMP441 is fixed-gain: mount it on the cot frame on a stiff lead, at least 30 cm from where the baby's head will be, never loose on the mattress. The hub refuses a saturated window (`the window is CLIPPED`), so the phone must be at least 15 cm from the mic when playing a test cry. (6) **The wearable page stays LED-free and now matches the firmware.** The BC547 haptic circuit is unchanged (GPIO7 -> 2.2 k base, 100 k base-emitter pulldown, 10 ohm in the motor's + lead, 1N5819 flyback with the cathode up, collector to motor -, emitter to GND) and the OLED stays on 3V3 / SDA GPIO5 / SCL GPIO6. The page says the OLED carries the state and shows LINK or SCAN today, and that the battery/SOC readout needs the GPIO3 divider wiring **plus** firmware that does not exist yet. **Nothing was flashed and no hardware was touched.** | `tools/wiring/build.py`, `tools/wiring/page_overview.py`, `tools/wiring/page_node.py`, `tools/wiring/page_hub.py`, `tools/wiring/page_wearable.py`, `SYSTEM_WIRING_DIAGRAMS.html`, `SYSTEM_WIRING_DIAGRAMS.pdf` | The generator's layout checker passes with **zero `LayoutError`s on all five pages** (no text-over-text, no text on a component, no wire crossing a label), and every rendered page was inspected as a PNG: no clipped label, no text on a wire, no overlapping parts. One build-side change was needed: the diagrams grew taller, so `svg.diagram` in the CSS gained a `max-height` so each page still fits one A4 landscape sheet - the PDF is 5 pages, one per diagram. **Left alone on purpose:** the page-5 LX-LCBST pad map and the bench procedure, and the `fp-2` repository itself (read only). **Still open:** the hub has no ADC read and no LED firmware, the node's GPIO7 SOC LED has none either, and the backlight MOSFET is a wiring change that has not been built - the live hub still has its panel LED pin on 3V3.

- 2026-11-06 (**the wiring diagram was reviewed line by line against the running firmware and is approved for soldering, with five specific corrections**) | The owner supplied the five-page `SYSTEM_WIRING_DIAGRAMS.pdf` they intend to solder from and asked for an approval. **Approved, with corrections — the diagram is sound in its structure and gets the pin assignments right, but five things must be fixed or added before the iron comes out.** *What the diagram gets right, confirmed against the code:* the node's microphone wiring (3V3 / GND / L/R→GND / GPIO4 SCK / GPIO5 WS / GPIO6 SD) is exactly `node_crynet.ino`'s `PIN_SCK 4 / PIN_WS 5 / PIN_DIN 6`; the **220 kΩ + 100 kΩ + 100 nF** divider is exactly what the firmware expects (it divides the pin reading by 3.2 and only trusts 800–1500 mV); the node's **GPIO10 blue link LED** is the `PIN_LINK_LED 10` the firmware already drives from the hub's heartbeat; the wearable's **OLED SDA GPIO5 / SCL GPIO6** and **GPIO7 BC547 haptic** match `PIN_SDA 5 / PIN_SCL 6 / PIN_MOTOR 7` exactly, and the BC547 circuit on page 4 — 2.2 kΩ base, 100 kΩ base-emitter pulldown, 10 Ω in the motor's + lead, 1N5819 with the cathode up — matches the firmware's `HAPTIC_MS 2500` / `HAPTIC_PULSE_MS 250` drive and the bench-proven haptic; the hub's TFT signals (CS 10, MOSI 11, SCK 12, DC 13, RST 14) match `hub_display.c`'s `PIN_MOSI 11 / PIN_SCLK 12 / PIN_CS 10 / PIN_DC 13 / PIN_RST 14`; the ESP32-S3 warnings are right (GPIO35–37 are the Octal PSRAM bus on the N16R8, GPIO0/45/46 are strapping). **The five corrections.** (1) **The hub TFT's backlight wire is missing.** The diagram draws the panel's `backlight` pin with *no wire to it*, but `hub_display.c` defines `PIN_BL 4` and `ili9341.c` drives it through LEDC for PWM dimming — **GPIO4 → panel backlight** must be drawn, and GPIO4 must not be reused. (2) **The node SOC LED on GPIO7 has no firmware.** The diagram says "firmware for the divider and the LEDs does not exist yet"; that is only half true. The **divider firmware does exist** — `PIN_BATTERY 3`, the 800–1500 mV trust gate and the ×3.2 scale are all committed and running, which is why the bench node prints `battery none (USB, or the divider is not wired)`. Only the **GPIO10 link LED and the divider** exist; the **GPIO7 SOC LED does not exist in the node firmware at all** (no GPIO7 anywhere in `node_crynet.ino`), so wiring it produces a dead LED until that firmware is written. (3) **The hub's GPIO1 divider and both hub LEDs have no firmware either.** The hub firmware contains no `analogReadMilliVolts` and no LED output at all — `battery_band_advance` and the `BAT_CURVE` in `hub_display.c` are for the *nodes'* reported cell voltage, not the hub's own. So the hub's divider on GPIO1 and its GPIO2/GPIO17 LEDs are plan only. (4) **The wearable's two LEDs contradict the other repository.** This diagram adds a blue link LED on GPIO1 and a red SOC LED on GPIO4 to the wearable, but `fp-2`'s `SYSTEM_WIRING.md` §7 says in terms "**The wearable has no status LEDs, and its normal OLED dashboard already shows LINK or SCAN as the connection state**" — and that repository's committed PDF (commit `f6104ca`, "the wearable has no status LEDs, the OLED carries the status") says the same on page 4. The attached PDF is a **newer revision than the committed one**

(it adds the two LEDs to the wearable and changes page 1's summary line to "haptic motor + battery sense + 2 LEDs"), so the two repositories have drifted. **Decision needed:** either keep the LEDs and write the firmware, or drop them and let the OLED carry the state as `fp-2` decided. They are cheap to leave unwired; wiring them without firmware gives two dark LEDs. (5) **The diagram is for three cot nodes, not the two this system has.** Every page says `Cot node x3`, page 2 says "repeat identically for nodes 2 and 3", and page 5's bench procedure says to buy "the remaining three". The system on the bench has **two** nodes. The wiring itself is identical per node, so this is a parts-count and labelling issue, not an electrical one — but the order should be **two** node assemblies. | `SYSTEM_UPDATE.md` (review only; the diagram file itself is not in this repository) | **Also checked and confirmed correct:** the LX-LCBST pad map on page 5 matches `SYSTEM_WIRING.md` §6 (B+/B− battery, VO+/VO− output, IN+/IN− left unconnected, the unlabelled front −/+ pair flagged "verify with meter", no T pad), and the three rules on page 1 are the right ones — switch off before charging, set the trimpot to 5.1 V with a meter before any board, and never power a board from USB and the boost at the same time. **The one electrical note the diagram does not carry:** the INMP441 is a fixed-gain part, so it must be mounted on the cot frame on a stiff lead at least 30 cm from where the baby's head will be, never loose on the mattress — the hub refuses a clipped window, and today's test showed exactly that failure when the phone sat too close. **Nothing was flashed and no firmware was changed by this review.**

- 2026-11-06 (**the owner is cleared to solder, and the wiring that goes on the board was read out of the running firmware rather than out of the documents, which disagree with each other**) | The owner asked whether it is safe to solder the system into its final form. It is — with the pin list below, which was read from the sketches that are actually on the bench, not from the planning documents. **The two planning documents disagree with each other and, in two places, with the firmware, so the firmware wins and the documents need correcting before anything is soldered.** *What is safe to solder now, exactly as read from the code:* **Cot node** — INMP441 `VDD`→C3 **3V3**, `GND`→**GND**, `L/R`→**GND** (left channel), `SCK`→**GPIO4**, `WS`→**GPIO5**, `SD`→**GPIO6**; optional 100 nF across the microphone's VDD/GND. The link LED is **GPIO10 → 1 kΩ → blue LED anode → GND** and the firmware already drives it. The battery sense is **GPIO3 (ADC1_CH3)**, and the firmware divides the pin reading by exactly **3.2** — so the divider must be **220 kΩ from B+ to the sense node, 100 kΩ from the sense node to B−, 100 nF across the 100 kΩ**. **Wearable** — OLED `SDA`→**GPIO5**, `SCL`→**GPIO6**, and the haptic is the owner's **BC547 low-side switch**, not the DRV2605L the two-node documents still describe: **GPIO7 → 2.2 kΩ → base, 100 kΩ base-emitter pulldown, 10 Ω in the motor's + lead, 1N5819 across the motor with the cathode up**, collector→motor −, emitter→GND. **The discrepancies found, which must be fixed before soldering.** (1) `HARDWARE_COMPONENTS.md` lists **100 kΩ + 100 kΩ** for the node divider and says that gives 2.1 V at 4.2 V; the firmware divides by 3.2, which is only correct for **220 kΩ + 100 kΩ**, and `SYSTEM_WIRING.md` §5.1 already says so — the bill of materials is wrong. (2) The same file says **330 Ω** for the indicator LEDs while `SYSTEM_WIRING.md` §5.1 says **1 kΩ**; the firmware's pin assignment is the same either way, so this is a brightness choice, but the two must agree. (3) `SYSTEM_WIRING.md` §2.3 still describes the wearable's haptic as a **DRV2605L breakout** with the motor on `OUT+/OUT−` — that is the old plan and it is **wrong for this build**: the bench hardware is the BC547 circuit above, and §5.3 in the same file contradicts §2.3. The BC547 detail is only written down in the *other* repository (`fp-2`, `SYSTEM_WIRING.md` §2.3 and page 4 of its `SYSTEM_WIRING_DIAGRAMS.pdf`). (4) The node **SOC LED on GPIO7 is not implemented in the firmware at all** — the node firmware has no GPIO7 and no SOC LED; only the GPIO10 link LED exists. Do not wire an SOC LED expecting it to work. **Also unverified and worth knowing before soldering:** the node firmware's battery reading is real but has never been exercised against an actual cell (the bench nodes report `battery none (USB, or the divider is not wired)`), and the hub's battery band has never seen a real reading either. **The order that costs least if something is wrong: build ONE node power assembly first, set the LX-LCBST trimpot to ≈5.1 V with a meter and no board attached, and run the working node firmware from it before making the second and third.** | `SYSTEM_UPDATE.md` (documents not yet corrected) | **What is settled and needs no more work:** the system's software chain is proven end to end (§1 of this log), and the wearable's installed flash is backed up at `research/out/backups/wearable_baseline_COM30_20261106.bin` (md5 `0ba262cd9dd6e6f255438e79dbeb729a`). **Still open before the system is finished:** the multi-cot candidate is unflashed (§3), the node I2S watchdog is unflashed (§4), the battery/LED firmware is a plan and not code, and the SOC LED does not exist in the firmware at all. **Safety, unchanged:** this is an academic prototype, not a certified infant monitor, and must not replace supervision. |

- 2026-11-06 (**the deployed system passed its first played-cry end-to-end test: both cot nodes detected, named and buzzed on the wrist, and cleared**). The owner played a recorded cry at each node in turn — node 1 first, then node 2 — while all four boards were captured at once (`research/out/capture_20261106_181058`, 900 s, `capture_multi.py` on COM31/COM32/COM37/COM30). **The chain is whole in both directions.** *Node 1:* the node froze windows in cry mode, the hub judged them `P(no_cry)=0.0000 P(cry)=0.9961`, logged `link: node 1: ALERT — cry` and `ble_alert: a cry on cot 1 at 100%`, and the wrist answered `ALERT cot 1 type 1 conf 100% event #1` and drove GPIO7 for 2500 ms. Eight seconds later the hub published `the cry on cot 1 at 0%` and the wrist sent `ALERT cot 1 type 2`. *Node 2:* the same sequence, `ble_alert: a cry on cot 2 at 100%` → `ALERT cot 2 type 1 conf 100% event #3` → haptic → `the cry on cot 2 at 0%` → `ALERT cot 2 type 2`. **The wrist's state mask moved exactly as it should: 0x00 → 0x01 → 0x00 → 0x02 → 0x00**, one bit per cot and never two at once, which is the behaviour the single-cot policy is built on. No false alert fired at any other time in the 900 s. **One real finding:** the hub refused two windows as evidence with `the window is CLIPPED (peak 33032 of 32767)` and `peak 31399 of 32767` — the phone was closer to a microphone than the 15 cm the demo notes ask for, so the fixed-gain INMP441 saturated. That is the known cause and it is correct behaviour: a clipped window is not evidence and is refused rather than guessed at. Both nodes still announced, because the freeze rule offers the next window too. **Also fixed:** `capture_multi.py` crashed with `AttributeError: 'NoneType'` when a port was already held by an earlier capture; it now reports the port that would not open and carries on, and its log writer opens the file on first use instead of assuming it exists. | `firmware/tools/capture_multi.py`, `SYSTEM_UPDATE.md` | **The wearable's installed flash was backed up before any of this:** `%TEMP%/wearable_baseline_COM30.bin` (4,194,304 bytes, md5 `0ba262cd9dd6e6f255438e79dbeb729a`), read with esptool from COM30 — this is the exact installed image, so a rollback is now possible. A copy of the three proven cry clips (`session4_clip_0012`, `session9_clip_0005`, `session1_clip_0005`) was put on the owner's Desktop in `CryNet-test-clips/` for the test. **What has NOT been tested yet:** the multi-cot candidate (two cots crying at once) — the wearable still runs the firmware that was on it before this test, and that candidate remains uncommitted and undeployed. The node I2S watchdog candidate also remains uncommitted and undeployed.

- 2026-11-06 — Multi-cot alert work started in wearable sketch: show all active cot detections during the held alert face, follow the remaining cot when one clears, avoid CLEARED while another detection remains, and trigger haptics when another cot joins. Build passed with Arduino ESP32 core and one job (685595 bytes program, 18684 bytes globals). **Hold: not deployed.** The owner will arrange space for testing later; both the existing deployed system and the candidate still need played-cry end-to-end tests. Review and test single-cot regression, dual-cot join/clear, repeated mask changes and reconnection before any deployment. The pre-change wearable source is preserved at commit `343db20`; it is a source baseline, not a verified backup of the exact installed flash. Capture and verify the installed wearable flash before any future upload to enable an exact rollback. Existing node watchdog candidate remains uncommitted and undeployed; neither node should be reflashed as part of this hold.

Date	Change	Files touched	Notes
2026-11-06 (**I2S watchdog work in progress**)	Node capture now detects five seconds of consecutive zero-valued 16-bit samples (resetting the run on short reads and latching the threshold through the rest of each read) and attempts to restart I2S from the capture task; warmup is bounded, restart failures reboot, and status reports the restart count.	`firmware/node/node_crynet/node_crynet.ino`	**Build passed** with Arduino ESP32 core 3.3.12 for `esp32:esp32:weact_studio_esp32c3` (991629 bytes flash, 166540 bytes globals). The retry with one compiler job built and flashed a temporary fault-injection sketch to node 2 (COM37), hash verified. Software-injected zeros produced `I2S zero watchdog restarts=1` and a refilled 32000-sample window; capture: `%TEMP%/crynet-watchdog-test.log`. This confirms execution, not recovery from the original physical fault. After the test,

node 2's original 4 MB backup `%TEMP%/fullflash_37.bin` was restored at offset 0 and esptool verified its hash; the temporary sketch is no longer installed. Live verification: node 2 booted with a full window, rms 0.0272 and hub known; the hub reported both nodes online with nonzero RMS. The candidate is not deployed. This detects exact zeros only; it does not establish the cause of the earlier zero-RMS observation or guarantee recovery. Assess capture concurrency and fault handling before deploying. |
485 | 2026-11-06 (**bench recovery: both cot nodes are back online, and the zero-rms fault is proven to be transient state, not firmware**) | Node 2 (COM37) had been reading **rms 0.0000 on every mic read** — the INMP441 delivering nothing — and node 1 (COM31) was found running the INMP441 diagnostic sketch instead of the node firmware (left over from the earlier mic debugging). **Node 2's fault cleared on a hard reset, no reflash needed.** Node 1 was restored by writing its full 4 MB flash backup (`fullflash_31.bin`) back at offset 0x0 with esptool — **hash of data verified** — and it boots Stage 2b again: node id 1, ESP-NOW on channel 6, healthy rms 0.02–0.05 with the adaptive floor working. To rule firmware out, both boards' full flash images were read (`fullflash_31.bin`, `fullflash_37.bin`, 4,194,304 bytes each) and diffed 4 KB block by block: **the only difference is 0x009000–0x00E000, the NVS partition** (WiFi calibration, ESP-NOW state) — the application firmware on both boards is byte-identical. The hub then confirmed the whole chain live: **node 1 online at link −25 dBm, node 2 online at −44 dBm**, both reporting real rms (0.008–0.038), and when one burst stalled at 275/286 fragments the hub's resend path asked for the 11 missing ones and recovered them. | (no code changes — bench recovery only; no firmware was rebuilt) | *Captures: %TEMP%/cap_after.txt (hub + both nodes + wearable), queries via %TEMP%/fp2_nodeq.py.* **Open risk:** node 2's zero-rms fault has no guard — if it recurs while nobody is watching, the cot goes silently deaf. A sustained-zeros watchdog in the node firmware that reinitialises I2S would make it self-heal; not yet written. **Bench rule:** never leave the diagnostic sketch flashed on a node; restore from the full-flash backup or rebuild from this repo. |
486 | 2026-11-06 (**the hub's panel finally says AMAZING'S CRYNET, the wearable no longer freezes when the hub is unplugged, the link stops going stale after a reconnect, and the haptic is a real motor on GPIO7**) | Four things were fixed, and every one of them was then flashed and proved on the bench. **(1) The hub's name.** The device name AMAZING'S CRYNET was on the wearable but not on the hub's panel, which still said just `CRYNET` on the boot face and on the title card's headline. The boot face now stacks it the way the wrist does — `AMAZING'S` at size two over `CRYNET` at size four — and the title card's headline reads `AMAZING'S CRYNET` on one line (a hundred and ninety pixels of a three-hundred-and-twenty-pixel panel). The waveform moved up and shrank (WAVE_Y 84 to 46, WAVE_MAX 26 to 12) to make the room, and nothing collides: the wave ends at 59, the name runs 68 to 116, the strapline at 134, the boot step at 190, the bar at 206, the footer at 222. **(2) The freeze.** The wearable's whole BLE life — scan, connect, subscribe, read — ran in `loop()`, and every one of those library calls blocks its caller: the two-argument `scan->start(2, false)` waits on a semaphore for the whole two-second window, `connect()` can hold for thirty seconds, and `readValue()` waits for a peer that may never answer. With the hub unplugged the screen froze for whole seconds at a time, exactly as reported. All BLE work now runs on its own FreeRTOS task (`ble_task`, 6144-byte stack, priority 1) and `loop()` only renders and ticks the haptic. **(3) The stale link.** NimBLE's disconnect handler does not clear the service cache and does not reset `m_haveServices`, so a reconnect handed back the *previous* connection's handles: the subscriptions went nowhere and the link looked alive but was deaf. The wearable now calls `getServices()` explicitly after every connect, which forces a fresh discovery, and a failure anywhere in the setup path nulls the characteristics and disconnects instead of leaving half a link behind. **(4) The state characteristic is subscribed to, not polled.** The hub already publishes it whenever it changes and at least once a second, so the poll (and with it the forever-blocking `readValue()`) is gone; the wrist's crying mask is now never more than a second stale. Two smaller things came with them: `Serial.setTxTimeoutMs(0)`, because a CDC write with no reader on the other end could stall the radio from inside a BLE callback, and the per-advertisement `[scan]` debug print, which is removed. **(5) The haptic is the motor.** The LED stand-in is gone: the motor hangs on the BC547 low-side switch the owner drew (GPIO7 to a 2.2 k base resistor to the base, emitter to ground, motor in the collector), and the firmware now holds GPIO7 high for 250 ms pulses in step with the band blinking across the top of the screen — one buzz pattern per cry, because the hub re-announces a cry when a louder cot joins it and the caregiver must not be buzzed twice for one cry. GPIO7 is driven low before anything else happens in `setup()`, so the transistor is off from the first instruction. |
`firmware/hub_infer/components/hub_display/hub_display.c`, `firmware/wearable/wearable_crynet/wearable_crynet.ino` | **The freeze was measured, not assumed.** The pre-fix firmware was rebuilt from git and instrumented with a once-a-second heartbeat from the UI loop, and pointed at a hub name that does not exist (the hub-unplugged state): its heartbeat came every **2044 ms** — one redraw per blocking scan window, the screen frozen the rest of the time. The new firmware's heartbeat is a steady **~1000 ms** with a redraw every 100 ms. **On the bench:** the hub was flashed with `flash_hub.py` (hash verified) and the wearable with arduino-cli (hash verified); three alert cycles in a row came through complete — console `alert 1` to the hub, `ALERT cot 1 type 1 conf 100%` on the wrist, the haptic fired, `clear` came back as `type 2` — and the hub was then reset mid-link: the wearable printed `hub lost — rescanning`, found it again, connected, subscribed to alerts *and* state, and received the next alert (event #1, the hub's counter restarted with it). **Still needs human eyes:** the hub's new boot face and title card have not been looked at on the glass, and the wrist's own screens were last reviewed before the haptic changed. **Next on the bench:** solder the motor to GPIO7 as drawn (2.2 k to the base, 1N5819 with the cathode up, 10 ohm in the motor's + lead) and run the same console test — the buzzer should run for 2.5 s in step with the blinking band. **One loose end in the other repo:** `fp-2`'s copy of the wearable sketch still blinks the LED instead of driving GPIO7; the two-node copy here is the one on the bench. |
487 | 2026-11-05 (**the wearable's own screens: no project title, the device name AMAZING'S CRYNET, and the panel's capitalisation**) | The owner reviewed the wearable's glass and asked for four changes, and all four are in. **(1) The project title is gone from the wearable.** The ninety-seven-character title was running as a marquee ticker across the title card, and on a hundred-and-twenty-eight-pixel screen it crowded out the name and the id number that matter more on a wrist — the ticker, the `TITLE_LINE` string and the `draw_marquee` function were deleted. The title stays on the hub's panel, which has three hundred and twenty pixels and room to set it across three lines. **(2) The device name is on the first screen.** The boot face no longer says just `CRYNET`; it now carries **AMAZING'S CRYNET**, stacked on two lines because sixteen characters at the panel's headline size need a hundred and ninety-two pixels and the screen has a hundred and twenty-eight. The title card carries the same name on one line at the smaller size, over the full name `AMAZING AYOMIKUN ADELOWO` and the id `22/05/04/0029`, with the panel's waveform along the bottom edge. **(3) Capitalisation now follows the panel.** `ALL CLEAR`, `CLEARED` and `CRYING` are in caps like the panel's banner; `Every cot is quiet`, `Searching for the hub...`, `The hub cannot hear that cot` and the boot steps (`The display is up`, `Bluetooth is awake`, `Searching for the hub`) start with a capital, as the panel's own sentences do; the event footer reads `Event #n  Ns ago`; the crying list reads `COT 1` rather than `cot1`. **(4) The wearable was flashed and verified**: arduino-cli, hash verified on COM30, and the console shows it attaching to the hub and reading `state: 2 cot(s), mask 0x00` on its own. The nodes were left alone — their firmware has not changed. | `firmware/wearable/wearable_crynet/wearable_crynet.ino` | The hub's panel is unchanged: it keeps the full project title, which fits there. **Still needs human eyes:** the new boot splash, the new title card and the inverted alert band on the wrist have not been looked at on the glass yet. |
488 | 2026-11-05 (**flashed and verified on the bench: the hub and the wearable run the new display firmware, and the wrist now knows the hub can hear both cots**) | **The new display firmware was flashed to the hub and the wearable, and a real bug fell out of the bench.** The hub was flashed with `flash_hub.py` (three images at 0x0 / 0x8000 / 0x10000, hash verified) and the wearable with arduino-cli (hash verified). The hub came up and heard **both** cots (`node 1 ... rms 0.0080 seq 234`, `node 2 ... rms 0.0080 seq 41`, one heartbeat a second each) — so the link was never the problem. The wearable attached on its own (`hub found`, `hub connected`, `subscribed to alerts`) but read **`0 cot(s), mask 0x00`** while the hub could plainly hear two cots. **The cause:** the BLE state characteristic was published only when a cry started or stopped, so a cot *joining* the system never reached the caregiver, and the wrist read the packet it was given at boot for ever. `ble_alert_publish_state` always updates the stored value (the notification is the part that needs a subscriber), so the fix was on the publishing side: the eight-byte packet is now rebuilt every cycle and published when anything in it has moved — the cry mask, the deaf mask, or **the number of cots the hub can hear** — and on a slow beat besides, so the value a wearable reads is never more than a second old. **Verified on the hardware:** the wrist now reads `state: 2 cot(s)`, and a console test alert moved the mask `0x00 -> 0x02` (cot 2 crying) and back `0x02 -> 0x00` on `clear`, with the wearable printing the matching `ALERT cot 2 type 2` cancel. The wearable also re-reads the state characteristic every 5 s now, so its crying

list cannot go stale. **The nodes were not reflashed:** their firmware has not changed since the bench flash, and the hub is parsing both of them cleanly (`0 bad, 0 bad payload`), so the two-cot link is running the code in the repo. | `firmware/hub_infer/src/link_stage.c`, `firmware/wearable/wearable_crynet/wearable_crynet.ino`, `SYSTEM_UPDATE.md` | **Still to see with eyes:** the panel's new title card and the wearable's new splash are on the glass but nobody has looked at them yet — the colours on that panel lift their dark end, so the title card and the inverted alert band want a human look before the demo. |
489 | 2026-11-05 (**the panel and the wearable share one visual identity; the panel gained a title card, the LEVEL bar gained its notch**) | **The hub's panel was given the project's identity and the wearable's OLED was rebuilt in the same language.** The panel now boots into two splash faces before the dashboard: a boot face (the waveform, the breathing `CRYNET` wordmark, the boot steps, a filling progress bar) and a title card carrying the full project title over three lines, **AMAZING AYOMIKUN ADELOWO** and id **22/05/04/0029**; the BOOT button cycles dashboard / help / colour check as before. The LISTENING card's bar was renamed for what it is: the field is `rms` now with a second field `trigger` — the RMS the cot ships audio over — drawn as a notch on the bar (`LEVEL_BAR_FULL 0.25`, square-rooted scale), so a caregiver can see when a cot is only being listened to and when it is sending audio to the model. The wearable's SSD1306 sketch was rewritten in the panel's theme in one bit: boot face with breathing wordmark and progress bar, title card with the name, id and the full 97-char project title as a marquee ticker, a dashboard with the waveform along its bottom edge (doubles its tempo while a cot cries), a solid/hollow status pill for LINK/SCAN, an inverted alert band that blinks while the wrist buzzes, a COT-n face at text size 3 with the confidence on a bar, a DEAF face for the hub-cannot-hear case, and the crying mask re-polled every 5 s instead of read once at connect (I2C to 400 kHz so ten frames a second fit the budget). **Verified:** hub builds clean with pio (`SUCCESS`, 35 s); wearable compiles with arduino-cli. | `firmware/hub_infer/components/hub_display/hub_display.c`, `include/hub_display.h`, `firmware/hub_infer/src/link_stage.c`, `firmware/wearable/wearable_crynet/wearable_crynet.ino` | Flash the hub and the wearable to see both; the splash runs `WEAR_BOOT_MS` 2.4 s + `WEAR_TITLE_MS` 6.4 s on the wearable before the dashboard. |
490 | 2026-11-05 (**flashed on the bench: the hub, both cots, and the wearable all run the two-cot firmware; the panel's link bars are live**) | **The whole two-cot system was flashed and verified on the real boards.** The bench map was read first from each board's own boot banner (`who_is_who.py`), because the hub, the two cots and the wearable are four different targets and the wrong port would have been a wrong firmware: **COM32** the hub (CH343), **COM30** the wearable, **COM31** cot 1, **COM37** cot 2 — **node 3 was not plugged in at all**, which is the two-cot system in hardware form. The hub was flashed with `flash_hub.py` (three images at 0x0 / 0x8000 / 0x10000, hash verified) and both cots with `flash_node.py`; **both node ids survived the flash** (COM31 derived id 1, COM37's saved id 2), which is the check that matters, because the id lives in NVS. **What the hardware showed:** the hub came up and heard **both** cots — `node 1 heard for the first time`, `node 2 heard for the first time`, `node 1 is online`, `node 2 is online` — with `993 frames off the air, 993 parsed, 0 bad, 0 bad payload`; the panel driver reported `the panel is up: 320 x 240, rotation 2` and the display task started. **A full burst crossed the link and was judged:** node 2's window came back `286/286 fragments in 0.99 s`, `window crc32 0xA8FCEBD5, the node advertised 0xA8FCEBD5 -> MATCH`, and the model ran on it (`P(no_cry)=0.0508 P(cry)=0.9492`, no cry — room noise, correctly). **The new network bars are live at both ends.** On the cots, `status` prints the node's own view of the link: `link   [####] -12 dBm (hub frame 445 ms ago)` on cot 1 and `link   [####] -31 dBm` on cot 2. **The hub's view of each cot was made visible too** — the console's `status` now reports the signal strength beside each cot's other numbers (`cot 2: rms 0.0230 ... link -43 dBm`, `cot 1: ... link -23 dBm`), because the RSSI was reaching the panel but nothing could confirm it from the console. **The alert path was exercised without a sound**: `alert 2` on the hub produced `ble_alert: a cry on cot 2 at 100%` and the wearable printed `[ble] ALERT cot 2 type 1 conf 100% event #1` with the haptic standing in, then `clear` produced the matching cancel (`type 2`). **The wearable reconnected to the new hub** on its own (`hub found`, `hub connected`, `subscribed to alerts`). | `firmware/hub_infer/src/link_stage.h`, `firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/src/console.c`, `SYSTEM_UPDATE.md` | **The panel's *colours* are still unverified** — the mock PNG is a layout check, and this glass lifts its dark end, so the new rows and the network bars want eyes on the real thing. **One loose end:** the wearable reported `0 cot(s)` in its state read, because it reads the state characteristic **once**, at subscribe, and the hub had not yet heard a cot at that moment; the hub publishes the state again when it changes, but the wearable does not re-read it. It is cosmetic (the alert path is unaffected and the cots were heard a second later) and it predates this branch, but it is a real gap: **the wrist's "crying: cot N" line is only right if the wearable connected after the hub had heard its cots.** Worth fixing on the next pass — a periodic re-read of the state characteristic — but it is not a soldering-day blocker. |
491 | 2026-11-05 (the heartbeat's framing was wrong; a host test now holds it) | **The new hub heartbeat was built with the wrong frame layout, and a host test was written to hold the frame right.** `crynet_proto_build()` gave the twelve-byte payload-less layout only to a REQUEST; a HEARTBEAT fell through to the full fourteen-byte header and came out **sixteen bytes**. The parser on the far end treats a heartbeat like a request — `len != CRYNET_PROTO_REQUEST_BYTES` is a bad frame — so **every heartbeat would have been rejected as malformed**, and the node's link bar and the GPIO10 connection LED would both have sat dark on a hub that was in fact answering. The fix is in `crynet_proto_build()`: a heartbeat now shares the request's twelve-byte layout. **The second fault was subtler:** ESP-NOW is broadcast, so a cot node also receives the *other* cot's telemetry and window fragments. The node's first version recorded the RSSI of *any* frame it heard, which would have reported the loudest cot in the room rather than the hub. It now records only the hub's own frame types — a request, a heartbeat, a resend request. **A host test now holds both rules** (`research/test_heartbeat_framing.py`, new): it compiles the real `crynet_proto.c` with zig (the machine has no gcc) and checks that a heartbeat is exactly twelve bytes, that it round-trips through build → parse as type 4, that a request, a telemetry frame and a resend request are unchanged, and that the two-cot id rule hears 1 and 2 while ignoring 0, 3 and 255. **The test was checked against the bug it was written for**: reverting the one-line framing fix makes it fail with `a heartbeat is 16 bytes, not 12` and `a heartbeat parses as bad (rc -1)`. | `firmware/hub_infer/components/crynet_proto/crynet_proto.c`, `firmware/node/node_crynet/node_crynet.ino`, `research/test_heartbeat_framing.py` (new) | The hub builds again with the fix (**SUCCESS**, RAM **13.0 %**, flash **63.6 %**) and the node compiles (**75 %** flash / **50 %** RAM). Run the test with `python research/test_heartbeat_framing.py` (needs Python 3.11 for ziglang). |
492 | 2026-11-05 (**a separate two-cot repository, built from the three-cot one; the hub panel redesigned for two cards with a network bar per cot**) | **The user asked for a whole new repository for the two-node version — the three-cot repo is left exactly as it was — and for the hub's display to be redesigned: only two cots shown, the third never appearing; a network bar on the node showing its connection status; and the RMS moved out of the congested area.** `C:\Users\HP\.t3\projects\fp-two-node` is a **fresh clone** of the three-cot repo at `99bb257` (the last commit there), with its own branch `feature/two-node-display` and **no remote** — the original repository is untouched and still owns `github.com/Amazo123/CryNet`. **What changed in the new repo, and why:** **(1) Two cots, enforced, not merely displayed.** `NODE_LINK_MAX_NODES` 4 → **2** and `HUB_DISPLAY_MAX_NODES` 3 → **2**, so the third cot has no slot to occupy at all; and `node_link.c` now **rejects any frame whose node id is outside 1..2** before it can claim a slot, so a third board cannot even be heard. The hub's own state array (`link_stage.c`, `NODE_LINK_MAX_NODES` entries) follows. **(2) The RMS moved off the crowded row.** It used to share the bottom line of each card with the battery — three readings on one line, which is what the user called too congested. The card is now taller (`CARD_H` 44 → **64**, `CARD_STEP` 47 → **70**, so the second card's bottom edge lands at 222 and the footer at 230 is clear) and the readings are **one pair per row**: the RMS bar with its number on the middle row, the link and the battery on the bottom row. The banner's bare number is now labelled `LEVEL 0.012`, because a bare number in a corner reads as a clock. **(3) The network bar, on both ends.** The hub's link layer already measured the RSSI of each cot's telemetry (`slot->tel_rssi`) but **never exposed it** — `node_link_telemetry_t` gained a `tel_rssi` field, `node_link_poll_telemetry()` and `node_link_last_telemetry()` now fill it, `link_stage.c` carries it into the display state, and `hub_display.h` gained an `rssi` field on the cot card. The panel draws a **four-bar network glyph** per cot, coloured green at three bars or more and amber below that, with the dBm beside it (`LINK -58 dBm`) and `LINK --` when there is no reading. The bar scale is the usual 2.4 GHz house scale: 4 bars above −60 dBm, 3 above −70, 2 above −80, 1 above −90. **(4) The node's own view of the link.** The node now records the RSSI of the last frame the hub sent (`g_hub_rssi`) and prints a network bar of its own on the serial console (`status` → `link   [####] -58 dBm`). For that reading to be live the hub must send something regularly — **the protocol already had a HEARTBEAT frame type (type 4) that

the node has always understood, but the hub never sent one.** `node_link_send_heartbeat()` now broadcasts one **every second** (`link_stage.c`, `last_heartbeat_us`), which is the frame the node's bar is built from. **(5) The connection LED, now implementable.** `SYSTEM_WIRING.md` 5.1 specifies a GPIO10 connection LED on the node, lit only while a valid hub heartbeat has arrived within a timeout — that was explicitly not implementable before, because the hub sent no heartbeat. It is now driven from the hub's heartbeat with a **3-second timeout**, slow-blinking when the board is alive but the hub is not answering (the wiring plan's rule). **(6) Node id bounded.** The node's `id <n>` command now accepts **1..2 only** and says so, and a node booted on a MAC-derived id outside that range prints a warning naming the fix — because the hub ignores every other number, a third board left on a derived id would transmit into a hub that never answers it. **Builds:** hub **SUCCESS** (RAM **13.0 %**, flash **63.6 %**), `node_crynet` **75 %** flash / **50 %** RAM, `wearable_crynet` unchanged. The panel was rendered to a mock PNG (`research/out/two_node_display_mock.png`) before flashing. | `firmware/hub_infer/components/hub_display/hub_display.c`, `components/hub_display/include/hub_display.h`, `components/node_link/node_link.c`, `components/node_link/include/node_link.h`, `src/link_stage.c`, `firmware/node/node_crynet/node_crynet.ino`, `SYSTEM_UPDATE.md` | **The user's other repo is not touched** — this is a separate clone at a separate path with its own branch and no remote, so the three-cot system stays exactly as it was. The vendored `esp-tflite-micro` and `esp-nn` trees are gitignored upstream and had to be copied in by hand for the build to resolve the component. The hub's `DEAF` warning is deliberately kept prominent even when the link's RSSI is strong: **online is not the same as a working audio path** — a cot whose windows never arrive is transmitting fine and still not being watched. |
| 493 | 2026-11-05 (**the system ships as two cots** — the pending firmware is build-verified on all three targets and committed) | **The user's decision: node 3 comes off the bench, and the system runs as hub + two cots + wearable.** Nothing in the code had to change for this: the hub's link layer is fully dynamic (`NODE_LINK_MAX_NODES 4` slots claimed at join, no node list anywhere), so an unpowered node 3 simply never joins. The pending working-tree firmware was verified to build on every target: the hub with the standalone PlatformIO (`C:\Users\HP\.platformio\penv\Scripts\platformio.exe run` — **SUCCESS**, RAM 13.1 %, flash 63.5 %), `node_crynet` (**75 %** flash, **50 %** RAM) and `wearable_crynet` (**51 %** flash, **5 %** RAM) with `arduino-cli` at FQBN `esp32:esp32:weact_studio_esp32c3` (the board definition lives in the esp32 3.3.12 package's `boards.txt`; `Serial` there is HWCDC, which is why `setTxTimeoutMs` compiles — the generic `esp32:esp32:esp32c3` FQBN builds `Serial` as a UART `HardwareSerial` without that method and fails; it is a FQBN mismatch, not a sketch bug). The whole pending change set now rides one commit on `fix/link-resend-bitmap-verification`: the deaf-cot backoff (`BURST_FAIL_LIMIT 3`, `BURST_BACKOFF_MS 60 s`, `link_stage_deaf_mask`, the wearable's type-3 DEAF buzz, the panel's `deaf` flag), the bitmap print at both ends, the promiscuous-callback fixes, and the node's `g_tx_failed` counter (known vacuous for broadcast; kept for the day the link goes unicast). | `SYSTEM_UPDATE.md`, `firmware/hub_infer/*` (hub build verified), `firmware/node/node_crynet/node_crynet.ino`, `firmware/wearable/wearable_crynet/wearable_crynet.ino` | **What the user does to ship:** flash the hub from this branch (`platformio run -t upload`), reflash the cots and the wearable from the sketchbook copies that `install_sketches.py` just synced, power node 3 off, and solder. The deaf-cot path has never been exercised on hardware — first bench session with two cots, watch for a `DEAF` line only if a link actually fails; with node 3 gone it may never fire, which is the point. `research/build_notebook.py`'s pending Kaggle-work changes and the untracked `research/out/` captures stay out of this commit. Merge to `main` is the user's call after a bench pass. |
| 494 | 2026-11-05 (the resend bitmap: **the two ends agree — the "274 vs 12" was two different rounds, and node 3 now completes every burst**) | **The open bitmap question is settled, and it was never a protocol fault.** Both ends of the wire were instrumented to print the resend bitmap as it goes over the air — the hub in `node_link.c` (`resend request for node %u: %d fragments asked for, bitmap %02X %02X %02X %02X %02X %02X`) and the node in `node_crynet.ino` (`resend request: seq %u, %lu of %u fragments asked for, bitmap %02X %02X %02X %02X`, a popcount of the received bitmap). **The cots were replugged and the node-3 board was reflashed with the instrumented firmware (`COM38`, MAC `90:DA:72:85:E0:4C`); the hub was captured alongside it** (`research/out/node3_diag3/`). In the very first exchange of that capture the two printed the **same count and the same bytes**: the hub `resend request for node 3: 10 fragments asked for, bitmap 00 00 00 00 00 00`, the node `resend request: seq 4, 10 of 286 fragments asked for, bitmap 00 00 00 00`. The node answered `resent 10 fragments of seq 4` and the burst finished `286/286 fragments in 1.11 s`. **The earlier "hub says 274, node says 12" was a mis-pairing of two different resend rounds**: the hub's 274 was its *first* round (it had received only 12 fragments by then), while the node's "12" was its *last* round (by then the hub was asking for the 12 that were still missing). The node's radio is saturated sending fragments and only catches the hub's later rounds; the two logs were being read as if they described one exchange. **The protocol is sound.** Node 3's real problem is loss on the air, not disagreement: in the failing capture (`research/out/node3_diag2/`) the hub received **only 12 of 286 fragments on the first round, then 28, 52, 60, 73, 86** across five rounds, and the node's own log shows `served 286 fragments of 286 in 745 ms` — the cot transmitted everything, the hub received a fraction. The decisive instrument is the hub's own built-in promiscuous diagnostic (`CRYNET_LINK_PROMISCUOUS`, `node_link.c`): it counts the frames the radio hears (`s_promisc_total`) against those carrying the CryNet magic (`s_promisc_magic`), and `link_stage.c` prints `air: %u frames seen raw, %u carrying our magic` next to `frames off the air, %u parsed`. **If the raw count greatly exceeds the parsed count, the fragments reach the hub's radio and the ESP-NOW layer is dropping them; if not, they never arrive.** The build flag `-DCRYNET_LINK_PROMISCUOUS=1` is in `platformio.ini`; the callback is now registered *before* promiscuous mode is enabled (the order the sniffer proved is required, or the radio hands over nothing), and it reads the frame length from `rx_ctrl.sig_len` instead of guessing 400 bytes. | `firmware/hub_infer/components/node_link/node_link.c`, `firmware/hub_infer/platformio.ini`, `firmware/node/node_crynet/node_crynet.ino` | The node-3 bitmap mystery is closed as **not a bug**. Two small fixes went in with it: the promiscuous callback now uses the real frame length (`pkt->rx_ctrl.sig_len`) and the payload pointer (`pkt->payload`) rather than walking 400 raw bytes, and the callback is registered *before* `esp_wifi_set_promiscuous(true)`. The node's counters (`lost 0 / acked N`) remain vacuous for broadcast frames and should not be read as evidence of anything. |
| 495 | 2026-11-05 (the resend bitmap: the hub asks for 274 fragments and the node reads twelve — and the cots dropped off the USB bus) | **The node-3 failure is narrowed to one specific disagreement between the two ends of the resend request, and a passive sniffer is built to settle it.** From the node's own log (`research/out/node3_diag2/node3_COM38.log`): `request: seq 12 first 0 count 286` → **`served 286 fragments of 286 in 745 ms`** → `resend request: seq 12` → **`resent 12 fragments of seq 12`** → `resend request: seq 12` → `resent 1 fragments of seq 12`. So the node **did send all 286 fragments, fast**; it is not the cot that is slow. The hub's side of the same exchange: `the burst stalled with 12 of 286 fragments (12 frames arrived); asking for the 274 that are missing`, and its arrival pattern `.....#.#.#......` shows the arrived fragments are the scattered ones. **The hub says it asked for 274 fragments and the node counted twelve bits set in the very same 36-byte bitmap.** The source was checked end to end and is self-consistent: the builder (`node_link.c`, `node_link_request_missing`) sets the bit for every fragment whose bit in `slot->bitmap` is clear — the *missing* ones — and the count it logs is the popcount of the bitmap it sends; the node reads the payload at offset 14 (the header size on both sides), verifies the payload CRC-16 over exactly those bytes, and resends every bit that is set. **Both CRCs are the same CCITT-FALSE (0xFFFF, 0x1021) and both frame layouts are identical, so the two ends cannot legitimately disagree — which means one of the two deployed binaries is not the source, or the frame on the air is not what the hub thinks it sent.** The hub's counters rule out corruption: `41,625 frames off the air, 41,625 parsed, 0 bad, 0 bad payload` — nothing arrived damaged; the fragments simply never arrived. **Two instruments are now in place to settle it in one capture:** (a) the node prints the popcount and the first four bytes of the bitmap it received, and the hub prints the popcount and the first six bytes of the bitmap it sends, so the two logs can be compared byte for byte; (b) a **passive promiscuous sniffer** (`firmware/node/sniffer_crynet/sniffer_crynet.ino`, new) listens on channel 6 and **decodes the CryNet protocol off the air** — it counts frames by type and prints the bitmap of every resend request exactly as it crossed the air. If the air shows 274 bits set, the node is misreading it; if the air shows twelve, the hub is building it wrongly. **Getting the sniffer to listen took four tries, and the reasons are worth writing down:** `esp_wifi_set_promiscuous(true)` succeeds and the radio then hears nothing unless `esp_wifi_set_promiscuous_rx_cb()` is called first (the callback has to be given to the driver); the filter must be `WIFI_PROMIS_FILTER_MASK_ALL` or beacons are dropped; the core's `WiFi.channel()` is the *scan* result and the radio's real channel is `esp_wifi_get_channel()`; and `arduino-cli upload` will happily flash a **stale cached binary** when the sketch fails to compile, so the build must be checked before the upload is trusted (this

happened once, and the sniffer ran an old build for two captures). **The three cot boards have now dropped off the USB bus** (COM31, COM37, COM38 are gone; only the hub and the wearable remain) — they are USB-powered, so they are off, and the flakiness of that bus is itself a candidate for node 3's intermittent failures: a board that brown-outs mid-burst sends a burst that stops halfway. **The user has to replug the cots**; the bench cannot be measured without them. | `firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/components/node_link/node_link.c`, `firmware/node/sniffer_crynet/sniffer_crynet.ino` (new), `research/tools/compare_air.py` (new) | The sniffer runs on the **wearable's board (COM30)** for the duration, so all three cots keep their normal firmware and their normal traffic; `flash_node.py --port COM30 --sketch wearable` puts the wearable back (it only knows `node` and `wearable`, so the sniffer was flashed with `arduino-cli upload --input-dir`). Node 2 (COM38) was already restored to `node_crynet`. The hub build with the bitmap print is running. **Also still open, unchanged:** the uncommitted deaf-cot work (hub-side), the vacuous `g_tx_failed` counter (broadcast frames are never acknowledged, so it can never fire), and the Kaggle notebook work. |

496 | 2026-11-05 (the node-3 link diagnosis — the node's clean sheet is meaningless, and a sniffer is on the bench) | **Node 3's burst failure is diagnosed as far as software can take it, and the next measurement is a third point of view: what is actually on the air.** Where the previous session stopped: node 3's bursts crawl (12→28→52→60→73→86 of 286 over 15 s, then timeout) while nodes 1 and 2 complete. Two conclusions from the existing logs (`research/out/node3_diag2/`, `hub_node3_verdict.log`): **(1) signal strength is ruled out** — node 2 completes 286/286 at −58 dBm while node 3 crawls at −50 dBm (10 dB *stronger*), and node 1 completes at −22 dBm; **(2) node 3's "clean sheet" is vacuous** — `radio_queue()` sends every fragment **broadcast** (unicast once wedged the send path, so it was abandoned, comment at `node_crynet.ino` line ~937), and a broadcast ESP-NOW frame is never MAC-acknowledged, so `frames sent 3336, lost 0, the driver could not carry 0` says nothing at all about delivery — the `g_tx_failed` counter added last session can never increment for broadcast. The smoking gun from last session (the hub's RESEND arriving 0.9 s into a ~1.15 s burst) was re-checked: the node queues a mid-burst resend and serves it *after* the burst pump finishes (`g_resend_pending`, `node_crynet.ino:1705`), so it does not corrupt the burst — the fragments are genuinely dying between node 3's radio and the hub's application. **The decisive experiment: a passive promiscuous sniffer** (`firmware/node/sniffer_crynet/sniffer_crynet.ino`, new) — listens on channel 6, counts DATA frames by source MAC, split small (telemetry/requests, <150 B on air) vs large (window fragments, ≥150 B), with RSSI and node 3's big-frame RSSI spread; it transmits nothing. Compiled clean (71 % flash, 10 % RAM) and flashed onto **node 2's board (COM38)** with `arduino-cli upload` (`flash_node.py` only knows `node`/`wearable`, so the raw CLI was used); `flash_node.py --port COM38 --sketch node` puts the node firmware back afterwards. Port map re-verified with `who_is_who.py`: COM32 hub, COM30 wearable, **COM31 node 3**, COM37 node 1, COM38 node 2. The capture (`research/out/node3_sniff/`, hub + node 3 + sniffer, 210 s, `info` sent to node 3's console at t+15 s) reads the air during node 3's bursts: **if the air carries the fragments and the hub still received a trickle, the hub's receive path is at fault; if the air itself only carries a trickle, node 3's transmit is the problem (antenna, power, or an interference source beside that one board).** | `firmware/node/sniffer_crynet/sniffer_crynet.ino` (new), `research/out/node3_sniff/` | The previous session's uncommitted hub work (deaf-cot detection: `BURST_FAIL_LIMIT 3`, `BURST_BACKOFF_MS 60000`, `ble_alert_publish_deaf`, panel `deaf` flag, wearable state bit) is still uncommitted and **not yet flashed to the hub** — the verdict log shows no `DEAF` lines, so the hub on the bench predates it. It stays pending until the node-3 link question is settled. Also still uncommitted: the node's `g_tx_failed` counter (now known to be vacuous for broadcast — either keep it for the day the link goes unicast, or drop it) and `research/build_notebook.py` (logged 2026-11-05, the self-contained notebook). |

497 | 2026-11-05 (the notebook is now self-contained: it fetches the speech corpus from a URL) | **Everything the notebook needs is now fetched by URL, so the only thing left to do is import it and press Run All.** The loud-speech corpus is published as a **public GitHub release asset** (`Amazo123/CryNet`, release `speech-corpus-v1`, `speech__fsd50k_speech.zip`, 792,972,265 bytes) and the notebook downloads it like any other corpus — with a **SHA-256 check** (`eeea2932…`) so a truncated or corrupted archive is caught at the fetch instead of showing up much later as a quietly smaller corpus. The release notes carry the attribution (FSD50K is CC-BY 4.0, Mozilla Common Voice is CC0/MIT). The corpus is **no longer optional**: if it cannot be fetched the run *stops*, because the one thing worse than not running is training on the old mix and thinking the speech problem was solved. The publishing script (`research/loudspeech/publish_corpus.py`) takes the credential from the machine's own git credential store, so no token is written to a file or a command line, and it verifies the asset as an anonymous visitor would. | `research/loudspeech/publish_corpus.py`, `research/build_notebook.py`, `notebooks/crynet_training.ipynb`, `notebooks/crynet_training_kaggle.py` | **What the user does now:** import `notebooks/crynet_training.ipynb` (or paste `notebooks/crynet_training_kaggle.py` into one cell), set Accelerator = GPU T4 and Internet = On, press Run All, then download `crynet_artifacts.zip` from the Output tab. Nothing to attach, nothing to upload. |

498 | 2026-11-04 (the retrain moves to Kaggle — the notebook is ready for the GPU) | **The retrain moves to Kaggle, at the user's suggestion.** The local CPU run is healthy but slow: **259 s per epoch** (685 steps at 378 ms/step, all 8 cores at 99 %), so 35 epochs plus the mining round is **~3 hours**. The same notebook on a Kaggle T4 is minutes of training, and Kaggle is where the authoritative model was trained before. There are no Kaggle credentials on this machine, so the notebook is prepared for the user to run: (a) **the loud-speech corpus is now found automatically** — `acquire()` falls back to looking for `speech__fsd50k_speech.zip` under `/kaggle/input`, so the user attaches the 757 MB zip as a Kaggle dataset and nothing else needs configuring; (b) **the notebook trains only `conv_cnn`** — the variant the hub runs — because on Kaggle there is nowhere to set an environment variable, and the other two variants are not deployed; (c) **a final cell packages the artifacts** into `crynet_artifacts.zip` in the working directory, with the file list printed, so the output is one download rather than a dozen files; (d) the header now says exactly what to attach, which accelerator, and where the zip comes from. A **single-file version is exported as well** (`notebooks/crynet_training_kaggle.py`, 2,186 lines) for the case where pasting one cell into a Kaggle notebook is easier than importing the `.ipynb`. | `research/build_notebook.py`, `notebooks/crynet_training.ipynb`, `research/export_single_cell.py`, `notebooks/crynet_training_kaggle.py` | **What the user does:** upload `research/_cache/speech__fsd50k_speech.zip` (792,972,265 bytes, 6,000 clips) as a Kaggle dataset, create a notebook from `notebooks/crynet_training.ipynb` (or paste `notebooks/crynet_training_kaggle.py` into one cell), attach that dataset, set Accelerator = GPU T4 and Internet = On, *Run All*, then download `crynet_artifacts.zip` from the Output tab. **Update (2026-11-05): the manual upload is no longer needed — the corpus is fetched from a URL.** The local run was left as a fallback and **died after 6 epochs on the machine's memory ceiling again** (`ArrayMemoryError: Unable to allocate 225. MiB for an array with shape (11386, 81, 64)`, during the validation pass in cell 15): the box has no page file, and the training process had ~4 GB resident with only ~4.7 GB free. It is not worth fixing — Kaggle has the memory to spare, and the split-assembly fix already took the peak from ~4.5 GB to ~2.2 GB. Nothing is deployed until the gate passes. |

499 | 2026-11-04 (the retrain's memory ceiling found and fixed — it is training again) | **The first full run with the speech corpus died on memory, not on the model.** It reached cell 12 and threw `numpy._core._exceptions._ArrayMemoryError: Unable to allocate 873. MiB for an array with shape (88324, 81, 64)`. The cause is the machine, not the recipe: **16 GB of RAM and no page file** (Windows reports virtual memory == physical, so an allocation that does not fit is a hard failure, not a slowdown). The split-assembly cell built the ~900 MB training matrix by concatenating copies, so **four live copies of it existed at once** (the parts, the first concatenate, the second concatenate, and the de-duplicated copy) on top of the ~1.2 GB of source matrices, and the run ran out. **The split is now assembled in place**: one preallocated array, filled a chunk at a time (2,048 rows), with every source matrix released the moment it falls out of use — the augmented views go as soon as the training rows are filled, the ICD matrices as soon as the validation and test folds are cut. The peak drops from ~4.5 GB to ~2.2 GB, and the order of the parts is unchanged so the de-duplication keeps exactly the same windows. Two other cuts: the **mining round no longer duplicates the training matrix** (it used to build an ~1.9 GB `X_mine`; the hard examples are now *mixed in* with `sample_from_datasets` at 1.4 %, which reproduces the old "each hard row appears twice" share without the copy — verified: 1.57 % of batches came from the hard set in a standalone test), and the **tf.data shuffle buffer is 8,192 windows instead of 32,768** (each buffered element is a whole 81x64 window, so the old buffer alone held ~340 MB). The notebook now prints its resident set size at the split and after the datasets are built, so the next run says how much headroom it has instead of dying. | `research/build_notebook.py`, `notebooks/crynet_training.ipynb` | The retrain was relaunched at the same settings (`conv_cnn`, 35 epochs, 1 mining round, full mode, the 6,000-clip speech corpus) and is running in the

background. **The fixed run split 87,737 training windows (35,380 cry / 52,357 non-cry) and is training at 4.0 GB resident with 4.7 GB free — the ceiling is gone.** The RSS print itself returned 0 (the process pseudo-handle was truncated to 32 bits), so it is fixed for the next run. Nothing is deployed until the gate passes. |
500 | 2026-11-04 (the retrain is running: the speech corpus is built and the notebook is training with it) | **The corpus is built and the retrain is running.** The training corpus is **6,000 speech clips in one zip** (`research/_cache/speech__fsd50k_speech.zip`, 757 MB): **3,500 FSD50K clips from the dev partition** — the nursery-relevant classes, weighted toward the ones that matter (`Laughter` 678, `Human_group_actions` 342, `Speech` 297, `Singing` 289, `Female_speech_and_woman_speaking` 279, `Crowd` 230, `Male_speech_and_man_speaking` 184, **`Child_speech_and_kid_speaking` 110**, `Giggle` 71, `Chatter` 46, `Conversation` 37) — and **2,500 Common Voice clips** from the training half, for the breadth of real speech across speakers and accents. **Median RMS 0.0723, and 4,760 of the 6,000 clips are above the hub's 0.030 trigger**, so the model is shown speech at the level it would actually be asked about. The file names carry the class first (`Speech__fsd12345.wav`), which is what the notebook turns into a subclass, so the run's own report can say which kind of speech it learned to refuse. **The notebook is training locally, in full mode, on the same architecture the hub runs** (`conv_cnn`, 122,770 parameters, 35 epochs, one hard-example mining round), with the corpus split 75 % training / 25 % held out for the cross-corpus score. | `research/_cache/speech__fsd50k_speech.zip` (not in the repo — it is 757 MB), `research/loudspeech/*` | **The gate is what decides, not the training run's own numbers.** A candidate is only deployed if it holds every line: the speech announcement rate down by at least ten times and **not one** announcement on a child's voice; music no worse than 0.21 %; ESC-50 no worse than 1.18 % with the 40 `crying_baby` clips still at 36; no false alarm in any of the room's confusion takes; and the per-window recall on the cry at 15 cm within reach of the deployed model's 86.4 %. |
501 | 2026-11-04 (the measurement that settles the training question: the deployed model announces on children talking) | **The deployed model was scored over real speech — 3,842 speech clips loud enough to reach the classifier — and it announced on 60 of them. The retrain is not optional; this is the missing training.** The corpora came from the internet as the user asked (FSD50K 16 kHz's speech classes and Common Voice 16 kHz, both from partitions and halves the model has never seen). **The false-alarm map of the deployed INT8 model:** 116 alarms at the 0.95 threshold and **60 announcements at the hub's saturation 0.99**, out of 3,842 speech clips that reached the model — **1.56 % of loud speech buzzes the caregiver's wrist.** Broken down by class, and it is worst exactly where a nursery is worst: **children's speech announces 16.1 % of the time (5 of 31)**, **singing 10.2 %**, **giggles 10.0 %**, **laughter 9.5 %**, a human voice 4.5 %, chatter 2.3 %, and read speech from thousands of speakers 0.68 %. **A baby monitor that buzzes when the baby giggles is a nuisance alarm, and the model does it more often than it rejects speech.** Against the other classes for scale: music is 0.21 % and ESC-50's non-cry 1.18 % after the last retrain — **speech is now the worst class by a wide margin**, because the corpus's 4,963 VoxCeleb speech clips are quiet, adult and clean, and a nursery is loud, and has children in it. The one measured conversation window (0.9727) was the visible tip of this: the model puts ordinary voices near the cry end of its range and only the saturation rule was standing between a conversation and a 3 a.m. buzz. **So the round the user asked about is now evidence-based rather than speculative.** The cry-adjacent FSD50K classes (`Crying_and_sobbing`, `Screaming`, `Shout` — 118 clips) were measured but deliberately **not** trained on: an adult crying is not a baby's cry, but it is close enough that the right answer is arguable, so they are reported on their own (11 alarms, max P(cry) 0.9961). | `research/out/eval_speech.json`, `research/eval_speech.py` | The next step is the retrain: the same measurement, run over the new model, is the gate. A model that fixes speech and loses the cry is no use, so recall on the in-domain cry sessions and the music/ESC-50 maps have to hold. |
502 | 2026-11-04 (the loud speech is sourced online: the tooling is built, the corpora are downloading) | **The speech is coming from the internet, as the user asked, and the whole pipeline for it is built while the corpora download.** The user said it plainly: *"you can source such online and do the training if you need to, as I don't have people around to do any such recording."* **Two corpora, chosen for what the model has never seen:** (1) **FSD50K at 16 kHz** (the `quinnlue/FSD50K-16k` mirror) — real-world recordings with class labels, and the classes that matter here are `Speech`, **`Child_speech_and_kid_speaking`**, `Conversation`, `Chatter`, `Crowd`, `Human_voice`, `Male_speech_and_man_speaking`, `Female_speech_and_woman_speaking`, `Laughter`, `Giggle`, `Singing` — a nursery has all of them and a child's voice is the closest thing to a baby's cry that a real room contains; (2) **Common Voice at 16 kHz** (the `DTU54DL/common-voice-test16k` mirror, MIT) — 16,354 real read sentences from thousands of speakers with accents, recorded close to the microphone in quiet rooms, which is the closest thing to a person talking in the next room. **The splits are honest and enforced in code** (`research/loudspeech/speech_common.py`): FSD50K's official split is **dev** (train) and **eval** (never trained on) — the mirror's file names say `train-*` but the `source_partition` field carries the truth — and Common Voice is halved by a hash of the clip id, with one half for training and one for measurement, so **no clip is ever both trained on and measured on**. `Crying_and_sobbing`, `Screaming` and `Shout` are deliberately **not** trained on: an adult crying is not a baby's cry, but it is close enough that the right answer is arguable, so they are measured and reported on their own. **New tooling:** `research/eval_speech.py` (the deployed INT8 model over real speech, through the notebook's own front end, reporting P(cry) and the false-alarm rate at both the deployed threshold 0.95 and the hub's saturation 0.99), `research/loudspeech/build_speech_zip.py` (the training corpus as one zip of 16 kHz WAVs with the class in the file name), `research/loudspeech/get_shards.py` (the fetch), and `research/build_notebook.py` patched so the notebook **takes the speech corpus as an optional negative source** — 75 % into training, 25 % held out for the cross-corpus score — and prints `[corpus not built - skipped]` when the zip is absent, so a Kaggle run (where the zip cannot be attached, this machine has no Hugging Face token) still trains and still says so. The notebook was regenerated from the builder (never hand-edited) and **a fast smoke run of it passes** with the corpus absent. | `research/eval_speech.py`, `research/loudspeech/*`, `research/build_notebook.py`, `notebooks/crynet_training.ipynb` | The measurement comes first, exactly as the plan says: the deployed model is scored over the untouched speech **before** anything is retrained, and the retrain happens only if that number says the margin is thin. |
503 | 2026-11-04 (two items closed or redirected: cot 2's microphone is fine, the loud speech is to be sourced online) | **Cot 2's microphone comes off the open list — it was never broken — and the loud-speech item is redirected from "record a conversation in the room" to "source real speech online".** The user asked both questions straight: *"what other training round — don't we have what we need?"* and *"what is wrong with cot 2's microphone?"*, and the answers are in the numbers. **Cot 2, tonight's run:** window `rms 0.1058, mean −316, peak 16918, crest 4.9` → P(cry) = **0.9961** → `ALERT cot 2` on the wrist at 100 %. The earlier scare came from `demo_run3` at 815.15 s, where the window read `peak 32470` of 32767 — **the INMP441 is fixed-gain and saturates when the phone is ~10 cm from it**, which is a *distance* problem, not a hardware one, and the hub now refuses a clipped window loudly rather than trusting it. The "DC offset of −640" is **not a fault**: the part has a small offset by nature and the model's per-window CMVN removes the mean before it judges anything (node 2's means swing from −640 to +879 with the room). At the kit's 15–20 cm, node 2 was clean. **The loud-speech item:** the one piece of evidence is a single window — a conversation in the room scored **0.9727** (node 3, rms 0.058, crest 3.5) — which sits *below* the saturation threshold (0.9900) and therefore does not announce today; **the rule catches it, not the model**. Put in proportion: the confusion session measured **0 false alarms in 308 windows** of speech, music and film (a loud film and loud music included, where the trigger fired on 46 % and 84 % of records and the model refused every one), and the 25-minute watch gave 153 windows, 0 alerts, not one window even reaching 0.99. So the model is not walking into false alarms on anything measured; what is missing is **margin**. **The user has nobody around to record a conversation**, so the plan is now: **source real conversation audio online** (public speech corpora, loud clips selected by level), **score it with the deployed model first** (the same way `eval_music.py` and `eval_esc50.py` measure the false-alarm map), and **only run a training round if that measurement says the margin is thin**. The cost of a round is named too: the last retrain already cost recall (0.867 → 0.800, 0.724 → 0.592 per-window), and cot 1's 0.8516 in `demo_run3` shows **recall on real in-room cries is the model's weaker side**, not speech false alarms. | `SYSTEM_UPDATE.md` | The demo does not depend on the retrain. The work now is: measure the deployed model against sourced loud speech, and decide from that number. |
504 | 2026-11-04 (the flapping fix, verified on the bench) | **One cry is one buzz, and the name holds — the fix works.** The user played the cry at node 1, then node 2, then node 3, and the wrist got **exactly three alerts in the whole run, one per cry, each naming the right cot**: `ALERT cot 1` at 53.38 s (P(cry) = **0.9961**), `ALERT cot 2` at 84.63 s (0.9961), `ALERT cot 3` at 109.50 s (0.9961) — each with a haptic event, each cleared 8 s later (61.38 / 92.64 / 124.41 s). **The name never changed hands mid-cry.** Node

3's cry was judged a second time at 116.41 s (P(cry) = 0.9961 again) and **did not buzz again and did not move the name** — it held on cot 3 until the clear. Before the fix, that same bench produced **twelve name changes in eleven seconds for a single cry** (`cot 1, cot 3, cot 1, cot 3…`). The whole run holds six `ble_alert` lines and no others. | `research/out/demo_run4/` (the capture), `SYSTEM_UPDATE.md` | **The flapping fault is closed.** The remaining open items are unchanged and are the bench/hardware ones: cot 2's microphone clips on ordinary room sound and reads a DC offset of −640, and cot 1 needs a little more level than the phone at that distance gave it. |
505 | 2026-11-04 (the flapping fix — one cry is one name) | **The wrist was told twelve names in eleven seconds for one physical cry, and the hub now holds the name.** The 21:16 run in `demo_run3` showed the fault: the old rule let a challenger take the alert whenever its *live telemetry reading* was 1.5× the named cot's — but two microphones a hand's width apart cross over every few hundred milliseconds, so the name changed hands (`cot 1, cot 3, cot 1, cot 3…`) while one cry was in the room. Three changes on the hub (branch `feature/stop-capped-freeze`, commits `215e0ae` + `8944488`): **(1)** a challenger is judged on the **window it froze** (`frozen_rms`), a number that holds for the seconds a window lasts, not the live reading; **(2)** it must stay clearly louder for **`COT_SWITCH_DWELL_MS` = 1500 ms** before the name moves — the named cot keeps the alert right up to that point, so one cry is one buzz; **(3)** while an alert is up the **display names the same cot the wrist does** (a panel that says cot 3 while the bracelet says cot 1 is worse than no panel). The dwell resets when the name moves or the cry ends. **Built, flashed to the hub (21:4x), and the hub came back up with all three cots live** (500 frames, 0 bad, BLE subscribed). **Not yet verified on the bench:** the user must replay a cry at the cots and confirm the wrist buzzes *once*, and the name stays. | `firmware/hub_infer/src/link_stage.c`, `research/out/demo_run3/` (the 21:16 evidence), `research/toolchain/flash_hub_cotswitch.log` | This was the fix the previous session died mid-build on (400 context limit at 21:25); the build finished before the session fell, and this session flashed the binary it left. |
506 | 2026-11-04 (the demonstration, first successful cry — the chain closed end to end) | **The chain now works end to end, in front of the user, with the wearable attached: phone → microphone → node freeze → ESP-NOW burst → CRC-32 match → INT8 model → hub names the cot → BLE → wrist.** The user played `session9_clip_0005.wav` at all three cots in turn. **Cot 3 announced:** window rms 0.041314, peak 7089, crest 5.2 → **P(cry) = 0.9961** → `node 3: ALERT — cry` at 364.28 s → `ble_alert: a cry on cot 3 at 100%` → the wearable printed `[ble] ALERT cot 3 type 1 conf 100% event #1` and `[haptic] alert 1 — motor not present, LED stands in` (the motor is not soldered yet; the LED stands in). Every burst in the run reached 286/286 with a CRC-32 MATCH — node 1 at −23 dBm, node 2 at −46 dBm, node 3 at −41 dBm — and the resend rounds recovered the shortfalls (19+2 and 20+3 fragments). **Cot 1 came close but did not announce**: its window (rms 0.05365, peak 13280, crest 7.6 — clean, no clip) scored **0.8516**, and the rule is that a cry is announced on the saturation (0.9900) and not before, so a 0.8516 is a maybe and a maybe announces nothing. **Cot 2's windows were flat and DC-heavy** (rms 0.112, peak 7681 but **mean −640 and −1031**, crest 2.1–2.4) and scored 0.0000: node 2's microphone is the one that clips on ordinary room sound and still reads a large offset — it is the cot that wants looking at. **What the fixes bought, measured:** node 1's window had sat advertised for seven minutes and un-pulled in the 19:26 run; now every cot's window is pulled within a second or two of the freeze. The freeze storm is over (node 1: 289 freezes per 7 minutes → 11), node 3 completes bursts it never once completed before, and no resend is refused anywhere. | `research/out/demo_run3/` (the capture), `SYSTEM_UPDATE.md` | **The remaining work is the bench, not the firmware:** cot 1 needs a little more level (0.8516 is close), and cot 2's microphone clips on ordinary sound and reads a DC offset of −640 — that is a hardware question for the day the boards are soldered. |
507 | 2026-11-04 (the first live play on the new firmware — the link is fixed, the source is not) | **The play at 20:50 proved the firmware fixes and isolated the last fault to the acoustics.** All three cots froze a **cry-grade** window (0.062, 0.071, 0.060 rms) — the hub **pulled all three**, which never happened before (node 1's window had sat un-pulled for seven minutes in the 19:26 run) — and **every burst finished 286/286 with a CRC-32 MATCH**, node 2 at −47 dBm and node 3 at −40 dBm, with the resend rounds recovering 19+2 and 20+3 lost fragments. Node 3, which had **not completed a single burst** in the whole of the 19:26 run (four resends refused, "no frozen window for it"), now completes every one. **The windows were then all refused as CLIPPED** (peaks 31358, 32706, 32255 of 32767) — the microphone had saturated — and the model scored them 0.0000, which is the correct verdict on a flat-topped cry. **One window was clean** (node 1, rms 0.1449, peak 21039, no clip) and the model *still* refused it (0.0000) at a **crest of 4.4** — a cry through a phone speaker and a room has a crest of 8–16, and 4.4 says the *source* was distorted: the phone's own speaker was overdriven at that volume, which saturates the microphone and flattens the waveform at the same time. **Since the CRC matched, the radio is not the suspect: the hub's window is bit-for-bit the node's window, so the fault is what the microphone heard.** The next play therefore has to be quieter at the source — phone volume down a couple of steps, 15–20 cm away — and the window's own rms, peak and crest in the hub's log are the numbers to steer by. | `research/out/demo_run2/` (the capture), `SYSTEM_UPDATE.md` | The chain is proven end to end for the first time: freeze → advertise → pull → 286/286 → CRC MATCH → model. What is left is not a firmware fault. |
508 | 2026-11-04 (the monitored bench run — three faults found, fixed, and the reason the demo announced nothing) | **The live run of 19:26–19:33 (`research/out/demo_run/`, five ports captured) produced **zero ALERTs**, and the capture explains every one of them.** The clip itself is fine: scored on the host with the deployed INT8 model, `session9_clip_0005` and `session1_clip_0005` read **0.9961 on every two-second window** (`research/score_demo_clips.py`, new). The fault was threefold. **(1) The node's three-second resend hold was inverted and cancelled itself the moment it started.** `g_frozen_hold_ms` is a deadline (`millis() + FROZEN_HOLD_MS`), so `now - deadline` is negative — past 0x80000000 as unsigned — for the whole hold; the test released the window on the very next reading and never released it afterwards. Measured: node 3 was refused four resends with "no frozen window for it" and **not one of its bursts finished**, while nodes 1 and 2 finished every one. Now a signed difference, and the window is dropped outright when the hold ends. **(2) The node could overwrite an unfetched window with ordinary room sound.** Node 1 froze a **0.1769 rms window — a genuine cry** — at 19:27:35, and replaced it with a **0.0104 rms window of room sound** three seconds later, because the hub had spent its airtime on node 2's window and had not come back; the freeze gate (`!g_frozen_served`) only blocks a re-freeze *after* the hub has taken one, so "the hub has not taken it yet" was exactly the condition that allowed the replacement. Now an advertised window may only be replaced by a much louder event (the existing three-times rule, extended to unfetched windows). **(3) The hub's pull gate judged only the live reading.** `loud` was streak ≥ 2 or rms ≥ 3× the node's level — both about the room *now* — but a stop-capped freeze deliberately lands *after* the cry has stopped, so node 1's 0.177 rms window sat advertised for seven seconds with streak 0 and was never pulled. The node reports the window's own level beside the live one (`win 0.1770`); a new gate `CRYNET_PULL_CRY_RMS 0.050f` pulls a window that loud no matter what the room is doing now (measured: the loudest window the nodes freeze from room sound is 0.036). Also: the monitor's own fault — the wearable's COM30 died on `ClearCommError` when the C3 re-enumerates after a reset, and the capture gave up on it; `capture_multi.py` now reopens the port and carries on. **Verified on the bench: the model and the clip are good (0.9961 on the host), the link is good on nodes 1 and 2 (286/286, CRC MATCH), and the wearable is alive — it rebooted on open and connected to the hub's BLE.** Next: flash the hub with the pull fix, flash nodes 1 and 2 (node 3 is already on the new sketch), re-run the capture with the wearable included, and have the user play the clip. | `firmware/node/node_crynet/node_crynet.ino` (hold inverted; unfetched window protected), `firmware/hub_infer/src/link_stage.c` + `link_stage.h` (pull on the frozen window's own level), `firmware/tools/capture_multi.py` (reopen on a board reset), `research/score_demo_clips.py` (new), `research/out/demo_run/` (the capture) | The three faults are one story: the node froze the right window, kept it three seconds, and the hub never asked for it — and when the hub did come back, the node had already thrown it away. |
509 | 2026-11-04 (the demonstration, rehearsed without a sound — and the yelp test set aside) | **The user asked the right question: on the day, the demo plays a real cry, not a selected yelp, so is the yelp test the right rehearsal? It is not — and the rehearsal that *is* right has now been done, with no sound at all.** **The yelp test answered a narrower question.** It was aimed at the freeze rule: would a short vocalisation be caught whole? That question was answered on the bench (the laptop's speaker cannot carry a short cry — 0.9961 as a file, 0.7422 as played), and it is a *rule* question, not a *demo* question. The demonstration plays a sustained cry, and the demo path for that was already verified live on 2026-11-01 (26 alerts at P 0.9922–0.9961, 45/45 bursts complete). **But the demo kit was stale, and that was worth catching before the soldering.** Every clip in `research/out/demo_kit/` was trimmed and scored under the **old peak-centred freeze rule**; the nodes now run the **hybrid stop-capped** rule, so the window the node freezes is a different two seconds, and the kit's scores no longer said what the hub would do. **So the kit was re-derived against the rule

that is on the boards** (`research/simulate_demo.py`, new): each clip is played into a real quiet room (a recorded stretch of session5), the node's own 100 ms records are computed over it — the adaptive floor, the event level, the loud streak, the peak marker — the hybrid stop-capped freeze is applied exactly as the sketch writes it, and the window the node would hand over is scored with the deployed INT8 model and put through the hub's gates. **The result: 27 of the 37 kit clips announce at the level they were recorded at; 10 do not, and 17 announce only if the phone is turned *down* — their window clips, and a clipped window is refused.** The freeze lands 2.0–5.4 s after the clip starts (median 2.9 s), and the verdict about 2 s after that. **The shortlist for the day is three clips** — `session9_clip_0005` (announces at every volume, the most headroom), `session4_clip_0012` (announces at every volume, wide crest) and `session1_clip_0005` (the loudest reliable one) — and the instruction that matters is: **set the phone's volume once before the demonstration and do not touch it**, because the margin between *announces* and *clipped and refused* is one volume step on most of the kit. **No sound was played.** | `research/simulate_demo.py` (new), `research/out/demo_rehearsal.{md,csv}` (new), `research/out/DEMO_KIT.md` (the shortlist; the stale 'announces from the first window' claim replaced) | The rehearsal is what caught this: the demo kit's own file-level scores were fine (0.9922–0.9961 on the best window of every clip) and would have gone into the demonstration with every one of them listed as safe. The scores were never wrong — they were scores of the *wrong window*. The node judges one window, and the rule that picks it changed on 2026-11-04. **Still open:** the three-cot attribution test (which cot is named when the cry is played at each cot) and the live clip test — neither needs a sound to plan and both need one to run. | 2026-11-04 (the whole bench on one laptop — three cots, the hub, the wearable, and the false-alarm rule that came out of it) | **Every unit was on one laptop at once, and putting them there turned up three things worth fixing: a false-alarm path the model can walk into, a burst that costs a second and a half of dead time, and the last unproven link in the system — which is now proven.** The bench is now the three cot nodes on a USB hub with the hub and the wearable straight into the laptop, and **`firmware/tools/capture_multi.py` (new) holds every port open at once**, writes each line with a timestamp, and can send a command to a port while it watches. **All five boards were identified by their boot banners** — COM30 wearable, COM31 node 1, COM32 hub (CH343), COM37 node 2, COM38 node 3 — and **nodes 2 and 3 were flashed with the hybrid stop-capped freeze rule**, so all three cots now run the same freeze rule. **The rule for announcing a cry has changed, and the change is measured both ways.** The hub's *loud* path used to announce a cry on a single window whenever rms ≥ 0.080 and P(cry) ≥ 0.95. The model does not only lose certainty on clipped cries: in a room with people talking it scored an ordinary conversation window **0.9727** (node 3, rms 0.058, crest 3.5), and the hub's own boot probe scores **0.9805 on synthetic noise** — so a conversation one step louder than that would have been announced as a cry at three in the morning. The loud path is now the saturation (`CRYNET_LOUD_ANNOUNCE_P` 0.9900, and the confirming window needs 0.9900 as well), so **a cry is announced when the model saturates and not before**. The cost, measured on the corpus: **7 of the 53 cry episodes the old rule caught peaked between 0.9688 and 0.9883 and are now missed** (all seven are strong cries — rms 0.09 to 0.22, crest 4.3 to 7.9); on the bench **three genuine cries out of three scored 0.9961 and all three still announce**. **A clipped window is now refused outright**, and the hub says so in the log — the INMP441 is a fixed-gain part whose own output saturates before the node's 16-bit window does, so there is no software cure; the honest answer is placement (the microphone must not be within a hand's width of the sound), and the honest behaviour is to refuse the window and let the clean cot's window announce the cry. **The burst stall is noticed in a tenth of a second instead of a second and a half.** The hub was measuring the gap between fragments from its own polling loop — nearly 200 ms on a busy hub — so it could not tell a burst that had *ended* from one still in the air, and it waited the whole 1.5 s before asking for the fragments that were missing. The fragment timing is now measured in the receive callback, where the arrivals actually happen, and **the median window fetch fell from 2.48 s to 1.88 s (p90 3.45 s → 2.33 s)** — a third of the radio's time handed back to a three-cot room. **The hub has a console now** (`help`, `status`, `alert <cot>`, `clear`): the ESP-IDF log owns the UART and nothing could be read from it, so the console installs the UART driver itself — and the log is untouched, because the log's write path does not use the driver. **`alert 2` made the wearable print `[ble] ALERT cot 2 type 1 conf 100%` and light its LED**: the BLE path is proven end to end, with the right cot, without a sound, which is exactly what was missing. | `firmware/hub_infer/src/link_stage.c`, `src/link_stage.h`, `src/console.c` (new), `src/console.h` (new), `src/main.c`, `src/CMakeLists.txt`, `components/node_link/node_link.c`, `components/node_link/include/node_link.h`, `components/ble_alert/ble_alert.c`, `firmware/tools/capture_multi.py` (new), `research/simulate_freeze.py`, `research/out/ambient_25min/` (new), `research/out/hub_new_build/` (new) | **The 25-minute watch is the false-alarm evidence, and it is clean: 153 windows scored, zero alerts, zero brownouts, zero resets, and not one window reached 0.99.** Three windows reached the 0.95–0.99 band and **all three were quiet** (rms 0.0168 to 0.0232) — which is the interesting part: the measured speech is not loud enough for the old loud path either, and what is *unmeasured* is how the model scores a loud conversation. That is the whole argument for the saturation, and it is why the old rule is one number away (`CRYNET_LOUD_ANNOUNCE_P` and `CRYNET_CONFIRM_P` back to 0.9500). **160 bursts completed and every one of them reached 286/286** — the resend path recovers all of it, at the cost of 303 stall rounds; the timing fix is what makes those rounds cheap. **The power is fine: the three nodes on the unpowered USB hub ran the whole watch with no brownout** (node 1: 6 385 frames sent, 0 lost, 0 times the radio had no room). The boot probe's decision now uses the same rule as the live path, so the banner and the running system cannot contradict each other, and the hub's log no longer prints "advertising is off (a phone is connected)" every time the radio's policy changes — it printed it seven times in five minutes and read like the phone was dropping off. **Still open:** the clipping is a placement question, and the model's weakness on loud speech is a retraining job — the corpus is missing loud-speech negatives, and a room with people talking is the best source of them going. |
510 | 2026-11-04 (node 1's full-chain cry test — the pass that lets the branch merge) | **The cry played at node 1 was caught three times, every time named `cot 1`, every time with a clean cancel — and the whole seven-minute watch produced no false alarm.** The user played `research/out/demo_kit/session9_clip_0005.wav` from the phone at node 1; the hub's raw log (`research/out/hub_node1_fullchain.log`, 7 minutes, 2 800 lines) has every verdict. **The three cry windows, all clean:** request 214.36 s → window **rms 0.112173, peak 22660, crest 6.2** (no clipping — peak well under 32767) → **P(cry)=0.9961** → `ALERT — cry` at 215.57 s and `a cry on cot 1 at 100%` on the BLE link at 215.58 s, cancelled by `the cry on cot 1 at 0%` at 223.57 s; the second at 255.59 s → rms 0.080885, peak 12682, crest 4.8 → 0.9961 → alert 256.81 s, cancel 264.80 s; the third at 276.80 s → rms 0.080854, peak 14141, crest 5.3 → 0.9961 → alert 278.08 s, cancel 286.08 s. **Latency: 1.21 s from the hub's request to the verdict, 1.85 s from the cry's first loud reading to the wrist being told** — the after-peak freeze put the cry inside the window every time, and the loud-window rule made the two quieter windows announceable without waiting for the saturation value. **The three quiet windows were all correctly refused** (P(cry) 0.0469, 0.0273, 0.0195 — one at rms 0.0178, one at 0.0347). The hub's own boot self-test still passes 4/4 (`RESULT: PASS — 4 probes`); its noise probe scores 0.9805, which is the reference agreeing with the host, not a live verdict. **Two things the same log exposes.** (1) **Every burst stalls at 280–283 of 286 fragments and needs a resend round** — all seven bursts completed (286/286 in 1.04–1.10 s), but the shortfall is systematic, not random, and worth chasing. (2) **Node 2 clipped on a loud sound at 414.9 s** (window rms 0.0831, **peak 31801** against 32767) and the model correctly refused it (P(cry)=0.0000) — node 2's microphone is clipping on ordinary room sound, so its gain or its place on the desk wants looking at before it is the cot that has to hear a cry. | `research/out/hub_node1_fullchain.log`, `firmware/tools/capture_hub_wait.py` (new — waits for the hub to be plugged in, then raw-logs its serial for 7 minutes), `SYSTEM_UPDATE.md` | **The branch `feature/cry-window-after-peak` is merged into `main`** after this test, as the user asked: `main` goes from `aae1c7a` to the tip of the branch. **Still open:** node 2's clipping (its peak 31801 on ordinary sound); the systematic 4-fragment shortfall at the start of every burst; node 3 never produced a window in this capture (it was on the air but nothing crossed its trigger); and the wearable was not attached, so the BLE link's *content* was logged from the hub's side only. |
511 | 2026-11-04 (the freeze rule, measured properly — and the third rule it points at) | **The freeze rule was the last unmeasured thing in the system, so it has been measured: the corpus was simulated through the node's own logic, and it says the after-peak rule is safe but loses the short cries. A third rule comes out of that, is on node 1, and works on the bench.** **(1) The simulator.** `research/simulate_freeze.py` (new) runs the node's real 100 ms records over every in-domain session — the mean of the 2 s window, the rms of the newest 100 ms around it, the adaptive floor and level, the loud streak, the peak marker, the freeze, the 2 s refill — then scores the frozen window with the deployed INT8 model and puts it through the hub's gates (the silence gate, the audible floor, the loud rule at 0.08 rms, the quiet saturation bar at 0.99, the crest/level condition). The clips' offsets in the session waveforms are

recovered by exact sample match, so every freeze is labelled *cry* or *room*. **(2) The result** (`research/out/freeze_simulation.{csv,md}`), three freeze rules over the same corpus: | rule | room windows | false alarms | worst room P(cry) | cry episodes caught | | two seconds after the peak (on the boards) | 69 | **0** | 0.4141 | 41 of 273 | | **stop-capped (new)** | 100 | **0** | 0.8516 | **52 of 294** | | peak in the middle (the rule it replaced) | 104 | **0** | 0.7656 | 52 of 297 |. So the after-peak rule is **safe** — not one room window announced under any of the three, and none of the room windows was even loud enough (worst 0.0350 rms) for the 0.95 rule to apply; the quiet bar of 0.99 is what carried them. But it is the **least sensitive of the three**: 41 of 273 cry episodes against the old rule's 52, because a yelp that is over before the two seconds have passed is frozen over the room instead of over the cry. **(3) The third rule.** The window is now frozen **when the cry stops** (the loud streak breaks) and the two seconds after the peak are only the *cap* for a cry that goes on: a short yelp is caught whole, a sustained cry still gets the two seconds that follow its peak. It catches 52 of 294 episodes — the old rule's recall — while the windows it hands over still hold the cry. `PEAK_STOP_DELAY_MS = 500` is the shortest wait before a stopped cry is frozen, so the window cannot be cut on the first dip below the level. **(4) On the bench.** Node 1 (COM31) was flashed with it and the sustained demo clip **announced on cot 1** — `P(cry)=0.9961`, `a cry on cot 1 at 100%`, cancelled cleanly 8 s later — so the working case is unbroken. The three short yelps were all refused (0.1953, 0.0625, 0.5859), which is right: they are not full cries. **(5) The honest limit on that last test:** the yelps were played from the **laptop's speaker**, and the laptop's speaker cannot carry a short cry — the clean `yelp_2.0s.wav` scores **0.9961** as a file, while the node's own frozen window for the same playback scores **0.7422** and its partials sit at 2.3 kHz with 40–50 Hz spacing, which is not a baby's cry. The system volume on this machine was also found to be far below full (the clip first reached the mic at rms 0.015; at full volume it reaches 0.165). **So the yelp comparison cannot be settled on the hardware from the laptop** — it needs the phone. The clips are staged for that in `C:\Users\HP\.t3\userdata\attachments\yelp_test\` (three yelps and the sustained demo clip). | `research/simulate_freeze.py` (new), `research/predict_yelps.py` (new), `research/dump_during_play.py` (new), `research/out/freeze_simulation.{csv,md}`, `research/out/yelp_predictions.md`, `research/out/short_yelps/*.wav`, `research/out/yelp2_window.wav`, `firmware/node/node_crynet/node_crynet.ino` (`PEAK_STOP_DELAY_MS`), `SYSTEM_UPDATE.md` | The two findings that were *not* defects, now measured rather than suspected: **node 2's "clipping" was a desk knock** — at the instant of the event node 2 read rms 0.0570 while node 1 read 0.0050 and node 3 read 0.0170, and in the quiet stretches the two cots' floors are identical (0.003–0.015), so the mics have the same gain and node 2 needs no change; the model refused it correctly (P(cry)=0.0000). **The "systematic 4-fragment shortfall" is random loss** of 2–6 frames per burst — the missing indices differ every burst (204/205, 181/182, 147/270, 93/206) — and the resend round recovered every one of them (286/286 in 7 of 7 bursts) for about 50 ms. **Still open:** nodes 2 and 3 still carry the after-peak rule and need the same flash (`python firmware/tools/flash_node.py --port <port>`); the branch `feature/stop-capped-freeze` awaits the user's decision; and the short-yelp case is still unproven on hardware (the phone test). |

512 | 2026-11-04 (the other two cots — the whole bench freezes after the peak) | **Nodes 1 and 2 were flashed with the branch's node build, so all three cots now carry the after-peak freeze rule, and both newly flashed boards were verified alive with the rule visibly working.** Both nodes were found plugged into this PC (node 1 COM31, node 2 COM37). The flash evidence: node 2's boot `Saved PC` changed `0x4038aab8` → `0x4038aa94` across the before/after captures, and both boards now boot the same binary. The behavioural evidence is the point of the fix: in the *before* capture (`research/out/who_now.log`) node 2 froze its window *during* the loud event — `froze window, burst seq 1` right as the event decayed, the window holding the peak at its end; in the *after* captures (`research/out/who_after_flash.log`, then a fresh 16 s run `research/out/freeze_verify_nodes12.log`) the same-shaped room event ran its whole course on both boards with **no freeze until it had decayed**: node 1 froze at 5.00 s and node 2 also at 5.00 s, each exactly two seconds after its own loud streak began (`PEAK_DELAY_MS` measured from streak start, which the code stamps when the streak reaches 2 records — the rms max sits mid-streak, inside the frozen window's two seconds). Neither board froze immediately any more, which is the old behaviour gone. Both boards boot clean (`node id 1 (derived from the MAC)` on COM31, `node id 2 (saved)` on COM37), stream telemetry, and answered the room's noise with bursts. | `firmware/node/node_crynet/node_crynet.ino` (build on both boards), `research/out/{who_now.log, who_after_flash.log, nodes_alive.log, freeze_verify_nodes12.log, hub_during_cry.log, cry_test_node3.log, node3_cry_window.wav, demo_kit/}`, `research/out/hub_raw_node2_v2.log` | A 4 s serial open resets the C3 (rst:0x15 USB_UART_CHIP_RESET), which is how the boot banners and `Saved PC` were read. **Still open:** the full-chain cry test at nodes 1 and 2 through the hub (the `hub_raw_node2_v2.log`-style retest, phone at 15–25 cm, panel 0.10–0.15) is owed before the whole bench is called verified end to end; node 1's id is still MAC-derived (set it with `id 1` over serial if it ever moves ports); and the branch `feature/cry-window-after-peak` awaits the user's merge decision. |

513 | 2026-11-04 (the retest — the right cot, named right) | **The retest closes the misattribution: the hub published `a cry on cot 2 at 100%`, twice, with clean cancels in between.** The user replayed the cry at node 2 with the phone held **further away** (15–25 cm instead of 10 cm) and quieter (panel ~0.10–0.15). The raw hub log (`research/out/hub_raw_node2_v2.log`) shows the whole thing: node 2's live rms crossed the level at **78.07 s**, its frozen window came up at 83.26 s (rms 0.0561, **peak 5489 — no clipping**, crest 3.0), the model scored **P(cry)=0.9961**, and the hub published **`a cry on cot 2 at 100%` at 83.46 s** — **5.4 s from the cry starting to the wrist being told**. The next window (0.1328) cancelled it (`the cry on cot 2 at 0%`), and the play continuing brought a second clean alert at 131.87 s with its cancel at 139.9 s. **The distance was the whole difference:** the same cot's windows at 10 cm had peaks of 33289 and 33389 against a full scale of 32767 — hard clipping — and clipped windows score 0.0078, 0.0273, 0.1602. **The quiet period after the play raised no false alarm** (2.5 minutes, every verdict logged): node 1's window at 0.9961 (rms 0.0155, crest 3.4) was correctly refused by the crest gate, and the 0.9023 / 0.9219 verdicts were below the bar. The loud-window rule (a window from 0.08 rms may announce at 0.95) did not fire on anything quiet, exactly as the 964-window screening predicted. | `firmware/hub_infer/src/link_stage.h`, `firmware/hub_infer/src/link_stage.c`, `research/out/hub_raw_node2_v2.log`, `SYSTEM_UPDATE.md` | Hub build SUCCESS 11.7 s (RAM 13.1 %, flash 62.6 %), flashed on COM32. Both fixes are on the branch `feature/cry-window-after-peak` (commits `8f5cf73`, `fce9cbb`), **not** on `main`. **Still open:** nodes 1 and 2 carry the old freeze rule and need `python firmware/tools/flash_node.py --port <port>` wherever they are plugged in; and the user's decision on merging the branch. |

514 | 2026-11-04 (the phone test — three cots, and the wrong cot named) | **All three cots now detect the cry played at them, the freeze fix works, and one real defect showed up: the alert named the wrong cot.** The user played the cry at each cot from the phone. The log (`research/out/hub_phonetest.log`, `research/out/node3_phonetest.log`) shows **node 3 alerting three times at P(cry)=0.9961** (node 3 carries the new freeze rule), **node 1 alerting at 0.9961**, and **node 2 reaching P(cry)=0.9961 at 284.3 s** — but node 2 took from ~152 s to ~284 s to get there. **Why node 2 was slow, measured:** its own windows were *clipped* — peak 21405, 33289, 33389 against a full scale of 32767 — because the phone was held ~10 cm from the microphone (live rms 0.255–0.359). Clipped cries are exactly what this model scores near zero on (0.0078, 0.0273, 0.1602), and the model needs ~1.5–1.75 s of clean cry in the window (measured 2026-11-04, earlier entry). Its last window, at rms 0.0975 and crest 6.9, was clean — and scored 0.9961. **Why the wrong cot was named:** at 231.7 s **node 1**'s window (rms 0.0207, peak 2449, clean) scored 0.9961 and was announced, *while the phone was at node 2* — node 1 is close enough to hear the same cry, and it heard it undistorted, so its window was the better evidence. Node 2's own window at that moment (rms 0.1801, peak 21405) scored 0.9805, which is a *maybe*, and a maybe has to wait for a second window to confirm — which never came, because the next windows were clipped. So the hub announced a cot it had never heard the cry at, and 42 s later announced the right one. **Two fixes, both in:** (1) **the node now freezes the two seconds *after* the peak** (`PEAK_DELAY_MS` 500 → 2000, `PEAK_EXPIRE_MS` 2000 → 4000 — measured: the model needs 1.5–1.75 s of cry in the window, and the old rule put the peak in the middle of it, leaving only ~0.5 s of cry in the window of a cry that had just started), flashed to **node 3 on COM38** and verified alive; (2) **a loud window may now announce on the model's own threshold** (`CRYNET_LOUD_ANNOUNCE_RMS 0.080` in `link_stage.h`: from 0.08 rms a window may announce at P ≥ 0.95 instead of waiting for the 0.99 saturation), because clipping lowers the score on an otherwise clear cry — node 2's 0.1801-rms window at 0.9805 would now have announced at 242 s instead of 284 s. **The rule was checked against every window of the corpus first:** of the 964 screened windows, **not one** of the 189 windows from the nine non-cry sessions reaches 0.95 at 0.08 rms (the loudest non-cry window of the whole corpus scores 0.0156), while 21 more loud cry windows become announceable (130 instead of 109). The quiet path keeps its saturation requirement, because that is where the model has been caught saturating on flat hiss. Hub rebuilt (RAM 13.1 %, flash 62.6 %, SUCCESS 11.7 s) and **flashed to the hub on COM32**. |

`firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/src/link_stage.h`, `firmware/hub_infer/src/link_stage.c`, `research/out/hub_phonetest.log`, `research/out/node3_phonetest.log`, `research/out/hub_raw_node2.log`, `SYSTEM_UPDATE.md` | On the branch `feature/cry-window-after-peak` (commit `8f5cf73`), **not** on `main` — `main` is protected until the user decides. **Still open:** nodes 1 and 2 carry the old freeze rule and need the same flash (node 1 is on the other laptop; node 2 is on the air — seq 50, level 0.005 — but not plugged into this PC; `python firmware/tools/flash_node.py --port <port>` does it wherever they are). The hub rebooted once mid-session with `rst:0x1 (POWERON)` — a power-on reset, not a crash or a watchdog. **The demo rule that follows from all this:** the phone plays 15–25 cm from the INMP441, with the panel reading 0.10–0.15 — closer than that and the window clips, which is the one thing this model cannot survive. |

515 | 2026-11-04 (the demo kit — what the model actually needs) | **The demo kit is built and screened, and the model turned out to have a hard operating rule nobody had measured: it needs ~1.5–1.75 s of continuous cry inside the 2 s window; half a window of cry scores 0.02–0.36.** `research/screen_clips.py` (new) scores every clip of every in-domain session with the *deployed* INT8 model and the *notebook's own* front end — 241 clips, 964 windows: **56 clips announce (P ≥ 0.99) in at least one window, 37 already in their first window**, and the nine non-cry sessions never reach 0.99 (their worst is 0.8906). `research/build_demo_kit.py` (new) then trims each candidate to the cry itself, re-scores it, and keeps only those that announce from their **first** window — the window the node freezes when the trigger fires at the head of the clip: **37 clips in `research/out/demo_kit/`**, with `research/out/DEMO_KIT.md` as the list. **Why the demo kept missing, now measured two ways.** (1) *Window placement:* the node freezes `[peak − 1.5 s, peak + 0.5 s]` (`PEAK_DELAY_MS = 500`), so a cry that starts at the peak puts only ~0.5 s of cry in the window — and the measured curve is 0.50 s → 0.004–0.36, 1.00 s → 0.02–0.23, 1.25 s → 0.10–0.95, 1.50 s → 0.21–0.99, 1.75 s → 0.77–0.9961, 2.00 s → 0.9961. Every kit clip is therefore trimmed to the cry, so whatever the node freezes is full of it. (2) *The source:* node 3's frozen window on 2026-11-03 **did** hold the cry (rms rising 0.0076 → 0.0476 across the window, crest 9.0, transfer bit-faithful) and still scored **0.0117** — while the same recording played by the phone scores **0.9961**. The PC/laptop speakers take a perfect signal down by a factor of ~100; the phone is the only source ever shown to work. **Two side findings:** a 100 Hz high-pass rescues 10 clips (0.148 → 0.9961 on `session10/clip_0007`) but also lifts a non-cry window in session 2 to 0.9609, so it is *not* a free win and needs its own bench test before it ships; and `session9/clip_0000` — the clip played on 2026-11-03 — is good from its *second* second (0.1484 → 0.9961), which is exactly the first-crossing freeze rule working against it. | `research/screen_clips.py` (new), `research/build_demo_kit.py` (new), `research/out/clip_screening.csv`, `research/out/clip_screening.md`, `research/out/demo_kit/*.wav` (37 clips), `research/out/DEMO_KIT.md` | Run the tools with **`C:\Users\HP\AppData\Local\Programs\Python\Python311\python.exe`** — it is the only interpreter on this machine with TF 2.21, librosa and pyserial (Python 3.15 has no numpy, 3.14 has no pyserial). `winsound` has **no `SND_SYNC`** (synchronous is the default) — `research/cry_audio_check.py` was fixed for it. **Next checkpoint (needs the phone):** play `research/out/demo_kit/session9_clip_0005.wav` from the phone, 10–15 cm from the INMP441, with the volume set so the panel reads 0.10–0.20, and watch the hub's verdict on all three cots. **The firmware lever, with its cost stated:** `PEAK_DELAY_MS` 500 → 2000 (with `PEAK_EXPIRE_MS` 2000 → 4000, or the freeze can never fire) would make the frozen window the two seconds *after* the peak — always full of cry — at the price of ~1.5 s more latency. |

516 | 2026-11-04 (early, the honest verdict the user asked for) | **The verdict is in, with the window-level proof: the mesh is innocent — the model and the playback source decide everything.** The question was why node 1 detects, node 2 is slow, node 3 "isn't detecting even when rms is high", and whether three nodes can ever all work. **(1) Node 3's chain is proven good end to end:** tonight's cry test (`research/out/cry_test_node3.log`) shows three windows pulled, every burst completing (1.0–2.5 s with resend rounds), and verdicts **2.1–2.7 s from request to P(cry)** — the same speed node 1 gets. **(2) The transfer is bit-faithful** — `research/cry_audio_check.py` (new) plays a cry on the PC, dumps node 3's own frozen window over USB and compares it with the hub's reassembled copy: rms 0.0383 / crest 9.0 at the node vs 0.0384 / 9.1 at the hub.** Nothing is lost on the air. **(3) The model refuses the window anyway: P(cry)=0.027** on a loud, unclipped, cry-shaped window. **(4) The root cause has two layers.** The *source*: the clean training clip itself scores only **P(cry)=0.148** on the host — it is one of the 41 % of in-domain cry windows the deployed model misses at the 0.95 bar (in-domain eval: recall 0.592, fpr 0), and the PC/laptop speakers were already convicted once (D-61: laptop playback P 0.38–0.95, never over). The *front end*: node 3's window carries 20 % of its energy below 300 Hz (partials at 0 and 52 Hz — speaker rumble and mic bias), and stripping it lifts the score 0.012 → 0.586 — still short of 0.95. **(5) The gates stack on top of a model that already misses 4 in 10 cry windows:** `research/analyze_gates.py` on the evening's hub log shows 297 windows offered and only 4 pulled (the pull gate 0.016 + live-loud turns away the rest), and windows pulled at rms 0.016–0.017 sit under the 0.020 announce floor and are refused unheard — that is exactly node 2's "crazy long" path: its cry windows land at 0.017, so it needs a louder or a confirmed window and pays an extra freeze cycle. **Answers:** all three nodes *can* detect — the mechanism is proven per node, and three cots do not slow one cot's verdict (bursts serialize at ~1–2.5 s); the speed floor is **~2.1–2.7 s from request to verdict, ~4–5 s from cry onset** (2 s window centred on the crossing + telemetry notice + burst + 150 ms inference), +4 s when the first window is a *maybe* that needs confirming; **reducing the node count fixes nothing**. **The move:** demo from the phone at panel level 0.10–0.20 at 10–15 cm (the verified path — five alerts at 0.9922+, D-74), pre-screen any demo clip on the host with `analyze_window.py` and only use clips scoring ≥ 0.99, and consider a gentle high-pass (≈100–120 Hz) on the node's capture after a bench test — it is worth ~50× on rumble-laden windows but *hurt* the clean clip (0.148 → 0.027), so it must be validated before it ships. | `research/analyze_gates.py`, `research/cry_audio_check.py`, `research/out/cry_test_node3.log`, `research/out/node3_cry_window.wav`, `research/out/hub_during_cry.log`, `SYSTEM_UPDATE.md` | Instruments that answered it: `analyze_gates.py` (which gate turned each window away), `cry_audio_check.py` (node window vs hub window, byte-faithful), `analyze_window.py` (what the model makes of a window, run under `Python311` — the only interpreter here with TF 2.21). The evening's hub log is `/tmp/hub_now.log` (297 windows, 881/248/718 telemetry lines from nodes 1/2/3). Node 3's RSSI tonight: −52 to −73 dBm at its desk — bursts still complete, just with resend rounds. |

517 | 2026-11-03 (night, merged) | **`feature/faster-detection` is merged into `main` — a fast-forward with no conflicts, and the branch is pushed to GitHub.** `main` goes from `9ff9a0f` to `51a15af`: the resend fix, the sticky BLE arbitration, the slow-while-busy advertising, the counter-dedupe, the node counters, the one-command node flash and the whole documentation trail. Every one of those commits was verified on real boards before it was made, so `main` now carries the state that was measured: three cots announcing a cry, bursts that always complete, and a wrist that attaches in seconds and names one cot per cry. | `main` (fast-forward, 25 commits) | The branch keeps its name on the remote (`crynet/feature/faster-detection`) as the record of the run; `main` is the working branch. **Still open, in order:** the battery dividers are not wired, so every cot reports "no reading" and the panel's `BAT` band stays dark; the 4:1 window compression (286 frames to 72) is a project of its own; this PC has no page file, which killed the C++ compiler twice; and node 3 rebooted once mid-session without its banner being caught. |

518 | 2026-11-03 (night, BLE arbitration, second round) | **The wrist now stays on one cot: the alert is sticky — and it is verified.** The first version of the arbitration named *the loudest* crying cot, and the log showed why that is not enough: two cots in one room both hear the cry and their levels cross over window by window, so the hub published `a cry on cot 1`, then `cot 2`, then `cot 1` again — five times in two seconds — and the wearable buzzed each time. The cot that was named now **keeps the alert** until it stops crying, or until another cot is clearly louder (1.5× margin). The panel's card uses the same rule, so it does not flicker between cots either. **Measured on the bench (hub serial + wearable serial):** the wearable attached in **4 s** after the hub came up and again in **0.4 s** after a hub reboot — the hub re-advertised at the slow 320 ms interval while its radio was busy and the wearable took it at once, where before it had to wait for a gap in a radio that was busy 89 % of the time. **The re-test after the sticky fix: two cry episodes, each one alert and one cancel, no alternating at all** — `a cry on cot 1 at 100%` / `the cry on cot 1 at 0%`, then `a cry on cot 2 at 100%` / `the cry on cot 2 at 0%`, and the wearable logged exactly `ALERT cot 1 type 1` + `type 2`, then `ALERT cot 2 type 1` + `type 2`. **It named the right cot both times:** during the second episode node 2 was hearing the cry (rms 0.159–0.165) while node 1 sat at 0.021–0.025. The wearable's firmware buzzes only on the transition *into* a cry, so a re-issue naming a different cot does not buzz again. The hub's log line said `the cry on cot cot 1` — the doubled word is fixed. | `firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/components/ble_alert/ble_alert.c`, `firmware/wearable/wearable_crynet/wearable_crynet.ino`, `firmware/tools/flash_node.py` (committed as `c352d48` and `f1a6538`; hub

build SUCCESS 21.9 s, RAM 13.1 %, flash 62.6 %, flashed on COM32; wearable build 678 475 B / 51 %, flashed on COM30 with `flash_node.py --sketch wearable`) | The alternating captures that exposed the flaw are `%TEMP%/hub_ble3.log` and `%TEMP%/wearable2.log`; the good ones are `%TEMP%/hub_ble4.log` and `%TEMP%/wearable3.log`. The two lessons worth keeping: the wearable is the thing that has to be *found*, so the hub must stay findable even while its radio is busy — a slow advertisement costs a burst a frame, and the resend path recovers frames; and a "loudest wins" rule needs a margin and a memory, because two microphones in one room will always cross over. |
519
520 | 2026-11-03 (night, BLE checkpoint in progress) | **The physical antenna check closed: node 3's RSSI rose from -48/-50 dBm at desk distance to -32/-34/-31 and finally -23/-22 dBm when held close, then returned to -49 dBm on the desk.** That is normal distance loss, not a bad board; no swap is needed. During the same capture the already-flashed resend fix worked: node 3's burst completed and CRC matched, and node 3 ALERTed at P(cry)=0.9961; node 1 and node 2 also ALERTed. **BLE checkpoint now being implemented (not yet built or flashed):** the wearable will stay discoverable during an ESP-NOW burst at a slower 320 ms interval (currently advertising stops entirely during busy/preparing), and the hub will publish one alert for the loudest crying cot rather than one per cot; the wearable will buzz only on the transition into a cry, not when the named cot changes. | `firmware/hub_infer/components/ble_alert/ble_alert.c`, `firmware/hub_infer/src/link_stage.c`, `firmware/wearable/wearable_crynet/wearable_crynet.ino`, `SYSTEM_UPDATE.md` | Antenna capture: `%TEMP%/hub_ant.log`; verdict: RSSI follows distance. Previous node3/hub capture after resend fix: `%TEMP%/node3_v2.log`, `%TEMP%/hub_v2.log`. The BLE code has **not yet been compiled or run**; next verify builds for hub and wearable, then flash and repeat wearable attach + multi-cot alert test. |
521 | 2026-11-03 (night, the three-cot verdict) | **All three cots now detect the cry — and the reason node 3 never did is a firmware line, not the board.** Node 3 was brought to the hub's own desk and the phone cry was played at it: the node's own console showed the microphone hearing it (rms to 0.29, `EVENT`), the trigger freezing (`win rms 0.1841 (cry mode)`), the radio serving **286/286 fragments in 733 ms**, and the hub reassembling the window with a **CRC MATCH**. So the link, the trigger and the freeze were never the fault. **The fault was the node's own resend rule.** The hub asks again about 670-770 ms after its last ask, and the node dropped any resend request naming the same window within **800 ms** — so it swallowed *every* round after the first. A burst that lost a single frame could therefore never be completed: the hub asked for the missing fragment five times (the log shows the missing index repeated verbatim), the node answered none of the five, the burst sat at 285 of 286 until it timed out, and the whole two-second window — the one holding the cry — was thrown away. One lost frame is the commonest failure on a weak link, which is why this cost the most. **The rule is now two-sided:** a request is a repeat only if it names the same window *and* the same fragments *and* arrives within 250 ms. A round naming a fragment the last round did not name is always answered. **Verified on the bench with both boards reflashed:** node 3's node log now shows two rounds answered per burst (16 then 1, 25 then 2, 31 then 4, 14 then 1), **every one of node 3's bursts completes** (270→285, 261→284, 255→282, 272→285, all CRC MATCH), and node 3 **ALERTed with P(cry)=0.9961** — as did node 1 (P(cry)=0.9961) and node 2. The air was clean: `10 923 frames off the air, 10 923 parsed, 0 bad, 0 bad payload, 55 requests sent, 0 failed`. The hub's stall line also stopped printing `4294963123 frames` — it subtracted the node's counters across a reboot and wrapped unsigned; the delta now counts zero across a restart and says so in a line of its own. | `firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/src/link_stage.c` (committed as `5d38b01`; hub build SUCCESS 32.7 s, RAM 13.1 %, flash 62.6 %, flashed on COM32; node build 990 545 B / 75 %, flashed on COM38, hash verified) | **The hardware suspicion that remains:** node 3's radio is far weaker than its own firmware should allow — on the hub's own desk it reads **-54 to -61 dBm**, while node 1 reads **-20 dBm** from the other room and node 2 **-50 dBm**. Node 3 is the *closest* cot to the hub and the *weakest* one heard, which no distance explains, and it loses 16-31 frames per burst where node 1 loses 11-13 at -20 dBm. **The decisive test is one minute of bench time: hold node 3 5 cm from the hub and read the RSSI.** At 5 cm a healthy board reads better than -30 dBm; if it stays near -55, the board's antenna is the fault (a damaged or obstructed PCB antenna), and that is a swap, not a firmware fix. **Also open:** node 3 rebooted once mid-session in the previous capture (its counters restarted and its window sequence began again at 1) and the boot banner was not caught — the console was open from the start in this session and it did not recur, so it may have been the USB cable or the power on that board. **Also noted on this machine:** it has **no page file** and its commit limit was exhausted, which made the C++ compiler die with `out of memory allocating 16 MB` and the Arduino library probe fail; the builds only ran once the memory was free. |
522 | 2026-11-03 (later still, 3) | **Nodes 2 and 3 are flashed with the new firmware — all three cots now run the same build.** The user swapped the bench around (hub and node 1 unplugged, nodes 2 and 3 plugged into the PC), so both remaining cots were flashed from here: COM37 and COM38, `arduino-cli compile` + `upload`, 990 469 bytes of program storage (75 %), hash verified on both. **The two boards were confirmed to be the right ones before flashing**, because both Pro Minis derive the *same* id (1) from their MACs and only their NVS-saved ids tell them apart: `firmware/tools/who_is_who.py` (new) opens a port, resets the board over RTS/EN and reads the boot banner — COM37 → `node id 2 (saved; MAC would have said 1)`, COM38 → `node id 3 (saved; MAC would have said 1)`. Both kept their ids across the flash (NVS is untouched by an app-partition upload) and both come up on **channel 6**, matching the hub. | `firmware/tools/who_is_who.py` (new), `firmware/tools/flash_node.py` (used) | **Bench plan the user has set:** node 2 stays on this PC with the hub, node 3 and node 1 go to the other laptop for the cry test. Node 1 already carries the new build (flashed and verified in the previous entry), node 2 and node 3 do too as of now. **Next:** play a cry at node 3 (on the other laptop) and at node 2 (here), and read the hub's verdict line on COM32: `its radio handed over N frames since the burst began, M of them left it, K were refused` — N=286 with M small and K=0 means the frames die in the air at that cot (move the cot or kill the interferer); K large means the node's own radio queue is the problem. Also still worth checking on the other laptop: `netsh wlan show interfaces` — the hub's ESP-NOW is on **channel 6**, so a Wi-Fi adapter sitting on channel 6 there is the likely interferer next to node 3. |
523 | 2026-11-03 (later still, 2) | **The hub's patience now follows each cot's own fragment rate, and the whole chain is verified end to end on the new build: node 1 ALERTed with P(cry)=0.9961 and a window CRC MATCH.** Raising the stall threshold to a flat 400 ms fixed node 1's recovery: its bursts went from 0.78 s to 1.4-2.3 s, because a fixed number cannot describe two links (60 ms is twenty times the spacing of a good link and a fifth of the spacing of a poor one). The hub now keeps a running mean of how far apart *this cot's* fragments arrive and waits eight times that for the next one -- 24 ms for a good link, 400 ms for a slow one -- and waits the missing count times the same spacing before repeating a resend round. Node 1's bursts are back to **0.77-0.85 s**. **Verified live on the bench with both new firmwares** (node 1 on COM31, hub on COM32, the PC's own speaker playing `research/in_domain/session1/clips/clip_0001.wav` at node 1's microphone): `window crc32 0x38FE8B00, node advertised 0x38FE8B00 -> MATCH`, `window rms 0.074393  mean 18  peak 15769  crest 6.5`, `P(cry)=0.9961`, **`ALERT — cry`**, and the BLE alert published `cry on cot 1 at 100%`. The hub's own boot self-test still passes (4 probes, front end and model agree with the host). The node's new counters read correctly over the air: `its radio handed over 314 frames since the burst began, 314 of them left it, 0 were refused, 0 times it had no room` -- node 1 puts every frame on the air and the hub loses a few in its own receive path, which is a different fault from node 3's. | `firmware/hub_infer/src/link_stage.c` (committed as `3261364`; hub build SUCCESS 21.7 s, flashed on COM32) | **Still to do on the bench:** flash node 2 and node 3 on the second laptop (`python firmware/tools/flash_node.py --port COM37` and `--port COM38`), then play a cry at node 3 and at node 2. The stall line now carries the verdict in one place: `its radio handed over N frames since the burst began, M of them left it, K were refused`. If N is 286, M is small and K is zero, the frames are dying in the air at that cot (move the cot, or find the interferer); if K is large, the node's own radio queue is the problem. |
524 | 2026-11-03 (later still) | **The link is fixed on paper and the counters that prove it are on the wire: a slow burst is no longer a dead burst, and the node now says whether it sent the fragments or the air ate them.** The user played the cry at all three cots again: node 1 detects it fast, node 2 detected it twice out of several trials, node 3 still never detected one. Re-reading `hub_3diag.log` against the firmware found two faults that the earlier diagnosis had not named, both of which hit node 3 hardest and neither of which is the hub's fault: **(1) The hub called a live burst dead after sixty milliseconds of quiet.** `FRAG_STALL_MS` was 60 (twenty times the 3 ms gap of a good link, and a fifth of the gap between the fragments of a poor one), so a burst that was merely slow was declared stalled and the three resend rounds were spent before the node had finished its first pass; the stall threshold is now **400 ms** and there are **ten** rounds instead of three, still bounded by the burst's own 15 s timeout. **(2) The node threw the

window away in the middle of a transfer.** The preempt rule (a window 3x louder than the frozen one takes its place) could cancel the served-hold at any moment, and in node 3's noisy room every reading is 3x the last window's level -- so it refroze every ~1.3 s and the fragments the hub was still missing went with the old window, which is why no burst of that cot could ever finish. A window the hub is fetching is now protected for 4 s past the last fragment the node sent, and the clock is refreshed per fragment, so a burst that takes ten seconds to cross a bad link keeps its window for the whole of it (the hold is no longer than the old 3 s one, so a cot is not left unable to freeze a new window any longer than before). **(3) Nobody could see where the fragments went**, so the node's own radio counters -- frames handed to the radio, frames the radio refused, frames that left it, times it had no room -- now ride in the telemetry (a new 34-byte tier, backward compatible with the 18-byte one) and the hub prints them with every stall: a node that handed all 286 over and saw few come back has lost them **in the air**; a node whose radio refused them has a **queue too small for the burst**. The two faults are fixed in opposite places, so the next stall line says which one it is instead of leaving it to a guess. Also fixed: the resend path printed a serial line per fragment, stalling the node's loop in the middle of a transfer. | `firmware/hub_infer/components/crynet_proto/*`, `firmware/hub_infer/components/node_link/*`, `firmware/hub_infer/src/link_stage.c`, `firmware/node/node_crynet/node_crynet.ino`, `firmware/tools/flash_node.py` (committed as `86e467a` on `feature/faster-detection`; hub build SUCCESS 37.0 s, node build 990 443 B / 75 % of flash, node 1 flashed on COM31 and the hub on COM32) | **The bench step now:** flash the two cots on the second laptop with the new build (`python firmware/tools/flash_node.py --port COM37` and `--port COM38` -- the new tool finds `arduino-cli` inside the Arduino IDE and falls back to the sketchbook copy if that laptop has no checkout), then play a cry at node 3 and at node 2. The hub's stall line now carries the verdict in one place: `its radio handed over N frames since the burst began, M of them left it, K were refused`. If M is small and K is zero, the frames are dying in the air at that cot and the answer is to move the cot (or find the interferer); if K is large, the node's own radio queue is the problem and the fix is in the node's pacing. **Still open:** the four-fold window compression (which would shorten every burst and shrink the airtime a cry costs), the BLE advertisement surviving a busy radio (D-61), the BLE alert path not arbitrating to the loudest cot, and the battery dividers not yet wired. |
525 | 2026-11-03 (this session, latest) | **The "node 2 and 3 don't detect" fault is diagnosed, and it is not the hub being overloaded — it is node 3's radio path eating the burst.** The user played a cry at all three cots (hub capture `%TEMP%/hub_3diag.log`, 243 KB, the new diagnostic build): node 1 ALERTed twice, node 2 ALERTed once (P=0.9961, 8 s after its window froze, burst 286/286 in 1.09 s), node 3 never announced. **Three findings, each from the diagnostics now in the firmware:** (1) **The hub polls all three fairly and is not saturated** — 55 pulls in the session, split 10/22/23 across nodes 1/2/3, bursts strictly one at a time, and node 2 *completed* every burst at a weaker signal (-62 to -66 dBm) than node 3 ever managed. (2) **Node 3's bursts die in the air.** The hub asked node 3 for a window 23 times and **not one of the 23 bursts ever completed**: each froze a loud, real cry window (rms 0.10–0.25, CRC advertised), the burst stalled with only 24–72 of 286 fragments arrived, and the arrival pattern is dots scattered uniformly across the whole burst — not a solid run cut short, which would mean the node stopped pumping. The RSSI diagnostic makes it conclusive: node 3's burst frames arrive at **-56 to -61 dBm, the same strength as its own telemetry frames (-56 to -59 dBm)**, which pass continuously; and node 2 completes full 286-frame bursts at -62 to -66 dBm. So the node *is* sending everything at full signal, and ~90 % of its 240-byte frames are lost while 32-byte frames sail through — length-discriminating loss, the signature of a local pulsed interferer near node 3 (USB3 port, laptop Wi-Fi/BT, phone hotspot) or a marginal antenna on that board, not distance and not congestion at the hub. (3) **A consequence-amplifier: node 3 churns its frozen windows** — its noisy room straddles its own trigger, so it refroze every ~1.3–3 s (seq 492→493→…→501 within 25 s), the hub rightly abandoned each in-flight burst for the newest ("the node moved to window X while Y was in flight"), and even a modest loss rate could then never be out-run; one request frame was itself lost, so the node's served-hold never engaged and the 3 s `FROZEN_HOLD_MS` expired early. Fixing the air loss alone should end the churn: with requests arriving, holds engage. Node 2's second-trial miss was the model refusing quieter windows (rms 0.047 crest 8.2 and rms 0.039 crest 2.7 both scored `no cry`) — a placement/loudness issue, not a link one. Node 1 at -28 dBm is on the bench next to the hub; that is why it wins every time. | `firmware/hub_infer/components/node_link/*`, `firmware/hub_infer/src/link_stage.c` (committed as `43ed8ab` on `feature/faster-detection`, hub build SUCCESS 38.6 s) | **Next bench test (decides hardware vs placement): physically swap node 2 and node 3's positions and play a cry at both.** If the loss follows the *position*, find the interferer (unplug the laptop's USB3 devices, move phones); if it follows the *board*, it is node 3's antenna/radio. During the same play, capture node 3's serial (`node_console.py --port COM38 --send stats`) — `served 286 fragments in ~1000 ms` proves the pump and `queue full 0` proves its TX, closing the node-side question for good. No reflash is needed: the diagnostic build is already running on the hub. |
526 | 2026-11-03 (this session, later) | **The system now answers a cry in about two seconds instead of five to seven — branch `feature/faster-detection`, measured on the bench.** The analysis came from the hub's own log (`research/live_phone_cry_2026-11-01.log`): a burst took **3.40–3.70 s** of the 5–7 s, the rest being 1.2 s of node-side waiting, up to 2.5 s of hub cooldown and 0.15 s of front end plus inference. **Three faults, found by reading the logs rather than by guessing, and each one measured:** (1) **The node handed the radio one fragment at a time and waited for that fragment's own acknowledgement** — the node's counters said `tx wait: avg 2339 us` per frame, 286 frames, plus a fixed 4 ms gap: about twelve milliseconds a fragment, of which the air time is a small part. The node now keeps four frames in the radio at once, waits for a *group*, and the fixed gap is gone. (2) **The first attempt made it worse — the burst went to 7.2 s — and the node's counters said why: `time req=7230 ms` beside `pump max_us=29954`. Reading the microphone while the radio was busy was meant to stop the node going deaf during a burst, but a microphone read blocks until the samples arrive, so the burst was paced by the rate the microphone produces audio and not by the radio. **The microphone now has a FreeRTOS task of its own** (`capture_task`), it owns the I2S peripheral, and the main loop never waits for audio: the radio is fed as fast as the radio can carry it and the node keeps listening while it is talking. (3) **One fragment in every burst was being lost, always among the first thirty, always exactly one.** The hub now counts frames that *arrived* separately from frames it *stored*, and the count said the frame never arrived at all. The cause is the hub's own Bluetooth: the ESP32-S3 has one radio for both, and while the Bluetooth stack sends an advertisement the Wi-Fi receiver is deaf; the advertising interval is 100 ms and the hub stopped advertising only *after* it had sent the request for the window, so the advertisement already in flight always slipped through and deafened the receiver for exactly the moment the first fragments were arriving. **The advertising now stops the moment the hub hears a cot getting loud** — half a second before the window arrives, so the radio is quiet by the time the burst starts, and nothing is added to the reaction time. Two more from the same logs: the node froze a new window every half second while a cry was going on (its "preempt the hold" rule compared against the *trigger level*, and a sustained cry is far above that, so the hold was cancelled every time), which made the hub abandon the burst it was assembling and ask for the newest window instead — it now compares against the window already frozen, which asks the question that was meant: is this a new event, or the same one going on?; and the hub's stall margin was 1200 ms, longer than a whole burst, so a genuinely stalled burst was never believed. **Measured: the burst 3.4 s → 0.90–1.00 s, eight bursts with no stall, no resend and every window's CRC-32 matching.** | `firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/components/node_link/*`, `firmware/hub_infer/components/ble_alert/*`, `firmware/tools/measure_latency.py`, `SYSTEM_UPDATE.md` | Node build: 75 % flash, 50 % RAM. Hub build: RAM 13.0 %, flash 62.6 %. **The next lever is sending fewer bytes, not sending them faster**: 286 frames × 240 bytes is 64 KB for two seconds of audio, and the radio now carries them as fast as it can (≈3.5 ms a frame). 4:1 compression of the window would take 286 frames to 72 and the burst to ≈0.3 s — a new wire format, a decoder on the hub, and the model's behaviour re-validated on decoded audio, so it is a project of its own. **Also measured:** the node's `battery none` trust gate still holds on every USB-powered cot, and the panel's card layout (the RMS number crowded against the COT label) is still the user's deferred UI item. |
527 | 2026-11-03 (this session) | **The chain that answers a cry is being cut from five to seven seconds down to about two — branch `feature/faster-detection` (off `feature/node-battery-soc`), compile-verified on both sides, NOT yet flashed.** The measurement came from the hub's own log (`research/live_phone_cry_2026-11-01.log`): a burst took **3.40–3.70 s** of the 5–7 s the system took to answer, and the rest was 1.2 s of node-side waiting, up to 2.5 s of hub cooldown, and 0.15 s of front end plus inference. The burst was slow because the node handed the radio **one fragment at a time and waited for that fragment's own acknowledgement** (the node's own counters: `tx wait: avg 2339 us` per frame, 286 frames, plus a fixed 4 ms gap) — about twelve milliseconds a fragment, of which

the air time is a small part. Four changes, all on the path: **(1) the node now keeps a few frames in the radio at once** (`TX_INFLIGHT_MAX 4`, counters for frames queued / acknowledged / in flight) and waits for a *group*, which is what ESP-NOW's ten-deep queue is for; **(2) the fixed gap between fragments is gone**; **(3) the microphone is read while the radio is busy** (`drain_mic`, ten times a second, taking only the audio that has arrived), because the node used to read nothing during a burst — the driver's DMA overflowed, the live window was thrown away, and the cot went deaf for the two seconds the window took to refill, which is also why the hub's cooldown had to be 2.5 s; the window is now only dropped if a read actually comes back short (`g_window_damaged`); **(4) the node's peak delay is 500 ms instead of 1000** — the one-second wait existed to centre the two-second window on the cry, and the model does not need the cry in the middle, only inside it. On the hub: the cooldown 2500 → 1500 ms, the stall margin `FRAG_PER_FRAGMENT_MS` 15 → 4, `RESEND_MIN_FLIGHT_MS` 2500 → 1000 (it was longer than a whole burst, so a genuinely stalled burst was never believed and the hub sat out the full 15 s timeout), and the main loop 50 → 20 ms. | `firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/src/link_stage.c`, `firmware/tools/measure_latency.py`, `SYSTEM_UPDATE.md` | Node build: 75 % flash, 50 % RAM. Hub build: SUCCESS, RAM 13.0 %, flash 62.5 % (the dead `s_swept` warning went with it). `measure_latency.py` now also reports **sound → verdict** and **sound → request**, which is the number the user actually feels, instead of only request → verdict. **Not yet on hardware — the boards are off the USB bus, so nothing here is measured.** The burst time is the number to watch: if it does not fall well under a second, the cost is the radio's own per-frame overhead and the next lever is sending fewer bytes (4:1 compression of the window), not more batching. |
528 | 2026-11-03 (later still) | **All three cots now run the battery firmware.** Node 2 (`90:DA:72:87:CB:98`, COM37) and node 3 (`90:DA:72:85:E0:4C`, COM38) were reflashed with the branch's node image — both keep their saved NVS ids (node 2 answers `node id 2 (saved; the MAC would have said 1)`, node 3 answers `node 3`), and both report `battery none (USB, or the divider is not wired)`, so the trust gate holds on every USB-powered cot. With node 1 (flashed earlier this session) all three now send the 18-byte telemetry, and the hub reads all three. Node 1 and the hub were off the USB bus for this session (device manager: present but disconnected) — the hub was already running the branch build, so nothing was left to write. | (no code changes - bench session) | The first node-1 upload of the session wrote the app at **0xe000** and the board refused to boot; `arduino-cli upload` puts it at **0x10000**, which is what this board's partition scheme uses. Always upload the node through `arduino-cli upload`, not a hand-written esptool command. |
529 | 2026-11-03 (later) | **The branch firmware is on the bench now: the hub shows the new cards, and node 1 runs the battery firmware — the "no RMS number" report was simply the old build still on the board.** The hub was plugged in and reporting LISTENING with no numbers under the bars; its log showed an app built **Oct 31**, i.e. the panel-fix build, because `feature/node-battery-soc` had only been compiled, never written. Flashed the branch build: **boot self-test PASS (4 probes)**, `ili9341: the panel is up: 320 x 240`, and all three cots heard at once — node 1 (`90:DA:72:85:C5:44`), node 2 (`90:DA:72:87:CB:98`), node 3 (`90:DA:72:85:E0:4C`) — with sensible levels (0.004–0.019) and no zeros from any of them. The radio-deaf offline rule and the "worth pulling" gate are therefore running on the board for the first time. Node 1 was then reflashed with the new node firmware: the first attempt wrote the app at **0xe000** and the board answered *"No bootable app partitions in the partition table"* (this board's Arduino partition scheme puts the app at **0x10000**) — re-uploaded with `arduino-cli upload`, which uses the board's own offsets, and it came up clean: `node id 1`, `hub is 30:30:F9:17:2F:80`, and the hub now sees its frames as **34 bytes** (the 18-byte payload) where nodes 2 and 3 still send 32. Its console says **`battery none (USB, or the divider is not wired)`** — the trust gate holding its ground on a floating pin, exactly as designed. Nodes 2 and 3 still run the old node firmware; they need no reflash until the dividers are wired (the hub reads by length), but they also send no battery field yet. | (no code changes - bench session) | A leftover `crynet_hub_watch.py` process was holding COM32 and had to be killed before anything could read the hub. The panel's **LISTENING** banner is the normal dashboard state, not a fault — the third page (the colour check) is behind the BOOT button. |
530 | 2026-11-03 (latest) | **The panel now shows each cot's own sound level and its battery band, and the cots measure their battery — branch `feature/node-battery-soc` (stacked on `feature/link-radio-deaf`), compile-verified on both sides, not yet flashed.** The cards carried a level bar but no number and no battery; under each bar there is now a detail line — the cot's own RMS on the left, and on the right `BAT ~78 %` (green), `BAT LOW - CHARGE SOON` (amber) or `BAT CRITICAL - SWITCH OFF AND CHARGE` (red), the strings §7 of `SYSTEM_WIRING.md` proposed. The node samples GPIO3/ADC1_CH3 through the designed 220 k/100 k divider (one averaged calibrated reading every 4 s, ×32/10 to the cell), gated by a trust rule — three consecutive in-range readings (0.80–1.50 V at the pin) before it believes the divider is wired, two out-of-range to stop believing — so every cot on today's USB bench reports **no reading** and the panel shows nothing rather than a floating-pin invention. The band moves on ±30 mV hysteresis so one radio dip cannot flash LOW; the percentage comes off a generic resting-voltage Li-Po curve, shown with a tilde, and `BAT_CURVE` in `hub_display.c` is where the real cell's profiled curve goes. The telemetry payload grew 16 → 18 bytes (cell mV after the CRC); the decoder reads by length, so old hubs decode new nodes and new hubs show nothing for old nodes — **the hub can be flashed first, the three cots only need reflash when the dividers are wired**. Cards grew 40 → 44 px (step 47); `research/ui_mock.py` renders the same and its three dashboard PNGs are regenerated. | `firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/components/crynet_proto/*`, `firmware/hub_infer/components/node_link/*`, `firmware/hub_infer/components/hub_display/*`, `firmware/hub_infer/src/link_stage.c`, `research/ui_mock.py` | Hub build: SUCCESS (RAM 13.0 %, flash 62.5 %). Node build: 75 % flash, 50 % RAM. The uncommitted radio-deaf offline fix from the previous session was compile-verified (SUCCESS, 13.7 s) and committed as `0186a54` on `feature/link-radio-deaf` — it has never run on the board. **Node 2 health check, from the captures:** in `hub_v22.log` (today 09:56) node 2 sent 540 telemetry lines with **zero** `rms 0.0000` readings and no reboots, against 94/470 (20 %) zeros in `hub_v21.log` (today 09:10) — but node 1 on the *same laptop* in v21 had 0/818 zeros and the node sketch is unchanged (`298d5c0`) between the sessions, so the zeros were board/port-specific, not firmware; they should be expected to return if node 2 goes back on that laptop's port. |
531
532 | 2026-11-03 (latest) | **Node 3 is flashed, named and on the air.** The third Pro Mini (`90:DA:72:85:E0:4C`, COM38) was flashed with `main`'s node firmware (clean compile, 74 % flash) and — exactly like node 2 before it — **derived the id 1 from its MAC**, which would have put two cots on one card; `id 3` was set and **verified across a hard reset** (esptool `run`): after the reset it still answers `node 3  mac 90:DA:72:85:E0:4C`, so the NVS id really does survive. The hub agrees — its log reads `node 3: rms 0.0210 ...` and it asked the new cot for windows (3 requests sent, 0 failed). **Node 2 needed no reflash**: the node sketch's last change is still `298d5c0`, the id-persist build node 2 was flashed with, and a reflash does not erase NVS anyway — so its id 2 is intact and only a read-only `status` check is worth doing. | `SYSTEM_UPDATE.md` | The node's capture path is unchanged from nodes 1 and 2, so no code changed for this. **Next:** all three cots plugged, to see three cards and confirm the hub still attributes a cry to the cot that hears it. |
533 | 2026-11-03 (later) | **The panel fix verified on the real panel, and the two-second rule confirmed from the hub's own log.** The hub was plugged in and flashed from `main` (RAM 13.0 %, flash 62.5 %); it booted clean, the model self-test PASSed (4 probes, front end and model agree with the host), and the panel then behaved as intended — COT 1 ONLINE while the node was plugged, OFFLINE within about two seconds of the unplug, the NODE pill green→amber, the banner LISTENING→SEARCHING FOR THE COTS, and ONLINE again on the replug. **The log proves the liveness window rather than leaving it to the eye:** while the node was plugged the hub sent **no** probes at all; two seconds after the last telemetry it began probing **every 2 s**; and the instant telemetry returned the probes stopped again. One rule — two seconds of silence — therefore governs the cards, the banner, the pill and the radio. (An apparent four-minute delay in the probe restart was a misreading of a capture file that was still being written, while the node was being plugged and unplugged.) | `SYSTEM_UPDATE.md` | No code changed since the merge; the hub ran `main`'s firmware for this session. Node 3 is wired but not yet enumerating on USB. |
534 | 2026-11-03 (earlier) | **The panel now notices an unplugged cot within about two seconds, and the header pills read NODE and WEARABLE with one shared colour rule (branch `feature/hub-display-status`, merged to `main`).** The user unplugged a node and the card stayed ONLINE — because `link_stage.c` latched `heard = true` forever: nothing ever cleared it, so the card, the banner and the RADIO pill froze on the last state a node was in. The fix: telemetry arrival now stamps `last_heard_us`, and a `node_live()` helper (silence < **2 s** — telemetry arrives every ~100 ms, so the timeout is twenty missed frames and cannot trip on one lost burst) replaces the raw flag in every caregiver-facing spot. A quiet cot keeps its card (the caregiver needs somewhere to look) but it reads **OFFLINE**

with the level bar empty, stops driving the banner and the pill, and — a small recovery bonus — the hub **resumes probing** once every cot has gone quiet instead of waiting forever beside a dead link. The nomenclature: **RADIO → NODE** (it is the cots' link, not a radio as such) and **PHONE → WEARABLE** (the thing on the wrist; eight characters still fit the 320 px header with a 4 px margin). The NODE pill's offline colour is **amber**, not the old dim grey, so both pills follow one rule the caregiver learns once: green working, amber waiting. `research/ui_mock.py` was kept in sync. Node 1 (Pro Mini, COM31) was reflashed with main's node firmware — untouched by this branch — and answers `node 1  mac 90:DA:72:85:C5:44`, pumping telemetry. | `firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/components/hub_display/hub_display.c`, `research/ui_mock.py`, `SYSTEM_UPDATE.md` | **The hub was unplugged for all of this, so the hub firmware is compile-verified only** (`pio run`: RAM 13.0 %, flash 62.5 %) — flash it from `main` at the next bench session and watch an unplug flip a card to OFFLINE within ~2 s. |
535 | 2026-11-03 (later) | **The distinguishing test PASSED properly this time — the nodes were on separate machines, the cry was played at node 1 first and then at node 2, and the hub attributed each cry to the cot that heard it.** Arrangement: hub + node 1 on the development PC (COM32 / COM31, node 1 = MAC `90:DA:72:85:C5:44`), node 2 + the wearable on the second laptop. Raw capture of both (`hub_v22.log`, `node1_v2.log`, 260 s). **The result, in the hub's own words:** at t=117.0 s node 1's window scored `P(cry)=0.9961` → **ALERT — cry** (rms 0.1434, peak 23463, crest 5.0); at t=147.5 s node 2's window scored `P(cry)=0.9961` → **ALERT — cry** (rms 0.2582, peak 32859, crest 3.9). **The attribution flipped exactly as it should**, and it was decided on level, not luck: during node 1's play node 2 was at rms 0.0219 (refused, `no cry`), and during node 2's play node 1 was at rms 0.0192 — refused by the **too-quiet gate** (`cry (too quiet to announce)`, P(cry)=0.9688 at rms 0.0192 < 0.020) rather than by the model. Node 2's own first cry window (rms 0.0999, P=0.9805) was correctly held as `cry (unconfirmed)` and confirmed by the next window. **The link was clean throughout: 56 bursts, all 286/286, 0 CRC mismatches, 0 timeouts, 22 660 frames off the air with 0 bad and 0 failed requests.** **Two follow-on facts, both real:** (a) **the wearable connected late, and the cause is now known** — the hub advertises only when `!s_connected && !s_radio_busy` (`ble_alert.c`), and this session's radio duty cycle was **89 %** (230 s of burst airtime in 260 s), so the hub was silent to BLE for most of the session and the wearable's scan had to wait for a gap. That is the D-61 advertising policy colliding with a busy radio, and it is worth fixing: the wearable is the thing that has to be found. (b) **The polling fairness improved a lot with the nodes apart** — node 1 took 38 requests and node 2 took 35 (round 2, same room, was 31 vs 19), and the 66 stall warnings remain the resend path working, not a failure. | (no code changes - verification only) | Logs: `%TEMP%/hub_v22.log`, `%TEMP%/node1_v2.log`. **Now proven:** the hub attributes a cry to the cot that actually hears it, with the two cots in different places. **Still open:** the BLE advertisement must survive a busy radio (the wearable's late connect), and the BLE alert path still publishes per-node without arbitrating to the loudest cot the way the panel does. |
536 | 2026-11-03 | **The two-node distinguishing test ran for real, but the arrangement has to be described honestly, because it changes what the result means.** Round 2 (fresh capture, hub COM32 + wearable COM30 raw-logged together, both nodes on the second laptop): the cry was played at the nodes' location, and the hub logged **5 ALERT lines — 3 for cot 1 and 2 for cot 2** (the fifth, at t=277.5 s, is a re-announce while cot 1 was already active, so the state-change publish rule sent no BLE event for it) — while the wearable's own log independently recorded **7 BLE events: `ALERT cot 2`, `ALERT cot 1`, `cancel cot 2`, `cancel cot 1`, `ALERT cot 1`, `cancel cot 1`, `ALERT cot 2`** — 4 cry alerts and 3 cancels, the counts matching the hub's announcements one for one. So both cots are visible on the wire and on the wrist unit, the ids travel correctly through the hub, and the cancel events arrive. **Two real findings came out of it, and one of them corrects an earlier claim.** (a) **The nodes were on the SAME laptop this time**, so they sat centimetres apart: the round-1 result (4 ALERTs, all cot 1) was measured with the nodes on different machines and is not comparable to this one — this round proves *both* cots are heard and correctly labelled, not that one node is preferred. (b) **The 75 `the burst stalled` lines are not a link failure**: every one of the 48 bursts still finished at **286/286 fragments with 0 CRC mismatches and 0 timeouts**, and the stalls are the hub asking for the fragments it did not have yet — the recovery path working, not the link breaking. The round-1 log showed *zero* stalls only because `listen_hub.py`'s regex matches `I (...)` lines and silently **drops every `W (...)` line**, so the stall warnings were never captured; the raw capture in round 2 shows them. (c) **A real defect the test exposed:** with both nodes in the same room, the hub asks both for windows at once and their bursts **overlap in the air (28 of 48 bursts, 36 of 50 requests)** — the radio is at **86 % duty cycle**, and node 1 got 31 requests against node 2's 19, so the two cots are **not polled fairly**. The panel already arbitrates to the loudest crying cot (`link_stage.c`, the `show` loop), but the **BLE alert path publishes per-node and does not arbitrate** — hence the wearable saw cot 2 and cot 1 alternating for what is one physical cry. | (no code changes - verification only) | Logs: `%TEMP%/hub_v21.log`, `%TEMP%/wearable_v4.log`. **Not yet proven:** that the hub attributes a cry to the *nearest* cot when two cots hear it (needs the nodes physically apart), and that the fair-polling and BLE-arbitration gaps above are worth fixing. |
537 | 2026-11-02 | **Node ids now persist in flash (branch `feature/node-id-persist`), because the second Pro Mini derived node 1's id.** The user's wearable confirmations landed first: the onboard LED blinked on every alert (the haptic stand-in fires) and the screen returned to idle via `cleared` — stage 2d's remaining checks closed. Then the node-2 board (`90:DA:72:87:CB:98`, COM37) was flashed with main's firmware and answered `node 1` — the MAC-derived rule `(mac[5] & 0x0F) + 1` collided across identical boards, and the old `id <n>` command was RAM-only, so a power cycle silently reverted it. The fix, small and backward-compatible: `id <n>` writes the id to NVS (`Preferences`, namespace `crynet`), boot loads the saved value and keeps the MAC-derived number as fallback for never-named boards, and the boot line says which source won (`saved; the MAC would have said 1`). **Verified on the real board: `id 2` → saved → hard reset via esptool → status still answers `node 2`.** Node 1 keeps running main's firmware (its derived id is 1 and correct). The distinguishing test — cry near cot 1 vs near cot 2, hub must attribute to the loudest — is next. | `firmware/node/node_crynet/node_crynet.ino`, `SYSTEM_UPDATE.md` | Branch `feature/node-id-persist` off `main@7758717`; compile 74 % flash; NVS survives normal reflashes (not a full `--erase-all`). |
538 | 2026-11-02 | **Wearable verified live: hub ALERT → wearable OLED, first time.** Bench: hub on COM32 + node 1 (Pro Mini, COM31) on the laptop, wearable SuperMini (COM30) on a second laptop for power — its serial lives there, so the OLED was read by eye. The wearable attached and subscribed on its own (read the state characteristic: `0 cot(s), mask 0x00`), then one cry play: **hub fired 8 × `ALERT — cry`; the wearable displayed `COT 1 / CRY 100%`** — microphone → node → ESP-NOW → model → BLE notify → OLED, witnessed end to end. The play's windows sat at win rms 0.0621 `(cry mode)` — right in the volume recipe. The user-observed LED blink (haptic stand-in) and the `cleared` transition are still to be confirmed by eye. Two bench findings: (1) **the power bank cycles even the wearable's light load** — screen dies after a while; every board needs stable USB power (second laptop works; powered hub planned); (2) the node on the bank showed the failure signature clearly — telemetry alive, bursts never completing, hub re-asking seq 1 — and on USB served 8/8 bursts with zero reboots, so the firmware is innocent. The scan-debug line in the sketch stays until the multi-node test; the alert test is **closed**. | `SYSTEM_UPDATE.md` | Logs in `%TEMP%/hub_v18, node_v16, hub_v19, node_v17, wearable_v1–v3.log`. Branch `feature/wearable-alert`; `main` untouched. |
539 | 2026-11-02 | **Wearable firmware written — the BLE client side of the alert contract, compiled but not yet flashed.** New sketch `firmware/wearable/wearable_crynet/wearable_crynet.ino` on branch `feature/wearable-alert`: scans for the hub (`INFANT_MONITOR_HUB`, service `19b10000-…`), subscribes to the six-byte alert characteristic, renders cot / confidence / event on the SSD1306 (SDA GPIO5, SCL GPIO6, 0x3C), reads the state characteristic once on connect so a mid-cry attach still shows the cry, reconnects by rescanning. The motor does not exist yet: `haptic_trigger()` is the DRV2605L hook and the onboard LED (GPIO8) blinks as the stand-in, so the notify→physical-reaction path is testable now. Compiles clean for `weact_studio_esp32c3` (51 % flash, 5 % RAM) against esp32 core 3.3.12 (by-value `onResult`, reused `BLEClient` — the core resets `m_haveServices` per connect, so one client object survives reconnects). `install_sketches.py` now installs the wearable too. The alert test (wearable + hub, play a cry) is next; nothing is merged to `main`. | `firmware/wearable/wearable_crynet/wearable_crynet.ino` (new), `firmware/tools/install_sketches.py`, `SYSTEM_UPDATE.md` | Not verified on hardware: OLED wiring untested, BLE interop with the NimBLE server unproven, LED pin 8 active-low assumed. Branch `feature/wearable-alert` off `main@7758717`. |
540 | 2026-11-02 | **ESP32-C3 Pro Mini verified as the node board (live, on the desk).** The user swapped the SuperMini for a Pro Mini and the same firmware was flashed unchanged via arduino-cli (`weact_studio_esp32c3`, native USB VID 303A:1001, COM31). Correlated captures (`$TEMP/hub_v14.log`, `node_v14.log`, then the play capture `hub_v15.log`, `node_v15.log`): link up at once (hub heard it as

node 1), bursts 286/286 served, quiet room ~20 windows all refused, then **two cry plays → two ALERTs**: seq 51 frozen at 69.7 s (win rms 0.0504, cry mode) → ALERT 73.3 s (~3.6 s later), seq 52 frozen 75.3 s (0.0420) → ALERT 77.8 s (~2.5 s later). Two tail `resend refused` lines were the known hub-side timing wart, present in earlier sessions too, not a board regression. **Board plan settled: nodes = Pro Mini (×3), the existing SuperMini moves to the wearable.** No firmware changes; `main` untouched. | `SYSTEM_UPDATE.md` | Verification session; logs in `%TEMP%/node_v14, hub_v14, node_v15, hub_v15.log`. |
541 | 2026-11-02 | Completed the battery-powered wiring design from the photographed LX-LCBST-TC-001 pad map (B+/B− battery, VO+/VO− output ×2, IN+/IN− input unused, USB-C charging, trimpot → 5.1 V). Fixed the wearable pins (SDA 5 / SCL 6, LEDs GPIO1/GPIO4 — GPIO0 is a C3 boot strap and the earlier assignment was wrong), assigned the hub BLE LED to GPIO2 (normal pin on the S3; S3 strapping is 0/3/45/46) and the SOC LED to GPIO17, and rebuilt the five-page diagram PDF with complete per-unit wiring (node ×3, hub, wearable, module pad map + bench procedure). The user's board swap is planned in: Pro Minis → nodes, the existing SuperMini → wearable. | `SYSTEM_WIRING.md`, `SYSTEM_WIRING_DIAGRAMS.html`, `SYSTEM_WIRING_DIAGRAMS.pdf`, `HARDWARE_COMPONENTS.md`, `SYSTEM_UPDATE.md` | All battery-chain wiring is proposed, not built: charge current, boost quiescent drain and the unlabelled front −/+ pad remain unmeasured; one module must pass bench validation before the rest are bought; firmware (divider, LEDs, heartbeat ACK, SOC display) does not exist yet and goes to a branch. |
542 | 2026-11-02 | Added a preliminary GPIO, ADC divider, LED and application-ACK design to the wiring document after user requested estimated SOC and real hub acknowledgement. | `SYSTEM_WIRING.md`, `SYSTEM_UPDATE.md` | Design only: exact board/protection topology, unpowered ADC isolation, GPIO exposure and firmware remain unverified; do not assemble from assumptions. |
543 | 2026-11-02 | Created a separate four-page illustrated wiring guide: verified node and hub pin-to-pin wiring, proposed wearable bus and a clearly labelled battery power architecture. | `SYSTEM_WIRING_DIAGRAMS.html`, `SYSTEM_WIRING_DIAGRAMS.pdf`, `SYSTEM_UPDATE.md` | PDF text and page boundaries checked; wearable and battery details remain unverified, not a build-ready battery schematic. |
544 | 2026-11-02 | Exported the wiring guide to a three-page PDF and checked its text is present through the final section. | `SYSTEM_WIRING.pdf`, `SYSTEM_UPDATE.md` | PDF is a printable text guide; proposed battery wiring remains unvalidated. |
545 | 2026-11-02 | Added a point-to-point USB test wiring guide and proposed battery-system architecture; explicitly marked wearable pins, battery charger power paths, LED/ADC pin choices and portable circuits as unverified pending component selection. | `SYSTEM_WIRING.md`, `SYSTEM_UPDATE.md` | Documentation only; no new circuit or firmware has been tested. |
546 | 2026-11-02 | Expanded the hardware BOM at the user's request to include proposed battery power, chargers, boost converters, switches, battery monitoring and indicator components down to individual resistor and capacitor counts for the full system. | `HARDWARE_COMPONENTS.md`, `SYSTEM_UPDATE.md` | Planning assumptions only: circuit design, component compatibility, node indicator/ADC firmware and battery testing remain open. |
547 | 2026-11-02 | Replaced the old hardware specification with a current, quantity-based purchasing document. The user confirmed the hub TFT is 2.8 inches and declined further colour checking; additional-node testing awaits the other nodes. The recommended wearable uses a DRV2605L rather than the discrete MOSFET circuit; wearable power and motor selection remain open. | `HARDWARE_COMPONENTS.md`, `SYSTEM_UPDATE.md` | Documentation only; wearable, battery power and multi-node operation remain unverified. |
548 | 2026-11-01 (later) | **The correlated three-minute session detected the cry three times and measured the last two problems precisely.** The mic heard the phone cry at RMS 0.26-0.53 (the panel 0.265 was real - an earlier "live" listener had stopped before the user played and had measured silence; lesson: a listener must span the whole playing window or it lies). The node froze one window per cry segment, all cry-level, marked `(cry mode)`; the link ran 286/286 with CRC MATCH every burst; **three windows scored P(cry)=0.9961 and alerted; zero false alarms** before and after. **Two findings:** (1) **the hold cost 6.5-7.0 s between every burst** - with cry mode capping freezes, the 8 s hold (D-67) lost its queue duty, so `FROZEN_HOLD_MS` returned to 4000 (D-71); (2) **the phone now clips the microphone** (peaks pegged at 32694-33100 of 32767) and clipped cries are a lottery for the model (3 of 13 loud windows recognized, vs every window in session 1 unclipped 45-75 % range) - the demo must play the phone quieter, targeting the panel level of ~0.10-0.20. | `firmware/node/node_crynet/node_crynet.ino` | *The node did not reboot in this session; the v8 reboot remains unexplained but has not recurred.* |
549 | 2026-11-01 (later) | **The second verification session found the real latency bug: the node froze faster than the hub could empty the queue.** D-66 removed the re-request loop (verified: no seq was asked for twice), but the first cry alert still took ~27 s. The node was freezing a new window every ~4 s (the hold was 4 s and during a continuous cry the loud streak never drops, so the node freezes the moment the hold expires) while the hub processes one window every ~4 s - the two rates matched exactly and the queue never drained: the first cry window (seq 17, frozen at 87.7 s) waited behind seqs 13, 14, 15 and 16 and was not judged until 115.4 s. **Fix (D-67): `FROZEN_HOLD_MS` goes from 4000 to 8000.** The node now freezes at most one window every 8 s while the hub empties one every 4 s, so the hub stays ahead. Eight seconds still covers the hub's three resend rounds. Expected first alert at 15 cm: ~5 s (1 s peak delay + ~4 s burst) instead of 27 s. **Not yet verified on the live boards** - built, flashed, node boots and runs. The second session also confirmed: **the 1 m cry is genuinely detected** - seq 21 (frozen at 109.85 s, win rms 0.0103) was judged at 141.98 s at P(cry)=0.9961, and by then the phone was at 1 m (rms dropped from 0.28 to 0.015 at 105 s). 14 verdicts, 3 cry alerts, 0 false alarms on the quiet windows (0.0000-0.9883, none announced). | `firmware/node/node_crynet/node_crynet.ino` (FROZEN_HOLD_MS) | *The second session also confirmed the burst still completes 286/286 with CRC MATCH every time, and the burst length is still a range (3.40-6.93 s) from WiFi contention. The "resend refused" fragility did not recur (D-66 side effect).* |
550 | 2026-11-01 (later) | **The re-request loop that hid the one-metre cry is fixed.** The first verification session (phone at 15 cm, then at 1 m) worked — the cry at 15 cm announced 7 times at P(cry)=0.9961, the cry at 1 m reached the model and was called a cry at P(cry)=0.9961 (win rms 0.0103, where the trigger had been 0.030 before), and **no false alarms** in the quiet windows (0.0000-0.0195) — the 0.9219 near-silent window was correctly not announced. **But the first alert at 15 cm took ~21 s, not 2.5 s.** Root cause: the hub re-requested the *same* frozen window up to four times in a row (seq 100 was asked for four times), and each serve reset the node's 4 s hold timer, so the node could not freeze the new loud window until the hub stopped asking — the new cry sat unfrozen for ~15 s. The re-request never helped anything: re-judging the same window gives the same answer, and the run rule counts distinct sequences, which a repeat never advances. **Fix: the hub now records the seq of every window it actually receives and judges (`last_completed_seq`) and will not request a frozen window whose seq it has already judged; it waits for the node to advertise a new one.** Side effect: the "resend refused: no frozen window for it" fragility (four times in the session) disappears, because the node is no longer asked to re-serve a window whose hold has expired. First alert at 15 cm should now land ~3 s after the cry starts. **Not yet verified on the live boards** — the fix is built and flashed, boot self-check passes, but the latency number is from analysis, not measurement. | `firmware/hub_infer/src/link_stage.c` (struct field + burst-complete assignment + request condition) | *The session also confirmed the burst still completes 286/286 with a CRC MATCH every time, and that the burst length is still a range (3.45-7.48 s) from WiFi contention on channel 6. The 1 m result is real but the transition window (seq 103) could be the tail of the 15 cm cry rather than the 1 m cry — the user's move timing is unknown; a second short session would settle it.* |
551 | 2026-11-01 (late) | **The adaptive trigger is built on both boards — the node owns the noise floor and the hub obeys it — and it exposed a real fault: the model is not a reliable judge of a near-silent window.** The multiple was chosen from the recorded sessions (`research/adaptive_trigger_study.py`): at **three times the node's own noise floor** the cry a metre away goes from 0 trigger events a minute to 39, the quiet room gives 5, and the loud sessions are capped by the old fixed 0.030, so the rule is never worse than before. The **node** owns the floor (it falls at once when the room gets quieter, rises with a 50 s time constant so a long cry cannot drag the floor up to its own level) and the level that follows from it; the level travels in the telemetry's payload 7 in milli-RMS, beside the frozen window's own level and the floor, and the **hub triggers on the number the node reports**, falling back to `CRYNET_TRIGGER_RMS` for an older node. The node now arms its freeze on **two** consecutive readings at or above the level — the rule the hub and the study both use — and the telemetry is 16 bytes. **Measured live:** the level tracks the room (0.006-0.016 in a noisy room; the node's own log reads `rms 0.0063 floor 0.0028 level 0.0083`), and the hub asks 5-11 times a minute instead of 1-2. **But the lower trigger asks the model about near-silent windows, and there the model is uncertain:** the same quiet room produced

verdicts of 0.0000 and of 0.9766 for windows that differ by nothing a person could hear — the second of those a false alarm. **So the announcement rule changed:** one window announces a cry only when it is unmistakable (P(cry) >= 0.99, where the model's output saturates at 0.9961 for a clear cry, so the panel still shows the cry two and a half seconds after it happened); between 0.95 and 0.99 it is a *maybe* that the next window has to confirm. A cry lasts for seconds and its windows agree; a noise fluctuation that looks like a cry is not repeated. **And the node's transmitter was turned down to 8.5 dBm** — the radio sits centimetres from the INMP441 and was putting its own noise into the audio it was about to be judged on. Five minutes of a noisy room afterwards: **32 verdicts, 0 false alerts.** Two new instruments came out of the diagnosis: `firmware/tools/dump_node_window.py` (writes the node's frozen window out over its USB console) and `research/analyze_window.py` (says what is in a window — where its energy sits, how much of it is slow wander — and what the model makes of it). | `firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/components/crynet_proto/{include/crynet_proto.h,crynet_proto.c}`, `firmware/hub_infer/components/node_link/{include/node_link.h,node_link.c}`, `firmware/hub_infer/src/{link_stage.c,link_stage.h}`, `firmware/tools/{dump_node_window.py,listen_hub.py}`, `research/analyze_window.py` (new), `research/out/{node_window_quiet.wav,radio_test/*,dump_*.wav}` | *The diagnosis behind it, in short: the quiet-room windows the node freezes with the adaptive trigger carry broadband noise the recorded quiet room does not have, and the model reads that shape as a cry now and then (0.84-0.98). The same room recorded with the radio off (`node_capture`, which has no radio at all) is refused decisively (0.0000-0.1953), so the microphone's DC drift and the room's own noise are both innocent. The false alarms are what the node's own transmitter put into the window; turning the power down halved them and the announcement rule covers the rest.* |
552 | 2026-11-01 (last thing tonight) | **The demo's sound path is verified on the live boards: the cry from the phone at the node's mic is called a cry, and the whole chain runs in 2.2-5.4 s.** The user played the cry from a phone 10-15 cm from the INMP441 while the hub's log was read live (`firmware/tools/listen_hub.py`, new - it plays nothing itself, it only watches). **42 verdicts over a four-minute listen: 26 "ALERT - cry" at P(cry) 0.9922-0.9961 and 16 "no cry" at P(cry) 0.0078-0.2148** - the model's INT8 output falls into two clean clusters with nothing near the 0.95 threshold, so no verdict was a coin toss. **The link was perfect: 45 bursts, 45/45 with all 286 fragments, and not one stall, timeout, CRC error or resend round.** Request-to-verdict: min 2.23 s, median 3.72 s, max 5.35 s (mean 4.03 s), of which 0.15 s is the front end plus the inference. **The burst's length is a range now, not a constant: min 2.06 s, median 3.55 s, max 5.19 s** - the node paces at 4 ms, but the hub and the node share WiFi channel 6 and the per-fragment transmit cost is the *channel's* contention, so with the phone's WiFi on the same band each frame waits longer. That is the same contention term that made the burst 5.5-6.9 s before D-61, now visible as a range. **The 16 "no cry" verdicts are the architecture showing itself, not a classifier failure:** the node freezes the *loudest* two seconds it has heard and the hub judges that one window, so a window containing the cry that is not the loudest is never offered to the model at all; the no-cry verdicts fall in blocks (sequences 6, 8, 9, 16, 17, 22) and the cry verdicts in blocks (7, 10-15, 18-20). **The demo must use the phone** - D-61's laptop-speaker finding stands - and the demo's own claim is what now holds: a cry at the cot is called a cry. | `firmware/tools/listen_hub.py` (new), `research/live_phone_cry_2026-11-01.log` (new), `DECISIONS.md` (D-62, D-61 amended) | *The one number the user asked for: **the phone path works on the live hub** - 26 confirmed cries out of 42 judged windows, with the link never dropping a fragment.* |
553 | 2026-11-01 (later still) | **The card fills swapped, and the word shortened.** The user looked again and read the fills against the intent: the accent-washed fill on a live card read **faint**, the border-washed fill on a dead card read **denser and more active** — so they are swapped exactly as asked (a live card wears the border wash, an offline card the faint accent wash; the outlines, which the previous round fixed, stay). And at the user's suggestion the dead card's word became **OFFLINE** — shorter, and the network term for the state. Contrast re-passed (every threshold cleared), `research/ui_mock.py` mirrors both changes, flashed to the hub: boot self-check **PASS (4 probes, 5184/5184 exact)**, node heard on channel 6 immediately. | `firmware/hub_infer/components/hub_display/hub_display.c`, `research/ui_mock.py`, `research/out/*` (previews), `DECISIONS.md` (D-59 amended) | *The user is ready to record the confusion session (O-9) once this is confirmed on the glass.* |
554 | 2026-11-01 (later) | **The panel's second look from the chair: darker ground, a teal header, and the online/offline treatments swapped.** The background dropped another step of the dark ramp (`(4,8,8)` → `(2,4,4)`) because the cards at one step up still read as the same colour on the glass (the panel lifts the dark end until a two-step gap reads as one). The header got its own colour `(7,20,15)` — five steps of green above blue — because the backlight's LEDs run along the top edge of the glass and lift that strip blue: a header painted the same colour as the cards photographed blue while the cards photographed teal. A live cot's outline is now the **full-strength vibrant teal** (rose when crying) and a no-signal card recedes — outline and empty bar washed towards the background — the reverse of the 45 % blend the live cards carried before, which had the dead cards reading as the livelier ones. Contrast re-checked (every threshold cleared), `research/ui_mock.py` mirrors the new treatments, and the board was reflashed: boot self-check **PASS, 4 probes, 5184/5184 exact**, radio clean (450 frames parsed, 0 bad), one full window 286/286 CRC MATCH at inference 103 ms. | `firmware/hub_infer/components/hub_display/hub_display.c`, `research/ui_mock.py`, `research/out/*` (previews), `DECISIONS.md` (D-59) | *The Kaggle zip in `research/kaggle_out/` is the same artefact already unpacked and deployed (D-58) — no further action.* |
555 | 2026-11-01 (evening) | **The Kaggle re-run came back, the retrained model was measured, deployed, and verified 4/4 on the board — and the music false-alarm rate fell from 8.19 % to 0.21 %.** The user ran the training notebook and dropped `crynet_out.zip` (the improved download cell worked first try). The retrained model is the same architecture (122,770 params, conv_cnn, cross-AUC 0.9987) with new weights and a **recalibrated input** (`IN_SCALE` 0.060125612 → 0.116329655, `IN_ZP` −4 → −70; threshold 0.95, silence gate, output calibration and geometry all unchanged). Measured on the repo's own evals (`eval_music.py`, `eval_esc50.py`, `eval_indomain.py`): **music 2/965 = 0.21 % (was 79/965 = 8.19 %)**, ESC-50 non-cry 17/1439 = 1.18 % (was 2.64 %), ESC-50 `crying_baby` **36/40 — identical**, quiet room 0/236 — identical, phone-cry sessions INT8 recall 0.867 → 0.800 (session 1) and 0.724 → 0.592 (session 4, AUC 0.928 → 0.941). The trade was accepted on purpose: a cry is sustained, so an episode still trips many windows in a row, and music was the requirement the old model structurally failed. Deployment: `model_data.h`, `crynet_tables.h`, the staged copies in `research/out/`, the mirrored macros in `crynet_model.h`, and **regenerated probe references** (`firmware/tools/gen_probe_refs.py`) — the boot self-check then reads **RESULT: PASS — 4 probes, 5184/5184 features exact on every probe**, inference 102 ms, the noise probe alerts at P(cry) 0.9805, and the radio is clean (339 frames, 0 bad). | `firmware/hub_infer/components/crynet_model/include/{model_data.h,crynet_tables.h,crynet_model.h,deploy_config.json}`, `firmware/hub_infer/components/crynet_dsp/include/crynet_probe_refs.h`, `research/out/*` (new staged artifacts + eval JSONs), `research/kaggle_out/` (the zip unpacked), `DECISIONS.md` (D-58, I-41, O-8) | *The drift guards earned their keep: the build stopped on the `CRYNET_INPUT_ZERO_POINT` static assert, and the first boot failed its feature check 0/5184 exact until the probe references were regenerated for the new quantisation (I-41). The old model is backed up at `research/out/deployed_model_backup/`.* |
556 | 2026-11-01 | **The teal was picked off the panel itself, and the rounded corners were un-flipped.** The user photographed the colour check again and named the colour they want: **the dark ramp's third step, `(4,8,8)`, one step darker than the background it replaces** - and because a ramp step is a colour the firmware *asked for*, the choice is exact regardless of the camera (I-38). The neutral family moved down one step of the same ramp with it: background `(4,8,8)`, surface `(7,15,16)`, surface-hi `(12,26,28)`; the border `(78,116,112)` and dim text `(122,150,146)` kept, since contrast only improves as the background darkens (re-checked: text 16.5:1, dim 6.3:1, accent 8.5:1, alert 5.8:1, card border 3.72:1). **The corners**: the user saw every rounded corner 'sort of flipped' but the bottom-right one - and they were right. The corner loop walked its offsets only *inward* from each disc's centre, painting all four arcs inside the shape; bottom-right only looked correct because its inward quadrant happens to end at the shape's own edge. Each disc is now walked **towards its own corner** (per-corner signs), and a new regression check (`research/check_corners.py`) requires the four corners to be exact mirror images of each other and the fill's arc to hug a circle within 1 px - the old loop fails it, the new one passes. A new `research/measure_ramp.py` samples the ramp out of a photograph. **All nine local commits are now pushed to GitHub** (`crynet/main`), and pushes follow every commit from now on. | `firmware/hub_infer/components/hub_display/hub_display.c`, `research/check_corners.py` (new), `research/measure_ramp.py` (new), `research/out/*` (previews regenerated), `DECISIONS.md` (D-57, I-39, I-40, O-3) | *Built (RAM 13.0 %, flash 62.4 %) and flashed; the radio is clean (410 frames parsed, 0 bad, 0 bad payload, node answering at rms 0.004-0.014). One scare in the middle: the first upload

attempts ran esptool against **the node's** port by mistake, which parked the C3 in its download mode and silenced the radio (0 frames for two minutes); `esptool run` on COM30 released it (I-40). The user has begun the Kaggle re-run; the retrained model is still outstanding.* |
557 | 2026-10-31 | **The colour-check page proved the hues are right and that this panel lifts the dark end - so the neutrals are now darker and teal-tinted, and the screen reads teal instead of blue.** The user pressed BOOT and photographed the swatches, and the page did exactly its job: **BLACK came back black, WHITE came back white, TEAL came back teal, GREEN green, AMBER amber and ROSE rose.** Every hue is right. But the fourth swatch - the old `(12,20,32)` navy - came back as a **saturated blue**, and the page's own background, the *same colour*, came back as a bright blue too, while the black swatch beside it stayed black. That is the panel lifting the dark end, and it lifts the **blue** channel hardest: the measured blue of the amber swatch (requested 36) came back at 145, about four times what was asked for. **The less blue the screen asks for, the less blue it gets**, so the neutrals were changed: the background is now `(7,15,16)` instead of `(12,20,32)`, the surfaces `(12,26,28)` and `(17,36,39)`, the border `(78,116,112)`, the dim text `(122,150,146)` - all darker and teal-tinted. The accent, green, amber and rose are untouched: what the screen *says* has not changed, only how much blue it asks for. Every contrast was re-checked: text on background 16.6:1, dim 6.1:1, accent 10.4:1, the card border 3.4:1 against the card it sits on. The colour check also grew a **dark ramp** - six steps of the same hue from black up - so the next photograph says in one row exactly how far up the dark end the glass lifts. | `firmware/hub_infer/components/hub_display/hub_display.c`, `research/ui_mock.py`, `DECISIONS.md` (D-56, I-38) | *Built (RAM 13.0 %, flash 62.4 %) and flashed; the radio is clean (110 frames off the air, 0 bad, 0 bad payload) with a node answering at rms ~0.005. The host-side previews are `research/out/ui_dashboard.png` and `research/out/ui_colour_check.png`. **What is needed now is one photograph of the colour check with the room lit** - in a dark room the phone opens up for the glowing panel, clips the highlights and drifts the white balance, which is why the swatches cannot be measured from the last photograph.* |
558 | 2026-10-31 | **The panel's colours were finally right, and the picture was still glaring: the backlight is now a brightness, and the BOOT button opens a colour check.** The second photograph of the panel settled the R/B swap: the `PHONE` square is now **warm orange** (it was blue), the accent underline under the header is **teal-green** (it was pink), and the `RADIO` square is cyan-green (it was olive) - all three confirmed against the *previous* photograph, which shows the same elements olive, pink and blue. **But the screen came out saturated blue overall, because the dark navy background was rendering as a bright blue.** The measurements say the panel is not doing that on its own: the old photograph, which was taken in a dark room and properly exposed, shows a near-black background rendering as a light pink-grey, and the new one is a close-up in a lit room whose highlights are clipped (the `RADIO` square measures (144,255,255) - fully blown out). A photograph of a glowing panel in a dark room is not a colour measurement: the camera opens up, the highlights clip and the white balance drifts with what it sees. So the fix is on the panel's side, not the palette's: **the backlight was a plain on/off pin at 100 %, and a TFT at full brightness in a dark room glares.** The pin is now a **LEDC PWM output** at 5 kHz with a default of **55 %** (`hub_display_set_brightness()`), and a **colour check page** - a grid of the palette's swatches, black and white at the ends of the first row, the alert colour last - opens on a press of the **BOOT button** (GPIO0). The swatches are the instrument: with them the panel's own colours can be read off a photograph without guessing, and the palette can be compensated if the panel's own black level really is lifted. | `firmware/hub_infer/components/hub_display/ili9341.c`, `ili9341.h`, `hub_display.c`, `include/hub_display.h`, `firmware/hub_infer/components/hub_display/CMakeLists.txt`, `research/ui_mock.py`, `research/measure_photo.py`, `research/compare_photo.py`, `DECISIONS.md` (D-55, I-37) | *Built (RAM 13.0 %, flash 62.4 %) and flashed: the panel comes up at 320x240 rotation 2 with `madctl 0x80`, and the radio is clean (109 frames off the air, 0 bad, 0 bad payload) with a node answering at rms ~0.01. `esp_driver_ledc` had to be added to the component's `REQUIRES`, and the CMake cache deleted for it to take. The host-side preview of the new page is `research/out/ui_colour_check.png`.* |
559 | 2026-10-31 | **The Kaggle download cell no longer assumes where the artifacts are.** The cell the user pasted failed with `FileNotFoundError: '/kaggle/working/out'`. The cause is in the notebook: its output directory is `OUT = Path(os.environ.get("CRYNET_OUT", "./out")).resolve()` - **relative to whatever the working directory was when the cell ran** - and the session that ran the notebook was not the session that looked for the result. The cell now (1) asks the notebook first, using the `OUT` variable that survives in the kernel if the export cell ran in this session, (2) otherwise searches `/kaggle/working`, the working directory, the home directory and `/tmp` for the artifacts **by name**, and (3) if it finds nothing, prints what *is* under `/kaggle/working` so the next session starts from facts instead of another guess. It still writes `/kaggle/working/crynet_out.zip` and offers it with `FileLink`, and the zip keeps the `out/` folder at the top of the archive. | `research/kaggle_download_cell.py` | *The same script is saved in the repo so it is not retyped. The retrained model is still outstanding: until the zip is unpacked, the deployed model false-alarms on 8.19 % of loud music.* |
560 | 2026-10-31 | **The footer's 'cots linked' count was counting the cards on the screen, not the cots answering - so the panel said '3 cots linked' above two cards that both read NO SIGNAL.** The count came from `st->node_count`, which is how many cot slots the hub has, so every card drawn was also a cot claimed as linked. It was a lie in the one place the caregiver is certain to look, and it was in the photograph the user sent: `3 cots linked` at the bottom left, `NO SIGNAL` on `COT 2` and `COT 3` right above it. The footer now counts the cots whose cards read ONLINE, and says **'no cots linked'** when none are answering. The host-side preview was changed the same way, so the two cannot drift. | `firmware/hub_infer/components/hub_display/hub_display.c`, `research/ui_mock.py`, `DECISIONS.md` (D-54, I-36) | *Built (RAM 12.9 %, flash 62.0 %) and flashed; the log still reads `rotation 2 (madctl 0x80)` and the radio is clean (553 frames off the air, 0 bad, 0 bad payload). The photograph that found this was worth taking on its own: one image caught the wrong colours, the wrong orientation and a lying footer.* |
561 | 2026-10-31 | **The panel's colours were arriving with red and blue exchanged - the teal theme was reaching the glass as olive and salmon. Fixed, flashed, and waiting on one more photograph.** The user's photograph of the panel was the first real look at the screen, and it confirmed the work of the last two sessions: **the layout, the text and the orientation are all right** - `CRYNET`, `S3 HUB`, `RADIO`, `PHONE`, `LISTENING`, `COT 1 / ONLINE`, `COT 2` and `COT 3 / NO SIGNAL`, `3 cots linked`, `up 0:39`. But the **colour was wrong**, and the user read it as a different, garish theme. The cause was not the palette: **the palette in the source was already the teal one.** The photo was *measured* rather than squinted at - the screen was located in the frame, mapped onto it, and the saturated pixels sampled (text and the 6x6 pill squares, which the palette pins down exactly): `CRYNET` (teal 45,212,191) measured **olive** (210,210,179); the `RADIO` square (green 52,211,153) measured **olive** (174,180,146); the `PHONE` square (amber 251,191,36) measured **blue** (5,175,247). Every hue was present and correct except that **red and blue were in the wrong places** - a textbook R/B swap. The reason is one bit: the driver's rotation setting carried `MADCTL_BGR`, which tells the controller that the *top* bits of each RGB565 word are blue, while `DISPLAY_RGB` puts **red** there. The bit had come from Adafruit's rotation-2 setting, which this panel's legacy firmware used. It is now cleared, and `INVOFF` is sent explicitly so the inversion state is pinned rather than inherited. | `firmware/hub_infer/components/hub_display/ili9341.c`, `DECISIONS.md` (D-53, I-35) | *Built (RAM 12.9 %, flash 62.0 %) and flashed: the panel comes up at 320x240 rotation 2 and the log now reads **`madctl 0x80`** - the `BGR` bit is gone - with the radio healthy (110 frames off the air, 0 bad, 0 bad payload). The host-side preview (`research/out/ui_dashboard.png`) was already correct, because it renders the palette in software: this bug lived only between the framebuffer and the glass. **One more photograph will settle it** - the teal should now appear as teal, the `PHONE` pill amber, and the background dark navy.* |
562 | 2026-10-31 | **The panel was showing hearts and diamonds instead of letters. The font was never corrupt — the generator was reading the table 32 characters out of place.** The screenshot of the new dashboard was full of little symbols where "CRYNET", "LISTENING" and "COT 1" should have been, and the host-side preview of the same screen showed the same nonsense, so the fault was visible before the hardware was. The cause: `Adafruit_GFX::drawChar` reads the classic font as **`font[c * 5 + i]` — the raw ASCII code, with no `- 0x20`** — because Adafruit's table begins with 32 decorative icons (hearts, diamonds) before the printable set at 0x20. The generator assumed the old layout and wrote `off = (code - FIRST) * 5`, so every character drew the glyph of the character 32 codes earlier: 'A' drew '@', a space drew the heart at 0x01. **The font file itself was fine** — the local `glcdfont.c`, the upstream copy and the cached copy are byte-identical, which is what finally killed the "the library's font is damaged" theory. Two more things fell out of the same fix: the renderer reads bit 0 as the top row, the way Adafruit's own `drawChar` does (`for (j = 0; j < 8; j++, line >>= 1) if (line & 1)`) — the *preview* had the opposite convention and was drawing every glyph upside down, which would have

hidden the fix behind a second bug; and the display source had been saved in Windows-1252 and then double-encoded into mojibake, so it is now valid UTF-8 with no non-ASCII byte inside a string literal. **The generator now detects the table's layout instead of assuming it**: it finds the classic exclamation mark (`00 00 5F 00 00`), works out the base from where it lands, and refuses to write the header unless the font is what it expects. | `research/build_font.py`, `firmware/hub_infer/components/hub_display/font5x7.h`, `firmware/hub_infer/components/hub_display/hub_display.c`, `research/ui_mock.py`, `research/toolchain/pio_build_font.log`, `research/hub_flash_font.log`, `DECISIONS.md` (I-33, I-34, D-51) | *Verified by decoding the generated header back into ASCII art — 'A', '0', 'g', '!' and 'R' all read correctly — and by rendering the preview, which now shows the real screen: `CRYNET`, `LISTENING`, `COT 1 / ONLINE`, `COT 3 / NO SIGNAL`. Built (RAM 12.9 %, flash 62.0 %) and flashed: the panel comes up at 320×240 rotation 2 and the radio hears the node with 0 bad frames. **One photograph is still needed to confirm the real glass.** |
563 | 2026-10-31 | **The panel now shows the caregiver's device, because the user asked for the old system's arrangement — "the status of all nodes and the wearable".** The header's `BLE` pill only said that the Bluetooth stack had started; it said nothing about whether anyone was there to hear the alert. It is now a **`PHONE` pill with three states**: green when a client is connected and subscribed, amber when the stack is up but nobody has attached yet, dim when Bluetooth is down. The wearable and the caregiver's phone connect to the *same* legacy GATT service, so the hub cannot tell them apart and does not pretend to — the pill says that a caregiver device is attached. And a **confirmed cry with nobody attached now reads `ALERT NOT DELIVERED`** under the banner, because a cry that reaches a hub with no listener has gone nowhere and the caregiver has to see that on the panel instead of discovering it by waiting for an alert that never comes. | `firmware/hub_infer/components/hub_display/include/hub_display.h`, `firmware/hub_infer/components/hub_display/hub_display.c`, `firmware/hub_infer/src/link_stage.c`, `research/ui_mock.py`, `DECISIONS.md` (D-52) | *The host-side preview (`research/ui_mock.py`) was rewritten to mirror the firmware page for page — it had drifted into a different design with a different name — so it is a true preview of the glass again, and the two cannot silently drift.* |
564 | 2026-10-31 | **The panel is alive, the picture is now the right way up, and the display has a face: an animated splash, then a dashboard.** Three things came out of the user's photograph of the panel. **(1) The screen was never broken — the *orientation* was.** The panel is a "landscape-first" ILI9341, and the legacy firmware drove it with **Adafruit's rotation 2 (`MADCTL_MY | BGR`) and a 320x240 logical framebuffer** (its `HubDisplay` class exists only to override Adafruit's 240 px clipping). My driver used rotation 1 (`MADCTL_MV`), which on this glass turns the whole picture a quarter turn — which is exactly what the photograph showed. The orientation now follows the legacy's proven setting, and the driver logs it (`rotation 2 (madctl 0x88)`). **(2) The UI was rebuilt.** The old one drew three strings and a bar on black; the new one composes everything in a **320x240 RGB565 framebuffer in PSRAM** (150 KB — the hub's internal RAM is not to be spared for it) and pushes one transfer per frame, so a frame is never torn. The screen now has **two faces**: an **animated splash** for the first 4.2 s — a scrolling waveform, the name, a progress bar and the boot steps — which then fades into the **dashboard**: a header with the radio and Bluetooth pills, a banner that says what the system is doing, a card per cot with its level, and the uptime. **(3) The palette was checked, not guessed.** The brief was "not too much red" (the legacy's dashboard was a wall of it), so the scheme is a dark navy with a teal accent, green for a cot that is fine, amber for a loud one, and **rose for a confirmed cry only** — and every colour is paired with a word (`ONLINE`, `NO SIGNAL`, `CRY 97%`), because colour on its own is not an accessible signal. Every pair was measured against WCAG: body text 15.9:1, dim text 5.5:1, the accent 10.1:1, the cry rose 6.3:1 — and the **card border failed at 1.40:1** (it was invisible), so it was lightened to (88,112,136), now 3.31:1. The system also gained a per-node display state: the dashboard shows every cot the hub has heard from, not just the loudest. | `firmware/hub_infer/components/hub_display/*`, `firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/components/ble_alert/*`, `research/toolchain/pio_build_ui.log`, `research/hub_flash3.log`, `research/hub_flash4.log`, `SYSTEM_UPDATE.md` | *The orientation is a property of this panel, not of the chip — the same class of fact as the pin assignments, and the same rule applies: when two systems disagree about a panel, the one that was photographed working wins. What is still open: one photograph of the new screen, to confirm it is the right way up and that the splash reads well.* |
565 | 2026-10-31 | **The TFT panel was wired, flashed and found to be showing noise — which turned out to be three bugs in my own ILI9341 driver, all now fixed.** The user wired the panel (CS 10, MOSI 11, SCK 12, DC 13, RST 14, backlight 4, 3V3, GND) and plugged in both boards; the hub was flashed and **the chain ran end to end on real audio**: the node's RMS climbed to 0.10-0.28, the hub pulled the window (**286/286 fragments, CRC-32 MATCH**), and the model called **P(cry) = 0.9961 on nine windows and 0.9883 on one** — `ALERT — cry`, `a cry on cot 1 at 100 %` over BLE — while the quiet room gave **P(cry) = 0.0000** and a non-cry loud sound at RMS 0.035-0.085 gave **P(cry) = 0.0000** (rejected, no false alarm). But the panel showed **static noise**, and the cause was mine, not the wiring: **(1) the address window was little-endian.** `CASET`/`PASET` take their four bytes big-endian; the code cast a `uint16_t` to `uint8_t *` and so sent the low byte first — 319 (0x013F) went out as 0x3F 0x01, which the panel read as column 16129. **(2) `CASET` was sent twice.** The window is one command for the columns and one for the rows; sending the column start and then the column *end* as a second `CASET` re-sets the start instead of the end, so the window collapsed to a single pixel. **(3) CS was released between a command and its parameters.** The ILI9341 resets its serial interface when CS goes high, so every parameter byte was arriving as a *command* — the panel was never initialised at all, and what the user was looking at was uninitialised display memory. The fix is in the driver: **D/C is now driven in the SPI driver's pre-transmission callback** (the only moment the level can safely change), **a command and its parameters are one transfer** via `SPI_TRANS_CS_KEEP_ACTIVE` with the bus acquired (the ESP-IDF only permits that on an acquired bus), and the address window is built big-endian, one command per axis. The text renderer was rewritten to draw **horizontal runs** instead of single pixels — about sixty times less SPI traffic, which is what makes a redraw cheap enough to run once a second. A **blue 12x12 marker now sits in the panel's top-right corner** so the layout's orientation can be fixed from one photograph. | `firmware/hub_infer/components/hub_display/{ili9341.c,ili9341.h,font5x7.h,hub_display.c}`, `research/toolchain/pio_build_display.log`, `research/hub_flash.log`, `research/hub_flash2.log`, `research/hub_live.log`, `SYSTEM_UPDATE.md` | *The pattern of the three bugs is worth remembering: the panel reported "up" the whole time, because nothing in the driver ever asked the panel whether it was up — the ILI9341's ID register can be read, but MISO is not wired, so it cannot be. **A display driver that cannot fail is a display driver that lies.** The marker square and the "is the text the right way up" question are how the remaining orientation question gets settled now.* The burst is now noticeably slower than in the last session — 5.7-7.6 s per window against 1.15 s — because it routinely stalls at 227-284 of 286 fragments and needs two or three resend rounds; that is the known BLE/Wi-Fi coexistence cost and it is the next thing to look at. |
566 | 2026-10-31 | **Music measured: the deployed model false-alarms on 8.19 % of loud music — the worst of any sound class, and a class it had never seen.** GTZAN (1,000 real music clips, ten genres, downloaded from Kaggle) was run through the deployed INT8 model with the notebook's own front end: **79 of the 965 clips that reach the model alarm — 8.19 %**, three times the rate for environmental sound (2.64 %) and five times the rate for speech (1.6 %). By genre: jazz 14.3 %, blues 13.0 %, pop 12.0 %, disco 10.0 %, rock 10.0 %, hiphop 9.0 %, metal 4.0 %, country 3.0 %, classical 2.9 %, reggae 2.0 %. The mechanism is plain once seen — a cry is a voiced, harmonic-rich sound, and a saxophone, a trumpet or a sung vowel looks like one in the log-Mel domain. **The threshold cannot rescue it:** the INT8 output saturates at P(cry) = 0.9961, so 45 of the 81 music alarms and 34 of the 36 true cry detections carry exactly the same value, and moving the threshold from 0.95 to 0.99 would remove 27 music alarms at the price of two real cry detections. A "two consecutive windows" confirmation was considered and deferred: music is continuous, so it would not help against music, and it would cost the demo a short single cry clip. The fix is the training round, and the corpus is in the notebook (see the row below). The practical consequence for the defence: **a room with music playing produces roughly one false alarm per twelve loud music windows** until the re-run happens — so the demo should be in a quiet room, or the music corpus should be trained in first. | `research/eval_music.py` (new), `research/out/eval_music.json`, `research/out/eval_music.log`, `DECISIONS.md` (I-29, D-51, D-52, O-8, O-9), `SYSTEM_UPDATE.md` | *This is the sharpest number in the project so far, and it is the one the user's stated requirement turns on: the system must tell a cry from music. Today it cannot, and the reason is structural rather than a tuning slip.* |
567 | 2026-10-31 | **Music goes into the training corpus, and the notebook that will be re-run on Kaggle is patched and validated locally.** The false-alarm map showed there is **no music anywhere in the training mix** (ICD: cries, VoxCeleb speech, DASEE domestic noise, human non-speech, cat meows, silence; supplementary: cries, ESC-50, snoring) and **no music in ESC-50 either** — so "the model

tells a cry from a song" was neither trained nor measured, which is exactly what a room with a radio or a television will contain. The patch, in `research/build_notebook.py` (never in the `.ipynb` by hand): **GTZAN** (Kaggle `carlthome/gtzan-genre-collection`, ten genres of real recorded music, ~1.2 GB) joins the corpora as an all-negative corpus; its clips are read with **random** windows because a thirty-second track is not one sound — the part that plays in the cot matters, not the average of the track; 75 % goes into training and 25 % into the held-out cross-corpus set, so the model is still measured on music it has never seen, and the cross-corpus breakdown now prints a row per genre. Two traps had to be handled: **GTZAN names every clip `00000.wav`..`00099.wav` in every genre folder**, so the reader's stem index would have silently kept one of the ten genres — `ZipReader` gained an `index_by_path()` and `build_corpus` falls back to it, which keeps all ten genres; and the class balance is untouched (music is a negative, and the weighted loss already handles the cry side). The local stub run proves the whole path: `cached gtzan 12.0 MB` → `GTZAN : 80 (music, all negative)` → `gtzan (160, 81, 64)` → `TRAIN (4923, 81, 64)` and `CROSS (1567, 81, 64) [cry4 + ESC-50 + snoring + music]`, and the training cells run. Also **fixed the display's choice of cot**: the cots are in one room, so all of them hear the cry, and the panel was naming the *first* crying cot it found in its array rather than the **loudest** one — the loudest is the cot the baby is in. The panel now names the loudest crying cot, and the BLE state characteristic's `crying_mask` still reports every cot that hears it. | `research/build_notebook.py`, `notebooks/crynet_training.ipynb` (regenerated), `research/make_stub.py`, `research/eval_music.py` (new), `firmware/hub_infer/src/link_stage.c`, `SYSTEM_UPDATE.md` | *The measured number for music is still to come: the real corpus is downloading, and `research/eval_music.py` will run the deployed INT8 model over it the way `eval_esc50.py` did. The Kaggle re-run itself is the user's to start; the notebook is ready for it.* |
568 | 2026-10-31 | **The false-alarm map: what the deployed model actually does when it is not a cry — and the discovery that the training corpus contains no music at all.** The user's requirement for the next phase is that the system must tell a cry from background noise, speech and music, and must say *which* cot the cry came from. The per-node side of that already exists (per-node inference, a per-node BLE alert, a `crying_mask` in the state characteristic) but has never been run with two nodes; the *discrimination* side had never been measured. `research/eval_esc50.py` (new) runs the **deployed INT8 model** — the very file flashed onto the hub — over ESC-50's 2,000 real-world clips, using the notebook's own frozen front end, and reports per category what the model said and whether the clip would even reach the classifier (the hub only pulls a window when the node's 100 ms RMS reaches 0.030). Result: **36 of the 40 real `crying_baby` clips alarm (90 %)** — the model does find real baby cries — and **38 of the 1,439 loud non-cry clips alarm (2.64 %)**. By kind of sound: urban 4.3 % (`car_horn` 6 of 38 reached, `hand_saw` 4 of 37, `church_bells` 1 of 39), animal 3.7 % (`crow` 4 of 32, `sheep` 2 of 33, `frog` 1 of 33), domestic 1.6 %, human non-speech 1.6 %, nature 0.8 %. The model's output saturates at P(cry) = 0.9961 (the INT8 output scale), so a single very confident window cannot be told apart from a single false alarm by confidence. **The structural finding: ESC-50 has no music and no speech** — it is environmental sound plus human non-speech vocalisations — and **no corpus in the training mix contains music**: ICD brings cry, VoxCeleb speech, DASEE domestic noise, human non-speech, cat meows and silence; the supplementary sets are cries, ESC-50 and snoring. So the model's behaviour on music was neither trained nor measured, which is exactly the case the demo room will contain. Music is now being pulled (GTZAN, from Kaggle `agrimamalviya/gtzan-from-desk` and the HF `Feanix/gtzan-5-sec` copy) so it can be measured now and added to the next training round. Also: **the two background shells left over from the last session were stopped** (they were only `grep`-polling two finished logs, and held no serial port), so the hub is free to disconnect; and **the TFT wiring was written down** for the user — CS 10, MOSI 11, SCK 12, DC 13, RST 14, backlight 4, VCC 3V3, GND, MISO unused, SPI2 at 40 MHz, landscape 320×240 — with the note that the driver never reads the panel's ID, so a miswire shows as a black screen and no warning in the log. | `research/eval_esc50.py` (new), `research/out/eval_esc50.json`, `research/out/eval_esc50.log`, `SYSTEM_UPDATE.md` | *Caveat, stated in the report itself: ESC-50's non-cry clips were part of the training corpus, so 2.64 % is optimistic for those categories — this is the category map, not the in-room number. The honest in-room number comes from a confusion session recorded with the INMP441 (speech, music, TV, clatter, and the phone-played cry), which is the next recording task. A "require two consecutive windows" rule was considered and deferred: a cry lasts seconds, but the demo plays a short clip through a phone, so one window may be all there is, and confidence saturates so it cannot be used as a tiebreak.* |
569 | 2026-10-31 | **The burst now completes and the whole system works end to end: the cry crosses the radio, reaches the classifier, and lands on the phone as an alert.** The blocker was that the hub's 286-fragment burst always came up short (267-274) and timed out. The cause was found by subtraction, not by reading: the node's own counters said 7,667 frames sent and **0 lost**, the hub's parse was clean, and building the hub **without BLE** produced **286/286 fragments in 3.75 s four times in a row** — so the loss was the hub's own Bluetooth advertising. BLE advertises on channels 37/38/39 and the hub's single radio must retune to reach them, during which the Wi-Fi receiver is deaf: about 6-9 % of the air, whatever the Wi-Fi channel and whatever the frame rate. That is physics, not a bug, so the fix had to be **recovery, not prevention**. **A new `RESEND` frame (type 5) carries a 36-byte bitmap of the fragments the hub is missing** — one bit per fragment, 54 bytes in all, so one request covers any scattered pattern — and the node re-sends exactly those out of the same frozen window with the same sequence number. Three further defects only the live test could find, each fixed and re-measured: **(1) the resend request is sent three times**, because an ESP-NOW broadcast is never acknowledged and the request goes out at exactly the moment the radio has been busiest (one lost request used to waste the whole round); **(2) the node answered a repeat of the request it was already serving**, so the hub's three copies made it send the same 128 fragments twice and the burst took 9.6 s instead of 5 — the node now answers a repeat only once, within 800 ms; **(3) the node refused the resend ("no frozen window for it")** because it froze the next window about three seconds after the upload, and the hub's request for the missing fragments often arrived just after that — so the node now **holds an uploaded window for four seconds** before freezing the next one, and the hub **abandons a burst the node can no longer complete** and asks for the newer window at once instead of waiting out the fifteen-second timeout. The stall detector was made proportional too: the wait before a burst counts as stalled grows with the number of fragments still missing, so the hub does not spend a round while the node is still sending a hundred fragments. **The final live test — hub, node, a real BLE client connected, the cry recording played through the PC's speakers — completed 14 bursts of 286/286 with the CRC-32 MATCHING the node's every time, zero timeouts, 9 alerts on the hub and 4 alert events on the phone (`ALERT {'cot': 1, 'alert': 'cry', 'confidence_pct': 100}`), and 6,327 frames off the air with 0 bad and 0 bad payloads.** The channel bugs found along the way are fixed too: the node was adding the hub peer on **channel 11 while its radio sat on channel 6**, and it was printing `WiFi.channel()` — the *scan result's* channel, which the Arduino core has no setter for — as the radio's channel. | `firmware/hub_infer/components/{crynet_proto,node_link}/*`, `firmware/hub_infer/src/link_stage.c`, `firmware/node/node_crynet/node_crynet.ino`, `DECISIONS.md` (new), `research/toolchain/{hub_cry_resend,hub_cry_ble_client,hub_cry_final2,node_console_live3,ble_client_final2}.log`, `SYSTEM_UPDATE.md` | **This is the first time the whole chain has run by itself: microphone -> node -> ESP-NOW -> hub -> TFLite -> the phone.** The burst takes 5.4-6.9 s (the burst itself is ~3.9 s, the resend rounds the rest); the whole path adds about a second of DSP and inference on top. The panel still needs eyes on the screen. **Also written this session: `DECISIONS.md`** — the project's report and documentation record: every decision (D-1 to D-50) with its reason, every issue (I-1 to I-28) with how it was diagnosed, what was done and what the result was, the measured results, and the open items. |
570 | 2026-10-31 | **The first live cry test since the rebuild, and three real bugs it found — one of them a silent killer.** The cry recording (`research/in_domain/session4/session.wav`, played through the PC's speakers) was fed to the real node while the hub listened: the node's RMS rose from the room's 0.002–0.012 to **0.057–0.082**, so the microphone and the trigger are fine — but the hub never asked for a window, and the chain stopped there. **(1) The node on the board was running stale firmware.** The dead session wrote the "freeze before you announce" change to `node_crynet.ino` but never reflashed the C3, so the node had no auto-freeze at all: `g_peak_pending` appears 0 times in the last committed node build and 7 times in the current source. Rebuilt and reflashed (978,133 B, 74 %). **(2) The node's freeze was cancelled in the same tick that created it.** `if (g_frozen_valid && (now - g_frozen_ms) > FROZEN_EXPIRE_MS)` used a `now` captured *before* `freeze_window()` ran, and `g_frozen_ms` is stamped *inside* it — so the subtraction underflowed to ~4 billion, the window expired the instant it was frozen, and the telemetry therefore never advertised `BURST_READY` or the window's CRC. The symptom was strange but unmistakable: the frozen seq advanced (1, 2, 3, 4, 5 …) while the flags stayed `0x06` and the CRC stayed `0x00000000`. Fixed by measuring the age from a fresh `millis()`. **(3) The hub was still in promiscuous Wi-Fi

mode** — the leftover diagnostic from the radio debugging — which hands the host every frame on the channel; with the laptop's own Wi-Fi on channel 1 that flooded the receive path. With it off, plus `ESP_WIFI_STATIC_RX_BUFFER_NUM` 16 → 32 and `DYNAMIC` 32 → 64 (affordable now that the RAM is free), and 3 ms of pacing between the node's fragments, the burst is the last thing standing between the system and a full alert. After (1) and (2) the chain runs for the first time end to end: the node freezes a window and advertises its CRC (`flags 0x07, crc 0x26DB0617`), the hub asks for it, and **267–274 of the 286 fragments arrive** — the burst then times out, which is what (3) is for. | `firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/components/node_link/node_link.c`, `firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/sdkconfig.defaults`, `research/toolchain/{hub_cry_test3.log,hub_cry_test4.log,node_stats.log,node_build*.log}`, `SYSTEM_UPDATE.md` | *The burst completing 286/286, the cry reaching the display and the BLE alert are still open.* The node's own numbers are healthy: 7,667 frames sent, **0 lost**, 2.3 ms average send wait, and the hub's parse is clean (0 bad frames, 0 bad payloads) — the fragments are lost between the two radios, not in either end's code. |
571 | 2026-10-31 | **Stage 2c's last half is finished and verified on the real hardware: the panel, the ESP-NOW radio and the BLE server now run together on the hub.** The dead session (the one that hit the 400 limit) had written the panel driver — `gen_font.py` regenerates `font5x7.h` (95 glyphs) from Adafruit's `glcdfont.c`, and `ili9341.c` drives the panel over SPI2 at 40 MHz, landscape 320×240 on the legacy wiring (CS 10, MOSI 11, SCK 12, DC 13, RST 14, backlight 4) — but the component's public half (`hub_display_start()`/`hub_display_update()`) was never written and nothing was wired to the build. This session added `hub_display.c` (a state snapshot behind a spinlock, redrawn by a low-priority task pinned to **core 1** so a 30 ms SPI burst never delays the link loop or the inference; changed lines only, and each line erases to the right edge so a shorter string cannot leave the tail of a longer one on the panel; the level bar is sqrt-scaled so a quiet room still shows a sliver and the 0.030 trigger lands about a third of the way up) and its `CMakeLists.txt`, and wired the component into `main.c`/`link_stage.c` (one snapshot per loop: the crying cot if there is one, else the loudest node). **The BLE/Wi-Fi conflict was not really an ordering problem — it was the hub's own memory.** The first fix attempt (radio before BLE) did move the failure to the other side: the radio came up perfectly, and then NimBLE **asserted** on `Failed to allocate memory for ble_store_config_vars` and reset the chip in a loop. The real cause was in the ELF: **154 KB of the hub's static RAM was boot-only buffers that never got handed back** — `main.c`'s `s_pcm` (64 KB) + `s_mel` (20.7 KB) + `s_q` (5.2 KB), and a 64 KB function-local static in `probes.c` that computes the four probe CRCs. They are needed only while the boot probes run, but as statics they held internal RAM for the life of the program — which is exactly the RAM NimBLE's store configuration wanted. They now live in PSRAM (borrowed and returned), and the probes' CRC buffer is freed after each use. **RAM went from 59.9 % (196,140 B) to 12.9 % (42,236 B).** With the RAM back, the whole stack fits: free internal heap 298,763 B at boot → 290,791 B after the probes → **215,155 B with the radio up → 149,271 B with Bluetooth up**. The init order is now radio → BLE → display, with `CONFIG_ESP_COEX_SW_COEXIST_ENABLE=y` so the two stacks share airtime. **Verified live on the boards:** the panel comes up (`the panel is up: 320 x 240, rotation 1`), the radio hears the node (`node 1 heard for the first time`), the BLE server advertises (`advertising as "INFANT_MONITOR_HUB"`), and a real BLE client **connected and subscribed while the radio kept running** — frames counted 21 → 308 over the session with **0 bad, 0 bad payloads, 0 failed requests**, no resets. The build also had to be taught three IDF 6 facts: the SPI/GPIO drivers live in `esp_driver_spi`/`esp_driver_gpio` (not `driver`), a stale CMake configure ignores a `REQUIRES` change until the build dir's CMake state is deleted, and `font5x7.h`'s rows need their own braces or `-Werror=missing-braces` stops the build. | `firmware/hub_infer/components/hub_display/*`, `firmware/hub_infer/src/{main.c,probes.c,link_stage.c,link_stage.h,CMakeLists.txt}`, `firmware/hub_infer/sdkconfig.defaults`, `research/toolchain/{gen_font.py,pio_build_2c*.log,hub_display_live3.log,hub_ble_coexist.log}`, `SYSTEM_UPDATE.md` | *What is still unverified: what the panel actually shows (it needs eyes on the screen), and a cry going all the way through to the display and the BLE alert.* |
572 | 2026-10-31 | **Stage 2c's BLE GATT server is written and verified from a real client — and the live test exposed that it cannot coexist with the ESP-NOW radio yet.** `components/ble_alert/` is a NimBLE peripheral: it advertises as `INFANT_MONITOR_HUB`, and serves the **legacy UUIDs** so the wearable firmware written against them keeps working — service `19b10000-e8f2-537e-4f6c-d104768a1214`, alert characteristic `19b10001-…` (read/notify/indicate, the legacy six-byte packet `cot id, alert type (1 = cry, 2 = cancel), confidence %, event id`), plus a new state characteristic `19b10002-…` carrying the whole system (`cots`, `crying_mask`, loudest confidence, how long the cry has run) so a phone that opens late still sees the current state. `link_stage.c` publishes on state *change* (cry starts / cry stops), not per window, and only after a cry is held past `CRY_HOLD_MS` so the gaps in a recording do not flap the alert. `firmware/tools/ble_alert_test.py` (a Bleak client) verified the server live: it found the hub, connected, discovered both characteristics and subscribed to both. **The failure the same test exposed:** with BLE up, `esp_wifi_init` fails (`esf_buf_setup_static: alloc eb fail(1)`) and the ESP-NOW radio never starts — BLE and the node link did not coexist in that build. Two demo tools came with it: `firmware/tools/serve_alert_page.py` serves `research/web/alert.html` over **HTTPS with a self-signed cert that has a SAN** (patch scripts `patch_serve_openssl.py`/`patch_serve_san.py`), because Web Bluetooth needs a secure context on a phone; the page talks to the hub directly over GATT in the browser. | `firmware/hub_infer/components/ble_alert/*`, `firmware/hub_infer/src/{main.c,link_stage.c,link_stage.h,CMakeLists.txt}`, `firmware/hub_infer/sdkconfig.defaults`, `firmware/tools/{ble_alert_test.py,serve_alert_page.py}`, `research/web/alert.html`, `research/toolchain/{ble_test.log,ble_test2.log,ble_test3.log,hub_ble_live.log,hub_ble_live2.log}`, `SYSTEM_UPDATE.md` | The BLE verification passed while the radio was down; the coexistence fix is the 2026-10-31 display entry above. The alert page has not been exercised against a real cry yet (`alert_page.log` is empty). |
573 | 2026-10-30 | **In-domain audio collected and the live system tested in the user's room. The deployed model works there — and the live test exposed a real defect in *which* window the hub classifies.** **Collection:** four sessions recorded with `node_capture` on the real INMP441 (`research/in_domain/session1..4`, WAVs gitignored, manifests kept): a 55 s rescued partial, a 299 s **quiet-room baseline** (110 negative clips, RMS 0.002-0.016), a 329 s **phone-speaker cry** session (87 cry clips, peak RMS 0.518) and a smoke test. The cry source is a **recording played through a phone speaker**, because the demo will be exactly that — so the deployed condition and the recorded condition are the same condition, and the clips are labelled as speaker-played rather than as a real cry in the room. **Evaluation of the shipped INT8 model on this audio** (`research/eval_indomain.py`, the notebook's own front end, so the numbers are what the hub would produce): on the room baseline **0 false alarms in 236 windows** and no room clip reaches the classifier through the 0.030 trigger; on the cry session **AUC 0.9283, recall 0.724, FPR 0.000** — the misses are 2-second windows that land in the gaps between cries. **Live test** (hub + node + phone, the real system): the radio was flawless — 5,976 frames, 0 bad, 0 bad payloads, 0 failed requests — and the model called the cry at **P(cry) = 0.9961** on five of sixteen pulls. The other three pulls said "no cry", and all three had the same cause: the hub triggers on the node's **100 ms** RMS but was served the **live 2-second** window, whose average is mostly the gaps between cries (trigger 0.0710 → window 0.0182 → no cry). **Three fixes, all now in:** (1) the node's `EVENT_RMS` 0.10 → 0.030, the same number as the hub's trigger, so the frozen window the protocol is built around actually exists — with 0.10 it existed in only 10 of 16 pulls, and the hub was served the live window instead; (2) the node now **freezes before it announces**, so the telemetry that says "I am loud" also carries the window's sequence number and its CRC-32; (3) the hub's trigger now needs the level to have been **sustained for two telemetry readings** (~200 ms), because the INMP441's slow DC drift produces one lone reading above the trigger on its own — in the quiet room the old trigger fired on the drift and pulled bursts continuously; the new one stays quiet. | `firmware/node/node_crynet/node_crynet.ino`, `firmware/hub_infer/src/link_stage.c`, `firmware/tools/label_session.py`, `firmware/tools/collect_clips.py`, `research/eval_indomain.py`, `SYSTEM_UPDATE.md` | The labeller cuts clips on a fixed grid and labels each by the share of its 100 ms records above the threshold (a cry, a no-cry, or uncertain), and falls back to computing the envelope from the audio itself when `records.csv` is missing — which is how an interrupted 55-second recording was rescued instead of thrown away. The node is back on `node_crynet` and both boards are flashed with the fixes. The distance curve (how far the phone can be) is still open: the phase cues did not line up, so the far-side measurement has to be repeated. Still to do for the demo: the TFT display and the BLE GATT server (the rest of Stage 2c), then the wearable (Stage 2d). |
574 | 2026-10-30 | **The in-domain collection session is prepared and the capture path is verified on the real node.** `node_capture` was compiled (308,982 B, 23%) and flashed to the node on COM30, and a 15-second smoke session recorded clean audio: 16

kHz mono 16-bit, 140 records, no CRC failures, room RMS **0.001-0.015 (mean 0.003)** — an order of magnitude below the hub's 0.030 trigger, so the room has clean headroom. Two tools were added for the session: `firmware/tools/label_session.py` cuts each 5-second clip from *inside* one region of the per-100 ms RMS trace (a fixed cut can straddle the start of a cry and be neither class), labels it from that trace, and writes a manifest plus a per-10-second timeline so the levels can be checked without listening; `collect_clips.py` no longer demands `--out` when `--list` is used. The recorded WAVs are gitignored (`research/in_domain/**/*.wav`) — they are large; the manifests are kept. | `firmware/tools/label_session.py`, `firmware/tools/collect_clips.py`, `.gitignore`, `SYSTEM_UPDATE.md` | The node is flashed with `node_capture` for the collection, so the radio path is dormant until `node_crynet` goes back on it. The cry source available today is a **recording played through a laptop speaker**, not a real cry — that is a proxy (compressed, band-limited, EQ'd by the speaker) and the dataset must be labelled as such; the negatives it produces (this room, this microphone) are authentic. |
575 | 2026-10-30 | **Stage 2c's radio committed as a checkpoint** (`git add -A && git commit && git push crynet main`, the project's backup rule). The commit carries the whole radio half: `crynet_proto` (the wire format), `node_link` (reception, reassembly, CRC-32), `src/link_stage.c` (the trigger, the request, the analysis), the node's transmit path and the `node_console.py` tool. Two tidy-ups were made immediately before committing: the hub's reject diagnostics were re-worded from "TEMP DEBUG" to permanent diagnostics (the first few rejects still print — that log line is what finally located the window-buffer bug), and a slot with no window buffer now logs an error instead of being dropped silently; `node_radio.ino`'s header now says what the investigation actually found. The hub was rebuilt after the edits: **SUCCESS**, 49.7% flash, 59.0% RAM. | `firmware/hub_infer/components/node_link/node_link.c`, `firmware/node/node_radio/node_radio.ino`, `SYSTEM_UPDATE.md` | The radio is verified on the real boards, so this checkpoint is a *working* state, not a work in progress. Still outstanding and not blocking: the node sketch carries status-only timing instrumentation, and there is no per-fragment retry (one burst in ten timed out at 277/286). |
576 | 2026-10-30 | **Stage 2c's radio verified end to end on the real boards: the hub pulls a two-second window over ESP-NOW and its CRC-32 matches the node's.** `node 1: 286/286 fragments in 1.15 s` -> `window crc32 0x23C2EE17, the node advertised 0x23C2EE17 -> MATCH` -> front end 47.5 ms, inference 102.4 ms, P(no_cry)=0.996 on the reassembled audio, `no cry`. Repeated bursts at **0 bad frames, 0 bad payloads** (a noisy room at RMS 0.06 kept the 0.03 trigger firing, burst after burst). Four real bugs were found and fixed on the way: **(1) A closed serial port starved the node's loop — the root cause of the week.** ESP32 Arduino's USB-CDC (HWCDC) write path waits `tx_timeout_ms` (default 100) x up to 20 consecutive timeouts = ~2 s **per print** when no host reads the port, and `usb_serial_jtag_is_connected()` stays true whenever the cable is plugged in, so `if (Serial)` gates did NOT help. The node prints its rms every 100 ms, so an unattended node crawled to ~3 loops/s: the mic starved, telemetry fell to 0.35 Hz, the window never filled, and the "INMP441 power-on transient" seen in hub captures was stale ring-buffer data, not the microphone. Cure: **`Serial.setTxTimeoutMs(0)`** in setup() — prints drop instead of block; telemetry hit 10 Hz within one flash. (The earlier sessions only worked because the Arduino IDE serial monitor held the port open.) **(2) `handle_telemetry` memset a new node's slot to zero, wiping the 64 KB window pointer `node_link_create` had pre-allocated — every fragment was rejected on `window == NULL` and not one burst ever reassembled.** The stats line decoded it: 702 raw, 702 parsed OK, 560 bad = parse passed, `handle_fragment` rejected; the only silent reject was the window pointer. Fix: save/restore the pointer across the memset. **(3) The hub's 2-second probe had become a 50 ms flood** (the heartbeat edit had deleted `last_probe_us = probe_now`), and the node wiped its whole live window after serving a one-fragment probe — fix restores the timestamp and the node only wipes after a real burst. **(4) `esp_now_add_peer` was called from the node's receive callback** (WiFi-task context — the hub's own `node_link.c` documents this exact trap): peer-add moved into `loop()`. Along the way: **unicast to the learned hub address was tried and rolled back** — it wedged the ESP-NOW send path (`esp_now_send` returned OK, frames never left, the send callback never fired, every send burned its wait cap and the loop starved); broadcast at 10 Hz is reliable now that the loop runs real time, which also disproved the "hub radio sleeps" theory, so **the 100 ms heartbeat was removed** (`WIFI_PS_NONE` plus constant node traffic is why the hub hears everything). Known limitation, recorded not fixed: one burst timed out at 277/286 (broadcast loses a few frames under load; no per-fragment retry yet). The node sketch still carries status-only timing instrumentation (pump/loop/esp_now_send timers, printed only on the `status` command) — strip when convenient. | `firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/src/link_stage.h`, `firmware/hub_infer/components/node_link/node_link.c`, `firmware/node/node_crynet/node_crynet.ino`, `SYSTEM_UPDATE.md` | Every step's log is in `research/toolchain/hub_*.log` (heartbeat_test, window_pull_test, frag_debug*, unicast_test, dual_test, fixed_test, calm_test, txtimeout_test, burst_fixed_test, final_test) and the patch scripts that produced each fix are `research/toolchain/patch_*.py`. Reads that paid off: the Arduino core's `HWCDC.cpp` (the 100 ms x 20 timeout ladder) and the hub's own comment about peer-add from the WiFi task. |
577 | 2026-10-30 | **Stage 2c begins: the hub's half of the node link is written, and the node's radio is fixed.** New ESP-IDF components under `firmware/hub_infer/components/`: **`crynet_proto`** (the hub's copy of the node's wire format — frame layout, CRC-16/CCITT-FALSE, CRC-32, build/parse) and **`node_link`** (ESP-NOW reception, per-node telemetry, burst reassembly into one 32000-sample window with every fragment's CRC checked, and the request frame). `src/link_stage.c` drives it: the hub asks for a window once a node's RMS reaches **0.03**, pulls the two seconds over ESP-NOW and runs it through the Stage 2a front end and model, which are still verified on every boot before the radio starts. Two node-side changes make the link honest: **(1)** the node now waits for the previous frame's ESP-NOW send callback before queueing the next one — the transmit queue is ten frames deep, so a 286-fragment burst sent back to back overflowed it and every frame past the tenth was lost for good; **(2)** the node computes a **CRC-32 of its frozen window** and advertises it in the telemetry, so the hub can prove that the window it reassembled from the radio is byte-for-byte the window the node froze. New tool `firmware/tools/node_console.py` drives the node's serial console (`status`, `freeze`, `stats`). | `firmware/hub_infer/components/crynet_proto/*`, `firmware/hub_infer/components/node_link/*`, `firmware/hub_infer/src/link_stage.c`, `firmware/hub_infer/src/link_stage.h`, `firmware/hub_infer/src/main.c`, `firmware/hub_infer/src/CMakeLists.txt`, `firmware/node/node_crynet/node_crynet.ino`, `firmware/tools/node_console.py`, `SYSTEM_UPDATE.md` | The telemetry payload grew from 8 to 12 bytes (the CRC-32 was appended) and the node's request handling is unchanged, so the two sides were changed together. The hub's 0.03 trigger is the frozen decision from the rebuild; the older firmware pulled audio at 0.18 and a soft cry never reached the classifier at all. The hub's window buffers are 64 KB each in PSRAM, one per node. |
578 | 2026-10-30 | **Both node sketches now discard the INMP441's power-on burst: a 500 ms warm-up after `begin()`.** Verifying the capture sketch after the rewire showed real, clean audio (9,237 distinct values, room RMS 6-11 milli, no clipping) — but its first 100 ms record read **0.656 of full scale**, decaying 656 → 79 → 41 → 22 → 12 milli over ~500 ms before settling at the room level. That is the INMP441's power-on transient, and it mattered twice over: clips are cut from t=0, so `clip_0000` of every session contained the burst, and on the deployed node the burst would freeze a garbage window the moment it crossed the 0.10 EVENT level and hand the hub a spurious upload after every boot or reconnect. Both sketches now read and discard 500 ms (8,000 stereo frames) right after the I2S starts. Verified on hardware: `node_crynet` boots and streams telemetry with a first RMS of 0.047 (never crossing EVENT), and a fresh `node_capture` session reads **7 milli on its very first record** — room level from sample 0, versus 656 before. | `firmware/node/node_crynet/node_crynet.ino`, `firmware/node/node_capture/node_capture.ino`, `SYSTEM_UPDATE.md` | The 500 ms of discarded audio simply delays the first telemetry/record; nothing else changes. The slower DC drift after power-up (minutes of settling from −1639 to −40) was already known and is absorbed by the per-window mean. |
579 | 2026-10-30 | **The node produces real audio: the rewired INMP441 works and the two sketches were settled with a measurement.** After the user rewired the board, `node_diag` shows the left I2S slot carrying real 24-bit samples (`EC4A5800`, low byte zero — the left-justified signature) with the right slot constant, and the pin scan identifies **SD on GPIO6 as the only live data pin**, so the wiring is SCK=4 WS=5 SD=6 as designed. `verify_burst.py` then reports **PASS, 286/286 fragments, 0 bad CRC**, and the WAV it wrote is a *real* waveform: 1,943 distinct values, RMS 349 counts (0.011 of full scale), peak ±1,150 — a quiet room, no clipping, sitting below the hub's 0.03 trigger and above its 0.0015 silence gate. The forked thread had left the two sketches contradicting each other (`node_crynet` MONO+SLOT_LEFT, `node_capture` STEREO+SLOT_LEFT); the diagnostic settles it: **STEREO with both slots, taking every other word.** MONO halves the I2S frame to 32 bits, which breaks the INMP441's framing — the same microphone read the same way gives −3500 counts of DC with the full frame and −620 with the halved one. Asking for stereo also makes the Arduino library force the both-

slots mask whatever mask is passed, so the mask argument never does what it appears to do. Both sketches now do the same thing and both compile with zero warnings. Also recorded: the INMP441's DC output drifts for some seconds after power-up (−700 settling to −50), which the hub's per-window mean removal absorbs. | `firmware/node/node_crynet/node_crynet.ino`, `firmware/node/node_capture/node_capture.ino`, `firmware/node/README.md`, `SYSTEM_UPDATE.md` | The forked thread's other changes were kept: both host tools now drain the serial port without ever waiting for a fixed block, because the node's small CDC transmit buffer overflows and loses bytes otherwise. |
580 | 2026-10-30 | **The node firmware runs on the real C3 and the whole capture path verifies end to end — but the microphone is not driving its data line.** `node_crynet` was flashed to the user's WeAct Studio ESP32-C3 SuperMini (COM30) and **`verify_burst.py` reports PASS: 286/286 fragments, 0 bad CRC, exactly 64,000 bytes**, written out as a 2.00 s WAV. So the I2S peripheral, the circular window, the framing and the CRCs are all proven on the device. **The audio itself is not real:** the captured window is flat at 0–1 count with 533 single-sample spikes, and the diagnostic (`node_diag`, a new bring-up sketch that reads both I2S slots and scans every exposed GPIO) shows the data line reading **exactly 0.00 in steady state**. The apparent "signal" on GPIO0 and GPIO6 is the peripheral's **start-up transient** — a repeat measurement of the identical configuration reads 0.00/0.00 where the first read 3961/2320, which is why the tool now measures the original pins twice and says whether the two agree. The SCK scan reads 0.00 on **every** pin, which is the clearest statement of the fault: with the clock on the wrong pin the microphone is never clocked, and with the clock on the right pin its data line still stays flat. **So the fault is in the wiring, not the firmware**: the INMP441 only drives SD when it is powered, clocked and given a valid word-select. | `firmware/node/node_diag/node_diag.ino`, `firmware/node/README.md`, `SYSTEM_UPDATE.md` | The user's Arduino IDE build failed with `cc1plus.exe: out of memory allocating 4194296 bytes`: the machine's **page file is disabled**, so the commit limit was 16,190 MB against 15,109 MB committed. `AutomaticManagedPagefile` was set to `True` (the Windows default), which takes effect at the next boot; meanwhile the sketches were built and flashed with `arduino-cli` outside the IDE. **The user's PowerShell `python` is 3.15.2 at `C:\Python315`, which had no pyserial** — it is now installed there so `python firmware/tools/...` works from any directory. |
581 | 2026-10-30 | **The node sketches are in the Arduino sketchbook and build from there.** `install_sketches.py` had been run from the wrong directory (the user was in `firmware/hub_infer`, so the relative path did not resolve), and their PowerShell `python` is **3.15.2** at `C:\Python315`, which had no **pyserial** — the node tools need it. pyserial 3.5 is now installed for 3.15, so `python firmware/tools/...` works with whichever interpreter is first on PATH; the 3.11 that has PlatformIO is still the one for the hub CLI. Both sketches are installed at `~/Documents/Arduino/node_crynet` and `~/Documents/Arduino/node_capture`, and **both were compiled from the sketchbook copies** for `weact_studio_esp32c3` with zero warnings, so the IDE will find and build them. | `firmware/tools/install_sketches.py`, `SYSTEM_UPDATE.md` | For the hub from a terminal, the cleanest command is the standalone `C:\Users\HP\.platformio\penv\Scripts\platformio.exe` — the same Core 6.2.0 the IDE drives, and it needs no Python on PATH. PlatformIO was deliberately **not** installed into 3.15: Core 6.2.0 predates that release. |
582 | 2026-10-30 | **The PlatformIO IDE works now, and the node's real hardware is identified.** (1) The VS Code extension refused to start with *"Could not find compatible PlatformIO Core"* — caused by my own setting. The extension's check is `get-platformio.py check core --version-spec ">=6.1.6"`, and with `useBuiltinPIOCore: false` it looked for a Core in `~/.platformio/penv/Scripts` and found none. The fix: `useBuiltinPIOCore: true` (the extension's own advice) and the Core installed into the penv by PlatformIO's own bootstrap script. The penv Core is **6.2.0** — the same version the command line uses — it shares `~/.platformio`, so ESP-IDF 6.1.0 and the Xtensa toolchain were reused and **nothing was re-downloaded**; the hub builds through it in **99.3 s**, RAM 52.0 %, flash 22.0 %. (2) The user's node is a **WeAct Studio ESP32-C3 SuperMini**, and its USB is the **ESP32-C3's own native USB** (`303a:1001` on **COM30**), *not* a CH340 — so UART0 on GPIO20/21 goes nowhere and **"USB CDC On Boot" must be Enabled** or the console is silent and the port vanishes when the app starts. The Arduino core has an exact board entry for it (`weact_studio_esp32c3`) with CDC-on-boot already on; both sketches were recompiled for it with zero warnings. (3) The **capture records now carry a CRC-16** over their header and payload, and the collector drops any record that fails it — without it a single dropped byte on the USB link would shift every later sample and quietly ruin the dataset. Both stream parsers were rewritten as testable classes and checked against synthetic streams: the burst parser finds 285 of 286 fragments and flags exactly the corrupted one; the capture parser accepts 19 of 20 records and rejects exactly the corrupted one, with every surviving payload byte-exact. | `.vscode/settings.json`, `firmware/hub_infer/.vscode/settings.json`, `firmware/node/node_capture/node_capture.ino`, `firmware/node/README.md`, `firmware/tools/collect_clips.py`, `firmware/tools/verify_burst.py`, `SYSTEM_UPDATE.md` | The tools also had to stop depending on the sketch's text banner: a USB CDC port can enumerate *after* the sketch has already printed it, so the parsers resynchronise on their own magic and the banner is only informational. `verify_burst.py` now also retries the `freeze` command, because the node refuses to freeze until its two-second window is full. |
583 | 2026-10-30 | **Node window bug found by reading the code before the hardware did.** After a burst is sent the live window is dropped and refills, so for two seconds the newest samples are not at the start of the array — but the mean summed from index 0, averaging the stale tail with the fresh head and making the reported RMS meaningless for exactly as long as the window was invalid. The valid region is now walked **backwards from the write cursor**, which is right whether the window is full or refilling, the cursor is no longer reset, and a request arriving while the window is incomplete is refused rather than answered with a window whose order is unknown. | `firmware/node/node_crynet/node_crynet.ino`, `firmware/node/README.md` | Both sketches recompiled clean. The wire format was also validated numerically: CRC-16/CCITT-FALSE matches the standard check vector `0x29B1`, and the C frame builder and the Python parser agree on every header offset and both CRCs. |
584 | 2026-10-30 | **Stage 2b: the node firmware is written and compiles, and the offline GUI path is set up.** Two Arduino sketches for the ESP32-C3 — `node_crynet` (I2S capture, a 2 s rolling window, 100 ms RMS telemetry over ESP-NOW, the 64 KB burst as 286 CRC-framed fragments, a serial console) and `node_capture` (streams the microphone to the PC for in-domain data). Both build for `esp32:esp32:esp32c3` against the installed core 3.3.12 with **zero warnings**: 977,617 B / 165,972 B and 311,120 B / 18,592 B. The I2S setup follows Espressif's own recipe for this microphone — **32-bit slots** (the INMP441 puts 24 bits in a 64-bit frame, so 16-bit slots would misalign the data), `I2S_SLOT_MODE_MONO` + `I2S_STD_SLOT_LEFT`, narrowed to int16 in software by keeping the top 16 bits; the C3 is I2S hardware v2, so the left slot is chosen in hardware. The node repeats the hub's DC lesson: an exact **integer** mean, and the 100 ms RMS computed with the mean of the **whole 2 s window** removed, so the node's RMS is the same quantity the hub's front end will compute — otherwise the hub's 0.03 trigger threshold means nothing. Two host tools came with it: `verify_burst.py` (asks the node for its window over the serial console, checks every frame's CRC, writes the WAV) and `collect_clips.py` (records a session and cuts it into 5 s clips), plus `install_sketches.py` to copy the sketches into the Arduino sketchbook. Separately, the **PlatformIO IDE in VS Code** was pointed at the pip-installed Core 6.2.0 (`useBuiltinPython: false`, `useBuiltinPIOCore: false`, `customPATH` at the Python `Scripts` folder) so it drives the toolchain that is already installed and needs no network. Also corrected a claim: the build is DIO throughout (`CONFIG_ESPTOOLPY_FLASHMODE "dio"`, flash-mode byte `0x02` in both images), so the application runs in DIO — the bootloader does *not* switch it back to QIO. | `firmware/node/node_crynet/node_crynet.ino`, `firmware/node/node_capture/node_capture.ino`, `firmware/node/README.md`, `firmware/tools/verify_burst.py`, `firmware/tools/collect_clips.py`, `firmware/tools/install_sketches.py`, `firmware/README.md`, `.vscode/settings.json`, `firmware/hub_infer/.vscode/settings.json`, `SYSTEM_UPDATE.md` | One real API error was caught by compiling: IDF 5.x/6.x changed `esp_now_recv_cb_t` to take an `esp_now_recv_info_t *` first argument. **The node has still never run on the board** — the wiring, the INMP441's actual level and DC offset, and the effective sample rate are all unmeasured. |
585 | 2026-10-30 | **Stage 2a closed: the firmware runs on the real chip and all four probes pass.** The user's ESP32-S3-DevKitC-1 (COM32) now reports `RESULT: PASS — 4 probes, front end and model agree with the host`, with the device's quantised features matching the host reference exactly on three probes and to within 1 LSB on three values of the fourth. Measured on the chip: PSRAM 8 MB present, flash 16 MB, front-end working set 5,380 B in internal SRAM, tensor arena 262,144 B reserved / **108,328 B used**, front end **41.7 ms**, inference **101.8 ms**. Five defects were only findable by running it: the **DC-removal mean accumulated in float32** (32,000 samples of 18,000 summed to 575,726,660 instead of 576,000,000, so 8.5 counts of DC survived and the per-window standardisation turned them into a completely different feature vector — the DC-offset probe failed with a max delta of 131; the sum

is now an exact `int32`); the build was at **`-Og`** because `-DCOMPILER_OPTIMIZATION_PERF` is an Arduino macro PlatformIO's ESP-IDF builder ignores (`CONFIG_COMPILER_OPTIMIZATION_PERF=y` now); the CPU ran at **160 MHz** (`CONFIG_ESP_DEFAULT_CPU_FREQ_MHZ_240=y`); the front end kept a **128 KB padded copy of the signal in PSRAM**, which is uncached external memory on the S3 and was streamed through 81 times per window (the copy is gone — each frame is built straight from the caller's int16 PCM — and the working set is 5.4 KB in internal SRAM, 135 ms → 41.7 ms); and a **combined flash image built with QIO flash mode makes the chip watchdog-reset in a loop** because the ROM cannot read this board's flash in QIO before the bootloader configures its own timing (the combined image must use DIO; the bootloader switches to QIO itself afterwards). Added `firmware/tools/flash_hub.py` (finds the board, flashes it, prints the log — nothing to download), `firmware/tools/capture_serial.py` and `firmware/README.md` documenting the build and the three offline flash routes. | `firmware/hub_infer/components/crynet_dsp/*`, `firmware/hub_infer/sdkconfig.defaults`, `firmware/hub_infer/platformio.ini`, `firmware/tools/flash_hub.py`, `firmware/tools/capture_serial.py`, `firmware/README.md`, `SYSTEM_UPDATE.md` | ESP-NN's SIMD kernels are linked in (`esp_nn_conv_s8_esp32s3`, `esp_nn_conv_s8_1x1`, `esp_nn_conv_s8_3x3_opt`). The FPU **is** used (`madd.s` in the disassembly) — an earlier check that appeared to show software float was run against the wrong compiler binary. The inference at 101.8 ms is 5 % of the 2 s window and acceptable for a triggered monitor, but the arena is still in PSRAM. |
586 | 2026-10-30 | **The Stage 2a firmware builds.** `firmware/hub_infer` compiles and links for the ESP32-S3: `firmware.bin` 468,800 B, RAM 52.1 %, flash 22.3 %, ESP-NN enabled, octal PSRAM at 80 MHz. Getting there meant fixing six separate things: (1) PlatformIO 6.2 installs a URL-pinned `framework-espidf` as if it were an **npm package**, so it read the wrong `package.json` and left `packages/framework-espidf` holding only a test fixture — CMake then failed with *"include could not find requested file: tools/cmake/project.cmake"*; fixed by installing the platform's own framework (**ESP-IDF 6.1.0**) by hand and dropping the `platform_packages` pin; (2) the top-level `CMakeLists.txt` **assigned** `EXTRA_COMPONENT_DIRS`, shadowing the `-DEXTRA_COMPONENT_DIRS:PATH=<src>` that PlatformIO passes for the application component, and pointed at a component instead of a directory that contains components; (3) the vendored components sat **outside** the project directory, and PlatformIO re-compiles IDF components with SCons using the *source basename* for object names — the TFLite Micro port has twelve basename collisions, so the build died with *"Multiple ways to build the same target were specified for …/circular_buffer.cc.o"*; moving the vendored components into `firmware/hub_infer/components/` makes the paths relative and distinct; (4) the port was wired to the registry copy of `esp-nn` while `esp-nn` was also vendored locally — a duplicate-component error — so `firmware/tools/patch_deps.py` now drops the registry dependency and adds `esp-nn` to the port's own requirements, keeping the build offline and pinned; (5) IDF 6.x moved `esp_flash.h` into the `spi_flash` component, changed `esp_flash_get_size()` to take a chip handle and an out-parameter, and requires an explicit capability argument on `heap_caps_calloc`; (6) the model copy is now aligned to 16 bytes by hand, because `heap_caps_malloc` only guarantees 4-byte alignment and the FlatBuffer verifier wants 16. | `firmware/hub_infer/*`, `firmware/tools/patch_deps.py`, `firmware/tools/fetch_deps.sh`, `.gitignore` | **`CONFIG_DDSP` no longer exists in IDF 6.x** and was dropped from `sdkconfig.defaults`. **The firmware has still never run on hardware** — there is no serial log yet, so the FlatBuffer alignment, the arena fit, the probe feature agreement and the latency are all still unverified. |
587 | 2026-10-30 | **The build finally configures — four real bugs found and fixed.** Diagnosis: PlatformIO 6.2 installs a URL-pinned `framework-espidf` by treating the archive as an npm package, so it read the wrong `package.json` out of the tree and left `packages/framework-espidf` holding only a test-fixture package — hence CMake's *"include could not find requested file: tools/cmake/project.cmake"*. Fixed by installing the platform's own framework by hand (**ESP-IDF 6.1.0**, `platform-espressif32` 8.0.0) and dropping the `platform_packages` URL pin. Also fixed: (1) the top-level `CMakeLists.txt` **assigned** `EXTRA_COMPONENT_DIRS`, shadowing the `-DEXTRA_COMPONENT_DIRS:PATH=<src>` that PlatformIO passes for the application component — it now *appends*, and points at `third_party/` (a directory that **contains** components) rather than at the component itself; (2) the vendored TFLM port was wired to the registry copy of `esp-nn` while `esp-nn` was also vendored locally — a duplicate-component error; `firmware/tools/patch_deps.py` now removes the registry dependency and adds `esp-nn` to the port's own requirements, keeping the build **offline and pinned**; (3) IDF 6.x moved `esp_flash.h` into the `spi_flash` component and changed `esp_flash_get_size()` to take a chip handle and an out-parameter; (4) `CONFIG_DDSP` no longer exists in IDF 6.x and was dropped from `sdkconfig.defaults`. | `firmware/hub_infer/CMakeLists.txt`, `firmware/hub_infer/platformio.ini`, `firmware/hub_infer/sdkconfig.defaults`, `firmware/hub_infer/src/main.c`, `firmware/hub_infer/src/CMakeLists.txt`, `firmware/tools/patch_deps.py`, `firmware/tools/fetch_deps.sh` | IDF 6.1 gives every component `freertos esp_hw_support heap log soc hal esp_rom esp_common esp_system` automatically, so the earlier `esp_heap_caps.h` ownership worry is moot. ESP-NN is worth keeping: on an S3 it is the difference between ~40 ms and ~2.3 s per convnet invoke. **The firmware has still never been compiled or run.** |---|
588 | 2026-10-29 | **Toolchain recovery documented.** The first ESP-IDF 5.3.2 install was killed mid-unpack by the background task's 30-minute limit, leaving a partial `framework-espidf` tree (no `tools/`, `main/`, `examples/`, `version.txt`). The partial tree was deleted and the build restarted; the 933 MB archive was already cached in `~/.platformio/.cache/downloads/`, so **no re-download was needed**. §4.3 now records the pin and a 3-step idempotent recovery. | `SYSTEM_UPDATE.md` | Restarted as background process `blewy4qwx` with a 2-hour limit. Still **NOT yet compiled**. |
589 | 2026-10-29 | **INT8-vs-float parity gap closed.** `research/verify_int8_parity.py` rebuilt the ICD validation fold (fold 4, 5,693 clips → 11,386 centre-cropped windows) from the cached archives with the notebook's own front end and ran the exported INT8 model and the float32 Keras model over it. **INT8 AUC 0.997926 vs float32 0.999299, ΔAUC −0.001373, decision agreement 99.956 %**; recall 0.9780 vs 0.9782 and FPR 0.0036 vs 0.0036 at the 0.95 threshold. The independently computed float AUC corroborates the Kaggle figure. Report: `research/out/verify_int8_parity.json`. | `research/verify_int8_parity.py`, `research/out/verify_int8_parity.json`, `SYSTEM_UPDATE.md` | Root cause of the NaN documented: the parity cell took a *prefix* of a label-sorted array, so the slice was 100 % cry and its AUC was undefined. The model was never affected. |
590 | 2026-10-29 | **Full Kaggle outputs received and staged.** `research/out/` contains the full run. Model `conv_cnn`, 137,384 bytes, SHA-256 `4aef55d8…126bc11`, threshold 0.95, input scale 0.060125612 / zp −4. `firmware/tools/check_artifact_drift.py` verifies the outputs and the staged copies against `deploy_config.json` — **PASS**. On-device probe references generated by `firmware/tools/gen_probe_refs.py` from the notebook's own front end. | `research/out/*`, `firmware/hub_infer/components/*`, `firmware/tools/*` | Val AUC 0.9994, test AUC 0.9995, cross AUC 0.9973. C/Python parity 6.905 × 10^{-5}. **Gap:** `int8_val_auc` and `float_val_auc` are NaN in the full-run config; parity needs to be re-measured. |
591 | 2026-10-29 | **Stage 2a (hub inference core) scaffolded.** New ESP-IDF project with `crynet_model`, `crynet_dsp` (PSRAM-aware, reference FFT), `crynet_infer` (TFLM INT8), and the four on-device verification probes. PlatformIO espressif32 toolchain download/build in progress (`research/toolchain/pio_build.log`). | `firmware/hub_infer/*`, `firmware/tools/*` | **NOT yet compiled or run on hardware.** The user has not connected the board. TFLM port is checked out under `firmware/third_party/esp-tflite-micro` for testing but is not yet vendored/pinned; dependency wiring still needs completion. |
592 | 2026-10-29 | **Checkpoint-2 plan written into §4** — where to place the Kaggle `out/` (`research/out/`, now un-gitignored so the trained weights are versioned), the "firmware written from scratch" contract (§4.1: re-derived vs deliberately kept), the 4-stage plan 2a→2d (§4.2), and the toolchain status (§4.3). | `SYSTEM_UPDATE.md`, `.gitignore` | PlatformIO 6.2.0 installed via pip so the firmware can be compiled and checked here — this machine had no ESP-IDF, PlatformIO, Arduino IDE or Xtensa compiler. |
593 | 2026-10-29 | **CryNet repo made public** (was private) at the user's request. | GitHub repo settings | `github.com/Amazo123/CryNet` → private: false. |
594 | 2026-10-29 | **Added §6 "Legacy system vs. this rebuild"** — a line-by-line comparison of the abandoned system (read directly from the `legacy` remote) against the rebuild: architecture, model size (6.15 MB FP32 vs 135 KB INT8), mel bands, CMVN, dataset strategy, trigger/threshold, verification discipline. Documents what was inherited on purpose (hardware, topology, wire protocol) vs. re-decided (everything model/feature/threshold related), and states plainly that checkpoint 2 firmware is still pending. | `SYSTEM_UPDATE.md` | Answers the user's question about whether the rebuild is genuinely new or a re-skin of the failed system. Evidence: legacy `TinyCryNet` (attention + GRU, 6.15 MB), `RMS_TRIGGER_THRESHOLD 0.18`, 128 mel, no CMVN, ICD as a git submodule. |
595 | 2026-10-29 | **Checkpoint 1 committed and pushed to GitHub.** Created `github.com/Amazo123/CryNet` (private), committed the

full checkpoint-1 deliverable as `e7be48a`, renamed the branch `master` → `main`, pushed with upstream tracking. The old repo was renamed remote `origin` → `legacy` and left as a read-only reference. | all checkpoint-1 files; `.gitignore` | `.gitignore` extended to exclude `research/_real_work/`, `research/_realfast_work/`, `*.exe`, `*.pdb`, `*.o`, `*.obj` (build scratch). Push verified: remote `main` = `e7be48a`, 34 files. |
2026-10-29	**Created `SYSTEM_UPDATE.md`** — standing living status/recovery log. Imported the full checkpoint-1 state from `PROJECT_HANDOFF.md` and verified it against disk (`research/_realfast_out/*`, `metrics.json`, `deploy_config.json`, `real_fast.log`).	`SYSTEM_UPDATE.md` (new)	First entry. Establishes the update-after-every-edit protocol and the context-management rules (§0).
2026-10-29	`PROJECT_HANDOFF.md` §5 file map now points at `SYSTEM_UPDATE.md` as the living log / recovery layer.	`PROJECT_HANDOFF.md`	One-line map addition.
2026-10-29	Checkpoint 1 completed and validated (before this file existed).	`notebooks/crynet_training.ipynb`, `research/*`	Details in `PROJECT_HANDOFF.md` §4.
2026-10-29	Initial commit.	—	`997cc9b`

---

6. Legacy system vs. this rebuild (what was inherited, what was re-decided)

Verified on 2026-10-29 by reading the legacy repo directly (`legacy` remote): `firmware/central_hub_s3/central_hub_s3.ino`, `dsp_mel_extractor.h`, `firmware/common/protocol.h`, `model_training/tiny_crynet_esp32.py`, `model_training/reference_pipeline.py`, `model_artifacts/README.md`, and the git tree.

Inherited on purpose (hardware + topology — the user's real boards)

Same 3× ESP32-C3 cot nodes → ESP32-S3 hub → ESP32-C3 wearable; same ESP-NOW telemetry/audio-fragment protocol; same 2 s @ 16 kHz window with 512-pt FFT and hop 400; same `wearable_alert_packet_t` BLE format; same TFT/GATT UI. Nothing here was re-litigated, because the hardware is fixed and the wire format already exists.

Re-decided from scratch (this is where the new work is)

Aspect	Legacy system	This rebuild
Architecture	`TinyCryNet` — 5 × BSConvU blocks + **CCA channel attention** + **GRU** temporal aggregator (PyTorch → ONNX)	3 plain CNN variants, **no attention, no RNN**, chosen so every op has a TFLite Micro kernel (asserted at export)
Model size	`model_data.h` = **6,157,199 bytes (6.15 MB)**, FP32 in flash; needed a bespoke **6.0 MB `app0` partition**	**135 KB INT8** — 45× smaller, fits comfortably in stock partitions
Input	`[1,1,81,128]` — 128 mel bands	`[1,81,64,1]` — 64 mel bands
Mel filter	`htk=True`, `fmin=0`	librosa **slaney**, `fmin=20 Hz`
Level handling	`power_to_db(top_db=80)` — **absolute dB, no normalisation** ⇒ the model was level-dependent	**per-window CMVN** ⇒ level-invariant (ICD cries sit at RMS 0.006 vs negatives at 0.3–0.46)
Dataset	ICD pulled in as a **git submodule** of an academic baseline repo (`ICD_MMSP`, tree mode 160000); one corpus, no audio-level verification, no external negatives	every corpus **profiled at audio level**; ICD as primary **combined with** cry4 + ESC-50 + Snoring; contaminated corpora rejected; negatives deliberately include cat meows, speech, snoring
Trigger	hard-coded `RMS_TRIGGER_THRESHOLD 0.18` + `CONFIRMATION_PROB 0.65`	trigger ~0.03 + silence gate 0.0015 + **threshold calibrated on data** (0.23 at ≥ 95 % recall) with the whole recall/FPR curve reported
Training	35-epoch SpeechBrain/PyTorch run, artifacts in a 105 MB LFS tarball; no recorded per-class false-alarm analysis	Keras/TFLite notebook, **hard-example mining**, **per-category FPR breakdown**, cross-corpus evaluation on corpora never trained on
Verification	`verify_dsp_parity.py` existed as a simulation script	**automated** C↔Python parity (max deviation 7.06 × 10^{-5}), INT8↔float parity (AUC delta −0.00007), TFLM kernel assertion at export, manifest↔archive 1:1 corpus check
Claimed latency	"≈ 40 ms" (never measured on the board)	**measured on the chip**: front end **41.7 ms**, inference **101.8 ms** per 2 s window, CPU 240 MHz

Why the legacy system failed — and what that means

The legacy repo's **last commit** is `fix(firmware): resolve false cry detection, packet loss, and full-screen landscape display` — i.e. the system was still being debugged on the hardware when it was abandoned, and the failures it was chasing were *system-level* (ESP-NOW packet loss, TFT layout, false triggers), not dataset-level. With `RMS_TRIGGER_THRESHOLD 0.18`, a soft cry (ICD cries sit at RMS ≈ 0.006) **never reached the classifier at all**, so the old model was never fairly tested on the real hardware. The rebuild therefore changed the model, the features, the trigger and the threshold — but the *dataset* was kept, because the evidence points at the firmware and the trigger logic, not at ICD.

What is still not new / not proven yet

- **Checkpoint 2 (firmware) is more than half written.** The hub's inference core (Stage 2a) is new, verified on the real chip, and committed. The node firmware (Stage 2b) runs on the real board and produces verified audio. The hub's radio (Stage 2c) is verified end to end — audio crosses the ESP-NOW link with a matching CRC-32 and the model runs on it. The TFT display and the BLE GATT server (the rest of Stage 2c) are written and running together on the real hub (2026-10-31); the wearable (Stage 2d) does not exist yet.
- **No in-domain data.** The model has never heard the user's INMP441, room or gain. ~50 self-recorded clips remain the single biggest accuracy win, and they are also the honest test of whether the new system actually works in the real room.

---

7. Quick reference

Run locally

```bash
pip install numpy librosa soundfile soxr scipy scikit-learn tensorflow huggingface_hub
cd research && python make_stub.py && bash run_stub.sh     # synthetic smoke test
bash run_real_fast.sh                                       # real corpus, subsampled, CPU
pip install ziglang && python test_c_parity.py              # C <-> Python parity
```

Notebook env switches

`CRYNET_FAST=1` (subsample + 3 epochs) · `CRYNET_OFFLINE=1` · `CRYNET_KINDS=conv_cnn,sep_cnn,temporal_cnn` · `CRYNET_EPOCHS=35` · `CRYNET_MINING_ROUNDS=1` · `CRYNET_CACHE` / `CRYNET_WORK` / `CRYNET_OUT`.

Caches

HF archives at `research/_cache/audio/positive.zip`, `.../audio/negative.zip`, `research/_cache/metadata/all28479.csv`. Kaggle archives at `research/_cache/<key>__<owner>__<slug>.zip`.

Verified local versions
Python 3.11.9 · TensorFlow 2.21.0 (CPU) · librosa 0.11.0 · soundfile 0.14.0 · numpy 2.4.6 · scipy 1.17.1 · ziglang 21.1.0 (C compiler for the parity test; no gcc installed).

Hardware

SYSTEM_WIRING.md

267 lines · 31,401 bytes

CryNet — USB Test and Complete-System Wiring Guide

Revision: 7 November 2026. Read alongside `HARDWARE_COMPONENTS.md`.

> **Note:** `SYSTEM_WIRING.pdf` in this repository is a snapshot exported on **2 November 2026** and is **out of date** — it predates the battery design, the two-cot split and the removal of the battery sense and the status LEDs (D-83). This file and `SYSTEM_WIRING_DIAGRAMS.pdf` are the live documents.

> **Important:** The node + hub signal wiring in section 2 reflects the working prototype. The wearable and battery power are **not implemented or verified**. There are **no status LEDs anywhere in the system** (D-83): the node's connection LED, the node's SOC LED, the hub's BLE-status LED and the hub's SOC LED have all been removed, and the battery sense divider has been removed with them. Their exact part-to-part power wiring and GPIO choices cannot safely be finalised until the actual board and charger modules, cell and motor are selected. Section 3 is a proposed connection architecture, not permission to wire an unspecified lithium charger. This academic prototype is not a certified infant-monitoring device and must not replace supervision. Keep cables, cells, modules and small parts out of reach of the infant.

1. System picture

```text
  Cot 1: INMP441 → C3  ── ESP-NOW ──┐
  Cot 2: INMP441 → C3  ── ESP-NOW ──┼─→ ESP32-S3 hub → 2.8-inch TFT
                                                  │
                                                  └── BLE → phone (tested)
                                                      or → C3 wearable (on the bench)
                                                               ├→ SSD1306 OLED
                                                               └→ BC547 → haptic motor
```

No audio or signal wires connect the cot nodes to the hub: the link is wireless. Each assembly has its own local power and ground. The node, hub and wearable do not need grounds connected to each other.

> **This is the two-cot system.** The three-node version is the sibling repository `fp-2`. The wearable's haptic is a **BC547 low-side switch**, not the DRV2605L the older planning documents describe (D-77).

2. Wiring A — USB-powered bench test

Before wiring: disconnect power; identify actual board/module pin labels; use a multimeter to check for shorts and verify the display module voltage. These are board pin labels, not physical header positions. Do not assume every breakout uses the same pin order.

2.1 Cot node: ESP32-C3 + INMP441

Repeat the same connections for each node. **Multi-node operation has been verified** — on 2026-11-04 three nodes, the hub and the wearable ran together on one laptop for twenty-five minutes with **no brownout, no reset and no false alarm** (node 1's own counters: 6 385 frames sent, 0 lost, 0 times the radio had no room). *That session ran three cots; this repository's build is two.*

From	To	Note
C3 **3V3**	INMP441 **VDD / 3V3**	3.3 V only; not USB 5 V.
C3 **GND**	INMP441 **GND**	Common local ground.
C3 **GND**	INMP441 **L/R**	Select left I²S channel.
C3 **GPIO4**	INMP441 **SCK**	I²S bit clock.
C3 **GPIO5**	INMP441 **WS**	I²S word select.
C3 **GPIO6**	INMP441 **SD**	Microphone data to C3.
Regulated USB supply or computer	C3 **USB connector**	Use a compatible data cable for flashing/serial.
Optional 100 nF capacitor	Between mic **VDD and GND**	As close as practical to microphone; check existing breakout decoupling. Ceramic is non-polar.

```text
 C3 3V3 ──────────────── VDD   INMP441
 C3 GND ─────┬────────── GND
             └────────── L/R
 C3 GPIO4 ──────────────── SCK
 C3 GPIO5 ──────────────── WS
 C3 GPIO6 ──────────────── SD
 USB source ────────────── C3 USB socket
```

The old PDF's GPIO5/6/7 microphone assignment does **not** match the current firmware. C3 GPIO4 is already used by SCK and must not also be connected to the old red status LED.

Where the microphone goes — a placement rule, measured. The INMP441 is a **fixed-gain** part: it has no gain register and its own output saturates before the node's 16-bit window ever does, so a sound close enough to it arrives flat-topped and the window cannot be trusted. Measured on the bench: node 2 clipped at peak 31 801 of 32 767 on ordinary room sound, and a phone held 10 cm from the microphone clipped the cry it was playing — that window scored 0.9805 instead of 0.9961, and the alert named the wrong cot for 42 seconds. There is no software cure, so the rules for the finished assembly are:

- **Keep the microphone out of the cot's near field.** Mount it on the cot frame or a wall bracket, on a short stiff lead, at least **30 cm** from where the baby's head will be, and out of the mattress's own sound path.
- **Do not point it at a speaker, a radiator or a window.** A reflective surface doubles the level at the mic.
- **Never leave it loose on the mattress.** A soft surface damps the high frequencies the model reads, and the cot frame moves.
- The hub now **refuses a clipped window and says so in its log** (`the window is CLIPPED`), so a bad mount is visible rather than silent. Watch for that line in the hub's log after the first night of use; it is the fastest way to find a microphone that is too close to the sound.

2.2 Hub: ESP32-S3 + 2.8-inch ILI9341

The project log records the following signal and 3.3 V connections on the existing display. Match the actual module's labels and confirm its supply and backlight circuit before copying it to a different TFT.

From S3	To display	Note
3V3	**VCC**	Recorded working connection for the existing module; do not assume another module has identical power requirements.
GND	**GND**	Local common ground.
GPIO10	**CS**	Chip select.
GPIO11	**MOSI / SDI**	SPI data to TFT.
GPIO12	**SCK / CLK**	SPI clock.
GPIO13	**DC / D-C**	Data/command.
GPIO14	**RST / RESET**	Display reset.
GPIO4	**BL / LED control**	Recorded on the existing board. Check the module's backlight drive current and circuit; do not drive an unknown bare LED backlight from a GPIO.
Not connected	**MISO / SDO**	The current driver does not read TFT data.
Regulated USB supply or computer	S3 **USB connector**	Match the actual connector used for programming.

No external 100 Ω backlight resistor should be added blindly; use the specific module's circuit. Touchscreen and SD-card pins, if present, are not used by the current display firmware. The S3's BOOT button is used by the existing UI; it is already on the development board.

2.3 USB wearable bench wiring — proposed, not yet built

> **SUPERSEDED — this section describes the DRV2605L option, which is not what was built.** The wearable that is on the bench drives a **real haptic motor through a discrete BC547 low-side switch** (D-77), and the firmware exists and has been verified end to end. **The BC547 wiring is in §5.3 — build from there, not from the table below.** The DRV2605L is retained below only as the record of the alternative that was considered, so the two do not get confused again.

The connection list for the **DRV2605L alternative** — not a tested pin map, and not the design in use:

From	To	Condition
C3 **GND**	OLED **GND** and DRV2605L **GND**	Common local ground.
C3 **3V3**	OLED **VCC**	Only if the specific OLED breakout accepts 3.3 V.
Appropriate regulated supply	DRV2605L breakout **VIN/VDD**	Follow the actual breakout's voltage requirements; do not assume its VIN and logic pins are identical to the IC pins.
Selected C3 I²C SDA	OLED **SDA** and driver **SDA**	Bus pull-ups must go to a safe 3.3 V logic rail. Check existing pull-ups before adding any.
Selected C3 I²C SCL	OLED **SCL** and driver **SCL**	Same. Verify address compatibility.
Driver **OUT+ / OUT−**	Compatible motor's two terminals	Use the specific breakout's labels and motor requirements. Do not connect either motor lead to 3V3 or ground as in the old MOSFET circuit.
USB source	C3 USB socket	Supply capability and driver power arrangement must be checked.

Use the selected breakout's documentation for EN, I/O supply, power, pull-ups and any other control pins; these vary by board. DRV2605L uses an H-bridge, unlike the old low-side MOSFET design. No separate MOSFET or old motor flyback-diode wiring is used for this driver. See [TI DRV2605L datasheet](https://www.ti.com/lit/ds/symlink/drv2605l.pdf).

3. Wiring B — proposed battery-powered complete system

3.1 Connections that remain the same

For each of the **two** nodes, retain **all six microphone connections** in section 2.1. For the hub, retain the TFT signal and ground connections in section 2.2. For the wearable, use only a finalised and tested version of section 2.3. Only the **power source** changes; never join the four batteries together. The **battery sensing and the status LEDs are gone** (D-83) — there is nothing on B+/B− but the cell, and no indicator of any kind on any unit.

3.2 Power architecture for each separate unit

```text
 USB charging input
        │
        ▼
 [charger + properly designed power-path/load sharing] ↔ [protected 1-cell battery]
        │ regulated/selected system output, as the module specifies
        ▼
```

```
 [suitable regulated converter, if needed] → [power switch as designed]
        │                                    │
        ├── appropriate board power input    └── local loads, if designed
        └── local ground / return
```

Concrete pad names, from the photographed board (§6): the charging input is the module's **USB-C socket**; the battery lands on **B+/B−** (printed 电池); the load chain starts at **VO+ / VO−** (printed 输出). IN+/IN− (printed 输入) stay unconnected.

For each of the **two** nodes and the hub, the BOM proposes a protected 1-cell battery and a 5 V boost if the selected board uses its 5 V input. The wearable's suitable voltage/converter remains undecided. A lithium cell can be **4.2 V fully charged**, so do not connect it to a 3.3 V rail or GPIO; do not apply 5 V to 3V3. The converter's output must remain within the board input specifications under measured peak loads and low cell voltage. A 100 µF bulk capacitor is a **candidate**, not an unconditional connection: confirm location, voltage rating and stability requirements in the actual converter datasheet, then connect + to the intended rail and − to local GND only if appropriate. Do not connect bench USB and a battery supply simultaneously without a designed, verified power path preventing backfeed.

Cell selection (D-81): the nodes and the hub take a **protected 18650** (3.7 V nominal, 4.2 V full, 3000 mAh) in a holder; the wearable takes a **1000 mAh Li-Po pouch**. The two are opposite choices for opposite reasons — a node sits on a cot frame within a baby's reach and needs a cell that cannot be punctured, while an 18650 is 18 × 65 mm and 48 g and cannot go on a wrist. **Never an LiFePO4 cell anywhere**: the module charges to 4.2 V and an LFP cell tops out at 3.65 V.

3.3 No battery sensing, and no status LEDs

This section used to describe a divider and two LEDs. It does not any more (D-83).

There is **no battery sense divider on any unit** — not on the nodes, not on the hub, not on the wearable. B+ and B− carry the cell and nothing else. There is **no 220 kΩ, no 100 kΩ and no 100 nF** in the system, and there is **no LED on any unit**: no connection LED, no SOC LED, no BLE-status LED. The parts that were on the bill of materials for them should not be bought and must not be fitted.

The **battery power itself is unchanged**: the cells, the LX-LCBST modules, the switches and the boost converters are exactly as planned in §3.2, §5.1, §5.2 and §5.3. What has been removed is only the *measurement* of the cell and the *indication* of it.

Why: the hub's panel already shows which cots are answering, and it shows it better than a second, slower LED on the cot could. The battery reading was always an estimate — a resting-voltage curve printed with a tilde — and the owner does not want it. The cost of removing it is that **nothing in the system will warn the caregiver that a cell is nearly flat**; the charge-prompt idea is gone with it, and §7 now records the absence.

For the record, the divider that was designed and then withdrawn. It was a fixed pair — B+ → 220 kΩ → sense node; sense node → 100 kΩ ‖ 100 nF → B−; sense node → GPIO3 (ADC1_CH3) on the nodes and the wearable, GPIO1 (ADC1_CH0) on the hub — a ratio of ×3.2, so 4.2 V on the cell read as ≈ 1.31 V at the pin, drawing ≈ 13 µA continuously. It was a fixed pair and **not** a potentiometer, on purpose (D-81): a **two-terminal** pot cannot divide at all, since the ADC pin's only path to ground is its own high-impedance input, so the pin still sits near 4.2 V — past the C3 IO's 3.6 V absolute maximum. A **three-terminal** pot used as a real divider does work, and the firmware does not need to read the wiper; what it costs is a **per-unit measured ratio**, **silent drift** with heat, vibration and wear, and **magnitude** — the SOC curve moved 100 mV = 20 percentage points around 3.8 V, so 1 % of ratio error was already ~10 points of SOC error, against a pot's 1–5 %. Two 1 % resistors gave a ratio the firmware knew exactly on every unit with no per-unit step. All of that was sound; it was simply not wanted. The reasoning is kept in `DECISIONS.md` (D-81, D-83) and is not repeated here.

3.4 Wearable battery and haptics

The wearable's power follows the same LX-LCBST pattern as the nodes (§5.3): VO+ → switch → SuperMini 5V pin; the OLED from the SuperMini's 3V3 output so the I²C bus stays at 3.3 V; the motor driven by the **BC547 low-side switch on GPIO7** (D-77), not by the DRV2605L the older documents describe. The proposed 1,000 mAh **Li-Po pouch** cell only buys runtime if the module's measured charge current suits it — at ~1 A it does not, so the charge current must come down to ~400–500 mA (D-81). The IRLML2502 and its gate resistors/diode belong to the alternative MOSFET drive and are **not wired into the BC547 path**.

4. Checks before power-up

1. With power disconnected, check continuity and make sure no 3.3 V or 5 V rail is shorted to ground. Inspect vero-board copper tracks, intentional cuts and solder bridges.
2. Confirm the exact module labels, voltage ratings, polarity, cell protection and regulator settings. Check converter output with a meter **before** connecting its load.
3. For initial testing use a current-limited bench supply where possible. Test one assembly at a time; avoid unattended charging.
4. Test the known USB node-to-hub setup first. The wearable and additional nodes require their own implementation and verification.
5. Before using battery and USB together, verify the actual charger/load-sharing and backfeed behaviour. Secure cells against puncture/shorts and use appropriate enclosures.

5. Proposed safe GPIO and component plan

This is the proposed design for the next firmware/hardware branch. It deliberately avoids the documented boot-strapping pins and the pins already used by the microphone, TFT or native USB. It is a **pin plan**, not yet flashed or measured.

5.1 Cot node — two identical units

Function	GPIO / connection	Components
I²S BCLK	GPIO4	Existing INMP441 SCK
I²S WS	GPIO5	Existing INMP441 WS
I²S data	GPIO6	Existing INMP441 SD
Battery sense	**none**	The divider is removed (D-83): no 220 kΩ, no 100 kΩ, no 100 nF, and nothing on GPIO3. B+/B− are the pads printed 电池 (battery) on the board photo, and they carry the cell only.

| Status LEDs | **none** | The GPIO10 connection LED and the GPIO7 SOC LED are removed (D-83). GPIO7 and GPIO10 are free. |
| Power switch | **VO+ → switch → C3 5V pin** | SPST switch rated ≥1 A. VO+ / VO− are the pads printed 输出 (output). B+/B− stay connected for charging while the node is off. Do not switch the battery side. |

There is no battery reading on this unit (D-83), and no LED of any kind. The divider and its 100 nF are not fitted; the ADC pin is not read; the telemetry carries no cell voltage; and the hub's panel carries no battery text. GPIO3, GPIO7 and GPIO10 are free.

What the node *does* still do is the part that mattered for the link, and it is unchanged: it sends a heartbeat to the hub and the hub answers with a matching ACK.

```text
node → hub: HEARTBEAT(node_id, counter, CRC)
hub → node: HEARTBEAT_ACK(node_id, counter, CRC)
node: connection_ok = a matching hub ACK received within the last 3 seconds
```

The heartbeat and its ACK stay, because they are how the node knows the hub is there. What has gone is the *output* — the LED that used to show the answer. The node now prints the same fact on its serial console (`status`), and the hub shows it on the panel, which is where the caregiver looks. The hub must still track the last heartbeat/telemetry time; an ESP-NOW send callback alone is not application delivery confirmation.

5.2 Hub — one unit

The current TFT assignments remain fixed: GPIO10 CS, GPIO11 MOSI, GPIO12 SCK, GPIO13 DC, GPIO14 RST and GPIO4 backlight. Proposed additions avoid those pins and the S3 strapping pins:

Function	GPIO / connection	Components
Battery sense	**none**	The divider is removed (D-83): no 220 kΩ, no 100 kΩ, no 100 nF, and nothing on GPIO1.
Status LEDs	**none**	The GPIO2 BLE-status LED and the GPIO17 SOC LED are removed (D-83). GPIO2 and GPIO17 are free.
Power switch	**VO+ → switch → S3 5V pin**	Battery remains on module B+/B− while the hub is off. Verify the DevKitC input and converter output under TFT/radio load; the hub's peak draw is the heaviest of the five units and is not yet measured.

The hub TFT keeps its working assignments (GPIO4 backlight, GPIO10–14 signals) and the hub has **no battery sense and no status LED** (D-83). Avoid GPIO0/3/45/46 (S3 strapping) and GPIO35–37 (Octal PSRAM bus on the N16R8). GPIO1, GPIO2 and GPIO17 are free.

5.3 Wearable — one unit

The wearable uses the same C3 board class, but the final board must be checked against its pin labels. **This unit is built and working on the bench** — the OLED, the BLE client and the haptic motor are all in place (D-76, D-77). What it does not have yet is its battery, and the rows below are what remains to be built:

Function	GPIO / connection	Components
OLED SDA	**GPIO5**	Shared I²C SDA; add no external pull-up until the OLED module's own pull-ups are checked
OLED SCL	**GPIO6**	Shared I²C SCL; pull-ups, if needed, to 3.3 V only
Haptic output	**GPIO7 → 2.2 kΩ → BC547 base**	The low-side switch that is actually fitted: GPIO7 → 2.2 kΩ → base, 100 kΩ base-emitter pulldown, motor in the collector leg, 1N5819 across the motor with the cathode up, emitter → GND, **22 Ω** in the motor's + lead. **No IRLML2502 and no DRV2605L in this path** (D-77).
Power switch	**VO+ → switch → C3 5V pin**	Same pattern as the node: disconnects the device, battery stays on B+/B− for charging.
Status LEDs	**none**	The wearable has **no status LEDs** — and now it has no battery sense either (D-76, D-83). An earlier revision gave it a blue link LED on GPIO1 and a red SOC LED on GPIO4; neither ever had firmware, and the OLED already carries LINK/SCAN. The OLED stays the only display on the wrist, and it says nothing about the cell.

Wearable power: the wearable uses the same LX-LCBST pattern as the node: VO+ → switch → the SuperMini's 5V pin, VO− → GND. The OLED is powered from the SuperMini's **3V3 output pin**. Check the board's 3.3 V regulator current budget before final assembly: C3 radio peaks (hundreds of mA) + OLED (~25 mA) + motor bursts all share it; if it sags or resets during vibration, the motor supply design gets revisited on a branch. **The motor's burst current is lower than the earlier notes assumed** — the fitted **22 Ω** cap in the motor's + lead means the motor never sees the full 3.3 V: it gets about **2.1 V** and draws about **55 mA** rather than the 85 mA of a 3 V ERM run straight off the rail. That is a bonus for the 3.3 V budget and a penalty on the buzz (§6 explains the 5 V pin's own cost). ESP32-C3 strapping pins are GPIO0/2/8/9 — the plan above avoids all of them.

Runtime (D-82): the chain **cell → boost to 5.1 V → the SuperMini's 5 V pin → its onboard linear regulator → 3.3 V** means the cell supplies roughly **1.8× the 3.3 V load current**, because a linear regulator passes current rather than power. On a 1000 mAh pouch that projects to **about six hours** — a projection, not a measurement. Feeding the SuperMini's **3V3 pin** from a 3.3 V buck-boost would roughly double it; the OLED's always-on ~25 mA is the other lever. The cell is a **1000 mAh Li-Po pouch** (D-81) and the module's ~1 A charger must be reduced to ~400–500 mA for it.

6. LX-LCBST-TC-001: decision and exact conditional wiring

Pad map confirmed from the user's front-and-back photograph (2026-11-02):

Pad group	Printed label	Meaning (Chinese on silk)	Use
B+ / B− (front) / B+ / B− (back)	电池	**battery**	The cell connects here — an 18650 on a node or the hub, a Li-Po pouch on the wearable (D-81)
VO+ / VO− (two pairs, back)	输出	**output**	Boost output — feeds the power switch
IN+ / IN− (back)	输入	**input**	Alternate charging input, 4.2–6.5 V; leave **unconnected** (charging runs from USB-C)
Front − / + pair (unlabelled)	—	ambiguous	Most likely a paralleled output tap; **verify with a meter before use** — compare it against VO+/VO− and B+/B− with the battery installed
Trimpot (front, beside the charger IC)	—	—	Sets the boost output, 4.2–28 V

No **T** pad is visible on this board — the family listing's T-shorting instructions belong to other variants and **do not apply here**; do not short anything.

```text
cell +        → B+   (pad printed 电池)   [18650 on nodes/hub, Li-Po pouch on wearable]
cell –        → B–
USB-C socket  → charging input (nothing wired to IN+/IN–)
VO+ → power switch → ESP32 board 5V pin
VO– → ESP32 board GND pin
```

Set the trimpot before any ESP32 is connected: battery attached, meter on VO+/VO–, adjust to ≈ 5.1 V. The module can ship set anywhere in its 4.2–28 V range; connecting a board before setting it can destroy the board. Re-check after the first full charge cycle and occasionally thereafter.

Supplying a board from the 5 V pin — what it costs, and the one thing that must never happen

The whole system is powered the same way at every unit: **cell → LX-LCBST boost to 5.1 V → VO+/VO– → switch → the board's 5 V pin → the board's own linear regulator → 3.3 V**. Three things follow from that chain, and all three are consequences of the regulator stage rather than of the boost.

The board's linear regulator is the weak point, and it gets hot. A linear regulator passes *current*, not power, so the difference between 5.1 V in and 3.3 V out is burned as heat inside the board: **(5.1 − 3.3) V × I**. At the C3's radio peaks of a few hundred mA that is **roughly 0.5–0.7 W** in a package with no heat sink. That is survivable on a bench and it is why the cell has to supply about **1.8× the 3.3 V load current** (D-82), but it is also the reason a board fed this way can brown out or reset under a radio burst with a nearly flat cell. **If a unit resets when the radio transmits, the LDO's heat and the cell's sag are the first two things to check** — not the radio. Feeding the **3V3 pin** from a 3.3 V buck-boost removes the dissipation entirely and roughly doubles the runtime (D-82); it needs a separate 3.3 V module, because the LX-LCBST cannot be set below its 4.2 V minimum.

Never connect the boost and USB at the same time. The LX-LCBST's output goes straight to the board's 5 V pin, and that pin is hard-wired to the USB 5 V rail through the board's own protection diode. With USB plugged in *and* the boost connected, the two supplies fight: the 5.1 V rail and the USB's 5 V rail are the same net, and current can be pushed back up the USB cable into the host — which is a fault on the laptop's side, not just on the board's. **Unplug USB before the battery goes on.** When a unit has to be watched on the console while it runs from its cell, the answer is a **USB cable with the 5 V (VBUS) wire cut** — the data lines still carry the serial log, and nothing can be pushed back into the host.

The OLED and the INMP441 stay on 3.3 V in every arrangement. The OLED's I²C pull-ups ride on its VCC rail, so putting 5 V there would put 5 V levels into non-5-V-tolerant C3 GPIOs. The 3V3 pin of the board is the only 3.3 V source in the design, whether it comes from the onboard regulator or from a buck-boost.

Two characteristics remain unknown and must come from the bench validation: the **actual charge current** (the charger's program resistor is too small to read in the photo; if it programmes near 1 A, the wearable's 1,000 mAh pouch needs it reduced to ~400–500 mA, and a node's 3000 mAh 18650 is fine at 1 A), and the boost converter's **quiescent drain while the unit is switched off** (the switch disconnects the load, not the converter — if off-state drain proves significant, the switch moves or the module changes). Note also that B– and VO– connect through the module's protection FET: in normal operation they are effectively the same ground, but at deep-discharge cutoff the battery disconnects and the unit dies — which, with no battery readout anywhere (D-83), is exactly the moment the caregiver will notice it. Take the unit off the cot when the cell is nearly flat rather than waiting for that.

Required one-board validation before buying four more

1. ~~Identify every terminal~~ — done from the photograph (§6 pad map). Remaining unknowns: actual charge current, boost quiescent drain, protection thresholds, the unlabelled front –/+ pair.
2. With no ESP32 attached, measure boost output range, no-load voltage, output under a dummy load and startup behaviour at full and partly discharged cell voltage.
3. Measure charging current and confirm termination at 4.2 V; verify whether the protection circuit disconnects at low voltage and short/overcurrent without deliberately abusing the cell.
4. Confirm that switch OFF disconnects the converter/load but leaves charging connected, and measure switch-off battery drain.
5. Run one node from it: compare reboot count, burst completion, radio range and microphone noise with the known USB baseline.
6. Only then decide whether it is suitable for the hub and whether a separate 3.3 V regulator is needed for the wearable.

7. There is no battery readout (D-83)

This section used to describe the SOC display and the charge prompt. Both are gone.

There is no battery text on the hub's panel, no battery readout on the wearable's OLED, and no battery LED anywhere. The hub's cot card shows the cot's name, whether it is answering, the RMS bar and its number, the trigger notch, the confidence, and the radio link — and nothing about the cell. The node's `status` command no longer prints a cell voltage.

The consequence is worth stating plainly: nothing in the system will tell the caregiver that a cell is nearly flat. That was the whole point of the feature, and it is the thing being given up. The cells still need watching by eye: charge each unit before it is put back on a cot, and take the unit off the cot when the cell is nearly flat rather than waiting for it to protect itself.

What was removed, for the record. The plan was a fixed 220 kΩ / 100 kΩ divider (ratio ×3.2) with a 100 nF across the bottom leg, read on ADC1 and averaged over sixteen samples every four seconds, with a trust gate that refused a reading until three landed in the expected window, and a banded result on the panel: `BAT ~80 %`, `BAT LOW`, `BAT CRIT`, the band moving on hysteresis so one radio-load dip could not flash a warning at the caregiver. The map from voltage to percent was a curve, not a line — **4.20 V = 100 %, 4.05 ≈ 90 %, 3.90 ≈ 65 %, 3.80 ≈ 40 %, 3.70 ≈ 20 %, 3.55 ≈ 8 %, 3.45 ≈ 3 %, 3.30 ≈ 0 %** — because a lithium cell is not linear between 3.0 V and 4.2 V, and a straight line would call a genuinely quarter-full cell "58 %". The bottom was anchored at 3.3 V, not 3.0 V, because 3.0 V is where a protected cell's protection circuit cuts out: that is end-of-life, not "empty".

If it is ever wanted back, the honest version is not the divider but a real fuel gauge — a **MAX17048** on I²C (the hub and the wearable have a spare bus; the node's microphone occupies I²S, so a node would need the ADC divider or a part on another interface). A gauge reports charge in the cell's own coulombs and does not care how much the terminal sags under load, which was the divider's other weakness. The reasoning is in `DECISIONS.md` (D-81 for the divider, D-83 for the withdrawal).

8. What is still required before this counts as verified

The LX-LCBST pad map is now known from the photograph; what remains is bench truth. Build **one** node power assembly exactly as in §5.1/§6, measure charge current, set the trimpot to 5.1 V, confirm switch-off behaviour and off-state drain, then run the working node firmware from it and compare with the USB baseline (reboots, bursts, mic noise). The **board swap is done**: the two cots run ESP32-C3 **Pro Mini** boards and the existing SuperMini is on the wearable. It was verified live on 2026-11-02 — the same firmware flashed unchanged, native USB, 4 MB flash, bursts 286/286 and two cry plays producing two alerts — because the firmware targets the chip, not the board. Confirm the silkscreen pin labels against GPIO4/5/6 when soldering a new Pro Mini; the electrical assignment does not change. The battery sense and the status LEDs have been removed rather than built (D-83), so the pins they would have used — GPIO3, GPIO7, GPIO10 on the node, GPIO1, GPIO2, GPIO17 on the hub — are free and need no verification. No merge into `main` without the user's decision.

Evidence: `firmware/node/node_crynet/node_crynet.ino` (GPIO4/5/6); `firmware/hub_infer/components/hub_display/hub_display.c` (TFT pins); `SYSTEM_UPDATE.md` (working display wiring and hardware tests); `HARDWARE_COMPONENTS.md` (proposed BOM).

HARDWARE_COMPONENTS.md

177 lines · 26,325 bytes

CryNet — Hardware Components and Purchasing Specification

Revision: 2 November 2026
System: two cot nodes, one central hub and one caregiver wearable
Purpose: replace the earlier `HARDWARE_COMPONENTS_DETAILED_SPECIFICATION.pdf` for planning and purchasing.

This is a **planning bill of materials for the complete system**, including a proposed independent rechargeable battery supply for each unit. **There is no battery indication and no status LED anywhere in the system (D-83):** the battery-sense divider and every indicator LED have been removed, so this list carries no divider resistors, no divider filter capacitor and no LEDs. The battery circuits and the wearable have not been designed and validated yet: component quantities below are planning quantities, **not a verified schematic or a final purchasing guarantee**. Modules are purchased assembled: their onboard microscopic components are not listed as separate purchases. Quantities are **total system quantities**, not necessarily quantities still to buy; subtract parts already owned. The user has confirmed a **2.8-inch**, not 2.4-inch, hub display.

> **Safety:** This is an academic prototype, not a certified medical device or a substitute for an adult caregiver. Detection, range, radio delivery, battery life and alarms have not been established to a safety-critical standard. Keep electronics, cords, batteries and small parts out of an infant's reach.

1. System overview and verification

Two ESP32-C3 microphone nodes transmit audio via ESP-NOW to an ESP32-S3 hub. The hub runs the cry classifier, shows status on an ILI9341 display and exposes a BLE alert service. An ESP32-C3 caregiver wearable receives the alerts and shows them on an OLED and buzzes a haptic motor.

- **Verified:** one real node, one hub, the display and a BLE phone client together; the cry classifier and radio burst transfer have run on the boards. The wearable is built, flashed and on the bench (see D-77).
- **Not yet verified:** two cots detecting at the same time, the battery runtime, and the battery-powered versions of any unit. The final battery-powered hardware has not been validated.
- **Not needed:** a separate PSRAM module; the hub board already has PSRAM. A colour-check photo is no longer required by the user.

2. Cot nodes — two identical units

Item	Per node	System total	Specification and purchasing guidance
ESP32-C3 node board	1	**2**	**Both cots run an ESP32-C3 Pro Mini.** The swap off the original SuperMini prototype was **made and verified live on 2026-11-02** (native USB `303A:1001`, 4 MB flash, same firmware unchanged, two cry plays → two alerts); the two boards on the bench are `90:DA:72:85:C5:44` (id 1) and `90:DA:72:87:CB:98` (id 2). The existing SuperMini moved to the wearable. The radio is built in.
INMP441 digital I²S microphone breakout	1	**2**	Use a 3.3 V-compatible INMP441 breakout. Keep the acoustic opening unobstructed.
USB data/power cable	1	**2**	Match the board connector; data-capable cables support programming. Existing cables can be shared for programming if both nodes are otherwise powered.
Regulated 5 V USB supply	1	**2**	One per simultaneously operating node, or an adequately rated multiport supply. Use a reputable supply with current headroom; verify connector and board power input.
Stripboard / vero board	As needed	As needed	Cut into two assemblies, with sufficient spacing and insulated underside.
Hook-up wire, headers and connectors	As needed	As needed	Suitable for power and I²S; mount the mic without covering its sound port.
Enclosure and mounting hardware	1 set	**3 sets**	Insulated, with sound access, secure mounting and cable strain relief.
100 nF ceramic capacitor, marked 104	Optional	Up to **2**	Supplemental mic-supply decoupling close to VDD/GND if needed; check whether the breakout already includes it. Not an established fix for the observed radio/audio interference.

Current firmware microphone pins: INMP441 SCK → GPIO4, WS → GPIO5, SD → GPIO6; L/R → ground for the tested left channel, and use the module's 3.3 V supply and ground. The old PDF's GPIO5/6/7 assignments do **not** match the current node firmware. Confirm the actual board labels before soldering.

Proposed rechargeable power and indicators — per node

The original document's battery arrangement is restored as a **planning design**, not as working firmware or a validated wiring diagram. For the two nodes, multiply each per-node quantity by two. The 5 V output must be regulated and connected only to the board's appropriate 5 V/USB input; verify the actual board and peak load first.

Cell choice for a node (D-81): an 18650 Li-ion cell, in a holder. It is 3.7 V nominal / 4.2 V full, the same chemistry as the pouch cell the module was designed around, and it differs from the pouch in the two ways that matter here: it holds far more

energy per unit volume, and its steel can cannot be punctured by a cot rail or a child's teeth. A node sits on a cot frame within reach of a baby, so the mechanical robustness is the deciding argument, not the price. Target **3000 mAh** or better. **Do not use an LiFePO4 (LiFe) cell** — it charges to 3.65 V, not 4.2 V, and the module's charger would destroy it. Ordinary LCO/NMC 18650 cells (3.7 V nominal, 4.2 V full) are correct. Buy **protected cells** (with the protection PCM built in) or add protection separately; an unprotected 18650 next to a boost converter is a fire risk.

Individual item	Per node	All nodes	Selection / purpose
Protected 18650 Li-ion cell (3.7 V nominal, 4.2 V full)	1	**2**	3000 mAh or better, with protection. **Never LiFePO4.** Runtime not yet measured — see the runtime note in §3.
18650 cell holder with leads	1	**2**	Soldered or bayonet type, sized for the cell. The holder must survive a cot rail being knocked; the cell must not be able to move.
1-cell charger module	1	**2**	TP4056-based module is a candidate; ensure suitable cell protection and **load sharing/power-path**, and set a charge current appropriate for the actual cell. A common TP4056 protection board alone does not guarantee safe simultaneous operation and charging.
5 V boost converter	1	**2**	MT3608 module is a candidate; select a converter with adequate **continuous** output at the lowest cell voltage and transient headroom, not merely its advertised peak figure. Verify shutdown, efficiency and actual ripple.
On/off switch	1	**2**	Rated for the chosen power-path current; final placement depends on charger/load-sharing design.
100 µF electrolytic capacitor	1	**2**	Initial bulk-capacitor allowance for the 5 V rail, voltage rating at least 10 V; respect polarity. Confirm value and placement with the selected converter.
Battery connector pair	1	**2 pairs**	Match cell and charger; check polarity physically, not by colour alone.

Pin warning: the old PDF assigns a red status LED to GPIO4, which is now used by the microphone — **do not wire the old LED circuit**. GPIO9 is a boot-strapping pin. The complete charger/power-path design must be selected before ordering or connecting these parts.

There are no indicators on a node (D-83). No link LED, no SOC LED, no LED at all: the hub's panel is the only place the system shows anything. The pins that would have carried them — GPIO10 for the link LED, GPIO7 for the SOC LED, GPIO3 for the battery ADC — are free, and nothing on the board drives them.

The battery sense is not in this bill of materials (D-83). The 220 kΩ / 100 kΩ divider, its 100 nF filter and the indicator LEDs were designed, costed and then withdrawn: the cots do not measure their cells and the hub's panel already shows the caregiver what matters. The reasoning for the design — a fixed 1 % pair rather than a potentiometer, the ×3.2 ratio, the 100 nF being load-bearing at the divider's 68.75 kΩ source impedance, the SOC curve rather than a linear voltage map, and the reading sagging 100–200 mV under the boost and radio load — is kept in `DECISIONS.md` (D-81) and is not repeated here. **Do not buy the divider resistors, the divider capacitor or the indicator LEDs.**

The one thing that stays true whatever else changes: **the battery voltage must never reach an ADC pin directly** — the C3's IO is 3.3 V only, and a full cell is 4.2 V.

If a battery readout is ever wanted back, the honest part is a **MAX17048** fuel gauge on I²C, not a divider: it reports a model-based SOC with no calibration table and does not care how much the terminal sags under load. The hub and the wearable have a spare I²C bus; the node's microphone occupies I²S, so a node would need the ADC divider or a part on another interface.

3. Central hub — one unit

Item	Total	Specification and purchasing guidance
ESP32-S3-DevKitC-1 with PSRAM	**1**	Existing hub, reported as N16R8: 16 MB flash and 8 MB PSRAM. ESP-NOW, BLE and model inference use the same board. Confirm markings when replacing it.
2.8-inch ILI9341 SPI colour TFT	**1**	Existing user-confirmed size; 240 × 320 pixels. Match the actual module and its power/backlight requirements rather than assuming all 2.8-inch boards are wired alike.
USB data/power cable	**1**	Match the actual hub USB connector; data capability is useful for flashing and logs.
Regulated 5 V USB power supply	**1**	Stable, suitably rated supply for the hub and display; assess actual current and the module's power arrangement before final assembly.
Stripboard, wire, headers and connectors	As needed	For secure signal and supply wiring.
Insulated enclosure and mounting hardware	**1 set**	Provide a TFT opening, secure fixings and cable strain relief.

Firmware signal assignments: TFT MOSI → GPIO11, SCK → GPIO12, CS → GPIO10, DC → GPIO13, reset → GPIO14 and backlight control → GPIO4. Verify the particular display module's power and backlight circuit before connecting it; GPIO4 is a control signal, not an instruction to power an unknown backlight directly.

Display backlight: module circuits differ. Do not automatically add the old document's 100 Ω series resistor or apply 5 V to an unidentified TFT pin. Verify the specific board's schematic/markings and the actual tested backlight wiring before permanent assembly.

Proposed rechargeable hub power — planning quantities

Individual item	Qty	Selection / purpose
Protected 18650 Li-ion cell (3.7 V nominal, 4.2 V full)	**1**	3000 mAh or better, with protection. **Never LiFePO4.** The hub draws the most current of any unit, so this is the cell most worth sizing carefully. Use a holder if the selected cell requires one.
18650 cell holder with leads	**1**	Sized for the cell; the cell must not be able to move in its holder.
Single-cell charger with suitable protection and power-path/load sharing	**1**	Select for the actual battery and simultaneous-load requirements. A basic TP4056 is not automatically a load-sharing solution.
Regulated 5 V boost converter	**1**	MT3608 is a candidate only if verified for the actual continuous and peak hub/display loads over the full battery range.
Power switch	**1**	Rated and placed for the final power path.
100 µF electrolytic capacitor	**1**	Initial 5 V rail bulk-capacitor allowance; ≥10 V rating, observe polarity, verify against converter requirements.
100 nF ceramic capacitor, marked 104	**1**	Supplemental local decoupling allowance; confirm existing board components and placement.
Battery connector pair	**1 pair**	Match the cell and charger; verify polarity.

These are a proposed design, not verified hub wiring. A portable power system needs measurement, a reviewed schematic and test before permanent assembly. The existing ESP32-S3 board already contains its own supporting circuitry; do not automatically add a separate TFT backlight resistor.

Runtime is the open question on every battery-powered unit, and the power chain is why. The chain is **cell → boost to 5.1 V → the board's 5 V pin → the board's own linear regulator → 3.3 V**. A linear regulator passes *current*, not power, so the current the cell must supply is set by the 3.3 V load while the difference between 5.1 V and 3.3 V is burned as heat. If the load draws 100 mA at 3.3 V, the cell is supplying roughly **160 mA** — about 1.8× the load current. A 1000 mAh pouch would therefore last on the order of **six hours**, and a 3000 mAh 18650 rather less than a day, at the hub's heavier load. *These are projections from the regulator's behaviour, not measurements — no runtime has been measured on this system.*

The largest single lever on runtime is to skip the 5 V→3.3 V linear stage. Feeding the board's **3V3 pin** from a 3.3 V buck-boost converter makes the cell current approach the load current and roughly doubles the runtime for the same cell. This changes the power architecture, so it is recorded as an option to evaluate at the bench (D-82), not something to rewire before the current build is finished. The present design is already drawn; changing it now would invalidate the wiring the owner is soldering.

4. Caregiver wearable — one unit, built and on the bench

The wearable **exists and has been verified on the bench**: it is flashed, it has received real alerts from the hub over BLE, it renders them on the OLED and it drives a real haptic motor. What is missing is the **battery** — the unit currently runs from USB. The list below is therefore the remaining work, not a proposal for something that has never existed.

Item	Total	Status / selection
ESP32-C3 development board	**1**	**The existing SuperMini is reassigned here** (the node role moves to the Pro Minis). Already on the bench and working. Check physical size and supply requirements before final wearable assembly.
SSD1306 OLED module	**1**	**0.96-inch, 128 × 64, I²C**, already fitted and working. SDA GPIO5, SCL GPIO6.
BC547 NPN transistor	**1**	**Fitted and working** — this is the haptic drive the owner actually built: GPIO7 → 2.2 kΩ → base, 100 kΩ base-emitter pulldown, motor in the collector leg, 1N5819 flyback with the cathode up (D-77).
Compatible vibration actuator	**1**	**A 3 V ERM coin motor is fitted and working** (85 mA). **The stall cap fitted in the motor's + lead is 22 Ω, not the 10 Ω this list used to name** — the owner had a 22 Ω and no 10 Ω, and the bench proved the drive works with it (2026-11-11). It is also the **safer** of the two: 3.3 V ÷ (37 Ω + 22 Ω) ≈ **56 mA** of stall current, against ≈70 mA with a 10 Ω, so it sits further inside the BC547's 100 mA rating. The cost is that the motor sees ≈2.1 V of the 3.3 V rail instead of ≈2.6 V, so the buzz is somewhat softer than a 10 Ω would give. A motor lower-impedance than 37 Ω would still need a 2N2222A.
Protected rechargeable 1-cell **Li-Po pouch** battery	**1**	**The opposite choice to the node's 18650 (D-81), and deliberately so.** An 18650 is 18 × 65 mm and 48 g — it will not sit on a wrist. A flat pouch cell is the right form factor here. Target **1000 mAh** (the original 500 mAh was too small to survive a night shift). Same chemistry as an 18650: 3.7 V nominal, 4.2 V full. **Never LiFePO4** — the module charges to 4.2 V.
1-cell charger with protection and load-sharing/power-path	**1**	The module's charger is ~1 A, which is **1 C into a 1000 mAh pouch — too high**. Reduce the charge current to about 400–500 mA, or fit a larger cell.
Regulated 5 V boost converter	**1 candidate**	Only if the selected C3 board is powered via its 5 V input; alternatively a properly specified 3.3 V regulator may be designed. Do not fit both by default.
On/off switch	**1**	Rated and positioned for the selected power path.
100 µF electrolytic capacitor	**1 candidate**	Bulk supply allowance, ≥10 V; final value and necessity depend on the converter.
100 nF ceramic capacitor	**1 candidate**	Local supply decoupling allowance; check what is already on the modules.
Battery connector pair	**1 pair**	Match the cell and charger and verify polarity.
4.7 kΩ resistors	**2 candidates**	One SDA and one SCL pull-up to 3.3 V **only if** adequate I²C pull-ups are not already fitted to the OLED module. Check combined resistance and voltage before adding.
Vero board, wires, headers and connectors	As needed	Small enough for the chosen enclosure; use insulation and secure all connections.
Wearable enclosure and strap	**1 set**	Secure, insulated, comfortable and appropriate for the chosen battery and motor. The pouch cell must be **insulated and physically protected** — it sits against skin and a punctured pouch is a thermal-runaway risk.

The motor-drive choice is settled, and this list no longer offers the DRV2605L as a recommendation. The DRV2605L haptic-driver breakout and the IRLML2502/AO3400 MOSFET alternatives were both in the original planning document. The hardware actually built and working is the **discrete BC547 low-side switch** with a 1N5819 flyback (D-77), driven by the firmware at 250 ms pulses for a 2.5 s buzz. The user already owns an IRLML2502TR; retain it as a spare. If a direct-drive alternative is chosen later, a SOT-23-to-through-hole adapter can make it usable on vero board, with its diode and gate resistors correctly designed for the motor.

The wearable has no status LEDs (D-76), and no battery sense either (D-83). Earlier revisions of the wiring diagram gave it a blue link LED on GPIO1 and a red SOC LED on GPIO4; neither ever had firmware, and the OLED already carries the state the caregiver needs (LINK or SCAN). The battery divider is gone too, so the OLED says nothing about the cell — the caregiver watches the charge state by eye. GPIO3 is free.

Power development: a suitable regulated USB supply may be used during bench development if compatible with the selected ESP32 board. It is not a substitute for the eventual portable battery circuit. A nominal 3.7 V lithium cell is about 4.2 V when fully charged; never assume it is safe to connect directly to a 3.3 V input or GPIO. Match charger current to the actual cell and use an appropriate protection solution; charging/power-path behaviour must be verified rather than assumed from a generic module name.

Runtime expectation for the wearable (D-82): roughly six hours on a 1000 mAh pouch, and the reason is the power chain, not the cell. The chain is **cell → boost to 5.1 V → the SuperMini's 5 V pin → its onboard linear regulator → 3.3 V**, and a linear regulator passes current rather than power, so the cell ends up supplying roughly 1.8× the 3.3 V load current. The OLED (~25 mA) and the C3's radio are both significant, and the radio is on continuously because the wearable stays subscribed to the hub's BLE service. *This is a projection, not a measurement.* Feeding the **3V3 pin** from a 3.3 V buck-boost would roughly double it, and the OLED's brightness is the other lever.

5. Assembly and development supplies

Item	Recommended amount	Notes
Vero board / stripboard	Enough for 2 nodes, hub and wearable	Final sizes depend on layout.
Insulated hook-up wire, pin headers and appropriate connectors	Assorted	Clearly label power and ground; use secure connections.

USB data cables	Up to **4** for simultaneous connection	Three for the two nodes and hub plus one for wearable; fewer suffice if shared or using power-only cables for already programmed units. Confirm each board's connector.
Regulated USB power outputs	**3** for two nodes + hub simultaneously	Plus one for wearable bench development if powered independently; a properly rated multiport supply may provide several outputs.
Solder and suitable soldering tools	As needed	Observe ventilation and safe soldering practice.
Heat-shrink / electrical insulation	Assorted	Cover joints and prevent shorts, especially near power connections.
Multimeter	**1**	Verify supply voltage, continuity, polarity and shorts before powering.
Breadboard and jumper wires	Optional	Useful during testing, not part of the final soldered build.
Fasteners, spacers and strain relief	As needed	Secure each assembly and cable in its enclosure.

6. Consolidated purchasing count

Main component	Total for complete system	Status
ESP32-C3 development boards	**3**	2 nodes + 1 wearable; subtract the boards already owned.
INMP441 microphone breakouts	**2**	1 per node; subtract the microphone already owned.
ESP32-S3-DevKitC-1 with PSRAM	**1**	Existing hub.
2.8-inch ILI9341 SPI TFT	**1**	Existing hub display.
0.96-inch SSD1306 I²C OLED	**1**	**Fitted and working** in the wearable.
BC547 NPN transistor	**1**	**Fitted and working** — the wearable's haptic switch (D-77).
Compatible vibration motor	**1**	A 3 V ERM coin motor is fitted.
LX-LCBST-TC-001 charger + boost modules	**4**	2 nodes, 1 hub, 1 wearable. Battery pads B+/B−, output VO+/VO−, charging via USB-C — pad map confirmed from the user's photograph. Bench-validate ONE board (charge current, trimpot-set 5.1 V output, off-state drain) before buying the rest.
Protected 18650 Li-ion cells, 3000 mAh	**3**	2 nodes + 1 hub. **Never LiFePO4.** Buy protected cells.
18650 cell holders with leads	**3**	One per 18650; the cell must not move in its holder.
Protected 1-cell Li-Po pouch cell, 1000 mAh	**1**	The wearable only — an 18650 is too large and heavy for a wrist (D-81).
On/off power switches	**4**	One per unit; match the actual current and power-path placement.
100 µF electrolytic capacitors	**3 + 1 candidate**	Proposed 2 node and 1 hub rail capacitors; wearable candidate.
100 nF ceramic capacitors, marked 104	**up to 2 candidates**	Supplemental mic-supply decoupling near the node's VDD/GND if needed. The node ADC filter, the hub allowance and the wearable divider filter are gone with the battery sense (D-83).
4.7 kΩ resistors	**up to 2**	Wearable I²C pull-ups only if module pull-ups are inadequate.
Matching battery connector pairs	**4**	One per unit; verify connector type and polarity.
Node enclosures / hub enclosure / wearable enclosure and strap	**2 / 1 / 1**	Select after layouts and power arrangements are known.

7. Differences from the old specification

1. Hub TFT corrected from **2.4 inches to the user's 2.8 inches**; the ILI9341 and 240 × 320 resolution remain relevant to the existing implementation.
2. Rechargeable power is included as a complete-system **proposal**, with individual quantities; unlike the USB prototype it is not yet implemented or validated. **There are no battery ADC parts and no indicator LEDs in the list** (D-83). A fixed TFT backlight resistor is not assumed.
3. **The system is two cot nodes, not three** — this repository is the two-node build, and the quantities above are for two nodes. The three-node version lives in the sibling repository `fp-2`.
4. **The wearable haptic is a BC547 low-side switch with a 1N5819 flyback**, not the DRV2605L breakout the old PDF specified; the discrete MOSFET and gate resistors are not baseline components (D-77).
5. **Battery cells are 18650 for the nodes and the hub, and a Li-Po pouch for the wearable** (D-81). The old document named one generic pouch cell for all five units, which was wrong on both counts: a pouch cell is mechanically wrong for a unit sitting on a cot within a baby's reach, and an 18650 is physically wrong for a wrist strap. **LiFePO4 cells must not be used anywhere in this system** — the module charges to 4.2 V and an LFP cell tops out at 3.65 V.
6. **There is no battery sense and no status LED anywhere in the system (D-83).** The fixed 220 kΩ / 100 kΩ divider that D-81 designed — chosen over a variable resistor for good reasons — and its 100 nF filter have been withdrawn before they were fitted, together with the node's link LED, the node's SOC LED, the hub's BLE-status LED and the hub's SOC LED. The reason is that the hub's panel already shows which cots are answering, and the battery figure was always an estimate. The cost is that nothing in the system will warn the caregiver that a cell is nearly flat.
7. Wearable charging, protection and regulation are acknowledged as necessary **if battery powered**, but remain to be selected together rather than copied as an unverified circuit.
8. Vero board, power supplies, enclosures, mounting and safety considerations are included. Bench breadboards are optional rather than required for the finished build.
9. The current model already runs using the hub's existing PSRAM; an extra memory component is not required.

References and project evidence

- User-supplied original: `HARDWARE_COMPONENTS_DETAILED_SPECIFICATION.pdf`, five pages, October 2026; reviewed as the superseded specification.
- Project status: `SYSTEM_UPDATE.md`; implementation and project decisions: `DECISIONS.md` and `firmware/`.
- [Texas Instruments DRV2605L datasheet](https://www.ti.com/lit/ds/symlink/drv2605l.pdf): supply and I²C limits, supported ERM/LRA actuators and output wiring — *retained as background only; this system uses the BC547 circuit, not this part.*
- [Infineon IRLML2502 datasheet](https://www.infineon.com/assets/row/public/documents/24/49/infineon-irlml2502-datasheet-en.pdf): existing MOSFET package and electrical ratings — *retained as a spare part.*
- [MAX17048 fuel-gauge data sheet](https://www.analog.com/media/en/technical-documentation/data-sheets/max17048-max17049.pdf): the I²C alternative if a true SOC percentage is wanted later (D-81, D-83).

Bench procedure

DEMO_RUNBOOK.md

146 lines · 6,275 bytes

````
# Demo runbook

**What this is:** the procedure for the day you demonstrate CryNet. One
page, in order, with the waiting times and the exact log lines to look
for.

**The system:** three ESP32-C3 cot nodes with INMP441 microphones, one
ESP32-S3 hub that runs the model, one BLE wearable on the caregiver's
wrist. A phone plays a recorded infant cry; the hub names the cot that
heard it; the wrist buzzes.

---

## Before you start (once)

1. **The phone's volume.** Set it once, to the level that makes the hub
   read **0.10–0.20 rms** on its panel, and **do not touch it during the
   demonstration**. On most of the clip kit, one volume step louder
   saturates the microphone and the window is refused — the margin
   between "announces" and "clipped" is one step. (Measured:
   `research/out/demo_rehearsal.md`.)
2. **The clip.** Put `session9_clip_0005.wav` on the phone. It is in
   `research/out/demo_kit/`, and it is the most forgiving of the kit:
   it announces at every volume and never clips.
   - Backups, in order: `session4_clip_0012.wav`, `session1_clip_0005.wav`.
   - **Do not** play from the laptop's speakers. They reshape the cry
     enough that the model refuses it (P 0.03–0.15 on a window the model
     scores 0.9961 from the same recording). The phone is not a
     nicety; it is the only source shown to work.
3. **The room.** Quiet if you can get it. The system was measured in a
   quiet room and holds no false alarm in 20.5 minutes of a room with
   people talking, but the demonstration is easier in a quiet room.

---

## The procedure

**The phone goes 15 cm from the microphone of the cot you are testing.**
Not 10 cm (that saturates the microphone) and not 1 m (the cry never
crosses the trigger). 15 cm.

### Cot 1

1. Hold the phone **15 cm from cot 1's microphone**.
2. Press play. **Let the whole 5-second clip run out.** Do not press
   twice, and do not stop it early.
3. **Wait 5 seconds.**
4. **Look at the hub's log**, not only at the wrist:

   ```
   node 1: front end 48.2 ms, inference 103.5 ms, P(no_cry)=0.0039 P(cry)=0.9961
   node 1: ALERT — cry
   ```

   **That second line, on the node you played at, is the pass condition.**
5. **Wait 10 more seconds.** The alert is held for 8 s and then clears
   (the wrist shows `cleared`). You must let it clear before the next
   cot, or the hub keeps naming cot 1 and the wrist shows the wrong cot.

### Cot 2 — the same five steps, ~15 seconds each

### Cot 3 — the same five steps

### After the third cot

1. **Wait 10 seconds of quiet.**
2. Check the hub's log: there should be **three `ALERT — cry` lines, one
   per cot, and nothing after the last one.** A quiet window must never
   announce.

**Total: about 45 seconds of playing, 75 seconds with the waits.**

---

## Reading the log
````

```
| the log says | what it means | what to do |
|---|---|---|
| `node N: ALERT — cry` | **pass** — the right cot named | move to the next cot |
| `the window is CLIPPED` | the phone is too close; the microphone saturated | move it **back** to 15 cm, do **not** turn the volume down |
| `no cry` (with P(cry) high) | the window caught a pause in the clip, not the cry | play once more, same place |
| `cry (too quiet to announce)` | below the 0.020 rms floor | phone closer, or the volume up |
| `cry (unconfirmed)` | a maybe: P(cry) was high but the window was not clean | play once more |
| `ALERT — cry` on the **wrong** cot | another cot heard it cleanly | wait out the 8 s hold, then play again |
| nothing at all | the cry never crossed the trigger | phone closer |

**The log is the evidence. The wrist is the output.** The wrist buzzes
once per cry and names the loudest cot, so "it buzzed" proves much less
than the log line does.

---

## Why one play, and not several

The node freezes **one** window per cry. Once it has, it will not freeze
again until the room goes quiet, so a second play of the same clip lands
on the tail of the clip — a pause between cries — and the model scores
that around 0.11. That "no cry" is the *clip* being wrong, not the
system being wrong.

Playing once and reading the log tells you what happened. Playing three
times and watching the wrist tells you almost nothing.

---

## If something goes wrong, in the order to try

1. **Move the phone back to 15 cm.** Clipping is the most common
   failure and the fix is distance, not volume.
2. **Play once more**, same place, and read the log.
3. **Check the hub is still alive** — the console answers `status`:

   ```
   I (12345) console: << status
   I (12346) console:   cot 1: rms 0.0154  level 0.0092  window 0.0121 (whole)  confidence 0%
   I (12347) console:   cot 2: rms 0.0148  level 0.0088  window 0.0119 (whole)  confidence 0%
   I (12348) console:   cot 3: rms 0.0161  level 0.0095  window 0.0124 (whole)  confidence 0%
   ```

   If the console does not answer, the hub has crashed or lost power.
4. **Check the node is alive** — its console answers `status` and
   `stats` too.
5. **The link**, if a window never arrives: the hub logs
   `the burst stalled with N of 286 fragments` and asks for the missing
   ones. The resend rounds recover nearly all of it; if a burst dies
   outright, move the cot away from whatever is on the same channel.

What was measured, and where

fact	source
the kit re-derived against the rule on the boards; 27 of 37 clips announce; 17 clip if turned up	`research/out/demo_rehearsal.md`
the clip list and what to use	`research/out/DEMO_KIT.md`
the freeze lands 2.0–5.4 s after the clip starts (median 2.9 s)	`research/out/demo_rehearsal.csv`
the model needs ~1.5–1.75 s of continuous cry in the window	`research/out/DEMO_KIT.md`
the alert is held 8 s and names one cot	`firmware/hub_infer/src/link_stage.c` (`CRY_HOLD_MS`, `COT_SWITCH_MARGIN`)
20.5 minutes with people talking, 0 false alarms	`research/out/ambient_25min/summary.md`
three cots on one hub, 25 minutes, no brownout or reset	`SYSTEM_UPDATE.md`

The one thing to remember

Set the volume before you start. Do not touch it during.
```

research/CONFUSION_SESSION.md

80 lines · 3,642 bytes

```
# The confusion session — what the system does when the room is not quiet

The model's behaviour on **real music** was never trained and has never been
measured (no corpus in the training mix contains any). The false-alarm map
```

over ESC-50 says the model rejects speech, household noise and human non-speech well (1.6 %), but ESC-50 has no music either.

The number that matters for the demo is the one measured **in this room, through this microphone, with this speaker**. That is this session.

What to record

Five recordings, one minute each, with the INMP441 in the cot and the hub listening (the hub is not needed for the recording — only the node is).

#	What	Why
1	**Quiet room** — nobody, nothing	the floor: what the room alone does
2	**Speech** — talk, read something, laugh, cough	the confusion class with the most energy
3	**Music** — play a track from the phone, loud	the class the model has never seen
4	**TV / other** — a talk show, a video game, a vacuum cleaner	broadband non-cry sound
5	**The cry** — the phone playing the baby-cry recording	the positive, at the same volume

Record 5 **twice**: once with the phone next to the node (loud) and once a metre away (quiet). The distance changes the level, and the hub's trigger sits at RMS 0.030, so a cry that is too quiet never reaches the model at all.

How

The node has to be running `node_capture` (not `node_crynet`):

```bash
ARDUINO_CLI="$LOCALAPPDATA/Programs/Arduino IDE/resources/app/lib/backend/resources/arduino-cli.exe"
cd firmware/node/node_capture
"$ARDUINO_CLI" compile --fqbn esp32:esp32:weact_studio_esp32c3 .
"$ARDUINO_CLI" upload -p COM30 --fqbn esp32:esp32:weact_studio_esp32c3 .
```

Then, one command per recording:

```bash
cd C:/Users/HP/.t3/projects/fp-2
python firmware/tools/collect_clips.py --port COM30 --out research/in_domain/session5 \
       --seconds 60 --note "quiet room"
python firmware/tools/collect_clips.py --port COM30 --out research/in_domain/session6 \
       --seconds 60 --note "speech"
python firmware/tools/collect_clips.py --port COM30 --out research/in_domain/session7 \
       --seconds 60 --note "music"
python firmware/tools/collect_clips.py --port COM30 --out research/in_domain/session8 \
       --seconds 60 --note "tv and household"
python firmware/tools/collect_clips.py --port COM30 --out research/in_domain/session9 \
       --seconds 60 --note "phone cry, loud"
python firmware/tools/collect_clips.py --port COM30 --out research/in_domain/session10 \
       --seconds 60 --note "phone cry, one metre away"
```

Then

```bash
for s in 5 6 7 8 9 10; do
  python firmware/tools/label_session.py --session research/in_domain/session$s
  python research/eval_indomain.py --session research/in_domain/session$s
done
```

Each session prints its own confusion matrix at the deployed threshold (0.95) and at the hub's trigger (0.030): **how many windows of that sound would reach the model, and how many of those would raise an alarm.**

What the numbers mean

- **Quiet room** — the number of false alarms must be zero.
- **Speech / music / TV** — the false-alarm rate per loud window. This is the number the demo room will produce.
- **The cry** — recall, and how far away the phone can be before the hub stops asking the model at all (the trigger is RMS 0.030).

Anything that alarms on music goes into the next training round as a hard negative, and the hard negatives go into `research/build_notebook.py` — not into the `.ipynb` by hand.

Entry points

README.md

149 lines · 11,757 bytes

```
# CryNet — infant cry detection for the ESP32-S3 hub

Training pipeline for the cry detector that runs on the **Central Intelligence Hub (ESP32-S3-DevKitC-1)** of the multi-node infant monitor.

**Status:** checkpoint 1 — dataset research done, training notebook written and validated.

---

## What is in this repo

| Path | What it is |
|---|---|
| `notebooks/crynet_training.ipynb` | **The deliverable.** Downloads the corpora, builds the feature set, trains, quantizes to INT8 and exports the firmware artifacts. Run it on Kaggle. |
| `research/` | The dataset investigation behind the notebook: scripts, corpus analysis, and the C↔Python parity test. |
| **`GETTING_STARTED_FLASHING.md`** | **Start here if you have never flashed anything.** The from-zero walkthrough: where every file is, what a COM port is, how to double-click `flash.bat`, and how PlatformIO IDE and the Arduino IDE each fit in. |
| **`FLASHING.md`** | **The reference.** Every option, every escape hatch, the reasoning behind the flash offsets. |
| `firmware/prebuilt/` | **The built flash images** — the hub, the cot node and the wearable, ready to write with `esptool` alone. No compiler, no network. |
| `firmware/prebuilt/flash.bat` | **Flash a unit by double-clicking it.** You press 1, 2 or 3 and type the port. Backs up first, then writes. |
| `firmware/prebuilt/watch.bat` | **Watch what a board says** — the Serial Monitor without the Arduino IDE. |
| `firmware/tools/copy_sketches.bat` | **Copy the node and wearable sketches into the Arduino sketchbook** before building them in the Arduino IDE. Only needed when you have *changed* the code. |
| `firmware/tools/flash_unit.py` | **Flash one unit from the prebuilt images:** `python firmware/tools/flash_unit.py --role node --port COM31`. |
| `firmware/tools/check_prebuilt.py` | **Check that the offline images are up to date with the source** — and say which file moved if they are not. |
| **`dashboard/`** | **The cry record — the caregiver's dashboard.** The hub writes every cry to its own flash and uploads it here when it has the network. `python dashboard/server.py` and it runs: Python's standard library and one SQLite file, no packages, no build step. Start here: **`dashboard/README.md`**. |
| `firmware/tools/test_dashboard_contract.py` | **Prove the hub and the dashboard agree** without any hardware: it builds the hub's upload body byte for byte, posts it, and checks what comes back. |

---

## The dataset decision

The research phase downloaded and profiled every public infant-cry corpus on Kaggle and Hugging Face (audio-level forensics: sample rate, duration, F0, spectral flatness, real-vs-synthetic). The result:

### Used

| Corpus | Role | Why |
|---|---|---|
| **[ICD — `hoyyu1/infant-cry-detection`](https://huggingface.co/datasets/hoyyu1/infant-cry-detection)** | **primary** | 28,478 clips, **all 16 kHz mono / 5 s**, pre-assigned 5 balanced folds. Cries from EnesBabyCries (5,026), CryCeleb2023 (4,552) and iFLYTEK 2020 (3,752) — three independent corpora, hundreds of different infants. Negatives: VoxCeleb speech (4,963), DASEE domestic noise (4,506), human non-speech (2,076), cat meows (1,953), silence (1,650). |
| **[Baby Cry 4-class — `karimeletriby/…`](https://www.kaggle.com/datasets/karimeletriby/infant-cry-classification-4-classes)** | cry diversity + cross-corpus test | 4,871 real cries, 16 kHz, LUFS-normalised, VAD-trimmed, cross-dataset deduplicated (1,297 near-duplicates removed). |
| **[ESC-50](https://github.com/karolpiczak/esc-50)** | negatives + cross-corpus test | 1,960 non-cry clips (speech, laughing, vacuum, clock alarm, door knocks…) plus 40 real `crying_baby` clips kept purely as an unseen cross-domain test. |
| **[Snoring](https://www.kaggle.com/datasets/tareqkhanemu/snoring)** | hard negatives | 1,000 adult-vocal recordings — the confusion class a nursery monitor hits constantly. |

### Rejected, and why

* **`mennaahmed23/baby-cry*` (five Kaggle mirrors)** — the "silence" class is **not silence**: RMS 0.22, peak 1.0, a constant 1225 Hz. The "noise" class is renamed ESC-50 files. Hundreds of files are synthetic `generated3_*` variants. Training on these teaches dataset artifacts, not cries.
* **`rahult25/five-class-…`** — same base corpus plus pitch-shifted derivatives (`uncomfortable_10 … _72`).
* **ICSD (`QingyuLiu1/ICSD`)** — the best-curated corpus in existence (3,641 weakly + 807 real strongly-labelled clips, event-level timestamps) but it is **gated on Hugging Face** and needs a token plus manual approval.
* **`AxonData/infant-cry-detection`** — 50 h of real, expertly-labelled domestic recordings, but the Kaggle upload is an 18-file teaser and the full set is paid.
* **Baby Chillanto (INAOE)** — 2,267 high-quality 16 kHz newborn cries, but no clean public mirror; it is request-only from the lab.
* **AudioSet / FSD50K** — would need YouTube extraction or a 30 GB pull for a handful of clips that ICD already covers.

### Two decisions that changed the design
```

1. **The old trigger threshold was the real bug.** The hub fired an audio pull at RMS ≥ 0.18, so a *soft* cry never reached any classifier. The new design feeds the model anything above ~0.03 RMS and lets it reject false triggers — rejecting speech, cat meows and household noise is exactly what it was trained for.
2. **The front end must match training exactly.** Log-mel computed by librosa and by the firmware's ESP-DSP code differ enough to destroy accuracy. So the notebook *generates* the Mel filterbank and Hann window as C tables, and `out/dsp_crynet.c` is verified against the Python implementation by a compiled parity test (measured deviation **7×10^{-5}**).

Running the training

On Kaggle (recommended)

1. New notebook → *Add Input* → attach nothing.
2. Accelerator **GPU T4**, Internet **On**.
3. Upload `notebooks/crynet_training.ipynb` → *Run All*.

The notebook pulls ~4.6 GB of audio (resumable via `huggingface_hub`), extracts features, trains four CNN variants, picks the best on validation AUC, calibrates the alarm threshold for ≥ 95 % recall, quantizes to INT8 and writes the artifacts to `/kaggle/working/out/`.

Artifact	What it is
`crynet_int8.tflite`	the deployable model
`model_data.h`	the model as a C byte array
`crynet_tables.h`	Mel filterbank + Hann window + quantisation constants
`dsp_crynet.c` / `.h`	portable C front end, numerically identical to training
`crynet_infer_example.c`	TFLite Micro glue: PCM → log-Mel → quantise → `Invoke()` → `P(cry)`
`crynet.keras`	the trained Keras model, so INT8 export can be repeated without retraining
`deploy_config.json`	input shape, quantisation params, operator list, threshold, metrics
`metrics.json`	per-variant AUC / F1 / recall
`c_parity.json`	measured C-vs-Python deviation of the front end

Locally

```bash
pip install numpy librosa soundfile soxr scipy scikit-learn tensorflow
CRYNET_FAST=1 python research/run_notebook_local.py     # smoke test, minutes
```

What was verified before handing it over

Check	Result
Whole notebook runs end to end (data → features → training → mining → INT8 → export)	✅ all 24 cells, on synthetic stand-in archives; and cells 1–15 on the **real** 28,478-clip corpus (feature extraction 128 s for 56,956 windows)
C front end vs. the Python training front end	✅ compiled and compared on 4 probes: **max deviation 7×10^{-5}** (`research/test_c_parity.py`, and again inside the notebook)
Every op in the exported model has a TFLite Micro kernel	✅ `CONV_2D`, `DEPTHWISE_CONV_2D`, `MAX_POOL_2D`, `MEAN`, `FULLY_CONNECTED`, `SOFTMAX` — the notebook asserts this at export time and records the list in `deploy_config.json`
INT8 vs float32 parity	✅ measured in the notebook (AUC delta printed; 0.00000 on the stand-in run)
Corpus integrity	✅ ICD manifest ↔ archives: 13,330 positives and 15,148 negatives, exact 1:1 match, all 16 kHz mono / 5.00 s
Exported model size	✅ ~135 KB INT8 (122,770 parameters) — fits the S3 flash with room to spare

Bugs found and fixed during validation (all would have surfaced on the device, not in training): `tf.py_function` erasing the batch shape, the TFLite converter rejecting the representative-dataset format, the front-end output tensor missing its channel axis, and a runaway silence gate that would have discarded quiet cries.

Measured on the real corpus (indicative)

From a deliberately small local CPU run — **3 epochs on an 8,490-window subsample** of the real data, not the full 74 k / 35-epoch Kaggle configuration. Treat these as a floor, not the final numbers.

Split	AUC	F1	Recall	FPR
ICD val (fold 4)	0.9807	0.9145	0.9504	0.1240
ICD test (fold 5)	0.9839	0.9131	0.9534	0.1058
cross-corpus (cry4 + ESC-50 + snoring, never trained on)	**0.9887**	0.9413	0.9544	0.0789

INT8 vs float32: AUC 0.9806 vs 0.9807 (**delta −0.00007**), 99.70 % decision agreement.

Cry recall on the **unseen** cry4 corpus: `needs` 98.8 %, `physical_pain` 96.9 %, `scared` 90.5 %, `burping` 82.0 %. On ESC-50's `crying_baby` (also unseen): 85 % recall, and 0 % false positives against the other 49 ESC-50 categories.

The weak spot, worth acting on: `coughing` produced a 57 % false-alarm rate, followed by `car_horn` and `church_bells` at 33 %. Coughing is acoustically close to crying — a vocal expulsion — and a baby monitor that alerts on coughs is a nuisance. The notebook therefore gained a **hard-example mining round**: after the main training, it scores its own training set, takes the 600 most cry-like non-cry windows and the 600 least cry-like cry windows, duplicates them, and fine-tunes for a few epochs at a 3× lower learning rate. Only training rows are mined, so the held-out folds stay honest. On the stand-in run this cut the cross-corpus false-alarm rate from 0.109 to 0.004.

If `coughing` is still the top confuser after the full Kaggle run, the next lever is a dedicated cough corpus (e.g. ESC-50's 40

```
cough clips are only partially in training; the FSDKaggle2018 cough set would add several hundred more).
118
119  ---
120
121
122  ---
123
124  ## The firmware contract
125
126  ```
127  int16_t pcm[32000]                 2.000 s @ 16 kHz mono
128    -> float x = pcm / 32768.0
129    -> subtract mean                 DC removal (INMP441 sits at ~+18000 counts)
130    -> zero-pad 256 front / 512 rear
131    -> 81 frames, 512-pt Hann, hop 400
132    -> |rfft|^2                     257 bins
133    -> 64 Mel bands                 librosa slaney mel, 20 Hz - 8 kHz
134    -> 10*log10(max(P,1e-10))       floored at -80 dB
135    -> per-window standardisation   (mean 0, std 1)
136    -> int8 quantise                q = round(v/scale) + zero_point
137    -> TFLite Micro int8 inference  -> P(cry)
138  ```
139
140  Decision rule: `P(cry) >= threshold` → alert. The threshold is *calibrated on the validation fold*, not hard-coded to 0.65.
141
142  ---
143
144  ## Next (checkpoint 2)
145
146  1. Port `dsp_crynet.c` to ESP-IDF, swapping the reference FFT for ESP-DSP `dsps_fft2r` + `dsps_cmplx_mag` (~10× faster).
147  2. Run TFLite Micro with an INT8 input `[1, 81, 64, 1]`; arena in PSRAM if it does not fit in internal SRAM.
148  3. **Record ~50 real clips from your own INMP441** (cries + room noise + speech + TV) and add them as a custom corpus. Matching
your actual microphone, gain and room is the single biggest accuracy win left.
149  4. Decide the alert policy for the wearable (`alert_type` / `confidence_pct` map straight onto the model output).
```

TWO_COT_README.md

102 lines · 4,633 bytes

```
1  # CryNet — two-cot edition
2
3  A separate build of CryNet, the infant cry monitor, for a hub that watches
4  **two cots** instead of three. The three-cot system lives in its own
5  repository and is not modified by anything here.
6
7  The chain is unchanged and was verified end to end before this fork: a
8  microphone on a cot node → a frozen two-second window over ESP-NOW → the
9  INT8 model on the hub → a named cot on the panel and on the caregiver's
10 wrist. What is different here is the **panel layout** and the **link
11 reporting**.
12
13 ## What this edition changes
14
15 **Two cots, enforced.** `NODE_LINK_MAX_NODES` and `HUB_DISPLAY_MAX_NODES`
16 are 2, and the hub's link layer rejects any frame whose node id falls
17 outside 1..2 before it can claim a slot. A third board is not hidden from
18 the panel — it is never heard.
19
20 **The panel, redesigned.** Two full-width cards, one per cot:
21
22 ```
23 ┌──────────────────────────────────┐
24 │ COT 1                  ONLINE    │   name and state
25 │ ▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬▬  RMS 0.012     │   level bar with its number
26 │ ▂▄▆▇  LINK -58 dBm               │   network bar and link
27 └──────────────────────────────────┘
28 ```
29
30 The RMS used to share its line with the link, which put two readings on one
31 row and made the card hard to read in a dark room. Each pair now has a row
32 of its own. There is no battery line on the card: the cots do not measure
33 their cells any more (D-83).
34
35 **A network bar on both ends.** The hub's link layer already measured each
36 cot's RSSI but never exposed it; it now reaches the panel, where each cot
37 gets a four-bar glyph — green at three bars or more, amber below — with the
38 dBm beside it and `LINK --` when there is no reading.
39
40 The cot nodes show the same bar for their own view of the link, on the
41 serial console (`status` prints `link   [####] -58 dBm`). For that reading
42 to be live the hub now sends a **heartbeat every second** — the protocol
43 already defined that frame type and the nodes always understood it, but the
```

```
hub had never sent one.

**There is no connection LED.** `SYSTEM_WIRING.md` 5.1 used to specify a
GPIO10 connection LED on the node, lit only while a valid hub heartbeat had
arrived within a timeout. It worked, and it has been removed (D-83): the
hub's panel already says which cots are answering, and a second, slower
copy of the same answer on the cot itself was saying it in a place nobody
was looking. The heartbeat and its ACK stay — they are how the node knows
the hub is there — and the node prints the same fact on its serial console
(`status`). GPIO10 is free.

## Layout on the bench

| Unit | Notes |
|---|---|
| Hub | ESP32-S3 DevKitC-1, N16R8. TFT on GPIO10–14, backlight on GPIO4 |
| Cot nodes | ESP32-C3 **Pro Mini** × 2, INMP441 on GPIO4/5/6. **Node ids 1 and 2** |
| Wearable | ESP32-C3 SuperMini, SSD1306 OLED, BLE to the hub |

The node ids matter now: set them over the node's serial console with
`id 1` and `id 2`. A node left on a MAC-derived id outside 1..2 prints a
warning at boot, because the hub ignores every other number.

## Building

The hub is a PlatformIO / ESP-IDF project:

```bash
cd firmware/hub_infer
C:/Users/HP/.platformio/penv/Scripts/platformio.exe run
```

The vendored `esp-tflite-micro` and `esp-nn` components are not in the
repository (they are gitignored upstream and pinned in
`firmware/tools/fetch_deps.sh`); copy them into `firmware/hub_infer/components/`
before the first build.

The node and wearable sketches are Arduino projects:

```bash
python firmware/tools/install_sketches.py
arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3 firmware/node/node_crynet
```

The FQBN matters: the generic `esp32:esp32:esp32c3` builds `Serial` as a
UART, and the sketches call `setTxTimeoutMs`, which only the USB-CDC
`HardwareSerial` has.

Status

Built and compile-verified on all three targets: hub RAM 13.0 % / flash
63.6 %, node 75 % flash / 50 % RAM, wearable 51 % flash / 5 % RAM. The
redesigned panel was rendered to a mock (`research/out/two_node_display_mock.png`)
before flashing. The hardware pass is still owed: flash the hub and the two
cots, confirm the two cards and their network bars on the real panel, and
confirm the heartbeat keeps the hub's view of each cot fresh.

`SYSTEM_UPDATE.md` holds the full history and the reasoning behind each
change.
```

firmware/README.md

152 lines · 5,730 bytes

```
# CryNet firmware

Two firmwares live here, and they are built with two different tools on
purpose.

| | Hub | Cot nodes / wearable |
|---|---|---|
| Board | ESP32-S3-DevKitC-1 (N16R8) | ESP32-C3 |
| Build | ESP-IDF (PlatformIO) | Arduino (`.ino` sketches) |
| Where | `hub_infer/` | `node/` (Stage 2b), `wearable/` (Stage 2d) |
| Does | DSP front end, TFLite Micro inference, radio, TFT, BLE | I2S capture, RMS telemetry, ESP-NOW burst, BLE notify |

The hub runs TensorFlow Lite Micro with the ESP-NN SIMD kernels, which is
an ESP-IDF-level component with its own build requirements; the nodes are
ordinary Arduino sketches. Keeping them apart means each one is built the
way that suits it, and the Arduino IDE stays the tool for the boards you
touch most often.

```

````
---

## Hub: build it

The vendored dependencies (the Espressif TFLite Micro port and ESP-NN)
are not tracked in git — they are pinned by commit and fetched on demand.
Do this once after cloning:

```bash
bash firmware/tools/fetch_deps.sh
```

It clones two repositories at fixed revisions and applies
`firmware/tools/patch_deps.py`, which points the TFLite Micro port at the
vendored ESP-NN instead of the component registry so the build stays
offline and reproducible.

Then build:

```bash
cd firmware/hub_infer
python -m platformio run
```

## Hub: flash it

Three ways, all offline, all equivalent.

**1 — one command, no arguments**

```bash
python firmware/tools/flash_hub.py
```

Finds the board, writes the three images, then prints the boot log.

**2 — PlatformIO**

```bash
cd firmware/hub_infer
python -m platformio run -t upload
python -m platformio device monitor
```

**3 — a graphical flasher (Arduino IDE, or anything that takes a .bin and
an offset)**

```bash
python firmware/tools/flash_hub.py --merged --flash-only
```

writes one combined file to
`firmware/hub_infer/.pio/build/esp32s3/crynet_hub_merged.bin`. Select it
in the flasher and flash it at offset **0x0**. The Arduino IDE has no
built-in binary flasher; the **"ESP32 Flash Tools"** extension adds one.

> **The combined image must use DIO flash mode.** The ROM on this board
> cannot read the flash in QIO before the bootloader has set up its own
> timing, and a QIO image makes the chip watchdog-reset in a loop right
> after power-on — the classic symptom is `rst:0x7 (TG0WDT_SYS_RST)`
> repeating on the serial log with no application output. The whole build
> is in fact DIO (`CONFIG_ESPTOOLPY_FLASHMODE "dio"`, and both
> `bootloader.bin` and `firmware.bin` carry flash-mode byte `0x02`), so
> the application runs in DIO too — which costs nothing at this workload.
> `flash_hub.py --merged` forces DIO when it combines the images.

If you flash the three images separately instead, the offsets are:

| File | Offset |
|---|---|
| `bootloader.bin` | `0x0` |
| `partitions.bin` | `0x8000` |
| `firmware.bin` | `0x10000` |

## Hub: read the log

The firmware prints a verification report on every boot: the platform
summary, then for each of four probes the PCM checksum, the front-end
latency, the feature comparison against the host reference, the inference
latency, the class probabilities and the alert decision.

```bash
python firmware/tools/capture_serial.py --port COM32 --reset
```
````

````
`--reset` pulses EN so the whole boot sequence is captured. The Arduino
IDE's Serial Monitor works too — 115200 baud, any COM port.

## Nodes: build and flash them

The node sketches are Arduino sketches and live in `node/`; they are
covered in **[`node/README.md`](node/README.md)** — the wiring, the
console commands, the ESP-NOW framing, and how to collect in-domain audio
from the INMP441.

In short:

```bash
python firmware/tools/install_sketches.py     # copy them into the sketchbook
# then compile and upload from the Arduino IDE
```

## Tools

| Script | What it does |
|---|---|
| `tools/fetch_deps.sh` | clones the pinned TFLite Micro port and ESP-NN, applies the patches |
| `tools/patch_deps.py` | points the port at the vendored ESP-NN (`--check` to verify) |
| `tools/flash_hub.py` | finds the board, flashes it, prints the boot log |
| `tools/capture_serial.py` | reads the serial log, optionally resetting the board first |
| `tools/check_artifact_drift.py` | verifies the model, the quantisation constants and the thresholds against `deploy_config.json` |
| `tools/gen_probe_refs.py` | regenerates the on-device probe reference features from the notebook's own front end |
| `tools/install_sketches.py` | copies the node sketches into the Arduino sketchbook |
| `tools/collect_clips.py` | records in-domain audio from the node and cuts it into clips |
| `tools/verify_burst.py` | checks the node's capture, framing and CRC on the real hardware |

## Layout

```
firmware/
├── hub_infer/                    the ESP-IDF hub project
│   ├── components/
│   │   ├── crynet_model/         the INT8 model as a C array + the constants
│   │   ├── crynet_dsp/           the log-Mel front end
│   │   ├── crynet_infer/         the TFLite Micro wrapper
│   │   ├── esp-tflite-micro/     vendored (not in git)
│   │   └── esp-nn/               vendored (not in git)
│   └── src/                      the application: the Stage 2a probes
├── node/                         the ESP32-C3 node sketches (Arduino)
│   ├── node_crynet/              the node firmware
│   └── node_capture/             the PC data collector
├── tools/                        the scripts listed above
└── README.md
```
````

firmware/node/README.md

299 lines · 11,325 bytes

````
# CryNet nodes — ESP32-C3 + INMP441

Two sketches for the cot nodes. Both are ordinary Arduino sketches and are
compiled and uploaded with the Arduino IDE, the way the rest of the
project's boards are.

| Sketch | What it is for |
|---|---|
| `node_crynet/` | **The node firmware.** Captures 16 kHz mono, keeps the last 2 s in RAM, reports a 100 ms RMS over ESP-NOW, and uploads the 2 s window in CRC-framed fragments when the hub asks. |
| `node_capture/` | **The data collector.** Streams the microphone to the PC over USB so in-domain audio can be recorded. |

The node never classifies anything. The model lives on the hub, and the
hub's front end consumes exactly the PCM bytes this node sends.

---

## Wiring

The INMP441 is a 3.3 V part — do not put it on 5 V.

| INMP441 | ESP32-C3 Pro Mini | Note |
|---|---|---|
| `VCC` | `3V3` | 3.3 V only |
| `GND` | `GND` | |
| `L/R` | `GND` | selects the **left** channel |
| `SCK` | `GPIO4` | I2S bit clock |
````

| `WS` | `GPIO5` | I2S word select — **must** be connected |
| `SD` | `GPIO6` | data from the microphone |

`WS` is not optional: the INMP441 decides which half of the I2S frame it drives from the word-select signal, and leaving it floating gives unsynchronised data.

The pins are at the top of `node_crynet.ino` and `node_capture.ino`; change them there if your breadboard is wired differently.

Why the I2S peripheral is 32-bit *and* stereo

Two things about this peripheral are not what you would guess, and both were measured on the real board rather than reasoned out:

32-bit slots. The INMP441 puts 24 bits of data into a standard I2S frame of two 32-bit slots, so the peripheral is configured for 32-bit slots and the samples are narrowed to int16 in software by keeping the top 16 bits. Narrowing it any other way would shift the audio by a byte.

Stereo, with the right slot dropped. Asking the peripheral for mono *halves the frame* to 32 bits, which breaks the INMP441's framing: the same microphone read the same way produced −3500 counts of DC with the full frame and −620 with the halved one. With the frame intact the DMA delivers the left and the right slot interleaved, and both sketches take every other word — the left slot, which is where L/R grounded puts the microphone. Asking for stereo also makes the Arduino library force the both-slots mask whatever mask is passed, so the mask argument is not doing what it appears to do either.

`node_diag/` prints both configurations side by side; that is where this was settled.

The INMP441's DC output also **drifts for a while after power-up** — about −700 counts in the first seconds, settling towards −50. The hub's front end removes the mean of each two-second window, so it does not matter, but it is why a freshly-booted node reads a different DC offset from one that has been running.

Build and upload

```bash
python firmware/tools/install_sketches.py
```

This copies both sketches into the Arduino sketchbook (`~/Documents/Arduino`), which is where the IDE looks for them. Then in the Arduino IDE:

1. **File → Open** → `node_crynet` (or `node_capture`).
2. **Tools → Board → WeAct Studio ESP32C3** — the entry targets the **ESP32-C3 chip**, not one board: it has the right flash size and the right USB settings for both the **Pro Mini** the cots use and the SuperMini on the wrist. The firmware is board-agnostic; only the silkscreen pin labels differ.
3. **Tools → Port** → the node's port. It enumerates as `USB Serial Device (COM30)`, VID:PID `303a:1001`.
4. **Tools → Upload**.

"USB CDC On Boot" must be enabled

This board's USB connector is wired to the **ESP32-C3's own native USB** (`303a:1001`), not to a CH340. So `Serial` — UART0 on GPIO20/GPIO21 — goes nowhere: those pins are not connected to the connector. With **Tools → USB CDC On Boot: Enabled**, `Serial` becomes the USB CDC device and everything works.

The WeAct board entry has this **on by default**, so choosing the right board is enough. If you select "ESP32C3 Dev Module" instead, you must set *USB CDC On Boot* to *Enabled* yourself, or the sketch will build and upload but the console will be silent and the port will disappear when the application starts.

The baud rate is then nominal — USB CDC is not a real UART — but the tools still open the port at the rate the sketch names, so use the rates given below.

Both sketches were compiled for `esp32:esp32:weact_studio_esp32c3` with the core 3.3.12 that is installed here, and they build with no warnings:

Sketch	Flash	Static RAM
`node_crynet`	975,557 B (74 %)	166,148 B (50 %)

| `node_capture` | 308,808 B (23 %) | 18,760 B (5 %) |

The node firmware's RAM is two 64 KB windows: the live one and the frozen snapshot that gets uploaded.

If the Arduino IDE cannot find the board, the C3's bootloader may be in the wrong mode: hold **BOOT**, tap **RST**, release **BOOT**, and the board reappears as a CDC port.

The node firmware

Console

115200 baud — nominal, because the console is a USB CDC device rather than a real UART. One command per line:

Command	What it does
`status`	the node id, the window state, the RMS, the free heap
`stats`	the counters
`freeze`	freeze the live window now
`burst`	dump the frozen window as frames, in the radio's framing
`id <n>`	set the node id (1..255)
`help`	the list

Telemetry

Every 100 ms the node sends a telemetry frame over ESP-NOW (broadcast, channel 1) carrying the RMS of the last 100 ms, the window length, and flags for "a burst is ready" and "the event level is being crossed".

The RMS is computed over the newest 100 ms with the **mean of the whole 2 s window** removed, so it is the same quantity the hub's front end will compute for that window — the hub's trigger threshold of 0.03 is meaningless otherwise.

The mean is an exact integer sum. A float accumulator loses about eight counts over 32000 samples, and the surviving DC offset is then amplified by the front end's standardisation; that was one of the four bugs the hub's on-device probes caught.

The burst

When the node's RMS crosses the event level (0.10, above the hub's 0.03 trigger so that ordinary room noise does not start an upload) it freezes the live window and increments the burst sequence number. The hub then requests a range of fragments, and the node replies with them.

Sending 286 fragments takes a second or two, and the microphone keeps running into a DMA buffer that is not being read, so the **live** window is dropped and refilled afterwards. The frozen copy that was sent is not affected.

Framing

One framing is used on the radio and on the serial console.

```
 0  0xC3            magic
 1  0x7E            magic
 2  type            1 = telemetry, 2 = burst fragment
 3  node id
 4  u16 burst seq
 6  u16 fragment index
 8  u16 fragment count
10  u16 payload length
12  u16 payload CRC-16/CCITT-FALSE
14  payload
..  u16 frame CRC-16/CCITT-FALSE over everything above
```

A fragment carries 224 bytes of PCM, so a 64,000-byte window is 286 fragments. Requests from the hub are 12 bytes:

```
 0  0xC3  1 0x7E  2 type=3  3 node id
 4  u16 burst seq   6 u16 first fragment   8 u16 fragment count
10  u16 CRC-16/CCITT-FALSE
```

The request's CRC covers its **first 10 bytes**; a frame's own CRC covers its header *and* its payload. A request for a node id of `0xFF` is for every node.

The hub's radio side is Stage 2c.

Checking it without the hub

```bash
python firmware/tools/verify_burst.py --port COM32
```

This asks the node for its frozen window over the serial console, checks every frame's CRC, reassembles the 64,000 bytes and writes them as a WAV. If it passes, then the microphone, the 32-bit-slot conversion, the circular window, the framing and the CRCs are all correct **on the device** — the part that cannot be checked on the host.

Collecting in-domain audio

The deployed model was trained on corpora that no microphone in this room has ever produced: a different capsule, a different pre-amplifier, a different gain, a different room. Until the model has heard this microphone, the reported accuracy is an estimate about somebody else's audio. This is the biggest accuracy gap left.

```bash
python firmware/tools/install_sketches.py     # once
# upload node_capture from the Arduino IDE, then:
python firmware/tools/collect_clips.py --out research/in_domain/session1
```

`collect_clips.py` writes three things into the session folder:

File	What it is
`session.wav`	the whole session, 16 kHz mono 16-bit
`records.csv`	the RMS of every 100 ms record, so the loud parts can be found
`clips/clip_NNNN.wav`	the session cut into 5 s clips, the format the training corpus uses

Each 100 ms record carries its own CRC-16/CCITT-FALSE, over its header and its payload:

```
 0  'C' 'R' 'Y' '1'   magic
 4  u16 sequence
 6  u16 sample count
 8  u32 rms x 1000
12  u16 CRC-16/CCITT-FALSE over bytes 0..11 and the payload
14  int16 samples, little endian
```

A record whose checksum does not check out is **dropped** and counted, rather than written into the WAV. Without it, one dropped byte on the USB link would shift every sample after it, and a dataset of quietly misaligned audio is worse than no dataset.

The PC can stop and resume the stream while it runs: it sends `S` or `R` back to the node, which is a separate wire from the audio, so nothing the PC sends can corrupt the recording.

Bringing the microphone up

`node_diag/` is a bring-up tool, not part of the running system. It reads the I2S stream in **stereo** — both slots, 32 bits each — and prints what is actually on the wire, then scans every exposed GPIO to find where the microphone's data really is.

```bash
python firmware/tools/install_sketches.py
# upload node_diag, then read the console at 115200
```

The INMP441 puts its 24-bit sample in the **left** slot of a 64-bit I2S frame and leaves the **right** slot silent, so a healthy reading is a real DC bias and a noise floor on the left and exactly zero on the right. The diagnostic prints that comparison and ends with a verdict.

It matters that the reading is **repeatable**. The peripheral's start-up transient looks like a signal for the first buffer or two, so the tool measures the original configuration a second time at the end and says whether the two agree.

````
282
283 ## Layout
284
285 ```
286 firmware/node/
287 ├── node_crynet/
288 │   └── node_crynet.ino    the node firmware
289 ├── node_capture/
290 │   └── node_capture.ino   the PC data collector
291 ├── node_diag/
292 │   └── node_diag.ino      the microphone bring-up diagnostic
293 └── README.md
294 ```
295
296 The two sketches repeat the I2S setup rather than sharing a header. The
297 Arduino IDE copies a sketch into a temporary folder before building it and
298 only copies the sketch's own folder, so a header in a sibling folder
299 would not be found.
````

research/README.md

34 lines · 2,916 bytes

````
  1 # research/ — how the dataset decision was made
  2
  3 Everything here is the investigation behind `../notebooks/crynet_training.ipynb`. Nothing in this folder is needed to *run* the notebook on Kaggle — it is kept so the choices can be audited and repeated.
  4
  5 ## Scripts
  6
  7 | Script | What it does |
  8 |---|---|
  9 | `scan_kaggle.py` | Queries the public Kaggle dataset API (no auth needed) and summarises every candidate: size, download count, file tree. |
 10 | `inspect_zips.py` | Reads the archive central directories to reveal folder/class structure, formats and metadata files without extracting anything. |
 11 | `analyze_audio.py` | Audio forensics on samples drawn straight from the archives: sample rate, duration, RMS, autocorrelation F0, spectral centroid, spectral flatness — enough to tell a real cry from a synthetic one. |
 12 | `build_notebook.py` | **Generates `../notebooks/crynet_training.ipynb`.** The notebook is generated rather than hand-written so the code stays reviewable and the build is reproducible. |
 13 | `run_notebook_local.py` | Executes the generated notebook cell by cell locally (so it can be validated before upload). |
 14 | `test_c_parity.py` | Compiles the exported `dsp_crynet.c` and compares its log-Mel against the Python training front end. Uses `gcc` if present, otherwise the `ziglang` wheel (`pip install ziglang`). |
 15 | `make_stub.py` | Builds tiny synthetic stand-in archives so the full pipeline can be exercised without a 4.6 GB download. |
 16
 17 ## What the forensics found
 18
 19 * **The `mennaahmed23` "Baby Cry" family is contaminated.** Its `silence` class measures RMS 0.22 with a constant 1225 Hz — it is not silence. Its `noise` class is renamed ESC-50 files. Hundreds of files are synthetic `generated3_*` derivatives. Training on it teaches dataset artifacts.
 20 * **DonateACry is 8 kHz.** Usable, but upsampling adds no information above 4 kHz, and the class distribution is badly skewed (382 of 457 files are `hungry`).
 21 * **The ICD corpus is uniform**: every clip is 16 kHz mono / 5.00 s, and the manifest matches the archives 1:1 (13,330 positives, 15,148 negatives). Its recordings are quiet (RMS ≈ 0.006) while the negatives are loud (RMS 0.3–0.46) — a level cue the model must not rely on, which is why the front end applies per-window standardisation.
 22 * **TFLite Micro kernel availability is not obvious.** `GlobalAveragePooling2D` lowers to the `MEAN` op; TFLM registers it in `kernels/reduce.cc` (not `mean.cc`, which is why a naive filename search suggests it is missing). The notebook now asserts that every op in the exported model has a TFLM kernel.
 23
 24 ## Local runs
 25
 26 ```bash
 27 # full pipeline on synthetic stand-ins (minutes)
 28 python make_stub.py && CRYNET_FAST=1 CRYNET_OFFLINE=1 python run_notebook_local.py
 29
 30 # C <-> Python parity
 31 python test_c_parity.py
 32 ```
 33
 34 The real corpora are cached under `_cache/` (not committed — see `.gitignore`).
````

The dashboard (Chapter 12)

dashboard/README.md

260 lines · 11,136 bytes

```
# The cry record — the caregiver's dashboard

Every cry the hub hears is written into the hub's own flash. When the hub
has the network, it sends what it has stored here, and this page turns it
into a record a caregiver can read: which cot, when, how long, how loud,
and whether the nights are getting better or worse.

**Nothing here is required for the monitor to work.** With no Wi-Fi, no
router, and this server switched off, the hub hears a cry and buzzes the
wrist exactly as it always did, and keeps the record in its own flash
until a network appears. The dashboard is a reader, not a dependency.

---

## 1. Running it

You need Python 3.10 or newer. Nothing else — no packages, no build step.

```bash
python dashboard/server.py
```

That is the whole installation. It serves on port 8787 and writes its
database to `dashboard/data/crynet.db`. Open the address it prints in a
browser on this machine, or on a phone on the same Wi-Fi.

The server prints the exact line to type into the hub. It looks like:

```
[dashboard]     net set http://192.168.1.50:8787/api/v1/events
```

Copy that line. You will paste it into the hub's serial console in §3.

Options

Flag	What it does
`--port 8787`	Which port to serve on.
`--db path\to\file.db`	Where the record lives. One file; copy it to back it up.
`--host 127.0.0.1`	Serve to this machine only. The default `0.0.0.0` lets the hub and your phone reach it.
`--token SECRET`	Require the hub to send `X-CryNet-Token: SECRET` on every upload. Optional.
`--demo`	Write a week of invented cries into the database and exit, so you can see the page before the hub has sent anything. Use a throwaway `--db` for this.

To see the page with invented data:

```bash
python dashboard/server.py --demo --db dashboard/data/demo.db
python dashboard/server.py --db dashboard/data/demo.db
```

To keep it running after you close the terminal, use whatever your machine
already has — Task Scheduler on Windows, `systemd` or `nohup` on Linux.
There is nothing special about the process; it is one Python file.

---

2. The one thing you must change on your router

> **Set the router's 2.4 GHz channel to 6.**

This is not optional and it is not a preference. The cot nodes and the hub
talk to each other over ESP-NOW on **channel 6, fixed**. The hub has one
radio. When it joins your Wi-Fi it moves that radio to your router's
channel — and if your router is on channel 1 or 11, the cots can no longer
be heard.

That failure is the dangerous kind: the dashboard would look healthy while
the wrist had gone silent. So the hub refuses to allow it. The moment it
associates it checks the channel, and if it is not 6 it drops the
connection immediately, puts the radio back on 6, prints an instruction,
and tries again in five minutes. The monitor always wins over the record.

In the router's admin page, under wireless settings for the **2.4 GHz**
band, change *Channel* from *Auto* to **6**. (5 GHz is irrelevant — the
ESP32 cannot see it.) Then the hub will join and stay.

If you cannot change the router, the system still works perfectly as a
monitor and still records every cry to the hub's flash. It simply has
nowhere to send them until it meets a network on channel 6.

```

3. Pointing the hub at this dashboard

The hub is already built with your Wi-Fi name and password, so it joins on boot without any setup. What it cannot know at build time is *where this server is* — that depends on which machine you run it on.

Connect the hub over USB and open its serial port at 115200. That is the same port its log goes to; the console listens on the same line. Type:

```
net set http://192.168.1.50:8787/api/v1/events
```

using the address the server printed. That is all. The setting is stored in the hub's own flash and **survives a reflash**, so you type it once.

Then check:

```
net
```

which prints whether the Wi-Fi is up, which channel the radio is on, where it is posting, and how much it has sent. To make it send immediately rather than wait for its next turn:

```
upload
```

Every console command for the record

Command	What it does
`history`	The last 10 records, newest first. `history 40` for more.
`log`	How many records the hub holds, how many are still unsent, and whether it knows the time.
`log clear`	Throw the hub's log away. The record numbers carry on, so the dashboard does not confuse new records with old ones. Already-uploaded records on the dashboard are not touched.
`net`	The network: connection, channel, address, what it has sent, last problem.
`net set <url>`	Where to post. An empty value turns uploading off and leaves the record alone.
`net ssid <name> <password>`	Join a different Wi-Fi.
`net name <name>`	The name this hub shows as on the dashboard.
`net token <secret>`	The shared secret, if you started the server with `--token`.
`upload`	Try to send the unsent records now.
`logtest`	Prove, on the hub itself, that a cry is never lost when two cots cry at once. Runs on its own scratch file; your log is untouched. Answers `PASS` or `FAIL`.

The older commands are unchanged: `help`, `status`, `alert <cot>`, `clear`.

---

4. What the page shows

Overview — the headline numbers for the range you pick (7, 14, 30 or 90 days), per cot or for all of them: how many cries, how long they lasted, how many nights were affected, the typical gap between cries. Then the shape of it: cries by day, by night, by hour of the clock, and a day-by-hour heatmap that shows at a glance whether the difficult stretch is 2 a.m. or 5 a.m. The most recent episodes are listed underneath, with the same record number the wrist showed.

Patterns — the analytics: how long cries last (a histogram), how confident the model was, the gaps between cries, night by night, and a plain-language reading of what stands out.

A night is 20:00 to 06:00, named by the evening it starts on. "Night of the 14th" means the night that began on the evening of the 14th and ran into the morning of the 15th. That is how a caregiver thinks about a night, and it is not the same as a calendar day.

Times are shown in your browser's own time zone. The hub records in UTC; the page converts. The hub's own console prints UTC.

Records with no time

The hub has no battery-backed clock (D-90). Until it has reached a time server it knows only how long it has been awake. Those records are marked *untimed*: they are **counted in the totals** — a cry happened, and that is true whether or not the hub knew the date — but they are **never placed on a time axis**, because placing them would be a guess. The page says how many there are. Once the hub reaches the network it learns the time and every record after that is placed properly.

System records

Not every record is a cry. The hub also writes down when it started, when a cot went **deaf** (its audio stopped reaching the hub, so it was not being watched), when that cot came back, and when a test alert was raised from the console. These appear in the system strip, because a quiet stretch on the chart means something quite different if the cot was deaf through it.

---

5. Where the data lives, and how much

On the hub: a 6 MB flash partition, holding up to 8192 records of 32 bytes. At 50 cries a night that is over five months of nights with no network at all. After a successful upload the hub trims to the most recent 2000 records. If it ever fills while offline it drops the oldest and says so out loud, on the console and in `log`.

On the dashboard: one SQLite file, wherever `--db` points. It keeps everything it is given, forever — there is no retention limit. Uploading the same record twice is harmless: records are keyed by hub and record number, so a retry after a dropped connection does not double-count.

On the wire: batches of 32 records, oldest first, over plain HTTP on your own network, or HTTPS if you give it an `https://` address (the hub carries the standard root certificates). The hub marks records as sent only when the server answers that it has them.

The full request and response shapes are in [`API.md`](API.md).

The same dashboard, online

`dashboard/server.py` is the whole offline dashboard: one file, the Python standard library, no installs. But it lives on whichever machine is running it, on that machine's network.

There is now an **online copy** — the same page, the same API, on Cloudflare's free tier (Workers + D1), reachable from anywhere a phone has a signal:

```sh
cd dashboard
npx wrangler deploy        # see dashboard/worker/README.md for the full steps
```

It is live, on the account's `amazingadelowo` subdomain:

```
https://amazings-crynet.amazingadelowo.workers.dev
```

The caregiver opens that address and logs in with the name `caregiver` and the dashboard password; the hub posts to it with the hub's own upload token. Both secrets are Cloudflare Worker secrets — they are not in this repository, and they are not the same secret.

Two things differ from the local copy, both because it is on the public internet:

- **It asks you to log in.** The local copy leaves the reads open (it is on your own network); the hosted one puts a login in front of every read, because an open dashboard on the internet is anyone else's record of your baby's nights.
- **It counts days and nights in your timezone.** The hosted copy runs in UTC; the page tells it which zone it is looking at, so "last night" means your last night.

Hosting changes nothing about the monitor. The hub still records every cry to its own flash first and uploads whenever it can reach the internet; if the hosted copy is down, unreachable or deleted, the monitor behaves exactly as it always has.

---

6. If it is not working

The page says nothing has been uploaded. That is the normal first-run state. Check, in order: is the server running; does `net` on the hub show the right address; does it show *connected*; does it show channel 6.

`net` says "NOT the cots' channel". Your router is not on channel 6. See §2. The hub has deliberately disconnected to protect the cot link.

`net` shows connected but nothing is sent. The address is probably wrong or unreachable from the hub. Check that the machine running the server is on the same network, that its firewall allows the port, and that

```
you used the machine's LAN address (not `localhost` or `127.0.0.1` — those
mean the hub itself). `net` prints the last problem it hit.

**The wrist works but the dashboard is empty.** Correct behaviour when
there is no network. The record is on the hub; `log` shows how much is
waiting.

**Records appear with no time.** The hub has not reached a time server.
It needs the network for that too. See §4.
```

dashboard/API.md

319 lines · 11,619 bytes

```
# The dashboard's API

The dashboard service (`dashboard/server.py`) speaks JSON over HTTP. The
web page in `dashboard/web/` is the only consumer that matters, but the
API is documented here because the **hub** is a consumer too: it uploads
its cry records to `POST /api/v1/events`.

Everything is plain HTTP on the local network. If the service is put
behind the internet, put a TLS-terminating proxy in front of it and have
the hub talk to the proxy.

**The hosted copy.** There is now one: a Cloudflare Worker in
`dashboard/worker/` that serves the same API from a D1 database, on
Cloudflare's free tier. See `dashboard/worker/README.md` for how it is
deployed and pointed at. It differs from the local copy in two ways, and
both matter:

- **Every read needs a login.** The local copy on a home network leaves
  the GETs open; the hosted copy puts a login in front of them, because
  on the internet an open `/api/v1/summary` publishes the baby's whole
  cry record to anyone who finds the URL. The hub is unaffected: it
  sends `X-CryNet-Token` on uploads, and the same token is accepted on
  reads for scripts.
- **Reads may carry `?tz=<IANA zone>`.** Days, hours and nights are
  counted in that zone, which is how a copy hosted in UTC can still show
  the caregiver's own night. The page sends it automatically. With no
  `tz`, the local copy uses the machine's own timezone and the hosted
  copy uses UTC.

---

## `POST /api/v1/events` — the hub uploads a batch

The hub uploads in batches, and it may upload the same batch twice if a
reply is lost on the way back. The store is idempotent: the pair
`(device, seq)` identifies a record, and a record that is already there is
acknowledged but not duplicated.

Request:

```json
{
  "device": "hub-7c9ebd12",
  "hub": { "fw": "2.0-dashboard", "name": "Nursery hub" },
  "events": [
    {
      "seq": 1042,
      "kind": "cry",
      "cot": 2,
      "start": 1794182400,
      "end": 1794182434,
      "duration_s": 34,
      "peak_conf": 99,
      "windows": 5,
      "peak_rms": 0.2134,
      "mean_rms": 0.0981,
      "boot": 118447,
      "uptime_ms": 512340
    }
  ]
}
```

field	type	meaning
`device`	string	the hub's identity (its MAC, in the firmware)
`hub.fw`	string	the firmware version, free-form
`hub.name`	string	a name for the caregiver to read
```

`events[].seq`	int	the hub's own record number; it only ever goes up, and it is what makes an upload idempotent
`events[].kind`	string	`cry`, `deaf`, `heard`, `boot`, or `test`
`events[].cot`	int	which cot the cry was heard on (1-based)
`events[].start`, `events[].end`	int	epoch seconds; `0` means the hub did not know the wall clock when it happened
`events[].duration_s`	int	how long the episode lasted
`events[].peak_conf`	int	the model's highest confidence for that episode, in percent
`events[].windows`	int	how many windows the model judged in that episode
`events[].peak_rms`, `events[].mean_rms`	float	the loudness of the episode
`events[].boot`	int	the hub's boot number, so a record can be traced to a session
`events[].uptime_ms`	int	milliseconds since that boot when the record was written

Response:

```json
{ "accepted": 1, "received": 1, "server_time": 1794182435 }
```

`accepted` is how many records were **new**. The hub advances its
"uploaded" pointer by the number of records it sent, not by `accepted` —
it must not re-send what it already sent.

`GET /api/v1/summary?days=14&cot=0`

The numbers and the series the dashboard draws. `cot=0` means every cot.

```json
{
  "device": { "device": "hub-7c9ebd12", "name": "Nursery hub", "fw": "2.0-dashboard",
              "first_seen": 1793000000, "last_seen": 1794182435 },
  "devices": [ ...same shape... ],
  "range": { "from": 1792978835, "to": 1794182435, "days": 14 },
  "kpis": {
    "total_cries": 37,
    "timed_cries": 35,
    "untimed_cries": 2,
    "cries_per_day": 2.64,
    "total_cry_seconds": 1180,
    "avg_cry_seconds": 31.9,
    "longest_cry_seconds": 214,
    "median_cry_seconds": 27.0,
    "cots_crying": 2,
    "nights_affected": 11,
    "nights_per_week": 5.5,
    "avg_cries_per_night": 3.36,
    "last_cry_at": 1794182400,
    "median_gap_min": 42.0
  },
  "per_cot": [
    { "cot": 1, "cries": 12, "seconds": 420, "max_seconds": 90,
      "avg_seconds": 35.0, "avg_conf": 98.4, "last_at": 1794100000 }
  ],
  "per_day": [
    { "day": "2026-11-20", "cries": 4, "seconds": 130, "cots": [1, 2] }
  ],
  "per_night": [
    { "night": "2026-11-20", "cries": 3, "seconds": 96, "cots": [2] }
  ],
  "per_hour": [ { "hour": 0, "cries": 0, "seconds": 0 }, ... 24 entries ... ],
  "per_weekday": [ { "weekday": 0, "cries": 4 }, ... 7 entries, Monday = 0 ... ],
  "heatmap": {
    "cots": [1, 2],
    "counts": [ [ 24 counts for cot 1 ], [ 24 counts for cot 2 ] ],
    "seconds": [ [ 24 totals for cot 1 ], [ 24 totals for cot 2 ] ]
  },
  "recent": [
    { "seq": 1042, "cot": 2, "kind": "cry", "start": 1794182400,
      "end": 1794182434, "duration_s": 34, "peak_conf": 99, "windows": 5,
      "peak_rms": 0.2134, "mean_rms": 0.0981 }
  ]
}
```

`recent` holds up to the last 400 episodes, oldest first — this is what
the timeline is drawn from.

Records with no time

A record whose `start` is `0` is one the hub made **before it knew the
time** — the hub has no battery-backed clock (D-90). Such a record is
evidence that a cry happened, and it is **counted**:

- `total_cries` includes it;
- `per_cot` includes it — which cot cried is known, even when the date

```
  is not;
- `untimed_cries` says how many there are, and `timed_cries` how many
  have a time.

It is **never placed on a time axis**. It appears in no `per_day`, no
`per_night`, no `per_hour`, no `per_weekday`, no `heatmap` entry, and it
contributes to no gap. `avg_cries_per_night` divides the timed count by
the number of nights, so the two sides of that fraction are like for
like.

For windowing, an untimed record falls back to the moment the dashboard
received it. That is the only honest bound available: we cannot know
when the cry was, but we know when we learned of it.

The same rule applies to `/api/v1/analytics`, where every pattern is a
statement about *when*: untimed records take no part in them, and
`untimed_cries` says how many were left out.

---

## `GET /api/v1/analytics?days=30`

The patterns and the plain-language findings.

```json
{
  "range": { "from": ..., "to": ..., "days": 30 },
  "duration_hist": [ { "label": "0-5s", "count": 2 }, { "label": "5-15s", "count": 9 } ],
  "confidence": { "min": 94, "median": 99.0, "max": 100 },
  "gaps": { "count": 36, "median_min": 42.0, "max_min": 611.0, "min_min": 3.0 },
  "nights": [
    { "night": "2026-11-20", "cries": 3, "seconds": 96, "cots": [2],
      "first": 1794182400, "last": 1794182434 }
  ],
  "per_cot": [
    { "cot": 2, "cries": 21, "seconds": 810, "avg_seconds": 38.6,
      "max_seconds": 214, "median_seconds": 33.0, "by_hour": [ 24 counts ],
      "peak_hour": 3, "peak_share": 42.9, "night_cries": 17 }
  ],
  "untimed_cries": 2,
  "insights": [ "Cot 2 is usually heard around 03:00 (42.9% of its cries fall in that hour)." ]
}
```

`insights` is written to be read by a person. The page shows them as they
are; it must not invent findings of its own.

`GET /api/v1/events?limit=200&cot=0&from=0&to=0&kind=cry`

The raw record, newest first. `from`/`to` are epoch seconds, `cot` is
1-based (0 = every cot), `kind` is one of the kinds above.

```json
{ "events": [ { "device": "...", "seq": 1042, "kind": "cry", "cot": 2,
                "start_at": 1794182400, "end_at": 1794182434, "duration_s": 34,
                "peak_conf": 99, "windows": 5, "peak_rms": 0.2134,
                "mean_rms": 0.0981, "boot": 118447, "uptime_ms": 512340,
                "received_at": 1794182435 } ] }
```

The field names here are not the same as the ones in `summary.recent`.
This endpoint returns the stored columns, so the times are `start_at` and
`end_at`; `summary.recent` returns them as `start` and `end`. A page that
reads both has to know which is which — the shipped page does, and says so
at the top of `web/app.js`.

`GET /api/v1/devices`

```json
{ "devices": [ { "device": "hub-7c9ebd12", "name": "Nursery hub",
                 "fw": "2.0-dashboard", "first_seen": ..., "last_seen": ... } ] }
```

`POST /api/v1/presence` — the hub says who is here

The cots have no network of their own — they talk to the hub over its
radio — so the hub is the only thing that can answer for them. It posts
this **once a minute**, and again immediately whenever a cot's state
changes, whether or not there is anything to record.

The owner still configures **one** address: the presence endpoint is the
```

events endpoint with its last path segment changed, so
`net set https://…/api/v1/events` needs no second setting.

Request:

```json
{
  "device": "hub-7c9ebd12",
  "hub": { "fw": "2.0-dashboard", "name": "Nursery hub" },
  "nodes": [
    { "cot": 1, "online": true, "deaf": false, "rssi": -47 },
    { "cot": 2, "online": false, "deaf": false, "rssi": 0 }
  ]
}
```

field	type	meaning
`nodes[].cot`	int	the cot number, 1-based
`nodes[].online`	bool	the hub has heard from it lately
`nodes[].deaf`	bool	the hub cannot get a window from it, so it cannot announce a cry from it
`nodes[].rssi`	int	the telemetry signal strength in dBm; `0` = unknown

Response:

```json
{ "ok": true, "cots": 2, "server_time": 1794182435 }
```

One row per cot is stored, replaced each time the hub speaks, so what is
stored is always the last thing the hub knew.

`GET /api/v1/presence`

Who is here: the hubs, and the cots each hub can see.

```json
{
  "now": 1794182435,
  "hubs": [
    { "device": "hub-7c9ebd12", "name": "Nursery hub", "fw": "2.0-dashboard",
      "at": 1794182435, "age_s": 0, "online": true, "demo": false }
  ],
  "nodes": [
    { "device": "hub-7c9ebd12", "cot": 1, "online": true,
      "deaf": false, "rssi": -47, "at": 1794182435 }
  ]
}
```

A hub is taken to be **there** if it has spoken in the last three
minutes. The hub says once a minute, so three is three chances missed
before the dashboard calls it gone: long enough that one dropped request
does not make the caregiver think the hub has died, short enough that a
hub that has been switched off is gone before anyone has looked away.

A cot's `online` is what the hub last said about it. `deaf` is a
separate thing: a cot can be answering and still deaf, which means it
is transmitting and the hub cannot get a window from it, so it cannot
announce a cry.

`GET /api/v1/health`

```json
{ "ok": true, "time": 1794182435, "devices": 1 }
```

Notes for the page

- Every timestamp is **epoch seconds in UTC**; the page shows them in the
  browser's own local time.
- `start` of `0` means the hub did not know the wall clock. Those records
  are counted in the totals but are not placed on a timeline.
- The page polls `/api/v1/summary` every 30 seconds. It must not assume
  the numbers change between two polls.
- The page must work with **no data at all**: an empty state that says so,
  rather than a broken chart.

dashboard/worker/README.md

127 lines · 4,806 bytes

```
# The hosted dashboard (Cloudflare Workers + D1, free)

This is the online copy of the caregiver dashboard.  The offline copy is
`dashboard/server.py` — one Python file, no installs — and it keeps
working exactly as before; nothing about hosting changes how the hub
behaves offline.  The hub still records every cry to its own flash and
uploads whenever it can reach the internet.

Everything here runs on Cloudflare's free tier: no credit card, no
server to keep alive, no idle-reclaim.  The limits are far above what
one household generates:

| what                | free allowance            | this system uses             |
|---------------------|---------------------------|------------------------------|
| Worker requests     | 100,000 / day             | ~100 / day (uploads + pages) |
| D1 rows read        | 5,000,000 / day           | the requested window only    |
| D1 rows written     | 100,000 / day             | ~50 / night of cries         |
| D1 storage          | 5 GB total (500 MB / db)  | ~6 MB per *decade* of cries  |

## One-time setup (already done once — this is the record)

Needs only Node.js on the machine.

```sh
cd dashboard
npx wrangler login                                  # opens the browser, click Allow
npx wrangler d1 create crynet                       # prints the database_id
#   paste that id into wrangler.jsonc, then:
npx wrangler d1 execute crynet --remote --file worker/schema.sql
npx wrangler secret put UPLOAD_TOKEN        # the hub's token  (net token <value>)
npx wrangler secret put DASHBOARD_PASSWORD  # the caregiver's login password
```

The username is `caregiver` (change with `DASHBOARD_USER`).

Deploy after a change

```sh
cd dashboard
npx wrangler deploy
```

The page is then at `https://amazings-crynet.<account>.workers.dev/`
— open it, log in with the password, done.

Pointing the hub at it

On the hub's serial console (the port its log goes to):

```
net set https://amazings-crynet.<account>.workers.dev/api/v1/events
net token <the UPLOAD_TOKEN value>
net          # check: URL, token set, Wi-Fi, channel
upload       # send everything stored but not yet confirmed
```

The hub's firmware already trusts the certificate (Cloudflare's edge
chains to GTS Root R4, which is in the ESP-IDF CA bundle the hub was
built with), so nothing on the board changes but those two settings.

Only one address to type. The hub also reports who is here, and it
does that to `POST /api/v1/presence` — but that address is *derived*:
it is the events URL with its last path segment changed. So `net set
https://…/api/v1/events` is the only address the owner ever types.

The token, kept off Git

The upload token is a secret and this repository is public, so it is
not in any tracked file. It lives in three places and nowhere else:
Cloudflare's secret store (`UPLOAD_TOKEN`), the hub's own NVS (set
once with `net token`), and a local, git-ignored `dashboard/.secrets`
on the machine that runs the offline copy:

```
CRYNET_TOKEN=<the UPLOAD_TOKEN value>
CRYNET_PASSWORD=<the DASHBOARD_PASSWORD value>
```

That file is what lets the offline dashboard be started against the same
```

````
token without typing it:

```sh
python dashboard/server.py --token "$(grep CRYNET_TOKEN dashboard/.secrets | cut -d= -f2)"
```

### One thing to know about Cloudflare and Python

Cloudflare's edge **rejects `Python-urllib` with error 1010** before
the request reaches the Worker, so a Python script that posts to the
hosted copy needs its own `User-Agent` header:

```python
req = urllib.request.Request(url, data=body, method="POST", headers={
    "Content-Type": "application/json",
    "X-CryNet-Token": token,
    "User-Agent": "CryNet-Tools/1.0",     # <- without this: 403
})
```

The hub is unaffected: it sends `ESP32 HTTP Client/1.0`, which the
edge passes.

## Backups

D1 keeps 7 days of point-in-time history ("Time Travel") on the free
plan.  For a copy that outlives that:

```sh
npx wrangler d1 export crynet --remote --output crynet-backup.sql
```

Run it now and then and keep the file; it is a plain SQLite dump the
offline server could be seeded from if the online copy ever went away.

## Changing the secrets

```sh
npx wrangler secret put UPLOAD_TOKEN        # then:  net token <new>   on the hub
npx wrangler secret put DASHBOARD_PASSWORD  # the browser asks again next login
```

## Local development without touching the live copy

`npx wrangler dev` serves the Worker on `http://127.0.0.1:8787` with a
local D1 (secrets come from a `.dev.vars` file, one `NAME=value` per
line).  Or skip all of it: `python dashboard/server.py` is the whole
offline dashboard and needs nothing but Python.
````

Flashing, including the offline route

FLASHING.md

798 lines · 32,157 bytes

```
# Flashing the CryNet units — offline, step by step

**Written 2026-11-07, against the D-83 firmware (no battery sense, no status LEDs).**

This guide is written so that **you can put the firmware on any unit yourself** —
a new hub, a spare cot node, a replacement wearable — without asking anyone and
without needing the internet.

Read it with `SYSTEM_WIRING.md` (what to wire) and `SYSTEM_UPDATE.md` (what has happened).

---

> **Have never flashed a chip before? Start with `GETTING_STARTED_FLASHING.md` instead.**
> It is written for someone who has never opened PlatformIO or the Arduino IDE:
> where every file is, what a COM port is, and three ways to do the job —
> double-click a `.bat`, use PlatformIO IDE, or use the Arduino IDE.
>
> This file is the **reference**: every option, every escape hatch, the reasoning.
> The other one is the **walkthrough**.

---

## 0. What you need

```

One program: `esptool`. That is the whole offline path. It is already on this machine:

```
%USERPROFILE%\.platformio\packages\tool-esptool\esptool.exe
```

`esptool` writes a firmware image to a board over USB. It does not compile anything, so it does not need the Arduino core, PlatformIO, a compiler or a network connection.

On a **new** machine, get it one of these ways:

Where	How
Copy the folder	copy `%USERPROFILE%\.platformio\packages\tool-esptool\` to the new machine and put it somewhere on the PATH
pip	`pip install esptool` — this gives `esptool` as a command
Arduino IDE	the IDE ships one too, under `resources\app\lib\backend\resources\`

The helper scripts in this guide are Python and need `pyserial` (`pip install pyserial`). They are a convenience, not a requirement: `esptool` on its own can do every step.

---

1. Get the code and the firmware images

The repository holds the firmware **source** *and* the **built flash images**, so you do not have to compile anything to flash a unit.

```powershell
git clone https://github.com/Amazo123/CryNet-TwoNode.git
cd CryNet-TwoNode
```

No network? Copy the folder across on a USB stick — it works just as well. The images you need are in `firmware/prebuilt/`:

```
firmware/prebuilt/
├── README.md          what is in here and the two rules that matter
├── SHA256SUMS         the hash of every image, so you can prove it is intact
├── BUILD_INFO.json    which source each image was built from, so you can
│                      prove it is up to date
├── hub/                bootloader.bin  partitions.bin  firmware.bin
├── node/               bootloader.bin  partitions.bin  boot_app0.bin  app.bin
└── wearable/           bootloader.bin  partitions.bin  boot_app0.bin  app.bin
```

If the copy you have predates these images, pull the branch that carries them:

```powershell
git fetch origin
git checkout feature/offline-flashing
```

If you are working on the units that are on the bench, the checkout that already exists is `C:\Users\HP\.t3\projects\fp-two-node`. Use that one.

---

2. Check the tools and the images before you touch a board

```powershell
python firmware/tools/flash_unit.py --check
```

It prints something like this:

```
== toolchain ==
  esptool     OK   C:\Users\HP\.platformio\packages\tool-esptool\esptool.exe
  pyserial    OK

== prebuilt images ==
  hub:
    0x000000  bootloader.bin       22560 B  aefe21ad9f40735b...  hash OK
    0x008000  partitions.bin        3072 B  463793a642ab4a90...  hash OK
    0x010000  firmware.bin      1335248 B  f7f09a34a7d115cd...  hash OK
  node:
    0x000000  bootloader.bin       19536 B  9b2fa5e4af7cf282...  hash OK
    0x008000  partitions.bin        3072 B  148b959cbff1c38a...  hash OK
    0x00e000  boot_app0.bin         8192 B  f94c5d786a7a8fab...  hash OK
    0x010000  app.bin             983328 B  aabb406f49335423...  hash OK
  wearable:
```

```
    ...
```

If a line says `HASH MISMATCH`, stop — the image is damaged (a truncated download, a disk problem). Re-copy the folder; do not flash it.

`--check` flashes nothing. It only reads.

---

3. Work out which board is which, **before** you flash anything

This is the step that saves the most grief, because **a COM port number is not an identity.** Windows assigns COM numbers from the USB path and the enumeration order, so the same physical board gets a different number when you move it to another socket or replug it. What identifies a board for good is its **MAC address**, which is burned into the chip.

The map as it stood on 2026-11-07, when the units were last on the bench:

Board	MAC	Port that day	How it appears on USB
Hub (ESP32-S3 DevKitC)	—	COM32	`USB-Enhanced-SERIAL CH343` — a USB-serial bridge
Cot node 1	`90:DA:72:85:C5:44`	COM31	`USB Serial Device` (Espressif native USB)
Cot node 2	`90:DA:72:87:CB:98`	COM37	`USB Serial Device` (Espressif native USB)
Wearable	—	COM30	`USB Serial Device` (Espressif native USB)

Treat that table as a memory aid, not as a script. The numbers will move.

3a. List what is on USB

```powershell
python firmware/tools/flash_unit.py --list
```

or, without Python:

```powershell
Get-CimInstance Win32_PnPEntity |
  Where-Object { $_.Name -match 'COM\d+' -and $_.Name -notmatch 'Bluetooth' } |
  ForEach-Object { "$($_.DeviceID)  |  $($_.Name)" }
```

Read it like this:

- **`USB-Enhanced-SERIAL CH343 (COM32)`** → the **hub**. The S3 DevKitC has a CH343 USB-serial bridge on it.
- **`USB Serial Device (COMxx)`** → an Espressif board on its own native USB. That is a cot node **or** the wearable; you cannot tell them apart from the USB descriptor alone.

3b. Ask each board who it is

```powershell
python firmware/tools/who_is_who.py COM30 COM31 COM32 COM37
```

For each port the tool resets the board over RTS/EN and reads the banner it prints at boot. **It does not flash anything.** A cot node answers with its id and its MAC:

```
CryNet node firmware — Stage 2b
node id 1 (derived from the MAC; change it with "id <n>")
...
node 1  mac 90:DA:72:85:C5:44
```

A node whose id was set by hand says so instead — `node id 2 (saved; the MAC would have said 7)` — which is how you know the id in NVS is yours and not the derived one.

The wearable answers with its OLED line and then its Bluetooth lines:

```
[oled] up
[ble] hub found (30:30:F9:17:2F:80)
[ble] connecting to 30:30:F9:17:2F:80
[ble] hub connected
[ble] state: 2 cot(s), mask 0x00
```

If a board prints something else entirely, it is running some other sketch — flash the right one onto it and move on.

3c. If a node is on the wrong id

The id lives in the board's own NVS and **survives a reflash** — the flash in §5 writes the application, not the NVS partition. A board that has never been named derives its id from its MAC — **the last byte of the MAC, masked and plus one**, so it can be anything from 1 to 16 — and two identical boards can derive the *same* number, so the MAC is what settles a doubt.

To rename a node, over its serial console (115200 baud):

```
id 2          <- sets and saves the id
status        <- prints the id, the MAC, the link bar and the window
```

The two-cot hub watches cots 1 and 2 only. A node left on id 3 — or on a MAC-derived id such as 7 — is ignored by it, and says so on its console: *"WARNING: this hub watches cots 1 and 2 only."*

3d. Identifying a board that has never been flashed

A brand-new board is running whatever the factory put on it, so it will not print a CryNet banner for `who_is_who.py` to read. Read its MAC address directly:

```powershell
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" --chip esp32c3 --port COM31 chip-id
```

That prints the MAC, and it also proves the board answers on that port at all. The MAC is the identity that never changes, so write it down next to the board.

---

4. The order: **hub first, then the nodes, then the wearable**

This is not a preference. The telemetry frame changed in this build:

- the **new** node sends a **32-byte** payload (16 base bytes, then the radio counters);
- the **old** node sent **34 bytes** (the same, plus two bytes of battery voltage in the middle).

The **new hub reads both** — it looks at the payload length and picks the right offsets — so a node that has not been reflashed yet cannot turn the hub's radio counters to noise. But the **old hub only understands 34 bytes.** So:

> **Flash the hub first.** Then a node never meets a hub that misreads it.

---

5. Flash a unit

One command per unit, with **its own port named**. The port is never guessed: a COM number is not an identity (§3), and the incident recorded as **I-40** in `DECISIONS.md` — a flash aimed at the wrong port parking a node in its bootloader — is the reason the scripts refuse to guess.

The hub (ESP32-S3)

```powershell
python firmware/tools/flash_unit.py --role hub --port COM32 --log 25
```

A cot node (ESP32-C3)

```powershell
python firmware/tools/flash_unit.py --role node --port COM31 --log 15
python firmware/tools/flash_unit.py --role node --port COM37 --log 15
```

The wearable (ESP32-C3)

```powershell
python firmware/tools/flash_unit.py --role wearable --port COM30 --log 15
```

`--log 25` reads the boot log for that many seconds after the flash, which is how you see the good lines in §7. Drop it if you do not want it.

The same thing, by hand

Every command above is a single `esptool` call. If you would rather run it yourself — or you are on a machine with no Python — this is exactly what runs:

```powershell
# the hub
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
```

```
    --chip esp32s3 --port COM32 --baud 921600 write-flash `
    0x0     firmware\prebuilt\hub\bootloader.bin `
    0x8000  firmware\prebuilt\hub\partitions.bin `
    0x10000 firmware\prebuilt\hub\firmware.bin

# a cot node or the wearable
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
    --chip esp32c3 --port COM31 --baud 921600 write-flash `
    0x0     firmware\prebuilt\node\bootloader.bin `
    0x8000  firmware\prebuilt\node\partitions.bin `
    0xe000  firmware\prebuilt\node\boot_app0.bin `
    0x10000 firmware\prebuilt\node\app.bin
```

NVS is never written. The offsets skip it (it sits at `0x9000`–`0xE000` on the node, `0x9000`–`0xF000` on the hub), so a node keeps its id across a reflash and the hub keeps its stored settings.

Do not use one combined image for the hub. A combined image has to be written in DIO mode; the ROM on this board cannot read the flash in QIO before the bootloader has set up its own timing, and a QIO image makes the chip watchdog-reset in a loop right after power-on. The three-image write above is the safe one — and the images in `firmware/prebuilt/hub/` are already DIO (the bootloader header says so).

---

6. A brand-new board

Nothing in §5 changes, but two things are worth knowing about a board that has never been flashed before:

1. **The node id.** A brand-new node has no id in NVS, so it derives one from its MAC (the last byte, masked and plus one — 1 to 16). Give it the right one straight away, over the serial console at 115200 baud (§3c): `id 1` for the first cot, `id 2` for the second. **Two new nodes can derive the same id**, and a derived id outside 1–2 is ignored by the hub outright; either way the board warns you on its console. Set the ids by hand — do not trust the derived value.

2. **The wearable** needs nothing: it finds the hub by Bluetooth and takes its state from it.

3. **The hub** needs nothing either. It comes up, runs its self-check and starts listening.

A board of a *different kind*

The prebuilt images are built for the boards this project uses:

Unit	Board the image is built for	What that fixes
hub	ESP32-S3 DevKitC-1 **N16R8**	16 MB flash, 8 MB octal PSRAM, the 16 MB partition table
node	WeAct Studio **ESP32-C3 SuperMini** (`esp32:esp32:weact_studio_esp32c3`)	4 MB flash, native USB, and the board's own pin map
wearable	the same SuperMini	4 MB flash, native USB, SDA 5 / SCL 6 for the OLED

The firmware targets the **chip**, not the board, so a different C3 board usually works — but three things have to line up, and they are easy to get wrong:

- **Flash size.** The image is a 4 MB layout. A board with 2 MB of flash cannot hold it; a board with 8 or 16 MB can, but its partition table has to be the one the image was built with, so **rebuild it for that board** rather than reusing it.
- **Native USB.** The serial console — the only way to set a node's id — comes over the USB port. A board that only has a UART bridge still flashes, but the console is then on the UART pins.
- **The microphone pins.** The node sketch hard-codes the INMP441 on **SCK 4, WS 5, SD 6**. A board that has those pins used for something else will not hear anything.

To build the images for a different board, change the `FQBN` at the top of `firmware/tools/make_prebuilt.py` and rebuild (§10):

```powershell
python firmware/tools/make_prebuilt.py --role node
```

If the new board is an **ESP32-S3 that is not a DevKitC-1 N16R8** (a different flash size or a different PSRAM), do not use these images — the hub image is built for 16 MB of flash and 8 MB of octal PSRAM. Check what it is before you flash it.

---

7. Verify, unit by unit

Unit	Check	Where to look

Hub	`RESULT: PASS — 4 probes`	the boot log
Hub	`node 1 is online`, `node 2 is online`	the boot log
Hub	**no `bat` field** in the `link:` lines	the boot log
Node 1	`node 1  mac 90:DA:72:85:C5:44`, and **no battery line** in `status`	`who_is_who.py COM31`
Node 2	`node 2  mac 90:DA:72:87:CB:98`, and **no battery line** in `status`	`who_is_who.py COM37`
Wearable	`[oled] up` (or the "not found" line) and `[ble] state: 2 cot(s), mask 0x00`	`who_is_who.py COM30`
All	the hub's cot cards show no battery text	the TFT

A good hub boot log looks like this:

```
I (3509) crynet: RESULT: PASS — 4 probes, front end and model agree with the host
I (4363) ili9341: the panel is up: 320 x 240, rotation 2 (madctl 0x80)
I (4367) hub_display: the display is up: 320 x 240
I (4393) link: node 1 is online
I (4395) link: node 2 is online
I (5038) link: node 1: rms 0.0120  seq 1  flags 0x07  crc 0x7438FF19  level 0.0050  streak 1  win 0.0380
```

Two things to read in it:

- **`RESULT: PASS`** — the hub's own self-check agrees with the host about the model's arithmetic. If it says `FAIL`, do not carry on; read the lines above it.
- **there is no `bat` field** in the `link:` lines. In the old build that line ended `... win 0.0100  bat 1520 mV`. Its absence is how you know the new firmware is running.

The panel is optional. If the TFT is not attached the hub says *"the display could not be started"* and carries on — the radio and the Bluetooth server still come up. You just cannot look at the cards.

The OLED is optional on the wearable too. Without it the wearable prints `[oled] SSD1306 not found at 0x3C — check SDA 5 / SCL 6` and carries on. With it, it prints `[oled] up`. Either way it should then find the hub and print `[ble] state: 2 cot(s), mask 0x00`. `2 cot(s)` is the hub telling it how many cots it can hear; `mask 0x00` means no cot is crying right now.

The audio, once the microphones are on

With no INMP441 attached, a node reads `rms 0.0000` and `mean 0` — **that is correct**, not a fault. There is nothing on the data line to be anything but zero.

The node firmware carries the I2S zero-watchdog (D-79): it restarts the I2S peripheral after five seconds of consecutive exact-zero samples, so that a mic which has died does not take the cot silent for good. **A board with no microphone attached produces nothing but exact zeros, so the watchdog will restart the I2S forever on it.** The `status` line shows the count:

```
I2S zero watchdog restarts=0
```

- **restarts=0 with a mic fitted** → correct; a live mic in a quiet room delivers noise, not zeros.
- **restarts climbing with a mic fitted** → the watchdog is firing on the *installed* mic. That is a fault to look at, not something to sit through.

Attach the microphones, then read a live window:

```powershell
python firmware/tools/verify_burst.py --port COM31
```

It writes the node's two-second window out as a WAV and reports its mean, RMS and peak. Then the end-to-end test: play a recorded cry at each cot in turn and watch the panel name the right cot and the wrist buzz. Keep the phone **at least 15 cm** from the microphone — the INMP441 is fixed-gain and saturates closer than that, and the hub correctly refuses a clipped window.

---

8. When something goes wrong

The board is sitting in the bootloader and nothing comes up

This is the most common failure, and it is the one from I-40. A flash that stops partway — wrong chip, wrong port, a yanked cable — leaves the target **in the bootloader**, not running its firmware. It looks dead: no banner, no telemetry, the hub stops hearing it.

Bring it back with `run`:

```powershell
# a cot node or the wearable
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" --chip esp32c3 --port COM31 run
```

```
# the hub
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" --chip esp32s3 --port COM32 run
```

`could not open port 'COMxx'`

The port does not exist — either the board is not plugged in, or **Windows
renumbered it**. Re-run §3a and §3b; do not guess.

`A fatal error occurred: Failed to connect to ESP32: Timed out waiting for packet header`

Two usual causes: the board is in the bootloader (see above), or the wrong board is on
that port. `chip-id` settles which:

```powershell
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" --chip esp32c3 --port COM31 chip-id
```

The flash wrote but the board still behaves oddly

Read what is actually on it. `esptool` can dump the application partition:

```powershell
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" --chip esp32c3 --port COM31 read-flash 0x10000 0x80000
"$env:TEMP\node_app.bin"
```

The hub shows a cot as offline

Look at the hub's log for the node's counters — they say where a burst went:

```
link: node 1: the burst stalled with 274 of 286 fragments (274 frames arrived)
link: node 1: its radio handed over 288 frames since the burst began, 288 of them left it, 0 were refused
```

`handed over == left it` with fragments missing means the frames were **lost in the
air**; `refused` non-zero means the node's radio queue is too small.

The panel is dark

The hub says *"the display could not be started"* when the TFT is absent, and that is
fine — but a **present** TFT that stays dark is a wiring problem: CS 10, MOSI 11,
SCK 12, DC 13, RST 14, backlight 4, 3V3, GND. GPIO35–37 are the Octal PSRAM bus —
never wire them.

---

9. Backup and rollback

Take a backup of each board's whole flash before you flash it. It takes two
minutes and it is the only way back if a flash goes wrong. The helper does it for
you, as the first step:

```powershell
python firmware/tools/flash_unit.py --role node --port COM31 --backup
```

By hand:

```powershell
# a cot node or the wearable (ESP32-C3, 4 MB flash)
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
    --chip esp32c3 --port COM31 read-flash 0x0 0x400000 "$env:TEMP\fullflash_31.bin"

# the hub (ESP32-S3, 16 MB flash) - bigger, so it takes longer
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
    --chip esp32s3 --port COM32 read-flash 0x0 0x1000000 "$env:TEMP\fullflash_hub_32.bin"
```

Rollback is then one command — write the whole thing back at `0x0`:

```powershell
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
    --chip esp32c3 --port COM31 write-flash 0x0 "$env:TEMP\fullflash_31.bin"

& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
    --chip esp32s3 --port COM32 write-flash 0x0 "$env:TEMP\fullflash_hub_32.bin"
```

`write_flash` also works; `write-flash` is the current spelling.

The backups that exist right now, from the 2026-11-06 bench work:

File	Board
`%TEMP%\fullflash_31.bin`	cot node 1 (4,194,304 bytes)
`%TEMP%\fullflash_37.bin`	cot node 2 (4,194,304 bytes)
`%TEMP%\wearable_baseline_COM30.bin`	the wearable (4,194,304 bytes, md5 `0ba262cd9dd6e6f255438e79dbeb729a`)

The node backups were diffed against each other once: the **only** difference was the NVS partition at `0x009000–0x00E000`, so their application firmware was byte-identical.

Copy them out of `%TEMP%` — Windows clears that folder on its own schedule, and a backup that disappears with it was never a backup.

---

10. Changing the firmware

You only need this section when you have **changed the code**. Rebuild the images and refresh the hashes:

```powershell
python firmware/tools/make_prebuilt.py
```

That runs the three builds and rewrites `firmware/prebuilt/SHA256SUMS` and `firmware/prebuilt/BUILD_INFO.json`:

Role	Built with	Needs
hub	PlatformIO / ESP-IDF (`firmware/hub_infer`)	the vendored `esp-tflite-micro` and `esp-nn` components — `bash firmware/tools/fetch_deps.sh` once, **which needs network**
node	`arduino-cli compile`	the ESP32 core 3.3.12 (installed)
wearable	`arduino-cli compile`	the ESP32 core 3.3.12 and `Adafruit_GFX` + `Adafruit_SSD1306` (in the sketchbook)

Then check, and flash:

```powershell
python firmware/tools/check_prebuilt.py    # are the images up to date with the source?
python firmware/tools/flash_unit.py --check
```

Keeping the images in step with the source

Nothing forces `firmware/prebuilt/` to match the code — a sketch can be edited and committed while the `.bin` files stay old, and the next person to flash a board gets firmware that is not in the repository. Three things prevent it:

1. **`BUILD_INFO.json`** records, for every unit, the SHA-256 of every source file that went into it. `check_prebuilt.py` recomputes them and compares; when they differ it names the files that moved. The hashes are of the *content*, with line endings normalised, so the answer is the same on every machine and in CI.
2. **`flash.bat` runs the same check before it writes anything**, in PowerShell (`firmware/prebuilt/check_prebuilt.ps1`) so it needs no Python. It refuses to flash a stale or damaged image.
3. **GitHub checks it on every push.** `.github/workflows/prebuilt-check.yml` rebuilds all three units from the source and fails if the result differs from what is committed.

When `firmware/prebuilt/` is copied on its own, with no source beside it, the freshness check cannot be made — the check says so and carries on. The images are still verified against `SHA256SUMS`, and they were verified as up to date when they were committed.

Two things worth knowing about the builds:

- **Compile the Arduino sketches with one job** (`-j 1`, which `make_prebuilt.py` already does). The parallel build runs the compiler out of memory on this 16 GB machine and dies with `cc1plus.exe: out of memory`.
- **The hub build is reproducible and needs no network** once the two vendored components are on disk. The rebuild of 2026-11-07 produced a `firmware.bin` **byte-identical** to the one that was flashed to the bench units.

If you prefer to build and flash in one step, the older scripts still work and compile from source: `flash_hub.py --port COM32` and `flash_node.py --port COM31` (`--sketch wearable` for the wearable). They are the right tool when you are iterating on the code; the prebuilt path in §5 is the right tool for putting firmware on a board.

---

11. What this build deliberately does **not** do

The battery sense and every status LED are gone by design (**D-83**), not by omission:

- **no divider** — no 220 kΩ, no 100 kΩ, no 100 nF anywhere in the system;
- **no cell-voltage telemetry**, and no battery text on the hub's panel;
- **no status LEDs** — not the node's link LED, not the node's SOC LED, not the hub's BLE-status LED, not the hub's SOC LED.

The consequence to keep in mind: **nothing in the system warns the caregiver that a cell is nearly flat.** Watch the charge by eye.

The battery *power* is untouched: the cells, the LX-LCBST modules, the switches and the boost converters are exactly as they were.

11a. The cry record and the dashboard — hub only

Since **D-87** the hub keeps a record of every cry in its own flash and, when it has the network, uploads it to a dashboard the caregiver can read. **Only the hub changed.** The nodes and the wearable do not need reflashing for this, and the wearable's alert face is exactly as it was — except that the number it prints ("event #1042") is now the same number the dashboard holds for that cry.

Reflash the hub with the same command as ever:

```powershell
python firmware/tools/flash_unit.py --role hub --port COM32 --log 25
```

Two new lines in the boot log say the record is alive:

```
I (xxxx) crylog: storage mounted: 0 of 6291456 bytes used
I (xxxx) crylog: boot 1: 0 records kept, 0 still to send, next number 1
I (xxxx) crynet: hub-7c9ebd12 ready: network "Amazyn", dashboard (not set -- log only)
```

If instead you see *"the cry log could not be opened"*, the hub carries on and still buzzes the wrist — it just keeps no record. Nothing else is affected.

After that, every cry writes **two** more lines — one when the episode opens and one when it closes:

```
I (xxxx) crylog: record 42: the number was made durable in 37 ms
I (xxxx) crylog: record 42: written in 12 ms
```

These are the cheapest possible proof that the record is being kept, and the millisecond figures are the real cost of the write on your hub's flash — the number is measured on the board, not estimated (D-93). A write that took more than a second would be worth mentioning on the mailing list; tens of milliseconds is normal.

The one command that proves the log keeps every cry

Type this on the hub's serial console:

```
logtest
```

It answers:

```
logtest: PASS -- a cry that finishes out of order is still sent
```

What it is checking, and why it matters. Two cots crying at once is the awkward case: the first cry to start holds the lower number but can still be running when the second one ends, so the file of records is *not* in number order. An uploader that searched the file as if it were sorted would skip the record it stepped over — that cry would never reach the dashboard, and the hub would say it had sent everything (D-94). The test runs that exact scenario against the real component on your hub, **on its own scratch file** — your real log and your real counters are not touched, not even for a moment — and puts everything back afterwards.

If it ever answers `FAIL`, do not trust the dashboard's record until it is looked at: type `log` and compare the numbers it reports with the records on the dashboard.

The three things to do once

1. **Set the router's 2.4 GHz channel to 6.** Not optional. The cots talk ESP-NOW on channel 6 and the hub has one radio; on any other channel the hub deliberately drops the Wi-Fi to keep the cots audible. Full explanation in

```
   [`dashboard/README.md`](dashboard/README.md) §2.

2. **Start the dashboard** on a machine on that network:

   ```powershell
   python dashboard/server.py
   ```

   It prints the exact line for step 3.

3. **Tell the hub where it is**, on the hub's serial console:

   ```
   net set http://192.168.1.50:8787/api/v1/events
   ```

   Then `net` to check and `upload` to send now. The setting is stored in NVS and
   survives a reflash — you type it once, not every time.

The hub is already built with the Wi-Fi name and password, so it joins on boot. To
change them: `net ssid <name> <password>`. **The password is plaintext in the
source** (`components/crynet_net/crynet_net.c`); that is the deliberate trade for a
hub that works out of the box (D-92).

What must not change

The monitor is unchanged. With no Wi-Fi, a wrong password, or the dashboard
switched off, the hub hears a cry and buzzes the wrist exactly as before, and keeps
the record in its flash until a network appears. **Verify this after flashing:**
turn the router off, raise a test alert with `alert 1`, and confirm the wrist still
buzzes. Then `log` should show the record waiting.

The full command list and the troubleshooting are in
[`dashboard/README.md`](dashboard/README.md).

---

12. The one-line version

```powershell
python firmware/tools/flash_unit.py --check                                  # once
python firmware/tools/flash_unit.py --role hub     --port COM32 --log 25     # the hub, first
python firmware/tools/flash_unit.py --role node    --port COM31 --log 15     # node 1
python firmware/tools/flash_unit.py --role node    --port COM37 --log 15     # node 2
python firmware/tools/flash_unit.py --role wearable --port COM30 --log 15     # the wearable
```

— with the ports **verified** by §3 first, not assumed.

---

13. The bench card

One page, in the order you actually do it. Print it, or keep it on the screen next
to the boards.

```
1. WHAT IS PLUGGED IN
   python firmware/tools/flash_unit.py --list
   python firmware/tools/who_is_who.py COM30 COM31 COM32 COM37
   -> a MAC, not a COM number, is the identity of a board

2. IS THE TOOLCHAIN GOOD
   python firmware/tools/flash_unit.py --check
   -> every image hash must say OK; HASH MISMATCH means do not flash

3. BACK UP (before every flash)
   python firmware/tools/flash_unit.py --role node --port COM31 --backup
   -> copy the .bin out of %TEMP% somewhere that is not %TEMP%

4. FLASH, HUB FIRST, THEN THE NODES, THEN THE WEARABLE
   python firmware/tools/flash_unit.py --role hub     --port COM32 --log 25
   python firmware/tools/flash_unit.py --role node    --port COM31 --log 15
   python firmware/tools/flash_unit.py --role node    --port COM37 --log 15
   python firmware/tools/flash_unit.py --role wearable --port COM30 --log 15

5. CHECK THE LOG
   hub:        "RESULT: PASS - 4 probes", "node 1 is online", "node 2 is online",
               and NO "bat" field in the link: lines
   node:       "node 1  mac 90:DA:72:85:C5:44" (id 1) / 90:DA:72:87:CB:98 (id 2)
   wearable:   "[ble] state: 2 cot(s), mask 0x00"

6. A NEW NODE NEEDS ITS ID SET (115200 baud, over the USB console)
   id 1        <- the first cot        id 2   <- the second cot
   -> the hub watches cots 1 and 2 only; a MAC-derived id is usually neither
```
```

```
7. NOTHING COMES UP?  IT IS IN THE BOOTLOADER
   & "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
       --chip esp32c3 --port COM31 run

8. ROLL BACK
   & "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
       --chip esp32c3 --port COM31 write-flash 0x0 "$env:TEMP\crynet_backup_node_COM31.bin"
```

The order in step 4 is not a preference: the new hub reads both the old 34-byte and
the new 32-byte telemetry, the old hub reads only 34 — so a node must never meet a
hub that misreads it.

GETTING_STARTED_FLASHING.md

833 lines · 29,622 bytes

```
# Flashing CryNet units — the guide for someone who has never flashed anything

**Written 2026-11-07.** For the D-83 firmware (no battery sense, no status LEDs).

This guide assumes you have **never** flashed a chip before. It does not
assume you know what PlatformIO is, what Arduino IDE is, where the
firmware files live, or what a COM port is. Every step is spelled out.

There are three ways to do this, and you only ever need **one** of them:

| | What you use | When you use it |
|---|---|---|
| **Way A** ⭐ | `flash.bat` — a file you double-click | **Most of the time.** Puts the finished firmware on a board. No IDE, no typing of long commands. |
| **Way B** | PlatformIO IDE (in VS Code) | You changed the hub's code and want to rebuild it and put it on the board. |
| **Way C** | Arduino IDE | You changed the node's or the wearable's code and want to rebuild it. |

Start with **Way A**. You can read B and C later, when you change code.

**Two questions that come up straight away:**

- *"Do I have to run `copy_sketches.bat` every time I upload to a node?"*
  **No.** That step belongs to Way C only, and Way C is only for when you
  have changed the node's code. To put the firmware on a node, use Way A.
  See [C3](#c3-the-one-rule-that-catches-everyone).
- *"How do I know the file it flashes is the current code?"* It checks
  itself, every time, and refuses to write a stale one. See
  [C7](#c7-keeping-the-offline-images-up-to-date).

---

## The one-paragraph version

A microcontroller board has no program on it when it comes out of the box.
"Flashing" means copying a file onto it over the USB cable. The file is
called a **firmware image**. We keep those images finished and checked in
the repository at `firmware\prebuilt\`, so you never have to compile
anything. You plug the board in, you double-click `flash.bat`, you press
`2` (if it is a cot node), you type the port it is on, and it is done.

That is the whole job.

---

# WAY A — flash a board with `flash.bat` (no IDE needed)

## A1. What you need

1. **The board**, plugged into this PC with a **USB-C data cable**.
   A cable that only charges will not work — this is the single most
   common reason a board does not show up. If you have several cables,
   try another one before you suspect the board.

2. **The repository**, which is on this PC at:

   ```
   C:\Users\HP\.t3\projects\fp-two-node
   ```

   You do not need to move or copy it. Everything below happens inside it.

3. **`esptool`** — the one program that writes firmware to a chip. It is
   already on this PC at:
```

   ```
   C:\Users\HP\.platformio\packages\tool-esptool\esptool.exe
   ```

   `flash.bat` finds it by itself. You never type that path.

A2. Where the files are

The firmware images are **already built** and sitting in the repository:

```
C:\Users\HP\.t3\projects\fp-two-node\firmware\prebuilt\
    hub\          bootloader.bin  partitions.bin  firmware.bin
    node\         bootloader.bin  partitions.bin  boot_app0.bin  app.bin
    wearable\     bootloader.bin  partitions.bin  boot_app0.bin  app.bin
```

Those `.bin` files **are** the firmware. Flashing = copying them onto the board. The folder `firmware\prebuilt\` also holds `SHA256SUMS`, a list of fingerprints. `flash.bat` checks every image against that list **before it writes anything**, so a damaged file (a truncated copy, a disk problem) is caught rather than flashed.

The tool you double-click is:

```
C:\Users\HP\.t3\projects\fp-two-node\firmware\prebuilt\flash.bat
```

A3. Do it

1. Plug the board in with the USB cable.

2. Open File Explorer (the yellow folder on the taskbar). Paste this into the address bar at the top and press Enter:

   ```
   C:\Users\HP\.t3\projects\fp-two-node\firmware\prebuilt
   ```

   You should see `hub`, `node`, `wearable`, `flash.bat`, `watch.bat`, `check_prebuilt.ps1`, `list_ports.ps1` and `SHA256SUMS`.

3. **Double-click `flash.bat`.** A black window opens. It shows the path of the `esptool` it found, then checks the images:

   ```
   ok       hub\bootloader.bin
   ok       hub\firmware.bin
   ...
   all 11 images verified.
   ```

   If any line says **BAD** or **MISSING**, it stops there and flashes nothing. Copy the `firmware\prebuilt\` folder again from a good copy and try once more.

   Then the menu appears:

   ```
   Which unit are you flashing?

     1 = hub        the ESP32-S3 board with the screen
     2 = cot node   the ESP32-C3 board with the microphone
     3 = wearable   the ESP32-C3 board with the wrist screen
     4 = just tell me which board is on which port - flashes nothing
   ```

4. **Type the number and press Enter.** `1` for the hub, `2` for a cot node, `3` for the wearable.

5. It lists the USB ports currently on the PC, then asks:

   ```
   Which port is the node on - for example COM31 -
   ```

   Type the port it showed you in the list and press Enter.

   > **What is a port?** Every USB device that can talk to Windows gets a
   > name like `COM31`. The board is one of them. The list printed just
   > above the prompt is the list of what is plugged in *right now*, so
   > just pick the one that is new. `flash.bat` refuses a port that is not
   > in that list, so you cannot flash a board that is not plugged in.

6. It **backs up the board first** (to `C:\Users\HP\CryNet_Backups\`),
   then writes the firmware, then says:

   ```
   DONE.  The node on COM31 now has the CryNet firmware.
   ```

7. Press any key to close the window.

A4. If the board is new and the backup fails

A brand-new board sometimes refuses the backup read. The script notices
and asks:

```
Flash anyway without a backup?  Type y to go ahead:
```

Type `y` and press Enter. Nothing is wrong — a board with no old firmware
on it has nothing worth backing up.

If the board will not connect at all, hold its **BOOT** button while you
plug the USB cable in, then let go. That puts it into its bootloader, and
it will answer.

A5. See what the board says

To read a board's messages (this is the Arduino IDE's "Serial Monitor",
without the IDE), double-click:

```
C:\Users\HP\.t3\projects\fp-two-node\firmware\prebuilt\watch.bat
```

It asks which port, then prints what the board says, live, at 115200 baud.
Press **Ctrl+C** to stop.

What a healthy board says:

Unit	Lines to look for
Hub	`RESULT: PASS — 4 probes`, then `node 1 is online`, `node 2 is online`
Cot node	`node 1 mac 90:DA:72:85:C5:44` (or `node 2 mac ...`)
Wearable	`[oled] up` and `[ble] state: 2 cot(s), mask 0x00`

The hub's `link:` lines must **not** contain a `bat` field — that was
removed in D-83.

A6. Which board is which? (finding out without guessing)

If you have several boards plugged in and do not know which is which,
run `flash.bat` and choose **4**. It reads the MAC address of every board
on every port and tells you which chip is on each:

```
    COM31 ...
      this is an ESP32-C3 - a cot node or the wearable
      MAC: 90:DA:72:87:CB:98
    COM32 ...
      this is an ESP32-S3 - the hub
      MAC: 30:30:F9:17:2F:80
```

Write the MAC down on the board itself. The COM number can change
when you unplug and replug something, or plug into a different USB
socket. The MAC never changes. It is the board's real identity.

The two cot nodes are told apart by their MAC:

Unit	MAC
Cot node 1	`90:DA:72:85:C5:44`
Cot node 2	`90:DA:72:87:CB:98`

A7. A brand-new node needs its number set

A node's number (which cot it is) is stored **on the board itself**, so
it survives a reflash. A brand-new board has no number, so it guesses
one from its MAC — and two new boards can guess the same one.

The hub only listens to **cot 1 and cot 2**. A node on any other number
is ignored, and **the node says so on its own console** at boot:

```
WARNING: this hub watches cots 1 and 2 only. Set this board with "id 1" or "id 2".
```

```

To set it, open `watch.bat` on that node's port and type:

```
id 1
```

for the first cot, or

```
id 2
```

for the second. The node saves it and keeps it through every future flash.

Type `status` to see its number, MAC and link bar. Type `help` for the
full list of commands.

A8. The order to flash in

If you are reflashing a whole set, do the **hub first**.

A new node sends 32 bytes of telemetry; an old one sends 34. A new hub
reads both. An old hub only understands 34. So a node that has not been
reflashed yet, talking to an old hub, is misread.

Hub first, then the nodes, then the wearable.

WAY B — PlatformIO IDE (for changing the hub's code)

You do not need this to flash a board. You need it when you have **edited
the hub's source** and want to rebuild and redeploy.

B1. What PlatformIO IDE is

It is an **add-on to VS Code** (the code editor you may already have).
It adds a sidebar panel with buttons like *Build*, *Upload* and *Monitor*.
VS Code is at:

```
C:\Users\HP\AppData\Local\Programs\Microsoft VS Code
```

The PlatformIO add-on is already installed inside it (you can see it in
the extensions list as `platformio.platformio-ide`). So there is nothing
to install.

B2. Where the hub's files are

```
C:\Users\HP\.t3\projects\fp-two-node\firmware\hub_infer\
    platformio.ini          <- the settings: which board, which chip
    CMakeLists.txt
    sdkconfig.defaults      <- ESP-IDF settings
    partitions_16mb.csv     <- the memory map
    src\                    <- the hub's actual code
        main.c
        probes.c
        link_stage.c
        console.c
    components\             <- the model's DSP, inference and display code
        crynet_dsp\  crynet_infer\  crynet_model\  crynet_proto\
        node_link\   ble_alert\    hub_display\
        esp-tflite-micro\  esp-nn\   <- fetched, see B6
```

`platformio.ini` is the file that says what board this is:

```ini
[env:esp32s3]
platform = espressif32 @ 8.0.0
board = esp32-s3-devkitc-1     ; the ESP32-S3 DevKitC-1 N16R8
framework = espidf
board_build.flash_size = 16MB
upload_speed = 921600
monitor_speed = 115200
```

B3. Open it

1. Start **VS Code**.
2. Menu **File → Open Folder…**
```

3. Pick `C:\Users\HP\.t3\projects\fp-two-node\firmware\hub_infer`
   (the `hub_infer` folder itself, not its parent).
4. VS Code reloads. On the left edge, a **blue alien-head icon** appears
   (the PlatformIO icon). Click it.

That panel is PlatformIO. It is where the buttons live.

B4. The buttons

There are **two** places to click, and they do the same things.

Place 1 — the PlatformIO sidebar. The blue alien-head icon on the far
left of VS Code opens the PlatformIO panel. It has two sections:

- **PROJECT TASKS** — a tree. Expand it and you see your project, then
  General, and under that the four actions:

Task	What it does
Build	Compiles the code into `.bin` files. Takes ~15 s. Writes nothing to the board.
Upload	Builds, then writes the new firmware to the board over USB.
Monitor	Opens a live window of what the board says at 115200 baud.
Clean	Deletes the build output, so the next build starts fresh.

  (There is also **Test** and sections for **Platform** and
  Libraries — you do not need those.)

- **QUICK ACCESS** — a shortcut to the PlatformIO home page.

Place 2 — the editor's title bar. When a source file is open, the
same actions appear as small icons at the **top right of the tab bar**:
a **✓** for Build, a **→** for Upload, a **🗑** for Clean. (If they are
not there, they are switched off — see *Settings* → search
`platformio-ide.showEditorTitleShortcuts` and turn it on.)

Upload is the one that flashes. **Build** alone never touches the
board.

Before you can upload, PlatformIO needs to know which port the board
is on. It usually finds it by itself and shows the name in the bottom
status bar. If it picked the wrong one, or shows "Auto", click the port
field in the **bottom-right status bar** and pick yours from the list.

B5. Build and upload, step by step

1. Plug the hub in.
2. Open `firmware\hub_infer` in VS Code as above.
3. Click the PlatformIO icon → press **Build** (✓). The terminal at the
   bottom shows the compile. It must end with:

   ```
   ========================= [SUCCESS] Took 15.23 seconds =========================
   ```

   ```
   RAM:   [=         ]  13.0% (used 42496 bytes from 327680 bytes)
   Flash: [======    ]  63.6% (used 1334819 bytes from 2097152 bytes)
   ```

4. Check the bottom-right status bar says the right port. If not, click
   it and choose the hub's port.
5. Press **Upload**. It writes the three images and the board reboots.
6. Press **Monitor**. You should see:

   ```
   I (3509) crynet: RESULT: PASS — 4 probes, front end model agree host
   I (4363) ili9341: panel up: 320 x 240, rotation 2 (madctl 0x80)
   I (4393) link: node 1 is online
   I (4395) link: node 2 is online
   ```

   `RESULT: PASS` means the hub's own self-check agrees with the host
   about the model's arithmetic. If it says `FAIL`, stop and read the
   lines above it.

   Press **Ctrl+C** to close the monitor.

B6. The two fetched dependencies

The hub uses Espressif's TFLite Micro port and ESP-NN. They are **not** in
git — they are pinned by commit and fetched once. If the folder
`firmware\hub_infer\components\esp-tflite-micro` or
`firmware\hub_infer\components\esp-nn` is missing, run this **once**
(needs internet):

```bash
bash firmware/tools/fetch_deps.sh
```

On this PC they are already there, so you will not need it.

B7. If you prefer the terminal

VS Code's terminal (Ctrl+`) opens in the project folder. The same three buttons, as commands.

Careful with the command. PlatformIO has its **own** Python, inside `C:\Users\HP\.platformio\penv\`. It is not the `python` you get on the command line, so plain `python -m platformio` says *"No module named platformio"*. Use PlatformIO's own executable instead:

```powershell
cd C:\Users\HP\.t3\projects\fp-two-node\firmware\hub_infer

# build
& "$env:USERPROFILE\.platformio\penv\Scripts\platformio.exe" run

# build and flash
& "$env:USERPROFILE\.platformio\penv\Scripts\platformio.exe" run -t upload

# watch the boot log
& "$env:USERPROFILE\.platformio\penv\Scripts\platformio.exe" device monitor
```

If you prefer the module form, it works with PlatformIO's Python:

```powershell
& "$env:USERPROFILE\.platformio\penv\Scripts\python.exe" -m platformio run
```

B8. Where the built files end up

After a build, the images PlatformIO produced are here:

```
C:\Users\HP\.t3\projects\fp-two-node\firmware\hub_infer\.pio\build\esp32s3\
    bootloader.bin     22,560 bytes   -> goes to 0x0
    partitions.bin      3,072 bytes   -> goes to 0x8000
    firmware.bin    1,335,248 bytes   -> goes to 0x10000
```

Those exact three bytes are also copied into `firmware\prebuilt\hub\` by `make_prebuilt.py`, so Way A and Way B flash the same firmware. The build is **reproducible**: rebuilding gives a byte-identical `firmware.bin` (checked 2026-11-07).

WAY C — Arduino IDE (for changing the node's or the wearable's code)

You need this when you have edited a **`.ino`** sketch. The node and the wearable are ordinary Arduino sketches; the hub is not (it is ESP-IDF).

C1. Why there are two IDEs

	Hub	Cot node / wearable
Board	ESP32-S3 DevKitC-1	ESP32-C3 SuperMini
Build	ESP-IDF (PlatformIO)	Arduino (`.ino`)
Where	`firmware\hub_infer\`	`firmware\node\node_crynet\`, `firmware\wearable\wearable_crynet\`

The hub runs TensorFlow Lite Micro with ESP-NN SIMD kernels, which are ESP-IDF-level components with their own build rules. The nodes are ordinary sketches. Keeping them apart means each is built the way that suits it — and the IDE you touch most often stays simple.

C2. Where the node's and wearable's files are

```
C:\Users\HP\.t3\projects\fp-two-node\firmware\node\node_crynet\node_crynet.ino
C:\Users\HP\.t3\projects\fp-two-node\firmware\wearable\wearable_crynet\wearable_crynet.ino
```

Each `.ino` is one self-contained file. Open it in a text editor and you can read the whole firmware.

C3. The one rule that catches everyone

> **Short answer to "do I have to run `copy_sketches.bat` every time I upload to a node?" — No.**

>
> You only need it when you have **changed the node's code** and want to
> build it in the Arduino IDE. If you are flashing a node that already has
> the firmware, use **Way A** (`flash.bat`) and never touch
> `copy_sketches.bat` at all — Way A writes the finished `.bin` straight
> to the chip and does not involve the IDE or the sketchbook.

Why the copy exists. The Arduino IDE's file picker starts in its own
sketchbook:

```
C:\Users\HP\Documents\Arduino\
```

The real source of truth is in the repository, so if you edit the `.ino`
in the repository and open *that* file in the IDE, you will be building
the **old** copy — this has caught out everyone who tried it.

So: edit in the repository, then copy across. The copy is one
double-click:

```
C:\Users\HP\.t3\projects\fp-two-node\firmware\tools\copy_sketches.bat
```

It copies `node_crynet`, `node_capture` and `wearable_crynet` into
`C:\Users\HP\Documents\Arduino\`, replacing what was there.

When to run it:

You are…	Run `copy_sketches.bat`?
Flashing a node that already has the firmware (Way A)	**No** — use `flash.bat`
Changing the node's code, then building in the IDE	**Yes**, before you build
Just watching what a node says	**No** — use `watch.bat`

C4. Build and upload a node in the Arduino IDE

1. **Only if you changed the node's code:** run `copy_sketches.bat`
   (double-click it). If you did not change the code, skip this and go
   straight to step 2.
2. Start the **Arduino IDE** (it is at
   `C:\Users\HP\AppData\Local\Programs\Arduino IDE\Arduino IDE.exe`).
3. Menu **File → Open…** (or **File → Sketch → Open**). Pick:
   `C:\Users\HP\Documents\Arduino\node_crynet\node_crynet.ino`
4. Menu **Tools → Board → WeAct Studio ESP32C3**
   (if the board list is not showing, click on "Select a Board" in the
   toolbar's first dropdown and type `esp32`).
5. Menu **Tools → Port** → pick the node's port. (Check `flash.bat`
   option 4, or the Arduino IDE's own bottom-right port dropdown, to
   see which port is which board.)
6. Menu **Tools → USB CDC On Boot** should say **Enabled**. It already
   is by default for this board — leave it. This option is what makes the
   serial console run over the native USB; with it off, the console is
   dead and you cannot set the node's number.
7. Click the **→ (Upload)** button in the toolbar. (The → is Upload; the
   ✓ is Verify, which only compiles and tells you the size.)

   The first build takes a minute or two. Arduino prints the sketch size
   at the end:
   ```
   Global variables use 9,437 bytes (3%) of dynamic memory
   ```
8. It reboots the board. **Wait ~8 seconds** before opening the serial
   monitor — the node takes a moment to boot and start its radio.

To watch it, use **Tools → Serial Monitor**, set the baud rate to
115200 at the bottom right, and press the board's **RESET** button.
You should see:

```
CryNet node firmware — Stage 2b
node id 1 (saved)
```

C5. Build and upload the wearable

Same as above, except:

- Open `C:\Users\HP\Documents\Arduino\wearable_crynet\wearable_crynet.ino`
- Board: **WeAct Studio ESP32C3** (same)
- USB CDC On Boot: **Enabled** (same)
- The OLED uses **SDA = GPIO 5, SCL = GPIO 6**

Its libraries (`Adafruit_GFX`, `Adafruit_SSD1306`) are already in the

sketchbook at `C:\Users\HP\Documents\Arduino\libraries\`, so the IDE finds them with no download.

C6. Where the Arduino build's files end up

The Arduino IDE keeps them in a temporary folder that it cleans up. To get a copy that stays put, use the repository's own build script:

```powershell
cd C:\Users\HP\.t3\projects\fp-two-node
python firmware/tools/make_prebuilt.py --role node
python firmware/tools/make_prebuilt.py --role wearable
```

That writes the finished images into `firmware\prebuilt\node\` and `firmware\prebuilt\wearable\` and refreshes `SHA256SUMS`, so they become usable by Way A.

Two things to know about that script:

- It compiles **one job at a time** (`-j 1`). A parallel Arduino build runs the compiler out of memory on this 16 GB PC and dies with `cc1plus.exe: out of memory`.
- **All three units rebuild byte-identical.** Re-checked 2026-11-07: the node's `app.bin` came back as `aabb406f…` and the wearable's as `193cdaaf…`, the same hashes as the shipped images, and the hub's `firmware.bin` as `f7f09a34…`. So a rebuild that changes nothing changes nothing in the image either — which is what makes `SHA256SUMS` a meaningful check.

C7. Keeping the offline images up to date

Way A writes the `.bin` files that sit in `firmware\prebuilt\`. Those files have to stay in step with the code, or you flash something that is not what the repository says. Nothing forces that — a sketch can be edited and committed while the `.bin` files stay old — so there are three guards.

1. `flash.bat` refuses to flash a stale image. Before it writes anything it runs `check_prebuilt.ps1`, which checks two things:

```
all 11 images verified.
up to date  hub  (52 source files)
up to date  node  (1 source files)
up to date  wearable  (1 source files)
the images are up to date with the source (3 of 3 units checked).
```

The first line says the images are intact (their SHA-256 matches `SHA256SUMS`). The rest says they were built from the source that is in the tree *now*. If the code has moved on, it says which files moved:

```
node: the source has changed since these images were built
      changed  firmware/node/node_crynet/node_crynet.ino
```

and flashes nothing.

2. The same check runs on its own, with no compiler needed:

```powershell
cd C:\Users\HP\.t3\projects\fp-two-node
python firmware/tools/check_prebuilt.py
```

3. GitHub checks it on every push. The workflow `.github\workflows\prebuilt-check.yml` rebuilds all three units from the source and fails if the result differs from what is committed. So a change to the code that does not rebuild the images cannot pass review.

If you copy `firmware\prebuilt\` on its own — onto a laptop with just `esptool` on it — there is no source next to it, so the freshness check cannot be made. It says so plainly and carries on:

```
not checked  hub  (the source is not in this folder)
```

The images are still verified against `SHA256SUMS`, so they are known to be intact. They were known to be up to date when they were committed.

When you change the code yourself, rebuild and commit the result:

```powershell
cd C:\Users\HP\.t3\projects\fp-two-node
python firmware/tools/make_prebuilt.py
```

That rebuilds all three units, rewrites `SHA256SUMS`, and writes
`BUILD_INFO.json` — the record of which source each unit was built from.
Commit `firmware\prebuilt\` along with your change.

When it fails

"There is no port called COM31 on this PC right now."

The board is not plugged in, or Windows gave it a different number.
Run `flash.bat` option 4 to see what is actually there.

`could not open port 'COMxx'`

Same cause. Check the cable — **a charge-only USB cable is the usual
reason**. Then check the port list again.

`Failed to connect to ESP32-C3: No serial data received`

The board did not answer. In order of likelihood:

1. **It is in its bootloader.** Hold **BOOT** while plugging it in,
   release, and retry. If the flash was interrupted, this is the usual
   state — it looks dead, prints nothing, and the hub stops hearing it.
2. **Wrong port.** Run `flash.bat` option 4.
3. **The wrong chip type was assumed.** `flash.bat` picks the chip from
   the unit you chose, so this only happens if a hub is plugged into
   the port you are using for a node.

To bring a board out of its bootloader:

```powershell
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" --chip esp32c3 --port COM31 run
```

(`--chip esp32s3` for the hub.)

The flash wrote, but the board still misbehaves

Read what is actually on it:

```powershell
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" --chip esp32c3 --port COM31 read-flash 0x10000 0x80000
"$env:TEMP\node_app.bin"
```

The hub says a cot is "lost in the air"

`refused` non-zero means the node's radio queue is too small, not that
the link is broken.

The hub's screen stays dark

The hub prints "the display could not be started" when the TFT is
missing — that is fine. If the TFT **is** there but the screen stays
dark, it is a wiring problem: CS 10, MOSI 11, SCK 12, DC 13, RST 14,
backlight 4, 3V3, GND. **GPIO 35–37 are the octal PSRAM bus — never wire
them.**

The node shows `rms 0.0000`

That is correct when no microphone is attached. There is nothing on
the data line but zero. The node's I2S zero-watchdog restarts the
peripheral after five seconds of exact zeros, so the cot does not go
permanently silent. Check the counter with `status`:

```
I2S zero watchdog restarts=0
```

- `restarts=0` with a mic fitted → correct.
- `restarts` climbing with a mic fitted → a fault to look at.

Backup and rollback

Take a backup before every flash. `flash.bat` does this
automatically, to `C:\Users\HP\CryNet_Backups\<unit>_<port>_backup.bin`.

> Windows clears `%TEMP%` on its own schedule. A backup left there
> disappears. `flash.bat` uses `C:\Users\HP\CryNet_Backups\`, which it
> does not.

To roll back, write the backup file back at address `0x0`:

```powershell
& "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
  --chip esp32c3 --port COM31 write-flash 0x0 "$env:USERPROFILE\CryNet_Backups\node_COM31_backup.bin"
```

The one-page bench card

Print this, or keep it on screen next to the boards.

```
1. WHAT IS PLUGGED IN
   flash.bat, choose 4          -> MAC of every board on every port
   write the MAC on the board   -> COM numbers move, MACs do not

2. FLASH IT
   flash.bat, choose 1/2/3, type the port
   hub FIRST, then nodes, then wearable

3. SET A NEW NODE'S NUMBER (USB console, 115200)
   id 1     first cot
   id 2     second cot
   -> the hub only listens to cots 1 and 2

4. CHECK THE LOG (watch.bat, or Arduino IDE Serial Monitor)
   hub:      "RESULT: PASS - 4 probes", "node 1 is online", "node 2 is online"
             and NO "bat" field in the link: lines
   node:     "node 1 mac 90:DA:72:85:C5:44"  (id 1)
             "node 2 mac 90:DA:72:87:CB:98"  (id 2)
   wearable: "[ble] state: 2 cot(s), mask 0x00"

5. NOTHING COMES UP?  IT IS IN ITS BOOTLOADER
   & "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
     --chip esp32c3 --port COM31 run

6. ROLL BACK
   & "$env:USERPROFILE\.platformio\packages\tool-esptool\esptool.exe" `
     --chip esp32c3 --port COM31 write-flash 0x0 `
     "$env:USERPROFILE\CryNet_Backups\node_COM31_backup.bin"
```

Why the firmware is written in pieces, not as one file

`flash.bat` writes each image at a fixed address:

Unit	Address	File
Hub	`0x0`	`bootloader.bin`
	`0x8000`	`partitions.bin`
	`0x10000`	`firmware.bin`
Node / wearable	`0x0`	`bootloader.bin`
	`0x8000`	`partitions.bin`
	`0xe000`	`boot_app0.bin`
	`0x10000`	`app.bin`

These offsets skip the NVS partition on purpose. NVS lives at
`0x9000`–`0xE000` on the C3 boards and `0x9000`–`0xF000` on the hub, and
it is where a node keeps its number. Writing over it would wipe that.
One merged image would fill the gaps with `0xFF` and destroy it.

The hub's images are DIO, not QIO. The ROM on the ESP32-S3 cannot
read flash in QIO mode before the bootloader has set up its own timing,
so a QIO image makes the chip watchdog-reset in a loop the moment it
powers up. The bootloader in `firmware\prebuilt\hub\bootloader.bin`
already has the DIO header, so writing these three files is safe. The
same holds for the C3 boards.

What changed in D-83 (so you recognise the log)

- **No battery sense.** No divider, no 100 nF capacitor, no telemetry of
  the cell voltage. The node's telemetry frame is 32 bytes, not 34.
- **No status LEDs.** Not the node's link LED, not the node's SOC LED,
  not the hub's BLE-status LED, not the hub's SOC LED. The wearable's

```
  OLED carries the status instead.

The consequence to keep in mind: **nothing in the system warns the
caregiver that a cell is nearly flat.** Watch the charge by eye. The
battery *power* side is untouched — cells, LX-LCBST modules, switches and
boost converters are exactly as they were.
```

firmware/prebuilt/README.md

65 lines · 3,026 bytes

```
# Prebuilt flash images

The built firmware for all three units, so a board can be flashed **offline** with
`esptool` alone — no Arduino core, no PlatformIO, no compiler, no network.

`FLASHING.md` is the procedure. The short version:

```powershell
python firmware/tools/flash_unit.py --check                                  # verify
python firmware/tools/flash_unit.py --role hub     --port COM32 --log 25    # the hub, first
python firmware/tools/flash_unit.py --role node    --port COM31 --log 15
python firmware/tools/flash_unit.py --role wearable --port COM30 --log 15
```

What is in here

Folder	Chip	Files (offset in flash)
`hub/`	ESP32-S3	`bootloader.bin` (0x0), `partitions.bin` (0x8000), `firmware.bin` (0x10000)
`node/`	ESP32-C3	`bootloader.bin` (0x0), `partitions.bin` (0x8000), `boot_app0.bin` (0xe000), `app.bin` (0x10000)
`wearable/`	ESP32-C3	the same four

`SHA256SUMS` holds the SHA-256 of every image. `flash_unit.py --check` compares the
files against it and refuses to flash if anything does not match.

`BUILD_INFO.json` holds the record of what each image was built from: for every unit,
the SHA-256 of every source file that went into it. `check_prebuilt.py` recomputes
those and compares, so a sketch that was edited while the `.bin` files stayed old is
caught before anyone flashes it.

Keeping them up to date

```powershell
python firmware/tools/check_prebuilt.py     # are the images current with the source?
python firmware/tools/make_prebuilt.py      # rebuild them and refresh the record
```

`flash.bat` runs the same check (in PowerShell, so it needs no Python) before it
writes anything, and refuses to flash an image that is stale or damaged. GitHub runs
it on every push: `.github/workflows/prebuilt-check.yml` rebuilds all three units and
fails if the result differs from what is committed.

Two rules that matter

Write the regions, not one merged file. A merged image fills the gaps between the
regions with `0xFF`, and writing that would **erase NVS** — which is where a node
keeps its id. The offsets above skip NVS on purpose (`0x9000`–`0xE000` on the C3
boards, `0x9000`–`0xF000` on the hub), so a reflash keeps the id.

The hub images are DIO. The ROM on the S3 cannot read the flash in QIO before the
bootloader has set up its own timing; a QIO image makes the chip watchdog-reset in a
loop right after power-on. The bootloader header in `hub/bootloader.bin` says DIO.

Rebuilding them

```powershell
python firmware/tools/make_prebuilt.py
```

That compiles all three from source, rewrites `SHA256SUMS` and writes `BUILD_INFO.json`.
It needs the Arduino core 3.3.12 and, for the hub, the vendored `esp-tflite-micro` and
`esp-nn` components (`bash firmware/tools/fetch_deps.sh`, once, with network).

The hub build is reproducible: rebuilding it produced a `firmware.bin` byte-identical
to the one flashed to the bench units on 2026-11-07.
```

Appendix B — Complete first-party source listings

Every line of first-party code in the system, with line numbers, reproduced verbatim from the working tree: the hub's ten components and its application, the cot node sketch, the wearable sketch and the motor's bench sketch, the training pipeline, the evaluation scripts, the host tools, the dashboard (the offline service, the page and the hosted Worker) and the record that guards the shipped offline images. Third-party code (TensorFlow Lite Micro, ESP-NN, NimBLE, ESP-IDF, Arduino) is not reproduced — it is fetched by the build. The 8,604-line generated model array `crynet_model/include/model_data.h` is omitted as pure data; it is regenerated from the INT8 artefact by the export cell.

103 files reproduced in full.

Hub - build configuration and application

`firmware/hub_infer/platformio.ini`

51 lines · 2,334 bytes

```
; CryNet hub — inference core (Stage 2a)
; ESP32-S3-DevKitC-1 N16R8: 16 MB flash, 8 MB octal PSRAM
; PlatformIO / ESP-IDF framework (framework = espidf)
;
; The project is a normal ESP-IDF project: firmware/hub_infer/CMakeLists.txt
; sets up EXTRA_COMPONENT_DIRS for the vendored TFLite Micro port, and
; src/CMakeLists.txt declares the main component.

[env:esp32s3]
platform = espressif32 @ 8.0.0
board = esp32-s3-devkitc-1
framework = espidf

; The framework is the one the platform ships: platform-espressif32 8.0.0
; bundles framework-espidf 4.60100.0, which is ESP-IDF 6.1.0. That is well
; above the >= 5.1 that the TFLite Micro port asks for.
;
; Do NOT pin framework-espidf with a `platform_packages` URL here. PlatformIO
; 6.2 installs a URL-specified framework by treating the archive as an npm
; package: it reads the wrong package.json out of the tree and leaves
; packages/framework-espidf holding only a test-fixture package instead of
; the framework, so CMake then fails with "include could not find requested
; file: .../tools/cmake/project.cmake". The framework is now installed
; manually and pinned by packages/framework-espidf/.piopm.
;
; Keep managed_components between builds: the IDF component manager downloads
; espressif/esp-nn there, and PlatformIO otherwise wipes it on every
; configure.
board_build.esp-idf.overwrite_managed_components = no

; --- memory layout -------------------------------------------------------
board_build.flash_mode = qio
board_upload.flash_size = 16MB
board_build.partitions = partitions_16mb.csv

; --- build ---------------------------------------------------------------
; The optimisation level is set the ESP-IDF way, in sdkconfig.defaults
; (CONFIG_COMPILER_OPTIMIZATION_PERF). The Arduino-style
; -DCOMPILER_OPTIMIZATION_PERF macro does nothing here.
build_flags =
    -DBOARD_HAS_PSRAM
    -DCORE_DEBUG_LEVEL=3
    # The air counters (how many frames the radio hears that carry the
    # CryNet magic, against how many the ESP-NOW layer hands up) are
    # what settle whether a fragment that never arrives was lost in the
    # air or dropped by the hub's receive path.  Build with
    # -DCRYNET_LINK_PROMISCUOUS=1 to bring them back.
```

```
    -DCRYNET_LINK_PROMISCUOUS=1

monitor_speed = 115200
upload_speed = 921600
```

firmware/hub_infer/src/CMakeLists.txt

14 lines · 711 bytes

```
#
# Main component of the CryNet hub.
#
# The app is deliberately thin at this stage: Stage 2a is about proving
# that the front end and the model behave on the real chip, so the entry
# point runs the verification probes and reports measurements.  Stage 2c
# adds the radio: link_stage.c listens to the nodes and runs the windows
# they send through the verified front end.
#
idf_component_register(
    SRCS "main.c" "probes.c" "link_stage.c" "console.c"
    INCLUDE_DIRS "."
    REQUIRES crynet_model crynet_dsp crynet_infer crynet_proto node_link ble_alert hub_display crynet_clock crynet_log
crynet_net spiffs esp_http_client esp_netif esp_wifi mbedtls esp_timer nvs_flash spi_flash driver
)
```

firmware/hub_infer/src/main.c

317 lines · 13,374 bytes

```
/*
 * CryNet hub — inference core (Stage 2a).
 *
 * Stage 2a exists to de-risk the firmware before any radio, display or BLE
 * code is written. On the real ESP32-S3 this app proves:
 *
 *   1. the front end produces the same features as the Python training
 *      pipeline (four probes, compared byte for byte against references
 *      generated on the host from the notebook's own front end);
 *   2. the INT8 model fits and runs under TFLite Micro on this MCU;
 *   3. inference latency and tensor-arena usage are measured, not guessed;
 *   4. the decision rule (silence gate + calibrated threshold) fires on a
 *      cry-like window and stays quiet on everything else.
 *
 * The three-node -> hub -> wearable transport, the TFT UI and the BLE GATT
 * server arrive in later stages, once this core is verified.
 */

#include <stdio.h>
#include <stdint.h>
#include <string.h>

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_chip_info.h"
#include "esp_heap_caps.h"
#include "esp_system.h"
#include "esp_log.h"
#include "esp_flash.h"
#include "esp_timer.h"
#include "nvs_flash.h"

#include "crynet_model.h"
#include "crynet_dsp.h"
#include "crynet_infer.h"
#include "probes.h"
#include "link_stage.h"
#include "console.h"
#include "ble_alert.h"
#include "hub_display.h"
#include "crynet_clock.h"
#include "crynet_log.h"
#include "crynet_net.h"

static const char *TAG = "crynet";

/* Worst per-byte difference we accept between the device's quantised
 * features and the host's. The host reference and the device compute the
 * same maths in float32; the only difference is the last bit of a
```

```
 * transcendental function, so a handful of LSBs may differ. */
#define FEATURE_TOLERANCE 1

/* The probe buffers: 90 KB in all.  They are only needed while the
 * probes run, but as file-scope statics they would sit in internal RAM
 * for the life of the program — and the Bluetooth stack needs that RAM
 * far more than the probes do (NimBLE's store configuration was
 * failing to allocate on a board whose radio worked perfectly).  So
 * they live in PSRAM and are handed back when the probes are done. */
static int16_t *s_pcm = NULL;
static float   *s_mel = NULL;
static int8_t  *s_q = NULL;

/* PSRAM if it is there, internal SRAM if it is not. */
static void *alloc_probe_buffer(size_t bytes)
{
    void *p = heap_caps_malloc(bytes, MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT);
    if (p == NULL) {
        p = heap_caps_malloc(bytes, MALLOC_CAP_8BIT);
    }
    return p;
}

static void free_probe_buffers(void)
{
    heap_caps_free(s_pcm);
    heap_caps_free(s_mel);
    heap_caps_free(s_q);
    s_pcm = NULL;
    s_mel = NULL;
    s_q = NULL;
}

static void report_platform(void)
{
    esp_chip_info_t chip;
    esp_chip_info(&chip);

    /* ESP-IDF 6.x changed the flash API: esp_flash_get_size() now takes a
     * chip handle (NULL selects the default chip) and writes the size
     * through an out-parameter. */
    uint32_t flash_bytes = 0;
    esp_flash_get_size(NULL, &flash_bytes);

    ESP_LOGI(TAG, "chip       : rev %d, %u cores", chip.revision, chip.cores);
    ESP_LOGI(TAG, "psram      : %d bytes (%s)",
             (int)heap_caps_get_total_size(MALLOC_CAP_SPIRAM),
             heap_caps_get_total_size(MALLOC_CAP_SPIRAM) > 0 ? "present" : "ABSENT");
    ESP_LOGI(TAG, "free int   : %d bytes", (int)heap_caps_get_free_size(MALLOC_CAP_INTERNAL));
    ESP_LOGI(TAG, "flash      : %u bytes", (unsigned)flash_bytes);
    ESP_LOGI(TAG, "model      : %u bytes (int8)", (unsigned)crynet_model_size());
    ESP_LOGI(TAG, "threshold  : %.4f (the model's own)   announce at %.4f"
                  "   silence gate %.6f",
             (double)CRYNET_THRESHOLD, (double)CRYNET_CLEAR_CRY,
             (double)CRYNET_SILENCE_RMS_GATE);
    ESP_LOGI(TAG, "input quant: scale %.9f  zero point %d",
             (double)CRYNET_INPUT_SCALE, CRYNET_INPUT_ZERO_POINT);
}

/* Compare the device's quantised features against the host reference. */
static bool check_features(int probe)
{
    const int8_t *expected = crynet_probe_expected(probe);
    int worst = 0;
    int exact = 0;
    int mismatched = 0;

    for (int i = 0; i < CRYNET_FEATURE_COUNT; i++) {
        const int d = (int)s_q[i] - (int)expected[i];
        const int ad = (d < 0) ? -d : d;
        if (ad > worst) {
            worst = ad;
        }
        if (ad == 0) {
            exact++;
        } else {
            mismatched++;
        }
    }

    const bool ok = (worst <= FEATURE_TOLERANCE) && (mismatched <= CRYNET_FEATURE_COUNT / 100);
    ESP_LOGI(TAG, "  features  : max delta %d, %d/%d exact, %d differing -> %s",
             worst, exact, CRYNET_FEATURE_COUNT, mismatched, ok ? "PASS" : "FAIL");
    return ok;
}
```

```

void app_main(void)
{
    ESP_LOGI(TAG, "================================================");
    ESP_LOGI(TAG, "CryNet hub — inference core (Stage 2a)");
    ESP_LOGI(TAG, "================================================");
    report_platform();

    /* The Bluetooth stack keeps its keys in NVS, and the store has to be
     * there before the host starts. */
    esp_err_t nvs_err = nvs_flash_init();
    if (nvs_err == ESP_ERR_NVS_NO_FREE_PAGES ||
        nvs_err == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        nvs_err = nvs_flash_init();
    }
    ESP_ERROR_CHECK(nvs_err);

    /* ---------------- front end ---------------- */
    static crynet_dsp_ctx_t dsp;
    if (!crynet_dsp_init(&dsp)) {
        ESP_LOGE(TAG, "front end: cannot allocate the working buffers");
        return;
    }
    ESP_LOGI(TAG, "front end  : %u bytes of working memory (%s)",
             (unsigned)dsp.bytes, dsp.from_psram ? "PSRAM" : "internal SRAM");

    /* ---------------- interpreter ---------------- */
    crynet_infer_t *infer = crynet_infer_create();
    if (!crynet_infer_ready(infer)) {
        ESP_LOGE(TAG, "interpreter: %s", crynet_infer_error(infer));
        crynet_infer_destroy(infer);
        crynet_dsp_free(&dsp);
        return;
    }
    ESP_LOGI(TAG, "interpreter: ready, arena %u bytes reserved, %u used",
             (unsigned)crynet_infer_arena_bytes(infer),
             (unsigned)crynet_infer_arena_used(infer));

    /* ---------------- probes ---------------- */
    s_pcm = alloc_probe_buffer(CRYNET_PROBE_SAMPLES * sizeof(int16_t));
    s_mel = alloc_probe_buffer(CRYNET_FEATURE_COUNT * sizeof(float));
    s_q = alloc_probe_buffer(CRYNET_FEATURE_COUNT * sizeof(int8_t));
    if (s_pcm == NULL || s_mel == NULL || s_q == NULL) {
        ESP_LOGE(TAG, "the probe buffers could not be allocated");
        return;
    }

    int failures = 0;

    for (int p = 0; p < CRYNET_PROBE_COUNT; p++) {
        ESP_LOGI(TAG, "--- probe %d: %s ---", p, crynet_probe_name(p));

        crynet_probe_pcm(p, s_pcm);

        const uint32_t pcm_crc = crynet_probe_pcm_crc(p);
        const uint32_t want_pcm_crc = crynet_probe_expected_pcm_crc(p);
        if (pcm_crc != want_pcm_crc) {
            ESP_LOGE(TAG, "  pcm       : CRC 0x%08x, expected 0x%08x -> FAIL",
                     (unsigned)pcm_crc, (unsigned)want_pcm_crc);
            failures++;
            continue;
        }
        ESP_LOGI(TAG, "  pcm       : CRC 0x%08x matches the host", (unsigned)pcm_crc);

        const int64_t t0 = esp_timer_get_time();
        crynet_dsp_features(&dsp, s_pcm, s_mel);
        const int64_t t1 = esp_timer_get_time();
        crynet_dsp_quantise(s_mel, s_q);
        const int64_t t2 = esp_timer_get_time();

        ESP_LOGI(TAG, "  front end : %.2f ms  (quantise %.2f ms)",
                 (double)(t1 - t0) / 1000.0, (double)(t2 - t1) / 1000.0);

        if (!check_features(p)) {
            failures++;
        }

        float p_no_cry = 0.0f, p_cry = 0.0f;
        const int64_t t3 = esp_timer_get_time();
        const bool ran = crynet_infer_run(infer, s_q, &p_no_cry, &p_cry);
        const int64_t t4 = esp_timer_get_time();
        if (!ran) {
            ESP_LOGE(TAG, "  inference : FAILED (%s)", crynet_infer_error(infer));
            failures++;
```

```
            continue;
        }

        const float rms = crynet_dsp_rms(s_pcm);
        const bool silent = rms < CRYNET_SILENCE_RMS_GATE;
        /* The same rule the live path uses, so the banner and the
         * system agree.  It used to be the model's own threshold alone,
         * and the banner said ALERT for the noise probe -- a window
         * the hub would have refused, so the boot report and the
         * running system contradicted each other. */
        const bool alert = (!silent) && (p_cry >= CRYNET_CLEAR_CRY);

        ESP_LOGI(TAG, "  inference : %.2f ms   P(no_cry)=%.4f  P(cry)=%.4f",
                 (double)(t4 - t3) / 1000.0, (double)p_no_cry, (double)p_cry);
        ESP_LOGI(TAG, "  decision  : rms %.6f  %s -> %s",
                 (double)rms,
                 silent ? "below the silence gate" : "above the silence gate",
                 alert ? "ALERT" : "no alert");
    }

    /* The probes are done with their buffers; hand them back so the
     * radio and the Bluetooth stack have the whole internal heap. */
    free_probe_buffers();
    ESP_LOGI(TAG, "free int   : %d bytes after the probes",
             (int)heap_caps_get_free_size(MALLOC_CAP_INTERNAL));

    ESP_LOGI(TAG, "================================================");
    if (failures == 0) {
        ESP_LOGI(TAG, "RESULT: PASS — %d probes, front end and model agree "
                      "with the host", CRYNET_PROBE_COUNT);
        ESP_LOGI(TAG, "================================================");

        /* The caregiver's phone connects to the hub over BLE and is told
         * when a cot hears a cry.  The hub's own link to the nodes is
         * ESP-NOW, so the two radios coexist — and the radio comes up
         * before the Bluetooth stack on purpose: NimBLE's pools and the
         * Wi-Fi driver's buffers come out of the same internal RAM, and
         * when BLE claimed its memory first the Wi-Fi driver ran out
         * mid-init and the ESP-NOW link never started.  The radio takes
         * what it needs while the heap is whole, and BLE lives on what
         * is left. */
        /* The hub's memory comes up before its radios.  Six megabytes
         * of the flash were reserved for this from the start and never
         * mounted; now they hold every cry the hub hears, whether or
         * not there is a network to send them to.  A flash that will
         * not mount is a warning, not a stop -- the monitoring is the
         * product, the record is a benefit. */
        crynet_clock_init();
        if (crynet_log_init()) {
            crynet_clock_restore(crynet_log_last_epoch_ms());
            /* The hub's own start is a record too: a gap in the cry
             * history means either a quiet night or a hub that was off,
             * and only this tells them apart. */
            crynet_log_write(CRYNET_LOG_BOOT, 0, 0, 0, 0, 0, 0);
        } else {
            ESP_LOGW(TAG, "the cry log could not be opened; "
                          "the hub carries on without a record");
        }

        if (!link_stage_radio_start()) {
            ESP_LOGW(TAG, "the radio could not be started; "
                          "the hub carries on without it");
        }
        ESP_LOGI(TAG, "free int   : %d bytes with the radio up",
                 (int)heap_caps_get_free_size(MALLOC_CAP_INTERNAL));
        /* The uploader shares the radio with ESP-NOW, so its interface
         * is created here, while the internal heap is still whole --
         * the same reason the radio comes up before Bluetooth.  With
         * no network configured it starts nothing and costs nothing. */
        if (!crynet_net_start()) {
            ESP_LOGW(TAG, "the uploader could not be started; "
                          "cries are still recorded in the hub's flash");
        }
        if (!ble_alert_start()) {
            ESP_LOGW(TAG, "the Bluetooth alert server could not be started; "
                          "the hub carries on without it");
        }
        ESP_LOGI(TAG, "free int   : %d bytes with Bluetooth up",
                 (int)heap_caps_get_free_size(MALLOC_CAP_INTERNAL));
        /* The panel is optional too: a dead display does not stop the
         * monitoring, it only stops the showing. */
        if (!hub_display_start()) {
            ESP_LOGW(TAG, "the display could not be started; "
                          "the hub carries on without it");
        }
```

```
        /* The console answers on the same UART the log is written to.
         * It is started last, so that every command it accepts can be
         * carried out: the radio is up, the Bluetooth server is up,
         * and a test alert has somewhere to go. */
        console_start();
        link_stage_run(&dsp, infer);
    } else {
        ESP_LOGE(TAG, "RESULT: FAIL — %d check(s) failed", failures);
        ESP_LOGE(TAG, "================================================");
        crynet_infer_destroy(infer);
        crynet_dsp_free(&dsp);
    }
}
```

firmware/hub_infer/src/probes.h

55 lines · 2,150 bytes

```
/*
 * CryNet — verification probes for Stage 2a.
 *
 * The host parity test proves that the C front end matches the Python
 * training pipeline on a desktop. These probes carry that check onto the
 * ESP32-S3: the device generates the same PCM from the same closed-form
 * definitions, runs its own front end, and compares the quantised
 * features against the ones the *Python* pipeline produced on the host.
 *
 * The expected features live in crynet_probe_refs.h, which is generated by
 * firmware/tools/gen_probe_refs.py from the notebook's own front end.
 *
 * Probe definitions (must stay in step with gen_probe_refs.py):
 *
 *   0  silence      every sample is 0
 *   1  dc_bias      every sample is 18000 — the INMP441's quiescent bias,
 *                   which the front end must remove completely
 *   2  tone         0.35*sin(2*pi*420*t) + 0.12*sin(2*pi*1350*t)
 *   3  noise        0.05 * uniform(-1, 1) from a 32-bit LCG
 *
 * t = n / 16000. PCM conversion is int16 = (int16)(clip(x, -1, 1) * 32767),
 * truncating toward zero.
 */

#ifndef CRYNET_PROBES_H
#define CRYNET_PROBES_H

#include <stdint.h>

#define CRYNET_PROBE_COUNT        4
#define CRYNET_PROBE_SAMPLES      32000
#define CRYNET_PROBE_FEATURE_BYTES (81 * 64)

/* Name of a probe, for the log. */
const char *crynet_probe_name(int index);

/* Fill `pcm` with CRYNET_PROBE_SAMPLES int16 samples for the given probe. */
void crynet_probe_pcm(int index, int16_t *pcm);

/* CRC32 (zlib convention) over the probe's PCM — lets the host confirm
 * that both sides generated bit-identical input. */
uint32_t crynet_probe_pcm_crc(int index);

/* Expected quantised features, from the Python pipeline. */
const int8_t *crynet_probe_expected(int index);

/* CRC32 over each probe's expected features (from the host). */
uint32_t crynet_probe_expected_crc(int index);

/* CRC32 over each probe's expected PCM (from the host) — the device
 * compares its own generated PCM against this to prove both sides built
 * bit-identical input. */
uint32_t crynet_probe_expected_pcm_crc(int index);

#endif /* CRYNET_PROBES_H */
```

firmware/hub_infer/src/probes.c

149 lines · 4,005 bytes

```c
/*
 * CryNet — verification probes for Stage 2a.
 *
 * See probes.h for the definitions; they must stay in step with
 * firmware/tools/gen_probe_refs.py.
 */

#include <math.h>
#include <string.h>

#include "probes.h"
#include "crynet_probe_refs.h"   /* generated: expected features */

#include "esp_heap_caps.h"

static const char *const kProbeNames[CRYNET_PROBE_COUNT] = {
    "silence",
    "dc_bias",
    "tone",
    "noise",
};

const char *crynet_probe_name(int index)
{
    if (index < 0 || index >= CRYNET_PROBE_COUNT) {
        return "?";
    }
    return kProbeNames[index];
}

static double clip(double v)
{
    if (v > 1.0) {
        return 1.0;
    }
    if (v < -1.0) {
        return -1.0;
    }
    return v;
}

/* 32-bit LCG, 24 high bits used as a uniform in [0, 1). */
static uint32_t lcg_next(uint32_t *state)
{
    *state = (uint32_t)(1103515245u * (*state) + 12345u);
    return (*state >> 8) & 0x00FFFFFFu;
}

void crynet_probe_pcm(int index, int16_t *pcm)
{
    switch (index) {
    case 0: /* silence */
        memset(pcm, 0, CRYNET_PROBE_SAMPLES * sizeof(int16_t));
        break;

    case 1: /* the INMP441's quiescent DC bias, and nothing else */
        for (int i = 0; i < CRYNET_PROBE_SAMPLES; i++) {
            pcm[i] = (int16_t)18000;
        }
        break;

    case 2: { /* two tones */
        for (int i = 0; i < CRYNET_PROBE_SAMPLES; i++) {
            const double t = (double)i / 16000.0;
            const double x = 0.35 * sin(2.0 * M_PI * 420.0 * t)
                           + 0.12 * sin(2.0 * M_PI * 1350.0 * t);
            pcm[i] = (int16_t)(clip(x) * 32767.0);
        }
        break;
    }

    case 3: { /* broadband noise */
        uint32_t state = 12345u;
        for (int i = 0; i < CRYNET_PROBE_SAMPLES; i++) {
            const double u = (double)lcg_next(&state) / 16777216.0;
            const double x = 0.05 * (2.0 * u - 1.0);
            pcm[i] = (int16_t)(clip(x) * 32767.0);
        }
        break;
    }

    default:
        memset(pcm, 0, CRYNET_PROBE_SAMPLES * sizeof(int16_t));
        break;
```

```
    }
}

static uint32_t crc32_buf(const void *data, size_t len)
{
    const uint8_t *p = (const uint8_t *)data;
    uint32_t crc = 0xFFFFFFFFu;

    for (size_t i = 0; i < len; i++) {
        crc ^= p[i];
        for (int b = 0; b < 8; b++) {
            const uint32_t mask = (uint32_t) - (int32_t)(crc & 1u);
            crc = (crc >> 1) ^ (0xEDB88320u & mask);
        }
    }
    return ~crc;
}

uint32_t crynet_probe_pcm_crc(int index)
{
    if (index < 0 || index >= CRYNET_PROBE_COUNT) {
        return 0;
    }
    /* A 64 KB buffer, needed only for the four probe CRCs at boot.  As
     * a function-local static it would hold 64 KB of internal RAM for
     * the life of the program, so it is borrowed from PSRAM and given
     * straight back. */
    int16_t *pcm = heap_caps_malloc(CRYNET_PROBE_SAMPLES * sizeof(int16_t),
                                    MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT);
    if (pcm == NULL) {
        pcm = heap_caps_malloc(CRYNET_PROBE_SAMPLES * sizeof(int16_t),
                               MALLOC_CAP_8BIT);
    }
    if (pcm == NULL) {
        return 0;
    }
    crynet_probe_pcm(index, pcm);
    const uint32_t crc = crc32_buf(pcm, CRYNET_PROBE_SAMPLES * sizeof(int16_t));
    heap_caps_free(pcm);
    return crc;
}

const int8_t *crynet_probe_expected(int index)
{
    if (index < 0 || index >= CRYNET_PROBE_COUNT) {
        return NULL;
    }
    return crynet_probe_refs[index];
}

uint32_t crynet_probe_expected_crc(int index)
{
    if (index < 0 || index >= CRYNET_PROBE_COUNT) {
        return 0;
    }
    return crynet_probe_ref_feature_crc[index];
}

uint32_t crynet_probe_expected_pcm_crc(int index)
{
    if (index < 0 || index >= CRYNET_PROBE_COUNT) {
        return 0;
    }
    return crynet_probe_ref_pcm_crc[index];
}
```

firmware/hub_infer/src/console.h

43 lines · 1,603 bytes

```
/*
 * CryNet hub — the bench console.
 *
 * The hub logs to the same UART it listens on, and until now it only
 * ever spoke.  This gives it a way to answer, so that the parts of the
 * system that can be checked without making a noise can be checked:
 *
 *   help              what the console understands
 *   status            the cots: their level, their window, their state
 *   alert <cot>       publish a test alert for that cot, as if it had
 *                     heard a cry -- the caregiver's phone buzzes and
 *                     the wearable shows it
```

```
 *   clear             take the test alert back
 *
 * The test alert is the reason the console exists.  The last unproven
 * link in the system was the BLE one: the hub published "a cry on cot
 * 1 at 100%" and the wearable was never attached to see it.  With this
 * command the whole BLE path -- the hub's publisher, the Bluetooth
 * link, the wearable's client and its display -- can be exercised at
 * any time, in any room, without waking anybody.
 *
 * The console is deliberately small: no history, no editing, no
 * commands that can change how the system behaves.  A line is read, a
 * word is looked up, and the answer is printed.
 */

#ifndef CONSOLE_H
#define CONSOLE_H

#ifdef __cplusplus
extern "C" {
#endif

/* Start the console task.  It reads the UART the hub logs to, so the
 * log and the console share the port -- a console line appears in the
 * log, and the log never disturbs a command. */
void console_start(void);

#ifdef __cplusplus
}
#endif

#endif /* CONSOLE_H */
```

firmware/hub_infer/src/console.c

444 lines · 16,685 bytes

```
/*
 * CryNet hub — the bench console.
 *
 * The hub has always logged to its UART; this gives it a way to answer
 * as well.  The commands are few and none of them changes how the
 * system behaves:
 *
 *   help         what the console understands
 *   status       the cots: their level, the window they have, their state
 *   alert <cot>  publish a test alert for that cot, as if it had heard
 *                a cry.  The caregiver's phone buzzes; the wearable
 *                shows "a cry on cot N at 100%"
 *   clear        take the test alert back
 *
 * and, for the record the hub keeps and the dashboard it feeds:
 *
 *   history [n]  the last n records, newest first (10 by default)
 *   log          how many records the hub holds and how many are unsent
 *   log clear    throw the log away; the record numbers carry on
 *   net          the network: whether it is up, and what it has sent
 *   net set <url>    where to post, e.g.
 *                    net set http://192.168.1.50:8787/api/v1/events
 *   net ssid <s> <p> which Wi-Fi to join
 *   net name <name>  the name the dashboard shows for this hub
 *   net token <t>    the shared secret, if the dashboard wants one
 *   upload       try to send the unsent records now
 *
 * None of the network commands changes what the hub does with a cry.
 * With no Wi-Fi and no address the hub records to its own flash and
 * tells the wrist, exactly as it did before the dashboard existed.
 *
 * Why it exists: the last unproven link in the system was the
 * Bluetooth one.  The hub published "a cry on cot 1 at 100%" and the
 * wearable was not attached to see it, so the whole path -- publisher,
 * link, client, display -- was untested.  With this command it can be
 * tested at any time, in any room, without a sound.
 *
 * The console reads the same UART the hub logs to, so a command and
 * the log share the port.  That is why the reader is on its own task
 * and why it only ever reads whole lines: a line of log arriving while
 * a command is being typed is discarded, and a command is only acted
 * on when a newline ends it.
 */

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>
#include <time.h>

#include "driver/uart.h"
#include "esp_err.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

#include "console.h"
#include "crynet_clock.h"
#include "crynet_log.h"
#include "crynet_net.h"
#include "link_stage.h"

static const char *TAG = "console";

/* The longest command the console accepts, and the longest line it
 * will hold while it waits for the newline.  A dashboard address can be
 * a hundred and sixty characters, so the line has to carry one whole. */
#define CONSOLE_LINE_MAX 200

/* The cots the status command reports. */
#define CONSOLE_MAX_COTS 8

static void print_help(void)
{
    ESP_LOGI(TAG, "commands:");
    ESP_LOGI(TAG, "  help          this list");
    ESP_LOGI(TAG, "  status        the cots and what they are hearing");
    ESP_LOGI(TAG, "  alert <cot>   publish a test alert for that cot");
    ESP_LOGI(TAG, "  clear         take the test alert back");
    ESP_LOGI(TAG, "  history [n]   the last n records, newest first");
    ESP_LOGI(TAG, "  log           how much is recorded and how much is unsent");
    ESP_LOGI(TAG, "  log clear     throw the log away");
    ESP_LOGI(TAG, "  net           the network and what it has sent");
    ESP_LOGI(TAG, "  net set <url> where to post the records");
    ESP_LOGI(TAG, "  net ssid <s> <p>  which Wi-Fi to join");
    ESP_LOGI(TAG, "  net name <name>   this hub's name on the dashboard");
    ESP_LOGI(TAG, "  net token <t>     the dashboard's shared secret");
    ESP_LOGI(TAG, "  upload        send the unsent records now");
    ESP_LOGI(TAG, "  logtest       prove, on this chip, that out-of-order "
                  "records are not lost");
}

/* The kind of record, in the word the caregiver would use. */
static const char *kind_name(uint8_t kind)
{
    switch (kind) {
    case CRYNET_LOG_CRY:   return "cry";
    case CRYNET_LOG_DEAF:  return "deaf";
    case CRYNET_LOG_HEARD: return "heard";
    case CRYNET_LOG_BOOT:  return "boot";
    case CRYNET_LOG_TEST:  return "test";
    default:               return "?";
    }
}

/* When a record happened.  The hub has no clock of its own: until it
 * has reached a time server it knows only how long it has been awake,
 * and it says so rather than inventing a date. */
static void format_when(char *out, size_t cap, const crynet_log_event_t *e)
{
    if (e->epoch_s != 0) {
        const time_t t = (time_t)e->epoch_s;
        struct tm tm_utc;
        gmtime_r(&t, &tm_utc);
        strftime(out, cap, "%Y-%m-%d %H:%M:%SZ", &tm_utc);
    } else {
        snprintf(out, cap, "up %lus (no clock)",
                 (unsigned long)(e->uptime_ms / 1000u));
    }
}

/* The history command.  Ten records at a time is what fits on a screen
 * without scrolling the useful part away. */
static void print_history(int want)
{
    if (want < 1) {
        want = 1;
    }
    if (want > 40) {
        want = 40;
    }
    crynet_log_event_t recs[40];
```

```
    const int n = crynet_log_recent(recs, want);
    if (n == 0) {
        ESP_LOGI(TAG, "the log is empty");
        return;
    }
    ESP_LOGI(TAG, "the last %d record(s), newest first "
                  "(times are UTC; the dashboard shows your own):", n);
    for (int i = 0; i < n; i++) {
        const crynet_log_event_t *e = &recs[i];
        char when[40];
        format_when(when, sizeof when, e);
        if (e->kind == CRYNET_LOG_CRY) {
            ESP_LOGI(TAG, "  #%-5lu %-5s cot %u  %s  %lu.%lus  "
                          "peak %u%%  rms %.4f/%.4f  %u windows",
                     (unsigned long)e->seq, kind_name(e->kind),
                     (unsigned)e->cot, when,
                     (unsigned long)(e->duration_ms / 1000u),
                     (unsigned long)((e->duration_ms % 1000u) / 100u),
                     (unsigned)e->peak_conf,
                     (double)e->peak_rms_milli / 1000.0,
                     (double)e->mean_rms_milli / 1000.0,
                     (unsigned)e->windows);
        } else {
            ESP_LOGI(TAG, "  #%-5lu %-5s cot %u  %s",
                     (unsigned long)e->seq, kind_name(e->kind),
                     (unsigned)e->cot, when);
        }
    }
}

static void print_log(void)
{
    const uint32_t total = crynet_log_total();
    const uint32_t pending = crynet_log_pending_count();
    const uint32_t dropped = crynet_log_dropped();
    ESP_LOGI(TAG, "the log holds %lu record(s), %lu still to send, "
                  "boot %lu, device %s",
             (unsigned long)total, (unsigned long)pending,
             (unsigned long)crynet_log_boot_id(), crynet_net_device_id());
    ESP_LOGI(TAG, "the clock %s",
             crynet_clock_known() ? "knows the time" :
             "does not know the time yet (records are marked by uptime)");
    if (dropped != 0) {
        ESP_LOGW(TAG, "%lu record(s) were lost: the log filled up while "
                      "the hub had nowhere to send them",
                 (unsigned long)dropped);
    }
}

static void print_net(void)
{
    crynet_net_status_t s;
    crynet_net_status(&s);
    ESP_LOGI(TAG, "wifi \"%s\": %s, channel %u%s",
             s.ssid[0] ? s.ssid : "(none set)",
             s.wifi_up ? "connected" : "not connected",
             (unsigned)s.channel,
             s.channel_ok ? "" : "  <-- NOT the cots' channel");
    if (s.ssid[0] && !s.channel_ok && s.channel != 6) {
        ESP_LOGW(TAG, "the cots talk on channel 6.  Set the router's "
                      "2.4 GHz channel to 6, or the wrist goes silent "
                      "while the dashboard looks healthy.");
    }
    ESP_LOGI(TAG, "dashboard: %s", s.url[0] ? s.url : "(none set)");
    ESP_LOGI(TAG, "sent this boot: %lu record(s) in %lu batch(es), "
                  "%lu failure(s) in a row",
             (unsigned long)s.uploaded, (unsigned long)s.batches,
             (unsigned long)s.failures);
    if (s.last_error[0]) {
        ESP_LOGW(TAG, "last problem: %s", s.last_error);
    }
}

/* Everything after the command word, with the spaces before it gone. */
static const char *arg_after(const char *line, size_t word)
{
    const char *p = line + word;
    while (*p == ' ') {
        p++;
    }
    return p;
}

/* The net commands.  They only write configuration; the uploader picks
 * the change up on its next turn, so nothing here touches the radio
```

```
 * from the console's task. */
static void run_net(const char *line)
{
    const char *arg = arg_after(line, 3);
    if (*arg == '\0' || strncmp(arg, "status", 6) == 0) {
        print_net();
        return;
    }
    if (strncmp(arg, "set ", 4) == 0) {
        const char *url = arg_after(arg, 3);
        if (*url == '\0') {
            ESP_LOGW(TAG, "net set needs an address: "
                          "net set http://192.168.1.50:8787/api/v1/events");
            return;
        }
        if (crynet_net_set_url(url)) {
            ESP_LOGI(TAG, "the dashboard is now %s", url);
        }
        return;
    }
    if (strncmp(arg, "ssid ", 5) == 0) {
        const char *p = arg_after(arg, 4);
        const char *space = strchr(p, ' ');
        if (space == NULL) {
            ESP_LOGW(TAG, "net ssid needs both: net ssid Amazyn me_mine_me");
            return;
        }
        char ssid[CRYNET_NET_SSID_MAX];
        const size_t n = (size_t)(space - p);
        if (n == 0 || n >= sizeof ssid) {
            ESP_LOGW(TAG, "a Wi-Fi name is 1 to 32 characters");
            return;
        }
        memcpy(ssid, p, n);
        ssid[n] = '\0';
        const char *pass = arg_after(space, 0);
        if (crynet_net_set_wifi(ssid, pass)) {
            ESP_LOGI(TAG, "the hub will join \"%s\"", ssid);
            ESP_LOGI(TAG, "remember: the router's 2.4 GHz channel must be 6");
        }
        return;
    }
    if (strncmp(arg, "name ", 5) == 0) {
        const char *name = arg_after(arg, 4);
        if (*name && crynet_net_set_name(name)) {
            ESP_LOGI(TAG, "the dashboard will call this hub \"%s\"", name);
        }
        return;
    }
    if (strncmp(arg, "token ", 6) == 0) {
        const char *tok = arg_after(arg, 5);
        if (crynet_net_set_token(tok)) {
            ESP_LOGI(TAG, "the token is stored (%s)",
                     *tok ? "set" : "cleared");
        }
        return;
    }
    ESP_LOGW(TAG, "net understands: status, set <url>, ssid <s> <p>, "
                  "name <name>, token <t>");
}

/* The status command: what the hub knows about each cot. */
static void print_status(void)
{
    link_stage_cot_t cots[CONSOLE_MAX_COTS];
    const int n = link_stage_cots(cots, CONSOLE_MAX_COTS);
    if (n == 0) {
        ESP_LOGI(TAG, "no cot has been heard yet");
        return;
    }
    for (int i = 0; i < n; i++) {
        const link_stage_cot_t *c = &cots[i];
        /* The signal strength is reported with the cot's other numbers,
         * because it is the one reading that says whether the cot is
         * merely quiet or out of reach.  Zero means no reading yet. */
        char link[24];
        if (c->rssi != 0) {
            snprintf(link, sizeof link, "  link %d dBm", (int)c->rssi);
        } else {
            snprintf(link, sizeof link, "  link --");
        }
        ESP_LOGI(TAG, "  cot %u: rms %.4f  level %.4f  window %.4f%s  "
                      "confidence %u%%%s",
                 (unsigned)c->node_id, (double)c->rms, (double)c->level,
                 (double)c->frozen_rms,
```

```
                c->window_full ? " (whole)" : " (filling)",
                (unsigned)c->confidence, link);
    }
}

/* Run one line.  The line has already had its newline removed. */
static void run_line(const char *line)
{
    ESP_LOGI(TAG, "<< %s", line);

    if (strcmp(line, "help") == 0 || strcmp(line, "?") == 0) {
        print_help();
        return;
    }
    if (strcmp(line, "status") == 0) {
        print_status();
        return;
    }
    if (strcmp(line, "clear") == 0) {
        link_stage_test_clear();
        return;
    }
    if (strncmp(line, "history", 7) == 0) {
        const char *arg = arg_after(line, 7);
        print_history(*arg ? atoi(arg) : 10);
        return;
    }
    /* The whole word, or the word followed by a space: a bare prefix
     * match would swallow "logtest" here and print the log instead of
     * running the self-test. */
    if (strcmp(line, "log") == 0 || strncmp(line, "log ", 4) == 0) {
        const char *arg = arg_after(line, 3);
        if (strcmp(arg, "clear") == 0) {
            if (crynet_log_clear()) {
                ESP_LOGW(TAG, "the log has been thrown away; the record "
                              "numbers carry on from here");
            }
            return;
        }
        print_log();
        return;
    }
    if (strncmp(line, "net", 3) == 0) {
        run_net(line);
        return;
    }
    if (strcmp(line, "upload") == 0) {
        crynet_net_upload_now();
        ESP_LOGI(TAG, "the uploader has been asked to try now; "
                      "watch for its next line, or type net");
        return;
    }
    if (strcmp(line, "logtest") == 0) {
        /* The proof the ordering rule holds on this chip.  It touches
         * only its own scratch file; the real log is untouched. */
        if (crynet_log_selftest()) {
            ESP_LOGW(TAG, "logtest: PASS");
        } else {
            ESP_LOGE(TAG, "logtest: FAIL -- read the line above: either a "
                          "record was lost or claimed before it was sent, "
                          "or the log could not be opened at all");
        }
        return;
    }
    if (strncmp(line, "alert", 5) == 0) {
        const char *arg = line + 5;
        while (*arg == ' ') {
            arg++;
        }
        if (*arg == '\0') {
            ESP_LOGW(TAG, "alert needs a cot number: alert 1");
            return;
        }
        const int cot = atoi(arg);
        if (cot < 1 || cot > 8) {
            ESP_LOGW(TAG, "a cot is numbered 1 to 8, not %d", cot);
            return;
        }
        link_stage_test_alert((uint8_t)cot);
        return;
    }
    ESP_LOGW(TAG, "unknown command: %s", line);
}

/* The console task. */
```

```
static void console_task(void *arg)
{
    (void)arg;
    char line[CONSOLE_LINE_MAX];
    int len = 0;

    ESP_LOGI(TAG, "the console is listening: help, status, alert <cot>, "
                  "clear, history, log, net, upload, logtest");

    for (;;) {
        uint8_t c;
        /* A short wait, so the task does not spin on a quiet line. */
        const int n = uart_read_bytes(UART_NUM_0, &c, 1, pdMS_TO_TICKS(50));
        if (n != 1) {
            continue;
        }
        if (c == '\r' || c == '\n') {
            if (len > 0) {
                line[len] = '\0';
                run_line(line);
                len = 0;
            }
            continue;
        }
        if (c == 8 || c == 127) {   /* backspace */
            if (len > 0) {
                len--;
            }
            continue;
        }
        if (len < CONSOLE_LINE_MAX - 1) {
            line[len++] = (char)c;
        }
    }
}

void console_start(void)
{
    /* The UART driver has to be installed before anything can be read
     * from the console UART.  The ESP-IDF log writes to that UART
     * without a driver -- it puts the bytes in the hardware's own
     * register -- and the driver, with the receive buffer a command
     * arrives in, is installed only by the console's REPL, which this
     * firmware does not start.
     *
     * Installing it does not disturb the log: the log's own write path
     * keeps using the hardware register, because the VFS only switches
     * to the driver when the REPL takes the port over. */
    const esp_err_t err = uart_driver_install(UART_NUM_0, 1024, 0, 0, NULL, 0);
    if (err != ESP_OK && err != ESP_ERR_INVALID_STATE) {
        ESP_LOGE(TAG, "the console UART could not be set up: %s",
                 esp_err_to_name(err));
        return;
    }
    /* The history command holds forty records on its stack, so the
     * console needs more than the bare line reader did. */
    xTaskCreate(console_task, "console", 5120, NULL, 3, NULL);
}
```

Hub - the signal contract and the front end

firmware/hub_infer/components/crynet_dsp/include/crynet_dsp.h

86 lines · 3,332 bytes

```
/*
 * CryNet — DSP front end (hub firmware).
 *
 * Turns a 2 s window of int16 PCM into the 81 x 64 standardised log-Mel
 * matrix the trained model expects, then quantises it to int8.
 *
 * The maths is identical to the Python training pipeline (verified on the
 * host to 6.9e-05 max absolute deviation) and to the notebook-generated
 * reference implementation in research/out/dsp_crynet.c. This component
 * differs from that reference in three ways that matter on the target:
 *
 *   - no 130 KB file-scope static buffers and no padded copy of the
 *     signal; the working memory is 5.4 KB, allocated once in internal
 *     SRAM, so it costs the hub's 512 KB of internal SRAM almost nothing;
```

```
 *   - the working set is owned by a context, so the inference task and the
 *     radio task cannot corrupt each other's state;
 *   - the decision rule (silence gate + calibrated threshold) lives next to
 *     the features it depends on.
 *
 * This header deliberately does NOT include the generated crynet_tables.h:
 * that header defines file-scope arrays, and in C a file-scope `const`
 * has external linkage, so it may only be included by one translation
 * unit in the whole firmware. crynet_dsp.c is that translation unit.
 */

#ifndef CRYNET_DSP_H
#define CRYNET_DSP_H

#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

/* Feature geometry — fixed by the trained weights. */
#define CRYNET_N_SAMPLES 32000
#define CRYNET_N_FRAMES  81
#define CRYNET_N_MELS    64
#define CRYNET_N_FFT     512
#define CRYNET_HOP       400
#define CRYNET_CENTER_PAD 256
#define CRYNET_TAIL_PAD  512

#define CRYNET_FEATURE_COUNT (CRYNET_N_FRAMES * CRYNET_N_MELS) /* 5184 */

/* Working memory for the front end. Opaque; call crynet_dsp_init first. */
typedef struct {
    void  *base;       /* the single allocation the rest point into   */
    float *power;      /* 257 power-spectrum bins                     */
    float *mel;        /* 64 mel bands                                */
    void  *fft;        /* 512 complex bins                            */
    bool   from_psram; /* true if the buffers live in PSRAM           */
    size_t bytes;      /* total bytes allocated                       */
} crynet_dsp_ctx_t;

/*
 * Allocate the working buffers. Prefers internal SRAM, because the FFT
 * and the Mel filterbank stream through them 81 times per window and
 * PSRAM is uncached external memory on the ESP32-S3. Falls back to PSRAM
 * if internal SRAM is unavailable. Returns false if allocation failed.
 */
bool crynet_dsp_init(crynet_dsp_ctx_t *ctx);

void crynet_dsp_free(crynet_dsp_ctx_t *ctx);

/*
 * 2 s window of int16 mono PCM @ 16 kHz -> CRYNET_FEATURE_COUNT floats of
 * standardised log-Mel, laid out frame-major: mel[frame * 64 + band].
 */
void crynet_dsp_features(const crynet_dsp_ctx_t *ctx,
                         const int16_t *pcm,
                         float *mel_out);

/* Standardised log-Mel -> int8 in the model's input quantisation. */
void crynet_dsp_quantise(const float *mel, int8_t *q);

/* RMS of a window after DC removal, in full scale units (0..1). */
float crynet_dsp_rms(const int16_t *pcm);

#ifdef __cplusplus
}
#endif

#endif /* CRYNET_DSP_H */
```

firmware/hub_infer/components/crynet_dsp/crynet_dsp.c

285 lines · 10,594 bytes

```
/*
 * CryNet — DSP front end implementation.
 *
 * This is the ONLY translation unit in the firmware that includes the
 * notebook-generated crynet_tables.h (file-scope `const` arrays have
```

```
 * external linkage in C, so including them twice is a link error).
 *
 * The signal chain, frozen and identical to the Python training pipeline:
 *
 *   int16 pcm[32000]                    2.000 s @ 16 kHz mono
 *     -> subtract mean                  DC removal (INMP441 sits at ~+18000)
 *     -> zero-pad 256 front / 512 rear
 *     -> 81 frames, 512-pt symmetric Hann, hop 400
 *     -> |rfft|^2                       257 bins
 *     -> 64 Mel bands (librosa slaney, 20 Hz - 8 kHz)
 *     -> 10*log10(max(P, 1e-10))        floored at -80 dB
 *     -> per-window standardisation     (mean 0, std 1)
 *     -> int8 quantise                  q = round(v/scale) + zero_point
 */

#include <math.h>
#include <string.h>

#include "esp_heap_caps.h"

#include "crynet_tables.h"  /* generated — see the note above */
#include "crynet_dsp.h"
#include "crynet_model.h"

/* Compile-time drift detection: the firmware mirrors a few of the
 * generated constants under its own names, because the generated header
 * may only be included once. If a new set of Kaggle artifacts changes a
 * value, this translation unit fails to compile instead of silently
 * classifying with stale numbers. */
_Static_assert(CRYNET_INPUT_SCALE == CRYNET_IN_SCALE, "input scale drift");
_Static_assert(CRYNET_INPUT_ZERO_POINT == CRYNET_IN_ZP, "input zero-point drift");
_Static_assert(CRYNET_OUTPUT_SCALE == CRYNET_OUT_SCALE, "output scale drift");
_Static_assert(CRYNET_OUTPUT_ZERO_POINT == CRYNET_OUT_ZP, "output zero-point drift");
_Static_assert(CRYNET_SILENCE_RMS_GATE == CRYNET_SILENCE_RMS, "silence gate drift");
_Static_assert(CRYNET_N_SAMPLES == 32000, "sample count drift");
_Static_assert(CRYNET_N_FRAMES == 81, "frame count drift");
_Static_assert(CRYNET_N_MELS == 64, "mel band count drift");

typedef struct {
    float re;
    float im;
} cpx_t;

/* ---------------------------------------------------------------- */
/* FFT: iterative radix-2, N = 512, in place.                       */
/* The twiddle recurrence is kept identical to the reference        */
/* implementation so the two agree bit-for-bit on the same input.   */
/* ---------------------------------------------------------------- */
static void fft512(cpx_t *a)
{
    unsigned i, j, len;

    for (i = 1, j = 0; i < CRYNET_N_FFT; i++) {
        unsigned bit = CRYNET_N_FFT >> 1;
        for (; j & bit; bit >>= 1) {
            j ^= bit;
        }
        j ^= bit;
        if (i < j) {
            cpx_t t = a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }

    for (len = 2; len <= CRYNET_N_FFT; len <<= 1) {
        const float ang = -2.0f * 3.14159265358979f / (float)len;
        const float wr = cosf(ang);
        const float wi = sinf(ang);

        for (i = 0; i < CRYNET_N_FFT; i += len) {
            float cr = 1.0f, ci = 0.0f;
            for (j = 0; j < len / 2; j++) {
                const cpx_t u = a[i + j];
                cpx_t v;
                v.re = a[i + j + len / 2].re * cr - a[i + j + len / 2].im * ci;
                v.im = a[i + j + len / 2].re * ci + a[i + j + len / 2].im * cr;
                a[i + j].re = u.re + v.re;
                a[i + j].im = u.im + v.im;
                a[i + j + len / 2].re = u.re - v.re;
                a[i + j + len / 2].im = u.im - v.im;

                const float nr = cr * wr - ci * wi;
                ci = cr * wi + ci * wr;
                cr = nr;
```

```
            }
        }
    }
}

/* ---------------------------------------------------------------- */

bool crynet_dsp_init(crynet_dsp_ctx_t *ctx)
{
    if (ctx == NULL) {
        return false;
    }
    memset(ctx, 0, sizeof(*ctx));

    /*
     * The working set is only the spectrum scratch now: the padded copy of
     * the signal is not materialised any more (each frame is built
     * straight from the caller's int16 PCM), so what is left is 5.4 KB.
     *
     * It lives in internal SRAM rather than PSRAM on purpose. PSRAM on the
     * ESP32-S3 is external memory and is not cached by default, so every
     * access is a SPI round trip; the FFT's butterfly passes and the Mel
     * filterbank's bin loop both stream through these buffers 81 times per
     * window, and leaving them in PSRAM made the front end cost 135 ms
     * instead of a few milliseconds. Internal SRAM is plentiful once the
     * 128 KB padded-signal buffer is gone.
     */
    const size_t total = (CRYNET_N_FFT / 2 + 1) * sizeof(float) /* power */
                         + CRYNET_N_MELS * sizeof(float)         /* mel    */
                         + CRYNET_N_FFT * sizeof(cpx_t);         /* fft    */

    uint8_t *mem = heap_caps_malloc(total, MALLOC_CAP_INTERNAL | MALLOC_CAP_8BIT);
    ctx->from_psram = false;
    if (mem == NULL) {
        mem = heap_caps_malloc(total, MALLOC_CAP_8BIT);
        ctx->from_psram = true;
    }
    if (mem == NULL) {
        return false;
    }

    ctx->bytes = total;
    ctx->base = mem;
    ctx->power = (float *)mem;
    ctx->mel = ctx->power + (CRYNET_N_FFT / 2 + 1);
    ctx->fft = ctx->mel + CRYNET_N_MELS;
    return true;
}

void crynet_dsp_free(crynet_dsp_ctx_t *ctx)
{
    if (ctx == NULL || ctx->base == NULL) {
        return;
    }
    if (ctx->from_psram) {
        heap_caps_free(ctx->base);
    } else {
        free(ctx->base);
    }
    memset(ctx, 0, sizeof(*ctx));
}

void crynet_dsp_features(const crynet_dsp_ctx_t *ctx,
                         const int16_t *pcm,
                         float *mel_out)
{
    cpx_t *spec = (cpx_t *)ctx->fft;
    float *power = ctx->power;
    float *mel = ctx->mel;

    /* 1. DC removal + normalise to [-1, 1]. The INMP441 sits at roughly
     *    +18000 counts, so this step is not optional.
     *
     *    The sum is accumulated in an integer on purpose. A float32
     *    accumulator loses the low bits badly at this magnitude: summing
     *    32,000 samples of 18000 in float32 gives 575,726,660 instead of
     *    576,000,000, so the "mean" comes out 8.5 counts low and 8.5
     *    counts of DC survive. That is invisible in a normal signal but
     *    fatal for the DC-offset probe, where it leaves a constant
     *    residual that the per-window standardisation then blows up into a
     *    completely different feature vector. int32 holds the sum of
     *    32,000 int16 samples exactly (at most 1,048,576,000), so the
     *    mean is the correctly rounded value the host reference computes
     *    — and integer adds are far cheaper than the double-precision
     *    ones the ESP32-S3 has to emulate in software. */
```

```
    int32_t sum = 0;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) {
        sum += (int32_t)pcm[i];
    }
    const float mean = (float)((double)sum / (double)CRYNET_N_SAMPLES);

    /* 2. frames -> FFT -> power -> mel -> dB
     *
     *    Each frame is built directly from the caller's PCM rather than
     *    from a padded copy of the whole signal: the copy was 128 KB of
     *    the working set and bought nothing. Frame f covers padded
     *    samples [f * HOP, f * HOP + N_FFT), and padded sample p holds
     *    PCM sample p - CENTER_PAD, with zeros outside the signal. */
    for (int fr = 0; fr < CRYNET_N_FRAMES; fr++) {
        const int start = fr * CRYNET_HOP - CRYNET_CENTER_PAD;

        for (int i = 0; i < CRYNET_N_FFT; i++) {
            const int s = start + i;
            float v = 0.0f;
            if (s >= 0 && s < CRYNET_N_SAMPLES) {
                v = ((float)pcm[s] - mean) / 32768.0f;
            }
            spec[i].re = v * CRYNET_HANN[i];
            spec[i].im = 0.0f;
        }
        fft512(spec);

        for (int i = 0; i <= CRYNET_N_FFT / 2; i++) {
            power[i] = spec[i].re * spec[i].re + spec[i].im * spec[i].im;
        }

        for (int m = 0; m < CRYNET_N_MELS; m++) {
            float acc = 0.0f;
            const int start = CRYNET_MEL_START[m];
            const int end = CRYNET_MEL_END[m];
            const float *wts = &CRYNET_MEL_WEIGHTS[CRYNET_MEL_OFFSET[m]];
            for (int b = start; b < end; b++) {
                acc += power[b] * wts[b - start];
            }
            mel[m] = acc;
        }

        for (int m = 0; m < CRYNET_N_MELS; m++) {
            float p = mel[m];
            if (p < CRYNET_LOG_EPS) {
                p = CRYNET_LOG_EPS;
            }
            float db = 10.0f * log10f(p);
            if (db < CRYNET_DB_FLOOR) {
                db = CRYNET_DB_FLOOR;
            }
            mel_out[fr * CRYNET_N_MELS + m] = db;
        }
    }

    /* 3. per-window standardisation — this is what makes the model
     *    level-invariant. Without it the network can learn "quiet = cry",
     *    because the ICD cries sit at RMS 0.006 and its negatives at
     *    0.3-0.46. */
    const int n = CRYNET_N_FRAMES * CRYNET_N_MELS;
    float mu = 0.0f;
    for (int i = 0; i < n; i++) {
        mu += mel_out[i];
    }
    mu /= (float)n;

    float var = 0.0f;
    for (int i = 0; i < n; i++) {
        const float d = mel_out[i] - mu;
        var += d * d;
    }
    const float sd = sqrtf(var / (float)n) + 1e-6f;

    for (int i = 0; i < n; i++) {
        mel_out[i] = (mel_out[i] - mu) / sd;
    }
}

void crynet_dsp_quantise(const float *mel, int8_t *q)
{
    const int n = CRYNET_N_FRAMES * CRYNET_N_MELS;
    for (int i = 0; i < n; i++) {
        float v = nearbyintf(mel[i] / CRYNET_IN_SCALE) + (float)CRYNET_IN_ZP;
        if (v > 127.0f) {
            v = 127.0f;
```

```
        }
        if (v < -128.0f) {
            v = -128.0f;
        }
        q[i] = (int8_t)v;
    }
}

float crynet_dsp_rms(const int16_t *pcm)
{
    /* Same double-precision accumulation as the front end — see the note
     * in crynet_dsp_features(). */
    int32_t sum = 0;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) {
        sum += (int32_t)pcm[i];
    }
    const float mean = (float)((double)sum / (double)CRYNET_N_SAMPLES);

    float acc = 0.0f;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) {
        const float d = ((float)pcm[i] - mean) / 32768.0f;
        acc += d * d;
    }
    return sqrtf(acc / (float)CRYNET_N_SAMPLES);
}
```

firmware/hub_infer/components/crynet_dsp/reference/dsp_crynet.h

15 lines · 366 bytes

```
#ifndef DSP_CRYNET_H
#define DSP_CRYNET_H
#include <stdint.h>
#include "crynet_tables.h"
#ifdef __cplusplus
extern "C" {
#endif
void crynet_log_mel(const int16_t *pcm, float *mel_out);
void crynet_quantise(const float *mel, int8_t *q);
float crynet_dequantise(int8_t v);
float crynet_window_rms(const int16_t *pcm);
#ifdef __cplusplus
}
#endif
#endif
```

firmware/hub_infer/components/crynet_dsp/reference/dsp_crynet.c

157 lines · 6,225 bytes

```

/* ============================================================================
 * dsp_crynet.c — CryNet log-Mel front end, numerically identical to the
 *                Python implementation used for training.
 *
 *  Host parity test:
 *      gcc -O2 -DCRYNET_SELFTEST -o dsp_test dsp_crynet.c -lm
 *      ./dsp_test probe.s16le
 *
 *  On the ESP32-S3 replace fft512() with ESP-DSP:
 *      dsps_fft2r() + dsps_cmplx_mag()  (roughly 10x faster than the
 *      reference FFT below, which is kept so the file builds anywhere).
 * ============================================================================*/
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#include "crynet_tables.h"

typedef struct { float re, im; } cpx_t;

/* ---- iterative radix-2 FFT, N = 512, in place ---------------------------- */
static void fft512(cpx_t *a) {
    unsigned i, j, len;
    cpx_t t;
    for (i = 1, j = 0; i < 512; i++) {
        unsigned bit = 512 >> 1;
```

```
        for (; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if (i < j) { t = a[i]; a[i] = a[j]; a[j] = t; }
    }
    for (len = 2; len <= 512; len <<= 1) {
        float ang = -2.0f * 3.14159265358979f / (float)len;
        float wr = cosf(ang), wi = sinf(ang);
        for (i = 0; i < 512; i += len) {
            float cr = 1.0f, ci = 0.0f;
            for (j = 0; j < len / 2; j++) {
                cpx_t u = a[i + j];
                cpx_t v;
                v.re = a[i + j + len / 2].re * cr - a[i + j + len / 2].im * ci;
                v.im = a[i + j + len / 2].re * ci + a[i + j + len / 2].im * cr;
                a[i + j].re = u.re + v.re;  a[i + j].im = u.im + v.im;
                a[i + j + len / 2].re = u.re - v.re;
                a[i + j + len / 2].im = u.im - v.im;
                float nr = cr * wr - ci * wi;
                ci = cr * wi + ci * wr;
                cr = nr;
            }
        }
    }
}

/* ---- log-Mel front end ------------------------------------------------- */
/*  pcm     : CRYNET_N_SAMPLES int16 mono samples @ 16 kHz
 *  mel_out : CRYNET_N_FRAMES * CRYNET_N_MELS floats (standardised dB)     */
void crynet_log_mel(const int16_t *pcm, float *mel_out) {
    static float x[CRYNET_N_SAMPLES + CRYNET_CENTER_PAD + CRYNET_TAIL_PAD];
    static cpx_t spec[CRYNET_N_FFT];
    static float power[CRYNET_N_FFT / 2 + 1];
    static float mel[CRYNET_N_MELS];

    /* 1. DC removal + normalise to [-1, 1] */
    float mean = 0.0f;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) mean += (float)pcm[i];
    mean /= (float)CRYNET_N_SAMPLES;

    memset(x, 0, sizeof(float) * (CRYNET_N_SAMPLES + CRYNET_CENTER_PAD + CRYNET_TAIL_PAD));
    for (int i = 0; i < CRYNET_N_SAMPLES; i++)
        x[CRYNET_CENTER_PAD + i] = ((float)pcm[i] - mean) / 32768.0f;

    /* 2. frames -> FFT -> power -> mel -> dB */
    for (int fr = 0; fr < CRYNET_N_FRAMES; fr++) {
        const float *w = x + fr * CRYNET_HOP;
        for (int i = 0; i < CRYNET_N_FFT; i++) {
            spec[i].re = w[i] * CRYNET_HANN[i];
            spec[i].im = 0.0f;
        }
        fft512(spec);
        for (int i = 0; i <= CRYNET_N_FFT / 2; i++)
            power[i] = spec[i].re * spec[i].re + spec[i].im * spec[i].im;
        for (int m = 0; m < CRYNET_N_MELS; m++) {
            float acc = 0.0f;
            int start = CRYNET_MEL_START[m], end = CRYNET_MEL_END[m];
            const float *wts = &CRYNET_MEL_WEIGHTS[CRYNET_MEL_OFFSET[m]];
            for (int b = start; b < end; b++) acc += power[b] * wts[b - start];
            mel[m] = acc;
        }
        for (int m = 0; m < CRYNET_N_MELS; m++) {
            float p = mel[m];
            if (p < CRYNET_LOG_EPS) p = CRYNET_LOG_EPS;
            float db = 10.0f * log10f(p);
            if (db < CRYNET_DB_FLOOR) db = CRYNET_DB_FLOOR;
            mel_out[fr * CRYNET_N_MELS + m] = db;
        }
    }

#if CRYNET_CMVN
    /* 3. per-window standardisation */
    float mu = 0.0f;
    const int n = CRYNET_N_FRAMES * CRYNET_N_MELS;
    for (int i = 0; i < n; i++) mu += mel_out[i];
    mu /= (float)n;
    float var = 0.0f;
    for (int i = 0; i < n; i++) { float d = mel_out[i] - mu; var += d * d; }
    float sd = sqrtf(var / (float)n) + 1e-6f;
    for (int i = 0; i < n; i++) mel_out[i] = (mel_out[i] - mu) / sd;
#endif
}

/* ---- quantisation helpers ---------------------------------------------- */
void crynet_quantise(const float *mel, int8_t *q) {
    const int n = CRYNET_N_FRAMES * CRYNET_N_MELS;
```

```
    for (int i = 0; i < n; i++) {
        float v = nearbyintf(mel[i] / CRYNET_IN_SCALE) + (float)CRYNET_IN_ZP;
        if (v > 127.0f) v = 127.0f;
        if (v < -128.0f) v = -128.0f;
        q[i] = (int8_t)v;
    }
}

float crynet_dequantise(int8_t v) {
    return ((float)v - (float)CRYNET_OUT_ZP) * CRYNET_OUT_SCALE;
}

/* ---- silence gate ------------------------------------------------------- */
float crynet_window_rms(const int16_t *pcm) {
    float mean = 0.0f;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) mean += (float)pcm[i];
    mean /= (float)CRYNET_N_SAMPLES;
    float acc = 0.0f;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) {
        float d = ((float)pcm[i] - mean) / 32768.0f;
        acc += d * d;
    }
    return sqrtf(acc / (float)CRYNET_N_SAMPLES);
}

/* ---- host parity test --------------------------------------------------- */
#ifdef CRYNET_SELFTEST
int main(int argc, char **argv) {
    if (argc < 2) { fprintf(stderr, "usage: %s raw_pcm_s16le.bin\n", argv[0]); return 2; }
    FILE *f = fopen(argv[1], "rb");
    if (!f) { perror("fopen"); return 2; }
    static int16_t pcm[CRYNET_N_SAMPLES];
    size_t got = fread(pcm, sizeof(int16_t), CRYNET_N_SAMPLES, f);
    fclose(f);
    if (got != CRYNET_N_SAMPLES) { fprintf(stderr, "short read\n"); return 2; }
    static float mel[CRYNET_N_FRAMES * CRYNET_N_MELS];
    crynet_log_mel(pcm, mel);
    FILE *o = fopen("mel_c.bin", "wb");
    fwrite(mel, sizeof(float), CRYNET_N_FRAMES * CRYNET_N_MELS, o);
    fclose(o);
    fprintf(stderr, "wrote mel_c.bin  rms=%.6f\n", crynet_window_rms(pcm));
    return 0;
}
#endif
```

Hub - inference

firmware/hub_infer/components/crynet_infer/include/crynet_infer.h

61 lines · 1,940 bytes

```
/*
 * CryNet — TFLite Micro inference component.
 *
 * A thin C API over the TFLite Micro interpreter. The interpreter itself
 * is C++, so it is hidden behind an opaque handle and this header is
 * usable from plain C.
 *
 * The model is the INT8 graph produced by the training notebook
 * (crynet_model_bytes()), so the weights, the input quantisation and the
 * output quantisation all come from the same artifact — the firmware
 * never carries its own copy of the calibration numbers.
 */

#ifndef CRYNET_INFER_H
#define CRYNET_INFER_H

#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

typedef struct crynet_infer crynet_infer_t;

/*
 * Create the interpreter. Allocates the tensor arena (PSRAM first) and
```

```
 * allocates the tensors. Returns NULL only if the model itself is
 * unusable; a returned handle with ready == false carries the reason in
 * crynet_infer_error().
 */
crynet_infer_t *crynet_infer_create(void);

void crynet_infer_destroy(crynet_infer_t *inf);

/*
 * Run one 2 s window. `features` is CRYNET_FEATURE_COUNT int8 values in
 * the model's input quantisation. p_no_cry / p_cry receive the
 * dequantised class probabilities; either may be NULL.
 */
bool crynet_infer_run(crynet_infer_t *inf,
                      const int8_t *features,
                      float *p_no_cry,
                      float *p_cry);

/* Bytes of tensor arena reserved, and how much of it the model used. */
size_t crynet_infer_arena_bytes(const crynet_infer_t *inf);
size_t crynet_infer_arena_used(const crynet_infer_t *inf);

/* Last error text ("" when there was none). */
const char *crynet_infer_error(const crynet_infer_t *inf);

/* True once the interpreter has allocated its tensors successfully. */
bool crynet_infer_ready(const crynet_infer_t *inf);

#ifdef __cplusplus
}
#endif

#endif /* CRYNET_INFER_H */
```

firmware/hub_infer/components/crynet_infer/crynet_infer.cpp

247 lines · 8,729 bytes

```
/*
 * CryNet — TFLite Micro inference implementation.
 *
 * Notes on three things that are easy to get wrong on the target:
 *
 *  1. The model byte array generated by the notebook has alignment 1, but
 *     a FlatBuffer must be aligned to its scalar width before it is read.
 *     Rather than change the generated header, the model is copied once
 *     at boot into a 16-byte aligned PSRAM buffer.
 *
 *  2. The tensor arena is the largest single allocation in the firmware.
 *     It goes to PSRAM when the board has it; the hub's 512 KB of
 *     internal SRAM is worth more to the radio, TFT and BLE stacks.
 *
 *  3. The output probabilities are dequantised with the *tensor's* own
 *     quantisation parameters, not with constants copied into the
 *     firmware, so the reported probability is always the one the model
 *     actually produced.
 */

#include "crynet_infer.h"

#include <new>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#include "esp_heap_caps.h"

#include "crynet_model.h"

#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/schema/schema_generated.h"

namespace {

/*
 * The notebook's hint is 120 KB. The arena is allocated with headroom
 * because the hint is an estimate of the *persistent* tensors only; the
 * allocator also needs scratch space for activations. If the model does
 * not fit, the failure is loud and measured rather than a silent
 * allocation error later.
 */
```

```
constexpr size_t kArenaBytes = 256 * 1024;

/* Number of ops the resolver can hold: conv, depthwise conv, max pool,
 * mean, fully connected, softmax. The selected conv_cnn model uses five
 * of them; depthwise conv is registered so the sep_cnn variant can be
 * dropped in without touching this file. */
constexpr size_t kResolverOps = 6;

}  // namespace

struct crynet_infer {
    void *interp_mem = nullptr;
    void *resolver_mem = nullptr;
    void *model_raw = nullptr;   /* unaligned base of the model allocation */
    uint8_t *arena = nullptr;
    size_t arena_bytes = 0;
    size_t arena_used = 0;
    bool ready = false;
    char error[192] = {0};

    tflite::MicroInterpreter *interp = nullptr;
    tflite::MicroMutableOpResolver<kResolverOps> *resolver = nullptr;
};

extern "C" {

crynet_infer_t *crynet_infer_create(void)
{
    auto *self = static_cast<crynet_infer_t *>(
        heap_caps_calloc(1, sizeof(crynet_infer_t),
                         MALLOC_CAP_8BIT | MALLOC_CAP_INTERNAL));
    if (self == nullptr) {
        return nullptr;
    }

    /* --- model, copied into an aligned buffer ------------------------ */
    const size_t model_len = crynet_model_size();
    /* The FlatBuffer verifier wants the model buffer aligned to 16 bytes.
     * heap_caps_malloc only guarantees 4-byte alignment, so over-allocate
     * by 16 bytes and align the copy by hand. */
    const size_t model_alloc = model_len + 16;
    void *model_raw = heap_caps_malloc(model_alloc, MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT);
    if (model_raw == nullptr) {
        model_raw = heap_caps_malloc(model_alloc, MALLOC_CAP_8BIT);
    }
    if (model_raw == nullptr) {
        snprintf(self->error, sizeof(self->error),
                 "cannot allocate %u bytes for the model", (unsigned)model_len);
        return self;
    }
    self->model_raw = model_raw;
    uint8_t *model_mem = reinterpret_cast<uint8_t *>(
        (reinterpret_cast<uintptr_t>(model_raw) + 15u) &
        ~static_cast<uintptr_t>(15u));
    memcpy(model_mem, crynet_model_bytes(), model_len);

    const tflite::Model *model = tflite::GetModel(model_mem);
    if (model->version() != TFLITE_SCHEMA_VERSION) {
        snprintf(self->error, sizeof(self->error),
                 "model schema %d, expected %d", (int)model->version(),
                 TFLITE_SCHEMA_VERSION);
        return self;
    }

    /* --- tensor arena ------------------------------------------------ */
    self->arena = static_cast<uint8_t *>(
        heap_caps_malloc(kArenaBytes, MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT));
    if (self->arena == nullptr) {
        self->arena = static_cast<uint8_t *>(
            heap_caps_malloc(kArenaBytes, MALLOC_CAP_8BIT));
    }
    if (self->arena == nullptr) {
        snprintf(self->error, sizeof(self->error),
                 "cannot allocate a %u byte tensor arena", (unsigned)kArenaBytes);
        return self;
    }
    self->arena_bytes = kArenaBytes;

    /* --- interpreter ------------------------------------------------- */
    self->resolver_mem = heap_caps_calloc(1, sizeof(*self->resolver),
                                          MALLOC_CAP_8BIT | MALLOC_CAP_INTERNAL);
    self->interp_mem = heap_caps_calloc(1, sizeof(*self->interp),
                                        MALLOC_CAP_8BIT | MALLOC_CAP_INTERNAL);
    if (self->resolver_mem == nullptr || self->interp_mem == nullptr) {
        snprintf(self->error, sizeof(self->error), "cannot allocate the interpreter");
```

```
        return self;
    }

    self->resolver = new (self->resolver_mem) tflite::MicroMutableOpResolver<kResolverOps>();
    self->resolver->AddConv2D();
    self->resolver->AddDepthwiseConv2D();
    self->resolver->AddMaxPool2D();
    self->resolver->AddMean();
    self->resolver->AddFullyConnected();
    self->resolver->AddSoftmax();

    self->interp = new (self->interp_mem) tflite::MicroInterpreter(
        model, *self->resolver, self->arena, self->arena_bytes);

    if (self->interp->AllocateTensors() != kTfLiteOk) {
        snprintf(self->error, sizeof(self->error),
                 "AllocateTensors failed (arena %u bytes)", (unsigned)self->arena_bytes);
        return self;
    }

    /* The interpreter reports the high-water mark of the arena. */
    self->arena_used = self->interp->arena_used_bytes();

    self->ready = true;
    return self;
}

void crynet_infer_destroy(crynet_infer_t *inf)
{
    if (inf == nullptr) {
        return;
    }
    if (inf->interp != nullptr) {
        inf->interp->~MicroInterpreter();
        heap_caps_free(inf->interp_mem);
    }
    if (inf->resolver != nullptr) {
        inf->resolver->~MicroMutableOpResolver<kResolverOps>();
        heap_caps_free(inf->resolver_mem);
    }
    if (inf->arena != nullptr) {
        heap_caps_free(inf->arena);
    }
    if (inf->model_raw != nullptr) {
        heap_caps_free(inf->model_raw);
    }
    heap_caps_free(inf);
}

bool crynet_infer_run(crynet_infer_t *inf,
                      const int8_t *features,
                      float *p_no_cry,
                      float *p_cry)
{
    if (inf == nullptr || !inf->ready || features == nullptr) {
        return false;
    }

    TfLiteTensor *input = inf->interp->input(0);
    if (input == nullptr || input->type != kTfLiteInt8) {
        snprintf(inf->error, sizeof(inf->error), "input tensor is not int8");
        return false;
    }
    if (input->bytes != CRYNET_N_FRAMES * CRYNET_N_MELS) {
        snprintf(inf->error, sizeof(inf->error),
                 "input tensor is %u bytes, expected %d",
                 (unsigned)input->bytes, CRYNET_N_FRAMES * CRYNET_N_MELS);
        return false;
    }

    memcpy(input->data.int8, features, input->bytes);

    if (inf->interp->Invoke() != kTfLiteOk) {
        snprintf(inf->error, sizeof(inf->error), "Invoke failed");
        return false;
    }

    const TfLiteTensor *output = inf->interp->output(0);
    if (output == nullptr || output->type != kTfLiteInt8 || output->bytes != 2) {
        snprintf(inf->error, sizeof(inf->error), "unexpected output tensor");
        return false;
    }

    const float scale = output->params.scale;
    const int32_t zero_point = output->params.zero_point;
```

```
    const float p0 = (static_cast<float>(output->data.int8[0]) - zero_point) * scale;
    const float p1 = (static_cast<float>(output->data.int8[1]) - zero_point) * scale;

    if (p_no_cry != nullptr) {
        *p_no_cry = p0;
    }
    if (p_cry != nullptr) {
        *p_cry = p1;
    }
    return true;
}

size_t crynet_infer_arena_bytes(const crynet_infer_t *inf)
{
    return inf != nullptr ? inf->arena_bytes : 0;
}

size_t crynet_infer_arena_used(const crynet_infer_t *inf)
{
    return inf != nullptr ? inf->arena_used : 0;
}

const char *crynet_infer_error(const crynet_infer_t *inf)
{
    return inf != nullptr ? inf->error : "no interpreter";
}

bool crynet_infer_ready(const crynet_infer_t *inf)
{
    return inf != nullptr && inf->ready;
}

}  // extern "C"
```

firmware/hub_infer/components/crynet_model/include/crynet_model.h

58 lines · 2,229 bytes

```
/*
 * CryNet — model component.
 *
 * Owns the trained INT8 TFLite model and the calibration constants that
 * the firmware needs. The model bytes and the Mel tables are generated by
 * the training notebook; this component is the ONLY translation unit that
 * includes the generated headers, so the array definitions in them are
 * never duplicated across translation units (which would be a link error
 * in C, where file-scope `const` has external linkage).
 *
 * Every other component reaches the model through the accessors below.
 */

#ifndef CRYNET_MODEL_H
#define CRYNET_MODEL_H

#include <stddef.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

/* The INT8 TFLite model, compiled into .rodata (flash). */
const uint8_t *crynet_model_bytes(void);
size_t         crynet_model_size(void);

/*
 * Calibration constants, mirrored from the generated crynet_tables.h.
 * crynet_dsp.c static-asserts each of these against the generated header
 * at compile time, and firmware/tools/check_artifact_drift.py re-checks
 * them against deploy_config.json — so the copies cannot silently drift
 * when a new set of Kaggle artifacts is dropped into research/out/.
 *
 * Values below are from the second full Kaggle run - the retrain with
 * the GTZAN music corpus in it (model SHA-256
 * bd592d541027b9f448bc218e6fdbef113e39d64b8a59f366a18586987dce43b9).
 * Only the input calibration moved: IN_SCALE 0.060125612 -> 0.116329655
 * and IN_ZP -4 -> -70; the threshold, the silence gate, the output
 * calibration and the geometry are all the same run to run.
 */
#define CRYNET_THRESHOLD          0.9500f    /* P(cry) >= this -> alert        */
#define CRYNET_SILENCE_RMS_GATE   0.001500f  /* below this the window is silent */
#define CRYNET_INPUT_SCALE        0.116329655f
```

```
#define CRYNET_INPUT_ZERO_POINT   (-70)
#define CRYNET_OUTPUT_SCALE       0.003906250f
#define CRYNET_OUTPUT_ZERO_POINT  (-128)

/* Feature geometry — fixed by the trained weights. */
#define CRYNET_N_SAMPLES          32000      /* 2.0 s @ 16 kHz */
#define CRYNET_N_FRAMES           81
#define CRYNET_N_MELS             64

#ifdef __cplusplus
}
#endif

#endif /* CRYNET_MODEL_H */
```

firmware/hub_infer/components/crynet_model/crynet_model.c

20 lines · 548 bytes

```
/*
 * CryNet — model component implementation.
 *
 * This is the single translation unit that includes the notebook-generated
 * model_data.h. Keeping the include here means the generated byte array
 * has exactly one definition in the firmware image.
 */

#include "model_data.h"   /* generated by the training notebook — do not edit */
#include "crynet_model.h"

const uint8_t *crynet_model_bytes(void)
{
    return (const uint8_t *)crynet_model;
}

size_t crynet_model_size(void)
{
    return (size_t)crynet_model_len;
}
```

firmware/hub_infer/components/crynet_model/include/crynet_tables.h

144 lines · 15,513 bytes

```
#ifndef CRYNET_TABLES_H
#define CRYNET_TABLES_H
#include <stdint.h>

/* Generated by the CryNet training notebook — do not edit by hand. */
#define CRYNET_SR            16000
#define CRYNET_N_SAMPLES     32000
#define CRYNET_N_FFT         512
#define CRYNET_HOP           400
#define CRYNET_N_MELS        64
#define CRYNET_N_FRAMES      81
#define CRYNET_CENTER_PAD    256
#define CRYNET_TAIL_PAD      512
#define CRYNET_DB_FLOOR      (-80.0f)
#define CRYNET_LOG_EPS       (0.0000000001f)
#define CRYNET_CMVN          1
#define CRYNET_IN_SCALE      0.116329655f
#define CRYNET_IN_ZP         -70
#define CRYNET_OUT_SCALE     0.003906250f
#define CRYNET_OUT_ZP        -128
#define CRYNET_THRESHOLD     0.9500f
#define CRYNET_SILENCE_RMS   0.001500f

/* Sparse Mel filterbank: band m uses FFT bins
   CRYNET_MEL_START[m] .. CRYNET_MEL_END[m]-1 with weights taken from
   CRYNET_MEL_WEIGHTS[CRYNET_MEL_OFFSET[m] + (bin - CRYNET_MEL_START[m])]. */
const int16_t CRYNET_MEL_START[CRYNET_N_MELS] = {
  1, 3, 4, 6, 7, 9, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26, 28, 29, 31, 32, 34, 35, 37, 39, 41, 43, 45, 47, 49, 51, 54,
56, 59, 62, 65, 68, 71, 75, 78, 82, 86, 90, 95, 99, 104, 109, 115, 120, 126, 132, 138, 145, 152, 160, 167, 176, 184, 193, 202, 212,
222, 233
};
const int16_t CRYNET_MEL_END[CRYNET_N_MELS] = {
  4, 6, 7, 9, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23, 25, 26, 28, 29, 31, 32, 34, 35, 37, 39, 41, 43, 45, 47, 49, 51, 54, 56,
59, 62, 65, 68, 71, 75, 78, 82, 86, 90, 95, 99, 104, 109, 115, 120, 126, 132, 138, 145, 152, 160, 167, 176, 184, 193, 202, 212, 222,
```

```
233, 245, 256
 };
 const int16_t CRYNET_MEL_OFFSET[CRYNET_N_MELS] = {
   0, 3, 6, 9, 12, 15, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 53, 56, 59, 62, 65, 68, 72, 76, 80, 84, 88, 92, 96, 101,
106, 111, 117, 123, 129, 135, 142, 149, 156, 164, 172, 181, 190, 199, 209, 220, 231, 242, 254, 266, 279, 293, 308, 323, 339, 356, 373,
391, 410, 430, 451, 474
 };
 const float CRYNET_MEL_WEIGHTS[497] = {
   0.00529403f, 0.01999968f, 0.00868042f, 0.01301245f, 0.01566764f, 0.00096200f, 0.00602523f, 0.02073087f, 0.00794922f,
0.01374365f,
   0.01493645f, 0.00023080f, 0.00675642f, 0.02146207f, 0.00721803f, 0.01447484f, 0.01420525f, 0.00748762f, 0.02119248f,
0.00648683f,
   0.00050039f, 0.01520604f, 0.01347406f, 0.00821881f, 0.02046128f, 0.00575564f, 0.00123159f, 0.01593723f, 0.01274286f,
0.00895001f,
   0.01973009f, 0.00502444f, 0.00196278f, 0.01666843f, 0.01201167f, 0.00968120f, 0.01899889f, 0.00429325f, 0.00269398f,
0.01739962f,
   0.01128047f, 0.01041240f, 0.01826770f, 0.00356205f, 0.00342517f, 0.01813082f, 0.01054928f, 0.01114359f, 0.01753650f,
0.00283086f,
   0.00415637f, 0.01886201f, 0.00981808f, 0.01187479f, 0.01680531f, 0.00209966f, 0.00488756f, 0.01959321f, 0.00908689f,
0.01237826f,
   0.01597954f, 0.00205200f, 0.00508658f, 0.01839938f, 0.00960351f, 0.01022684f, 0.01662575f, 0.00518864f, 0.00260424f,
0.01351035f,
   0.01277637f, 0.00237660f, 0.00541794f, 0.01533487f, 0.01030567f, 0.00084916f, 0.00695676f, 0.01597423f, 0.00895729f,
0.00035848f,
   0.00746330f, 0.01566288f, 0.00851178f, 0.00069288f, 0.00714506f, 0.01460095f, 0.00878197f, 0.00167224f, 0.00617886f,
0.01295850f,
   0.00960879f, 0.00314390f, 0.00471476f, 0.01087949f, 0.01085767f, 0.00497915f, 0.00287956f, 0.00848516f, 0.01241534f,
0.00707000f,
   0.00172467f, 0.00077983f, 0.00587699f, 0.01097416f, 0.00932648f, 0.00446596f, 0.00313943f, 0.00777428f, 0.01167412f,
0.00725445f,
   0.00283478f, 0.00034212f, 0.00455659f, 0.00877107f, 0.01003310f, 0.00601429f, 0.00199548f, 0.00137423f, 0.00520646f,
0.00903869f,
   0.00910170f, 0.00544740f, 0.00179309f, 0.00175440f, 0.00523904f, 0.00872369f, 0.00874113f, 0.00541827f, 0.00209541f,
0.00161383f,
   0.00478241f, 0.00795100f, 0.00883301f, 0.00581153f, 0.00279005f, 0.00106430f, 0.00394550f, 0.00682670f, 0.00927677f,
0.00652934f,
   0.00378191f, 0.00103448f, 0.00020067f, 0.00282055f, 0.00544042f, 0.00806030f, 0.00748916f, 0.00499092f, 0.00249267f,
0.00148527f,
   0.00386753f, 0.00624978f, 0.00862167f, 0.00635001f, 0.00407836f, 0.00180671f, 0.00000483f, 0.00217101f, 0.00433719f,
0.00650338f,
   0.00780344f, 0.00573783f, 0.00367221f, 0.00160660f, 0.00040314f, 0.00237285f, 0.00434257f, 0.00631228f, 0.00742691f,
0.00554864f,
   0.00367038f, 0.00179212f, 0.00039800f, 0.00218906f, 0.00398012f, 0.00577118f, 0.00740176f, 0.00569386f, 0.00398595f,
0.00227805f,
   0.00057014f, 0.00007470f, 0.00170331f, 0.00333192f, 0.00496053f, 0.00658914f, 0.00609824f, 0.00454524f, 0.00299224f,
0.00143925f,
   0.00098654f, 0.00246743f, 0.00394833f, 0.00542923f, 0.00669822f, 0.00528607f, 0.00387393f, 0.00246179f, 0.00104965f,
0.00009863f,
   0.00144521f, 0.00279179f, 0.00413837f, 0.00548495f, 0.00615625f, 0.00487219f, 0.00358813f, 0.00230407f, 0.00102000f,
0.00031431f,
   0.00153876f, 0.00276321f, 0.00398765f, 0.00521210f, 0.00594464f, 0.00477704f, 0.00360944f, 0.00244184f, 0.00127425f,
0.00010665f,
   0.00022896f, 0.00134235f, 0.00245574f, 0.00356913f, 0.00468252f, 0.00579591f, 0.00493288f, 0.00387119f, 0.00280949f,
0.00174779f,
   0.00068609f, 0.00091993f, 0.00193234f, 0.00294474f, 0.00395715f, 0.00496955f, 0.00528226f, 0.00431685f, 0.00335145f,
0.00238605f,
   0.00142065f, 0.00045525f, 0.00032568f, 0.00124626f, 0.00216684f, 0.00308742f, 0.00400800f, 0.00492858f, 0.00489881f,
0.00402097f,
   0.00314313f, 0.00226529f, 0.00138745f, 0.00050961f, 0.00044235f, 0.00127943f, 0.00211652f, 0.00295360f, 0.00379068f,
0.00462777f,
   0.00477888f, 0.00398066f, 0.00318244f, 0.00238422f, 0.00158600f, 0.00078778f, 0.00031929f, 0.00108045f, 0.00184161f,
0.00260277f,
   0.00336393f, 0.00412509f, 0.00486681f, 0.00414098f, 0.00341516f, 0.00268934f, 0.00196352f, 0.00123770f, 0.00051187f,
0.00000905f,
   0.00070118f, 0.00139330f, 0.00208542f, 0.00277755f, 0.00346967f, 0.00416180f, 0.00445536f, 0.00379537f, 0.00313538f,
0.00247539f,
   0.00181540f, 0.00115541f, 0.00049542f, 0.00018551f, 0.00081486f, 0.00144421f, 0.00207356f, 0.00270291f, 0.00333225f,
0.00396160f,
   0.00428438f, 0.00368425f, 0.00308412f, 0.00248399f, 0.00188386f, 0.00128373f, 0.00068360f, 0.00008347f, 0.00014270f,
0.00071496f,
   0.00128723f, 0.00185950f, 0.00243176f, 0.00300403f, 0.00357630f, 0.00414857f, 0.00375836f, 0.00321266f, 0.00266696f,
0.00212127f,
   0.00157557f, 0.00102987f, 0.00048417f, 0.00044799f, 0.00096835f, 0.00148871f, 0.00200908f, 0.00252944f, 0.00304980f,
0.00357016f,
   0.00397591f, 0.00347971f, 0.00298350f, 0.00248730f, 0.00199110f, 0.00149489f, 0.00099869f, 0.00050249f, 0.00000628f,
0.00005335f,
   0.00052651f, 0.00099968f, 0.00147285f, 0.00194601f, 0.00241918f, 0.00289234f, 0.00336551f, 0.00383867f, 0.00339918f,
0.00294799f,
   0.00249679f, 0.00204559f, 0.00159439f, 0.00114319f, 0.00069200f, 0.00024080f, 0.00042480f, 0.00085505f, 0.00128530f,
0.00171555f,
   0.00214580f, 0.00257605f, 0.00300630f, 0.00343655f, 0.00347485f, 0.00306458f, 0.00265430f, 0.00224403f, 0.00183375f,
0.00142348f,
   0.00101321f, 0.00060293f, 0.00019266f, 0.00018243f, 0.00057366f, 0.00096489f, 0.00135611f, 0.00174734f, 0.00213857f,
0.00252979f,
   0.00292102f, 0.00331225f, 0.00329808f, 0.00292502f, 0.00255195f, 0.00217889f, 0.00180583f, 0.00143276f, 0.00105970f,
```

```
0.00068664f,
  76    0.00031358f, 0.00018869f, 0.00054443f, 0.00090018f, 0.00125592f, 0.00161166f, 0.00196740f, 0.00232315f, 0.00267889f,
0.00303463f,
  77    0.00327956f, 0.00294033f, 0.00260111f, 0.00226188f, 0.00192265f, 0.00158343f, 0.00124420f, 0.00090498f, 0.00056575f,
0.00022652f,
  78    0.00005158f, 0.00037506f, 0.00069853f, 0.00102201f, 0.00134549f, 0.00166896f, 0.00199244f, 0.00231592f, 0.00263939f,
0.00296287f,
  79    0.00307639f, 0.00276794f, 0.00245948f, 0.00215102f, 0.00184256f, 0.00153410f, 0.00122564f, 0.00091719f, 0.00060873f,
0.00030027f,
  80    0.00009772f, 0.00039186f, 0.00068600f, 0.00098014f, 0.00127427f, 0.00156841f, 0.00186255f, 0.00215669f, 0.00245082f,
0.00274496f,
  81    0.00302384f, 0.00274336f, 0.00246288f, 0.00218240f, 0.00190192f, 0.00162143f, 0.00134095f, 0.00106047f, 0.00077999f,
0.00049951f,
  82    0.00021903f, 0.00000710f, 0.00027456f, 0.00054202f, 0.00080948f, 0.00107694f, 0.00134440f, 0.00161186f, 0.00187932f,
0.00214678f,
  83    0.00241423f, 0.00268169f, 0.00283467f, 0.00257963f, 0.00232459f, 0.00206955f, 0.00181451f, 0.00155946f, 0.00130442f,
0.00104938f,
  84    0.00079434f, 0.00053930f, 0.00028426f, 0.00002921f, 0.00005329f, 0.00029649f, 0.00053969f, 0.00078289f, 0.00102609f,
0.00126929f,
  85    0.00151249f, 0.00175569f, 0.00199889f, 0.00224209f, 0.00248529f, 0.00272849f, 0.00255101f, 0.00231910f, 0.00208719f,
0.00185528f,
  86    0.00162337f, 0.00139146f, 0.00115955f, 0.00092764f, 0.00069573f, 0.00046382f, 0.00023191f,
  87  };
  88
  89  const float CRYNET_HANN[CRYNET_N_FFT] = {
  90    0.00000000f, 0.00003780f, 0.00015118f, 0.00034013f, 0.00060463f, 0.00094463f, 0.00136008f, 0.00185091f, 0.00241706f,
0.00305844f,
  91    0.00377495f, 0.00456648f, 0.00543291f, 0.00637411f, 0.00738995f, 0.00848026f, 0.00964488f, 0.01088363f, 0.01219634f,
0.01358279f,
  92    0.01504278f, 0.01657609f, 0.01818249f, 0.01986173f, 0.02161357f, 0.02343773f, 0.02533393f, 0.02730190f, 0.02934134f,
0.03145194f,
  93    0.03363337f, 0.03588531f, 0.03820742f, 0.04059934f, 0.04306072f, 0.04559118f, 0.04819034f, 0.05085782f, 0.05359319f,
0.05639606f,
  94    0.05926599f, 0.06220255f, 0.06520531f, 0.06827380f, 0.07140756f, 0.07460611f, 0.07786898f, 0.08119567f, 0.08458568f,
0.08803850f,
  95    0.09155359f, 0.09513044f, 0.09876850f, 0.10246722f, 0.10622604f, 0.11004440f, 0.11392171f, 0.11785739f, 0.12185084f,
0.12590148f,
  96    0.13000865f, 0.13417177f, 0.13839021f, 0.14266330f, 0.14699043f, 0.15137093f, 0.15580413f, 0.16028938f, 0.16482598f,
0.16941325f,
  97    0.17405050f, 0.17873703f, 0.18347214f, 0.18825510f, 0.19308519f, 0.19796169f, 0.20288384f, 0.20785092f, 0.21286216f,
0.21791682f,
  98    0.22301412f, 0.22815330f, 0.23333357f, 0.23855416f, 0.24381429f, 0.24911314f, 0.25444993f, 0.25982383f, 0.26523405f,
0.27067977f,
  99    0.27616015f, 0.28167439f, 0.28722161f, 0.29280102f, 0.29841176f, 0.30405295f, 0.30972379f, 0.31542337f, 0.32115087f,
0.32690543f,
 100    0.33268613f, 0.33849216f, 0.34432256f, 0.35017651f, 0.35605314f, 0.36195150f, 0.36787075f, 0.37380999f, 0.37976828f,
0.38574475f,
 101    0.39173850f, 0.39774862f, 0.40377420f, 0.40981433f, 0.41586810f, 0.42193457f, 0.42801285f, 0.43410203f, 0.44020116f,
0.44630933f,
 102    0.45242563f, 0.45854908f, 0.46467885f, 0.47081393f, 0.47695342f, 0.48309642f, 0.48924196f, 0.49538910f, 0.50153697f,
0.50768459f,
 103    0.51383108f, 0.51997548f, 0.52611679f, 0.53225422f, 0.53838676f, 0.54451346f, 0.55063349f, 0.55674583f, 0.56284958f,
0.56894386f,
 104    0.57502770f, 0.58110023f, 0.58716041f, 0.59320748f, 0.59924048f, 0.60525841f, 0.61126047f, 0.61724567f, 0.62321317f,
0.62916207f,
 105    0.63509142f, 0.64100033f, 0.64688796f, 0.65275335f, 0.65859562f, 0.66441399f, 0.67020744f, 0.67597520f, 0.68171632f,
0.68742996f,
 106    0.69311529f, 0.69877142f, 0.70439750f, 0.70999265f, 0.71555609f, 0.72108692f, 0.72658432f, 0.73204744f, 0.73747551f,
0.74286771f,
 107    0.74822313f, 0.75354105f, 0.75882065f, 0.76406109f, 0.76926166f, 0.77442145f, 0.77953982f, 0.78461587f, 0.78964895f,
0.79463822f,
 108    0.79958290f, 0.80448234f, 0.80933571f, 0.81414235f, 0.81890142f, 0.82361233f, 0.82827431f, 0.83288670f, 0.83744872f,
0.84195971f,
 109    0.84641904f, 0.85082597f, 0.85517985f, 0.85948002f, 0.86372590f, 0.86791676f, 0.87205195f, 0.87613094f, 0.88015306f,
0.88411772f,
 110    0.88802427f, 0.89187217f, 0.89566082f, 0.89938968f, 0.90305817f, 0.90666568f, 0.91021168f, 0.91369575f, 0.91711718f,
0.92047560f,
 111    0.92377049f, 0.92700124f, 0.93016744f, 0.93326867f, 0.93630433f, 0.93927401f, 0.94217736f, 0.94501376f, 0.94778293f,
0.95048445f,
 112    0.95311779f, 0.95568264f, 0.95817864f, 0.96060532f, 0.96296239f, 0.96524948f, 0.96746618f, 0.96961224f, 0.97168732f,
0.97369105f,
 113    0.97562313f, 0.97748339f, 0.97927141f, 0.98098695f, 0.98262984f, 0.98419970f, 0.98569638f, 0.98711962f, 0.98846924f,
0.98974502f,
 114    0.99094671f, 0.99207419f, 0.99312729f, 0.99410582f, 0.99500966f, 0.99583864f, 0.99659270f, 0.99727166f, 0.99787539f,
0.99840391f,
 115    0.99885708f, 0.99923480f, 0.99953705f, 0.99976379f, 0.99991494f, 0.99999052f, 0.99999052f, 0.99991494f, 0.99976379f,
0.99953705f,
 116    0.99923480f, 0.99885708f, 0.99840391f, 0.99787539f, 0.99727166f, 0.99659270f, 0.99583864f, 0.99500966f, 0.99410582f,
0.99312729f,
 117    0.99207419f, 0.99094671f, 0.98974502f, 0.98846924f, 0.98711962f, 0.98569638f, 0.98419970f, 0.98262984f, 0.98098695f,
0.97927141f,
 118    0.97748339f, 0.97562313f, 0.97369105f, 0.97168732f, 0.96961224f, 0.96746618f, 0.96524948f, 0.96296239f, 0.96060532f,
0.95817864f,
 119    0.95568264f, 0.95311779f, 0.95048445f, 0.94778293f, 0.94501376f, 0.94217736f, 0.93927401f, 0.93630433f, 0.93326867f,
0.93016744f,
```

```
    0.92700124f, 0.92377049f, 0.92047560f, 0.91711718f, 0.91369575f, 0.91021168f, 0.90666568f, 0.90305817f, 0.89938968f,
0.89566082f,
    0.89187217f, 0.88802427f, 0.88411772f, 0.88015306f, 0.87613094f, 0.87205195f, 0.86791676f, 0.86372590f, 0.85948002f,
0.85517985f,
    0.85082597f, 0.84641904f, 0.84195971f, 0.83744872f, 0.83288670f, 0.82827431f, 0.82361233f, 0.81890142f, 0.81414235f,
0.80933571f,
    0.80448234f, 0.79958290f, 0.79463822f, 0.78964895f, 0.78461587f, 0.77953982f, 0.77442145f, 0.76926166f, 0.76406109f,
0.75882065f,
    0.75354105f, 0.74822313f, 0.74286771f, 0.73747551f, 0.73204744f, 0.72658432f, 0.72108692f, 0.71555609f, 0.70999265f,
0.70439750f,
    0.69877142f, 0.69311529f, 0.68742996f, 0.68171632f, 0.67597520f, 0.67020744f, 0.66441399f, 0.65859562f, 0.65275335f,
0.64688796f,
    0.64100033f, 0.63509142f, 0.62916207f, 0.62321317f, 0.61724567f, 0.61126047f, 0.60525841f, 0.59924048f, 0.59320748f,
0.58716041f,
    0.58110023f, 0.57502770f, 0.56894386f, 0.56284958f, 0.55674583f, 0.55063349f, 0.54451346f, 0.53838676f, 0.53225422f,
0.52611679f,
    0.51997548f, 0.51383108f, 0.50768459f, 0.50153697f, 0.49538910f, 0.48924196f, 0.48309642f, 0.47695342f, 0.47081393f,
0.46467885f,
    0.45854908f, 0.45242563f, 0.44630933f, 0.44020116f, 0.43410203f, 0.42801285f, 0.42193457f, 0.41586810f, 0.40981433f,
0.40377420f,
    0.39774862f, 0.39173850f, 0.38574475f, 0.37976828f, 0.37380999f, 0.36787075f, 0.36195150f, 0.35605314f, 0.35017651f,
0.34432256f,
    0.33849216f, 0.33268613f, 0.32690543f, 0.32115087f, 0.31542337f, 0.30972379f, 0.30405295f, 0.29841176f, 0.29280102f,
0.28722161f,
    0.28167439f, 0.27616015f, 0.27067977f, 0.26523405f, 0.25982383f, 0.25444993f, 0.24911314f, 0.24381429f, 0.23855416f,
0.23333357f,
    0.22815330f, 0.22301412f, 0.21791682f, 0.21286216f, 0.20785092f, 0.20288384f, 0.19796169f, 0.19308519f, 0.18825510f,
0.18347214f,
    0.17873703f, 0.17405050f, 0.16941325f, 0.16482598f, 0.16028938f, 0.15580413f, 0.15137093f, 0.14699043f, 0.14266330f,
0.13839021f,
    0.13417177f, 0.13000865f, 0.12590148f, 0.12185084f, 0.11785739f, 0.11392171f, 0.11004440f, 0.10622604f, 0.10246722f,
0.09876850f,
    0.09513044f, 0.09155359f, 0.08803850f, 0.08458568f, 0.08119567f, 0.07786898f, 0.07460611f, 0.07140756f, 0.06827380f,
0.06520531f,
    0.06220255f, 0.05926599f, 0.05639606f, 0.05359319f, 0.05085782f, 0.04819034f, 0.04559118f, 0.04306072f, 0.04059934f,
0.03820742f,
    0.03588531f, 0.03363337f, 0.03145194f, 0.02934134f, 0.02730190f, 0.02533393f, 0.02343773f, 0.02161357f, 0.01986173f,
0.01818249f,
    0.01657609f, 0.01504278f, 0.01358279f, 0.01219634f, 0.01088363f, 0.00964488f, 0.00848026f, 0.00738995f, 0.00637411f,
0.00543291f,
    0.00456648f, 0.00377495f, 0.00305844f, 0.00241706f, 0.00185091f, 0.00136008f, 0.00094463f, 0.00060463f, 0.00034013f,
0.00015118f,
    0.00003780f, 0.00000000f,
};

#endif /* CRYNET_TABLES_H */
```

firmware/hub_infer/components/crynet_model/include/deploy_config.json

71 lines · 1,811 bytes

```
{
  "model": "crynet_int8.tflite",
  "model_sha256": "bd592d541027b9f448bc218e6fdbef113e39d64b8a59f366a18586987dce43b9",
  "model_bytes": 137384,
  "input": {
    "name": "mel",
    "shape": [
      1,
      81,
      64,
      1
    ],
    "dtype": "int8",
    "scale": 0.11632965505123138,
    "zero_point": -70,
    "preprocessing": [
      "int16 PCM, 16000 Hz mono, 32000 samples (2.0s)",
      "subtract mean (DC removal)",
      "zero-pad 256 front / 512 rear",
      "81 frames, 512-pt Hann, hop 400",
      "64 Mel bands (librosa slaney, 20.0-8000.0 Hz)",
      "10*log10(max(P, 1e-10)) floored at -80.0 dB",
      "per-window standardisation",
      "q = round(v/scale) + zero_point"
    ]
  },
  "output": {
    "shape": [
      1,
      2
    ],
```

```
    "dtype": "int8",
    "scale": 0.00390625,
    "zero_point": -128,
    "classes": [
      "no_cry",
      "cry"
    ]
  },
  "operators": [
    "CONV_2D",
    "FULLY_CONNECTED",
    "MAX_POOL_2D",
    "MEAN",
    "SOFTMAX"
  ],
  "tflm_kernel_check": "ok",
  "decision": {
    "threshold": 0.9500000000000002,
    "silence_rms_gate": 0.0015,
    "rule": "P(cry) >= threshold  ->  ALERT"
  },
  "tflite_micro": {
    "arena_bytes_hint": 120000,
    "note": "put the tensor arena in PSRAM if it does not fit in internal SRAM"
  },
  "metrics": {
    "selected_model": "conv_cnn",
    "parameters": 122770,
    "val_auc": 0.9993736539019185,
    "test_auc": 0.9993031191248095,
    "cross_auc": 0.9984265123278009,
    "val_f1": 0.987518910741301,
    "test_f1": 0.9885514239757782,
    "cross_f1": 0.9623748705557473,
    "val_recall": 0.9797373358348969,
    "val_precision": 0.9954250857796416,
    "int8_val_auc": 0.9978066188376747,
    "float_val_auc": 0.9990680042199986
  }
}
```

Hub - the wire protocol

firmware/hub_infer/components/crynet_proto/include/crynet_proto.h

224 lines · 9,570 bytes

```
/*
 * CryNet — the node <-> hub wire protocol.
 *
 * This is the hub's copy of a contract that already exists in the node's
 * sketch (firmware/node/node_crynet/node_crynet.ino).  Both sides were
 * written from the same table, and this component exists so that the hub
 * has the layout in a place a compiler can check, rather than as prose in
 * a comment.
 *
 * Frame (node -> hub):
 *
 *     0  0xC3            magic
 *     1  0x7E            magic
 *     2  type            1 = telemetry, 2 = burst fragment
 *     3  node id
 *     4  u16 burst seq
 *     6  u16 frag index
 *     8  u16 frag count
 *    10  u16 payload len
 *    12  u16 payload CRC-16/CCITT-FALSE
 *    14  payload
 *    ..  u16 frame CRC-16/CCITT-FALSE over everything above
 *
 * A request (hub -> node) is twelve bytes.  It has no payload, so it
 * carries only the CRC of its first ten bytes:
 *
 *     0  0xC3  1 0x7E  2 type=3  3 node id
 *     4  u16 burst seq   6 u16 first fragment   8 u16 fragment count
 *    10  u16 CRC-16/CCITT-FALSE over the first ten bytes
 *
 * The telemetry payload is sixteen bytes:
 *
```

```
 *     0  u32 rms x 1000          the node's 100 ms reading
 *     4  u16 window ms
 *     6  u8  flags
 *     7  u16 event level x 1000  the level the node triggers at
 *     9  u16 frozen rms x 1000   the level of the frozen window itself
 *    11  u8  floor x 1000        the node's noise floor
 *    12  u32 frozen crc          CRC-32 of the frozen window
 *
 * The event level is the node's own: it tracks the room's noise floor,
 * so the level that follows from it is the node's to decide, and the hub
 * triggers on the number the node reports.  A node that does not report
 * one (an older node) is given the hub's fixed level instead.
 */

#ifndef CRYNET_PROTO_H
#define CRYNET_PROTO_H

#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

#define CRYNET_PROTO_MAGIC0        0xC3u
#define CRYNET_PROTO_MAGIC1        0x7Eu

#define CRYNET_PROTO_TYPE_TELEMETRY 1u
#define CRYNET_PROTO_TYPE_FRAGMENT  2u
#define CRYNET_PROTO_TYPE_REQUEST   3u
/* A heartbeat: the hub sends one every 100 ms so that its radio stays
 * awake.  A radio that is asleep between transmissions misses almost
 * everything the nodes send. */
#define CRYNET_PROTO_TYPE_HEARTBEAT 4u

/*
 * A resend request: the hub's list of the fragments of a burst that did
 * not arrive.  It carries the full fourteen-byte header and a payload,
 * and the payload is a bitmap — one bit per fragment, the bit set for
 * the fragments the hub is still missing.  Thirty-six bytes cover all
 * 286 fragments, so one request says everything that is missing however
 * scattered the losses are, and the whole frame is 54 bytes, well inside
 * the 255-byte ESP-NOW limit.
 *
 * The node answers by sending exactly the fragments whose bit is set,
 * out of the same frozen window and with the same sequence number, so
 * the fragments that did arrive are not sent again and the ones that did
 * not are now covered.
 */
#define CRYNET_PROTO_TYPE_RESEND   5u

#define CRYNET_PROTO_HEADER_BYTES  14u
#define CRYNET_PROTO_REQUEST_BYTES 12u

/* The bitmap in a resend request: one bit per fragment. */
#define CRYNET_PROTO_RESEND_BITMAP_BYTES \
    ((CRYNET_PROTO_FRAG_COUNT + 7u) / 8u)

/* ESP-NOW carries at most 255 bytes in one frame. */
#define CRYNET_PROTO_MAX_FRAME     255u
#define CRYNET_PROTO_MAX_PAYLOAD   (CRYNET_PROTO_MAX_FRAME - CRYNET_PROTO_HEADER_BYTES - 2u)

/* The signal contract: 2 s of 16 kHz mono int16. */
#define CRYNET_PROTO_WINDOW_SAMPLES 32000u
#define CRYNET_PROTO_WINDOW_BYTES   (CRYNET_PROTO_WINDOW_SAMPLES * 2u)

/* A fragment carries this many payload bytes; the node's frame is
 * 14 + 224 + 2 = 240 bytes, comfortably inside the ESP-NOW limit. */
#define CRYNET_PROTO_FRAG_PAYLOAD   224u
#define CRYNET_PROTO_FRAG_COUNT     \
    ((CRYNET_PROTO_WINDOW_BYTES + CRYNET_PROTO_FRAG_PAYLOAD - 1u) / CRYNET_PROTO_FRAG_PAYLOAD)

/* The telemetry payload is 16 bytes. */
#define CRYNET_PROTO_TELEMETRY_BYTES 16u

/* The radio-counter tier: the same 16 bytes, then the node's own radio
 * counters — the frames it handed to the radio, the frames the radio
 * refused, the frames that actually left it, and the number of times it
 * had to wait for the radio to have room.
 *
 * These are the counters that say where a burst went.  A node that hands
 * over all 286 fragments and sees only a few of them come back has lost
 * them in the air; a node whose radio refused them has a queue that is
 * too small for the burst.  The hub cannot tell the two apart from the
```

```
 * fragments alone, and the difference is the difference between moving
 * a cot and rewriting a radio's pacing.
 *
 * There was a battery tier between this one and the base tier — the
 * node's cell voltage, in two bytes.  It is gone (D-83): the cots do
 * not measure their cells, so the tier would only have carried zeroes,
 * and the counters moved down to fill the gap. */
#define CRYNET_PROTO_TELEMETRY_BYTES_COUNTERS \
    (CRYNET_PROTO_TELEMETRY_BYTES + 16u)

/* The payload a node flashed before the battery tier was removed sent:
 * the same bytes, plus the battery voltage, plus the counters.  The hub
 * still reads it, so a node that has not been reflashed yet does not
 * turn the hub's counters into noise while the units are being
 * reflashed one at a time. */
#define CRYNET_PROTO_TELEMETRY_BYTES_OLD_COUNTERS \
    (CRYNET_PROTO_TELEMETRY_BYTES + 2u + 16u)

/* The payload an older node sent: the same fields, without the event
 * level, the window's own level or the noise floor, and with the CRC
 * four bytes earlier. */
#define CRYNET_PROTO_TELEMETRY_BYTES_LEGACY 12u

/* Telemetry flags. */
#define CRYNET_PROTO_FLAG_BURST_READY   0x01u  /* the node has a frozen window  */
#define CRYNET_PROTO_FLAG_EVENT         0x02u  /* the node's event level is crossed */
#define CRYNET_PROTO_FLAG_WINDOW_FULL   0x04u  /* the live window holds 2 s     */

/* CRC-16/CCITT-FALSE: polynomial 0x1021, init 0xFFFF, no reflection. */
uint16_t crynet_proto_crc16(const uint8_t *data, size_t len);

/*
 * CRC-32, the IEEE 802.3 one: reflected, polynomial 0xEDB88320, init
 * 0xFFFFFFFF, final XOR 0xFFFFFFFF — the same function zlib's crc32()
 * computes.  The node puts this over its frozen window and advertises it
 * in the telemetry; the hub puts it over the window it reassembles from
 * the radio.  The two agreeing is the proof that the audio crossed the
 * link unchanged.
 */
uint32_t crynet_proto_crc32(const uint8_t *data, size_t len);

/*
 * Build a frame into `out`, which must have room for CRYNET_PROTO_MAX_FRAME
 * bytes.  Returns the total length including the trailing frame CRC, or 0
 * if the payload is too long.  A request frame (type 3) is built with the
 * twelve-byte layout; every other type uses the full header.
 */
size_t crynet_proto_build(uint8_t *out,
                          uint8_t type,
                          uint8_t node_id,
                          uint16_t seq,
                          uint16_t frag,
                          uint16_t frag_count,
                          const uint8_t *payload,
                          uint16_t payload_len);

/* Parse results. */
#define CRYNET_PROTO_BAD_FRAME    (-1)  /* malformed, or the frame CRC failed  */
#define CRYNET_PROTO_BAD_PAYLOAD  (-2)  /* the frame is sound, the payload is not */

/*
 * Parse a frame received from the radio.  On success the payload length is
 * returned (0 for a request) and the out parameters are filled in; for a
 * request, `payload` is set to NULL.  On failure one of the CRYNET_PROTO_BAD_*
 * codes is returned.
 */
int crynet_proto_parse(const uint8_t *data,
                       size_t len,
                       uint8_t *type,
                       uint8_t *node_id,
                       uint16_t *seq,
                       uint16_t *frag,
                       uint16_t *frag_count,
                       const uint8_t **payload);

/* Decode a telemetry payload. */
typedef struct {
    float    rms;        /* the node's 100 ms RMS, in full-scale units */
    uint16_t window_ms;  /* the length of the node's window           */
    uint8_t  flags;
    /* The level the node triggers at, in full-scale units.  Zero means
     * the node did not report one. */
    float    event_rms;
    /* The level of the node's frozen window itself, in full-scale
     * units: what the model will be shown, as opposed to the live
```

```
     * reading beside it. */
    float    frozen_rms;
    /* The node's noise floor, in full-scale units. */
    float    floor_rms;
    uint32_t frozen_crc; /* CRC-32 of the node's frozen window, 0 if none */
    /* The node's radio counters.  Zero means the node did not report
     * them: its firmware predates the field. */
    uint32_t frames_sent;   /* frames handed over to the radio   */
    uint32_t frames_lost;   /* frames the radio refused outright  */
    uint32_t frames_acked;  /* frames that actually left the radio */
    uint32_t radio_full;    /* times the radio had no room        */
} crynet_proto_telemetry_t;

bool crynet_proto_decode_telemetry(const uint8_t *payload,
                                   size_t len,
                                   crynet_proto_telemetry_t *out);

#ifdef __cplusplus
}
#endif

#endif /* CRYNET_PROTO_H */
```

firmware/hub_infer/components/crynet_proto/crynet_proto.c

239 lines · 8,276 bytes

```
/*
 * CryNet — the node <-> hub wire protocol.  See crynet_proto.h.
 */

#include "crynet_proto.h"

#include <string.h>

uint16_t crynet_proto_crc16(const uint8_t *data, size_t len)
{
    uint16_t crc = 0xFFFFu;

    for (size_t i = 0; i < len; i++) {
        crc ^= (uint16_t)data[i] << 8;
        for (int bit = 0; bit < 8; bit++) {
            crc = (crc & 0x8000u) ? (uint16_t)((crc << 1) ^ 0x1021u)
                                  : (uint16_t)(crc << 1);
        }
    }
    return crc;
}

uint32_t crynet_proto_crc32(const uint8_t *data, size_t len)
{
    uint32_t crc = 0xFFFFFFFFu;

    for (size_t i = 0; i < len; i++) {
        crc ^= data[i];
        for (int bit = 0; bit < 8; bit++) {
            crc = (crc & 1u) ? ((crc >> 1) ^ 0xEDB88320u) : (crc >> 1);
        }
    }
    return crc ^ 0xFFFFFFFFu;
}

static void put_u16(uint8_t *p, uint16_t v)
{
    p[0] = (uint8_t)(v & 0xFFu);
    p[1] = (uint8_t)(v >> 8);
}

static uint16_t get_u16(const uint8_t *p)
{
    return (uint16_t)p[0] | ((uint16_t)p[1] << 8);
}

static uint32_t get_u32(const uint8_t *p)
{
    return (uint32_t)p[0] | ((uint32_t)p[1] << 8) |
           ((uint32_t)p[2] << 16) | ((uint32_t)p[3] << 24);
}

size_t crynet_proto_build(uint8_t *out,
                          uint8_t type,
                          uint8_t node_id,
```

```
                          uint16_t seq,
                          uint16_t frag,
                          uint16_t frag_count,
                          const uint8_t *payload,
                          uint16_t payload_len)
{
    if (type == CRYNET_PROTO_TYPE_REQUEST ||
        type == CRYNET_PROTO_TYPE_HEARTBEAT) {
        /* The twelve-byte request layout, which the heartbeat shares: no
         * payload, no payload CRC.  The parser treats the two the same
         * way and rejects any other length, so a heartbeat built with
         * the full header would be a malformed frame on the far end. */
        out[0] = CRYNET_PROTO_MAGIC0;
        out[1] = CRYNET_PROTO_MAGIC1;
        out[2] = type;
        out[3] = node_id;
        put_u16(&out[4], seq);
        put_u16(&out[6], frag);
        put_u16(&out[8], frag_count);
        put_u16(&out[10], crynet_proto_crc16(out, 10));
        return CRYNET_PROTO_REQUEST_BYTES;
    }

    if (payload_len > CRYNET_PROTO_MAX_PAYLOAD) {
        return 0;
    }

    out[0] = CRYNET_PROTO_MAGIC0;
    out[1] = CRYNET_PROTO_MAGIC1;
    out[2] = type;
    out[3] = node_id;
    put_u16(&out[4], seq);
    put_u16(&out[6], frag);
    put_u16(&out[8], frag_count);
    put_u16(&out[10], payload_len);

    uint16_t pcrc = 0;
    if (payload_len > 0) {
        memcpy(&out[CRYNET_PROTO_HEADER_BYTES], payload, payload_len);
        pcrc = crynet_proto_crc16(payload, payload_len);
    }
    put_u16(&out[12], pcrc);

    const size_t total = CRYNET_PROTO_HEADER_BYTES + (size_t)payload_len;
    put_u16(&out[total], crynet_proto_crc16(out, total));
    return total + 2u;
}

int crynet_proto_parse(const uint8_t *data,
                       size_t len,
                       uint8_t *type,
                       uint8_t *node_id,
                       uint16_t *seq,
                       uint16_t *frag,
                       uint16_t *frag_count,
                       const uint8_t **payload)
{
    if (len < CRYNET_PROTO_REQUEST_BYTES) {
        return CRYNET_PROTO_BAD_FRAME;
    }
    if (data[0] != CRYNET_PROTO_MAGIC0 || data[1] != CRYNET_PROTO_MAGIC1) {
        return CRYNET_PROTO_BAD_FRAME;
    }

    const uint8_t t = data[2];
    if (t != CRYNET_PROTO_TYPE_TELEMETRY &&
        t != CRYNET_PROTO_TYPE_FRAGMENT &&
        t != CRYNET_PROTO_TYPE_REQUEST &&
        t != CRYNET_PROTO_TYPE_HEARTBEAT &&
        t != CRYNET_PROTO_TYPE_RESEND) {
        return CRYNET_PROTO_BAD_FRAME;
    }

    if (t == CRYNET_PROTO_TYPE_REQUEST || t == CRYNET_PROTO_TYPE_HEARTBEAT) {
        if (len != CRYNET_PROTO_REQUEST_BYTES) {
            return CRYNET_PROTO_BAD_FRAME;
        }
        if (crynet_proto_crc16(data, 10) != get_u16(&data[10])) {
            return CRYNET_PROTO_BAD_FRAME;
        }
        *type = t;
        *node_id = data[3];
        *seq = get_u16(&data[4]);
        *frag = get_u16(&data[6]);
        *frag_count = get_u16(&data[8]);
```

```
        *payload = NULL;
        return 0;
    }

    const uint16_t plen = get_u16(&data[10]);
    if (plen > CRYNET_PROTO_MAX_PAYLOAD) {
        return CRYNET_PROTO_BAD_FRAME;
    }
    const size_t total = CRYNET_PROTO_HEADER_BYTES + (size_t)plen;
    if (len != total + 2u) {
        return CRYNET_PROTO_BAD_FRAME;
    }
    if (crynet_proto_crc16(data, total) != get_u16(&data[total])) {
        return CRYNET_PROTO_BAD_FRAME;
    }

    *type = t;
    *node_id = data[3];
    *seq = get_u16(&data[4]);
    *frag = get_u16(&data[6]);
    *frag_count = get_u16(&data[8]);

    if (plen > 0) {
        if (crynet_proto_crc16(&data[CRYNET_PROTO_HEADER_BYTES], plen) != get_u16(&data[12])) {
            return CRYNET_PROTO_BAD_PAYLOAD;
        }
        *payload = &data[CRYNET_PROTO_HEADER_BYTES];
    } else {
        *payload = NULL;
    }
    return (int)plen;
}

bool crynet_proto_decode_telemetry(const uint8_t *payload,
                                   size_t len,
                                   crynet_proto_telemetry_t *out)
{
    if (payload == NULL ||
        len < CRYNET_PROTO_TELEMETRY_BYTES_LEGACY) {
        return false;
    }

    const uint32_t rms_milli = (uint32_t)payload[0] |
                               ((uint32_t)payload[1] << 8) |
                               ((uint32_t)payload[2] << 16) |
                               ((uint32_t)payload[3] << 24);
    out->rms = (float)rms_milli / 1000.0f;
    out->window_ms = get_u16(&payload[4]);
    out->flags = payload[6];

    if (len >= CRYNET_PROTO_TELEMETRY_BYTES) {
        /* The event level the node triggers at, the level of the frozen
         * window itself, and the node's noise floor. */
        out->event_rms  = (float)get_u16(&payload[7]) / 1000.0f;
        out->frozen_rms = (float)get_u16(&payload[9]) / 1000.0f;
        out->floor_rms  = (float)payload[11] / 1000.0f;
        out->frozen_crc = (uint32_t)payload[12] |
                          ((uint32_t)payload[13] << 8) |
                          ((uint32_t)payload[14] << 16) |
                          ((uint32_t)payload[15] << 24);
    } else {
        /* An older node: the level, the window's own level and the floor
         * are not on the wire, and the CRC sits four bytes earlier. */
        out->event_rms  = 0.0f;
        out->frozen_rms = 0.0f;
        out->floor_rms  = 0.0f;
        out->frozen_crc = (uint32_t)payload[8] |
                          ((uint32_t)payload[9] << 8) |
                          ((uint32_t)payload[10] << 16) |
                          ((uint32_t)payload[11] << 24);
    }

    /* The radio counters ride after the CRC.  A node that does not send
     * them decodes as zeroes.
     *
     * The counters used to start at byte 18, behind the two bytes of
     * battery voltage that used to sit there (D-83).  A node flashed
     * before that removal still sends the longer payload, so it is read
     * from where that node put them, and a node on the current firmware
     * from where this one puts them.  The length tells the two apart:
     * 32 bytes is the new layout, 34 the old one. */
    if (len >= CRYNET_PROTO_TELEMETRY_BYTES_OLD_COUNTERS) {
        out->frames_sent  = get_u32(&payload[18]);
        out->frames_lost  = get_u32(&payload[22]);
        out->frames_acked = get_u32(&payload[26]);
```

```
        out->radio_full   = get_u32(&payload[30]);
    } else if (len >= CRYNET_PROTO_TELEMETRY_BYTES_COUNTERS) {
        out->frames_sent  = get_u32(&payload[16]);
        out->frames_lost  = get_u32(&payload[20]);
        out->frames_acked = get_u32(&payload[24]);
        out->radio_full   = get_u32(&payload[28]);
    } else {
        out->frames_sent  = 0u;
        out->frames_lost  = 0u;
        out->frames_acked = 0u;
        out->radio_full   = 0u;
    }
    return true;
}
```

Hub - the radio

firmware/hub_infer/components/node_link/include/node_link.h

208 lines · 8,515 bytes

```
/*
 * CryNet — the hub's half of the node link.
 *
 * The node is a dumb sensor: it reports its 100 ms RMS over ESP-NOW every
 * 100 ms, and when the hub asks, it uploads its two-second window as 286
 * CRC-framed fragments.  This component is the hub's side of that
 * conversation:
 *
 *   - it receives and validates every frame the nodes send;
 *   - it keeps the telemetry of every node it has heard;
 *   - it reassembles a burst into one 32000-sample int16 window, in the
 *     right order, with every fragment's CRC checked;
 *   - it asks a node for that window.
 *
 * The window it hands back is the same PCM the node's microphone produced,
 * so the front end on the hub sees exactly the signal the contract
 * describes — no resampling, no gain change, nothing.
 *
 * Design note — reassembly tolerates loss by design.  A fragment that never
 * arrives leaves a hole, and the burst is reported as incomplete rather than
 * silently analysed as if it were whole.  The hub's decision is only ever
 * made on a window with every fragment present.
 */

#ifndef NODE_LINK_H
#define NODE_LINK_H

#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

#include "crynet_proto.h"

#ifdef __cplusplus
extern "C" {
#endif

/* This hub accepts cot IDs 1 and 2 only. */
#define NODE_LINK_MAX_NODES 2

typedef struct node_link node_link_t;

/* The newest telemetry seen from a node. */
typedef struct {
    uint8_t  node_id;
    uint16_t burst_seq;   /* the sequence of the window the node would send */
    float    rms;         /* the node's 100 ms RMS, in full-scale units */
    uint16_t window_ms;
    uint8_t  flags;
    /* The level the node triggers at, 0 if the node did not report one.
     * The node owns the room's noise floor, so it owns this. */
    float    event_rms;
    /* The level of the node's frozen window itself: what the model will
     * be shown, as opposed to the live reading beside it. */
    float    frozen_rms;
    float    floor_rms;   /* the node's noise floor */
    uint32_t frozen_crc;  /* the node's frozen-window CRC-32, 0 if it has none */
    uint32_t age_ms;      /* how long ago this telemetry arrived */
```

```
    int8_t   tel_rssi;    /* RSSI of the telemetry frame, in dBm; 0 = unknown */
    /* The node's own radio counters.  These are what say whether a
     * burst was lost in the air or refused by the node's radio: a node
     * that handed every fragment over and saw few of them come back has
     * lost them on the way, and a node whose radio refused them has a
     * queue too small for the burst. */
    uint32_t frames_sent;   /* frames handed to the radio   */
    uint32_t frames_lost;   /* frames the radio refused     */
    uint32_t frames_acked;  /* frames that left the radio   */
    uint32_t radio_full;    /* times the radio had no room  */
} node_link_telemetry_t;

/* The state of one node's burst. */
typedef struct {
    uint8_t  node_id;
    uint16_t burst_seq;
    uint16_t frag_want;
    uint16_t frag_have;
    uint16_t frag_seen;   /* fragment frames that arrived, stored or not */
    /* When the last fragment arrived, and how long this cot's
     * fragments have been taking to arrive.  Both are measured in the
     * receive callback, where the arrival happened: a gap measured by
     * the main loop is the gap between two of its own iterations, and
     * on a busy hub that is nearly two hundred milliseconds -- long
     * enough to mistake a burst that has ended for one that is still
     * in the air. */
    uint32_t last_frag_age_ms;
    uint32_t gap_ewma_ms;
    bool     complete;
    uint32_t bad_payloads;  /* fragments whose payload CRC failed */
    uint32_t missing;       /* fragments still outstanding */
    /* How loudly the burst's frames arrived, in dBm.  A burst that was
     * never heard has a weak maximum; one that was heard and lost is a
     * different fault altogether, and the number says which it was. */
    int8_t   rssi_min;
    int8_t   rssi_max;
    int      rssi_avg;
    /* The node's last telemetry reading, for comparison. */
    int8_t   tel_rssi;
} node_link_burst_t;

/* Counters for the whole link. */
typedef struct {
    uint32_t raw_frames;   /* frames off the air, before any parsing */
    uint32_t frag_frames;  /* fragment frames received, whatever they carried */
    uint32_t frames_ok;
    uint32_t frames_bad;      /* malformed or frame-CRC failures */
    uint32_t payloads_bad;    /* the frame was sound, the payload was not */
    uint32_t requests_sent;
    uint32_t requests_failed;
    uint32_t resends_sent;   /* requests for the fragments that went missing */
} node_link_stats_t;

/* Send the hub's regular heartbeat so cot nodes can measure the hub-side
 * signal strength from the RSSI of the frame they receive. */
bool node_link_send_heartbeat(node_link_t *nl);

/* Create the link.  Returns NULL only if the memory for the windows
 * cannot be found. */
node_link_t *node_link_create(void);

void node_link_destroy(node_link_t *nl);

/*
 * Bring up Wi-Fi and ESP-NOW and start listening.  Returns false if the
 * radio could not be started; the reason is logged by the component.
 *
 * The NVS partition must have been initialised (nvs_flash_init) before
 * this is called: the Wi-Fi driver keeps its calibration and MAC tables
 * there and opens that partition during esp_wifi_init().
 */
bool node_link_start(node_link_t *nl);

/* True once the radio is up. */
bool node_link_ready(const node_link_t *nl);

/*
 * Take the newest telemetry from any node.  Returns false when nothing new
 * has arrived since the last call.
 */
bool node_link_poll_telemetry(node_link_t *nl, node_link_telemetry_t *out);

/* The last telemetry seen from a node, whether or not it is new. */
bool node_link_last_telemetry(const node_link_t *nl, uint8_t node_id,
                              node_link_telemetry_t *out);
```

```

/*
 * Ask a node for its two-second window.  `burst_seq` is the sequence the
 * node advertised in its telemetry; sending it makes the node serve the
 * frozen snapshot it took when it crossed its event level.
 */
bool node_link_request_window(node_link_t *nl, uint8_t node_id, uint16_t burst_seq);

/* The state of a node's burst.  False if the node is unknown. */
bool node_link_burst_state(const node_link_t *nl, uint8_t node_id,
                           node_link_burst_t *out);

/* Which fragments of the burst in flight are still missing. */
int node_link_burst_missing(node_link_t *nl, uint8_t node_id,
                            uint16_t *out, int max);

/* The arrival pattern of the burst in flight: '#' for a fragment that
 * arrived, '.' for one that did not, one character per fragment of the
 * first part of the burst.  The number of characters written is
 * returned. */
int node_link_burst_pattern(node_link_t *nl, uint8_t node_id,
                            char *buf, int max);

/*
 * Ask the node to send the fragments of the burst in flight that never
 * arrived.  The request carries a bitmap of exactly the missing
 * fragments, so one request covers a scattered set of losses.
 *
 * Returns the number of fragments asked for, 0 if nothing was missing,
 * or -1 if the request could not be built or sent.
 */
int node_link_request_missing(node_link_t *nl, uint8_t node_id, uint16_t seq);

/*
 * Copy a node's reassembled window into `dst`, which must have room for
 * CRYNET_PROTO_WINDOW_SAMPLES int16 samples.  Returns false if the window
 * is not complete.
 */
bool node_link_take_window(node_link_t *nl, uint8_t node_id,
                           int16_t *dst, size_t dst_samples);

/*
 * Add any node whose address was learned since the last call.  Must be
 * called regularly from an application task — esp_now_add_peer() waits for
 * the Wi-Fi task, so it must never be called from the receive callback.
 */
void node_link_service_peers(node_link_t *nl);

/* Diagnostic counters from the promiscuous receive callback. */
void node_link_sweep_channel(uint8_t ch);

uint32_t node_link_promisc_total(void);
uint32_t node_link_promisc_magic(void);

/* Forget a node's partial burst, e.g. after it timed out. */
void node_link_reset_burst(node_link_t *nl, uint8_t node_id);

void node_link_get_stats(const node_link_t *nl, node_link_stats_t *out);


#ifdef __cplusplus
}
#endif

#endif /* NODE_LINK_H */
```

firmware/hub_infer/components/node_link/node_link.c

1152 lines · 42,766 bytes

```

/*
 * CryNet — the hub's half of the node link.  See node_link.h.
 */

#include "node_link.h"

#include <stdlib.h>
#include <string.h>

#include "freertos/FreeRTOS.h"
#include "freertos/semphr.h"
#include "freertos/task.h"
```

```
#include "esp_event.h"
#include "esp_heap_caps.h"
#include "esp_log.h"
#include "esp_netif.h"
#include "esp_now.h"
#include "esp_wifi.h"
#include "esp_mac.h"

static const char *TAG = "node_link";

/* The radio channel the nodes use.  Both sides set it explicitly, because
 * ESP-NOW does not roam: a node on channel 1 and a hub on channel 6 hear
 * nothing of each other. */
#define NODE_LINK_CHANNEL 6

#define NODE_LINK_MAX_FRAME 260

typedef struct {
    bool     used;
    uint8_t  id;
    uint8_t  src_mac[6];

    /* telemetry */
    uint16_t burst_seq;
    float    rms;
    uint16_t window_ms;
    uint8_t  flags;
    float    event_rms;
    float    frozen_rms;
    float    floor_rms;
    uint32_t frozen_crc;
    uint32_t last_seen_ms;
    bool     telemetry_new;

    /* burst reassembly */
    bool     in_burst;
    uint16_t frag_seq;
    uint16_t frag_want;
    uint16_t frag_have;
    /* How many fragment frames have arrived for this burst, counting
     * the ones that were already stored.  It is the difference between
     * "the frame never arrived" and "the frame arrived and was not
     * counted", which are two very different faults. */
    uint16_t frag_seen;
    uint32_t bad_payloads;
    /* When the last fragment of this burst arrived, and how long the
     * fragments have been taking to arrive.
     *
     * Both are measured here, in the receive callback, and not by the
     * main loop: the loop only notices that the fragment count has
     * changed when it next runs, so a gap it measures is the gap
     * between two of its own iterations -- which on a busy hub is
     * nearly two hundred milliseconds, and made the hub wait the
     * whole of that before it noticed a burst had stopped.  The
     * fragments of a burst cross a good link every six milliseconds,
     * and the hub has to know that to tell a burst that is still in
     * the air from one that has ended.
     *
     * The clock is this component's own millisecond clock, and what
     * the hub is given is the *age* of the last fragment, so that the
     * two clocks never have to be compared. */
    uint32_t last_frag_ms;    /* when the last fragment arrived */
    uint32_t gap_ewma_ms;     /* how this cot's fragments arrive */
    /* The signal strength of the burst's frames, and of the node's last
     * telemetry.  The two together say whether a burst that never
     * arrived was too weak to be heard or was never sent: a node whose
     * telemetry reads as strongly as its neighbours' and whose burst
     * frames read far weaker is not being heard at all, and one whose
     * frames read well was heard and something else went wrong. */
    int8_t   rssi_min;
    int8_t   rssi_max;
    int32_t  rssi_sum;
    uint16_t rssi_n;
    int8_t   tel_rssi;
    /* The node's own radio counters, from its telemetry. */
    uint32_t frames_sent;
    uint32_t frames_lost;
    uint32_t frames_acked;
    uint32_t radio_full;
    /* The sequence the hub asked for.  A fragment that carries any other
     * sequence belongs to an earlier window and is dropped: the radio
     * delivers frames in the order it wants to, and one straggler from
     * the last burst must not wipe the burst being assembled now. */
    uint16_t want_seq;
    bool     have_want;
```

```
    uint8_t  bitmap[(CRYNET_PROTO_FRAG_COUNT + 7u) / 8u];

    int16_t *window;
} node_slot_t;

struct node_link {
    bool           up;
    uint32_t       raw_shown;   /* how many raw frames have been printed */
    node_slot_t    slots[NODE_LINK_MAX_NODES];
    SemaphoreHandle_t lock;
    node_link_stats_t stats;
};

/* ESP-NOW has one global receive callback and no user-data pointer, so the
 * hub keeps a single instance and the callback finds it here.  There is one
 * hub. */
static node_link_t *s_instance = NULL;

static void link_lock(const node_link_t *nl)
{
    if (nl->lock != NULL) {
        xSemaphoreTake(nl->lock, portMAX_DELAY);
    }
}

static void link_unlock(const node_link_t *nl)
{
    if (nl->lock != NULL) {
        xSemaphoreGive(nl->lock);
    }
}

/* A millisecond clock.  The link needs nothing finer: telemetry arrives
 * every 100 ms and a burst is measured in seconds.  The FreeRTOS tick is
 * used rather than esp_timer so that this component does not have to
 * depend on it. */
static uint32_t now_ms(void)
{
    return (uint32_t)xTaskGetTickCount() * portTICK_PERIOD_MS;
}

/* The addresses of the peers the hub has heard from.  Each node's address
 * is learned from the frames it sends and then added to the ESP-NOW peer
 * list, so the radio's receive filter includes it.
 *
 * The addition is *deferred* to the application task: the receive callback
 * runs in the Wi-Fi task, and esp_now_add_peer() waits for the Wi-Fi task
 * to service it.  Called from inside that callback it blocks the very task
 * it is waiting for, and the radio stops delivering frames until something
 * else nudges it loose — the hub was receiving one frame in sixty. */
#define KNOWN_PEERS NODE_LINK_MAX_NODES
static uint8_t g_known[KNOWN_PEERS][6];
static int     g_known_count = 0;
static bool    g_pending[KNOWN_PEERS];

static void ensure_peer(const uint8_t *mac)
{
    for (int slot = 0; slot < KNOWN_PEERS; slot++) {
        if (g_known[slot][0] == mac[0] && g_known[slot][1] == mac[1] &&
            g_known[slot][2] == mac[2] && g_known[slot][3] == mac[3] &&
            g_known[slot][4] == mac[4] && g_known[slot][5] == mac[5]) {
            return;
        }
    }
    for (int slot = 0; slot < KNOWN_PEERS; slot++) {
        if (g_known[slot][0] != 0) {
            continue;   /* this slot is taken */
        }
        memcpy(g_known[slot], mac, 6);
        g_pending[slot] = true;
        return;
    }
}

/*
 * Add any peer whose address was learned since the last call.  Called from
 * the application task, where esp_now_add_peer() is safe.
 */
void node_link_service_peers(node_link_t *nl)
{
    if (nl == NULL) {
        return;
    }
    for (int slot = 0; slot < KNOWN_PEERS; slot++) {
        if (!g_pending[slot]) {
```

```
            continue;
        }
        esp_now_peer_info_t peer = {};
        memcpy(peer.peer_addr, g_known[slot], 6);
        peer.channel = NODE_LINK_CHANNEL;
        peer.ifidx = WIFI_IF_STA;
        peer.encrypt = false;
        if (esp_now_add_peer(&peer) == ESP_OK) {
            g_pending[slot] = false;
            ESP_LOGI(TAG, "peer %02X:%02X:%02X:%02X:%02X:%02X added",
                     g_known[slot][0], g_known[slot][1], g_known[slot][2],
                     g_known[slot][3], g_known[slot][4], g_known[slot][5]);
        }
    }
}

/* Diagnostic: promiscuous mode hands up every frame on the channel.  The
 * buffer is opaque, so the node's own magic (C3 7E) is looked for in it.
 * If the counter moves, the node's transmissions are on the air and the
 * ESP-NOW layer is losing them; if it does not, the node is not
 * transmitting them. */
static uint32_t s_bad_dbg = 0;   /* the first few rejects are printed, then quiet */
static volatile uint32_t s_promisc_total = 0;
static volatile uint32_t s_promisc_magic = 0;

#ifdef CRYNET_LINK_PROMISCUOUS
static void promisc_rx(void *buf, wifi_promiscuous_pkt_type_t type)
{
    (void)type;
    s_promisc_total++;
    /* The buffer is a wifi_promiscuous_pkt_t: the frame length is in
     * its rx_ctrl, and the bytes themselves are in its payload.  The
     * node's magic is looked for anywhere in the frame, because the
     * CryNet frame does not start at a fixed place inside the
     * 802.11 frame. */
    const wifi_promiscuous_pkt_t *pkt = (const wifi_promiscuous_pkt_t *)buf;
    const uint8_t *p = pkt->payload;
    int len = pkt->rx_ctrl.sig_len;
    if (len > (int)sizeof(pkt->payload)) {
        len = (int)sizeof(pkt->payload);
    }
    for (int i = 0; i + 1 < len; i++) {
        if (p[i] == 0xC3 && p[i + 1] == 0x7E) {
            s_promisc_magic++;
            return;
        }
    }
}
#endif

/* Diagnostic: move the radio to another channel and count what is there. */
void node_link_sweep_channel(uint8_t ch)
{
    esp_wifi_set_channel(ch, WIFI_SECOND_CHAN_NONE);
}

uint32_t node_link_promisc_total(void) { return s_promisc_total; }
uint32_t node_link_promisc_magic(void) { return s_promisc_magic; }

/* ------------------------------------------------------------ reception -- */

static void handle_telemetry(node_link_t *nl, uint8_t node_id, uint16_t seq,
                             const uint8_t *payload, size_t payload_len)
{
    crynet_proto_telemetry_t tel;
    if (!crynet_proto_decode_telemetry(payload, payload_len, &tel)) {
        return;
    }

    node_slot_t *slot = NULL;
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        if (nl->slots[i].used && nl->slots[i].id == node_id) {
            slot = &nl->slots[i];
            break;
        }
    }
    if (slot == NULL) {
        for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
            if (!nl->slots[i].used) {
                slot = &nl->slots[i];
                /* The window buffer was allocated for every slot at create;
                 * a plain memset here zeroed that pointer, and every
                 * fragment afterwards hit "window == NULL" and was counted
                 * bad — no burst ever reassembled. */
                void *win = slot->window;
```

```
                memset(slot, 0, sizeof(*slot));
                slot->window = win;
                slot->used = true;
                slot->id = node_id;
                ESP_LOGI(TAG, "node %u heard for the first time", node_id);
                break;
            }
        }
    }
    if (slot == NULL) {
        ESP_LOGW(TAG, "node %u ignored: the hub tracks %d nodes",
                 node_id, NODE_LINK_MAX_NODES);
        return;
    }

    slot->burst_seq = seq;
    slot->rms = tel.rms;
    slot->window_ms = tel.window_ms;
    slot->flags = tel.flags;
    slot->event_rms = tel.event_rms;
    slot->frozen_rms = tel.frozen_rms;
    slot->floor_rms = tel.floor_rms;
    slot->frozen_crc = tel.frozen_crc;
    slot->frames_sent = tel.frames_sent;
    slot->frames_lost = tel.frames_lost;
    slot->frames_acked = tel.frames_acked;
    slot->radio_full = tel.radio_full;
    slot->last_seen_ms = now_ms();
    slot->telemetry_new = true;
}

static void handle_fragment(node_link_t *nl, uint8_t node_id, uint16_t seq,
                            uint16_t frag, uint16_t frag_count,
                            const uint8_t *payload, int payload_len,
                            int rssi)
{
    node_slot_t *slot = NULL;
    nl->stats.frag_frames++;
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        if (nl->slots[i].used && nl->slots[i].id == node_id) {
            slot = &nl->slots[i];
            break;
        }
    }
    if (slot == NULL) {
        return;  /* fragments without telemetry are not ours */
    }

    if (frag_count != CRYNET_PROTO_FRAG_COUNT || frag >= CRYNET_PROTO_FRAG_COUNT) {
        if (s_bad_dbg < 8u) {
            s_bad_dbg++;
            ESP_LOGI(TAG, "frag reject: count %u frag %u (want %u)",
                     frag_count, frag, (unsigned)CRYNET_PROTO_FRAG_COUNT);
        }
        nl->stats.frames_bad++;
        return;
    }
    if (payload_len <= 0 || payload == NULL) {
        if (s_bad_dbg < 8u) {
            s_bad_dbg++;
            ESP_LOGI(TAG, "frag reject: payload_len %d", payload_len);
        }
        nl->stats.frames_bad++;
        return;
    }
    if (slot->window == NULL) {
        /* A slot without its window buffer: every fragment for it would
         * be dropped silently, which is exactly what happened once. */
        if (s_bad_dbg < 8u) {
            s_bad_dbg++;
            ESP_LOGE(TAG, "frag reject: node %u has no window buffer", node_id);
        }
        nl->stats.frames_bad++;
        return;
    }

    /* A fragment that belongs to another window is not ours to use. */
    if (slot->have_want && seq != slot->want_seq) {
        nl->stats.frames_bad++;
        return;
    }

    /* A new burst sequence starts a new reassembly. */
    if (!slot->in_burst || slot->frag_seq != seq) {
        if (slot->in_burst) {
```

```
            /* The sequence changed while a burst was in flight: the
             * fragments that arrived so far are discarded and the count
             * starts again.  This is worth a line of its own — it is
             * the difference between a burst that lost frames in the
             * air and one the hub counted wrongly. */
            ESP_LOGW(TAG, "node %u: the burst sequence changed from %u to %u "
                          "with %u of %u fragments in hand",
                     node_id, (unsigned)slot->frag_seq, (unsigned)seq,
                     (unsigned)slot->frag_have, (unsigned)slot->frag_want);
        }
        memset(slot->bitmap, 0, sizeof(slot->bitmap));
        slot->in_burst = true;
        slot->frag_seq = seq;
        slot->frag_want = frag_count;
        slot->frag_have = 0;
        slot->frag_seen = 0;
        slot->rssi_min = 0;
        slot->rssi_max = 0;
        slot->rssi_sum = 0;
        slot->rssi_n = 0;
    }
    /* It belongs to this burst and it arrived, whatever happens to it
     * next. */
    slot->frag_seen++;

    /* When it arrived, and how long the fragments have been taking.
     * Both are measured here, in the Wi-Fi task, where the arrival
     * actually happened: the hub's main loop only sees the count
     * change when it next runs, and a gap it measured that way was
     * the gap between two of its own iterations -- nearly two hundred
     * milliseconds on a busy hub, which is how a burst that had
     * stopped looked like a burst that was still in the air. */
    {
        const uint32_t now = now_ms();
        if (slot->last_frag_ms > 0) {
            const uint32_t gap = now - slot->last_frag_ms;
            /* A gap of more than half a second says nothing about the
             * rate: it is the pause between two bursts. */
            if (gap > 0 && gap < 500u) {
                slot->gap_ewma_ms = slot->gap_ewma_ms
                    ? (slot->gap_ewma_ms * 3 + gap) / 4
                    : gap;
            }
        }
        slot->last_frag_ms = now;
    }

    /* How loudly it arrived.  The frames of a burst are the biggest the
     * link carries, so they are the first thing to go when the path is
     * short of signal -- and the number says how much of the burst was
     * lost to the path rather than to anything the node did. */
    if (slot->rssi_n == 0u) {
        slot->rssi_min = (int8_t)rssi;
        slot->rssi_max = (int8_t)rssi;
    } else {
        if ((int8_t)rssi < slot->rssi_min) {
            slot->rssi_min = (int8_t)rssi;
        }
        if ((int8_t)rssi > slot->rssi_max) {
            slot->rssi_max = (int8_t)rssi;
        }
    }
    slot->rssi_sum += rssi;
    slot->rssi_n++;

    const uint8_t bit = (uint8_t)(frag & 7u);
    const uint8_t mask = (uint8_t)(1u << bit);
    uint8_t *byte = &slot->bitmap[frag >> 3];
    if (*byte & mask) {
        return;  /* a duplicate; the first copy was already stored */
    }
    *byte |= mask;

    /* The payload is the window's raw little-endian int16 bytes. */
    const size_t offset = (size_t)frag * CRYNET_PROTO_FRAG_PAYLOAD;
    const size_t room = CRYNET_PROTO_WINDOW_BYTES - offset;
    const size_t copy = ((size_t)payload_len < room) ? (size_t)payload_len : room;
    memcpy((uint8_t *)slot->window + offset, payload, copy);
    slot->frag_have++;

    ESP_LOGD(TAG, "node %u fragment %u/%u (seq %u)", node_id, frag + 1u,
             frag_count, seq);
}

static void on_recv(const esp_now_recv_info_t *info, const uint8_t *data, int len)
```

```
{
    node_link_t *nl = s_instance;
    if (nl == NULL || data == NULL || len <= 0) {
        return;
    }

    uint8_t type = 0, node_id = 0;
    uint16_t seq = 0, frag = 0, frag_count = 0;
    const uint8_t *payload = NULL;

    nl->stats.raw_frames++;
    if (nl->raw_shown < 5u) {
        nl->raw_shown++;
        ESP_LOGI(TAG, "raw frame %u bytes from %02X:%02X:%02X:%02X:%02X:%02X "
                      "(first bytes %02X %02X %02X %02X)",
                 (unsigned)len, info->src_addr[0], info->src_addr[1],
                 info->src_addr[2], info->src_addr[3], info->src_addr[4],
                 info->src_addr[5], data[0], len > 1 ? data[1] : 0,
                 len > 2 ? data[2] : 0, len > 3 ? data[3] : 0);
    }

    const int parsed = crynet_proto_parse(data, (size_t)len, &type, &node_id,
                                          &seq, &frag, &frag_count, &payload);
    if (parsed == CRYNET_PROTO_BAD_FRAME) {
        /* Print the first few rejects with the fields the parse checked,
         * so the failing check can be seen from the log. */
        if (s_bad_dbg < 5u && data[0] == CRYNET_PROTO_MAGIC0) {
            s_bad_dbg++;
            const uint16_t plen = (uint16_t)(data[10] | ((uint16_t)data[11] << 8));
            const uint16_t show = (plen > CRYNET_PROTO_MAX_PAYLOAD)
                                  ? CRYNET_PROTO_MAX_PAYLOAD : plen;
            uint16_t fcrc = 0;
            if (len >= (int)(CRYNET_PROTO_HEADER_BYTES + show + 2u)) {
                fcrc = (uint16_t)(data[CRYNET_PROTO_HEADER_BYTES + show] |
                                  ((uint16_t)data[CRYNET_PROTO_HEADER_BYTES + show + 1] << 8));
            }
            ESP_LOGI(TAG, "BAD frame: len %u type %u plen %u fcrc %04X computed %04X",
                     (unsigned)len, data[2], plen, fcrc,
                     (unsigned)crynet_proto_crc16(data,
                          CRYNET_PROTO_HEADER_BYTES + show));
        }
        nl->stats.frames_bad++;
        return;
    }
    if (parsed == CRYNET_PROTO_BAD_PAYLOAD) {
        nl->stats.payloads_bad++;
        return;
    }
    if (type == CRYNET_PROTO_TYPE_REQUEST) {
        return;  /* a request is the other direction; ignore it */
    }

    /* This dedicated hub monitors only cot IDs 1 and 2.  Reject other
     * IDs before they can occupy a slot or influence alerts. */
    if (node_id < 1 || node_id > NODE_LINK_MAX_NODES) {
        return;
    }

    nl->stats.frames_ok++;

    /* The node's address is learned from the frames it sends, and added
     * to the peer list.  The Wi-Fi driver's receive filter is built from
     * that list, and a node whose address is not in it is not heard. */
    if (info != NULL) {
        ensure_peer(info->src_addr);
    }

    link_lock(nl);
    if (type == CRYNET_PROTO_TYPE_TELEMETRY) {
        handle_telemetry(nl, node_id, seq, payload, (size_t)parsed);
        if (info != NULL) {
            for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
                if (nl->slots[i].used && nl->slots[i].id == node_id) {
                    memcpy(nl->slots[i].src_mac, info->src_addr, 6);
                    nl->slots[i].tel_rssi =
                        (info->rx_ctrl != NULL) ? info->rx_ctrl->rssi : 0;
                    break;
                }
            }
        }
    } else if (type == CRYNET_PROTO_TYPE_FRAGMENT) {
        handle_fragment(nl, node_id, seq, frag, frag_count, payload, parsed,
                        (info != NULL && info->rx_ctrl != NULL) ? info->rx_ctrl->rssi : 0);
    }
    link_unlock(nl);
```

```
}

/* ---------------------------------------------------------- lifecycle -- */

node_link_t *node_link_create(void)
{
    node_link_t *nl = calloc(1, sizeof(node_link_t));
    if (nl == NULL) {
        return NULL;
    }

    nl->lock = xSemaphoreCreateMutex();
    if (nl->lock == NULL) {
        free(nl);
        return NULL;
    }

    /* The window buffers are the bulk of the memory: 64 KB per node.  They
     * live in PSRAM, which the S3 has in 8 MB, and they are only touched
     * when a burst is being assembled. */
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        nl->slots[i].window = heap_caps_malloc(CRYNET_PROTO_WINDOW_BYTES,
                                               MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT);
        if (nl->slots[i].window == NULL) {
            nl->slots[i].window = heap_caps_malloc(CRYNET_PROTO_WINDOW_BYTES,
                                                   MALLOC_CAP_8BIT);
        }
        if (nl->slots[i].window == NULL) {
            ESP_LOGE(TAG, "no memory for a %u byte window buffer",
                     (unsigned)CRYNET_PROTO_WINDOW_BYTES);
            node_link_destroy(nl);
            return NULL;
        }
    }

    s_instance = nl;
    return nl;
}

void node_link_destroy(node_link_t *nl)
{
    if (nl == NULL) {
        return;
    }
    if (s_instance == nl) {
        s_instance = NULL;
    }
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        free(nl->slots[i].window);
    }
    if (nl->lock != NULL) {
        vSemaphoreDelete(nl->lock);
    }
    free(nl);
}

bool node_link_start(node_link_t *nl)
{
    if (nl == NULL) {
        return false;
    }

    if (esp_netif_init() != ESP_OK) {
        ESP_LOGE(TAG, "esp_netif_init failed");
        return false;
    }
    const esp_err_t ev = esp_event_loop_create_default();
    if (ev != ESP_OK && ev != ESP_ERR_INVALID_STATE) {
        ESP_LOGE(TAG, "event loop failed");
        return false;
    }

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    if (esp_wifi_init(&cfg) != ESP_OK) {
        ESP_LOGE(TAG, "esp_wifi_init failed");
        return false;
    }
    if (esp_wifi_set_storage(WIFI_STORAGE_RAM) != ESP_OK) {
        ESP_LOGE(TAG, "wifi storage failed");
        return false;
    }
    if (esp_wifi_set_mode(WIFI_MODE_STA) != ESP_OK) {
        ESP_LOGE(TAG, "wifi mode failed");
        return false;
    }
```

```
    /* The station interface has to exist *before* the radio starts.
     * esp_netif adds the interface to lwIP when the driver reports that
     * the station has started, so one created after that never joins the
     * stack: it keeps an MTU of zero, and lwIP then refuses to run DHCP
     * on it ("MTU is too small").  The symptom is a hub that joins the
     * Wi-Fi and never gets an address -- it uploads nothing and the
     * clock never starts -- while everything else looks healthy. */
    if (esp_netif_create_default_wifi_sta() == NULL) {
        ESP_LOGE(TAG, "cannot create the station interface");
        return false;
    }

    if (esp_wifi_start() != ESP_OK) {
        ESP_LOGE(TAG, "esp_wifi_start failed");
        return false;
    }

    /* Power save has to be turned off *after* the driver is started, and
     * this is not a detail: with the radio left to sleep between beacons
     * the hub received one node frame in six, because it was asleep for
     * the other fifty-nine out of every sixty.  The node's sketch calls
     * WiFi.setSleep(false) after WiFi.mode() for the same reason.  A
     * burst of 286 fragments would be lost almost entirely. */
    esp_wifi_set_ps(WIFI_PS_NONE);
    if (esp_wifi_set_channel(NODE_LINK_CHANNEL, WIFI_SECOND_CHAN_NONE) != ESP_OK) {
        ESP_LOGE(TAG, "channel %d failed", NODE_LINK_CHANNEL);
        return false;
    }

    if (esp_now_init() != ESP_OK) {
        ESP_LOGE(TAG, "esp_now_init failed");
        return false;
    }

    esp_now_peer_info_t peer = {};
    static const uint8_t bcast[6] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
    memcpy(peer.peer_addr, bcast, 6);
    /* The channel is set explicitly rather than left at 0.  A peer with
     * channel 0 is on "the current channel", but the driver's receive
     * filter is built from the peer list, and an explicit channel is
     * what it needs to listen on it. */
    peer.channel = NODE_LINK_CHANNEL;
    peer.ifidx = WIFI_IF_STA;
    peer.encrypt = false;
    if (esp_now_add_peer(&peer) != ESP_OK) {
        ESP_LOGE(TAG, "esp_now_add_peer failed");
        return false;
    }

    /* The nodes' addresses are configured here rather than discovered.
     *
     * The Wi-Fi driver's receive filter is built from the peer list, and a
     * frame from an address that is not in it is not delivered to the
     * ESP-NOW callback.  A node cannot be discovered by the frames it
     * sends, because those frames are exactly the ones being filtered —
     * the hub was hearing one frame in sixty from a node whose address it
     * had never been told.  So the addresses go in the table below and
     * every node is a peer from the first frame.
     *
     * The addresses are printed by the node's "status" command. */
    static const uint8_t node_addresses[][6] = {
        {0xE0, 0x72, 0xA1, 0x72, 0xAC, 0x14},   /* node 1 */
    };
    for (size_t i = 0; i < sizeof(node_addresses) / 6; i++) {
        esp_now_peer_info_t np = {};
        memcpy(np.peer_addr, node_addresses[i], 6);
        np.channel = NODE_LINK_CHANNEL;
        np.ifidx = WIFI_IF_STA;
        np.encrypt = false;
        if (esp_now_add_peer(&np) == ESP_OK) {
            /* Remember it, so the address is not added a second time when
             * the first frame from it arrives. */
            if (g_known_count < KNOWN_PEERS) {
                memcpy(g_known[g_known_count], node_addresses[i], 6);
                g_known_count++;
            }
            ESP_LOGI(TAG, "node %u peer %02X:%02X:%02X:%02X:%02X:%02X added",
                     (unsigned)(i + 1), node_addresses[i][0], node_addresses[i][1],
                     node_addresses[i][2], node_addresses[i][3],
                     node_addresses[i][4], node_addresses[i][5]);
        } else {
            ESP_LOGE(TAG, "node %u peer could not be added", (unsigned)(i + 1));
        }
    }
```

```

    if (esp_now_register_recv_cb(on_recv) != ESP_OK) {
        ESP_LOGE(TAG, "esp_now_register_recv_cb failed");
        return false;
    }

    /* Promiscuous mode hands the host *every* frame on the channel, and
     * it is off by default: with the laptop's own Wi-Fi sitting on
     * channel 1, that traffic flooded the receive path and cost the hub
     * about fifteen fragments out of every 286-fragment burst.  The
     * counters below are the diagnostic it was used for; build with
     * -DCRYNET_LINK_PROMISCUOUS=1 to bring it back. */
#ifdef CRYNET_LINK_PROMISCUOUS
    /* The callback has to be given to the driver *before* promiscuous
     * mode is switched on: with the callback missing, promiscuous mode
     * reports success and the radio hands over nothing at all. */
    esp_wifi_set_promiscuous_rx_cb(promisc_rx);
    if (esp_wifi_set_promiscuous(true) == ESP_OK) {
        ESP_LOGW(TAG, "promiscuous mode is on (diagnostic)");
    } else {
        ESP_LOGW(TAG, "promiscuous mode could not be switched on");
    }
#else
    ESP_LOGI(TAG, "promiscuous mode is off (the air counters are idle)");
#endif

    uint32_t version = 0;
    if (esp_now_get_version(&version) == ESP_OK) {
        ESP_LOGI(TAG, "ESP-NOW protocol version %u", (unsigned)version);
    }

    uint8_t mac[6] = {0};
    if (esp_read_mac(mac, ESP_MAC_WIFI_STA) == ESP_OK) {
        ESP_LOGI(TAG, "the hub's STA address is %02X:%02X:%02X:%02X:%02X:%02X",
                 mac[0], mac[1], mac[2], mac[3], mac[4], mac[5]);
    }

    uint8_t channel = 0;
    wifi_second_chan_t second = WIFI_SECOND_CHAN_NONE;
    if (esp_wifi_get_channel(&channel, &second) == ESP_OK) {
        ESP_LOGI(TAG, "the radio is on channel %u", (unsigned)channel);
    }

    nl->up = true;
    ESP_LOGI(TAG, "ESP-NOW ready on channel %d", NODE_LINK_CHANNEL);
    return true;
}

bool node_link_ready(const node_link_t *nl)
{
    return nl != NULL && nl->up;
}

/* ------------------------------------------------------------ telemetry -- */

bool node_link_poll_telemetry(node_link_t *nl, node_link_telemetry_t *out)
{
    if (nl == NULL) {
        return false;
    }

    bool found = false;
    link_lock(nl);
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        node_slot_t *slot = &nl->slots[i];
        if (!slot->used || !slot->telemetry_new) {
            continue;
        }
        slot->telemetry_new = false;
        out->node_id = slot->id;
        out->burst_seq = slot->burst_seq;
        out->rms = slot->rms;
        out->window_ms = slot->window_ms;
        out->flags = slot->flags;
        out->event_rms = slot->event_rms;
        out->frozen_rms = slot->frozen_rms;
        out->floor_rms = slot->floor_rms;
        out->frozen_crc = slot->frozen_crc;
        out->frames_sent = slot->frames_sent;
        out->frames_lost = slot->frames_lost;
        out->frames_acked = slot->frames_acked;
        out->radio_full = slot->radio_full;
        out->age_ms = now_ms() - slot->last_seen_ms;
        out->tel_rssi = slot->tel_rssi;
```

```
        found = true;
        break;
    }
    link_unlock(nl);
    return found;
}

bool node_link_last_telemetry(const node_link_t *nl, uint8_t node_id,
                              node_link_telemetry_t *out)
{
    if (nl == NULL) {
        return false;
    }

    bool found = false;
    link_lock(nl);
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        const node_slot_t *slot = &nl->slots[i];
        if (!slot->used || slot->id != node_id) {
            continue;
        }
        out->node_id = slot->id;
        out->burst_seq = slot->burst_seq;
        out->rms = slot->rms;
        out->window_ms = slot->window_ms;
        out->flags = slot->flags;
        out->event_rms = slot->event_rms;
        out->frozen_rms = slot->frozen_rms;
        out->floor_rms = slot->floor_rms;
        out->frozen_crc = slot->frozen_crc;
        out->frames_sent = slot->frames_sent;
        out->frames_lost = slot->frames_lost;
        out->frames_acked = slot->frames_acked;
        out->radio_full = slot->radio_full;
        out->age_ms = now_ms() - slot->last_seen_ms;
        out->tel_rssi = slot->tel_rssi;
        found = true;
        break;
    }
    link_unlock(nl);
    return found;
}

/* ------------------------------------------------------------------ bursts -- */

bool node_link_send_heartbeat(node_link_t *nl)
{
    if (nl == NULL || !nl->up) {
        return false;
    }

    /* A heartbeat is the twelve-byte request layout with a type of its
     * own: it asks for nothing, and it exists so that the cot nodes can
     * see the hub's signal strength from the RSSI of a frame they
     * receive.  The node's own link bar is built from exactly that. */
    uint8_t frame[CRYNET_PROTO_REQUEST_BYTES];
    const size_t len = crynet_proto_build(frame, CRYNET_PROTO_TYPE_HEARTBEAT,
                                          0xFF, 0u, 0u, 0u, NULL, 0u);
    if (len == 0) {
        return false;
    }

    static const uint8_t bcast[6] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
    return esp_now_send(bcast, frame, len) == ESP_OK;
}

bool node_link_request_window(node_link_t *nl, uint8_t node_id, uint16_t burst_seq)
{
    if (nl == NULL || !nl->up) {
        return false;
    }

    uint8_t frame[CRYNET_PROTO_REQUEST_BYTES];
    const size_t len = crynet_proto_build(frame, CRYNET_PROTO_TYPE_REQUEST,
                                          node_id, burst_seq, 0u,
                                          CRYNET_PROTO_FRAG_COUNT, NULL, 0u);
    if (len == 0) {
        return false;
    }

    /* The old burst is stale, and it is cleared *before* the request goes
     * out: a fragment that arrives between the send and the clear would
     * otherwise be counted into the new burst and then wiped. */
    link_lock(nl);
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
```

```
        node_slot_t *slot = &nl->slots[i];
        if (slot->used && slot->id == node_id) {
            slot->in_burst = false;
            slot->frag_have = 0;
            slot->frag_seen = 0;
            slot->frag_want = 0;
            slot->frag_seq = burst_seq;
            slot->want_seq = burst_seq;
            slot->have_want = true;
            /* The arrival clock starts again with the new burst.  The
             * rate it keeps (gap_ewma_us) is a property of the path,
             * not of the burst, and it is what the hub measures the
             * next burst against. */
            slot->last_frag_ms = 0;
            memset(slot->bitmap, 0, sizeof(slot->bitmap));
        }
    }
    link_unlock(nl);

    static const uint8_t bcast[6] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
    if (esp_now_send(bcast, frame, len) != ESP_OK) {
        nl->stats.requests_failed++;
        return false;
    }
    nl->stats.requests_sent++;
    return true;
}

/*
 * Ask the node to send the fragments of a burst that never arrived.
 *
 * The link loses frames for a reason that is outside its control: the
 * hub's one radio also carries Bluetooth, and while it is retuned to a
 * BLE channel the Wi-Fi receiver hears nothing.  A burst of 286 frames
 * sent as fast as the radio allows therefore loses a handful of them,
 * and the hole is not where the request was lost or where the burst
 * ended -- it is scattered through the middle.
 *
 * So the hub says exactly which fragments it is missing, as a bitmap,
 * and the node sends only those.  The fragments that did arrive are not
 * sent again, and the ones that did not are covered.
 *
 * Returns the number of fragments asked for, 0 if there was nothing
 * missing, or -1 if the request could not be built or sent.
 */
int node_link_request_missing(node_link_t *nl, uint8_t node_id, uint16_t seq)
{
    if (nl == NULL || !nl->up) {
        return -1;
    }

    uint8_t bitmap[CRYNET_PROTO_RESEND_BITMAP_BYTES];
    memset(bitmap, 0, sizeof bitmap);

    int missing = 0;
    bool found = false;
    link_lock(nl);
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        node_slot_t *slot = &nl->slots[i];
        if (!slot->used || slot->id != node_id) {
            continue;
        }
        found = true;
        if (slot->in_burst && slot->frag_seq == seq) {
            for (uint16_t f = 0; f < slot->frag_want; f++) {
                if ((slot->bitmap[f >> 3] & (1u << (f & 7u))) == 0) {
                    bitmap[f >> 3] |= (uint8_t)(1u << (f & 7u));
                    missing++;
                }
            }
        }
        break;
    }
    link_unlock(nl);

    if (!found) {
        return -1;
    }
    if (missing == 0) {
        return 0;
    }

    uint8_t frame[CRYNET_PROTO_HEADER_BYTES + CRYNET_PROTO_RESEND_BITMAP_BYTES + 4u];
    const size_t len = crynet_proto_build(frame, CRYNET_PROTO_TYPE_RESEND,
                                          node_id, seq, 0u,
```

```
                                           CRYNET_PROTO_FRAG_COUNT,
                                           bitmap, (uint16_t)sizeof bitmap);
    if (len == 0) {
        return -1;
    }

    /* The request is sent three times.
     *
     * ESP-NOW broadcast has no acknowledgement: the sender is told the
     * frame left the queue, not that it arrived.  The hub's radio is
     * deaf whenever it is retuned to a Bluetooth channel, and the
     * resend is sent at exactly the moment the burst has just gone
     * quiet -- which is when the radio has been busiest.  One lost
     * request means the whole round is wasted and the burst times out
     * with the fragments still missing, so the request is repeated.
     * It is fifty-four bytes and it asks for a handful of frames, so
     * three copies cost less than one lost round. */
    static const uint8_t bcast[6] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
    bool sent = false;
    for (int attempt = 0; attempt < 3; attempt++) {
        if (attempt > 0) {
            vTaskDelay(pdMS_TO_TICKS(20));
        }
        if (esp_now_send(bcast, frame, len) != ESP_OK) {
            nl->stats.requests_failed++;
            continue;
        }
        sent = true;
    }
    if (!sent) {
        return -1;
    }
    nl->stats.requests_sent++;
    nl->stats.resends_sent++;
    /* The bitmap as it goes on the air, printed.  The node prints the
     * same bytes from its side, so the two logs can be compared
     * directly: if the hub says it asked for 274 fragments and the node
     * reads twelve bits out of the very same request, the disagreement
     * is between the two ends of the wire and this is where it is
     * visible. */
    ESP_LOGW(TAG, "resend request for node %u: %d fragments asked for, "
                  "bitmap %02X %02X %02X %02X %02X %02X",
             (unsigned)node_id, missing,
             bitmap[0], bitmap[1], bitmap[2], bitmap[3], bitmap[4],
             bitmap[5]);
    return missing;
}

bool node_link_burst_state(const node_link_t *nl, uint8_t node_id,
                           node_link_burst_t *out)
{
    if (nl == NULL) {
        return false;
    }

    bool found = false;
    link_lock(nl);
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        const node_slot_t *slot = &nl->slots[i];
        if (!slot->used || slot->id != node_id) {
            continue;
        }
        out->node_id = slot->id;
        out->burst_seq = slot->frag_seq;
        out->frag_want = slot->frag_want;
        out->frag_have = slot->frag_have;
        out->frag_seen = slot->frag_seen;
        out->last_frag_age_ms = (slot->last_frag_ms > 0)
                                    ? (now_ms() - slot->last_frag_ms) : 0u;
        out->gap_ewma_ms = slot->gap_ewma_ms;
        out->complete = slot->in_burst &&
                        (slot->frag_want > 0) &&
                        (slot->frag_have >= slot->frag_want);
        out->bad_payloads = slot->bad_payloads;
        out->missing = (slot->frag_want > slot->frag_have)
                           ? (slot->frag_want - slot->frag_have) : 0u;
        out->rssi_min = slot->rssi_min;
        out->rssi_max = slot->rssi_max;
        out->rssi_avg = (slot->rssi_n > 0u)
                            ? (int)(slot->rssi_sum / (int32_t)slot->rssi_n) : 0;
        out->tel_rssi = slot->tel_rssi;
        found = true;
        break;
    }
    link_unlock(nl);
```

```
    return found;
}

/* Which fragments of the burst in flight have not arrived.  The indices
 * are written into `out` (up to `max` of them) and the number written
 * is returned.  This is a diagnostic: a burst that times out is usually
 * missing a handful of frames, and *which* ones says whether they were
 * lost at the start of the burst, at the end, or scattered through it. */
int node_link_burst_missing(node_link_t *nl, uint8_t node_id,
                            uint16_t *out, int max)
{
    if (out == NULL || max <= 0) {
        return 0;
    }
    int found = 0;
    link_lock(nl);
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        const node_slot_t *slot = &nl->slots[i];
        if (!slot->used || slot->id != node_id || !slot->in_burst) {
            continue;
        }
        for (uint16_t f = 0; f < slot->frag_want && found < max; f++) {
            if ((slot->bitmap[f >> 3] & (1u << (f & 7u))) == 0) {
                out[found++] = f;
            }
        }
        break;
    }
    link_unlock(nl);
    return found;
}

/* The arrival pattern of the burst in flight: one character for each of
 * the first fragments, '#' for one that arrived and '.' for one that did
 * not.  A burst whose frames arrive as a solid run of '#' and then stop
 * was cut off part way through -- the node stopped sending, or the
 * frames stopped being heard -- while one that arrives dotted all over
 * was heard and lost, which is a different fault with a different cure. */
int node_link_burst_pattern(node_link_t *nl, uint8_t node_id,
                            char *buf, int max)
{
    if (buf == NULL || max <= 0) {
        return 0;
    }
    buf[0] = '\0';
    int n = 0;
    link_lock(nl);
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        const node_slot_t *slot = &nl->slots[i];
        if (!slot->used || slot->id != node_id || !slot->in_burst) {
            continue;
        }
        int lim = (slot->frag_want < 64) ? (int)slot->frag_want : 64;
        if (lim > max) {
            lim = max;
        }
        for (int f = 0; f < lim; f++) {
            buf[f] = (slot->bitmap[f >> 3] & (1u << (f & 7u))) ? '#' : '.';
        }
        n = lim;
        break;
    }
    link_unlock(nl);
    if (n < max) {
        buf[n] = '\0';
    }
    return n;
}

bool node_link_take_window(node_link_t *nl, uint8_t node_id,
                           int16_t *dst, size_t dst_samples)
{
    if (nl == NULL || dst == NULL || dst_samples < CRYNET_PROTO_WINDOW_SAMPLES) {
        return false;
    }

    bool ok = false;
    link_lock(nl);
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        node_slot_t *slot = &nl->slots[i];
        if (!slot->used || slot->id != node_id) {
            continue;
        }
        if (slot->in_burst && slot->frag_want == CRYNET_PROTO_FRAG_COUNT &&
            slot->frag_have >= CRYNET_PROTO_FRAG_COUNT && slot->window != NULL) {
```

```
            memcpy(dst, slot->window, CRYNET_PROTO_WINDOW_BYTES);
            ok = true;
        }
        break;
    }
    link_unlock(nl);
    return ok;
}

void node_link_reset_burst(node_link_t *nl, uint8_t node_id)
{
    if (nl == NULL) {
        return;
    }
    link_lock(nl);
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        node_slot_t *slot = &nl->slots[i];
        if (slot->used && slot->id == node_id) {
            slot->in_burst = false;
            slot->frag_have = 0;
            slot->frag_seen = 0;
            slot->frag_want = 0;
            memset(slot->bitmap, 0, sizeof(slot->bitmap));
        }
    }
    link_unlock(nl);
}

void node_link_get_stats(const node_link_t *nl, node_link_stats_t *out)
{
    if (nl == NULL || out == NULL) {
        return;
    }
    *out = nl->stats;
}
```

Hub - the decision logic

firmware/hub_infer/src/link_stage.h

228 lines · 11,284 bytes

```
/*
 * CryNet — the radio stage of the hub.
 *
 * Stage 2c's first artifact: the hub listens to the nodes, decides when
 * there is something worth classifying, pulls the two-second window over
 * ESP-NOW and runs it through the Stage 2a inference core.
 *
 * The decision rule is the one frozen at the start of the rebuild: the
 * hub asks for audio once the node's RMS reaches 0.03, and the *model*
 * decides whether that audio is a cry.  The older firmware pulled audio
 * at RMS 0.18, so a soft cry never reached the classifier at all.
 */

#ifndef LINK_STAGE_H
#define LINK_STAGE_H

#include "crynet_dsp.h"
#include "crynet_infer.h"

#ifdef __cplusplus
extern "C" {
#endif

/* The level at which the hub asks a node for its window.  Anything above
 * this is worth classifying; the model rejects the false triggers.
 *
 * This is now only the fallback: the node tracks the room's noise floor
 * and reports the level it follows from it, and the hub triggers on the
 * node's number.  A fixed level of 0.030 put the system's reach at a
 * constant — a cry a metre from the node, with a median RMS of 0.0070,
 * never once reached it — so the floor is now tracked where the audio
 * actually is.  This constant is what a node that reports no level of
 * its own gets. */
#define CRYNET_TRIGGER_RMS 0.030f

/* The confidence at which one window is enough to announce a cry.
 *
```

```
 * The model's answer for a window that is very nearly silent is not
 * reliable: measured on the real boards, the same quiet room produced
 * verdicts of 0.0000 and of 0.9766 for windows that differ by nothing a
 * person could hear, and the second of those is a false alarm.  The
 * model's output saturates at 0.9961 for a clear cry, so a window that
 * reaches this level is announcing itself, and it is announced at once
 * — the panel still shows the cry two and a half seconds after it
 * happened, provided the window carries real energy (see
 * CRYNET_ANNOUNCE_MIN_RMS below).  Anything between the model's own
 * threshold and this one is a maybe, and a maybe is not announced: it
 * is counted (see CRYNET_CRY_RUN in link_stage.c) and nothing more.
 *
 * The threshold is the saturation and not the model's own 0.95, because
 * a conversation in the room reaches 0.9727 (measured 2026-11-04) and
 * the hub's own boot probe scores 0.9805 on synthetic noise.  Nothing
 * below the saturation is safe to announce on its own. */
#define CRYNET_CLEAR_CRY 0.9900f

/* The rms below which no window may announce a cry at all, however
 * confident the model says it is.
 *
 * The adaptive trigger sends the model windows the old fixed 0.030
 * trigger never would, and the model is not a reliable judge of a
 * window that is nearly silent.  Measured on the real boards: the
 * quietest genuine cry that reached the model carried 0.0103 rms (the
 * phone cry at one metre), while quiet-room windows at 0.002-0.012 have
 * scored up to 0.9883.  This gate sits under the quietest genuine cry
 * with room to spare, so the one-metre cry still announces and a
 * near-silent window cannot. */
#define CRYNET_ANNOUNCE_MIN_RMS 0.006f

/* How many windows in a row have to say cry before a maybe is
 * announced.  Two: a cry lasts for seconds and its windows agree, while
 * a noise fluctuation that happens to look like a cry is not repeated
 * in the next window.
 *
 * That assumption failed in a quiet room: the trigger pins at its
 * minimum there, the hub judges near-silent window after near-silent
 * window, their verdicts scatter across 0.95-0.99 for no audible
 * reason, and two of them in a row confirmed each other into fourteen
 * false alarms in one evening.  A maybe therefore only counts toward
 * the run when its window carries real energy (CRYNET_MAYBE_MIN_RMS). */
#define CRYNET_CRY_RUN 2u

/* The rms a window needs before its maybe can count toward the
 * confirming run.  Every measured genuine cry announced saturated
 * (>= 0.99, the unmistakable path) — the run exists for a real cry that
 * scores just over the threshold, and that comes with sound behind it.
 * The measured false maybes all sat at 0.012 rms or under. */
#define CRYNET_MAYBE_MIN_RMS 0.020f

/* The frozen window's own rms at which the hub pulls it even though the
 * room has gone quiet again.
 *
 * The live gates -- the node's streak and the reading above its own
 * level -- judge the room as it is *now*.  A frozen window is evidence
 * of the moment it was frozen, and the stop-capped freeze deliberately
 * lands after the cry has stopped, so by the time the hub looks the
 * live reading has usually fallen back to the room's own level and
 * neither live gate says anything.  The node reports the level of the
 * window it is offering beside the live one, and that number is what
 * the model will be shown, so it is the number the pull should follow.
 *
 * 0.050 sits above every window the nodes freeze out of ordinary room
 * sound (measured on the bench, 2026-11-04: the loudest such window
 * was 0.036 rms, on node 2) and well under every window a phone cry at
 * 15 cm froze (0.177 and 0.184 on the same bench).  Quieter frozen
 * windows are still pulled the moment the live reading is loud, so a
 * cry at one metre is not lost by this -- it is pulled while it is
 * happening, as it always was. */
#define CRYNET_PULL_CRY_RMS 0.050f

/* The rms at which a window carries so much sound that the model's
 * score may be taken at face value.
 *
 * It used to mean "a window this loud may announce on the model's own
 * threshold (0.95), without waiting for the saturation (0.99)".  That
 * rule was a band-aid for a clipped recording, and it opened the way
 * for a conversation in the room to be announced as a cry. */
#define CRYNET_LOUD_ANNOUNCE_RMS 0.080f

/* The confidence a loud window needs to announce on its own.
 *
 * It is the saturation, 0.9900, which is the same number the quiet
 * path uses -- so today a cry is announced when the model saturates
```

```
 * and not before.
 *
 * The trade is measured, both ways:
 *
 *   - the cost of the saturation: on the corpus, 7 of the 53 cry
 *     episodes that the old rule caught peaked between 0.9688 and
 *     0.9883 and are now missed.  All seven are strong cries -- rms
 *     0.09 to 0.22, crest 4.3 to 7.9 -- and on the bench three
 *     genuine cries out of three scored 0.9961 and still announce.
 *
 *   - the cost of the old rule: in twenty minutes of a room with
 *     people talking, the model scored four windows between 0.9688 and
 *     0.9805 that were not cries, and the hub's own boot probe scores
 *     0.9805 on synthetic noise.  None of the four was loud enough
 *     (0.080 rms) for the old rule to let it through, but that is the
 *     one thing about them that was not measured: how the model
 *     scores a *loud* conversation is unknown, and the old rule would
 *     have announced it.
 *
 * So this number is the saturation, and it is the only thing between
 * a loud conversation and a false alarm at three in the morning.  Set
 * it to 0.9500 to get the old rule back -- 7 more episodes on the
 * corpus, and the way open again. */
#define CRYNET_LOUD_ANNOUNCE_P 0.9900f

/* The confidence the second window of a run has to reach before the
 * run may announce a cry.
 *
 * It is the saturation as well, which means that in practice the run
 * announces nothing: a window that saturates announces on its own.
 * The run is kept because it says how close the model came, and
 * because setting this to 0.9500 together with CRYNET_LOUD_ANNOUNCE_P
 * brings back the whole old rule -- the loud window's own threshold
 * and the two-window confirmation. */
#define CRYNET_CONFIRM_P 0.9900f

/* The peak a window reaches before it counts as clipped.
 *
 * The microphone's own output saturates first -- the INMP441 is a
 * fixed-gain part -- so a sound close enough to it arrives flat-topped
 * and the window cannot be trusted.  A clipped window is refused: it
 * may not announce a cry, and the operator is told why. */
#define CRYNET_CLIP_PEAK 31000

/* The peak-to-rms ratio (crest factor) a quiet window needs before its
 * saturated verdict may announce a cry.  A loud window (>= 0.020 rms)
 * announces on saturation alone; a quiet one must also have a cry's
 * shape, because the model has been known to saturate on flat hiss.
 * Measured on the real boards: every quiet window that scored highly
 * and was not a cry had a crest of 2.2-2.9 (room hiss is Gaussian, no
 * impulses), and the quietest genuine cry — the phone cry at one metre,
 * P(cry)=0.9961 at 0.0103 rms — had a crest of 5.6; the 15 cm cries
 * measured 6.6-6.9.  A loud bump has a high crest too, but the model
 * refuses bumps on its own (measured 0.0977), so this gate only does
 * work exactly where the model is least trustworthy. */
#define CRYNET_QUIET_CREST 3.5f

/* Bring the radio up: NVS, then ESP-NOW.  Call it before the Bluetooth
 * stack starts — NimBLE's pools come out of the same internal RAM the
 * Wi-Fi driver allocates its buffers from, and when BLE claimed its
 * memory first the Wi-Fi driver ran out mid-init and the link never
 * started.  Returns false if the radio could not be started. */
bool link_stage_radio_start(void);

/* Run the radio stage.  Never returns.  The radio must already be up. */
void link_stage_run(crynet_dsp_ctx_t *dsp, crynet_infer_t *infer);

/* Whether the hub is announcing a cry for a node.  A confirmed cry is
 * held for a few seconds, so this stays true across the gaps in a
 * recording.  The display and the BLE server read this. */
bool link_stage_crying(uint8_t node_id);

/* One cot, as the console shows it. */
typedef struct {
    uint8_t  node_id;
    float    rms;          /* its last reading */
    float    level;         /* the level it triggers at */
    float    frozen_rms;    /* the level of the window it has frozen */
    uint8_t  confidence;    /* the last confirmed confidence, in percent */
    bool     crying;        /* the hub is announcing a cry from it */
    bool     window_full;   /* it has a whole two-second window to give */
    bool     deaf;          /* its windows never arrive: it cannot announce */
    int8_t   rssi;          /* the signal strength of its telemetry, in dBm;
                             * 0 = no reading */
} link_stage_cot_t;
```

```
208
209 /* The cots the hub cannot get a window from.  A deaf cot's audio never
210  * reaches the model, so it can never announce a cry -- the caregiver
211  * has to be told rather than left to take silence for a quiet cot.
212  * Bit 0 is cot 1, bit 1 is cot 2, and so on. */
213 uint8_t link_stage_deaf_mask(void);
214
215 /* The cots the hub knows about.  Returns how many were written. */
216 int link_stage_cots(link_stage_cot_t *out, int max);
217
218 /* The console's test alert: publish a cry for that cot as if it had
219  * heard one, and take it back.  This is how the caregiver's phone and
220  * the wearable are checked without making a sound. */
221 void link_stage_test_alert(uint8_t cot_id);
222 void link_stage_test_clear(void);
223
224 #ifdef __cplusplus
225 }
226 #endif
227
228 #endif /* LINK_STAGE_H */
```

firmware/hub_infer/src/link_stage.c

1824 lines · 90,664 bytes

```
1  /*
2   * CryNet — the radio stage of the hub.  See link_stage.h.
3   */
4
5  #include "link_stage.h"
6  #include "ble_alert.h"
7  #include "hub_display.h"
8
9  #include <math.h>
10 #include <string.h>
11
12 #include "freertos/FreeRTOS.h"
13 #include "freertos/task.h"
14 #include "esp_heap_caps.h"
15 #include "esp_log.h"
16 #include "esp_timer.h"
17 #include "esp_now.h"
18 #include "nvs_flash.h"
19
20 #include "crynet_model.h"
21 #include "crynet_proto.h"
22 #include "crynet_log.h"
23 #include "crynet_net.h"
24 #include "node_link.h"
25
26 static const char *TAG = "link";
27
28 /* How long to wait for a burst before giving up on it.  The node sends
29  * 286 fragments with the radio's flow control in the way, so a couple of
30  * seconds is normal; fifteen is generous. */
31 #define BURST_TIMEOUT_MS 15000
32
33 /* A cot whose windows keep failing is left alone for a while.  A burst
34  * that cannot cross the air will not cross it by being asked harder, and
35  * the retries are the loudest thing on the frequency: they starve the
36  * cots that *can* be heard.  Measured on the bench, one cot at -70 dBm
37  * pushed 34,853 frames trying to deliver windows it never delivered,
38  * and the cots beside it lost fragments to the flood.
39  *
40  * A cot that has failed this many windows in a row is *deaf*: the hub
41  * cannot get a window from it, so the model never sees its audio and it
42  * can never announce a cry.  The caregiver has to be told, because a
43  * cot that cannot be heard is not a cot that is being watched. */
44 #define BURST_FAIL_LIMIT 3
45 #define BURST_BACKOFF_MS 60000
46
47 /* The pause after a window has been analysed.  The node used to need two
48  * seconds to refill its window after an upload -- it threw the live
49  * window away because the microphone was not read while the burst was on
50  * the air -- so asking again sooner was pointless.  The node now reads
51  * the microphone throughout the burst and keeps its window whole, so the
52  * wait is only for the next freeze to happen. */
53 #define RECOOLDOWN_MS 1500
54
55 /* How long a confirmed cry keeps being announced.  A cry lasts for
56  * seconds; the model's windows are two seconds apart, and a recording
```

```
 * played through a speaker has gaps between the cries, so the display
 * must not flicker back to "quiet" the moment one window comes back
 * negative. */
#define CRY_HOLD_MS 8000

/* The shortest a burst may go without a fragment before the hub asks
 * for the missing ones again.  It is a floor, not a fixed threshold:
 * the hub waits eight times the cot's own fragment spacing, which is
 * twenty-four milliseconds for a burst that crosses a good link in
 * three, and four hundred milliseconds for one that manages a fragment
 * every fifty.  The floor is what a good link gets.
 *
 * The threshold used to be fixed, at one and a half seconds, and then
 * at sixty milliseconds, and both were wrong for the same reason: one
 * number cannot describe two links.  A fixed second and a half made a
 * single lost fragment in three hundred cost the system a second and a
 * half of dead time, and a fixed sixty milliseconds called a slow
 * burst dead while it was still crossing the air -- which is what
 * happened at the cot whose link is poor, where the hub spent its
 * resend rounds before the node had finished its first pass. */
#define FRAG_STALL_MS 100

/* How long the burst must have been in flight before a stall may be
 * believed.  The node is asked for the window and then starts sending
 * it; the first fragments take a moment to arrive, and a stall
 * detector that fires before the node has begun would ask for the whole
 * window a second time for nothing.
 *
 * It was two and a half seconds, which is longer than a whole burst
 * takes now, so a stalled burst was never believed at all: the hub sat
 * out the whole fifteen-second timeout instead.  A burst crosses the
 * air in well under a second, so a quarter of a second is plenty to
 * tell "the node has started" from "the node has stopped". */
#define RESEND_MIN_FLIGHT_MS 250

/* How many times the hub may ask for the missing fragments of one
 * burst.  Each round costs a few frames of air and covers the handful
 * the last round still lost.  Three is enough in practice: the first
 * round asks for everything that is missing, the second for the handful
 * that round also lost, and the third for the one or two that the
 * second lost.
 *
 * It is now ten, for the cot whose link is poor.  Its bursts arrive at
 * a fraction of the rate a good link manages, so a round that recovers
 * a third of what is missing needs several rounds to finish, and three
 * rounds was the hub giving up while the node was still sending.  The
 * rounds are bounded by the burst's own timeout, so a round that cannot
 * help does not cost more than the time it takes to notice that
 * nothing arrived. */
#define RESEND_MAX_ROUNDS 10

/* How long the node takes to send one fragment.  The node used to wait
 * ten milliseconds between fragments, so a hundred fragments took a
 * second and a half, and the hub had to allow for that when it decided
 * a burst had stalled: asking again while the node was still sending
 * wastes a round and floods the air.  The fragments now go out as fast
 * as the radio carries them, so the margin is a few milliseconds each. */
#define FRAG_PER_FRAGMENT_MS 4

/* The longest a burst may go without a fragment before the hub calls
 * it stopped, whatever rate the cot is managing.  A burst that is
 * merely slow is caught by the cot's own fragment spacing -- eight
 * times it -- and this is the ceiling on that patience, so that a burst
 * which has really stopped is noticed and asked for again. */
#define FRAG_STALL_MS_MAX 1500

/* The longest the hub waits for the fragments of a resend round to
 * arrive before it asks for them a second time.  A round that is
 * answered slowly is not worth repeating while it is still in the air,
 * but a round that is never answered must not hold the burst open for
 * the whole of its timeout. */
#define FRAG_SETTLE_MS_MAX 3000

/* How many bursts are in the air.  While one is being received the
 * hub's single radio must not also be advertising: every advertisement
 * is a moment when the Wi-Fi receiver is deaf, and the burst loses
 * frames in the air and has to be asked for again.  The Bluetooth stack
 * is told to go quiet for the duration and to start again afterwards.
 *
 * It is also the guard on the pull itself: the hub asks for one
 * node's window at a time.  Three cots bursting together put three
 * times the frames on one channel, the channel saturates, and the
 * bursts starve each other -- measured 2026-11-03 with all three cots
 * loud at once: the nearest cot's burst completed 286/286 in 0.79 s
 * while the next lost up to 46 fragments and the farthest never got
```

```
 * past 113 of 286, and every pull churned from window to window
 * without ever finishing one.  One burst at a time crosses the air in
 * under a second, so each cot is still served well inside the few
 * seconds its frozen window is held. */
static int s_bursts_in_flight = 0;

/* How long the hub's own radio has been deaf.  While a burst is on the
 * air the Wi-Fi receiver cannot hear anything at all — not that burst's
 * own fragments, and not any other cot's telemetry — so a node that
 * says nothing for a few seconds may simply have been talking while
 * the hub was shouting.  That time is counted here, and it is not held
 * against the node. */
static int64_t s_busy_total_us = 0;  /* the deaf time so far */
static int64_t s_busy_since_us = 0;  /* when the current stretch began */
static bool    s_radio_deaf = false;

/* The deaf time so far, the stretch in progress included. */
static int64_t radio_deaf_now_us(int64_t now_us)
{
    return s_busy_total_us +
           (s_radio_deaf ? (now_us - s_busy_since_us) : 0);
}

static void burst_started(void)
{
    const int64_t now = esp_timer_get_time();
    if (!s_radio_deaf) {
        s_radio_deaf = true;
        s_busy_since_us = now;
    }
    s_bursts_in_flight++;
    ble_alert_set_radio_busy(true);
}

static void burst_finished(void)
{
    const int64_t now = esp_timer_get_time();
    if (s_bursts_in_flight > 0) {
        s_bursts_in_flight--;
        if (s_bursts_in_flight == 0 && s_radio_deaf) {
            s_busy_total_us += now - s_busy_since_us;
            s_radio_deaf = false;
        }
    }
    ble_alert_set_radio_busy(s_bursts_in_flight > 0);
}

typedef struct {
    bool     heard;
    int64_t  last_heard_us;  /* when telemetry last arrived */
    int64_t  deaf_at_heard_us;  /* and how deaf the hub's radio was then */
    uint8_t  node_id;
    uint16_t seq;
    uint32_t crc;
    uint8_t  flags;
    float    rms;
    bool     window_full;  /* the node says its live window holds 2 s */
    uint8_t  loud_streak;  /* consecutive telemetry readings above the
                            * trigger level */
    /* How many windows in a row have said cry, and the sequence of the
     * last one counted.  The same window is judged more than once when
     * the hub asks for it again, and it always gives the same answer,
     * so it must only count once. */
    uint8_t  cry_run;
    uint16_t cry_seq;
    /* The level the node triggers at.  The node owns the room's noise
     * floor, so it owns the level that follows from it, and the hub
     * triggers on the number the node reports.  A node that reports
     * none is given CRYNET_TRIGGER_RMS instead. */
    float    event_rms;
    float    frozen_rms;   /* the level of the window the node froze */
    float    floor_rms;    /* the node's noise floor */
    int8_t   rssi;         /* telemetry RSSI in dBm, 0 = unknown */
    /* The node's own radio counters, from its telemetry: what it handed
     * to the radio, what the radio refused, what actually left it, and
     * how often the radio had no room.  They are what tell a burst that
     * was lost in the air from one the node's radio would not carry. */
    uint32_t frames_sent;
    uint32_t frames_lost;
    uint32_t frames_acked;
    uint32_t radio_full;

    bool     requested;      /* a burst is in flight */
    int64_t  requested_at_us;
    uint16_t request_seq;
```

```
    uint32_t request_crc;
    uint32_t request_raw;    /* the hub's frame counter when it asked */
    uint32_t request_frags;  /* the hub's fragment counter when it asked */
    /* How many windows this cot has failed to deliver, one after the
     * other.  A window that never completes can never be judged, so a
     * cot that keeps failing can never announce a cry -- it is deaf,
     * and the caregiver has to be told. */
    uint8_t  burst_failures;
    bool     deaf;           /* the hub cannot get a window from it */
    int64_t  backoff_until_us; /* and it is not asked again before this */
    /* The node's radio counters when it was asked, so that a stalled
     * burst can say what the node did with the fragments it was sent. */
    uint32_t request_sent;
    uint32_t request_lost;
    uint32_t request_acked;
    uint16_t last_have;      /* the fragment count at the last poll */
    uint32_t gap_ewma_ms;    /* how this cot's fragments arrive */
    /* How far apart this cot's fragments arrive, as a running mean.
     * The hub's patience has to follow the cot's own rate: a burst that
     * crosses a good link in three milliseconds a fragment is not a
     * burst that has stopped when it goes quiet for a tenth of a
     * second, and a burst that is crossing a poor link at one fragment
     * a second is not dead when it goes quiet for a moment. */

    int64_t  last_resend_us; /* when the last resend went out */
    uint8_t  resend_rounds;  /* how many resends this burst has had */

    bool     cry_active;      /* the hub is announcing a cry */
    int64_t  cry_until_us;     /* and it does so until this moment */
    int64_t  cry_started_us;   /* when the current cry began */
    uint8_t  last_confidence;  /* the last confirmed confidence, in percent */
    bool     published_cry;    /* what the caregiver's phone was last told */
    /* The record being written for the cry that is happening now.  The
     * number is taken when the episode opens, so the wrist can show it
     * while the cry is still going, and the measurements are gathered
     * window by window until it ends.  Zero means no episode is open. */
    uint32_t log_seq;
    uint32_t log_last_seq;     /* and the last one, kept after it closes,
                                * so "the cry is over" names the cry that
                                * just ended */
    uint8_t  log_peak_conf;
    uint16_t log_peak_rms_milli;
    uint32_t log_rms_sum_milli;
    uint16_t log_windows;
    uint16_t log_window_seq;   /* the same window is judged twice; it is
                                * measured once */
    /* When this cot first became clearly louder than the named one.
     * The name only moves when that has been true for
     * COT_SWITCH_DWELL_MS, so a pair of microphones that cross over
     * cannot hand the caregiver's wrist back and forth. */
    int64_t  louder_since_us;

    int64_t  last_log_us;      /* when the telemetry was last printed */
    int64_t  last_request_us;   /* when the last window was asked for   */
    uint16_t last_flags;
    uint16_t last_seq;
    uint16_t last_completed_seq;   /* the seq of the last window the hub
                                    * actually received and judged; the
                                    * hub will not ask for the same
                                    * frozen window twice, since a second
                                    * ask only re-serves the snapshot
                                    * the hub already has and keeps the
                                    * node from freezing the next one */
} node_state_t;

/* A cot that has stopped answering is offline, and the panel has to say
 * so promptly: the caregiver unplugs a node to move it and reads the
 * panel for the change.  Radio has no disconnect message — the frames
 * just stop — so silence is the only signal there is.
 *
 * But silence is only evidence if the hub's radio was open to hear it.
 * A burst takes five or six seconds of airtime, and while it is on the
 * air the Wi-Fi receiver hears nothing at all — not that burst's own
 * fragments, and not any other cot's telemetry.  With three cots the
 * hub's radio is busy most of the time, and a plain two-second timeout
 * read those gaps as "the node is gone": the cards flickered offline
 * while every cot was perfectly alive.  So the clock that decides
 * whether a cot has gone runs on the time the radio was *open*, and the
 * hub's own deafness is discounted. */
#define NODE_LIVE_TIMEOUT_US (2LL * 1000000LL)

/* The one cap the discount cannot lift: a cot that has said nothing
 * for this long is gone, whatever the radio was doing.  It is generous
 * because with three cots talking the radio is open for only a part of
 * the time. */
```

```
#define NODE_LIVE_MAX_US (20LL * 1000000LL)

/* One entry per node the hub has heard from. */
static node_state_t state[NODE_LINK_MAX_NODES];

/* Whether each cot was on the panel the last time round, so that a
 * change can be announced instead of only shown. */
static bool s_was_live[NODE_LINK_MAX_NODES];

/* How often the hub hands the dashboard the list of who is here.  It is
 * far quicker than the hub's own once-a-minute post, so the picture the
 * dashboard has is never more than a few seconds old when it is sent. */
#define PRESENCE_EVERY_MS 2000
static int64_t s_presence_at_us = 0;

/* The dashboard's presence report must not take the air from under a
 * cot's burst: while a window is in flight the Wi-Fi receiver is deaf
 * to that burst's own fragments, and a report landing in the middle of
 * one costs fragments and a round of resends.  So the network stage
 * asks this before it posts, and waits while a burst is running.  The
 * network stage caps the wait, so this can delay a report a little but
 * never stop the reports. */
static bool radio_busy_for_presence(void)
{
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        if (state[i].heard && state[i].requested) {
            return true;
        }
    }
    return false;
}

/* The cry record.
 *
 * One record per episode, not one per window: a baby who cries for a
 * minute produces twenty-odd windows and one episode, and it is the
 * episode the caregiver wants to see -- when it started, how long it
 * went on, how loud it was.  So the episode opens a record, each window
 * adds its measurements to it, and the record is written when the cry
 * has been quiet long enough to be over.
 *
 * None of this can hold up the announcement.  Every call into the log
 * is best-effort: if the flash will not mount the number comes back
 * nought, these helpers do nothing, and the wrist is told exactly what
 * it is told today. */
static void cry_episode_open(node_state_t *st, uint8_t node_id)
{
    st->log_seq = crynet_log_reserve(CRYNET_LOG_CRY, node_id);
    st->log_last_seq = st->log_seq;
    st->log_peak_conf = 0;
    st->log_peak_rms_milli = 0;
    st->log_rms_sum_milli = 0;
    st->log_windows = 0;
    st->log_window_seq = 0xFFFFu;
}

static void cry_episode_note(node_state_t *st, float rms, float p_cry)
{
    if (st->log_seq == 0) {
        return;
    }
    if (st->log_windows > 0 && st->request_seq == st->log_window_seq) {
        return;
    }
    st->log_window_seq = st->request_seq;
    uint32_t milli = (uint32_t)(rms * 1000.0f + 0.5f);
    if (milli > 65535u) {
        milli = 65535u;
    }
    if (milli > st->log_peak_rms_milli) {
        st->log_peak_rms_milli = (uint16_t)milli;
    }
    st->log_rms_sum_milli += milli;
    if (st->log_windows < 65535u) {
        st->log_windows++;
    }
    const uint8_t conf = (uint8_t)(p_cry * 100.0f + 0.5f);
    if (conf > st->log_peak_conf) {
        st->log_peak_conf = conf;
    }
}

static void cry_episode_close(node_state_t *st)
{
    if (st->log_seq == 0) {
```

```
        return;
    }
    /* The hold is a display artefact: the cry ended CRY_HOLD_MS before
     * the hold expired, so the hold is taken back off.  A window covers
     * two seconds of audio, so two seconds is the honest floor for an
     * episode that was heard at all. */
    int64_t end_us = st->cry_until_us - (int64_t)CRY_HOLD_MS * 1000;
    if (end_us < st->cry_started_us) {
        end_us = st->cry_started_us;
    }
    uint32_t dur_ms = (uint32_t)((end_us - st->cry_started_us) / 1000);
    if (dur_ms < 2000u) {
        dur_ms = 2000u;
    }
    const uint16_t mean = st->log_windows
        ? (uint16_t)(st->log_rms_sum_milli / st->log_windows) : 0;
    const uint8_t windows = st->log_windows > 255u ? 255u
                                                   : (uint8_t)st->log_windows;
    crynet_log_commit(st->log_seq, dur_ms, st->log_peak_conf, windows,
                      st->log_peak_rms_milli, mean);
    st->log_seq = 0;
}

/* The record number a cot's alert should carry. */
static uint32_t cot_record(uint8_t node_id)
{
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        if (state[i].heard && state[i].node_id == node_id) {
            return state[i].log_last_seq;
        }
    }
    return 0;
}

/* The difference between two readings of one of the node's own radio
 * counters.  The counters restart when the node reboots, and subtracting
 * two unsigned numbers across a restart wraps around: the log once said
 * the node had handed over 4 294 963 123 frames.  A counter that has
 * gone backwards says nothing about this burst, so it counts as zero. */
static uint32_t counter_delta(uint32_t now_v, uint32_t base)
{
    return (now_v >= base) ? (now_v - base) : 0u;
}

/* A node is live while its telemetry is recent.  Everything the
 * caregiver sees — the cards, the banner, the NODE pill — reads this,
 * not the latched `heard` flag, or an unplugged cot would stay online
 * forever. */
static bool node_live(const node_state_t *n, int64_t now_us)
{
    if (!n->heard) {
        return false;
    }
    const int64_t age = now_us - n->last_heard_us;
    if (age < NODE_LIVE_TIMEOUT_US) {
        return true;   /* plainly alive */
    }
    if (age > NODE_LIVE_MAX_US) {
        return false;  /* gone, whatever the radio was doing */
    }
    /* Only the silence the hub could have heard through counts. */
    int64_t deaf = radio_deaf_now_us(now_us) - n->deaf_at_heard_us;
    if (deaf > age) {
        deaf = age;
    }
    return (age - deaf) < NODE_LIVE_TIMEOUT_US;
}

/* How long the Bluetooth advertising stays off after a cot is heard
 * getting loud.  The window arrives half a second later, and the radio
 * only has to be quiet *before* the burst starts, so this is generous. */
#define PREPARE_QUIET_MS 3000

/* When the radio may go back to advertising. */
static int64_t s_prepare_until_us = 0;

/* The cot the caregiver was last told about, and whether that cot is
 * crying.  The alert that reaches the wrist names *one* cot: the
 * loudest one, the same rule the panel uses.  When two cots hear one
 * cry the hub used to publish an alert for each of them, and the
 * wearable showed "cot 2", then "cot 1", then "cot 2" again for a
 * single physical cry — three buzzes for one baby. */
static uint8_t s_ble_cot = 0;

/* The cots the hub could not get a window from, as a mask.  Bit 0 is
```

```
 * cot 1.  The caregiver's phone and the wearable are told when this
 * changes, because a cot that cannot be heard is not being watched. */
static uint8_t s_deaf_mask = 0;
static bool    s_ble_cry = false;

/* What the last published state packet said, so the hub can tell when
 * the caregiver's view has gone stale.  0xFF is a cot count that cannot
 * happen, so the first cycle always publishes. */
static uint8_t  s_state_cots_published  = 0xFF;
static uint8_t  s_state_deaf_published  = 0xFF;
static int64_t  s_state_published_ms    = -1000;

/* The cot a console test alert is standing in for, 0 when there is
 * none.  The console (see console.c) sets it, and it is published as
 * if that cot had heard a cry -- which is how the caregiver's phone
 * and the wearable are checked without making a sound. */
static uint8_t s_test_cot = 0;
static uint32_t s_test_seq = 0;   /* its record in the cry log */

/* How much louder another cot has to be before the caregiver is told
 * about it instead of the cot that was already named. */
#define COT_SWITCH_MARGIN 1.5f

/* How long that has to be true before the name moves.
 *
 * Measured on the bench, 2026-11-04 21:16: three cots a hand's width
 * apart on one desk, one cry from a phone, and the name changed hands
 * twelve times in eleven seconds -- the wrist buzzed "cot 1, cot 3,
 * cot 1, cot 3" for a single physical cry.  The test that let it through
 * was on the *live* reading: a hundred milliseconds of a signal that
 * is a cry, and two microphones that close cross over every few
 * hundred milliseconds.  So a challenger has to be clearly louder on
 * the *window* it froze -- a number that holds for the seconds a
 * window lasts -- and has to stay so for this long before the
 * caregiver's wrist is told about a different cot. */
#define COT_SWITCH_DWELL_MS 1500

/* When the hub came up, for the uptime on the display. */
static int64_t s_boot_us = 0;

/* The window the hub analyses.  64 KB, so it lives in PSRAM. */
static int16_t *s_window = NULL;
static float    *s_mel = NULL;
static int8_t   *s_q = NULL;

/* The link object the radio stage works through: created by
 * link_stage_radio_start(), used by link_stage_run(). */
static node_link_t *s_nl = NULL;

/* The exact integer mean of a window, and its RMS after that mean has been
 * removed.  The node reports the same numbers over the telemetry, so the
 * two can be compared. */
static int32_t window_mean(const int16_t *pcm, uint32_t n)
{
    int64_t sum = 0;
    for (uint32_t i = 0; i < n; i++) {
        sum += (int64_t)pcm[i];
    }
    return (int32_t)(sum / (int64_t)n);
}

static float window_rms(const int16_t *pcm, uint32_t n, int32_t mean)
{
    int64_t acc = 0;
    for (uint32_t i = 0; i < n; i++) {
        const int32_t d = (int32_t)pcm[i] - mean;
        acc += (int64_t)d * d;
    }
    return (float)(sqrt((double)acc / (double)n) / 32768.0);
}

static int32_t window_peak(const int16_t *pcm, uint32_t n, int32_t mean)
{
    int32_t peak = 0;
    for (uint32_t i = 0; i < n; i++) {
        const int32_t d = (int32_t)pcm[i] - mean;
        if (d < 0 ? -d > peak : d > peak) {
            peak = (d < 0) ? -d : d;
        }
    }
    return peak;
}

/* Analyse one window and report what the model says about it. */
static void analyse(crynet_dsp_ctx_t *dsp, crynet_infer_t *infer,
```

```
                    node_state_t *st, const int16_t *pcm)
{
    const uint8_t node_id = st->node_id;
    const int32_t mean = window_mean(pcm, CRYNET_N_SAMPLES);
    const float rms = window_rms(pcm, CRYNET_N_SAMPLES, mean);
    const int32_t peak = window_peak(pcm, CRYNET_N_SAMPLES, mean);
    /* Crest factor: a cry carries impulsive voiced peaks; room hiss is
     * flat (see CRYNET_QUIET_CREST). */
    const float crest = (rms > 0.0f)
                            ? ((float)peak / (rms * 32768.0f))
                            : 0.0f;

    ESP_LOGI(TAG, "node %u: window rms %.6f  mean %d  peak %d  crest %.1f",
             node_id, (double)rms, (int)mean, (int)peak, (double)crest);

    /* A clipped window is not evidence.
     *
     * The INMP441 is a fixed-gain microphone: its own output saturates
     * before the node's 16-bit window does, so a sound close enough to
     * the microphone -- a hand on the desk, a toy, a baby crying right
     * at the mic -- arrives flat-topped and the harmonics the model
     * reads are gone.  Measured on the bench: node 2 clipped at peak
     * 31801 of 32767 on ordinary room sound, and a phone held 10 cm
     * from the microphone clipped the cry it was playing, which cost
     * that window its saturation (0.9805 instead of 0.9961) and made
     * the alert name the wrong cot for 42 seconds.
     *
     * There is no software cure: the microphone's output is already
     * flat when the node sees it.  So the window is refused outright,
     * the operator is told, and the clean cot's window announces the
     * cry instead. */
    const bool clipped = (peak >= CRYNET_CLIP_PEAK);
    if (clipped) {
        ESP_LOGW(TAG, "node %u: the window is CLIPPED (peak %d of 32767) — "
                      "the microphone is too close to the sound; this window "
                      "is not evidence and will not announce a cry",
                 node_id, (int)peak);
    }

    const int64_t t0 = esp_timer_get_time();
    crynet_dsp_features(dsp, pcm, s_mel);
    crynet_dsp_quantise(s_mel, s_q);
    const int64_t t1 = esp_timer_get_time();

    float p_no_cry = 0.0f, p_cry = 0.0f;
    if (!crynet_infer_run(infer, s_q, &p_no_cry, &p_cry)) {
        ESP_LOGE(TAG, "inference failed: %s", crynet_infer_error(infer));
        return;
    }
    const int64_t t2 = esp_timer_get_time();

    const bool silent = rms < CRYNET_SILENCE_RMS_GATE;
    const bool says_cry = (!silent) && (p_cry >= CRYNET_THRESHOLD);
    /* No window this quiet may announce a cry at all: the model's
     * verdict on near-silence is noise (see CRYNET_ANNOUNCE_MIN_RMS). */
    const bool audible = rms >= CRYNET_ANNOUNCE_MIN_RMS;
    /* Only a saturated window may announce.
     *
     * The loud window's own rule -- "a window this loud may announce
     * on the model's threshold (0.95) instead of waiting for the
     * saturation" -- was a band-aid for a clipped recording, and it
     * opened the door this rule now closes.
     *
     * The band-aid was for a measured case: a phone held 10 cm from
     * the microphone distorted the cry it played, the window clipped,
     * and the model scored the damaged window 0.9805 instead of
     * 0.9961, so the quiet path refused a real cry and the alert named
     * the wrong cot.  But the model does not only lose certainty on
     * clipped cries: on 2026-11-04, in a room with people talking, it
     * scored an ordinary conversation window 0.9727 (node 3, rms 0.058,
     * crest 3.5).  One step louder, that window was on the loud path's
     * threshold, and the loud path announces on one window alone -- a
     * false alarm, at 0.080 rms, from somebody talking in the next
     * room.  The hub's own boot probe scores 0.9805 on a window of
     * synthetic noise, so no level below the saturation is safe.
     *
     * So a cry is announced when the model saturates and not before.
     * A clipped window does not saturate, and is refused -- which is
     * what the loud path was invented to avoid, but it is refused on
     * the *clean* cot's window instead, where the same cry saturates.
     * The cost is measured: on the corpus, 8 of the 57 episodes that
     * the old rule detected peaked between 0.96 and 0.99 and are now
     * missed; on the bench, three genuine cries out of three scored
     * 0.9961 and all three still announce.  One number moves it back
     * when the model is retrained against loud speech. */
```

```
    const bool saturated = (p_cry >= CRYNET_CLEAR_CRY);
    /* A loud window carries enough energy to announce on its own
     * confidence; a quiet one must also have the cry's impulsive
     * shape, not flat hiss.  Both thresholds are the saturation today;
     * the loud one is a separate number so that the old, more sensitive
     * rule is one edit away (see CRYNET_LOUD_ANNOUNCE_P). */
    const bool unmistakable = audible && !clipped &&
        ((rms >= CRYNET_LOUD_ANNOUNCE_RMS)
             ? (p_cry >= CRYNET_LOUD_ANNOUNCE_P)  /* loud: the model's word */
             : (p_cry >= CRYNET_CLEAR_CRY)) &&    /* quiet: it must saturate */
        ((rms >= CRYNET_MAYBE_MIN_RMS) || (crest >= CRYNET_QUIET_CREST));

    /* The announcement rule.
     *
     * The model is not a reliable judge of a window that is very nearly
     * silent: on the real boards, the same quiet room has produced
     * verdicts of 0.0000 and of 0.9766 for windows that differ by
     * nothing a person could hear, and the second of those is a false
     * alarm.  So one window announces a cry on its own only when it is
     * unmistakable — the model's output saturates at 0.9961 for a clear
     * cry, and that window is announced at once if it carries real
     * energy and, when quiet, has the impulsive shape of a cry.
     *
     * A window between the model's threshold (0.95) and the saturation
     * (0.99) is a maybe, and a maybe is never announced on its own:
     * the loud path that used to let one out is gone (see the note
     * above), because a conversation in the next room reaches 0.9727
     * and one step louder would have been announced.  The run of
     * maybes is kept as a counter — it says how close the model came
     * — but it no longer announces anything, because a room that
     * produces a maybe produces the next one too.
     *
     * What is left of the confirmation rule is the level: a near-silent
     * window can neither announce nor be counted, and a real cry
     * saturates and announces alone. */
    bool confirmed = false;
    if (says_cry) {
        if (rms >= CRYNET_MAYBE_MIN_RMS && st->cry_seq != st->request_seq) {
            st->cry_seq = st->request_seq;
            if (st->cry_run < 255u) {
                st->cry_run++;
            }
        }
        /* The run is a counter now, not a licence: it records how many
         * windows in a row have said cry, and the announcement itself
         * is the unmistakable window's own verdict.  The second number
         * is here so that the whole old rule can be brought back with
         * one edit (see CRYNET_CONFIRM_P). */
        confirmed = unmistakable ||
            (st->cry_run >= CRYNET_CRY_RUN && rms >= CRYNET_MAYBE_MIN_RMS &&
             p_cry >= CRYNET_CONFIRM_P);
    } else {
        st->cry_run = 0;
    }

    /* A confirmed cry is held: the announcement outlives the window
     * that confirmed it, so the display does not flicker while the
     * recording's gaps pass by. */
    const int64_t now = esp_timer_get_time();
    if (confirmed) {
        if (!st->cry_active) {
            st->cry_started_us = now;
            cry_episode_open(st, node_id);
        }
        st->cry_active = true;
        st->cry_until_us = now + (CRY_HOLD_MS * 1000);
        st->last_confidence = (uint8_t)(p_cry * 100.0f + 0.5f);
    } else if (st->cry_active && now > st->cry_until_us) {
        cry_episode_close(st);
        st->cry_active = false;
    }
    if (st->cry_active) {
        cry_episode_note(st, rms, p_cry);
    }

    ESP_LOGI(TAG, "node %u: front end %.1f ms, inference %.1f ms, "
                  "P(no_cry)=%.4f P(cry)=%.4f",
             node_id, (double)(t1 - t0) / 1000.0, (double)(t2 - t1) / 1000.0,
             (double)p_no_cry, (double)p_cry);
    ESP_LOGI(TAG, "node %u: %s", node_id,
             st->cry_active ? (confirmed ? "ALERT — cry" : "cry (held)")
                            : (says_cry ? (rms < CRYNET_MAYBE_MIN_RMS
                                               ? "cry (too quiet to announce)"
                                               : "cry (unconfirmed)")
                                        : (silent ? "silent" : "no cry")));
```

```
}

/* Whether the hub is announcing a cry for a node.  The display and the
 * BLE server read this, so they show one state and not two. */
bool link_stage_crying(uint8_t node_id)
{
    /* A console test alert shows on the panel as well as on the wrist. */
    if (s_test_cot != 0 && node_id == s_test_cot) {
        return true;
    }
    const int64_t now = esp_timer_get_time();
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        if (state[i].heard && state[i].node_id == node_id) {
            if (state[i].cry_active && now > state[i].cry_until_us) {
                cry_episode_close(&state[i]);
                state[i].cry_active = false;
            }
            return state[i].cry_active;
        }
    }
    return false;
}

/* The console's window into the cots. */
void link_stage_test_alert(uint8_t cot_id)
{
    s_test_cot = cot_id;
    s_test_seq = crynet_log_write(CRYNET_LOG_TEST, cot_id, 0, 0, 0, 0, 0);
    ESP_LOGW(TAG, "the console is holding a TEST ALERT for cot %u",
             (unsigned)cot_id);
}

void link_stage_test_clear(void)
{
    s_test_cot = 0;
    ESP_LOGW(TAG, "the console has taken the test alert back");
}

int link_stage_cots(link_stage_cot_t *out, int max)
{
    int n = 0;
    for (int i = 0; i < NODE_LINK_MAX_NODES && n < max; i++) {
        if (!state[i].heard) {
            continue;
        }
        out[n].node_id = state[i].node_id;
        out[n].rms = state[i].rms;
        out[n].level = state[i].event_rms;
        out[n].frozen_rms = state[i].frozen_rms;
        out[n].crying = link_stage_crying(state[i].node_id);
        out[n].confidence = state[i].last_confidence;
        out[n].window_full = state[i].window_full;
        out[n].deaf = state[i].deaf;
        out[n].rssi = state[i].rssi;
        n++;
    }
    return n;
}

/* The cots the hub cannot get a window from.  See link_stage.h. */
uint8_t link_stage_deaf_mask(void)
{
    uint8_t mask = 0;
    for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
        const node_state_t *st = &state[i];
        if (!st->heard || !st->deaf) {
            continue;
        }
        if (st->node_id >= 1u && st->node_id <= 8u) {
            mask |= (uint8_t)(1u << (st->node_id - 1u));
        }
    }
    return mask;
}

bool link_stage_radio_start(void)
{
    s_boot_us = esp_timer_get_time();
    /* The Wi-Fi driver opens the NVS partition during esp_wifi_init(), so
     * it has to be initialised first. */
    const esp_err_t nv = nvs_flash_init();
    if (nv != ESP_OK && nv != ESP_ERR_NVS_NO_FREE_PAGES) {
        ESP_LOGE(TAG, "nvs_flash_init failed: %s", esp_err_to_name(nv));
        return false;
    }
```

```
822
823     s_nl = node_link_create();
824     if (s_nl == NULL) {
825         ESP_LOGE(TAG, "the link could not be created");
826         return false;
827     }
828     if (!node_link_start(s_nl)) {
829         ESP_LOGE(TAG, "the radio could not be started");
830         return false;
831     }
832
833     /* The presence report yields to a burst: hand the network stage the
834      * one function that knows whether a cot's window is in flight. */
835     crynet_net_set_radio_guard(radio_busy_for_presence);
836
837     ESP_LOGI(TAG, "listening on ESP-NOW; asking for audio at RMS >= %.3f "
838                   "(the level a node reports wins; this is the fallback)",
839              (double)CRYNET_TRIGGER_RMS);
840     return true;
841 }
842
843 void link_stage_run(crynet_dsp_ctx_t *dsp, crynet_infer_t *infer)
844 {
845     s_window = heap_caps_malloc(CRYNET_N_SAMPLES * sizeof(int16_t),
846                                 MALLOC_CAP_SPIRAM | MALLOC_CAP_8BIT);
847     if (s_window == NULL) {
848         s_window = heap_caps_malloc(CRYNET_N_SAMPLES * sizeof(int16_t),
849                                     MALLOC_CAP_8BIT);
850     }
851     s_mel = heap_caps_malloc(CRYNET_FEATURE_COUNT * sizeof(float), MALLOC_CAP_8BIT);
852     s_q = heap_caps_malloc(CRYNET_FEATURE_COUNT * sizeof(int8_t), MALLOC_CAP_8BIT);
853     if (s_window == NULL || s_mel == NULL || s_q == NULL) {
854         ESP_LOGE(TAG, "no memory for the analysis buffers");
855         return;
856     }
857
858     /* The radio was started by link_stage_radio_start(), before the
859      * Bluetooth stack took its share of the internal RAM. */
860     node_link_t *nl = s_nl;
861     if (nl == NULL) {
862         ESP_LOGE(TAG, "the radio is not running; "
863                       "call link_stage_radio_start() first");
864         return;
865     }
866
867     int64_t last_stats_us = 0;
868     int64_t last_probe_us = 0;
869     int64_t last_heartbeat_us = 0;
870
871     for (;;) {
872         /* ---------------- telemetry ---------------- */
873         node_link_telemetry_t tel;
874         while (node_link_poll_telemetry(nl, &tel)) {
875             node_state_t *st = NULL;
876             for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
877                 if (!state[i].heard) {
878                     state[i].heard = true;
879                     state[i].node_id = tel.node_id;
880                     st = &state[i];
881                     break;
882                 }
883                 if (state[i].node_id == tel.node_id) {
884                     st = &state[i];
885                     break;
886                 }
887             }
888             if (st == NULL) {
889                 continue;
890             }
891             const int64_t heard_at = esp_timer_get_time();
892             st->last_heard_us = heard_at;
893             st->deaf_at_heard_us = radio_deaf_now_us(heard_at);
894
895             st->seq = tel.burst_seq;
896             st->crc = tel.frozen_crc;
897             st->flags = tel.flags;
898             st->rms = tel.rms;
899             st->window_full = (tel.flags & CRYNET_PROTO_FLAG_WINDOW_FULL) != 0;
900
901             /* The level to trigger at comes from the node.  The node
902              * tracks the room's noise floor and turns it into the level
903              * it freezes a window at, and the hub is asked to trigger on
904              * the same number -- the node is the one that freezes the
905              * window, so the level that matters is the one it used.  A
906              * node that reports none is an older node, and it gets the
```

```
             * fixed level. */
            st->event_rms = (tel.event_rms > 0.0f) ? tel.event_rms
                                                   : CRYNET_TRIGGER_RMS;
            st->frozen_rms = tel.frozen_rms;
            st->floor_rms = tel.floor_rms;

            st->rssi = tel.tel_rssi;

            /* The node's radio counters.  They are kept so that a
             * stalled burst can say whether the node handed the
             * fragments over and lost them in the air, or whether its
             * own radio would not carry them. */
            st->frames_sent = tel.frames_sent;
            st->frames_lost = tel.frames_lost;
            st->frames_acked = tel.frames_acked;
            st->radio_full = tel.radio_full;

            /* The node has moved on to a newer window.  The burst in
             * flight can never be completed -- the node no longer
             * holds the fragments of the old one, so it cannot send
             * them again -- and waiting out the timeout would only
             * delay the next attempt.  The burst is abandoned and the
             * newer window is asked for instead. */
            if (st->requested && st->seq != st->request_seq &&
                (tel.flags & CRYNET_PROTO_FLAG_BURST_READY) != 0) {
                ESP_LOGW(TAG, "node %u: the node moved to window %u while "
                              "window %u was in flight; asking for the new one",
                         tel.node_id, (unsigned)tel.burst_seq,
                         (unsigned)st->request_seq);
                node_link_reset_burst(nl, tel.node_id);
                st->requested = false;
                st->last_request_us = 0;
                burst_finished();
            }

            /* How long the level has been sustained.  A cry lasts for
             * seconds; the microphone's slow DC drift produces one lone
             * reading above the trigger, and a single reading is not
             * worth a 286-fragment burst. */
            if (tel.rms >= st->event_rms) {
                if (st->loud_streak < 255u) {
                    st->loud_streak++;
                }
            } else {
                st->loud_streak = 0;
            }

            const int64_t now = esp_timer_get_time();
            /* The cot is getting loud, so a window is half a second away.
             * The Bluetooth advertising stops now, while there is still
             * time for the radio to go quiet: the ESP32-S3 has one radio
             * for both, and an advertisement that slips through while
             * the burst is on the air deafens the Wi-Fi receiver for a
             * moment.  It was costing exactly one fragment of every
             * burst -- always the same one, always among the first
             * thirty, which is the advertising interval. */
            if (tel.rms >= st->event_rms &&
                (now + PREPARE_QUIET_MS * 1000) > s_prepare_until_us) {
                s_prepare_until_us = now + (PREPARE_QUIET_MS * 1000);
                ble_alert_set_preparing(true);
            }
            const bool changed = (tel.flags != st->last_flags) ||
                                 (tel.burst_seq != st->last_seq);
            if (changed || (now - st->last_log_us) > 1000000) {
                st->last_log_us = now;
                st->last_flags = tel.flags;
                st->last_seq = tel.burst_seq;
                /* Everything the hub's own decision depends on, in one
                 * line: the level the cot triggers at, how many
                 * readings in a row have been above it, and the level of
                 * the window it froze.  When a cry goes unheard, this
                 * line is where the answer is. */
                ESP_LOGI(TAG, "node %u: rms %.4f  seq %u  flags 0x%02X"
                         "  crc 0x%08X  level %.4f  streak %u  win %.4f",
                         tel.node_id, (double)tel.rms,
                         (unsigned)tel.burst_seq, (unsigned)tel.flags,
                         (unsigned)tel.frozen_crc, (double)st->event_rms,
                         (unsigned)st->loud_streak, (double)st->frozen_rms);
            }
        }

        /* ---------------- requests and bursts ---------------- */
        const int64_t now = esp_timer_get_time();

        /* The frozen window the next pull will ask for: the loudest
```

```
         * eligible one anywhere in the room, picked below and pulled
         * after the loop so only one burst is ever on the air. */
        node_state_t *pull = NULL;
        for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
            node_state_t *st = &state[i];
            if (!st->heard) {
                continue;
            }

            if (st->requested) {
                node_link_burst_t burst;
                if (node_link_burst_state(nl, st->node_id, &burst)) {
                    /* Watch the fragment count: a burst that stops
                     * growing has stopped, and that is the moment to ask
                     * for what is missing rather than to wait out the
                     * timeout. */
                    if (burst.frag_have != st->last_have) {
                        st->last_have = burst.frag_have;
                        /* The rate comes from the receive callback now,
                         * where the arrivals are actually seen; the
                         * loop's own view of the gap was the gap
                         * between two iterations of this loop, and on a
                         * busy hub that is nearly two hundred
                         * milliseconds -- so the hub waited all of that
                         * before it noticed the burst had ended. */
                        if (burst.gap_ewma_ms > 0) {
                            st->gap_ewma_ms = burst.gap_ewma_ms;
                        }
                    }
                    if (burst.complete) {
                        const int64_t t0 = esp_timer_get_time();
                        if (node_link_take_window(nl, st->node_id, s_window,
                                                  CRYNET_N_SAMPLES)) {
                            const uint32_t crc = crynet_proto_crc32(
                                (const uint8_t *)s_window, CRYNET_N_SAMPLES * 2u);
                            /* The CRC can only be checked when the node
                             * served the window the hub asked for.  If it
                             * moved on to a newer one, comparing the two
                             * would report a mismatch that means
                             * nothing, so the log says "not verified"
                             * instead. */
                            const bool checkable = (st->request_crc != 0) &&
                                                   (burst.burst_seq == st->request_seq);
                            const bool match = checkable && (crc == st->request_crc);
                            ESP_LOGI(TAG, "node %u: %u/%u fragments in %.2f s "
                                          "(signal %d dBm, from %d to %d; its "
                                          "telemetry reads %d dBm)",
                                     st->node_id, (unsigned)burst.frag_have,
                                     (unsigned)burst.frag_want,
                                     (double)(t0 - st->requested_at_us) / 1000000.0,
                                     (int)burst.rssi_avg, (int)burst.rssi_min,
                                     (int)burst.rssi_max, (int)burst.tel_rssi);
                            if (checkable) {
                                ESP_LOGI(TAG, "node %u: window crc32 0x%08X, the node "
                                              "advertised 0x%08X -> %s",
                                         st->node_id, (unsigned)crc,
                                         (unsigned)st->request_crc,
                                         match ? "MATCH" : "MISMATCH");
                            } else if (st->request_crc != 0) {
                                ESP_LOGI(TAG, "node %u: window crc32 0x%08X, not verified "
                                              "(the node served window %u, the hub "
                                              "asked for %u)",
                                         st->node_id, (unsigned)crc,
                                         (unsigned)burst.burst_seq,
                                         (unsigned)st->request_seq);
                            } else {
                                ESP_LOGI(TAG, "node %u: live window, crc32 0x%08X "
                                              "(the node had no frozen window to "
                                              "compare against)",
                                         st->node_id, (unsigned)crc);
                            }
                            analyse(dsp, infer, st, s_window);
                        st->last_completed_seq = st->request_seq;
                        /* The window arrived whole.  Whatever was wrong
                         * with the link is over, so the cot is heard
                         * again and the caregiver is told so. */
                        if (st->burst_failures != 0 || st->deaf) {
                            ESP_LOGI(TAG, "node %u: the link is working again "
                                          "after %u failed windows",
                                     st->node_id, (unsigned)st->burst_failures);
                        }
                        st->burst_failures = 0;
                        st->deaf = false;
                        st->backoff_until_us = 0;
                        }
```

```
                        st->requested = false;
                        st->last_log_us = esp_timer_get_time();
                        burst_finished();
                    } else {
                        /* The burst is short of fragments.  If it has
                         * stopped arriving, ask the node for the ones
                         * that are missing: the hub's radio also carries
                         * Bluetooth, and while it is retuned to a BLE
                         * channel the Wi-Fi receiver is deaf, so a
                         * burst sent flat out always loses a few frames
                         * in the air.  The node keeps the window it
                         * froze, so it can send them again. */
                        /* The wait before a burst counts as stalled
                         * grows with the number of fragments the node
                         * still has to send: a hundred missing
                         * fragments take a second and a half to arrive,
                         * and a burst that is quiet for less than that
                         * is a burst still in flight. */
                        const int still_missing =
                            (int)burst.frag_want - (int)burst.frag_have;
                        /* How long this cot's fragments take to arrive,
                         * as its own burst has shown it.  A burst that
                         * has not started yet has no rate of its own, so
                         * the estimate for a good link is used until it
                         * has one. */
                        int64_t per_fragment_ms = FRAG_PER_FRAGMENT_MS;
                        if (st->gap_ewma_ms > 0) {
                            per_fragment_ms = st->gap_ewma_ms;
                            if (per_fragment_ms < 1) {
                                per_fragment_ms = 1;
                            }
                        }
                        /* A burst has stopped when it has been quiet
                         * for sixteen times its own fragment spacing --
                         * a hundred milliseconds for a burst that
                         * crosses a good link in six milliseconds, and
                         * eight hundred for one that manages a fragment
                         * every fifty.  Sixteen, not eight: the node
                         * hands the radio a group of four frames at a
                         * time, so the gap between two arrivals is
                         * small inside a group and a pause between
                         * groups, and eight of the average was close
                         * enough to the pause to call a burst dead
                         * while it was still crossing the air. */
                        int64_t stall_ms = per_fragment_ms * 16;
                        if (stall_ms < FRAG_STALL_MS) {
                            stall_ms = FRAG_STALL_MS;
                        }
                        if (stall_ms > FRAG_STALL_MS_MAX) {
                            stall_ms = FRAG_STALL_MS_MAX;
                        }
                        const int64_t stall_us = stall_ms * 1000;
                        const bool stalled =
                            burst.last_frag_age_ms > (uint32_t)stall_ms;
                        const bool long_enough =
                            (now - st->requested_at_us) > (RESEND_MIN_FLIGHT_MS * 1000);
                        /* A second round waits out the fragments the
                         * last round asked for: they are on the air, and
                         * asking again before they arrive only spends a
                         * round.  How long they need is this cot's own
                         * fragment spacing times the number of them --
                         * a second for a good link, and the few seconds
                         * a poor one needs. */
                        int64_t settle_ms = (int64_t)still_missing *
                                            per_fragment_ms;
                        if (settle_ms > FRAG_SETTLE_MS_MAX) {
                            settle_ms = FRAG_SETTLE_MS_MAX;
                        }
                        if (settle_ms < stall_ms) {
                            settle_ms = stall_ms;
                        }
                        const bool settled =
                            (now - st->last_resend_us) > (settle_ms * 1000);
                        if (stalled && long_enough && settled &&
                            st->resend_rounds < RESEND_MAX_ROUNDS) {
                            /* Which fragments?  One in three hundred is
                             * being lost on the way, and the index says
                             * whether it is a frame that never left
                             * the node, a frame the hub's receiver
                             * dropped, or a frame the hub counted
                             * twice. */
                            uint16_t miss[8];
                            const int n_miss = node_link_burst_missing(
                                nl, st->node_id, miss,
                                (int)(sizeof miss / sizeof miss[0]));
```

```
                            if (n_miss > 0) {
                                ESP_LOGW(TAG, "node %u: missing fragments:",
                                         st->node_id);
                                for (int m = 0; m < n_miss; m++) {
                                    ESP_LOGW(TAG, "  %u of %u",
                                             (unsigned)miss[m],
                                             (unsigned)burst.frag_want);
                                }
                            }
                            const int asked = node_link_request_missing(
                                nl, st->node_id, st->request_seq);
                            if (asked > 0) {
                                st->resend_rounds++;
                                st->last_resend_us = now;
                                char pattern[72];
                                node_link_burst_pattern(nl, st->node_id,
                                                        pattern, sizeof pattern);
                                ESP_LOGW(TAG, "node %u: the burst stalled with "
                                              "%u of %u fragments (%u frames "
                                              "arrived); asking for the %u "
                                              "that are missing (signal %d dBm, "
                                              "from %d to %d; its telemetry "
                                              "reads %d dBm)",
                                         st->node_id, (unsigned)burst.frag_have,
                                         (unsigned)burst.frag_want,
                                         (unsigned)burst.frag_seen,
                                         (unsigned)asked,
                                         (int)burst.rssi_avg,
                                         (int)burst.rssi_min,
                                         (int)burst.rssi_max,
                                         (int)burst.tel_rssi);
                                /* What the node's own radio did with the
                                 * fragments it was sent.  A node that
                                 * handed them all over and saw few of
                                 * them come back has lost them in the
                                 * air; a node whose radio refused them
                                 * has a queue that is too small for the
                                 * burst.  The two faults are fixed in
                                 * opposite places, so the log says which
                                 * one it is. */
                                ESP_LOGW(TAG, "node %u: its radio handed over "
                                              "%u frames since the burst began, "
                                              "%u of them left it, %u were "
                                              "refused, %u times it had no room",
                                         st->node_id,
                                         (unsigned)counter_delta(st->frames_sent,
                                                                 st->request_sent),
                                         (unsigned)counter_delta(st->frames_acked,
                                                                 st->request_acked),
                                         (unsigned)counter_delta(st->frames_lost,
                                                                 st->request_lost),
                                         (unsigned)st->radio_full);
                                if (st->frames_sent < st->request_sent) {
                                    ESP_LOGW(TAG, "node %u: its counters went "
                                                  "backwards -- the node has "
                                                  "restarted since the burst "
                                                  "began, so the numbers above "
                                                  "count only what it has "
                                                  "handed over since",
                                             st->node_id);
                                }
                                ESP_LOGW(TAG, "node %u: arrival pattern: %s",
                                         st->node_id, pattern);
                            }
                        } else if ((now - st->requested_at_us) >
                                   (BURST_TIMEOUT_MS * 1000)) {
                        ESP_LOGW(TAG, "node %u: the burst timed out with %u of %u "
                                      "fragments",
                                 st->node_id, (unsigned)burst.frag_have,
                                 (unsigned)burst.frag_want);
                        {
                            node_link_stats_t rs;
                            node_link_get_stats(nl, &rs);
                            ESP_LOGW(TAG, "node %u: %u frames reached the hub "
                                          "during the burst, %u of them "
                                          "fragments",
                                     st->node_id,
                                     (unsigned)(rs.raw_frames - st->request_raw),
                                     (unsigned)(rs.frag_frames - st->request_frags));
                        }
                        /* Which ones are missing?  A burst that lost its
                         * first frames lost them to the request; one that
                         * lost its last frames never had them sent; a
                         * scattered set was lost in the air. */
                        uint16_t missing[16];
```

```
                        const int n_missing = node_link_burst_missing(
                            nl, st->node_id, missing,
                            (int)(sizeof missing / sizeof missing[0]));
                        if (n_missing > 0) {
                            ESP_LOGW(TAG, "node %u: missing fragments:",
                                     st->node_id);
                            for (int m = 0; m < n_missing; m++) {
                                ESP_LOGW(TAG, "  %u", (unsigned)missing[m]);
                            }
                        }
                        node_link_reset_burst(nl, st->node_id);
                        st->requested = false;
                        burst_finished();
                        /* The window did not arrive.  Count it, and
                         * after a few in a row stop asking: a burst
                         * that cannot cross the air does not cross it
                         * by being asked harder, and the retries take
                         * the airtime the other cots need. */
                        if (st->burst_failures < 255u) {
                            st->burst_failures++;
                        }
                        if (st->burst_failures >= BURST_FAIL_LIMIT) {
                            if (!st->deaf) {
                                st->deaf = true;
                                ESP_LOGW(TAG, "node %u: DEAF — %u windows in a "
                                              "row did not arrive; it cannot "
                                              "announce a cry until the link "
                                              "improves. Asking it again in %d s.",
                                         st->node_id,
                                         (unsigned)st->burst_failures,
                                         BURST_BACKOFF_MS / 1000);
                            }
                            st->backoff_until_us =
                                now + (BURST_BACKOFF_MS * 1000);
                        }
                        }
                    }
                }
                continue;
            }

            /* Nothing in flight.  Ask for a window when the node's own
             * level says there is something to hear and its window holds a
             * full two seconds. */
            /* Two conditions, and both have to hold.  The hub's own: the
             * node's RMS has reached the level above which audio is worth
             * classifying, and has stayed there for two readings (a cry
             * lasts for seconds; the microphone's DC drift produces one
             * lone reading).  The level is the node's own — it tracks the
             * room's noise floor and reports the level that follows from
             * it.  And the node's: it froze a window, which is the two
             * seconds centred on the moment it crossed that level, and
             * the one whose CRC-32 can be checked against what the node
             * advertises.
             *
             * A third, added on 2026-11-04: a window the node has
             * *frozen* is evidence of the moment it froze, not of the
             * moment the hub is looking at.  The stop-capped freeze
             * lands up to two seconds after the peak, and the hub only
             * polls the pull rule every so often, so by the time it
             * looks the live reading has often fallen back into the
             * room's own noise — and the loud window sat there
             * un-pulled.  Measured on the bench, the same minute:
             * node 1 advertised a window of 0.177 rms (a genuine cry)
             * for seven seconds, streak 0 and rms at the room's own
             * level the whole time, and the hub never asked for it.
             * The node also advertises the window's own level beside
             * the live reading — that is the number the model will
             * actually be shown — and a window that loud is worth
             * pulling no matter what the room is doing now. */
            const bool loud = (st->loud_streak >= 2u) ||
                              (st->rms >= 3.0f * st->event_rms) ||
                              (st->frozen_rms >= CRYNET_PULL_CRY_RMS);
            /* Only ask for a window the node has actually frozen.  The
             * node's live window is emptied by an upload and needs two
             * seconds to refill, and a request that arrives in that gap
             * is ignored — the burst times out with nothing received.
             * The node freezes at the same level the hub triggers at, so
             * a loud telemetry always carries this flag. */
            const bool frozen =
                (st->flags & CRYNET_PROTO_FLAG_BURST_READY) != 0;
            /* Only ask for a window worth the radio.  The node tells the
             * hub how loud the window it froze was, before the hub asks
             * for it, and a window quieter than the level the hub would
             * ever announce is one the hub would refuse as "too quiet
```

```
             * to announce" anyway.  Pulling it spends five or six
             * seconds of airtime — during which the hub's own receiver
             * is deaf and no other cot can be heard — for nothing.
             *
             * This matters because the cots' triggers sit just above
             * their own noise floor in a quiet room, so the cots freeze
             * a window every few seconds and the hub was pulling every
             * one of them.  With three cots the radio was deaf three
             * quarters of the time, the fetches stalled and were asked
             * for again, and a real cry at the third cot could not be
             * fetched in time to be classified.  The margin below the
             * gate keeps the hub honest about the level the model
             * sees, which is not exactly the node's own count. */
            const bool worth_pulling =
                (st->frozen_rms <= 0.0f) ||
                (st->frozen_rms >= 0.8f * CRYNET_MAYBE_MIN_RMS);
            /* One burst on the air at a time: a pull only starts while
             * no burst is in flight, and when several cots have frozen
             * windows the loudest window is the one asked for first.
             * The loudest is also the most likely to be announced, so
             * spending the airtime on it first is the right order even
             * before fairness. */
            if (loud && frozen && worth_pulling && st->window_full &&
                (st->seq != st->last_completed_seq) &&
                (now - st->last_request_us) > (RECOOLDOWN_MS * 1000) &&
                (now >= st->backoff_until_us) &&
                s_bursts_in_flight == 0 &&
                (pull == NULL || st->frozen_rms > pull->frozen_rms)) {
                pull = st;
            }
        }

        /* The chosen window goes on the air, alone.  Nothing else may
         * share the channel with it: a burst that loses frames to
         * another cot's burst needs resend rounds to recover, and the
         * node refreezes a window every few seconds, so a pull that
         * drags past that abandons a nearly whole window and starts
         * over -- which is exactly the churn three concurrent bursts
         * were stuck in, and why nodes 2 and 3 never finished one. */
        if (pull != NULL) {
            node_state_t *st = pull;
            if (node_link_request_window(nl, st->node_id, st->seq)) {
                st->requested = true;
                burst_started();
                st->requested_at_us = now;
                st->request_seq = st->seq;
                st->request_crc = st->crc;
                {
                    /* How many frames the hub has seen, so the
                     * timeout can say how many arrived during the
                     * burst: if the number is the number of
                     * fragments, they reached the hub and the
                     * reassembly is at fault; if it is short, they
                     * were lost on the way. */
                    node_link_stats_t rs;
                    node_link_get_stats(nl, &rs);
                    st->request_raw = rs.raw_frames;
                    st->request_frags = rs.frag_frames;
                    st->request_sent = st->frames_sent;
                    st->request_lost = st->frames_lost;
                    st->request_acked = st->frames_acked;
                }
                st->last_request_us = now;
                st->last_have = 0;

                st->last_resend_us = 0;
                st->resend_rounds = 0;
                /* The node drops its live window when it serves a
                 * request and refills it over the next two seconds,
                 * so until it says the window is whole again there is
                 * nothing to ask for. */
                st->window_full = false;
                ESP_LOGI(TAG, "node %u: asking for the window (seq %u, "
                              "rms %.4f, level %.4f, win %.4f)",
                         st->node_id, (unsigned)st->seq, (double)st->rms,
                         (double)st->event_rms, (double)st->frozen_rms);
            } else {
                ESP_LOGW(TAG, "node %u: the request could not be sent",
                         st->node_id);
            }
        }

        /* ---------------- the caregiver's phone ---------------- */
        /* The advertising goes back on once the burst has come and
         * gone: the radio has to be quiet before a burst, not during
```

```
         * the whole quiet period afterwards. */
        if (s_prepare_until_us != 0 && now > s_prepare_until_us) {
            s_prepare_until_us = 0;
            ble_alert_set_preparing(false);
        }
        /* The per-cot state first: which cots are crying. */
        bool any_state_changed = false;
        for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
            node_state_t *st = &state[i];
            if (!st->heard) {
                continue;
            }
            const bool crying = link_stage_crying(st->node_id);
            if (crying != st->published_cry) {
                st->published_cry = crying;
                any_state_changed = true;
            }
        }

        /* The alert names *one* cot, and the same cot for the whole
         * episode.  Publishing an alert per cot meant that two cots
         * hearing one cry produced two buzzes and the wearable showed
         * "cot 2", then "cot 1", then "cot 2" again for a single
         * physical cry.  Naming the loudest cot is not enough on its
         * own: two cots in one room both hear the cry, and their
         * levels cross over from window to window, so "the loudest"
         * changed hands every few hundred milliseconds.  The cot that
         * was named keeps the alert until it stops crying, or until
         * another cot is clearly louder. */
        node_state_t *loudest = NULL;
        for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
            node_state_t *st = &state[i];
            if (!st->heard || !st->published_cry) {
                continue;
            }
            if (loudest == NULL || st->rms > loudest->rms) {
                loudest = st;
            }
        }
        if (s_ble_cry && loudest != NULL && loudest->node_id != s_ble_cot) {
            /* The named cot is still crying: it keeps the alert unless
             * the other one is clearly louder -- and "clearly louder"
             * is now judged on the *window* the challenger froze, and
             * has to hold for COT_SWITCH_DWELL_MS before the name
             * moves.  See the note on COT_SWITCH_DWELL_MS: the live
             * reading crosses over between two microphones a hand's
             * width apart every few hundred milliseconds, and the
             * caregiver's wrist was told twelve times about one cry.
             *
             * The named cot keeps the alert right up to that point, so
             * one cry is one buzz. */
            node_state_t *incumbent = NULL;
            for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
                node_state_t *st = &state[i];
                if (st->heard && st->node_id == s_ble_cot &&
                    st->published_cry) {
                    incumbent = st;
                    break;
                }
            }
            if (incumbent != NULL) {
                const bool clearly_louder =
                    loudest->frozen_rms >
                        incumbent->frozen_rms * COT_SWITCH_MARGIN;
                if (clearly_louder) {
                    if (loudest->louder_since_us == 0) {
                        loudest->louder_since_us = now;
                    }
                    if ((now - loudest->louder_since_us) <
                            (COT_SWITCH_DWELL_MS * 1000)) {
                        loudest = incumbent;
                    }
                    /* otherwise the challenger takes the alert over */
                } else {
                    loudest->louder_since_us = 0;
                    loudest = incumbent;
                }
            }
        }
        /* A test alert from the console stands in for the cots: it is
         * what the bench uses to check the caregiver's phone and the
         * wearable without making a sound, and it is held until the
         * console says otherwise. */
        if (s_test_cot != 0) {
            if (!s_ble_cry || s_ble_cot != s_test_cot) {
```

```
                ble_alert_publish(s_test_cot, true, 100, s_test_seq);
                s_ble_cot = s_test_cot;
                s_ble_cry = true;
                any_state_changed = true;
            }
        } else if (loudest != NULL) {
            if (!s_ble_cry || loudest->node_id != s_ble_cot) {
                ble_alert_publish(loudest->node_id, true,
                                  loudest->last_confidence,
                                  loudest->log_last_seq);
                s_ble_cot = loudest->node_id;
                s_ble_cry = true;
                any_state_changed = true;
                /* The name has moved: nobody is "clearly louder" now,
                 * so the dwell starts again from scratch if it is ever
                 * challenged. */
                for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
                    state[i].louder_since_us = 0;
                }
            }
        } else if (s_ble_cry) {
            /* The cry is over: the cot that was named is told so. */
            ble_alert_publish(s_ble_cot, false, 0, cot_record(s_ble_cot));
            s_ble_cry = false;
            s_ble_cot = 0;
            any_state_changed = true;
            for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
                state[i].louder_since_us = 0;
            }
        }

        /* A cot that has gone deaf is announced, and the announcement is
         * withdrawn when the link comes back.  A cot the hub cannot hear
         * is not a cot that is being watched, and silence from it would
         * read as "all quiet" on the wrist. */
        {
            const uint8_t deaf_now = link_stage_deaf_mask();
            if (deaf_now != s_deaf_mask) {
                const uint8_t deaf_new = (uint8_t)(deaf_now & ~s_deaf_mask);
                const uint8_t deaf_gone = (uint8_t)(s_deaf_mask & ~deaf_now);
                s_deaf_mask = deaf_now;
                any_state_changed = true;
                for (int j = 0; j < NODE_LINK_MAX_NODES; j++) {
                    if (!state[j].heard || state[j].node_id < 1u ||
                        state[j].node_id > 8u) {
                        continue;
                    }
                    const uint8_t bit =
                        (uint8_t)(1u << (state[j].node_id - 1u));
                    if (deaf_new & bit) {
                        /* A cot that has gone deaf will send no more
                         * windows, so its cry episode can never be seen
                         * to end.  It is closed here, or the record
                         * would never be written at all. */
                        cry_episode_close(&state[j]);
                        const uint32_t rec = crynet_log_write(
                            CRYNET_LOG_DEAF, state[j].node_id, 0, 0, 0, 0, 0);
                        ble_alert_publish_deaf(state[j].node_id, rec);
                    } else if (deaf_gone & bit) {
                        /* The link is back: the cot is watched again. */
                        const uint32_t rec = crynet_log_write(
                            CRYNET_LOG_HEARD, state[j].node_id, 0, 0, 0, 0, 0);
                        ble_alert_publish(state[j].node_id, false, 0, rec);
                    }
                }
            }
        }

        /* The state characteristic carries the whole system: which cots
         * are crying, how many the hub can hear at all, and which have
         * gone deaf.  The packet is eight bytes of arithmetic, so it is
         * rebuilt every cycle and published when anything in it has
         * moved -- and on a slow beat besides, so the value a wearable
         * reads is never more than a second old.
         *
         * It used to be published only when a cry started or stopped.
         * That meant a cot *joining* the system never reached the
         * caregiver at all: the wrist read "0 cots" and went on reading
         * it, while the hub could hear both cots perfectly well.  The
         * count of cots the hub can hear is part of the state for the
         * same reason the cry mask is: without it, a caregiver cannot
         * tell "nothing is wrong" from "nothing is being watched". */
        {
            ble_state_packet_t sp;
            memset(&sp, 0, sizeof sp);
```

```
            sp.deaf_mask = s_deaf_mask;
            for (int j = 0; j < NODE_LINK_MAX_NODES; j++) {
                if (!state[j].heard) {
                    continue;
                }
                sp.cots++;
                if (state[j].published_cry) {
                    if (state[j].node_id >= 1 && state[j].node_id <= 8) {
                        sp.crying_mask |= (uint8_t)(1u << (state[j].node_id - 1));
                    }
                    if (state[j].last_confidence > sp.confidence_pct) {
                        sp.confidence_pct = state[j].last_confidence;
                    }
                    if (state[j].cry_started_us > 0) {
                        const int64_t age = (esp_timer_get_time() -
                                             state[j].cry_started_us) / 1000000;
                        if ((uint32_t)age > sp.started_at_s) {
                            sp.started_at_s = (uint32_t)age;
                        }
                    }
                }
            }
            const int64_t beat_ms = esp_timer_get_time() / 1000;
            const bool changed = any_state_changed ||
                                 sp.cots != s_state_cots_published ||
                                 sp.deaf_mask != s_state_deaf_published;
            if (changed || (beat_ms - s_state_published_ms) >= 1000) {
                s_state_cots_published = sp.cots;
                s_state_deaf_published = sp.deaf_mask;
                s_state_published_ms  = beat_ms;
                ble_alert_publish_state(&sp);
            }
        }

        /* While no node is live, probe.  The probe asks for a single
         * fragment, which is enough to prove that the two boards can
         * talk and which way the traffic is failing.  A hub whose cots
         * were all unplugged goes back to probing once they time out,
         * instead of waiting forever beside a dark nursery. */
        const int64_t probe_now = esp_timer_get_time();
        bool any_heard = false;
        for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
            if (node_live(&state[i], probe_now)) {
                any_heard = true;
            }
        }
        /* Learned peers are added here rather than in the receive
         * callback: the callback runs in the Wi-Fi task and
         * esp_now_add_peer() waits for that task, which would block the
         * radio. */
        node_link_service_peers(nl);

        /* The heartbeat goes out every second.  It asks the nodes for
         * nothing; it is there so that each cot can measure the hub-side
         * signal strength from the RSSI of a frame it receives, which is
         * what its own link bar is built from. */
        if ((probe_now - last_heartbeat_us) > 1000000) {
            last_heartbeat_us = probe_now;
            node_link_send_heartbeat(nl);
        }

        if (!any_heard && (probe_now - last_probe_us) > 2000000) {
            last_probe_us = probe_now;
            uint8_t frame[CRYNET_PROTO_REQUEST_BYTES];
            const size_t flen = crynet_proto_build(frame,
                                                   CRYNET_PROTO_TYPE_REQUEST,
                                                   0xFF, 0u, 0u, 1u, NULL, 0u);
            static const uint8_t bcast[6] =
                {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};
            const esp_err_t err = esp_now_send(bcast, frame, flen);
            ESP_LOGI(TAG, "probe: sent to everybody, %s",
                     (err == ESP_OK) ? "ok" : esp_err_to_name(err));
        }

        /* ---------------- the display ---------------- */
        /* One snapshot per loop: the crying cot if there is one,
         * otherwise the loudest node the hub has heard from.
         *
         * When two cots hear the same cry — they are in the same room,
         * so they all hear it — the one that hears it *loudest* is the
         * one the baby is in.  So the display names the loudest crying
         * cot, not the first one the hub happens to have heard from. */
        const int64_t snap_now = esp_timer_get_time();
        node_state_t *show = NULL;
        /* The cot the caregiver was told about keeps the display as
```

```
         * well: two cots in one room cross over from window to
         * window, and a card that changes every few hundred
         * milliseconds is unreadable.  While an alert is up the
         * display names *the same cot the wrist does* -- a panel that
         * says "cot 3" while the bracelet says "cot 1" is worse than
         * no panel at all. */
        if (s_ble_cry) {
            for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
                if (state[i].node_id == s_ble_cot &&
                    node_live(&state[i], snap_now) &&
                    link_stage_crying(state[i].node_id)) {
                    show = &state[i];
                    break;
                }
            }
        }
        if (show == NULL) {
            /* No alert is up: the loudest crying cot, or the loudest
             * cot the hub has heard from. */
            for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
                if (node_live(&state[i], snap_now) &&
                    link_stage_crying(state[i].node_id)) {
                    if (show == NULL || state[i].rms > show->rms) {
                        show = &state[i];
                    }
                }
            }
        }
        if (show == NULL) {
            for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
                if (node_live(&state[i], snap_now) &&
                    (show == NULL || state[i].rms > show->rms)) {
                    show = &state[i];
                }
            }
        }
        hub_display_state_t ds;
        memset(&ds, 0, sizeof ds);
        ds.node_count = HUB_DISPLAY_MAX_NODES;
        ds.ble_ready = ble_alert_is_running();
        ds.ble_clients = (uint8_t)ble_alert_clients();
        ds.uptime_s = (uint32_t)((snap_now - s_boot_us) / 1000000);
        /* The wireless link.  The panel says whether the hub is on the
         * network, because the hub's own record is what the network
         * carries: a grey mark means the cries are sitting in the
         * hub's flash and nobody can see them yet. */
        {
            crynet_net_status_t netst;
            crynet_net_status(&netst);
            ds.wifi_up = netst.wifi_up;
        }
        if (show != NULL) {
            ds.linked = true;
            ds.crying = link_stage_crying(show->node_id);
            ds.cot_id = show->node_id;
            ds.confidence = show->last_confidence;
            ds.rms = show->rms;
            /* The trigger the cot sends audio over.  The panel draws it
             * as a notch on the level bar, which is what tells the
             * caregiver whether a cot is only being listened to or is
             * shipping its audio to the model. */
            ds.trigger = show->event_rms;
        }
        /* Every cot the hub has heard from, so the dashboard can show
         * them one by one.  The slot is the cot number, so the cards
         * are always in the same order. */
        for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
            if (!state[i].heard) {
                continue;
            }
            const int slot = (int)state[i].node_id - 1;
            if (slot < 0 || slot >= HUB_DISPLAY_MAX_NODES) {
                continue;
            }
            ds.nodes[slot].id = state[i].node_id;
            /* A cot that has gone quiet keeps its card — the caregiver
             * needs somewhere to look — but it reads OFFLINE, and it
             * stops driving the banner and the NODE pill. */
            const bool live = node_live(&state[i], snap_now);
            ds.nodes[slot].online = live;
            ds.nodes[slot].crying = live && link_stage_crying(state[i].node_id);
            ds.nodes[slot].rms = state[i].rms;
            ds.nodes[slot].trigger = state[i].event_rms;
            ds.nodes[slot].confidence = state[i].last_confidence;
            /* A cot whose windows never arrive is online but deaf: it
```

```
             * is transmitting, and the hub still cannot get its audio,
             * so it cannot announce a cry. */
            ds.nodes[slot].deaf = state[i].deaf;
            ds.nodes[slot].rssi = state[i].rssi;
            /* Say it when a cot appears or disappears, so the panel's
             * offline cards can be counted rather than squinted at. */
            if (live != s_was_live[i]) {
                s_was_live[i] = live;
                ESP_LOGI(TAG, "node %u is %s", state[i].node_id,
                         live ? "online" : "offline");
            }
        }
        hub_display_update(&ds);

        /* Tell the dashboard who is here.  The cots have no network of
         * their own -- they talk to the hub over its radio -- so the
         * hub is the only one that can answer for them, and it does so
         * every minute whether or not there is anything to record. */
        if (snap_now - s_presence_at_us >= PRESENCE_EVERY_MS * 1000) {
            s_presence_at_us = snap_now;
            crynet_net_node_t pres[NODE_LINK_MAX_NODES];
            uint8_t pres_n = 0;
            for (int i = 0; i < NODE_LINK_MAX_NODES; i++) {
                if (!state[i].heard) {
                    continue;
                }
                if (state[i].node_id < 1 ||
                    state[i].node_id > CRYNET_NET_MAX_NODES) {
                    continue;
                }
                pres[pres_n].cot = state[i].node_id;
                pres[pres_n].online = node_live(&state[i], snap_now);
                pres[pres_n].deaf = state[i].deaf;
                pres[pres_n].rssi = state[i].rssi;
                pres_n++;
            }
            crynet_net_set_presence(pres, pres_n);
        }

        const int64_t stats_now = esp_timer_get_time();
        if ((stats_now - last_stats_us) > 5000000) {
            last_stats_us = stats_now;
            node_link_stats_t st;
            node_link_get_stats(nl, &st);
            ESP_LOGI(TAG, "link: %u frames off the air, %u parsed, %u bad, "
                          "%u bad payload, %u requests sent, %u failed",
                     (unsigned)st.raw_frames, (unsigned)st.frames_ok,
                     (unsigned)st.frames_bad, (unsigned)st.payloads_bad,
                     (unsigned)st.requests_sent, (unsigned)st.requests_failed);
            /* The air counters only move when promiscuous mode is on,
             * which it is not by default; skip the line rather than
             * print two zeroes every five seconds. */
            if (node_link_promisc_total() > 0) {
                ESP_LOGI(TAG, "air: %u frames seen raw, %u carrying our magic",
                         (unsigned)node_link_promisc_total(),
                         (unsigned)node_link_promisc_magic());
            }

        }

        /* The loop polls the radio, asks for windows and shows the
         * panel.  At fifty milliseconds a decision could sit waiting for
         * the next turn of the loop -- a fifth of the time a cry now
         * takes to be answered, and there is nothing in the loop heavy
         * enough to want that much of the CPU. */
        vTaskDelay(pdMS_TO_TICKS(20));
    }
}
```

Hub - Bluetooth

firmware/hub_infer/components/ble_alert/include/ble_alert.h

116 lines · 5,037 bytes

```
/*
 * CryNet — the BLE alert server.
 *
 * The hub is a GATT server: the caregiver's phone connects to it and
```

```
 * subscribes to the alert characteristic.  The service and the
 * characteristic use the UUIDs the legacy firmware published, so the
 * wearable firmware that was written against them keeps working:
 *
 *   device name   INFANT_MONITOR_HUB
 *   service       19b10000-e8f2-537e-4f6c-d104768a1214
 *   alert         19b10001-e8f2-537e-4f6c-d104768a1214
 *
 * A second characteristic, 19b10002-..., carries the *state* of every cot
 * (which cot is crying, how confident, and when it started).  The legacy
 * protocol had only the alert, but a caregiver's phone wants to show the
 * current state when it opens, not only the changes.
 *
 * The alert value is the legacy's six-byte packet, unchanged:
 *
 *   cot id, alert type (1 = cry, 2 = cancel), confidence in percent,
 *   event id.
 */

#ifndef BLE_ALERT_H
#define BLE_ALERT_H

#include <stdbool.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

/* The alert packet, the same six bytes the legacy firmware sent. */
typedef struct __attribute__((packed)) {
    uint8_t  cot_id;
    uint8_t  alert_type;      /* 1 = a cry, 2 = the cry has stopped,
                               *   3 = that cot is deaf (its audio never
                               *   arrives, so it cannot announce a cry) */
    uint16_t confidence_pct;
    uint16_t event_id;
} ble_alert_packet_t;

/* The state of every cot, as the state characteristic reports it. */
typedef struct __attribute__((packed)) {
    uint8_t  cots;             /* how many cots the hub knows about */
    uint8_t  crying_mask;      /* bit 0 is cot 1, bit 1 is cot 2, ... */
    uint8_t  deaf_mask;        /* the cots the hub cannot hear; bit 0 is
                                * cot 1.  A deaf cot is not being watched,
                                * so the caregiver is told. */
    uint8_t  confidence_pct;   /* the loudest cry's confidence */
    uint32_t started_at_s;     /* when the current cry began */
} ble_state_packet_t;

/* Start the Bluetooth stack and begin advertising.  Returns false if the
 * stack could not be started; the hub carries on without it. */
bool ble_alert_start(void);

/* Publish a cot's state.  `crying` true raises the alert, false cancels
 * it.  Called when the state changes, not on every window.
 *
 * `record` is the cry log's number for this event, and it is what the
 * wearable prints as "event #N" -- the same number the dashboard holds,
 * so a caregiver who saw an alert at three in the morning can find that
 * exact cry afterwards.  The packet's field is sixteen bits, so the
 * number wraps at 65536 (about three years of nights at fifty cries a
 * night); the dashboard keeps the whole number.  Pass 0 when the log
 * could not be opened: the hub then counts its own events, as it always
 * did, and the wrist still shows a number that rises. */
void ble_alert_publish(uint8_t cot_id, bool crying, uint16_t confidence_pct,
                       uint32_t record);

/* Tell the caregiver that a cot has gone deaf: its windows never arrive
 * at the hub, so the model never sees its audio and it cannot announce a
 * cry.  Silence from a cot that is not being watched is the one thing a
 * monitor must not do.  `record` as above. */
void ble_alert_publish_deaf(uint8_t cot_id, uint32_t record);

/* Publish the whole system's state, for the state characteristic. */
void ble_alert_publish_state(const ble_state_packet_t *state);

/* How many clients are connected and subscribed. */
int ble_alert_clients(void);

/* Tell the Bluetooth stack that the Wi-Fi radio needs the air.
 *
 * The ESP32-S3 has one radio and it carries both Bluetooth and Wi-Fi:
 * while the Bluetooth stack transmits an advertisement the Wi-Fi
 * receiver is deaf, and the hub's window arrives as 286 ESP-NOW
```

```
 * fragments that must not be missed.  The hub calls this with true
 * while a burst is in flight and with false when it is done, and the
 * Bluetooth stack stops advertising until it is told to start again.
 *
 * A connected caregiver's phone already stops the advertising on its
 * own, so in the demo case the radio is free from the moment the phone
 * connects. */
void ble_alert_set_radio_busy(bool busy);

/* The Wi-Fi radio is about to need the air.
 *
 * The hub hears a cot getting loud half a second before the window
 * arrives, and the advertising stops there and then: an advertisement
 * that slips through while the burst is on the air deafens the Wi-Fi
 * receiver for a moment, because the ESP32-S3 has one radio for both
 * Bluetooth and Wi-Fi.  That was costing exactly one fragment of every
 * burst -- always one, always among the first thirty. */
void ble_alert_set_preparing(bool preparing);

/* Whether the Bluetooth stack came up at all. */
bool ble_alert_is_running(void);

#ifdef __cplusplus
}
#endif

#endif /* BLE_ALERT_H */
```

firmware/hub_infer/components/ble_alert/ble_alert.c

445 lines · 15,451 bytes

```
/*
 * CryNet — the BLE alert server (NimBLE).
 *
 * The structure follows the ESP-IDF's own NimBLE peripheral example; what
 * is CryNet's own is the service, the UUIDs (the legacy's) and the value
 * format (the legacy's six-byte alert packet).
 */

#include <string.h>

#include "esp_log.h"
#include "host/ble_hs.h"
#include "host/ble_uuid.h"
#include "host/util/util.h"
#include "nimble/nimble_port.h"
#include "nimble/nimble_port_freertos.h"
#include "services/gap/ble_svc_gap.h"
#include "services/gatt/ble_svc_gatt.h"

#include "ble_alert.h"

/* The host's NVS-backed store.  The store's own header declares the read
 * and write calls but not this one; the ESP-IDF's own test app declares
 * it the same way. */
void ble_store_config_init(void);

static const char *TAG = "ble_alert";

#define DEVICE_NAME "INFANT_MONITOR_HUB"

/* The legacy's UUIDs, written least-significant byte first, which is the
 * order BLE_UUID128_INIT takes them in. */
static const ble_uuid128_t s_svc_uuid =
    BLE_UUID128_INIT(0x14, 0x12, 0x8a, 0x76, 0x04, 0xd1, 0x6c, 0x4f,
                     0x7e, 0x53, 0xf2, 0xe8, 0x00, 0x00, 0xb1, 0x19);
static const ble_uuid128_t s_alert_uuid =
    BLE_UUID128_INIT(0x14, 0x12, 0x8a, 0x76, 0x04, 0xd1, 0x6c, 0x4f,
                     0x7e, 0x53, 0xf2, 0xe8, 0x01, 0x00, 0xb1, 0x19);
static const ble_uuid128_t s_state_uuid =
    BLE_UUID128_INIT(0x14, 0x12, 0x8a, 0x76, 0x04, 0xd1, 0x6c, 0x4f,
                     0x7e, 0x53, 0xf2, 0xe8, 0x02, 0x00, 0xb1, 0x19);

static ble_alert_packet_t   s_alert_value;
static ble_state_packet_t   s_state_value;
static uint16_t s_alert_handle;
static uint16_t s_state_handle;

static uint16_t s_conn_handle = BLE_HS_CONN_HANDLE_NONE;
static bool     s_running = false;
static bool     s_alert_notify = false;
```

```
static bool     s_state_notify = false;
/* The number the wrist shows.  It is the cry log's record number when
 * there is a log, so the wrist and the dashboard name the same cry.
 * This counter is only the fallback for a hub whose flash would not
 * mount: the caregiver still gets a number that rises. */
static uint16_t s_event_id = 0;

static uint16_t event_number(uint32_t record)
{
    if (record != 0) {
        return (uint16_t)(record & 0xFFFFu);
    }
    return ++s_event_id;
}

/* The advertising policy.
 *
 * The ESP32-S3 has one radio and it carries both Bluetooth and Wi-Fi.
 * While the Bluetooth stack is transmitting an advertisement the Wi-Fi
 * receiver is deaf, and the hub's window arrives as 286 ESP-NOW
 * fragments that it must not miss.  NimBLE's default advertising
 * interval is its fastest (0x0020 = 20 ms) and each advertisement goes
 * out on all three BLE channels - about 150 radio transmissions a
 * second, and every one of them is a moment when the burst loses a
 * frame.
 *
 * So the hub advertises as little as it can get away with:
 *
 *   - while a caregiver's phone is connected, not at all: nobody needs
 *     to *find* a hub that is already connected, and the notifications
 *     travel over the connection, not over advertising;
 *   - while a burst is in the air, *slowly*: the wearable is the thing
 *     that has to be found, and it used to be left waiting for a gap in
 *     the radio's traffic -- with three cots the radio was busy 89 % of
 *     the time, so the wearable could take a minute to attach.  An
 *     advertisement every 320 ms costs the burst a frame or two, and
 *     the resend path recovers them; silence cost the caregiver the
 *     device altogether;
 *   - otherwise, quickly: 100 ms, which the phone still finds in well
 *     under a second.
 */
#define ADV_INTERVAL       0x00A0  /* 160 units of 0.625 ms = 100 ms */
#define ADV_INTERVAL_SLOW  0x0200  /* 512 units of 0.625 ms = 320 ms */

static bool s_advertising = false;   /* the hub is advertising right now */
static bool s_connected = false;     /* a caregiver's phone is connected */
static bool s_radio_busy = false;    /* the Wi-Fi radio needs the air */
static bool s_preparing = false;    /* the Wi-Fi radio will need it */
/* The interval the advertising is running at, so a change of policy
 * restarts it with the new interval. */
static uint16_t s_adv_interval = ADV_INTERVAL;

static int gap_event(struct ble_gap_event *event, void *arg);
static void update_advertising(void);

/* ------------------------------------------------------------------ GATT -- */

static int gatt_access(uint16_t conn_handle, uint16_t attr_handle,
                       struct ble_gatt_access_ctxt *ctxt, void *arg)
{
    (void)conn_handle;
    (void)arg;

    switch (ctxt->op) {
    case BLE_GATT_ACCESS_OP_READ_CHR: {
        const uint8_t *src;
        uint16_t len;
        if (attr_handle == s_alert_handle) {
            src = (const uint8_t *)&s_alert_value;
            len = (uint16_t)sizeof s_alert_value;
        } else {
            src = (const uint8_t *)&s_state_value;
            len = (uint16_t)sizeof s_state_value;
        }
        int rc = os_mbuf_append(ctxt->om, src, len);
        return (rc == 0) ? 0 : BLE_ATT_ERR_INSUFFICIENT_RES;
    }
    default:
        return BLE_ATT_ERR_WRITE_NOT_PERMITTED;
    }
}

static const struct ble_gatt_svc_def s_svcs[] = {
    {
        .type = BLE_GATT_SVC_TYPE_PRIMARY,
```

```
        .uuid = &s_svc_uuid.u,
        .characteristics = (struct ble_gatt_chr_def[]){
            {
                .uuid = &s_alert_uuid.u,
                .access_cb = gatt_access,
                .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_NOTIFY |
                         BLE_GATT_CHR_F_INDICATE,
                .val_handle = &s_alert_handle,
            },
            {
                .uuid = &s_state_uuid.u,
                .access_cb = gatt_access,
                .flags = BLE_GATT_CHR_F_READ | BLE_GATT_CHR_F_NOTIFY,
                .val_handle = &s_state_handle,
            },
            { 0 },
        },
    },
    { 0 },
};

static void gatt_register_cb(struct ble_gatt_register_ctxt *ctxt, void *arg)
{
    (void)arg;
    ESP_LOGI(TAG, "the GATT server registered (operation %d)", (int)ctxt->op);
}

/* Register the service with the server.  The count has to be told to the
 * stack before the services are added: it is how the stack reserves the
 * attribute handles. */
static int gatt_svr_init(void)
{
    ble_svc_gatt_init();
    int rc = ble_gatts_count_cfg(s_svcs);
    if (rc != 0) {
        return rc;
    }
    return ble_gatts_add_svcs(s_svcs);
}

/* ------------------------------------------------------------------ GAP -- */

static bool advertise(void)
{
    struct ble_gap_adv_params adv_params;
    struct ble_hs_adv_fields fields;
    const char *name = DEVICE_NAME;
    int rc;

    memset(&fields, 0, sizeof fields);
    fields.flags = BLE_HS_ADV_F_DISC_GEN | BLE_HS_ADV_F_BREDR_UNSUP;
    fields.tx_pwr_lvl_is_present = 1;
    fields.tx_pwr_lvl = BLE_HS_ADV_TX_PWR_LVL_AUTO;
    fields.name = (uint8_t *)name;
    fields.name_len = (uint8_t)strlen(name);
    fields.name_is_complete = 1;

    rc = ble_gap_adv_set_fields(&fields);
    if (rc != 0) {
        ESP_LOGE(TAG, "the advertisement fields could not be set: %d", rc);
        return false;
    }

    memset(&adv_params, 0, sizeof adv_params);
    adv_params.conn_mode = BLE_GAP_CONN_MODE_UND;
    adv_params.disc_mode = BLE_GAP_DISC_MODE_GEN;
    /* The slow interval: the phone still finds the hub quickly, and the
     * radio is free for the burst three to five times as often.  NimBLE
     * would otherwise use its own default of 30-60 ms. */
    adv_params.itvl_min = s_adv_interval;
    adv_params.itvl_max = s_adv_interval;
    rc = ble_gap_adv_start(BLE_OWN_ADDR_PUBLIC, NULL, BLE_HS_FOREVER,
                           &adv_params, gap_event, NULL);
    if (rc != 0) {
        ESP_LOGE(TAG, "advertising could not be started: %d", rc);
        return false;
    }
    ESP_LOGI(TAG, "advertising as \"" DEVICE_NAME "\" every %u ms",
             (unsigned)(s_adv_interval * 5u / 8u));
    return true;
}

/* Bring the advertising into line with the policy: whenever nobody is
 * connected, but slowly while the Wi-Fi radio is carrying a burst. */
static void update_advertising(void)
```

```
{
    const bool wanted = !s_connected;
    const uint16_t interval =
        (s_radio_busy || s_preparing) ? ADV_INTERVAL_SLOW : ADV_INTERVAL;

    if (wanted == s_advertising && interval == s_adv_interval) {
        return;
    }
    if (wanted) {
        /* A different interval means the advertising is started again
         * with the new one: NimBLE takes the interval when the
         * advertising starts. */
        if (s_advertising) {
            int rc = ble_gap_adv_stop();
            if (rc != 0 && rc != BLE_HS_EALREADY) {
                ESP_LOGW(TAG, "advertising could not be stopped: %d", rc);
            }
            s_advertising = false;
        }
        s_adv_interval = interval;
        s_advertising = advertise();
    } else {
        /* Only say something when the advertising was actually running.
         * This function is called again every time the radio's policy
         * changes -- and the radio's policy changes every time a burst
         * starts or stops -- so an unconditional line here printed
         * "advertising is off (a phone is connected)" seven times in
         * five minutes of ordinary running, and read like the phone
         * was dropping off.  It was not: the phone stayed connected
         * the whole time. */
        if (s_advertising) {
            int rc = ble_gap_adv_stop();
            if (rc != 0 && rc != BLE_HS_EALREADY) {
                ESP_LOGW(TAG, "advertising could not be stopped: %d", rc);
            }
            s_advertising = false;
            ESP_LOGI(TAG, "advertising is off (a phone is connected)");
        }
    }
}

static int gap_event(struct ble_gap_event *event, void *arg)
{
    (void)arg;

    switch (event->type) {
    case BLE_GAP_EVENT_CONNECT:
        if (event->connect.status == 0) {
            s_conn_handle = event->connect.conn_handle;
            s_connected = true;
            ESP_LOGI(TAG, "a caregiver's phone connected");
            /* Nobody has to find a hub that is already connected, and
             * the radio is wanted for the burst. */
            update_advertising();
        } else {
            ESP_LOGW(TAG, "the connection failed: %d", event->connect.status);
        }
        break;

    case BLE_GAP_EVENT_DISCONNECT:
        ESP_LOGI(TAG, "the caregiver's phone disconnected (reason %d)",
                 event->disconnect.reason);
        s_conn_handle = BLE_HS_CONN_HANDLE_NONE;
        s_connected = false;
        s_alert_notify = false;
        s_state_notify = false;
        update_advertising();   /* so the next phone can find the hub */
        break;

    case BLE_GAP_EVENT_SUBSCRIBE:
        if (event->subscribe.attr_handle == s_alert_handle) {
            s_alert_notify = event->subscribe.cur_notify;
            ESP_LOGI(TAG, "the alert characteristic is %s",
                     s_alert_notify ? "subscribed" : "no longer subscribed");
        } else if (event->subscribe.attr_handle == s_state_handle) {
            s_state_notify = event->subscribe.cur_notify;
            ESP_LOGI(TAG, "the state characteristic is %s",
                     s_state_notify ? "subscribed" : "no longer subscribed");
        }
        break;

    default:
        break;
    }
    return 0;
```

```
}

static void on_reset(int reason)
{
    ESP_LOGW(TAG, "the Bluetooth stack reset (reason %d)", reason);
    s_conn_handle = BLE_HS_CONN_HANDLE_NONE;
    s_alert_notify = false;
    s_state_notify = false;
}

static void on_sync(void)
{
    int rc = ble_hs_util_ensure_addr(0);
    if (rc != 0) {
        ESP_LOGE(TAG, "no usable Bluetooth address: %d", rc);
        return;
    }
    update_advertising();
}

/* ------------------------------------------------------------- public -- */

static void notify_value(uint16_t handle, bool subscribed, const void *value,
                         uint16_t len)
{
    if (s_conn_handle == BLE_HS_CONN_HANDLE_NONE || !subscribed) {
        return;
    }
    struct os_mbuf *om = ble_hs_mbuf_from_flat(value, len);
    if (om == NULL) {
        ESP_LOGE(TAG, "the notification could not be built");
        return;
    }
    int rc = ble_gatts_notify_custom(s_conn_handle, handle, om);
    if (rc != 0) {
        ESP_LOGW(TAG, "the notification was not sent: %d", rc);
    }
}

void ble_alert_publish(uint8_t cot_id, bool crying, uint16_t confidence_pct,
                       uint32_t record)
{
    s_alert_value.cot_id = cot_id;
    s_alert_value.alert_type = crying ? 1u : 2u;
    s_alert_value.confidence_pct = confidence_pct;
    s_alert_value.event_id = event_number(record);

    ESP_LOGI(TAG, "%s cot %u at %u%%",
             crying ? "a cry on" : "the cry on", cot_id,
             (unsigned)confidence_pct);

    notify_value(s_alert_handle, s_alert_notify, &s_alert_value,
                 (uint16_t)sizeof s_alert_value);
}

void ble_alert_publish_deaf(uint8_t cot_id, uint32_t record)
{
    s_alert_value.cot_id = cot_id;
    s_alert_value.alert_type = 3u;
    s_alert_value.confidence_pct = 0;
    s_alert_value.event_id = event_number(record);

    ESP_LOGW(TAG, "cot %u is deaf: the hub cannot hear it", cot_id);

    notify_value(s_alert_handle, s_alert_notify, &s_alert_value,
                 (uint16_t)sizeof s_alert_value);
}

void ble_alert_publish_state(const ble_state_packet_t *state)
{
    s_state_value = *state;
    notify_value(s_state_handle, s_state_notify, &s_state_value,
                 (uint16_t)sizeof s_state_value);
}

int ble_alert_clients(void)
{
    return (s_conn_handle == BLE_HS_CONN_HANDLE_NONE) ? 0 : 1;
}

bool ble_alert_is_running(void)
{
    return s_running;
}

```

```
void ble_alert_set_preparing(bool preparing)
{
    if (s_preparing == preparing) {
        return;
    }
    s_preparing = preparing;
    update_advertising();
}

void ble_alert_set_radio_busy(bool busy)
{
    if (s_radio_busy == busy) {
        return;
    }
    s_radio_busy = busy;
    update_advertising();
}

static void host_task(void *param)
{
    (void)param;
    nimble_port_run();          /* returns only when the host stops */
    nimble_port_freertos_deinit();
}

bool ble_alert_start(void)
{
    int rc = nimble_port_init();
    if (rc != 0) {
        ESP_LOGE(TAG, "the Bluetooth host could not be started: %d", rc);
        return false;
    }

    ble_hs_cfg.reset_cb = on_reset;
    ble_hs_cfg.sync_cb = on_sync;
    ble_hs_cfg.gatts_register_cb = gatt_register_cb;
    ble_hs_cfg.store_status_cb = ble_store_util_status_rr;

    ble_svc_gap_device_name_set(DEVICE_NAME);
    ble_svc_gap_init();

    rc = gatt_svr_init();
    if (rc != 0) {
        ESP_LOGE(TAG, "the GATT service could not be registered: %d", rc);
        return false;
    }

    /* The host's own store, in NVS. */
    ble_store_config_init();

    nimble_port_freertos_init(host_task);
    s_running = true;
    ESP_LOGI(TAG, "the Bluetooth host is starting");
    return true;
}
```

Hub - the cry log, the clock, and the network (Chapter 12)

`firmware/hub_infer/components/crynet_log/include/crynet_log.h`

150 lines · 6,904 bytes

```
/*
 * CryNet hub — the cry log.
 *
 * Every cry the hub hears is written here, in the hub's own flash, and
 * it is written whether or not there is a network.  That is the whole
 * point: the system has to work exactly as it does now with no internet
 * at all, and the record must survive a power cut.  The network, when
 * it is there, takes the records off the hub and puts them on the
 * dashboard; until then they sit in the hub's flash.
 *
 * Where it lives: the partition table reserves six megabytes for it --
 *
 *     storage,  data, spiffs,  0x210000, 0x600000,
 *
 * and the comment on that line says what it is for.  It had never been
 * mounted, which is why the hub kept nothing: the space was there and
 * nothing used it.  This component mounts it and writes to it.
```

```
 *
 * What is written: one fixed-size record per event, thirty-two bytes,
 * appended to a file.  A record is one cry episode -- it starts when the
 * model says cry and ends when the cry has been held quiet long enough
 * to be sure it is over -- together with how long it lasted, how
 * confident the model was, and how loud it was.  The other kinds of
 * record (a cot going deaf, the hub starting) are the system's own
 * health, which belongs beside the cry data.
 *
 * The record is written once, when the episode ends, and it is never
 * changed afterwards.  The number the caregiver's wrist shows ("event
 * #1042") is the same number the dashboard shows, so the two can be
 * matched up after the fact.
 */

#ifndef CRYNET_LOG_H
#define CRYNET_LOG_H

#include <stdbool.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

/* The kinds of record. */
#define CRYNET_LOG_CRY  1u   /* a cry episode */
#define CRYNET_LOG_DEAF 2u   /* a cot went deaf */
#define CRYNET_LOG_HEARD 3u  /* a deaf cot came back */
#define CRYNET_LOG_BOOT 4u   /* the hub started */
#define CRYNET_LOG_TEST 5u   /* a test alert from the console */

/* One record.  Thirty-two bytes, so that a hundred and twenty-eight of
 * them fill a flash page exactly. */
typedef struct {
    uint32_t seq;             /* the hub's own number for this record */
    uint32_t boot_id;         /* which run of the hub it belongs to */
    uint32_t uptime_ms;       /* when it happened, from the hub's start */
    uint32_t epoch_s;         /* and in the world, if the clock knew */
    uint32_t duration_ms;     /* how long the episode lasted */
    uint16_t peak_rms_milli;  /* the loudest moment, in thousandths */
    uint16_t mean_rms_milli;  /* and the average over the episode */
    uint8_t  kind;            /* one of the kinds above */
    uint8_t  cot;             /* which cot, 1-based */
    uint8_t  peak_conf;       /* the model's highest confidence, percent */
    uint8_t  windows;         /* how many windows the model judged */
    uint32_t reserved;        /* room for the next field */
} __attribute__((packed)) crynet_log_event_t;

/* Mount the log and open the file.  Returns false if the flash could
 * not be mounted -- the hub carries on without a log rather than
 * refusing to work. */
bool crynet_log_init(void);

/* Take the next record number for an event that is happening now, and
 * remember when it started.  The number is returned so that the
 * caregiver's wrist can show it while the episode is still running.
 * Returns 0 if the log is not available. */
uint32_t crynet_log_reserve(uint8_t kind, uint8_t cot);

/* Write the record for an event that has finished.  The numbers are the
 * ones the hub measured while it was happening. */
bool crynet_log_commit(uint32_t seq, uint32_t duration_ms, uint8_t peak_conf,
                       uint8_t windows, uint16_t peak_rms_milli,
                       uint16_t mean_rms_milli);

/* Write a record for an event that needs no waiting: a cot going deaf,
 * the hub starting.  Returns the record's number, or 0 if the log is
 * not available -- the number is what the wrist shows, so it has to
 * come back to the caller. */
uint32_t crynet_log_write(uint8_t kind, uint8_t cot, uint32_t duration_ms,
                          uint8_t peak_conf, uint8_t windows,
                          uint16_t peak_rms_milli, uint16_t mean_rms_milli);

/* The records that have not been sent yet, in the order the file holds
 * them.  Returns how many were written into `out`, and the number the
 * caller may claim as sent once the batch has gone -- everything up to
 * it has been sent, and nothing above it has.  That number is not
 * simply the highest one in the batch: two cots crying at once can end
 * in the opposite order to the one they started in, and an episode
 * that has started but not yet ended holds a number of its own.  The
 * log accounts for both, so no record is ever claimed as sent before
 * it has gone. */
int crynet_log_pending(crynet_log_event_t *out, int max, uint32_t *last_seq);

```

```
/* Say that everything up to this number has been sent. */
bool crynet_log_mark_uploaded(uint32_t seq);

/* The last records, newest first, for the console's history command. */
int crynet_log_recent(crynet_log_event_t *out, int max);

/* How many records the log holds, and how many are still to send. */
uint32_t crynet_log_total(void);
uint32_t crynet_log_pending_count(void);

/* Keep the log to a rolling window once the records have been sent.
 * Records that have not been sent are never dropped unless the log has
 * run out of room, and that is said out loud when it happens. */
void crynet_log_trim(void);

/* Throw the log away.  The record numbers carry on, so the dashboard
 * does not confuse the new records with the old ones. */
bool crynet_log_clear(void);

/* The last wall-clock time the hub knew, kept across a restart so that
 * the clock can start from it.  Zero if it never knew.  See
 * crynet_clock.h for why this is needed and what it costs. */
uint64_t crynet_log_last_epoch_ms(void);
bool crynet_log_save_epoch_ms(uint64_t epoch_ms);

/* How many records have been lost to a full log.  It should be nought;
 * anything else is said out loud, because it means the hub was offline
 * long enough to fill six megabytes. */
uint32_t crynet_log_dropped(void);

/* Which run of the hub this is, counting from the first time the log
 * was made. */
uint32_t crynet_log_boot_id(void);

/* Run the on-board proof of the ordering rule: two cots crying at
 * once, the first cry to start ending last, the uploader running in
 * between -- every record must still be sent.  Runs on the chip,
 * against the real component, on its own scratch file: the real log
 * is not touched.  Returns true if the proof held.  See
 * firmware/tools/test_log_order.py for the host model of the same
 * property. */
bool crynet_log_selftest(void);

#ifdef __cplusplus
}
#endif

#endif /* CRYNET_LOG_H */
```

firmware/hub_infer/components/crynet_log/crynet_log.c

1125 lines · 38,919 bytes

```
/*
 * CryNet hub — the cry log.  See include/crynet_log.h for what it is for.
 *
 * The shape of it: one file of fixed-size records, appended to, and a
 * second tiny file holding the counters.  Two files rather than one
 * because SPIFFS cannot rewrite four bytes in the middle of a file
 * cheaply -- it rewrites the whole file -- and the counters change on
 * every record while the records themselves never change at all.
 *
 * Nothing here is allowed to break the hub.  Every call is best-effort:
 * if the flash will not mount, or a write fails, the log says so and
 * returns, and the cot still cries on the caregiver's wrist exactly as
 * it did before any of this existed.
 */

#include <errno.h>
#include <fcntl.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>

#include "esp_log.h"
#include "esp_spiffs.h"
#include "esp_timer.h"
#include "freertos/FreeRTOS.h"
```

```
#include "freertos/semphr.h"

#include "crynet_clock.h"
#include "crynet_log.h"

static const char *TAG = "crylog";

#define LOG_DIR     "/log"
#define EVENTS_PATH "/log/events.bin"
#define META_PATH   "/log/meta.bin"
#define EVENTS_TMP  "/log/events.tmp"

/* The self-test's own files, on the same partition.  They are deleted
 * when it finishes and are never the real log. */
#define SELFTEST_PATH      "/log/selftest.bin"
#define SELFTEST_META_PATH "/log/selftest.meta"

/* Where the counters live.  A variable rather than a constant so the
 * self-test can point it at its own file: the real counters must not
 * be touched by a test, not even for a moment. */
static const char *s_meta_path = META_PATH;

#define REC_SIZE    32u
#define HEADER_SIZE 16u

/* "CRY1" and "CRYM", read as little-endian words. */
#define EVENTS_MAGIC 0x31595243u
#define META_MAGIC   0x4D595243u
#define FORMAT_VER   1u

/* Six megabytes would hold a hundred and ninety thousand records; the
 * cap is far below that on purpose.  SPIFFS slows down as a file grows
 * and wears the flash while it does it, and a caregiver does not need
 * eight thousand episodes on the hub once they are on the dashboard.
 * At a busy ten cries a night, eight thousand is two years. */
#define MAX_RECORDS       8192u
#define KEEP_AFTER_UPLOAD 2000u

/* Records copied at a time when the log is compacted.  Four kilobytes
 * of stack-free heap, so compaction needs no large allocation. */
#define COPY_CHUNK 128u

/* Episodes that have started and not yet finished.  Two cots can cry
 * at once and a deaf cot can come back mid-cry, so eight is generous. */
#define OPEN_SLOTS 8u

_Static_assert(sizeof(crynet_log_event_t) == REC_SIZE,
               "the record must stay thirty-two bytes");

typedef struct {
    uint32_t magic;
    uint32_t next_seq;       /* the number the next record will get */
    uint32_t uploaded_seq;   /* everything up to here has been sent */
    uint32_t boot_id;
    uint32_t dropped;
    uint32_t reserved;
    uint64_t last_epoch_ms;
} log_meta_t;

_Static_assert(sizeof(log_meta_t) == 32, "the meta file is thirty-two bytes");

typedef struct {
    uint32_t seq;            /* nought means the slot is free */
    uint32_t start_epoch_s;
    uint32_t start_uptime_ms;
    uint8_t  kind;
    uint8_t  cot;
} open_slot_t;

static bool              s_ready = false;
static SemaphoreHandle_t s_lock = NULL;
static int               s_fd = -1;
static log_meta_t        s_meta;
static open_slot_t       s_open[OPEN_SLOTS];

/* ------------------------------------------------------------------ */
/* Opening a file that may not be there yet.                          */
/* ------------------------------------------------------------------ */

/* ESP-IDF's VFS only honours O_CREAT when it arrives with O_EXCL or
 * O_TRUNC (see spiffs_mode_conv in esp_spiffs.c): a plain O_CREAT is
 * dropped on the way down, and the open then fails with ENOENT on a
 * file that does not exist.  On a board whose storage was just
 * formatted -- which is every brand-new board, flashed from scratch --
 * that meant the hub could not create its own log, and it went on
```

```
 * running with no record at all.  So: create it with O_EXCL, and fall
 * back to a plain open when it is already there. */
static int open_create_if_missing(const char *path, int flags)
{
    int fd = open(path, flags | O_CREAT | O_EXCL, 0644);
    if (fd >= 0 || errno != EEXIST) {
        return fd;
    }
    return open(path, flags, 0644);
}

/* SPIFFS has neither pread nor pwrite: the VFS answers both with
 * "function not implemented".  A read at an offset is a seek and a
 * read, and a write at an offset is a seek and a write.  The log is
 * the one place this matters, because every record is read back by
 * number and the counters are rewritten in place. */
static bool read_at(int fd, void *buf, size_t n, off_t off)
{
    if (lseek(fd, off, SEEK_SET) != off) {
        return false;
    }
    return read(fd, buf, n) == (ssize_t)n;
}

static bool write_at(int fd, const void *buf, size_t n, off_t off)
{
    if (lseek(fd, off, SEEK_SET) != off) {
        return false;
    }
    return write(fd, buf, n) == (ssize_t)n;
}

/* ------------------------------------------------------------------ */
/* The file, with the lock already held.                              */

static uint32_t count_locked(void)
{
    struct stat st;
    if (s_fd < 0 || fstat(s_fd, &st) != 0) {
        return 0;
    }
    if (st.st_size <= (off_t)HEADER_SIZE) {
        return 0;
    }
    return (uint32_t)((st.st_size - HEADER_SIZE) / REC_SIZE);
}

static bool read_at_locked(uint32_t index, crynet_log_event_t *out)
{
    const off_t off = (off_t)(HEADER_SIZE + index * REC_SIZE);
    return read_at(s_fd, out, REC_SIZE, off);
}

static bool save_meta_locked(void)
{
    /* Not O_TRUNC.  Truncating would delete the file and build it
     * again, which is more pages of flash than writing the same
     * thirty-two bytes where they already are.  The counters file is
     * written on the path between a cot's microphone and the
     * caregiver's wrist, so what it costs is worth counting. */
    const int fd = open_create_if_missing(s_meta_path, O_WRONLY);
    if (fd < 0) {
        ESP_LOGE(TAG, "cannot write the counters: %s", strerror(errno));
        return false;
    }
    bool ok = write_at(fd, &s_meta, sizeof(s_meta), 0);
    if (ok && ftruncate(fd, (off_t)sizeof(s_meta)) != 0) {
        ok = false;
    }
    if (ok) {
        fsync(fd);
    }
    close(fd);
    if (!ok) {
        ESP_LOGE(TAG, "the counters did not write: %s", strerror(errno));
    }
    return ok;
}

static bool open_events_locked(void)
{
    if (s_fd >= 0) {
        close(s_fd);
        s_fd = -1;
    }
```

```
    s_fd = open_create_if_missing(EVENTS_PATH, O_RDWR);
    if (s_fd < 0) {
        ESP_LOGE(TAG, "cannot open the log: %s", strerror(errno));
        return false;
    }

    struct stat st;
    if (fstat(s_fd, &st) != 0) {
        return false;
    }

    if (st.st_size == 0) {
        uint32_t head[4] = { EVENTS_MAGIC, FORMAT_VER, REC_SIZE, 0 };
        if (write(s_fd, head, HEADER_SIZE) != (ssize_t)HEADER_SIZE) {
            ESP_LOGE(TAG, "cannot start the log: %s", strerror(errno));
            return false;
        }
        fsync(s_fd);
        return true;
    }

    uint32_t head[4] = { 0, 0, 0, 0 };
    if (!read_at(s_fd, head, HEADER_SIZE, 0) ||
        head[0] != EVENTS_MAGIC || head[2] != REC_SIZE) {
        /* Not a log we wrote, or one from a format we no longer read.
         * Start again rather than append nonsense to it.  The record
         * numbers carry on so the dashboard is never confused. */
        ESP_LOGW(TAG, "the log file is not one of ours; starting a new one");
        close(s_fd);
        unlink(EVENTS_PATH);
        s_fd = open(EVENTS_PATH, O_RDWR | O_CREAT | O_TRUNC, 0644);
        if (s_fd < 0) {
            return false;
        }
        head[0] = EVENTS_MAGIC; head[1] = FORMAT_VER;
        head[2] = REC_SIZE;     head[3] = 0;
        return write(s_fd, head, HEADER_SIZE) == (ssize_t)HEADER_SIZE;
    }

    /* A power cut mid-append can leave half a record.  Cut it off, so
     * that every record after it still lands on its own boundary. */
    const off_t whole = (off_t)(HEADER_SIZE +
                                ((st.st_size - HEADER_SIZE) / REC_SIZE) * REC_SIZE);
    if (whole != st.st_size) {
        ESP_LOGW(TAG, "a half-written record at the end of the log; trimming it");
        if (ftruncate(s_fd, whole) != 0) {
            ESP_LOGE(TAG, "cannot trim the log: %s", strerror(errno));
        }
        fsync(s_fd);
    }
    return true;
}

static bool append_locked(const crynet_log_event_t *rec)
{
    if (s_fd < 0 && !open_events_locked()) {
        return false;
    }
    if (lseek(s_fd, 0, SEEK_END) == (off_t)-1) {
        return false;
    }
    if (write(s_fd, rec, REC_SIZE) != (ssize_t)REC_SIZE) {
        ESP_LOGE(TAG, "record %lu did not write: %s",
                 (unsigned long)rec->seq, strerror(errno));
        /* The next write must not land after a partial one. */
        const uint32_t n = count_locked();
        ftruncate(s_fd, (off_t)(HEADER_SIZE + n * REC_SIZE));
        return false;
    }
    fsync(s_fd);
    return true;
}

/* What the file holds that has not been sent yet.
 *
 * The file is in the order the records were *written*, and that is not
 * the order they were *numbered*.  Two cots crying at once can end in
 * the opposite order to the one they started in -- cot 1's long cry
 * holds its number while cot 2's short one finishes first -- so a low
 * number can sit after a high one.  Nothing here may assume the file
 * is sorted.
 *
 * A binary search over it answered wrongly in exactly the case that
 * matters: asked for the first record above a number, it could land
 * past a record whose number was lower, and that record would then
```

```
 * never be sent.  So the file is walked in order instead.  The walk is
 * on the uploader's own task, never on the path between a cot's
 * microphone and the caregiver's wrist. */
typedef struct {
    int      count;      /* how many records were collected */
    uint32_t last_seq;   /* the highest number among them */
    uint32_t stuck;      /* the lowest number that did not fit, 0 if none */
    uint32_t total;      /* how many are still to send in all */
    uint32_t first;      /* where the first of them sits in the file */
} pending_t;

/* How many records are read at a time when the file is walked.  A
 * kilobyte at a time: the file is read in blocks rather than record by
 * record, because SPIFFS charges for the call and not for the bytes. */
#define SCAN_RECORDS 32u

static crynet_log_event_t s_scan[SCAN_RECORDS];

/* The batch is the *lowest* numbers still to send, not the first ones
 * in the file.  The file is in the order the records were written,
 * which is not the order they were numbered, so "the first ones in
 * the file" could be a record with a high number while a lower one
 * sits behind it -- and then that lower one would be sent over and
 * over and the log would never move on.  These are the candidates,
 * held in number order; one more than a full batch, so the lowest
 * number that did not fit can be seen. */
#define PICK_MAX 32u

typedef struct {
    uint32_t seq;
    uint32_t index;
} pick_t;

static pick_t s_pick[PICK_MAX + 1u];

/* Everything in the file whose number is above `after`.  The lowest
 * `max` of them are written into `out` (which may be null, to count
 * them instead) and the rest are described by `stuck`. */
static pending_t collect_locked(uint32_t after, crynet_log_event_t *out,
                                int max)
{
    pending_t p;
    p.count = 0;
    p.last_seq = 0;
    p.stuck = 0;
    p.total = 0;
    p.first = UINT32_MAX;

    if (max > (int)PICK_MAX) {
        max = (int)PICK_MAX;
    }
    const uint32_t cap = (max > 0) ? (uint32_t)max + 1u : 1u;
    uint32_t npick = 0;

    const uint32_t n = count_locked();
    for (uint32_t i = 0; i < n; ) {
        uint32_t take = n - i;
        if (take > SCAN_RECORDS) {
            take = SCAN_RECORDS;
        }
        const off_t off = (off_t)(HEADER_SIZE + i * REC_SIZE);
        if (!read_at(s_fd, s_scan, take * REC_SIZE, off)) {
            /* A short read: the flash is failing, or the file changed
             * under us.  Stop rather than guess. */
            ESP_LOGE(TAG, "the log could not be read at record %lu",
                     (unsigned long)i);
            break;
        }
        for (uint32_t j = 0; j < take; j++) {
            const crynet_log_event_t *rec = &s_scan[j];
            if (rec->seq <= after) {
                continue;          /* already sent */
            }
            if (p.first == UINT32_MAX) {
                p.first = i + j;
            }
            p.total++;
            if (out == NULL) {
                continue;          /* counting only */
            }
            /* Keep the lowest `cap` numbers seen so far. */
            if (npick == cap) {
                if (rec->seq >= s_pick[npick - 1u].seq) {
                    continue;      /* not among the lowest */
                }
```

```
                npick--;
            }
            uint32_t k = npick;
            while (k > 0 && s_pick[k - 1u].seq > rec->seq) {
                s_pick[k] = s_pick[k - 1u];
                k--;
            }
            s_pick[k].seq = rec->seq;
            s_pick[k].index = i + j;
            npick++;
        }
        i += take;
    }

    if (out != NULL) {
        const uint32_t take = (npick < (uint32_t)max) ? npick : (uint32_t)max;
        for (uint32_t k = 0; k < take; k++) {
            if (!read_at_locked(s_pick[k].index, &out[k])) {
                break;
            }
            p.count++;
            p.last_seq = s_pick[k].seq;
        }
        /* The lowest number that did not fit in the batch. */
        if (npick == cap && cap > (uint32_t)max) {
            p.stuck = s_pick[max].seq;
        }
    }
    return p;
}

/* The lowest number that has been taken but not yet written. */
static uint32_t lowest_open_locked(void)
{
    uint32_t lowest = 0;
    for (int i = 0; i < OPEN_SLOTS; i++) {
        if (s_open[i].seq == 0) {
            continue;
        }
        if (lowest == 0 || s_open[i].seq < lowest) {
            lowest = s_open[i].seq;
        }
    }
    return lowest;
}

/* The number the uploader may claim as sent.
 *
 * It is not simply the highest number in the batch.  A record that is
 * still being measured -- an episode that has started but not ended --
 * holds a number lower than some records that have already gone, and
 * claiming past it would lose that record for good.  So the claim
 * stops below the lowest number still in progress, and below the
 * lowest number that did not fit in the batch. */
static uint32_t watermark_locked(uint32_t last_seq, uint32_t stuck)
{
    uint32_t w = last_seq;
    if (stuck != 0 && (w == 0 || (stuck - 1u) < w)) {
        w = stuck - 1u;
    }
    const uint32_t open = lowest_open_locked();
    if (open != 0) {
        const uint32_t cap = open - 1u;
        if (cap < w) {
            w = cap;
        }
    }
    if (w < s_meta.uploaded_seq) {
        w = s_meta.uploaded_seq;
    }
    return w;
}

/* ------------------------------------------------------------------ */
/* Starting up.                                                       */

bool crynet_log_init(void)
{
    if (s_ready) {
        return true;
    }
    if (s_lock == NULL) {
        s_lock = xSemaphoreCreateMutex();
        if (s_lock == NULL) {
            return false;
```

```
        }
    }

    const esp_vfs_spiffs_conf_t conf = {
        .base_path = LOG_DIR,
        .partition_label = "storage",
        .max_files = 4,
        .format_if_mount_failed = true,
    };
    esp_err_t err = esp_vfs_spiffs_register(&conf);
    if (err != ESP_OK) {
        ESP_LOGE(TAG, "the storage partition will not mount (%s); "
                      "the hub carries on without a log",
                 esp_err_to_name(err));
        return false;
    }

    size_t total = 0, used = 0;
    if (esp_spiffs_info("storage", &total, &used) == ESP_OK) {
        ESP_LOGI(TAG, "storage mounted: %u of %u bytes used",
                 (unsigned)used, (unsigned)total);
    }

    xSemaphoreTake(s_lock, portMAX_DELAY);

    memset(&s_meta, 0, sizeof(s_meta));
    memset(s_open, 0, sizeof(s_open));
    s_meta.magic = META_MAGIC;
    s_meta.next_seq = 1;

    const int fd = open(s_meta_path, O_RDONLY);
    if (fd >= 0) {
        log_meta_t stored;
        if (read(fd, &stored, sizeof(stored)) == (ssize_t)sizeof(stored) &&
            stored.magic == META_MAGIC && stored.next_seq != 0) {
            s_meta = stored;
        } else {
            ESP_LOGW(TAG, "the counters were unreadable; starting from one");
        }
        close(fd);
    }
    s_meta.boot_id++;
    s_meta.reserved = 0;

    if (!open_events_locked()) {
        xSemaphoreGive(s_lock);
        esp_vfs_spiffs_unregister("storage");
        return false;
    }

    /* The counters file and the records file can fall out of step: a
     * power cut between writing a record and writing the counters leaves
     * the file holding a record whose number the counters have not
     * caught up with, and the next number handed out would be handed out
     * twice.  The numbers in the file are the truth, so the highest
     * number in the tail of the file sets the next one.  The tail is
     * enough: numbers only ever grow, and the file is written in the
     * order the records finished, so the highest one is at the end (a
     * cry that is still being measured holds its number, and there are
     * at most eight of those open at once). */
    const uint32_t held = count_locked();
    const uint32_t look = held < SCAN_RECORDS ? held : SCAN_RECORDS;
    uint32_t highest = 0;
    if (look > 0) {
        const off_t off = (off_t)(HEADER_SIZE + (held - look) * REC_SIZE);
        if (read_at(s_fd, s_scan, look * REC_SIZE, off)) {
            for (uint32_t j = 0; j < look; j++) {
                if (s_scan[j].seq > highest) {
                    highest = s_scan[j].seq;
                }
            }
        }
    }
    if (highest + 1 > s_meta.next_seq) {
        ESP_LOGW(TAG, "the counters said the next number is %lu but the log "
                      "holds %lu; taking the log's word",
                 (unsigned long)s_meta.next_seq, (unsigned long)highest);
        s_meta.next_seq = highest + 1;
    }

    save_meta_locked();

    const uint32_t n = count_locked();
    const pending_t p = collect_locked(s_meta.uploaded_seq, NULL, 0);
    const uint32_t pending = p.total;
```

```
    s_ready = true;

    ESP_LOGI(TAG, "boot %lu: %lu records kept, %lu still to send, "
                  "next number %lu",
             (unsigned long)s_meta.boot_id, (unsigned long)n,
             (unsigned long)pending, (unsigned long)s_meta.next_seq);
    if (s_meta.dropped) {
        ESP_LOGW(TAG, "%lu records were lost to a full log before this boot",
                 (unsigned long)s_meta.dropped);
    }

    xSemaphoreGive(s_lock);
    return true;
}

/* ---------------------------------------------------------------- */
/* Writing.                                                         */

/* The bodies of the two writing calls, with the lock already held.
 * They are split out so the self-test below can run the very same
 * code against a scratch file without taking the lock twice. */

static uint32_t reserve_locked(uint8_t kind, uint8_t cot)
{
    const int64_t t0 = esp_timer_get_time();
    const uint32_t seq = s_meta.next_seq++;
    /* Saved now, not at commit: if the hub loses power mid-cry, the
     * number is spent and the next boot leaves a hole rather than
     * handing the same number to a different cry.
     *
     * This is the one flash write that happens while a cot is crying,
     * so its cost is measured and printed: the owner can see what it
     * actually costs on the board in front of them rather than take
     * an estimate on trust. */
    save_meta_locked();
    const int64_t t1 = esp_timer_get_time();
    ESP_LOGI(TAG, "record %lu: the number was made durable in %lld ms",
             (unsigned long)seq, (long long)((t1 - t0) / 1000));

    int slot = -1;
    for (int i = 0; i < OPEN_SLOTS; i++) {
        if (s_open[i].seq == 0) {
            slot = i;
            break;
        }
    }
    if (slot < 0) {
        /* Should not happen with two cots.  Say so, and drop the
         * oldest rather than refuse the new cry. */
        slot = 0;
        for (int i = 1; i < OPEN_SLOTS; i++) {
            if (s_open[i].seq < s_open[slot].seq) {
                slot = i;
            }
        }
        ESP_LOGW(TAG, "more than %u episodes open at once; forgetting %lu",
                 OPEN_SLOTS, (unsigned long)s_open[slot].seq);
    }

    const uint64_t epoch_ms = crynet_clock_epoch_ms();
    s_open[slot].seq = seq;
    s_open[slot].kind = kind;
    s_open[slot].cot = cot;
    s_open[slot].start_epoch_s = (uint32_t)(epoch_ms / 1000u);
    s_open[slot].start_uptime_ms = (uint32_t)crynet_clock_uptime_ms();
    return seq;
}

uint32_t crynet_log_reserve(uint8_t kind, uint8_t cot)
{
    if (!s_ready) {
        return 0;
    }
    xSemaphoreTake(s_lock, portMAX_DELAY);
    const uint32_t seq = reserve_locked(kind, cot);
    xSemaphoreGive(s_lock);
    return seq;
}

static bool commit_locked(uint32_t seq, uint32_t duration_ms, uint8_t peak_conf,
                          uint8_t windows, uint16_t peak_rms_milli,
                          uint16_t mean_rms_milli)
{
    int slot = -1;
    for (int i = 0; i < OPEN_SLOTS; i++) {
```

```
        if (s_open[i].seq == seq) {
            slot = i;
            break;
        }
    }
    if (slot < 0) {
        ESP_LOGW(TAG, "record %lu was not open; nothing written",
                 (unsigned long)seq);
        return false;
    }

    crynet_log_event_t rec;
    memset(&rec, 0, sizeof(rec));
    rec.seq = seq;
    rec.boot_id = s_meta.boot_id;
    rec.uptime_ms = s_open[slot].start_uptime_ms;
    rec.epoch_s = s_open[slot].start_epoch_s;
    rec.duration_ms = duration_ms;
    rec.peak_rms_milli = peak_rms_milli;
    rec.mean_rms_milli = mean_rms_milli;
    rec.kind = s_open[slot].kind;
    rec.cot = s_open[slot].cot;
    rec.peak_conf = peak_conf;
    rec.windows = windows;
    rec.reserved = 0;

    const int64_t t0 = esp_timer_get_time();
    const bool ok = append_locked(&rec);
    const int64_t t1 = esp_timer_get_time();
    ESP_LOGI(TAG, "record %lu: %s in %lld ms",
             (unsigned long)seq, ok ? "written" : "NOT written",
             (long long)((t1 - t0) / 1000));
    memset(&s_open[slot], 0, sizeof(s_open[slot]));

    /* Remember the time for the next boot while we are here -- but
     * only when it has moved.  The clock is set once by SNTP and then
     * runs on its own, so a cry episode normally does not change it,
     * and every page of flash rewritten here is a page the radio path
     * has to wait for. */
    const uint64_t now_ms = crynet_clock_epoch_ms();
    if (now_ms != 0 &&
        (s_meta.last_epoch_ms == 0 ||
         now_ms > s_meta.last_epoch_ms + 60000u ||
         now_ms + 60000u < s_meta.last_epoch_ms)) {
        s_meta.last_epoch_ms = now_ms;
        save_meta_locked();
    }

    return ok;
}

bool crynet_log_commit(uint32_t seq, uint32_t duration_ms, uint8_t peak_conf,
                       uint8_t windows, uint16_t peak_rms_milli,
                       uint16_t mean_rms_milli)
{
    if (!s_ready || seq == 0) {
        return false;
    }
    xSemaphoreTake(s_lock, portMAX_DELAY);
    const bool ok = commit_locked(seq, duration_ms, peak_conf, windows,
                                  peak_rms_milli, mean_rms_milli);
    xSemaphoreGive(s_lock);
    return ok;
}

uint32_t crynet_log_write(uint8_t kind, uint8_t cot, uint32_t duration_ms,
                          uint8_t peak_conf, uint8_t windows,
                          uint16_t peak_rms_milli, uint16_t mean_rms_milli)
{
    const uint32_t seq = crynet_log_reserve(kind, cot);
    if (seq == 0) {
        return 0;
    }
    return crynet_log_commit(seq, duration_ms, peak_conf, windows,
                             peak_rms_milli, mean_rms_milli) ? seq : 0;
}

/* ------------------------------------------------------------- */
/* Reading, for the uploader and the console.                    */

int crynet_log_pending(crynet_log_event_t *out, int max, uint32_t *sent_upto)
{
    if (sent_upto) {
        *sent_upto = 0;
    }
```

```
    if (!s_ready || out == NULL || max <= 0) {
        return 0;
    }
    xSemaphoreTake(s_lock, portMAX_DELAY);

    const pending_t p = collect_locked(s_meta.uploaded_seq, out, max);
    if (sent_upto) {
        /* The number the caller may claim as sent once the batch has
         * gone.  It is worked out here, while the state that decides
         * it is still true: an episode that is being measured holds a
         * number of its own, and if it finished between this call and
         * the claim, the claim must not have been made past it. */
        *sent_upto = watermark_locked(p.last_seq, p.stuck);
    }

    xSemaphoreGive(s_lock);
    return p.count;
}

bool crynet_log_mark_uploaded(uint32_t seq)
{
    if (!s_ready) {
        return false;
    }
    xSemaphoreTake(s_lock, portMAX_DELAY);
    bool ok = true;
    if (seq > s_meta.uploaded_seq) {
        s_meta.uploaded_seq = seq;
        ok = save_meta_locked();
    }
    xSemaphoreGive(s_lock);
    return ok;
}

int crynet_log_recent(crynet_log_event_t *out, int max)
{
    if (!s_ready || out == NULL || max <= 0) {
        return 0;
    }
    xSemaphoreTake(s_lock, portMAX_DELAY);

    const uint32_t n = count_locked();
    uint32_t take = (n < (uint32_t)max) ? n : (uint32_t)max;
    for (uint32_t i = 0; i < take; i++) {
        if (!read_at_locked(n - 1 - i, &out[i])) {
            take = i;
            break;
        }
    }

    xSemaphoreGive(s_lock);
    return (int)take;
}

uint32_t crynet_log_total(void)
{
    if (!s_ready) {
        return 0;
    }
    xSemaphoreTake(s_lock, portMAX_DELAY);
    const uint32_t n = count_locked();
    xSemaphoreGive(s_lock);
    return n;
}

uint32_t crynet_log_pending_count(void)
{
    if (!s_ready) {
        return 0;
    }
    xSemaphoreTake(s_lock, portMAX_DELAY);
    const pending_t p = collect_locked(s_meta.uploaded_seq, NULL, 0);
    xSemaphoreGive(s_lock);
    return p.total;
}

uint32_t crynet_log_dropped(void)
{
    return s_meta.dropped;
}

uint32_t crynet_log_boot_id(void)
{
    return s_meta.boot_id;
}
```

```
795
796 uint64_t crynet_log_last_epoch_ms(void)
797 {
798     return s_meta.last_epoch_ms;
799 }
800
801 bool crynet_log_save_epoch_ms(uint64_t epoch_ms)
802 {
803     if (!s_ready || epoch_ms == 0) {
804         return false;
805     }
806     xSemaphoreTake(s_lock, portMAX_DELAY);
807     /* Only worth a flash write when it has moved on a minute; the
808      * uploader calls this every time it talks to the dashboard. */
809     bool ok = true;
810     if (epoch_ms > s_meta.last_epoch_ms + 60000u ||
811         s_meta.last_epoch_ms == 0) {
812         s_meta.last_epoch_ms = epoch_ms;
813         ok = save_meta_locked();
814     }
815     xSemaphoreGive(s_lock);
816     return ok;
817 }
818
819 /* ------------------------------------------------------------- */
820 /* Keeping it to a rolling window.                               */
821
822 void crynet_log_trim(void)
823 {
824     if (!s_ready) {
825         return;
826     }
827     xSemaphoreTake(s_lock, portMAX_DELAY);
828
829     const uint32_t n = count_locked();
830     const pending_t p = collect_locked(s_meta.uploaded_seq, NULL, 0);
831     const uint32_t pending = p.total;
832     /* Where the first unsent record sits.  The records before it have
833      * been sent; the ones after it have not. */
834     const uint32_t first_unsent = (p.first == UINT32_MAX) ? n : p.first;
835
836     /* Keep everything still to send, plus a window of what has been
837      * sent so the console's history still has something to show. */
838     uint32_t target = pending;
839     if (target < KEEP_AFTER_UPLOAD) {
840         target = KEEP_AFTER_UPLOAD;
841     }
842     if (target > MAX_RECORDS) {
843         target = MAX_RECORDS;   /* out of room: something has to go */
844     }
845     if (n <= target) {
846         xSemaphoreGive(s_lock);
847         return;
848     }
849
850     /* The old records go from the front, and only those that have
851      * been sent: the unsent ones can sit anywhere in the file, because
852      * two cots crying at once can end in the opposite order to the one
853      * they started in. */
854     uint32_t from = n - target;
855     if (from > first_unsent) {
856         from = first_unsent;
857     }
858     if (n - from > MAX_RECORDS) {
859         /* The log is over its cap and what cannot be kept are records
860          * that have not been sent.  That is a loss, and it is said out
861          * loud, because it means the hub was offline long enough to
862          * fill six megabytes. */
863         const uint32_t keep_from = n - MAX_RECORDS;
864         const uint32_t lost = (keep_from > first_unsent)
865                               ? (keep_from - first_unsent) : 0;
866         s_meta.dropped += lost;
867         ESP_LOGW(TAG, "the log is full and %lu unsent records are being "
868                       "dropped; the hub has been offline too long",
869                  (unsigned long)lost);
870         from = keep_from;
871     }
872     const uint32_t keep = n - from;
873
874     /* Copy the tail into a new file, then swap.  A power cut during
875      * the copy leaves the original untouched. */
876     crynet_log_event_t *buf = malloc(COPY_CHUNK * REC_SIZE);
877     if (buf == NULL) {
878         ESP_LOGE(TAG, "no memory to compact the log; leaving it as it is");
879         xSemaphoreGive(s_lock);
```

```
        return;
    }

    const int tmp = open(EVENTS_TMP, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (tmp < 0) {
        ESP_LOGE(TAG, "cannot start the new log: %s", strerror(errno));
        free(buf);
        xSemaphoreGive(s_lock);
        return;
    }
    uint32_t head[4] = { EVENTS_MAGIC, FORMAT_VER, REC_SIZE, 0 };
    bool ok = write(tmp, head, HEADER_SIZE) == (ssize_t)HEADER_SIZE;

    for (uint32_t done = 0; ok && done < keep; ) {
        uint32_t take = keep - done;
        if (take > COPY_CHUNK) {
            take = COPY_CHUNK;
        }
        const off_t off = (off_t)(HEADER_SIZE + (from + done) * REC_SIZE);
        if (!read_at(s_fd, buf, take * REC_SIZE, off)) {
            ok = false;
            break;
        }
        if (write(tmp, buf, take * REC_SIZE) != (ssize_t)(take * REC_SIZE)) {
            ok = false;
            break;
        }
        done += take;
    }
    if (ok) {
        fsync(tmp);
    }
    close(tmp);
    free(buf);

    if (!ok) {
        ESP_LOGE(TAG, "compacting the log failed: %s; keeping the old one",
                 strerror(errno));
        unlink(EVENTS_TMP);
        xSemaphoreGive(s_lock);
        return;
    }

    close(s_fd);
    s_fd = -1;
    unlink(EVENTS_PATH);
    if (rename(EVENTS_TMP, EVENTS_PATH) != 0) {
        ESP_LOGE(TAG, "cannot put the compacted log in place: %s",
                 strerror(errno));
        open_events_locked();
        xSemaphoreGive(s_lock);
        return;
    }
    if (!open_events_locked()) {
        s_ready = false;
        xSemaphoreGive(s_lock);
        return;
    }

    /* Everything below the oldest kept record is gone; the sent pointer
     * must not point into thin air. */
    crynet_log_event_t oldest;
    if (read_at_locked(0, &oldest) && s_meta.uploaded_seq < oldest.seq) {
        s_meta.uploaded_seq = oldest.seq - 1;
    }
    save_meta_locked();

    ESP_LOGI(TAG, "log compacted: %lu dropped, %lu kept",
             (unsigned long)from, (unsigned long)keep);

    xSemaphoreGive(s_lock);
}

bool crynet_log_clear(void)
{
    if (!s_ready) {
        return false;
    }
    xSemaphoreTake(s_lock, portMAX_DELAY);

    memset(s_open, 0, sizeof(s_open));
    const bool ok = open_events_locked() &&
                    ftruncate(s_fd, (off_t)HEADER_SIZE) == 0;
    if (ok) {
        fsync(s_fd);
```

```
        /* The numbers carry on: a new record must never reuse an old
         * one's number, or the dashboard would ignore it. */
        s_meta.uploaded_seq = s_meta.next_seq - 1;
        save_meta_locked();
        ESP_LOGI(TAG, "log cleared; numbering carries on at %lu",
                 (unsigned long)s_meta.next_seq);
    }

    xSemaphoreGive(s_lock);
    return ok;
}

/* ---------------------------------------------------------------- */
/* The self-test.                                                   */

/* Run the awkward case against the real code, on a scratch file, and
 * report whether it passed.
 *
 * The awkward case is two cots crying at once.  The first cry to start
 * holds the lower number, and it can still be running when the second
 * one ends -- so the record of the second cry is written first and
 * holds the higher number, and the file of records is not in number
 * order.  A record must still be sent.
 *
 * The real log is not touched: the test runs on its own file with its
 * own counters, and everything is put back afterwards. */
bool crynet_log_selftest(void)
{
    if (!s_ready || s_lock == NULL) {
        /* Not a failed proof: there is no log to prove anything about.
         * The flash could not be mounted, so the hub is keeping no
         * record at all -- a different fault, and the owner should be
         * told which one it is. */
        ESP_LOGE(TAG, "the self-test cannot run: the log is not available, "
                      "so the hub is keeping no record");
        return false;
    }
    xSemaphoreTake(s_lock, portMAX_DELAY);

    /* Put the real state away.  The counters file's name goes with it:
     * taking a record number rewrites the counters, and a test must not
     * write over the real ones, not even for a moment. */
    const int saved_fd = s_fd;
    const log_meta_t saved_meta = s_meta;
    const bool saved_ready = s_ready;
    const char *const saved_meta_path = s_meta_path;
    open_slot_t saved_open[OPEN_SLOTS];
    memcpy(saved_open, s_open, sizeof(saved_open));

    bool ok = false;
    int fd = open(SELFTEST_PATH, O_RDWR | O_CREAT | O_TRUNC, 0644);
    if (fd < 0) {
        ESP_LOGE(TAG, "the self-test cannot open its scratch file: %s",
                 strerror(errno));
        goto done;
    }
    {
        uint32_t head[4] = { EVENTS_MAGIC, FORMAT_VER, REC_SIZE, 0 };
        if (write(fd, head, HEADER_SIZE) != (ssize_t)HEADER_SIZE) {
            ESP_LOGE(TAG, "the self-test cannot start its scratch file");
            close(fd);
            goto done;
        }
    }

    /* Fresh state on the scratch file. */
    s_meta_path = SELFTEST_META_PATH;
    s_fd = fd;
    s_ready = true;
    memset(&s_meta, 0, sizeof(s_meta));
    s_meta.magic = META_MAGIC;
    s_meta.next_seq = 1;
    s_meta.uploaded_seq = 0;
    memset(s_open, 0, sizeof(s_open));

    {
        crynet_log_event_t batch[8];

        /* Two cots.  The first cry starts first and is still running
         * when the second one ends. */
        const uint32_t long_cry = reserve_locked(CRYNET_LOG_CRY, 1);
        const uint32_t short_cry = reserve_locked(CRYNET_LOG_CRY, 2);
        if (long_cry == 0 || short_cry == 0) {
            ESP_LOGE(TAG, "the self-test could not take a record number");
            goto restore;
```

```
        }
        commit_locked(short_cry, 5000, 98, 2, 300, 120);

        /* The uploader runs now, while the long cry is still going. */
        uint32_t mark = 0;
        int n = 0;
        pending_t p = collect_locked(s_meta.uploaded_seq, batch, 8);
        mark = watermark_locked(p.last_seq, p.stuck);
        n = p.count;
        bool seen = false;
        for (int i = 0; i < n; i++) {
            if (batch[i].seq == short_cry) {
                seen = true;
            }
        }
        if (!seen) {
            ESP_LOGE(TAG, "the self-test failed: the cry that finished "
                          "first was not sent");
            goto restore;
        }
        /* The claim must not have passed the cry still running. */
        if (mark >= long_cry) {
            ESP_LOGE(TAG, "the self-test failed: the hub claimed #%lu as "
                          "sent while #%lu was still running",
                     (unsigned long)mark, (unsigned long)long_cry);
            goto restore;
        }
        s_meta.uploaded_seq = mark;

        /* The long cry ends now, and its record must go out too. */
        commit_locked(long_cry, 60000, 99, 20, 900, 400);
        p = collect_locked(s_meta.uploaded_seq, batch, 8);
        seen = false;
        for (int i = 0; i < p.count; i++) {
            if (batch[i].seq == long_cry) {
                seen = true;
            }
        }
        if (!seen) {
            ESP_LOGE(TAG, "the self-test failed: the cry that finished "
                          "late was never sent");
            goto restore;
        }
        mark = watermark_locked(p.last_seq, p.stuck);
        s_meta.uploaded_seq = mark;

        /* And nothing is left behind. */
        p = collect_locked(s_meta.uploaded_seq, batch, 8);
        if (p.count != 0) {
            ESP_LOGE(TAG, "the self-test failed: %d records were left "
                          "behind", p.count);
            goto restore;
        }
        ok = true;
    }

restore:
    /* Put the real state back, exactly as it was. */
    s_fd = saved_fd;
    s_meta = saved_meta;
    s_ready = saved_ready;
    s_meta_path = saved_meta_path;
    memcpy(s_open, saved_open, sizeof(s_open));
    if (fd >= 0) {
        close(fd);
        unlink(SELFTEST_PATH);
        unlink(SELFTEST_META_PATH);
    }
done:
    xSemaphoreGive(s_lock);
    if (ok) {
        ESP_LOGW(TAG, "logtest: PASS -- a cry that finishes out of order "
                      "is still sent");
    }
    return ok;
}
```

firmware/hub_infer/components/crynet_clock/include/crynet_clock.h

66 lines · 2,544 bytes

```
/*
 * CryNet hub — the wall clock.
 *
 * The hub records every cry it hears, and a record without a time is not
 * much of a record: the whole point of the dashboard is that a caregiver
 * can see that the baby was up at three in the morning, in that cot,
 * most nights this week.  So the hub needs to know what time it is.
 *
 * It has no real-time clock and no battery-backed one.  It gets the time
 * from the network (SNTP) when it is on the network, and it keeps the
 * time it was last given for as long as it is running.  Two rules make
 * that work:
 *
 *   - The hub's own monotonic uptime (the FreeRTOS tick, which never
 *     runs backwards and is not affected by anything else) is the
 *     authority for *how long* it has been running.  The wall clock is
 *     stored as a pair -- "at this uptime, the time was that" -- so a
 *     correction from the network moves the wall clock without moving
 *     the uptime, and no record is ever stamped twice.
 *
 *   - When the hub is restarted, the last time it knew is written to
 *     NVS and read back at boot.  The clock then starts from that time
 *     and runs on the uptime again.  It is wrong by however long the hub
 *     was switched off, which is the price of not having a battery, and
 *     the network corrects it within a minute or two of connecting.
 *
 * The one thing it cannot do is invent a time it never had: before the
 * first correction of a boot, the clock reads as "unknown" and the
 * records written in that window are marked as untimed.  The dashboard
 * counts those and does not place them on a timeline.
 */

#ifndef CRYNET_CLOCK_H
#define CRYNET_CLOCK_H

#include <stdbool.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

/* Start the clock.  It knows nothing until it is given a time. */
void crynet_clock_init(void);

/* Set the clock to a trusted time, now.  The network does this. */
void crynet_clock_set(uint64_t epoch_ms);

/* Start the clock from a stored time, at boot.  The time is a little
 * stale -- it is as old as the hub's last contact with the network. */
void crynet_clock_restore(uint64_t epoch_ms);

/* Whether the clock knows what time it is. */
bool crynet_clock_known(void);

/* The time now, in milliseconds since the epoch, or 0 when unknown. */
uint64_t crynet_clock_epoch_ms(void);

/* Milliseconds since the hub started.  Always right. */
uint64_t crynet_clock_uptime_ms(void);

#ifdef __cplusplus
}
#endif

#endif /* CRYNET_CLOCK_H */
```

firmware/hub_infer/components/crynet_clock/crynet_clock.c

69 lines · 1,794 bytes

```
/*
 * CryNet hub — the wall clock.  See include/crynet_clock.h for why it
 * is built the way it is.
 */

#include <stdint.h>

#include "esp_log.h"
#include "esp_timer.h"

```

```c
#include "crynet_clock.h"

static const char *TAG = "clock";

/* The pair that makes the wall clock: "at this uptime, the time was
 * that".  Both are zero when the clock knows nothing. */
static uint64_t s_base_epoch_ms = 0;
static uint64_t s_base_uptime_ms = 0;
static bool     s_known = false;

void crynet_clock_init(void)
{
    s_base_epoch_ms = 0;
    s_base_uptime_ms = 0;
    s_known = false;
    ESP_LOGI(TAG, "the clock starts empty; it is told the time by the network");
}

void crynet_clock_set(uint64_t epoch_ms)
{
    s_base_epoch_ms = epoch_ms;
    s_base_uptime_ms = esp_timer_get_time() / 1000;
    s_known = true;
    ESP_LOGI(TAG, "the clock is set to %llu ms since the epoch "
                  "(uptime %llu ms)",
             (unsigned long long)epoch_ms, (unsigned long long)s_base_uptime_ms);
}

void crynet_clock_restore(uint64_t epoch_ms)
{
    if (epoch_ms == 0) {
        return;
    }
    s_base_epoch_ms = epoch_ms;
    s_base_uptime_ms = 0;   /* the uptime of this boot starts at zero */
    s_known = true;
    ESP_LOGI(TAG, "the clock starts from the stored time %llu ms since the "
                  "epoch; the network will correct it",
             (unsigned long long)epoch_ms);
}

bool crynet_clock_known(void)
{
    return s_known;
}

uint64_t crynet_clock_uptime_ms(void)
{
    return esp_timer_get_time() / 1000;
}

uint64_t crynet_clock_epoch_ms(void)
{
    if (!s_known) {
        return 0;
    }
    const uint64_t now = esp_timer_get_time() / 1000;
    return s_base_epoch_ms + (now - s_base_uptime_ms);
}
```

firmware/hub_infer/components/crynet_net/include/crynet_net.h

127 lines · 5,444 bytes

```c
/*
 * CryNet hub — the network, when there is one.
 *
 * The rule this component lives by: the hub must work exactly as it
 * does now with no internet at all.  So nothing here is on the path
 * between a cot's microphone and the caregiver's wrist.  The log is
 * written by the audio loop whether or not this component ever runs;
 * this component only reads the log afterwards and posts it somewhere.
 * If the network is absent, misconfigured, or broken, the only thing
 * that changes is that the records stay on the hub.
 *
 * The one real hazard is the radio.  The cot nodes and the hub talk
 * ESP-NOW on channel 6, fixed -- see node_link.c.  Joining a Wi-Fi
 * network moves the radio to the router's channel, and if that is not
 * 6 the cot link dies: the wrist goes silent while the dashboard looks
 * healthy.  So this component checks the channel the moment it
 * associates, and if it is not 6 it drops the connection immediately,
 * puts the radio back on 6, and says what to do about it.  The router's
```

```
 * 2.4 GHz channel has to be set to 6.  That is a one-time change in the
 * router's admin page, and it is the price of one radio doing two jobs.
 *
 * What it does when the network is there:
 *   - asks an NTP server what time it is, and tells crynet_clock, so
 *     that records get real timestamps;
 *   - posts the records that have not been sent, oldest first, in
 *     batches, and marks them sent only when the dashboard says it has
 *     them;
 *   - trims the log once it has been emptied.
 */

#ifndef CRYNET_NET_H
#define CRYNET_NET_H

#include <stdbool.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

#define CRYNET_NET_URL_MAX  160
#define CRYNET_NET_SSID_MAX 33

/* How many cots the hub can report on. */
#define CRYNET_NET_MAX_NODES 4

/* What the hub knows about one cot, right now.  The cots have no
 * network of their own -- they talk to the hub over its radio -- so the
 * hub is the only thing that can tell the dashboard who is in the room.
 */
typedef struct {
    uint8_t cot;      /* the cot number, 1-based */
    bool    online;   /* the hub has heard from it lately */
    bool    deaf;     /* the hub cannot get a window from it */
    int8_t  rssi;     /* the telemetry signal strength in dBm; 0 = unknown */
} crynet_net_node_t;

typedef struct {
    char     ssid[CRYNET_NET_SSID_MAX];
    char     url[CRYNET_NET_URL_MAX];
    bool     wifi_up;          /* associated and has an address */
    bool     channel_ok;       /* the router is on the cots' channel */
    uint8_t  channel;          /* the channel the radio is on now */
    bool     clock_known;
    uint32_t uploaded;         /* records sent this boot */
    uint32_t batches;          /* successful posts this boot */
    uint32_t failures;         /* consecutive failures right now */
    uint32_t last_ok_epoch_s;  /* when the dashboard last answered */
    char     last_error[80];
} crynet_net_status_t;

/* Bring up the station and the uploader.  The radio must already be
 * started by node_link_start() -- this component shares it, it does
 * not own it.  Safe to call when there is no network configured: it
 * starts nothing and costs nothing. */
bool crynet_net_start(void);

/* Where to post.  A full URL, e.g.
 *   http://192.168.1.50:8787/api/v1/events
 * Stored in NVS, so it survives a reflash.  An empty string turns
 * uploading off but leaves the clock and the log alone. */
bool crynet_net_set_url(const char *url);

/* Which network to join.  Stored in NVS. */
bool crynet_net_set_wifi(const char *ssid, const char *password);

/* An optional shared secret, sent as X-CryNet-Token. */
bool crynet_net_set_token(const char *token);

/* The name the caregiver sees on the dashboard. */
bool crynet_net_set_name(const char *name);

void crynet_net_status(crynet_net_status_t *out);

/* Ask the uploader to try now rather than wait for its next turn. */
void crynet_net_upload_now(void);

/* Say who is here.  The hub hands over what it knows about each cot --
 * whether it is answering, whether it has gone deaf, how strong its
 * radio is -- and this component posts it to the dashboard, whether or
 * not there is anything in the log to send.
 *
 * That is the point of it: a hub that has been quiet for a week is
```

```
 * still a hub that is working, and a dashboard that only learns of a
 * hub when a cry happens cannot say whether the silence is a quiet baby
 * or a dead hub.  The copy is taken here, so the caller's array does
 * not have to outlive the call. */
void crynet_net_set_presence(const crynet_net_node_t *nodes, uint8_t count);

/* Name the thing that decides whether a cot's radio traffic is in
 * flight -- a window being pulled from a cot, say -- and the presence
 * will wait for it rather than take the air from under it.  The link
 * stage registers this, because the link stage is the one that knows.
 * The wait is capped inside the network stage, so a guard that is
 * stuck saying "busy" can delay a report but never stop the reports:
 * the dashboard may be a few seconds behind, never absent.  Pass NULL
 * to remove one. */
void crynet_net_set_radio_guard(bool (*busy)(void));

/* This hub's identity on the dashboard, e.g. "hub-7c9ebd12". */
const char *crynet_net_device_id(void);

#ifdef __cplusplus
}
#endif

#endif /* CRYNET_NET_H */
```

firmware/hub_infer/components/crynet_net/crynet_net.c

896 lines · 32,986 bytes

```
/*
 * CryNet hub — the network.  See include/crynet_net.h for the rule it
 * lives by: nothing here may stand between a cry and the wrist.
 *
 * One task, ticking once a second, doing whatever is next: join the
 * network, ask the time, post the records.  It is deliberately not
 * clever.  A supervisor that ticks is easy to reason about when the
 * radio is doing three jobs at once, and a second of latency means
 * nothing to a record that is already safe in flash.
 */

#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/time.h>

#include "esp_crt_bundle.h"
#include "esp_event.h"
#include "esp_heap_caps.h"
#include "esp_http_client.h"
#include "esp_log.h"
#include "esp_mac.h"
#include "esp_netif.h"
#include "esp_netif_sntp.h"
#include "esp_timer.h"
#include "esp_wifi.h"
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "freertos/semphr.h"
#include "freertos/task.h"
#include "nvs.h"
#include "nvs_flash.h"

#include "crynet_clock.h"
#include "crynet_log.h"
#include "crynet_net.h"

static const char *TAG = "crynet";

/* ------------------------------------------------------------------
 * The network to join, and where to post.
 *
 * These are the defaults baked into the image so that a freshly
 * flashed hub joins the owner's network with no further steps.  The
 * password is in the source in plain text: this is a nursery monitor
 * on a home network, and the alternative -- a provisioning flow the
 * owner has to run before the hub works -- costs more than it protects.
 * Anything set later with the console's `net` commands is stored in
 * NVS and takes precedence, and NVS is not touched by reflashing.
 *
 * The URL is empty on purpose.  The dashboard runs on the owner's own
```

```
 * machine and its address is not knowable at build time; the dashboard
 * prints the exact `net set ...` line to type once it is running.
 * ------------------------------------------------------------------- */
#define DEFAULT_SSID "Amazyn"
#define DEFAULT_PASS "me_mine_me"
#define DEFAULT_URL  ""
#define DEFAULT_NAME "Nursery hub"

#define CRYNET_FW_VERSION "2.0-dashboard"

/* The cots' channel.  node_link.c pins the radio here; joining a
 * router on any other channel would break the link. */
#define COT_CHANNEL 6

#define NVS_NAMESPACE "crynet"

#define BATCH_RECORDS 32
#define BATCHES_PER_TURN 32
#define JSON_CAP 12288
#define RESP_CAP 256

#define UPLOAD_EVERY_S 30
#define RECONNECT_EVERY_S 15
#define WRONG_CHANNEL_RETRY_S 300

/* How often the hub says who is here.  The owner asked for the hub's
 * connection to be checked "as often as possible", and then added the
 * constraint that decides it: nothing may cost the cots anything.
 * Each report is a TLS handshake, which takes the radio for a moment,
 * and the cots share that radio -- so the interval is thirty seconds,
 * and the presence also *yields*: while a window is in flight from a
 * cot, the report waits for it to finish (see the guard below, and
 * PRESENCE_MAX_WAIT_US).  Thirty seconds is 2880 requests a day
 * against a free tier's hundred thousand. */
#define PRESENCE_EVERY_S 30

/* How long the presence will wait for a cot's window to finish before
 * it posts anyway.  The wait keeps a report from landing in the middle
 * of a burst; the cap keeps a busy radio from delaying the reports
 * into a silence the dashboard would read as a hub gone away.  The
 * worst gap between reports is therefore PRESENCE_EVERY_S plus this --
 * fifty seconds -- which stays inside the dashboard's own idea of how
 * long a hub may go without speaking (three reports missed). */
#define PRESENCE_MAX_WAIT_US (20LL * 1000000LL)

#define BIT_IP    (1u << 0)
#define BIT_LOST  (1u << 1)

static char s_ssid[CRYNET_NET_SSID_MAX];
static char s_pass[64];
static char s_url[CRYNET_NET_URL_MAX];
static char s_token[80];
static char s_name[48];
static char s_device[16];

static EventGroupHandle_t s_events;
static bool     s_started = false;
static bool     s_connected = false;
static bool     s_wrong_channel = false;
static bool     s_config_dirty = false;
static bool     s_force_upload = false;
static bool     s_sntp_up = false;
static bool     s_channel_reported = false;
static uint8_t  s_channel = 0;

static int64_t  s_next_connect_us = 0;
static int64_t  s_next_upload_us = 0;
static int64_t  s_channel_retry_us = 0;

static crynet_net_status_t s_st;
static char s_resp[RESP_CAP];
static int  s_resp_len = 0;

static char *s_json = NULL;
static crynet_log_event_t *s_recs = NULL;

/* Who is here.  The cots have no network of their own, so the hub is
 * the only one that can say.  The copy is taken under a lock because
 * the caller hands it over from another task. */
static SemaphoreHandle_t s_presence_lock = NULL;
static crynet_net_node_t s_nodes[CRYNET_NET_MAX_NODES];
static uint8_t s_node_count = 0;
static bool s_presence_known = false;
static int64_t s_next_presence_us = 0;
/* How long the current report has been waiting for a cot's window, 0
```

```
 * if it is not waiting.  Touched only under s_presence_lock. */
static bool (*s_radio_busy)(void) = NULL;
static int64_t s_presence_wait_since_us = 0;

void crynet_net_set_radio_guard(bool (*busy)(void))
{
    s_radio_busy = busy;
}

/* ------------------------------------------------------------------ */
/* Configuration in NVS.                                               */

static void load_str(nvs_handle_t h, const char *key, char *dst, size_t cap,
                     const char *fallback)
{
    size_t len = cap;
    if (nvs_get_str(h, key, dst, &len) != ESP_OK || dst[0] == 0) {
        snprintf(dst, cap, "%s", fallback);
    }
    dst[cap - 1] = 0;
}

static bool store_str(const char *key, const char *value)
{
    nvs_handle_t h;
    if (nvs_open(NVS_NAMESPACE, NVS_READWRITE, &h) != ESP_OK) {
        return false;
    }
    const esp_err_t err = nvs_set_str(h, key, value);
    const esp_err_t commit = nvs_commit(h);
    nvs_close(h);
    return err == ESP_OK && commit == ESP_OK;
}

static void load_config(void)
{
    nvs_handle_t h;
    if (nvs_open(NVS_NAMESPACE, NVS_READONLY, &h) != ESP_OK) {
        snprintf(s_ssid, sizeof(s_ssid), "%s", DEFAULT_SSID);
        snprintf(s_pass, sizeof(s_pass), "%s", DEFAULT_PASS);
        snprintf(s_url, sizeof(s_url), "%s", DEFAULT_URL);
        snprintf(s_name, sizeof(s_name), "%s", DEFAULT_NAME);
        s_token[0] = 0;
        return;
    }
    load_str(h, "ssid", s_ssid, sizeof(s_ssid), DEFAULT_SSID);
    load_str(h, "pass", s_pass, sizeof(s_pass), DEFAULT_PASS);
    load_str(h, "url", s_url, sizeof(s_url), DEFAULT_URL);
    load_str(h, "name", s_name, sizeof(s_name), DEFAULT_NAME);
    load_str(h, "token", s_token, sizeof(s_token), "");
    nvs_close(h);
}

/* ------------------------------------------------------------------ */
/* The radio.  Shared with node_link, which started it.                */

static void radio_events(void *arg, esp_event_base_t base, int32_t id,
                         void *data)
{
    if (base == WIFI_EVENT && id == WIFI_EVENT_STA_CONNECTED) {
        uint8_t primary = 0;
        wifi_second_chan_t second = WIFI_SECOND_CHAN_NONE;
        esp_wifi_get_channel(&primary, &second);
        s_channel = primary;
        if (primary != COT_CHANNEL) {
            /* The router has moved the radio off the cots' channel.
             * Every ESP-NOW frame from here would be lost, so leave
             * now rather than look connected while the wrist is deaf. */
            s_wrong_channel = true;
            s_channel_retry_us = esp_timer_get_time() +
                                 (int64_t)WRONG_CHANNEL_RETRY_S * 1000000;
            esp_wifi_disconnect();
            return;
        }
        s_wrong_channel = false;
        s_channel_reported = false;
        ESP_LOGI(TAG, "joined \"%s\" on channel %u -- the cots' channel, "
                      "so the link is safe", s_ssid, primary);
    } else if (base == WIFI_EVENT && id == WIFI_EVENT_STA_DISCONNECTED) {
        s_connected = false;
        xEventGroupClearBits(s_events, BIT_IP);
        xEventGroupSetBits(s_events, BIT_LOST);
    } else if (base == IP_EVENT && id == IP_EVENT_STA_GOT_IP) {
        s_connected = true;
        xEventGroupSetBits(s_events, BIT_IP);
```

```
        const ip_event_got_ip_t *ev = (const ip_event_got_ip_t *)data;
        ESP_LOGI(TAG, "address " IPSTR, IP2STR(&ev->ip_info.ip));
    }
}

static void apply_wifi_config(void)
{
    wifi_config_t cfg;
    memset(&cfg, 0, sizeof(cfg));
    /* The radio's fields are fixed-size and not necessarily terminated:
     * a 32-character name fills its field exactly.  The buffers were
     * zeroed above, so a copy of at most the field's length is right. */
    size_t n = strnlen(s_ssid, sizeof(cfg.sta.ssid));
    memcpy(cfg.sta.ssid, s_ssid, n);
    n = strnlen(s_pass, sizeof(cfg.sta.password));
    memcpy(cfg.sta.password, s_pass, n);
    cfg.sta.threshold.authmode = s_pass[0] ? WIFI_AUTH_WPA2_PSK : WIFI_AUTH_OPEN;
    cfg.sta.sae_pwe_h2e = WPA3_SAE_PWE_BOTH;
    esp_wifi_set_config(WIFI_IF_STA, &cfg);
}

/* Put the radio back where the cots can hear it. */
static void restore_cot_channel(void)
{
    uint8_t primary = 0;
    wifi_second_chan_t second = WIFI_SECOND_CHAN_NONE;
    esp_wifi_get_channel(&primary, &second);
    if (primary != COT_CHANNEL) {
        esp_wifi_set_channel(COT_CHANNEL, WIFI_SECOND_CHAN_NONE);
        ESP_LOGW(TAG, "radio put back on channel %u for the cots", COT_CHANNEL);
    }
}

/* ------------------------------------------------------------------ */
/* The clock.                                                         */

static void on_time_sync(struct timeval *tv)
{
    if (tv == NULL || tv->tv_sec < 1700000000) {
        return;   /* not a believable answer */
    }
    const uint64_t ms = (uint64_t)tv->tv_sec * 1000u +
                        (uint64_t)tv->tv_usec / 1000u;
    crynet_clock_set(ms);
}

static void start_sntp(void)
{
    if (s_sntp_up) {
        return;
    }
    /* Two servers, so one being unreachable does not leave the hub
     * without a clock.  The list has to go through ESP_SNTP_SERVER_LIST:
     * a bare brace pair would read as two macro arguments, not one. */
    esp_sntp_config_t cfg = ESP_NETIF_SNTP_DEFAULT_CONFIG_MULTIPLE(
        2, ESP_SNTP_SERVER_LIST("pool.ntp.org", "time.google.com"));
    cfg.sync_cb = on_time_sync;
    cfg.smooth_sync = false;
    if (esp_netif_sntp_init(&cfg) == ESP_OK) {
        s_sntp_up = true;
        ESP_LOGI(TAG, "asking the network what time it is");
    }
}

/* ------------------------------------------------------------------ */
/* Posting the records.                                               */

static const char *kind_name(uint8_t kind)
{
    switch (kind) {
    case CRYNET_LOG_CRY:   return "cry";
    case CRYNET_LOG_DEAF:  return "deaf";
    case CRYNET_LOG_HEARD: return "heard";
    case CRYNET_LOG_BOOT:  return "boot";
    case CRYNET_LOG_TEST:  return "test";
    default:               return "cry";
    }
}

/* A string as JSON needs it.  Only the hub's name can contain anything
 * awkward -- the owner types it at the console -- but a name with a
 * quote in it would otherwise produce a body the dashboard cannot read,
 * and the records would never be accepted. */
static void json_escape(char *out, size_t cap, const char *in)
{
```

```
    size_t o = 0;
    for (const unsigned char *p = (const unsigned char *)in; *p; p++) {
        char buf[8];
        int len;
        if (*p == '"' || *p == '\\') {
            buf[0] = '\\';
            buf[1] = (char)*p;
            len = 2;
        } else if (*p < 0x20) {
            len = snprintf(buf, sizeof buf, "\\u%04x", (unsigned)*p);
        } else {
            buf[0] = (char)*p;
            len = 1;
        }
        if (o + (size_t)len >= cap) {
            break;
        }
        memcpy(out + o, buf, (size_t)len);
        o += (size_t)len;
    }
    out[o] = 0;
}

/* Hand-built JSON: ESP-IDF 6.1 ships no cJSON, and the shape is fixed
 * and small enough that a parser would be more code than this. */
static int build_batch(const crynet_log_event_t *recs, int n)
{
    char name[sizeof(s_name) * 6 + 1];
    json_escape(name, sizeof name, s_name);
    int used = snprintf(s_json, JSON_CAP,
                        "{\"device\":\"%s\",\"hub\":{\"fw\":\"%s\","
                        "\"name\":\"%s\"},\"events\":[",
                        s_device, CRYNET_FW_VERSION, name);
    if (used < 0 || used >= JSON_CAP) {
        return -1;
    }
    for (int i = 0; i < n; i++) {
        const crynet_log_event_t *r = &recs[i];
        const uint32_t dur_s = (r->duration_ms + 500u) / 1000u;
        const uint32_t end = r->epoch_s ? r->epoch_s + dur_s : 0;
        const int add = snprintf(
            s_json + used, JSON_CAP - used,
            "%s{\"seq\":%lu,\"kind\":\"%s\",\"cot\":%u,"
            "\"start\":%lu,\"end\":%lu,\"duration_s\":%lu,"
            "\"peak_conf\":%u,\"windows\":%u,"
            "\"peak_rms\":%.4f,\"mean_rms\":%.4f,"
            "\"boot\":%lu,\"uptime_ms\":%lu}",
            i ? "," : "",
            (unsigned long)r->seq, kind_name(r->kind), (unsigned)r->cot,
            (unsigned long)r->epoch_s, (unsigned long)end,
            (unsigned long)dur_s,
            (unsigned)r->peak_conf, (unsigned)r->windows,
            r->peak_rms_milli / 1000.0, r->mean_rms_milli / 1000.0,
            (unsigned long)r->boot_id, (unsigned long)r->uptime_ms);
        if (add < 0 || add >= JSON_CAP - used) {
            return -1;   /* would not fit; send fewer next time */
        }
        used += add;
    }
    if (used + 3 >= JSON_CAP) {
        return -1;
    }
    s_json[used++] = ']';
    s_json[used++] = '}';
    s_json[used] = 0;
    return used;
}

static esp_err_t http_event(esp_http_client_event_t *ev)
{
    if (ev->event_id == HTTP_EVENT_ON_DATA && ev->data_len > 0) {
        int room = RESP_CAP - 1 - s_resp_len;
        if (room > 0) {
            int take = ev->data_len < room ? ev->data_len : room;
            memcpy(s_resp + s_resp_len, ev->data, take);
            s_resp_len += take;
            s_resp[s_resp_len] = 0;
        }
    }
    return ESP_OK;
}

/* Post a body to an address.  Both the records and the presence go
 * this way: the same token, the same client, the same answer to a
 * refusal. */
```

```
static bool http_post(const char *url, const char *body, int len)
{
    s_resp_len = 0;
    s_resp[0] = 0;

    esp_http_client_config_t cfg = {
        .url = url,
        .method = HTTP_METHOD_POST,
        .timeout_ms = 10000,
        .event_handler = http_event,
        .buffer_size = 2048,
        .buffer_size_tx = 2048,
        .keep_alive_enable = true,
#ifdef CONFIG_MBEDTLS_CERTIFICATE_BUNDLE
        .crt_bundle_attach = esp_crt_bundle_attach,
#endif
    };
    esp_http_client_handle_t client = esp_http_client_init(&cfg);
    if (client == NULL) {
        snprintf(s_st.last_error, sizeof(s_st.last_error),
                 "cannot start the HTTP client");
        return false;
    }
    esp_http_client_set_header(client, "Content-Type", "application/json");
    if (s_token[0]) {
        esp_http_client_set_header(client, "X-CryNet-Token", s_token);
    }
    esp_http_client_set_post_field(client, body, len);

    const esp_err_t err = esp_http_client_perform(client);
    const int status = esp_http_client_get_status_code(client);
    esp_http_client_cleanup(client);

    if (err != ESP_OK) {
        /* The HTTP client answers a 401 by trying to meet the server's
         * authentication challenge, and with nothing to answer it with
         * it returns ESP_ERR_NOT_SUPPORTED instead of the status.  So
         * the status is put in the message here: "HTTP 401" says what
         * actually happened, where the bare error name says nothing. */
        if (status > 0) {
            snprintf(s_st.last_error, sizeof(s_st.last_error),
                     "HTTP %d (%s)", status, esp_err_to_name(err));
        } else {
            snprintf(s_st.last_error, sizeof(s_st.last_error),
                     "%s", esp_err_to_name(err));
        }
        return false;
    }
    if (status < 200 || status >= 300) {
        snprintf(s_st.last_error, sizeof(s_st.last_error),
                 "HTTP %d: %.48s", status, s_resp);
        return false;
    }
    return true;
}

static bool post_batch(const crynet_log_event_t *recs, int n)
{
    const int len = build_batch(recs, n);
    if (len <= 0) {
        snprintf(s_st.last_error, sizeof(s_st.last_error),
                 "a batch would not fit in the buffer");
        return false;
    }
    return http_post(s_url, s_json, len);
}

static void upload_turn(void)
{
    int batches = 0;
    for (; batches < BATCHES_PER_TURN; batches++) {
        /* `mark` is the number the log says may be claimed as sent once
         * this batch has gone.  It is not simply the highest number in
         * the batch: two cots crying at once can end in the opposite
         * order to the one they started in, and claiming past a record
         * that has not gone would lose it. */
        uint32_t mark = 0;
        const int n = crynet_log_pending(s_recs, BATCH_RECORDS, &mark);
        if (n <= 0) {
            break;
        }
        if (!post_batch(s_recs, n)) {
            s_st.failures++;
            ESP_LOGW(TAG, "upload failed (%lu in a row): %s",
                     (unsigned long)s_st.failures, s_st.last_error);
```

```
            break;
        }
        crynet_log_mark_uploaded(mark);
        s_st.uploaded += (uint32_t)n;
        s_st.batches++;
        s_st.failures = 0;
        s_st.last_error[0] = 0;
        s_st.last_ok_epoch_s = (uint32_t)(crynet_clock_epoch_ms() / 1000u);
        ESP_LOGI(TAG, "sent %d records, up to #%lu (%s)", n,
                 (unsigned long)mark, s_resp);
        crynet_log_save_epoch_ms(crynet_clock_epoch_ms());
        if (n < BATCH_RECORDS) {
            batches++;
            break;
        }
        /* Let the audio loop and the radio have the chip between
         * batches; nothing here is in a hurry. */
        vTaskDelay(pdMS_TO_TICKS(100));
    }
    if (batches > 0) {
        crynet_log_trim();
    }
}

/* ------------------------------------------------------------------ */
/* Saying who is here.                                                */
/* ------------------------------------------------------------------ */

/* The presence goes to the same service as the records, at the same
 * address with the last word changed: the owner types one address, and
 * the hub works out the rest.  If the address is not one this can make
 * sense of, there is nowhere to post to and the hub says so once. */
static bool presence_url(char *out, size_t cap)
{
    const char *slash = strrchr(s_url, '/');
    if (slash == NULL || slash == s_url || slash[1] == 0) {
        out[0] = 0;
        return false;
    }
    snprintf(out, cap, "%.*s/presence", (int)(slash - s_url), s_url);
    return true;
}

static int build_presence(const crynet_net_node_t *nodes, uint8_t count)
{
    char name[sizeof(s_name) * 6 + 1];
    json_escape(name, sizeof name, s_name);
    int used = snprintf(s_json, JSON_CAP,
                        "{\"device\":\"%s\",\"hub\":{\"fw\":\"%s\","
                        "\"name\":\"%s\"},\"nodes\":[",
                        s_device, CRYNET_FW_VERSION, name);
    if (used < 0 || used >= JSON_CAP) {
        return -1;
    }
    for (uint8_t i = 0; i < count; i++) {
        const crynet_net_node_t *n = &nodes[i];
        const int add = snprintf(
            s_json + used, JSON_CAP - used,
            "%s{\"cot\":%u,\"online\":%s,\"deaf\":%s,\"rssi\":%d}",
            i ? "," : "",
            (unsigned)n->cot,
            n->online ? "true" : "false",
            n->deaf ? "true" : "false",
            (int)n->rssi);
        if (add < 0 || add >= JSON_CAP - used) {
            return -1;
        }
        used += add;
    }
    if (used + 3 >= JSON_CAP) {
        return -1;
    }
    s_json[used++] = ']';
    s_json[used++] = '}';
    s_json[used] = 0;
    return used;
}

static bool post_presence(void)
{
    static crynet_net_node_t nodes[CRYNET_NET_MAX_NODES];
    uint8_t count = 0;

    /* Take a copy under the lock: the caller may be handing over a
     * newer set of cots while this is being built. */
```

```
    if (s_presence_lock != NULL &&
        xSemaphoreTake(s_presence_lock, pdMS_TO_TICKS(50)) == pdTRUE) {
        count = s_node_count;
        memcpy(nodes, s_nodes, sizeof(nodes[0]) * count);
        xSemaphoreGive(s_presence_lock);
    } else {
        return false;
    }

    char url[CRYNET_NET_URL_MAX + 16];
    if (!presence_url(url, sizeof(url))) {
        return false;
    }
    const int len = build_presence(nodes, count);
    if (len <= 0) {
        snprintf(s_st.last_error, sizeof(s_st.last_error),
                 "the presence would not fit in the buffer");
        return false;
    }
    if (!http_post(url, s_json, len)) {
        ESP_LOGW(TAG, "presence not posted: %s", s_st.last_error);
        return false;
    }
    ESP_LOGI(TAG, "presence posted: %u cot(s)", (unsigned)count);
    return true;
}

/* ------------------------------------------------------------------ */
/* The supervisor.                                                    */

static void net_task(void *arg)
{
    (void)arg;
    for (;;) {
        vTaskDelay(pdMS_TO_TICKS(1000));
        const int64_t now = esp_timer_get_time();

        if (s_config_dirty) {
            s_config_dirty = false;
            esp_wifi_disconnect();
            apply_wifi_config();
            s_next_connect_us = 0;
            s_wrong_channel = false;
            s_channel_retry_us = 0;
        }

        if (s_ssid[0] == 0) {
            continue;   /* no network configured: do nothing at all */
        }

        if (s_wrong_channel) {
            restore_cot_channel();
            if (!s_channel_reported) {
                s_channel_reported = true;
                ESP_LOGE(TAG, "the router \"%s\" is on channel %u, but the "
                              "cots talk on channel %u. Left the network so "
                              "the cots keep working. Set the router's "
                              "2.4 GHz channel to %u and the hub will join "
                              "by itself.",
                         s_ssid, s_channel, COT_CHANNEL, COT_CHANNEL);
                snprintf(s_st.last_error, sizeof(s_st.last_error),
                         "router is on channel %u, cots need %u",
                         s_channel, COT_CHANNEL);
            }
            if (now < s_channel_retry_us) {
                continue;
            }
            s_wrong_channel = false;
            s_channel_retry_us = 0;
        }

        if (!s_connected) {
            if (now >= s_next_connect_us) {
                s_next_connect_us = now + (int64_t)RECONNECT_EVERY_S * 1000000;
                esp_wifi_connect();
            }
            continue;
        }

        start_sntp();

        if (s_url[0] == 0) {
            continue;   /* clock only; nothing to post to */
        }

```

```
        /* The presence goes every thirty seconds whatever else is
         * happening.  A hub that is working but has nothing to say still
         * has to be able to say that it is there -- and it has to say
         * it often enough that a hub which has just been switched off
         * is gone from the dashboard before anyone goes looking.
         *
         * But a cot's window outranks it.  A burst takes the air, and
         * while it is on it the Wi-Fi receiver hears nothing -- not the
         * burst's own fragments, and nothing else.  So while the link
         * stage says a burst is in flight, the report waits for it.
         * The wait is capped, because a hub that has been switched off
         * must still be reported gone: a report may be twenty seconds
         * late, it may not be missing. */
        bool presence_due = false;
        if (s_presence_lock != NULL &&
            xSemaphoreTake(s_presence_lock, pdMS_TO_TICKS(50)) == pdTRUE) {
            presence_due = s_presence_known && now >= s_next_presence_us;
            if (presence_due && s_radio_busy != NULL && s_radio_busy()) {
                if (s_presence_wait_since_us == 0) {
                    s_presence_wait_since_us = now;
                    presence_due = false;
                } else if (now - s_presence_wait_since_us >=
                           PRESENCE_MAX_WAIT_US) {
                    s_presence_wait_since_us = 0;
                    /* waited long enough; post anyway */
                } else {
                    presence_due = false;
                }
            } else {
                s_presence_wait_since_us = 0;
            }
            if (presence_due) {
                s_next_presence_us =
                    now + (int64_t)PRESENCE_EVERY_S * 1000000;
            }
            xSemaphoreGive(s_presence_lock);
        }
        if (presence_due) {
            post_presence();
        }

        if (!s_force_upload && now < s_next_upload_us) {
            continue;
        }
        s_force_upload = false;
        /* Back off when it is failing, so a dashboard that is switched
         * off does not cost a connection attempt every thirty seconds. */
        const uint32_t backoff = s_st.failures > 6 ? 6 : s_st.failures;
        s_next_upload_us = now +
            (int64_t)UPLOAD_EVERY_S * 1000000 * (1 + backoff);
        upload_turn();
    }
}

/* ------------------------------------------------------------------ */
/* Starting and steering.                                             */

bool crynet_net_start(void)
{
    if (s_started) {
        return true;
    }

    uint8_t mac[6] = { 0 };
    esp_read_mac(mac, ESP_MAC_WIFI_STA);
    snprintf(s_device, sizeof(s_device), "hub-%02x%02x%02x%02x",
             mac[2], mac[3], mac[4], mac[5]);

    load_config();
    memset(&s_st, 0, sizeof(s_st));
    snprintf(s_st.ssid, sizeof(s_st.ssid), "%s", s_ssid);
    snprintf(s_st.url, sizeof(s_st.url), "%s", s_url);

    s_events = xEventGroupCreate();
    s_presence_lock = xSemaphoreCreateMutex();
    s_json = heap_caps_malloc(JSON_CAP, MALLOC_CAP_SPIRAM);
    if (s_json == NULL) {
        s_json = malloc(JSON_CAP);
    }
    s_recs = heap_caps_malloc(BATCH_RECORDS * sizeof(crynet_log_event_t),
                              MALLOC_CAP_SPIRAM);
    if (s_recs == NULL) {
        s_recs = malloc(BATCH_RECORDS * sizeof(crynet_log_event_t));
    }
    if (s_events == NULL || s_json == NULL || s_recs == NULL ||
```

```
        s_presence_lock == NULL) {
        ESP_LOGE(TAG, "no memory for the uploader; the log still records");
        return false;
    }

    /* node_link has already initialised the radio for ESP-NOW, and it
     * created the station interface before the radio started -- which is
     * the only order that works (see node_link.c).  So the interface is
     * already there: look it up rather than making a second one. */
    wifi_mode_t mode = WIFI_MODE_NULL;
    if (esp_wifi_get_mode(&mode) != ESP_OK) {
        ESP_LOGE(TAG, "the radio is not up; start node_link first");
        return false;
    }
    esp_netif_t *sta = esp_netif_get_handle_from_ifkey("WIFI_STA_DEF");
    if (sta == NULL) {
        /* It cannot be created here: the interface has to exist before
         * the radio starts, or lwIP never gets it (see node_link.c). */
        ESP_LOGE(TAG, "the station interface is missing; node_link must "
                      "make it before the radio starts");
        return false;
    }
    esp_event_handler_register(WIFI_EVENT, ESP_EVENT_ANY_ID, radio_events, NULL);
    esp_event_handler_register(IP_EVENT, IP_EVENT_STA_GOT_IP, radio_events, NULL);

    apply_wifi_config();
    s_started = true;

    if (s_ssid[0] == 0) {
        ESP_LOGI(TAG, "no network configured; the hub records to flash only");
        return true;
    }

    /* HTTPS posts run their TLS handshake on this task's stack.  Give it
     * extra headroom; the observed overflow was on the separate main task. */
    BaseType_t ok = xTaskCreate(net_task, "crynet", 8192, NULL, 3, NULL);
    if (ok != pdPASS) {
        ESP_LOGE(TAG, "cannot start the uploader; the log still records");
        return false;
    }
    ESP_LOGI(TAG, "%s ready: network \"%s\", dashboard %s",
             s_device, s_ssid, s_url[0] ? s_url : "(not set -- log only)");
    return true;
}

bool crynet_net_set_url(const char *url)
{
    if (url == NULL) {
        return false;
    }
    if (url[0] && strncmp(url, "http://", 7) != 0 &&
        strncmp(url, "https://", 8) != 0) {
        ESP_LOGE(TAG, "the address must start with http:// or https://");
        return false;
    }
    if (strlen(url) >= sizeof(s_url)) {
        ESP_LOGE(TAG, "that address is too long");
        return false;
    }
    snprintf(s_url, sizeof(s_url), "%s", url);
    snprintf(s_st.url, sizeof(s_st.url), "%s", s_url);
    s_next_upload_us = 0;
    s_force_upload = true;
    return store_str("url", s_url);
}

bool crynet_net_set_wifi(const char *ssid, const char *password)
{
    if (ssid == NULL || ssid[0] == 0 || strlen(ssid) >= sizeof(s_ssid)) {
        return false;
    }
    if (password && strlen(password) >= sizeof(s_pass)) {
        return false;
    }
    snprintf(s_ssid, sizeof(s_ssid), "%s", ssid);
    snprintf(s_pass, sizeof(s_pass), "%s", password ? password : "");
    snprintf(s_st.ssid, sizeof(s_st.ssid), "%s", s_ssid);
    s_config_dirty = true;
    const bool a = store_str("ssid", s_ssid);
    const bool b = store_str("pass", s_pass);
    return a && b;
}

bool crynet_net_set_token(const char *token)
{
```

```
    if (token == NULL || strlen(token) >= sizeof(s_token)) {
        return false;
    }
    snprintf(s_token, sizeof(s_token), "%s", token);
    return store_str("token", s_token);
}

bool crynet_net_set_name(const char *name)
{
    if (name == NULL || name[0] == 0 || strlen(name) >= sizeof(s_name)) {
        return false;
    }
    snprintf(s_name, sizeof(s_name), "%s", name);
    return store_str("name", s_name);
}

void crynet_net_status(crynet_net_status_t *out)
{
    if (out == NULL) {
        return;
    }
    *out = s_st;
    out->wifi_up = s_connected;
    out->channel_ok = !s_wrong_channel;
    out->channel = s_channel;
    out->clock_known = crynet_clock_known();
    snprintf(out->ssid, sizeof(out->ssid), "%s", s_ssid);
    snprintf(out->url, sizeof(out->url), "%s", s_url);
}

void crynet_net_upload_now(void)
{
    s_force_upload = true;
    s_next_upload_us = 0;
}

void crynet_net_set_presence(const crynet_net_node_t *nodes, uint8_t count)
{
    if (count > CRYNET_NET_MAX_NODES) {
        count = CRYNET_NET_MAX_NODES;
    }
    if (s_presence_lock == NULL) {
        return;   /* the network was never started; nothing to tell it */
    }
    if (xSemaphoreTake(s_presence_lock, pdMS_TO_TICKS(50)) != pdTRUE) {
        return;
    }

    /* Has anything the caregiver would notice changed?  The signal
     * strength moves constantly and is not one of those things, so it
     * is carried along but not counted as a change. */
    bool changed = (count != s_node_count) || !s_presence_known;
    for (uint8_t i = 0; !changed && i < count && i < s_node_count; i++) {
        changed = nodes[i].cot != s_nodes[i].cot ||
                  nodes[i].online != s_nodes[i].online ||
                  nodes[i].deaf != s_nodes[i].deaf;
    }
    if (nodes != NULL && count > 0) {
        memcpy(s_nodes, nodes, sizeof(s_nodes[0]) * count);
    }
    s_node_count = count;
    s_presence_known = true;

    /* A change goes out at once: a cot that has just dropped off the
     * air should not wait a minute to be noticed.  The first one goes
     * out straight away too, so a hub that has just been set up appears
     * on the dashboard before anyone has the chance to wonder whether
     * it is there. */
    if (changed) {
        s_next_presence_us = 0;
    }
    xSemaphoreGive(s_presence_lock);
}

const char *crynet_net_device_id(void)
{
    return s_device;
}
```

Hub - the panel

firmware/hub_infer/components/hub_display/include/hub_display.h

102 lines · 4,253 bytes

```
/*
 * CryNet — the hub's display.
 *
 * The hub carries a 2.8" ILI9341 SPI colour display on GPIO 10-14, the
 * wiring the legacy firmware used:
 *
 *   CS 10   MOSI 11   SCK 12   DC 13   RST 14   backlight 4
 *
 * The panel driver is a small ILI9341 implementation for ESP-IDF (the
 * ESP-IDF ships drivers for the ST7789 and the SSD1306, not for this
 * one), and the text uses the classic 5x7 font Adafruit's GFX library
 * has used since 1.0, so the display looks the way the legacy one did.
 *
 * The screen has three faces.  A short animated splash runs first — the
 * waveform, the name, the boot steps — and then the dashboard: the
 * hub's own state at the top, and a card for every cot.
 *
 * The BOOT button (GPIO 0, active low) cycles the panel through the
 * other two faces: the help page, which explains what each number on a
 * cot card means, and the colour check, a grid of the palette's
 * swatches, which is how the panel's own colours are judged (a
 * photograph of a glowing screen in a dark room is not a colour
 * measurement).
 */

#ifndef HUB_DISPLAY_H
#define HUB_DISPLAY_H

#include <stdbool.h>
#include <stdint.h>

#ifdef __cplusplus
extern "C" {
#endif

/* RGB565. */
#define DISPLAY_RGB(r, g, b) \
    ((uint16_t)(((r) & 0xF8) << 8 | ((g) & 0xFC) << 3 | ((b) >> 3)))

#define DISPLAY_COLOUR_BLACK  DISPLAY_RGB(0, 0, 0)
#define DISPLAY_COLOUR_WHITE  DISPLAY_RGB(255, 255, 255)
#define DISPLAY_COLOUR_GREY   DISPLAY_RGB(64, 72, 84)
#define DISPLAY_COLOUR_RED    DISPLAY_RGB(220, 40, 40)
#define DISPLAY_COLOUR_GREEN  DISPLAY_RGB(40, 170, 80)
#define DISPLAY_COLOUR_AMBER  DISPLAY_RGB(230, 160, 30)
#define DISPLAY_COLOUR_BLUE   DISPLAY_RGB(30, 90, 200)

/* How many cots the hub knows about. */
#define HUB_DISPLAY_MAX_NODES 2

/* One cot, as the dashboard shows it. */
typedef struct {
    uint8_t id;            /* 1..3 */
    bool    online;        /* the hub has heard from it */
    bool    crying;        /* the hub is announcing a cry from it */
    float   rms;           /* how loud the room is at it, right now */
    float   trigger;       /* the level it sends audio over — the notch
                            * on the level bar.  Zero = not known */
    uint8_t confidence;    /* the last confirmed confidence, in percent */
    int8_t  rssi;          /* telemetry RSSI in dBm; 0 = unknown */
    bool    deaf;          /* the hub cannot get a window from it, so it
                            * cannot announce a cry: it is not being
                            * watched, and the card has to say so */
} hub_display_node_t;

/* The display's own state. */
typedef struct {
    bool     linked;        /* a node has been heard from */
    bool     wifi_up;       /* the hub is on the wireless network */
    bool     crying;        /* the hub is announcing a cry */
    uint8_t  cot_id;        /* which cot — the loudest one */
    uint8_t  confidence;    /* the last confirmed confidence, in percent */
    float    rms;           /* that cot's last RMS */
    float    trigger;       /* that cot's trigger level */
    uint8_t  node_count;    /* how many cots the hub knows about */
```

```
    bool     ble_ready;     /* the Bluetooth server is advertising */
    uint8_t  ble_clients;   /* caregiver devices connected and subscribed */
    uint32_t uptime_s;      /* how long the hub has been up */
    hub_display_node_t nodes[HUB_DISPLAY_MAX_NODES];
} hub_display_state_t;

/* Start the display and its refresh task.  Returns false if the panel
 * did not answer; the hub carries on without it. */
bool hub_display_start(void);

/* Give the display something new to show. */
void hub_display_update(const hub_display_state_t *state);

/* The backlight's brightness, as a percentage: 0 is off, 100 is full.
 * The panel is not meant to be looked at at full brightness in a dark
 * room — the glare is what makes a colour look like the wrong colour.
 * The default is set at boot; this changes it. */
void hub_display_set_brightness(uint8_t percent);

/* The backlight's brightness, as a percentage. */
uint8_t hub_display_brightness(void);

#ifdef __cplusplus
}
#endif

#endif /* HUB_DISPLAY_H */
```

firmware/hub_infer/components/hub_display/ili9341.h

55 lines · 1,614 bytes

```
/*
 * A small ILI9341 driver for ESP-IDF: the panel on the hub.
 */

#ifndef ILI9341_H
#define ILI9341_H

#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>

#include "driver/spi_master.h"

#ifdef __cplusplus
extern "C" {
#endif

typedef struct {
    spi_device_handle_t spi;
    int dc_pin;
    int reset_pin;
    int backlight_pin;
    int width;
    int height;
    uint8_t rotation;
    uint8_t dc_level;     /* the level the next transaction drives D/C to */
} ili9341_t;

/* Bring the panel up.  Returns false if it did not answer. */
bool ili9341_init(ili9341_t *dev, int mosi, int sclk, int cs, int dc,
                  int rst, int backlight, uint8_t rotation);

/* Fill a rectangle with one colour. */
void ili9341_fill(ili9341_t *dev, int x, int y, int w, int h, uint16_t colour);

/* Push a rectangle of RGB565 pixels to the panel.  The pixels are in
 * the host's byte order and are swapped on the way out; `stride` is the
 * distance between one row and the next, in pixels — so a whole screen
 * can be sent in one call. */
void ili9341_blit(ili9341_t *dev, int x, int y, int w, int h,
                  const uint16_t *pixels, size_t stride);

/* The backlight. */
void ili9341_backlight(ili9341_t *dev, bool on);

/* The backlight's brightness, as a percentage: 0 is off, 100 is full.
 * The pin is a LEDC output, so the panel can be turned down instead of
 * only being on or off - see the note in ili9341.c. */
void ili9341_backlight_set(ili9341_t *dev, uint8_t percent);

```

```
#ifdef __cplusplus
}
#endif

#endif /* ILI9341_H */
```

firmware/hub_infer/components/hub_display/ili9341.c

413 lines · 15,708 bytes

```
/*
 * A small ILI9341 driver for ESP-IDF.
 *
 * The ESP-IDF's esp_lcd ships drivers for the ST7789 and the SSD1306 but
 * not for this panel, so this is the ILI9341 brought up directly on the
 * SPI master driver: the address window and the pixel stream are the
 * only two things it has to do.
 *
 * Three things about the ILI9341's serial interface are easy to get
 * wrong, and all three of them are fixed here:
 *
 *   1. **CS must stay down across a command and its parameters.**  The
 *      panel resets its serial interface when CS goes high, so a
 *      command byte and the bytes that follow it have to be one
 *      continuous transfer.  The ESP-IDF's `SPI_TRANS_CS_KEEP_ACTIVE`
 *      does exactly that, and the bus has to be acquired for it to be
 *      allowed.
 *   2. **D/C is the only thing that tells a command from data.**  The
 *      level is changed in the transaction's pre-transmission callback,
 *      which the driver calls once CS is asserted and just before the
 *      bytes go out — the only moment at which it is safe to change it.
 *   3. **The address window is big-endian, and it is one command per
 *      axis.**  CASET takes four bytes (start column, end column) and
 *      PASET takes four bytes (start row, end row).  Sending the column
 *      address twice — once for the start and once for the end —
 *      instead re-sets the start, and the panel ends up with a window
 *      of one pixel and displays uninitialised memory.
 *   4. **The colour order is set by one bit in MADCTL.**  The BGR bit
 *      tells the panel that the top bits of each RGB565 word are blue.
 *      The framebuffer holds red there, so leaving BGR set swaps red
 *      and blue on the glass: the teal accent came out olive, and the
 *      amber "no phone" pill came out blue.  The bit is cleared here.
 *
 * The orientation is a property of *this* panel, not of the chip: it is
 * a "landscape-first" 320x240 glass, so the upright landscape setting
 * is `MADCTL_MY` with a 320x240 logical framebuffer — the same thing
 * the legacy firmware used, via Adafruit's rotation 2 with its 240 px
 * clipping overridden.  Getting this wrong rotates the whole picture
 * by a quarter turn, which is exactly what happened the first time.
 */

#include "ili9341.h"

#include <string.h>

#include "driver/gpio.h"
#include "driver/ledc.h"
#include "driver/spi_master.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

static const char *TAG = "ili9341";

/* The panel's commands. */
#define ILI9341_SWRESET  0x01
#define ILI9341_SLPOUT   0x11
#define ILI9341_INVOFF   0x20
#define ILI9341_DISPON   0x29
#define ILI9341_CASET    0x2A
#define ILI9341_RAMWR    0x2C
#define ILI9341_PASET    0x2B
#define ILI9341_MADCTL   0x36
#define ILI9341_PIXFMT   0x3A

/* Memory access control. */
#define MADCTL_MX  0x40
#define MADCTL_MY  0x80
#define MADCTL_MV  0x20
#define MADCTL_BGR 0x08

/* The backlight.  It used to be a plain on/off pin, and the panel was
```

```
 * left at full brightness.  A TFT at full brightness in a dark room
 * glares: the picture is blown out, the dark navy background reads as a
 * bright blue, and the teal accent is washed out with it.  The pin is
 * now a LEDC output, so the brightness is a number that can be turned
 * down.  5 kHz is above anything the eye can notice as a flicker. */
#define BL_MODE      LEDC_LOW_SPEED_MODE
#define BL_TIMER     LEDC_TIMER_0
#define BL_CHANNEL   LEDC_CHANNEL_0
#define BL_FREQ_HZ   5000
#define BL_RES       LEDC_TIMER_10_BIT
#define BL_STEPS     (1u << BL_RES)

/* The init sequence, in Adafruit's "command, count, data" format. */
static const uint8_t s_init[] = {
    0xEF, 3, 0x03, 0x80, 0x02,
    0xCF, 3, 0x00, 0xC1, 0x30,
    0xED, 4, 0x64, 0x03, 0x12, 0x81,
    0xE8, 3, 0x85, 0x00, 0x78,
    0xCB, 5, 0x39, 0x2C, 0x00, 0x34, 0x02,
    0xF7, 1, 0x20,
    0xEA, 2, 0x00, 0x00,
    0xC0, 1, 0x23,              /* power control 1 */
    0xC1, 1, 0x10,              /* power control 2 */
    0xC5, 2, 0x3e, 0x28,        /* VCM control */
    0xC7, 1, 0x86,              /* VCM control 2 */
    ILI9341_MADCTL, 1, 0x40,    /* memory access control: the rotation
                                 * is written again below, once the
                                 * orientation is known */
    0xB6, 1, 0x00,              /* display function control */
    0xF2, 1, 0x00,              /* 3 gamma function disable */
    0x26, 1, 0x01,              /* gamma set */
    0xE0, 15, 0x0F, 0x31, 0x2B, 0x0C, 0x0E, 0x08,
              0x4E, 0xF1, 0x37, 0x07, 0x10, 0x03, 0x0E, 0x09, 0x00,
    0xE1, 15, 0x00, 0x0E, 0x14, 0x03, 0x11, 0x07,
              0x31, 0xC1, 0x48, 0x08, 0x0F, 0x0C, 0x31, 0x36, 0x0F,
    ILI9341_SLPOUT, 0x80,       /* sleep out */
    ILI9341_DISPON, 0x80,       /* display on */
};

/* The D/C line: the level the next transaction drives it to.  The SPI
 * driver reads it in the pre-transmission callback, which runs inside
 * the driver with the CS line already down. */
static void dc_pre_cb(spi_transaction_t *t)
{
    ili9341_t *dev = (ili9341_t *)t->user;
    gpio_set_level((gpio_num_t)dev->dc_pin, dev->dc_level ? 1 : 0);
}

/* Send bytes with D/C at the given level.  With keep_cs the chip select
 * stays down afterwards, so the next call continues the same transfer. */
static void tx(ili9341_t *dev, uint8_t dc, const uint8_t *buf, size_t len,
               bool keep_cs)
{
    dev->dc_level = dc;
    spi_transaction_t t = {
        .length = len * 8,
        .tx_buffer = buf,
        .user = dev,
    };
    if (keep_cs) {
        t.flags |= SPI_TRANS_CS_KEEP_ACTIVE;
    }
    spi_device_polling_transmit(dev->spi, &t);
}

/* A command, and the parameters that belong to it, in one transfer. */
static void write_command_data(ili9341_t *dev, uint8_t cmd,
                               const uint8_t *data, size_t len)
{
    spi_device_acquire_bus(dev->spi, portMAX_DELAY);
    tx(dev, 0, &cmd, 1, len > 0);
    if (len) {
        tx(dev, 1, data, len, false);
    }
    spi_device_release_bus(dev->spi);
}

bool ili9341_init(ili9341_t *dev, int mosi, int sclk, int cs, int dc,
                  int rst, int backlight, uint8_t rotation)
{
    memset(dev, 0, sizeof *dev);
    dev->dc_pin = dc;
    dev->reset_pin = rst;
    dev->backlight_pin = backlight;

```

```
    gpio_config_t out = {
        .pin_bit_mask = (1ULL << dc) | (1ULL << rst),
        .mode = GPIO_MODE_OUTPUT,
    };
    ESP_ERROR_CHECK(gpio_config(&out));

    /* The panel's reset line. */
    gpio_set_level((gpio_num_t)rst, 1);
    vTaskDelay(pdMS_TO_TICKS(10));
    gpio_set_level((gpio_num_t)rst, 0);
    vTaskDelay(pdMS_TO_TICKS(20));
    gpio_set_level((gpio_num_t)rst, 1);
    vTaskDelay(pdMS_TO_TICKS(150));

    spi_bus_config_t bus = {
        .mosi_io_num = mosi,
        .miso_io_num = -1,
        .sclk_io_num = sclk,
        .quadwp_io_num = -1,
        .quadhd_io_num = -1,
        .max_transfer_sz = 4096,
    };
    esp_err_t err = spi_bus_initialize(SPI2_HOST, &bus, SPI_DMA_CH_AUTO);
    if (err != ESP_OK && err != ESP_ERR_INVALID_STATE) {
        ESP_LOGE(TAG, "the SPI bus could not be started: %s",
                 esp_err_to_name(err));
        return false;
    }

    spi_device_interface_config_t iface = {
        .clock_speed_hz = 40 * 1000 * 1000,
        .mode = 0,
        .spics_io_num = cs,
        .queue_size = 7,
        .pre_cb = dc_pre_cb,
    };
    err = spi_bus_add_device(SPI2_HOST, &iface, &dev->spi);
    if (err != ESP_OK) {
        ESP_LOGE(TAG, "the display could not be added to the bus: %s",
                 esp_err_to_name(err));
        return false;
    }

    /* The init sequence. */
    for (size_t i = 0; i < sizeof s_init;) {
        const uint8_t cmd = s_init[i++];
        const uint8_t count = s_init[i++];
        if (count == 0x80) {
            /* A command with no data, and a pause after it. */
            write_command_data(dev, cmd, NULL, 0);
            vTaskDelay(pdMS_TO_TICKS(120));
            continue;
        }
        write_command_data(dev, cmd, &s_init[i], count);
        i += count;
    }

    /* The rotation.  On this panel the upright landscape setting is
     * MADCTL_MY with a 320x240 logical framebuffer — see the note at
     * the top of this file.
     *
     * MADCTL_BGR is deliberately NOT set.  It would tell the panel that
     * the top bits of each RGB565 word are blue; the framebuffer puts
     * red there, so the picture would arrive with red and blue
     * exchanged — the teal accent olive, the amber pill blue.  The
     * picture in the first photograph of the panel proved it. */
    uint8_t madctl;
    switch (rotation & 3) {
    case 0:  madctl = MADCTL_MX; dev->width = 240; dev->height = 320; break;
    case 1:  madctl = MADCTL_MV; dev->width = 320; dev->height = 240; break;
    case 2:  madctl = MADCTL_MY; dev->width = 320; dev->height = 240; break;
    default: madctl = MADCTL_MX | MADCTL_MY | MADCTL_MV;
             dev->width = 320; dev->height = 240; break;
    }
    write_command_data(dev, ILI9341_MADCTL, &madctl, 1);

    /* Pin the inversion state.  The chip's default after a reset is
     * already "inversion off", but saying so explicitly means the
     * picture cannot depend on what the last user of this glass left
     * in the controller. */
    write_command_data(dev, ILI9341_INVOFF, NULL, 0);

    const uint8_t fmt = 0x55;      /* 16 bits per pixel */
    write_command_data(dev, ILI9341_PIXFMT, &fmt, 1);

```

```
    dev->rotation = rotation;

    ili9341_backlight(dev, true);
    ESP_LOGI(TAG, "the panel is up: %d x %d, rotation %d (madctl 0x%02X)",
             dev->width, dev->height, rotation, madctl);
    return true;
}

void ili9341_backlight(ili9341_t *dev, bool on)
{
    ili9341_backlight_set(dev, on ? 100 : 0);
}

void ili9341_backlight_set(ili9341_t *dev, uint8_t percent)
{
    if (dev->backlight_pin < 0) {
        return;
    }
    if (percent > 100) {
        percent = 100;
    }

    /* The first call sets the pin up as a LEDC output; after that the
     * duty alone sets the brightness.  If the LEDC cannot be had the
     * pin stays a plain output, so the panel is still lit. */
    static bool configured = false;
    if (!configured) {
        configured = true;
        ledc_timer_config_t timer = {
            .speed_mode = BL_MODE,
            .duty_resolution = BL_RES,
            .timer_num = BL_TIMER,
            .freq_hz = BL_FREQ_HZ,
            .clk_cfg = LEDC_USE_APB_CLK,
        };
        ledc_channel_config_t channel = {
            .gpio_num = dev->backlight_pin,
            .speed_mode = BL_MODE,
            .channel = BL_CHANNEL,
            .timer_sel = BL_TIMER,
            .duty = 0,
            .hpoint = 0,
            .sleep_mode = LEDC_SLEEP_MODE_NO_ALIVE_NO_PD,
        };
        if (ledc_timer_config(&timer) != ESP_OK) {
            ESP_LOGE(TAG, "the backlight timer could not be set up");
            configured = false;
        } else if (ledc_channel_config(&channel) != ESP_OK) {
            ESP_LOGE(TAG, "the backlight channel could not be set up");
            configured = false;
        }
        if (!configured) {
            gpio_config_t out = {
                .pin_bit_mask = 1ULL << dev->backlight_pin,
                .mode = GPIO_MODE_OUTPUT,
            };
            gpio_config(&out);
            gpio_set_level((gpio_num_t)dev->backlight_pin, percent ? 1 : 0);
            return;
        }
    }

    /* The backlight's polarity was never settled in the legacy firmware:
     * its diagnostic toggled the pin and asked the user which way was
     * right.  Driving it high is the usual wiring, and the panel was
     * lit when the pin was high. */
    const uint32_t duty = (percent == 0) ? 0 : (BL_STEPS * percent) / 100;
    ledc_set_duty(BL_MODE, BL_CHANNEL, duty);
    ledc_update_duty(BL_MODE, BL_CHANNEL);
}

/* Point the panel's address window at a rectangle.  The caller must be
 * holding the bus. */
static void set_window(ili9341_t *dev, int x, int y, int w, int h)
{
    const uint16_t x0 = (uint16_t)x;
    const uint16_t x1 = (uint16_t)(x + w - 1);
    const uint16_t y0 = (uint16_t)y;
    const uint16_t y1 = (uint16_t)(y + h - 1);
    const uint8_t cols[4] = {
        (uint8_t)(x0 >> 8), (uint8_t)(x0 & 0xFF),
        (uint8_t)(x1 >> 8), (uint8_t)(x1 & 0xFF),
    };
    const uint8_t rows[4] = {
        (uint8_t)(y0 >> 8), (uint8_t)(y0 & 0xFF),
```

```
        (uint8_t)(y1 >> 8), (uint8_t)(y1 & 0xFF),
    };
    const uint8_t cmd_caset = ILI9341_CASET;
    const uint8_t cmd_paset = ILI9341_PASET;
    tx(dev, 0, &cmd_caset, 1, true);
    tx(dev, 1, cols, 4, true);
    tx(dev, 0, &cmd_paset, 1, true);
    tx(dev, 1, rows, 4, true);
}

void ili9341_fill(ili9341_t *dev, int x, int y, int w, int h, uint16_t colour)
{
    if (x < 0) { w += x; x = 0; }
    if (y < 0) { h += y; y = 0; }
    if (x + w > dev->width)  { w = dev->width - x; }
    if (y + h > dev->height) { h = dev->height - y; }
    if (w <= 0 || h <= 0) {
        return;
    }

    /* The pixel stream: the same colour, over and over, in one
     * transfer with CS down the whole way. */
    uint8_t chunk[512];
    const uint16_t px = colour;
    for (size_t i = 0; i < sizeof chunk; i += 2) {
        chunk[i] = (uint8_t)(px >> 8);
        chunk[i + 1] = (uint8_t)(px & 0xFF);
    }
    const size_t pixels = (size_t)w * h;
    const size_t chunk_pixels = sizeof chunk / 2;

    spi_device_acquire_bus(dev->spi, portMAX_DELAY);
    set_window(dev, x, y, w, h);
    const uint8_t cmd_ramwr = ILI9341_RAMWR;
    tx(dev, 0, &cmd_ramwr, 1, true);
    for (size_t sent = 0; sent < pixels; sent += chunk_pixels) {
        size_t n = pixels - sent;
        if (n > chunk_pixels) {
            n = chunk_pixels;
        }
        const bool last = (sent + n >= pixels);
        tx(dev, 1, chunk, n * 2, !last);
    }
    spi_device_release_bus(dev->spi);
}

void ili9341_blit(ili9341_t *dev, int x, int y, int w, int h,
                  const uint16_t *pixels, size_t stride)
{
    if (x < 0) { w += x; x = 0; }
    if (y < 0) { h += y; y = 0; }
    if (x + w > dev->width)  { w = dev->width - x; }
    if (y + h > dev->height) { h = dev->height - y; }
    if (w <= 0 || h <= 0) {
        return;
    }

    spi_device_acquire_bus(dev->spi, portMAX_DELAY);
    set_window(dev, x, y, w, h);
    const uint8_t cmd_ramwr = ILI9341_RAMWR;
    tx(dev, 0, &cmd_ramwr, 1, true);

    /* The panel wants RGB565 big endian; the framebuffer holds it in
     * the host's order, so each pixel is swapped on the way out. */
    const size_t row_px = (size_t)w;
    uint8_t chunk[512];
    for (int row = 0; row < h; row++) {
        const uint16_t *src = pixels + (size_t)row * stride;
        size_t done = 0;
        while (done < row_px) {
            size_t n = row_px - done;
            if (n * 2 > sizeof chunk) {
                n = sizeof chunk / 2;
            }
            for (size_t i = 0; i < n; i++) {
                const uint16_t px = src[done + i];
                chunk[i * 2] = (uint8_t)(px >> 8);
                chunk[i * 2 + 1] = (uint8_t)(px & 0xFF);
            }
            const bool last = (row == h - 1) && (done + n >= row_px);
            tx(dev, 1, chunk, n * 2, !last);
            done += n;
        }
    }
```

```
    spi_device_release_bus(dev->spi);
}
```

firmware/hub_infer/components/hub_display/font5x7.h

121 lines · 5,279 bytes

```
/*
 * The classic 5x7 bitmap font, as ASCII 0x20 to 0x7E.
 *
 * Generated from Adafruit's glcdfont.c (the font Adafruit's GFX library
 * has used since 1.0) by research/build_font.py.  Five column
 * bytes per character; the low seven bits of each byte are the seven
 * rows of the glyph, the topmost row first - the same order
 * Adafruit_GFX::drawChar reads them in.
 *
 * The table in glcdfont.c is indexed by the raw ASCII code, so the glyph
 * for ASCII c is glcdfont.c's bytes [base + c] * 5 .. +5, where the
 * generator detected the base (0) rather than assuming it.
 */

#ifndef FONT5X7_H
#define FONT5X7_H

#define FONT5X7_FIRST 0x20
#define FONT5X7_LAST  0x7E
#define FONT5X7_WIDTH 5
#define FONT5X7_HEIGHT 7

static const uint8_t s_font5x7[][5] = {
    {0x00, 0x00, 0x00, 0x00, 0x00},   /*   */
    {0x00, 0x00, 0x5F, 0x00, 0x00},   /* ! */
    {0x00, 0x07, 0x00, 0x07, 0x00},   /* " */
    {0x14, 0x7F, 0x14, 0x7F, 0x14},   /* # */
    {0x24, 0x2A, 0x7F, 0x2A, 0x12},   /* $ */
    {0x23, 0x13, 0x08, 0x64, 0x62},   /* % */
    {0x36, 0x49, 0x56, 0x20, 0x50},   /* & */
    {0x00, 0x08, 0x07, 0x03, 0x00},   /* ' */
    {0x00, 0x1C, 0x22, 0x41, 0x00},   /* ( */
    {0x00, 0x41, 0x22, 0x1C, 0x00},   /* ) */
    {0x2A, 0x1C, 0x7F, 0x1C, 0x2A},   /* * */
    {0x08, 0x08, 0x3E, 0x08, 0x08},   /* + */
    {0x00, 0x80, 0x70, 0x30, 0x00},   /* , */
    {0x08, 0x08, 0x08, 0x08, 0x08},   /* - */
    {0x00, 0x00, 0x60, 0x60, 0x00},   /* . */
    {0x20, 0x10, 0x08, 0x04, 0x02},   /* / */
    {0x3E, 0x51, 0x49, 0x45, 0x3E},   /* 0 */
    {0x00, 0x42, 0x7F, 0x40, 0x00},   /* 1 */
    {0x72, 0x49, 0x49, 0x49, 0x46},   /* 2 */
    {0x21, 0x41, 0x49, 0x4D, 0x33},   /* 3 */
    {0x18, 0x14, 0x12, 0x7F, 0x10},   /* 4 */
    {0x27, 0x45, 0x45, 0x45, 0x39},   /* 5 */
    {0x3C, 0x4A, 0x49, 0x49, 0x31},   /* 6 */
    {0x41, 0x21, 0x11, 0x09, 0x07},   /* 7 */
    {0x36, 0x49, 0x49, 0x49, 0x36},   /* 8 */
    {0x46, 0x49, 0x49, 0x29, 0x1E},   /* 9 */
    {0x00, 0x00, 0x14, 0x00, 0x00},   /* : */
    {0x00, 0x40, 0x34, 0x00, 0x00},   /* ; */
    {0x00, 0x08, 0x14, 0x22, 0x41},   /* < */
    {0x14, 0x14, 0x14, 0x14, 0x14},   /* = */
    {0x00, 0x41, 0x22, 0x14, 0x08},   /* > */
    {0x02, 0x01, 0x59, 0x09, 0x06},   /* ? */
    {0x3E, 0x41, 0x5D, 0x59, 0x4E},   /* @ */
    {0x7C, 0x12, 0x11, 0x12, 0x7C},   /* A */
    {0x7F, 0x49, 0x49, 0x49, 0x36},   /* B */
    {0x3E, 0x41, 0x41, 0x41, 0x22},   /* C */
    {0x7F, 0x41, 0x41, 0x41, 0x3E},   /* D */
    {0x7F, 0x49, 0x49, 0x49, 0x41},   /* E */
    {0x7F, 0x09, 0x09, 0x09, 0x01},   /* F */
    {0x3E, 0x41, 0x41, 0x51, 0x73},   /* G */
    {0x7F, 0x08, 0x08, 0x08, 0x7F},   /* H */
    {0x00, 0x41, 0x7F, 0x41, 0x00},   /* I */
    {0x20, 0x40, 0x41, 0x3F, 0x01},   /* J */
    {0x7F, 0x08, 0x14, 0x22, 0x41},   /* K */
    {0x7F, 0x40, 0x40, 0x40, 0x40},   /* L */
    {0x7F, 0x02, 0x1C, 0x02, 0x7F},   /* M */
    {0x7F, 0x04, 0x08, 0x10, 0x7F},   /* N */
    {0x3E, 0x41, 0x41, 0x41, 0x3E},   /* O */
    {0x7F, 0x09, 0x09, 0x09, 0x06},   /* P */
    {0x3E, 0x41, 0x51, 0x21, 0x5E},   /* Q */
    {0x7F, 0x09, 0x19, 0x29, 0x46},   /* R */
    {0x26, 0x49, 0x49, 0x49, 0x32},   /* S */
```

```
    {0x03, 0x01, 0x7F, 0x01, 0x03},   /* T */
    {0x3F, 0x40, 0x40, 0x40, 0x3F},   /* U */
    {0x1F, 0x20, 0x40, 0x20, 0x1F},   /* V */
    {0x3F, 0x40, 0x38, 0x40, 0x3F},   /* W */
    {0x63, 0x14, 0x08, 0x14, 0x63},   /* X */
    {0x03, 0x04, 0x78, 0x04, 0x03},   /* Y */
    {0x61, 0x59, 0x49, 0x4D, 0x43},   /* Z */
    {0x00, 0x7F, 0x41, 0x41, 0x41},   /* [ */
    {0x02, 0x04, 0x08, 0x10, 0x20},   /* \ */
    {0x00, 0x41, 0x41, 0x41, 0x7F},   /* ] */
    {0x04, 0x02, 0x01, 0x02, 0x04},   /* ^ */
    {0x40, 0x40, 0x40, 0x40, 0x40},   /* _ */
    {0x00, 0x03, 0x07, 0x08, 0x00},   /* ` */
    {0x20, 0x54, 0x54, 0x78, 0x40},   /* a */
    {0x7F, 0x28, 0x44, 0x44, 0x38},   /* b */
    {0x38, 0x44, 0x44, 0x44, 0x28},   /* c */
    {0x38, 0x44, 0x44, 0x28, 0x7F},   /* d */
    {0x38, 0x54, 0x54, 0x54, 0x18},   /* e */
    {0x00, 0x08, 0x7E, 0x09, 0x02},   /* f */
    {0x18, 0xA4, 0xA4, 0x9C, 0x78},   /* g */
    {0x7F, 0x08, 0x04, 0x04, 0x78},   /* h */
    {0x00, 0x44, 0x7D, 0x40, 0x00},   /* i */
    {0x20, 0x40, 0x40, 0x3D, 0x00},   /* j */
    {0x7F, 0x10, 0x28, 0x44, 0x00},   /* k */
    {0x00, 0x41, 0x7F, 0x40, 0x00},   /* l */
    {0x7C, 0x04, 0x78, 0x04, 0x78},   /* m */
    {0x7C, 0x08, 0x04, 0x04, 0x78},   /* n */
    {0x38, 0x44, 0x44, 0x44, 0x38},   /* o */
    {0xFC, 0x18, 0x24, 0x24, 0x18},   /* p */
    {0x18, 0x24, 0x24, 0x18, 0xFC},   /* q */
    {0x7C, 0x08, 0x04, 0x04, 0x08},   /* r */
    {0x48, 0x54, 0x54, 0x54, 0x24},   /* s */
    {0x04, 0x04, 0x3F, 0x44, 0x24},   /* t */
    {0x3C, 0x40, 0x40, 0x20, 0x7C},   /* u */
    {0x1C, 0x20, 0x40, 0x20, 0x1C},   /* v */
    {0x3C, 0x40, 0x30, 0x40, 0x3C},   /* w */
    {0x44, 0x28, 0x10, 0x28, 0x44},   /* x */
    {0x4C, 0x90, 0x90, 0x90, 0x7C},   /* y */
    {0x44, 0x64, 0x54, 0x4C, 0x44},   /* z */
    {0x00, 0x08, 0x36, 0x41, 0x00},   /* { */
    {0x00, 0x00, 0x77, 0x00, 0x00},   /* | */
    {0x00, 0x41, 0x36, 0x08, 0x00},   /* } */
    {0x02, 0x01, 0x02, 0x04, 0x02},   /* ~ */
};

#endif /* FONT5X7_H */
```

firmware/hub_infer/components/hub_display/hub_display.c

1183 lines · 48,163 bytes

```
/*
 * CryNet — the hub's display.  See include/hub_display.h.
 *
 * The public API is deliberately dumb: hub_display_update() just copies
 * the newest state under a spinlock, and a low-priority task on the
 * other core redraws whatever changed.  Only this task ever touches the
 * panel, so the driver needs no locking of its own.
 *
 * The screen has three faces.
 *
 *   * A splash, in two acts.  First the boot: a waveform that moves,
 *     the name, the boot steps.  Then the title card: the project's
 *     own title, the name of the person who built it and the number it
 *     is registered under.  A final-year project is judged partly on
 *     what it is and who did it, and a device that says so on its own
 *     screen answers the examiner before they ask.
 *
 *   * The dashboard.  The hub's own state at the top (cot link,
 *     bluetooth, uptime), a banner that says what the system is doing,
 *     and a card for every cot: its name, whether it is online, and its
 *     level.
 *
 *   * Two more faces, summoned with the BOOT button: the help page,
 *     which explains what each number on a card means, and the colour
 *     check, a grid of the palette's swatches.  The colour check exists
 *     because a photograph of a glowing panel in a dark room is not a
 *     colour measurement — the camera opens up, the highlights clip and
 *     the white balance drifts — so the swatches are how the panel's
 *     own colours are judged.
 *
 * The palette is a calm near-black teal with a teal accent.  Red is
```

```
 * kept for one thing only — a confirmed cry — because a panel that is
 * red all the time stops meaning anything.
 *
 * Everything is composed in a framebuffer (320x240 RGB565, in PSRAM)
 * and pushed to the panel in one transfer, so a frame is never torn.
 */

#include "hub_display.h"

#include <math.h>
#include <stdio.h>
#include <string.h>

#include "esp_heap_caps.h"
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

#include "driver/gpio.h"

#include "font5x7.h"
#include "ili9341.h"

static const char *TAG = "hub_display";

/* The panel hangs on the pins the legacy firmware used: CS 10, MOSI 11,
 * SCK 12, DC 13, RST 14, backlight 4. */
#define PIN_MOSI 11
#define PIN_SCLK 12
#define PIN_CS   10
#define PIN_DC   13
#define PIN_RST  14
#define PIN_BL   4

/* The BOOT button on the devkit is GPIO0, and it is active low.  It
 * opens and closes the colour check. */
#define PIN_BUTTON 0

/* This panel is "landscape-first": the upright landscape setting is
 * MADCTL_MY with a 320x240 framebuffer, which is what the legacy
 * firmware used (Adafruit's rotation 2, with its 240 px clipping
 * overridden).  Rotation 1 (MADCTL_MV) turns the whole picture a
 * quarter turn on this glass, which is what happened the first time. */
#define ROTATION_LANDSCAPE 2

#define FB_W 320
#define FB_H 240

/* ---- the palette ------------------------------------------------------- */
/* A dark, calm scheme: a near-black teal background, a teal accent,
 * green for a cot that is fine, amber for a cot that is loud, and rose
 * — not fire-engine red — for a confirmed cry.
 *
 * The neutrals are darker and teal-tinted rather than navy, and that is
 * a measured change.  The colour-check page (the BOOT button) put a
 * (12,20,32) navy on the glass and a (0,0,0) black next to it, and the
 * navy came back as a **saturated blue while the black stayed black**:
 * this panel lifts the dark end, and it lifts the blue channel most of
 * all, so the *less* blue the screen asks for, the less blue it gets.
 * A background that asks for 32 blue reads as a bright blue; one that
 * asks for 16 reads as a dark one.  The theme is unchanged in what it
 * says - teal for the system, green for fine, amber for loud, rose for
 * a cry - but the screen now reads teal instead of blue.
 *
 * The level itself was picked off the glass, not out of a chart: shown
 * the dark ramp, the reader chose its third step - (4,8,8) - as the teal
 * the screen should be, and the whole neutral family moved down one step
 * of the same ramp with it.  On the glass that made the cards sit at the
 * same value as the background - the panel lifts the dark end until a
 * two-step gap reads as one - so the background has dropped another step:
 * the cards keep (7,15,16) and the ground under them is now (2,4,4).
 *
 * The header strip is its own colour for the same reason, one edge at a
 * time: the backlight's LEDs run along the top of the glass and lift
 * that strip blue, so a header painted the same (7,15,16) as the cards
 * photographed blue while the cards photographed teal.  (7,20,15) asks
 * for five steps of green above blue, which the lift spends on its way
 * in; what the reader sees is the teal the cards show. */
#define C_BG         DISPLAY_RGB(2, 4, 4)
#define C_SURFACE    DISPLAY_RGB(7, 15, 16)
#define C_SURFACE_HI DISPLAY_RGB(12, 26, 28)
#define C_HEADER     DISPLAY_RGB(7, 20, 15)
/* The border is a UI component, so it has to clear 3:1 against the
 * card it sits on — the first choice (40,57,76) measured 1.40:1 and
 * was effectively invisible. */
```

```
#define C_BORDER     DISPLAY_RGB(78, 116, 112)
#define C_TEXT       DISPLAY_RGB(228, 240, 236)
#define C_DIM        DISPLAY_RGB(122, 150, 146)
#define C_ACCENT     DISPLAY_RGB(45, 212, 191)
#define C_ACCENT_DIM DISPLAY_RGB(22, 124, 112)
#define C_BLUE       DISPLAY_RGB(96, 165, 250)
#define C_GREEN      DISPLAY_RGB(52, 211, 153)
#define C_AMBER      DISPLAY_RGB(251, 191, 36)
#define C_ALERT      DISPLAY_RGB(251, 113, 133)

/* ---- the layout ---------------------------------------------------------- */
#define HEADER_H     30
#define BANNER_X     10
#define BANNER_Y     40
#define BANNER_W     300
#define BANNER_H     44
#define CARD_X       10
#define CARD_W       300
/* Two cot cards, taller now: each carries the cot's name and state,
 * an RMS bar with its number, and a bottom row with the radio link --
 * a small signal bar plus the RSSI.  88 + 2*70 puts the last card's
 * bottom edge at 222, leaving room for the footer at 230. */
#define CARD_H       64
#define CARD_Y0      88
#define CARD_STEP    70
#define FOOT_Y       230

/* The splash: how long it runs, how long each fade takes, and how fast
 * it animates.
 *
 * The splash runs in two acts.  The first is the boot: the waveform, the
 * name, the boot steps and a progress bar.  The second is the title
 * card: the project this device is, the person who built it, and the
 * number it is registered under.  A final-year project is judged partly
 * on what it is and who did it, and a device that says so on its own
 * screen answers the examiner before they ask.
 *
 * The timeline, in milliseconds from the start of the splash:
 *
 *   0 .. BOOT_MS           the boot face
 *   BOOT_MS .. +FADE_MS    the boot face fading out
 *   +FADE_MS .. SPLASH_MS  the title card
 *   SPLASH_MS .. +FADE_MS  the whole splash fading out
 *   .. +2*FADE_MS          the dashboard fading in
 */
#define SPLASH_MS    5600
#define BOOT_MS      2400
#define TITLE_MS     (SPLASH_MS - BOOT_MS - FADE_MS)
#define FADE_MS      600
#define SPLASH_TICK  100        /* ms between frames: ten a second */

/* The backlight, as a percentage.  Full brightness glares in a dark
 * room, and a glare is what makes a colour look like the wrong colour:
 * the photograph of the panel came out blue because the panel was
 * brighter than the room and the camera's white balance went with it.
 * 55 % is the level that keeps a dark navy background dark and lets
 * the teal read as teal; it can be changed at run time. */
#define BACKLIGHT_DEFAULT  55

static ili9341_t s_dev;
static uint16_t *s_fb;          /* 320 x 240 RGB565, in PSRAM */

static hub_display_state_t s_state;   /* guarded by s_lock */
static portMUX_TYPE s_lock = portMUX_INITIALIZER_UNLOCKED;
static volatile uint32_t s_version;   /* bumped on every update */

/* The backlight's brightness, and which of the panel's three faces is
 * up.  Both are read by the display task and written by the API. */
static volatile uint8_t s_brightness = BACKLIGHT_DEFAULT;
/* 0 = the dashboard, 1 = the help page, 2 = the colour check. */
static volatile int s_face = 0;

/* The region of the panel that has changed and still has to be pushed. */
static int s_dirty_x0, s_dirty_y0, s_dirty_x1, s_dirty_y1;

/* ---------------------------------------------------------------------- */
/* The framebuffer                                                        */
/* ---------------------------------------------------------------------- */

static void mark_dirty(int x, int y, int w, int h)
{
    if (w <= 0 || h <= 0) {
        return;
    }
    if (s_dirty_x1 < s_dirty_x0) {
```

```
        s_dirty_x0 = x; s_dirty_y0 = y;
        s_dirty_x1 = x + w - 1; s_dirty_y1 = y + h - 1;
        return;
    }
    if (x < s_dirty_x0) { s_dirty_x0 = x; }
    if (y < s_dirty_y0) { s_dirty_y0 = y; }
    if (x + w - 1 > s_dirty_x1) { s_dirty_x1 = x + w - 1; }
    if (y + h - 1 > s_dirty_y1) { s_dirty_y1 = y + h - 1; }
}

static inline void fb_px(int x, int y, uint16_t c)
{
    if ((unsigned)x >= FB_W || (unsigned)y >= FB_H) {
        return;
    }
    s_fb[(size_t)y * FB_W + x] = c;
}

static void fb_fill(int x, int y, int w, int h, uint16_t c)
{
    if (w <= 0 || h <= 0) {
        return;
    }
    if (x < 0) { w += x; x = 0; }
    if (y < 0) { h += y; y = 0; }
    if (x + w > FB_W) { w = FB_W - x; }
    if (y + h > FB_H) { h = FB_H - y; }
    for (int j = 0; j < h; j++) {
        uint16_t *row = &s_fb[(size_t)(y + j) * FB_W + x];
        for (int i = 0; i < w; i++) {
            row[i] = c;
        }
    }
    mark_dirty(x, y, w, h);
}

static void fb_hline(int x, int y, int w, uint16_t c)
{
    fb_fill(x, y, w, 1, c);
}

static void fb_vline(int x, int y, int h, uint16_t c)
{
    fb_fill(x, y, 1, h, c);
}

static void fb_rect(int x, int y, int w, int h, uint16_t c)
{
    fb_hline(x, y, w, c);
    fb_hline(x, y + h - 1, w, c);
    fb_vline(x, y, h, c);
    fb_vline(x + w - 1, y, h, c);
}

/* Blend two colours.  t = 0 gives a, t = 1 gives b. */
static uint16_t blend(uint16_t a, uint16_t b, float t)
{
    if (t <= 0.0f) { return a; }
    if (t >= 1.0f) { return b; }
    const int ar = (a >> 11) & 0x1F, ag = (a >> 5) & 0x3F, ab = a & 0x1F;
    const int br = (b >> 11) & 0x1F, bg = (b >> 5) & 0x3F, bb = b & 0x1F;
    const int f = (int)(t * 256.0f);
    const int r = (ar * (256 - f) + br * f) >> 8;
    const int g = (ag * (256 - f) + bg * f) >> 8;
    const int bl = (ab * (256 - f) + bb * f) >> 8;
    return (uint16_t)(((r & 0x1F) << 11) | ((g & 0x3F) << 5) | (bl & 0x1F));
}

/* A filled rectangle with rounded corners. */
static void fb_round_fill(int x, int y, int w, int h, int r, uint16_t c)
{
    if (w <= 0 || h <= 0) {
        return;
    }
    if (r <= 0) {
        fb_fill(x, y, w, h, c);
        return;
    }
    if (r > w / 2) { r = w / 2; }
    if (r > h / 2) { r = h / 2; }
    fb_fill(x + r, y, w - 2 * r, h, c);
    fb_fill(x, y + r, r, h - 2 * r, c);
    fb_fill(x + w - r, y + r, r, h - 2 * r, c);

    /* The four corner discs.  Each disc is centred on the point the
```

```
     * straight edges stop at, and its quarter is walked *towards its
     * own corner* (sx, sy).  An earlier version walked every disc
     * inward from its centre, which painted all four arcs on the
     * inside of the shape; the bottom-right one only looked right
     * because its inward quadrant happened to end at the shape's own
     * edge. */
    const int cx[4] = { x + r, x + w - 1 - r, x + r, x + w - 1 - r };
    const int cy[4] = { y + r, y + r, y + h - 1 - r, y + h - 1 - r };
    const int sx[4] = { -1, 1, -1, 1 };
    const int sy[4] = { -1, -1, 1, 1 };
    for (int k = 0; k < 4; k++) {
        for (int dx = 0; dx <= r; dx++) {
            for (int dy = 0; dy <= r; dy++) {
                if (dx * dx + dy * dy > r * r) {
                    continue;
                }
                fb_px(cx[k] + sx[k] * dx, cy[k] + sy[k] * dy, c);
            }
        }
    }
    mark_dirty(x, y, w, h);
}

/* An outlined rectangle with rounded corners. */
static void fb_round_rect(int x, int y, int w, int h, int r, uint16_t c)
{
    if (w <= 0 || h <= 0) {
        return;
    }
    if (r <= 0) {
        fb_rect(x, y, w, h, c);
        return;
    }
    if (r > w / 2) { r = w / 2; }
    if (r > h / 2) { r = h / 2; }
    fb_hline(x + r, y, w - 2 * r, c);
    fb_hline(x + r, y + h - 1, w - 2 * r, c);
    fb_vline(x, y + r, h - 2 * r, c);
    fb_vline(x + w - 1, y + r, h - 2 * r, c);

    /* The corner arcs - the same discs as fb_round_fill, but only the
     * pixels a ring's width from the arc itself, and walked towards
     * their own corners (see the note there). */
    const int cx[4] = { x + r, x + w - 1 - r, x + r, x + w - 1 - r };
    const int cy[4] = { y + r, y + r, y + h - 1 - r, y + h - 1 - r };
    const int sx[4] = { -1, 1, -1, 1 };
    const int sy[4] = { -1, -1, 1, 1 };
    for (int k = 0; k < 4; k++) {
        for (int dx = 0; dx <= r; dx++) {
            for (int dy = 0; dy <= r; dy++) {
                const int d2 = dx * dx + dy * dy;
                if (d2 > r * r || d2 < (r - 1) * (r - 1)) {
                    continue;
                }
                fb_px(cx[k] + sx[k] * dx, cy[k] + sy[k] * dy, c);
            }
        }
    }
    mark_dirty(x, y, w, h);
}

/* Fade the whole framebuffer towards one colour.  t = 0 leaves it
 * alone, t = 1 washes it out completely. */
static void fb_fade(uint16_t towards, float t)
{
    if (t <= 0.0f) {
        return;
    }
    if (t > 1.0f) { t = 1.0f; }
    const int f = (int)(t * 256.0f);
    const uint16_t tr = (towards >> 11) & 0x1F;
    const uint16_t tg = (towards >> 5) & 0x3F;
    const uint16_t tb = towards & 0x1F;
    for (size_t i = 0; i < (size_t)FB_W * FB_H; i++) {
        const uint16_t v = s_fb[i];
        const int vr = (v >> 11) & 0x1F, vg = (v >> 5) & 0x3F, vb = v & 0x1F;
        const int r = (vr * (256 - f) + tr * f) >> 8;
        const int g = (vg * (256 - f) + tg * f) >> 8;
        const int b = (vb * (256 - f) + tb * f) >> 8;
        s_fb[i] = (uint16_t)(((r & 0x1F) << 11) | ((g & 0x3F) << 5) | (b & 0x1F));
    }
    mark_dirty(0, 0, FB_W, FB_H);
}

/* ------------------------------------------------------------------ */
```

```
/* Text                                                               */
/* ------------------------------------------------------------------ */

static int fb_text_w(const char *text, uint8_t scale)
{
    if (scale == 0) { scale = 1; }
    const int n = (int)strlen(text);
    if (n == 0) { return 0; }
    return n * (FONT5X7_WIDTH + 1) * scale - scale;
}

static void fb_text(int x, int y, const char *text, uint16_t c, uint8_t scale)
{
    if (scale == 0) { scale = 1; }
    int cursor = x;
    for (const char *p = text; *p; p++) {
        unsigned char ch = (unsigned char)*p;
        if (ch < FONT5X7_FIRST || ch > FONT5X7_LAST) {
            ch = '?';
        }
        const uint8_t *glyph = s_font5x7[ch - FONT5X7_FIRST];
        for (int col = 0; col < FONT5X7_WIDTH; col++) {
            const uint8_t bits = glyph[col];
            if (!bits) { continue; }
            for (int row = 0; row < FONT5X7_HEIGHT; row++) {
                if (bits & (1 << row)) {
                    fb_fill(cursor + col * scale, y + row * scale,
                            scale, scale, c);
                }
            }
        }
        cursor += (FONT5X7_WIDTH + 1) * scale;
    }
}

static void fb_text_centre(int y, const char *text, uint16_t c, uint8_t scale)
{
    fb_text((FB_W - fb_text_w(text, scale)) / 2, y, text, c, scale);
}

/* ------------------------------------------------------------------ */
/* The splash                                                         */
/* ------------------------------------------------------------------ */

#define WAVE_BARS 36
#define WAVE_STEP 7
#define WAVE_X0   ((FB_W - (WAVE_BARS * WAVE_STEP - 3)) / 2)
#define WAVE_Y    46
#define WAVE_MAX  12

/* A scrolling waveform: the shape of sound arriving at the mic. */
static void draw_waveform(int frame)
{
    const float t = (float)frame * 0.18f;
    for (int i = 0; i < WAVE_BARS; i++) {
        const float p = t + i * 0.42f;
        const float a = 0.5f + 0.5f * sinf(p * 1.7f);
        const float b = 0.5f + 0.5f * sinf(p * 0.53f + 1.7f);
        const float h = (float)WAVE_MAX * (0.18f + 0.82f * a * b);
        const int bh = (int)h;
        const int x = WAVE_X0 + i * WAVE_STEP;
        fb_fill(x, WAVE_Y - bh, 4, bh, C_ACCENT);
        fb_fill(x, WAVE_Y + 1, 4, bh, C_ACCENT_DIM);
    }
}

static const char *const s_boot_steps[] = {
    "starting the display",
    "the classifier is loaded",
    "the radio is listening",
    "bluetooth is advertising",
    "searching for the cots",
};

/* The boot face: the waveform, the name, and what the hub is doing as it
 * comes up. */
static void draw_boot_face(int64_t elapsed_ms, int frame)
{
    fb_fill(0, 0, FB_W, FB_H, C_BG);

    draw_waveform(frame);

    /* The device's full name -- stacked, the way the wearable sets it,
     * because sixteen characters at the headline size four are wider
     * than the panel.  The wordmark breathes a little. */
```

```
    const float pulse = 0.5f + 0.5f * sinf((float)frame * 0.12f);
    fb_text_centre(68, "AMAZING'S",
                   blend(C_TEXT, C_ACCENT, 0.25f + 0.35f * pulse), 2);
    fb_text_centre(88, "CRYNET",
                   blend(C_TEXT, C_ACCENT, 0.25f + 0.35f * pulse), 4);
    fb_text_centre(134, "INFANT CRY MONITOR", C_DIM, 1);

    /* The boot steps and a progress bar that fills as they go. */
    int step = (int)(elapsed_ms / (BOOT_MS / 5));
    if (step > 4) { step = 4; }
    fb_text_centre(190, s_boot_steps[step], C_ACCENT, 1);

    const int bar_x = 90, bar_w = 140, bar_y = 206;
    const float prog = (float)elapsed_ms / (float)BOOT_MS;
    int filled = (int)(bar_w * (prog > 1.0f ? 1.0f : prog));
    fb_fill(bar_x, bar_y, bar_w, 6, C_SURFACE_HI);
    if (filled > 0) {
        fb_fill(bar_x, bar_y, filled, 6, C_ACCENT);
    }
    fb_rect(bar_x - 1, bar_y - 1, bar_w + 2, 8, C_BORDER);

    fb_text_centre(222, "esp32-s3 hub", C_DIM, 1);
}

/* The title card.
 *
 * This is the project's own face, and it carries three things: what the
 * device is, who built it, and the number it is registered under.  The
 * font is upper case only, so the name is set the way it would be set on
 * a certificate, and the id number is given room to be read.
 *
 * The full project title is long — eighty characters — and the panel is
 * 320 pixels wide, which is fifty-three characters at the small size.
 * So it is set across three lines, and the last word of the second line
 * hangs a little short, which is what a title does.
 */
static void draw_title_face(int64_t elapsed_ms, int frame)
{
    fb_fill(0, 0, FB_W, FB_H, C_BG);

    /* A quiet accent band across the top, so the card has a top edge. */
    fb_fill(0, 0, FB_W, 3, C_ACCENT_DIM);
    fb_fill(0, 0, 60, 3, C_ACCENT);

    fb_text_centre(22, "AMAZING'S CRYNET", C_ACCENT, 2);
    fb_text_centre(44, "A MULTI-NODE INFANT CRY DETECTION SYSTEM",
                   C_DIM, 1);

    /* The title, in three lines, in the dim colour so it reads as the
     * caption rather than as the headline. */
    fb_text_centre(74, "DESIGN AND CONSTRUCTION OF A", C_DIM, 1);
    fb_text_centre(86, "MULTI-NODE INFANT CRY DETECTION", C_DIM, 1);
    fb_text_centre(98, "SYSTEM WITH WEARABLE CAREGIVER ALERT", C_DIM, 1);

    /* A rule, then the name and the id number.  These are the two
     * things a panel has to say if the person standing in front of it
     * is the person who built it. */
    fb_fill(60, 120, 200, 1, C_BORDER);

    /* The name, breathing gently like the wordmark does on the boot
     * face — the same hand, so the two faces belong together. */
    const float pulse = 0.5f + 0.5f * sinf((float)frame * 0.09f);
    fb_text_centre(136, "AMAZING AYOMIKUN ADELOWO",
                   blend(C_TEXT, C_ACCENT, 0.10f + 0.20f * pulse), 2);

    fb_text_centre(170, "22/05/04/0029", C_DIM, 1);

    /* The last line: what this device is in the project, so the panel
     * names its own role as well as its owner. */
    fb_text_centre(206, "ESP32-S3 HUB  -  TWO COT NODES", C_DIM, 1);

    /* A waveform along the bottom edge, tying the card to the boot
     * face: the same bars, moving. */
    const float t = (float)frame * 0.18f;
    for (int i = 0; i < 48; i++) {
        const float p = t + i * 0.42f;
        const float a = 0.5f + 0.5f * sinf(p * 1.7f);
        const float b = 0.5f + 0.5f * sinf(p * 0.53f + 1.7f);
        const int bh = (int)(6.0f * (0.18f + 0.82f * a * b));
        fb_fill(i * 7, FB_H - 1 - bh, 4, bh, C_ACCENT_DIM);
    }
}

/* The splash, whichever act of it is running now. */
static void draw_splash(int64_t elapsed_ms, int frame)
```

```
{
    if (elapsed_ms < BOOT_MS) {
        draw_boot_face(elapsed_ms, frame);
        return;
    }
    if (elapsed_ms < BOOT_MS + FADE_MS) {
        /* The boot face dissolving into the title card. */
        draw_boot_face(elapsed_ms, frame);
        fb_fade(C_BG, (float)(elapsed_ms - BOOT_MS) / (float)FADE_MS);
        return;
    }
    draw_title_face(elapsed_ms, frame);
}

/* ------------------------------------------------------------------ */
/* The dashboard                                                      */
/* ------------------------------------------------------------------ */

/* A pill with the colour chosen by the caller: the NODE pill has two
 * states, and the WEARABLE pill has three — attached, waiting to be
 * attached, and the Bluetooth stack itself down. */
static void draw_pill_col(int x, int y, const char *label, uint16_t col)
{
    fb_fill(x, y + 2, 6, 6, col);
    fb_text(x + 10, y, label, col, 1);
}

/* The full-scale of the level bar: the RMS that fills it.
 *
 * The bar used to be scaled against 1.0, which is the RMS of a signal
 * that is clipping, and a cry does not clip: the loudest genuine cry
 * measured on the bench carried an RMS of 0.18, and the quietest one
 * the system has to catch carried 0.010.  Against a full scale of 1.0
 * even a cry filled a third of the bar, so the bar never said anything
 * and the number beside it was the only thing worth reading.
 *
 * 0.25 is the full scale now: a scream fills most of the bar, a cry at
 * one metre fills about two thirds, and a quiet room shows a sliver.
 * The scale is square-rooted, so the quiet end of the range is not
 * squashed into nothing -- the interesting part of this bar is the
 * bottom fifth of the scale, where the trigger sits. */
#define LEVEL_BAR_FULL 0.25f

/* Where the cot's own trigger sits on the bar: the level it sends
 * audio over.  This is the number the notch marks, and it is the one
 * the "level" in the project's vocabulary means: the threshold, not
 * the loudness.  The loudness is the RMS. */
static void draw_level_bar(int x, int y, int w, int h, float level,
                           float trigger, bool crying, bool known)
{
    /* An empty bar in a full-strength frame reads as "active" - part of
     * why the no-signal cards looked busier than the live ones - so a
     * dead card's track and frame are washed towards the background. */
    fb_fill(x, y, w, h, known ? C_SURFACE_HI : C_SURFACE);

    float v = level;
    if (v < 0.0f) { v = 0.0f; }
    if (v > LEVEL_BAR_FULL) { v = LEVEL_BAR_FULL; }
    const float frac = sqrtf(v / LEVEL_BAR_FULL);

    /* The colour says whether the cot is over its own trigger: a cot
     * below it is only being listened to, a cot over it is sending
     * audio to the hub. */
    uint16_t col = C_GREEN;
    if (crying) {
        col = C_ALERT;
    } else if (trigger > 0.0f && level >= trigger) {
        col = C_AMBER;
    }

    const int fill_w = (int)(frac * (float)(w - 2) + 0.5f);
    if (fill_w > 0) {
        fb_fill(x + 1, y + 1, fill_w, h - 2, col);
    }

    /* The notch: where this cot's trigger sits.  It is what turns the
     * bar from "a number in colour" into a picture of the decision the
     * cot is about to make -- the bar crosses the notch, the cot sends
     * its audio, and the hub gets to judge it. */
    if (known && trigger > 0.0f) {
        float t = trigger;
        if (t > LEVEL_BAR_FULL) { t = LEVEL_BAR_FULL; }
        const int nx = x + 1 + (int)(sqrtf(t / LEVEL_BAR_FULL) *
                                     (float)(w - 2) + 0.5f);
        if (nx > x && nx < x + w - 1) {
```

```
            fb_fill(nx, y, 1, h, C_TEXT);
        }
    }

    fb_rect(x, y, w, h, known ? C_BORDER : blend(C_BORDER, C_BG, 0.45f));
}

/* How many bars a telemetry RSSI is worth.  2.4 GHz links in a house:
 * anything above -60 dBm is a strong link, below -90 is barely there. */
static int rssi_bars(int8_t rssi)
{
    if (rssi == 0) {
        return 0;
    }
    if (rssi >= -60) { return 4; }
    if (rssi >= -70) { return 3; }
    if (rssi >= -80) { return 2; }
    if (rssi >= -90) { return 1; }
    return 0;
}

/* A small signal-strength glyph: four vertical bars, bottom aligned,
 * filled up to `bars`.  Drawn at the bottom-left of a cot card.
 *
 * The filled bars are coloured by how good the link is, not just by
 * being filled: a cot the hub can barely hear is an amber bar rather
 * than a green one, so the caregiver sees a weak link without reading
 * a number.  A cot that is not answering at all has no bars and the
 * card says OFFLINE, so nothing here has to carry that. */
static void draw_signal_bars(int x, int y, int bars)
{
    const uint16_t col = (bars >= 3) ? C_GREEN : C_AMBER;
    for (int i = 0; i < 4; i++) {
        const int h = 4 + i * 3;
        fb_fill(x + i * 5, y - h, 3, h, (i < bars) ? col : C_SURFACE_HI);
    }
}

/* The wireless mark: two arcs over a dot, the shape every phone shows
 * for a connection.  There is no glyph for it in the panel's font and
 * no room for a picture, so it is drawn from the same pixels as
 * everything else: a dot for the router, and two rings over it for the
 * signal reaching out.
 *
 * The arcs are walked by angle, not by column.  A column-by-column walk
 * of a shallow arc leaves gaps at the sides -- and the sides are where
 * the eye looks first, because that is where the arcs end.  Each arc is
 * two pixels thick, drawn as a pair of radii.
 *
 * The mark is drawn only while the hub is on the network.  A mark that
 * is always there has to be read to be understood: grey means "not
 * connected", and that is a second thing to learn beside the two
 * pills.  With the mark present exactly when it means something, the
 * panel has one rule -- *a mark here means the hub is on the
 * network* -- and no other state to confuse it with.  Where the hub
 * is not on the network, the dashboard says so instead, and there the
 * question can be asked with room to answer it.
 *
 * The flag is the same one the uploader uses to decide whether a
 * record can leave the hub, so the glass cannot disagree with the
 * network: the panel shows a mark only when the records are actually
 * on their way. */
static void draw_wifi_glyph(int cx, int base_y, uint16_t col)
{
    /* The router, at the foot of the mark. */
    fb_fill(cx - 1, base_y - 2, 3, 3, col);

    static const int radii[2] = { 8, 13 };
    for (int k = 0; k < 2; k++) {
        for (int band = 0; band < 2; band++) {
            const int r = radii[k] - band;
            for (int a = 40; a <= 140; a += 2) {
                const float rad = (float)a * 0.01745329f;   /* degrees */
                const float fx = (float)r * cosf(rad);
                const float fy = (float)r * sinf(rad);
                const int x = cx + (int)(fx + (fx >= 0 ? 0.5f : -0.5f));
                const int y = base_y - (int)(fy + 0.5f);
                fb_fill(x, y, 1, 1, col);
            }
        }
    }
}

static void draw_dashboard(const hub_display_state_t *st)
{
```

```
    fb_fill(0, 0, FB_W, FB_H, C_BG);

    /* ---- the header ---- */
    fb_fill(0, 0, FB_W, HEADER_H, C_HEADER);
    fb_text(12, 8, "CRYNET", C_ACCENT, 2);
    fb_text(90, 12, "S3 HUB", C_DIM, 1);
    /* The wireless link, between the hub's name and the cot link: the
     * two things the hub depends on, side by side.  It is drawn only
     * while the hub is on the network, so the rule is one line long:
     * a mark here means the records are on their way. */
    if (st->wifi_up) {
        draw_wifi_glyph(160, 21, C_GREEN);
    }
    /* The cot link: green while at least one cot is answering, amber
     * once every card has gone quiet.  Amber, not the dim grey the
     * first version used, so both pills read the same way — green is
     * working, amber is waiting, and there is one rule to learn. */
    draw_pill_col(196, 11, "NODE", st->linked ? C_GREEN : C_AMBER);
    /* The caregiver's device — the wearable, which uses the same GATT
     * service — is the other half of the system, so its state belongs
     * on the panel: green when one is connected and subscribed, amber
     * when the stack is up but nobody has attached yet, dim when
     * Bluetooth itself is down. */
    const uint16_t wearable_col = (st->ble_clients > 0) ? C_GREEN
                                : (st->ble_ready ? C_AMBER : C_DIM);
    draw_pill_col(258, 11, "WEARABLE", wearable_col);
    fb_fill(0, HEADER_H, FB_W, 2, C_ACCENT_DIM);

    /* ---- the banner: what the system is doing ---- */
    uint16_t banner_col = C_DIM;
    const char *banner = "SEARCHING FOR THE COTS";
    if (st->crying) {
        banner = "CRY DETECTED";
        banner_col = C_ALERT;
    } else if (st->linked) {
        banner = "LISTENING";
        banner_col = C_ACCENT;
    }
    fb_round_fill(BANNER_X, BANNER_Y, BANNER_W, BANNER_H, 8,
                  blend(C_SURFACE, banner_col, 0.12f));
    fb_round_rect(BANNER_X, BANNER_Y, BANNER_W, BANNER_H, 8,
                  blend(C_BORDER, banner_col, st->crying ? 0.85f : 0.25f));
    /* A short bar of the state colour down the left edge. */
    fb_fill(BANNER_X + 3, BANNER_Y + 10, 3, BANNER_H - 20, banner_col);
    fb_text(BANNER_X + 16, BANNER_Y + 9, banner, banner_col, 2);

    char right[24];
    if (st->crying) {
        snprintf(right, sizeof right, "COT %u  %u%%", st->cot_id,
                 st->confidence);
    } else if (st->linked) {
        /* The banner's number is the loudness of the room at the cot
         * the hub is listening to.
         *
         * It used to be labelled LEVEL, which was wrong twice over:
         * the number is an RMS, and "level" already means the trigger
         * threshold everywhere else in this project — the console
         * prints "rms" and "level" as two different numbers, and the
         * word on the panel meant the third thing again.  One word,
         * one number: the panel says RMS, and the help page says what
         * an RMS is. */
        snprintf(right, sizeof right, "COT %u  RMS %.3f", st->cot_id,
                 (double)st->rms);
    } else {
        snprintf(right, sizeof right, "--");
    }
    fb_text(BANNER_X + BANNER_W - 12 - fb_text_w(right, 1), BANNER_Y + 26,
            right, C_DIM, 1);

    /* A cry that nobody is attached to hear goes nowhere, and the
     * caregiver has to know that before they go looking for their
     * phone. */
    if (st->crying && st->ble_clients == 0) {
        fb_text(BANNER_X + 16, BANNER_Y + 26, "ALERT NOT DELIVERED",
                C_AMBER, 1);
    }

    /* ---- a card for every cot ---- */
    const int count = st->node_count > HUB_DISPLAY_MAX_NODES
                    ? HUB_DISPLAY_MAX_NODES : st->node_count;
    unsigned online = 0;
    for (int i = 0; i < count; i++) {
        const int y = CARD_Y0 + i * CARD_STEP;
        const hub_display_node_t *n = &st->nodes[i];
        const bool known = n->online;
```

```
        const uint16_t accent = n->crying ? C_ALERT
                              : (known ? C_ACCENT : C_BORDER);
        /* The fills are the one thing swapped by hand.  The reader saw
         * the accent-washed fill as faint and the border-washed one as
         * the denser, more active surface - the opposite of what the
         * accent wash was meant to say - so a live card wears the
         * border-washed fill and a dead card the faint accent wash. */
        const uint16_t fill_col = n->crying ? C_ALERT
                                : (known ? C_BORDER : C_ACCENT);

        if (known) {
            online++;
        }

        fb_round_fill(CARD_X, y, CARD_W, CARD_H, 6,
                      blend(C_SURFACE, fill_col, n->crying ? 0.16f : 0.05f));
        /* The outline is the state, at full strength: a cot that is
         * answering gets the vibrant teal - rose, when it cries - and a
         * cot that has gone quiet gets the border washed towards the
         * background.  The first version had this backwards in effect:
         * the online outline was a 45 % blend while the no-signal cards
         * carried the plain border, so the dead cards read as the
         * livelier ones from the chair. */
        fb_round_rect(CARD_X, y, CARD_W, CARD_H, 6,
                      known || n->crying ? accent
                                         : blend(C_BORDER, C_BG, 0.45f));

        char name[8];
        snprintf(name, sizeof name, "COT %u", (unsigned)(i + 1));
        fb_text(CARD_X + 14, y + 8, name, known ? C_TEXT : C_DIM, 2);

        const char *state;
        uint16_t state_col;
        if (n->crying) {
            snprintf(right, sizeof right, "CRY %u%%", (unsigned)n->confidence);
            state = right;
            state_col = C_ALERT;
        } else if (n->deaf) {
            /* It is transmitting and the hub still cannot get its
             * audio, so it cannot announce a cry.  It is not being
             * watched, and the card has to say so. */
            state = "DEAF";
            state_col = C_ALERT;
        } else if (known) {
            state = "ONLINE";
            state_col = C_GREEN;
        } else {
            state = "OFFLINE";
            state_col = C_DIM;
        }
        fb_text(CARD_X + CARD_W - 14 - fb_text_w(state, 1), y + 11, state,
                state_col, 1);

        /* The RMS bar sits in the middle of the card, with its number
         * beside it.  The number used to share the bottom row with the
         * link, which put two readings on one line and made the card
         * hard to read at a glance in a dark room; the bar and its
         * number are a pair, and the link is another pair, and each
         * pair has a row of its own.
         *
         * The bar carries a notch at the cot's own trigger: the level
         * it sends audio over.  A cot under its notch is only being
         * listened to; a cot over it is shipping its audio to the hub
         * for the model to judge.  The number beside the bar is the
         * RMS itself — how loud the room is, right now — so the reader
         * can compare the two without guessing. */
        char rms[16];
        snprintf(rms, sizeof rms, "RMS %.3f", (double)(known ? n->rms : 0.0f));
        const int rms_w = fb_text_w(rms, 1);
        draw_level_bar(CARD_X + 14, y + 26, CARD_W - 28 - rms_w - 6, 8,
                       known ? n->rms : 0.0f, known ? n->trigger : 0.0f,
                       n->crying, known);
        fb_text(CARD_X + CARD_W - 14 - rms_w, y + 26, rms,
                known ? C_TEXT : C_DIM, 1);

        /* Bottom row: the radio link -- a small network bar and the
         * RSSI, or "--" when there is no reading. */
        const int bars = known ? rssi_bars(n->rssi) : 0;
        draw_signal_bars(CARD_X + 14, y + 58, bars);
        char link[24];
        if (!known || n->rssi == 0) {
            snprintf(link, sizeof link, "LINK --");
        } else {
            snprintf(link, sizeof link, "LINK %d dBm", (int)n->rssi);
        }
```

```
        fb_text(CARD_X + 14 + 4 * 5 + 4, y + 51, link, C_DIM, 1);
    }

    /* ---- the footer ----
     * The footer counts the cots that are actually answering, not the
     * cards on the screen.  A card for a cot that has gone quiet reads
     * NO SIGNAL, so "3 cots linked" printed above two NO SIGNAL cards
     * would be a lie — and the panel is the one place the caregiver is
     * certain to look. */
    char foot[40];
    if (online == 0) {
        snprintf(foot, sizeof foot, "no cots linked");
    } else {
        snprintf(foot, sizeof foot, "%u cot%s linked", online,
                 online == 1 ? "" : "s");
    }
    fb_text(12, FOOT_Y, foot, C_DIM, 1);

    char up[24];
    const uint32_t up_s = st->uptime_s;
    snprintf(up, sizeof up, "up %u:%02u", (unsigned)(up_s / 60),
             (unsigned)(up_s % 60));
    fb_text(FB_W - 12 - fb_text_w(up, 1), FOOT_Y, up, C_DIM, 1);
}

/* ------------------------------------------------------------------ */
/* The colour check                                                   */
/* ------------------------------------------------------------------ */

/* The palette, as swatches.
 *
 * A photograph of the panel cannot settle whether a colour is right:
 * the camera opens up for a glowing screen in a dark room, the
 * highlights clip, and the white balance drifts with what it sees.  The
 * first photograph of this panel came out with every colour exchanged
 * and the second came out blue, and neither could be trusted without
 * the swatches below to say what the panel was actually asked to draw.
 *
 * The grid is ordered so that the answers are unambiguous: black and
 * white at the ends of the first row, the accent in the middle, and
 * the alert colour last, where a red that is really a swapped blue
 * cannot hide.  Under it is a ramp of the darks, which is where this
 * panel does something unusual: a (12,20,32) navy comes back as a
 * saturated blue while a (0,0,0) black stays black, so the ramp says
 * in one row how far up the dark end the glass lifts. */
static void draw_colour_check(void)
{
    fb_fill(0, 0, FB_W, FB_H, C_BG);

    fb_fill(0, 0, FB_W, HEADER_H, C_HEADER);
    fb_text(12, 8, "COLOUR CHECK", C_TEXT, 2);
    fb_fill(0, HEADER_H, FB_W, 2, C_ACCENT_DIM);

    static const struct {
        const char *name;
        uint16_t colour;
    } swatches[] = {
        {"BLACK",  DISPLAY_RGB(0, 0, 0)},
        {"WHITE",  DISPLAY_RGB(255, 255, 255)},
        {"TEAL",   C_ACCENT},
        {"NAVY",   DISPLAY_RGB(12, 20, 32)},
        {"GREEN",  C_GREEN},
        {"AMBER",  C_AMBER},
        {"ROSE",   C_ALERT},
        {"GREY",   C_DIM},
    };
    const int n = (int)(sizeof swatches / sizeof swatches[0]);
    const int cols = 4;
    const int w = 74, h = 56, gap = 6;
    const int x0 = (FB_W - (cols * w + (cols - 1) * gap)) / 2;

    for (int i = 0; i < n; i++) {
        const int cx = x0 + (i % cols) * (w + gap);
        const int cy = 40 + (i / cols) * (h + 24);
        fb_fill(cx, cy, w, h, swatches[i].colour);
        fb_rect(cx, cy, w, h, C_BORDER);
        fb_text(cx + (w - fb_text_w(swatches[i].name, 1)) / 2, cy + h + 5,
                swatches[i].name, C_DIM, 1);
    }

    /* The dark ramp: the same hue, six steps.  The step that reads
     * brightest is the one the panel is lifting, and how it lifts is
     * the panel's own curve rather than the camera's. */
    static const uint16_t ramp[] = {
        DISPLAY_RGB(0, 0, 0),   DISPLAY_RGB(2, 4, 4),
```

```
        DISPLAY_RGB(4, 8, 8),   DISPLAY_RGB(7, 15, 16),
        DISPLAY_RGB(12, 26, 28), DISPLAY_RGB(24, 52, 56),
    };
    const int steps = (int)(sizeof ramp / sizeof ramp[0]);
    const int rw = (FB_W - 8) / steps;
    for (int i = 0; i < steps; i++) {
        fb_fill(4 + i * rw, 190, rw, 14, ramp[i]);
    }
    fb_rect(3, 189, rw * steps + 2, 16, C_BORDER);
    fb_text(4, 206, "the dark ramp", C_DIM, 1);

    char foot[48];
    snprintf(foot, sizeof foot, "backlight %u%%  -  press BOOT to go back",
             (unsigned)s_brightness);
    fb_text_centre(FB_H - 14, foot, C_DIM, 1);
}

/* ------------------------------------------------------------------ */
/* The help page                                                      */
/* ------------------------------------------------------------------ */

/* What the numbers on the dashboard mean.
 *
 * The question is the obvious one: the panel says RMS 0.023 next to a
 * bar, and a bar with a notch in it, and LINK -43 dBm, and none of
 * those words explain themselves.  A caregiver standing in front of
 * the panel at three in the morning cannot go and look up what an RMS
 * is, so the panel says it.
 *
 * It is reached from the dashboard with the BOOT button, next to the
 * colour check: the two pages are both for the person looking at the
 * panel, and neither is the running state.
 */
static void draw_help(void)
{
    fb_fill(0, 0, FB_W, FB_H, C_BG);

    fb_fill(0, 0, FB_W, HEADER_H, C_HEADER);
    fb_text(12, 8, "READING THE PANEL", C_TEXT, 2);
    fb_fill(0, HEADER_H, FB_W, 2, C_ACCENT_DIM);

    /* The body is set in the small font with a line every ten pixels:
     * twenty lines fit, and the wording is short because a caregiver
     * reads this once and then remembers it. */
    static const char *const lines[] = {
        "RMS 0.023   how loud the room is right now,",
        "            out of 1.0.  A quiet room is",
        "            near 0.005, a cry at one metre",
        "            is about 0.03 to 0.05.",
        "",
        "THE NOTCH    the level that cot sends audio",
        "            over.  Below it the cot only",
        "            listens; above it the hub gets",
        "            the audio and the model judges.",
        "            The bar turns amber there.",
        "",
        "LINK -43dBm  how strong the radio link to that",
        "            cot is.  -60 and better is strong,",
        "            -90 and worse is barely there.",
        "            The bars are the same number.",
        "",
        "CONF 98%     how sure the model was.",
    };
    const int n = (int)(sizeof lines / sizeof lines[0]);
    for (int i = 0; i < n; i++) {
        const uint16_t col = (lines[i][0] == '\0') ? C_BG
                           : (i < 4 ? C_TEXT : C_DIM);
        fb_text(12, 42 + i * 10, lines[i], col, 1);
    }

    fb_text_centre(FB_H - 14, "press BOOT to go back", C_DIM, 1);
}

/* ------------------------------------------------------------------ */
/* The task                                                           */
/* ------------------------------------------------------------------ */

static void flush_dirty(void)
{
    if (s_dirty_x1 < s_dirty_x0) {
        return;
    }
    const int x = s_dirty_x0, y = s_dirty_y0;
    const int w = s_dirty_x1 - s_dirty_x0 + 1;
    const int h = s_dirty_y1 - s_dirty_y0 + 1;
```

```
    s_dirty_x0 = FB_W; s_dirty_y0 = FB_H;
    s_dirty_x1 = -1; s_dirty_y1 = -1;
    ili9341_blit(&s_dev, x, y, w, h, &s_fb[(size_t)y * FB_W + x], FB_W);
}

static void display_task(void *arg)
{
    const TickType_t t0 = xTaskGetTickCount();
    uint32_t seen = 0;
    int frame = 0;
    TickType_t last_dash = 0;
    int button = 1;

    for (;;) {
        vTaskDelay(pdMS_TO_TICKS(SPLASH_TICK));

        /* The BOOT button cycles the panel through its three faces:
         * the dashboard, the page that explains the numbers on it, and
         * the colour check.  It is active low, so the press is the edge
         * from high to low. */
        const int pressed = gpio_get_level(PIN_BUTTON);
        if (button == 1 && pressed == 0) {
            s_face = (s_face + 1) % 3;
            ESP_LOGI(TAG, "the button was pressed: %s",
                     s_face == 0 ? "the dashboard"
                       : s_face == 1 ? "the help page" : "the colour check");
        }
        button = pressed;

        hub_display_state_t st;
        uint32_t version;
        portENTER_CRITICAL(&s_lock);
        st = s_state;
        version = s_version;
        portEXIT_CRITICAL(&s_lock);

        const TickType_t now = xTaskGetTickCount();
        const int64_t ms = (int64_t)(now - t0) * portTICK_PERIOD_MS;

        if (ms < SPLASH_MS) {
            draw_splash(ms, frame++);
            flush_dirty();
        } else if (ms < SPLASH_MS + FADE_MS) {
            draw_splash(ms, frame++);
            fb_fade(C_BG, (float)(ms - SPLASH_MS) / (float)FADE_MS);
            flush_dirty();
        } else if (ms < SPLASH_MS + 2 * FADE_MS) {
            draw_dashboard(&st);
            fb_fade(C_BG, 1.0f - (float)(ms - SPLASH_MS - FADE_MS) /
                            (float)FADE_MS);
            flush_dirty();
        } else if (s_face == 1) {
            draw_help();
            flush_dirty();
        } else if (s_face == 2) {
            draw_colour_check();
            flush_dirty();
        } else if (version != seen ||
                   ((now - last_dash) * portTICK_PERIOD_MS) > 1000) {
            seen = version;
            last_dash = now;
            draw_dashboard(&st);
            flush_dirty();
        }
    }
}

bool hub_display_start(void)
{
    if (!ili9341_init(&s_dev, PIN_MOSI, PIN_SCLK, PIN_CS, PIN_DC, PIN_RST,
                      PIN_BL, ROTATION_LANDSCAPE)) {
        return false;
    }

    /* The framebuffer.  320 x 240 RGB565 is 150 KB, which is more than
     * the hub has to spare in internal RAM, so it lives in PSRAM. */
    s_fb = heap_caps_malloc((size_t)FB_W * FB_H * sizeof(uint16_t),
                            MALLOC_CAP_SPIRAM);
    if (s_fb == NULL) {
        ESP_LOGE(TAG, "there is no room for the framebuffer");
        return false;
    }
    memset(s_fb, 0, (size_t)FB_W * FB_H * sizeof(uint16_t));

    s_dirty_x0 = FB_W; s_dirty_y0 = FB_H;
```

```
    s_dirty_x1 = -1; s_dirty_y1 = -1;

    /* The BOOT button, which opens the colour check.  It is pulled up
     * here: if the board has no button on it the pin simply stays
     * high, and the dashboard carries on as before. */
    gpio_config_t button = {
        .pin_bit_mask = 1ULL << PIN_BUTTON,
        .mode = GPIO_MODE_INPUT,
        .pull_up_en = GPIO_PULLUP_ENABLE,
    };
    gpio_config(&button);

    /* The backlight, at the level that does not glare. */
    ili9341_backlight_set(&s_dev, s_brightness);

    /* Core 1: the link loop and both radio stacks run on core 0, and a
     * 30 ms SPI burst must not delay a window analysis. */
    if (xTaskCreatePinnedToCore(display_task, "hubdisp", 4096, NULL,
                                3, NULL, 1) != pdPASS) {
        return false;
    }
    ESP_LOGI(TAG, "the display is up: %d x %d", FB_W, FB_H);
    return true;
}

void hub_display_update(const hub_display_state_t *state)
{
    portENTER_CRITICAL(&s_lock);
    s_state = *state;
    s_version++;
    portEXIT_CRITICAL(&s_lock);
}

void hub_display_set_brightness(uint8_t percent)
{
    if (percent > 100) {
        percent = 100;
    }
    s_brightness = percent;
    ili9341_backlight_set(&s_dev, percent);
    ESP_LOGI(TAG, "the backlight is at %u%%", (unsigned)percent);
}

uint8_t hub_display_brightness(void)
{
    return s_brightness;
}
```

The cot node

firmware/node/node_crynet/node_crynet.ino

1808 lines · 84,734 bytes

```
/*
 * CryNet cot node — ESP32-C3 Pro Mini + INMP441 I2S microphone.
 *
 * The board is the ESP32-C3 Pro Mini (the cot nodes moved off the original
 * SuperMini prototype on 2026-11-02; the SuperMini went to the wearable).
 * Nothing here is board-specific: the Pro Mini and the SuperMini are the
 * same ESP32-C3 chip with the same 4 MB flash and native USB, so the FQBN
 * weact_studio_esp32c3 covers both and this image is unchanged by the swap.
 *
 * Stage 2b.  The node is deliberately a dumb sensor.  It captures 16 kHz
 * mono PCM, keeps the last two seconds in RAM, reports a 100 ms RMS over
 * ESP-NOW, and uploads the two-second window as CRC-framed fragments when
 * the hub asks for it.  It never classifies anything: the model lives on
 * the hub, and the front end on the hub consumes the exact same PCM bytes
 * this node sends (the frozen signal contract).
 *
 * Wiring — INMP441 to ESP32-C3 Pro Mini (GPIO4/5/6 on either board):
 *     VCC -> 3V3        GND -> GND        L/R -> GND   (left channel)
 *     SCK -> GPIO4      WS  -> GPIO5      SD  -> GPIO6
 *
 * The INMP441 drives a standard I2S frame of two 32-bit slots, so the
 * peripheral is configured for 32-bit slots and the samples are converted
 * to int16 in software by keeping the top 16 bits of the 24-bit word.
 * That is the only place the mic's 24 bits are narrowed; everything after
 * this point is int16, exactly like the hub's input.
```

```
 *
 * Serial console — 115200 baud, one command per line:
 *     status   print the node state
 *     stats    print the counters
 *     freeze   freeze the live window now (and make it the upload target)
 *     burst    dump the frozen window as fragment frames, in the same
 *              framing the radio uses, so a PC can check it end to end
 *     id <n>   set the node id (1..255)
 *     help     list the commands
 *
 * Frames.  One framing is used on the radio and on the serial console:
 *
 *     0  0xC3            magic
 *     1  0x7E            magic
 *     2  type            1 = telemetry, 2 = burst fragment
 *     3  node id
 *     4  u16 burst seq   (telemetry: the sequence of the pending burst)
 *     6  u16 frag index  (telemetry: 0)
 *     8  u16 frag count  (telemetry: 0)
 *    10  u16 payload len
 *    12  u16 payload CRC-16/CCITT-FALSE
 *    14  payload
 *    ..  u16 frame CRC-16/CCITT-FALSE over everything above
 *
 * Requests from the hub are 12 bytes:
 *
 *     0  0xC3  1 0x7E  2 type=3  3 node id
 *     4  u16 burst seq   6 u16 first fragment   8 u16 fragment count
 *    10  u16 CRC-16/CCITT-FALSE
 *
 * The telemetry payload has three tiers, told apart by length.  The
 * first twelve bytes are the original ones: u32 rms x 1000, u16 window
 * ms, u8 flags, u8 reserved, u32 CRC-32 of the frozen window (0 when
 * there is none).  The hub recomputes that CRC over the window it
 * reassembles from the radio, so the two agreeing proves the audio
 * crossed the link intact.  Four more bytes follow on every node
 * flashed since the adaptive trigger: u16 event level x 1000, u16
 * frozen window level x 1000, u8 noise floor x 1000, then the same CRC
 * at the end.  The newest tier appends the node's own radio counters.
 *
 * There used to be a third tier between them, carrying the node's
 * battery cell voltage.  It is gone (D-83): the cots do not measure
 * their cells any more, so there is nothing to send, and the tier
 * would only have carried zeroes.  The counters move down to fill the
 * gap, which is why the hub reads them by length too.
 *
 * The hub reads by length, so every tier decodes on every hub.
 */

#include <Arduino.h>
#include <WiFi.h>
#include <esp_timer.h>
#include <esp_now.h>
#include <esp_wifi.h>
#include <esp_mac.h>
#include <ESP_I2S.h>
#include <Preferences.h>
#include <freertos/FreeRTOS.h>
#include <freertos/task.h>

/* ------------------------------------------------------------------ pins -- */

static const uint8_t PIN_SCK = 4;   /* INMP441 SCK -> I2S BCLK */
static const uint8_t PIN_WS  = 5;   /* INMP441 WS            */
static const uint8_t PIN_DIN = 6;   /* INMP441 SD  -> I2S in  */

/* There is no status LED on this board (D-83).  There used to be a
 * connection LED on GPIO10, driven from the hub's heartbeat, and it
 * is gone: the hub's panel already says which cots are answering, and
 * a second, slower copy of that answer on the cot itself was saying
 * the same thing in a place nobody was looking.  GPIO10 is free.
 */

/* ------------------------------------------------------------- contract -- */

static const uint32_t SAMPLE_RATE    = 16000;
static const uint32_t WINDOW_SAMPLES = 32000;              /* 2.000 s */
static const uint32_t BURST_BYTES    = WINDOW_SAMPLES * 2;  /* 64000  */
static const uint32_t TELEMETRY_MS   = 100;
static const uint32_t RMS_SAMPLES    = SAMPLE_RATE / 10;    /* 100 ms  */

/* A fragment frame is 16 bytes of header plus the payload plus 2 bytes of
 * CRC.  ESP-NOW carries at most 255 bytes in one frame, so 224 bytes of
 * payload leaves comfortable headroom. */
static const uint16_t FRAG_PAYLOAD = 224;
```

```
static const uint16_t FRAG_COUNT   = (BURST_BYTES + FRAG_PAYLOAD - 1) / FRAG_PAYLOAD;

/* The level at which the node freezes its two-second window, and the level
 * the hub is asked to trigger at.  It used to be a fixed 0.030 — the same
 * number as the hub's trigger — and the consequence was that a cry played
 * a metre from the node (median RMS 0.0070, not one of its 590 readings
 * over 0.030) was never offered to the model at all.  The system's reach
 * was set by a constant, not by the classifier.
 *
 * So the level is now relative to the node's own noise floor: three times
 * it.  The floor falls at once when the room gets quieter and rises very
 * slowly (FLOOR_RISE below), so a cry that lasts a minute cannot drag the
 * floor up to the cry's own level and switch the trigger off half way
 * through the episode.  The multiple and the clamps were chosen from the
 * recorded sessions by research/adaptive_trigger_study.py: at three times
 * the floor the cry a metre away crosses the level and the quiet room
 * does not.
 *
 * The upper clamp is the fixed level this replaces, so the rule is never
 * worse than it was in a loud room.  The lower clamp keeps the level
 * above the microphone's own noise, where a trigger would mean nothing. */
static const float TRIGGER_MULT = 3.0f;
static const float TRIGGER_MIN = 0.005f;
static const float TRIGGER_MAX = 0.030f;

/* How much of the way towards a louder reading the floor moves each time.
 * There are ten readings a second, so 0.002 is a fifty-second time
 * constant: long enough that a cry does not raise the floor, short enough
 * that the room's own noise is followed within a minute. */
static const float FLOOR_RISE = 0.002f;

/* The ESP-NOW channel.  The hub uses the same one. */
static const uint8_t LINK_CHANNEL = 6;

/* How many cots this hub watches.  The hub ignores every other node
 * number, so a node must be 1 or 2 to be heard at all. */
static const uint8_t CRYNET_COT_COUNT = 2;

/* The frame types on the radio.  These mirror crynet_proto.h on the hub;
 * both sides were written from the same table. */
#define LINK_TYPE_TELEMETRY 1
#define LINK_TYPE_FRAGMENT  2
#define LINK_TYPE_REQUEST   3
#define LINK_TYPE_HEARTBEAT 4
#define LINK_TYPE_RESEND    5

/* The bitmap in a resend request: one bit per fragment. */
#define RESEND_BITMAP_BYTES ((FRAG_COUNT + 7) / 8)

/* How long after the loudest moment the window is frozen.
 *
 * The ring holds two seconds, so whatever is frozen is always the two
 * seconds that ended at the moment of the freeze.  This constant
 * therefore decides *which* two seconds the hub classifies:
 *
 *   PEAK_DELAY_MS = 0    ->  the two seconds BEFORE the peak
 *   PEAK_DELAY_MS = 500  ->  the peak in the middle of the window
 *   PEAK_DELAY_MS = 2000 ->  the two seconds AFTER the peak
 *
 * It used to be 500, so the window was the peak in the middle of it and
 * the second half of the cry was in it.  That was defended by the
 * belief that "the model was trained on two-second windows in which the
 * cry is wherever it fell, so a cry that sits a little later in the
 * window is judged the same".  On 2026-11-04 that belief was measured,
 * on every clip of the in-domain sessions, with the deployed INT8 model
 * (research/screen_clips.py):  it is false for a cry that has only just
 * started.  P(cry) against the amount of continuous cry inside the
 * two-second window:
 *
 *     0.50 s -> 0.004 .. 0.36      1.25 s -> 0.10 .. 0.95
 *     1.00 s -> 0.02  .. 0.23      1.50 s -> 0.21 .. 0.99
 *                                  1.75 s -> 0.77 .. 0.9961
 *                                  2.00 s -> 0.9961
 *
 * The model needs roughly 1.5 to 1.75 seconds of continuous cry inside
 * the window; half a window of cry is refused however loud it is.  A
 * 500 ms delay freezes the peak in the middle of the window, so a cry
 * that starts at the peak puts only about half a second of cry in it
 * -- and the window is refused.  That is a structural cause of the
 * misses, and of the misses that looked like "the node is not
 * detecting" on 2026-11-03.
 *
 * So the window is now the two seconds AFTER the peak: for a cry that
 * lasts longer than two seconds -- which is every real cry and every
 * demo clip in research/out/demo_kit/ -- the whole window is cry, and
```

```
 * the model scores it 0.9961.  The price is honest and stated: the
 * freeze now happens two seconds after the peak instead of half a
 * second after it, so the detection is about 1.5 s later.
 *
 * That price was measured against the whole corpus on 2026-11-04
 * (research/simulate_freeze.py simulates the node's own records, the
 * freeze and the hub's gates over every in-domain session):
 *
 *     rule                room windows   false alarms   cry episodes caught
 *     peak in the middle        104            0                 52
 *     two seconds after          69            0                 41
 *     stop-capped (below)       100            0                 52
 *
 * The two seconds after the peak lose the short cries: a yelp that ends
 * before the delay has passed would be frozen over the room, not over
 * the cry.  So the window is now frozen when the cry STOPS -- the
 * streak breaks -- and the two seconds after the peak are only the cap
 * for a cry that goes on.  A short yelp is caught whole; a sustained
 * cry still gets the two seconds that follow its peak.
 *
 * PEAK_STOP_DELAY_MS is the shortest wait before a stopped cry is
 * frozen: the window has to hold the cry itself, so the freeze cannot
 * happen on the first dip below the level.
 *
 * PEAK_EXPIRE_MS has to outlast the delay: the peak must still be
 * alive when the window is frozen, or the freeze never happens at all.
 */
static const uint32_t PEAK_DELAY_MS = 2000;
static const uint32_t PEAK_STOP_DELAY_MS = 500;
static const uint32_t PEAK_EXPIRE_MS = 4000;

/* How long the node holds a frozen window after it has been uploaded.
 *
 * The upload is not the end of the conversation about that window: the
 * hub asks for the fragments it missed, and only the node can send them
 * again.  The node used to freeze the next window as soon as its live
 * window refilled, which is about three seconds after the upload -- and
 * the hub's first request for the missing fragments often arrived just
 * after that, when the node had already thrown the old window away.
 * The node refused those requests ("no frozen window for it"), the hub
 * ran out of rounds, and the burst timed out.
 *
 * So the snapshot is held for four seconds after the upload.  Four
 * seconds covers the burst itself and the first resend round.  It was
 * raised to eight seconds for the queue problem (D-67), but cry mode
 * (D-69) has since capped the freezes at one per cry segment, so the
 * hold no longer has queue duty -- it is pure resend coverage now, and
 * the extra four seconds was only adding wait between windows: the
 * three-minute session of 2026-11-01 (late) measured 6.5-7.0 s of
 * hold-driven gap between every burst.  (D-71.)  It was cut from four
 * seconds to three when the burst stopped taking three and a half
 * seconds to cross the air: the resend round the hold has to cover now
 * starts about a second and a half after the upload, not four. */
static const uint32_t FROZEN_HOLD_MS = 3000;

/* How long a window the hub is still fetching may not be taken away by
 * a louder moment.
 *
 * The hold above is three seconds and the preempt rule -- a new window
 * that is three times louder than the frozen one takes its place -- was
 * allowed to cancel it at any moment.  In a noisy room that is exactly
 * what happens: the room is loud enough that each new reading is three
 * times the last window's level, so the node froze a new window every
 * second or two, and every freeze threw away the fragments the hub was
 * still missing.  The hub could never finish a burst at such a cot: it
 * asked for the pieces that were missing, the node had already thrown
 * them away, and the burst died with a third of it on the floor.
 *
 * So a window that is being fetched is now protected.  The protection
 * is refreshed every time the node sends a fragment, so it lasts while
 * the burst is crossing the air -- which is the whole of a slow cot's
 * transfer -- and it ends TRANSFER_HOLD_MS after the last fragment the
 * node sent.  That is a little longer than the gap between the hub's
 * resend rounds, so the window survives the wait for the next round,
 * and it is about as long as the hold that was there before, so a cot
 * is not left unable to freeze a new window any longer than it was. */
static const uint32_t TRANSFER_HOLD_MS = 4000;

/* How long a frozen window may sit unserved before it is dropped.
 *
 * It has to outlast the whole conversation about one window: the burst
 * itself takes nearly four seconds to send, and the hub may then ask
 * for the fragments that did not arrive.  Five seconds was not enough
 * for that -- the window expired while the hub was still waiting for
 * the pieces it had asked for. */
```

```
static const uint32_t FROZEN_EXPIRE_MS = 20000;

/* ------------------------------------------------------------- battery -- */

/* There is no battery sense on this board any more (D-83).  There used
 * to be one: a 220 kohm resistor from B+ to GPIO3 and a 100 kohm
 * resistor, with a 100 nF across it, from GPIO3 to B- (SYSTEM_WIRING.md
 * 3.3), read as ADC1_CH3 and reported to the hub, which put a battery
 * band on the panel.
 *
 * The whole feature is gone -- the divider, the reading, the trust
 * gate around it, the telemetry field and the panel's battery text.
 * The reason is not that the reading was wrong: it was a voltage
 * estimate printed with a tilde, and it was honest about that.  The
 * reason is that the owner does not want it, and the hub's panel
 * already shows what the caregiver needs to know.  Nothing about the
 * battery *power* has changed: the cell, the module and the switch
 * are still what they were.
 *
 * GPIO3 is ADC1_CH3 and is now free.  Nothing drives it, and nothing
 * reads it. */

/* ---------------------------------------------------------------- state -- */

static I2SClass  g_i2s;
static int32_t   g_raw[512];            /* I2S staging, 32-bit slots      */
static int16_t   g_window[WINDOW_SAMPLES]; /* live, circular, chronological */
static int16_t   g_frozen[WINDOW_SAMPLES]; /* snapshot offered for upload */
static uint32_t  g_write = 0;           /* next write index into g_window */
static uint32_t  g_filled = 0;          /* valid samples, up to 32000     */
static uint32_t  g_frozen_fill = 0;     /* valid samples in g_frozen      */

static bool      g_frozen_valid = false;
/* The frozen window has already been uploaded once.  It stays on the
 * node afterwards -- the hub may ask for the fragments it missed -- but
 * it must not stop the node freezing the *next* window, which is what
 * g_frozen_valid used to guard on its own. */
static bool      g_frozen_served = false;
static uint32_t  g_frozen_hold_ms = 0;   /* when the hold on it ends */
/* When the hub last asked this node for the window it froze.  A window
 * that is being fetched is not taken away by a louder moment until
 * TRANSFER_HOLD_MS has passed without a word from the hub. */
static uint32_t  g_transfer_ms = 0;
static bool      g_peak_pending = false;   /* a loud moment is waiting to
                                              be frozen, one second late */
static bool      g_cry_mode = false;       /* a loud window is frozen; do
                                              not freeze again until the
                                              room goes quiet, so a cry
                                              does not build a queue */
static uint32_t  g_peak_ms = 0;
static uint32_t  g_frozen_ms = 0;
static uint32_t  g_frozen_crc = 0;    /* CRC-32 of the frozen window's bytes */
static float     g_frozen_rms = 0.0f; /* the frozen window's own level */
static uint16_t  g_burst_seq = 0;
static uint8_t   g_node_id = 1;
static Preferences g_prefs;   /* a serial-set node id survives reboots here */

static uint16_t  g_telemetry_seq = 0;
static uint32_t  g_last_tick = 0;
static float     g_rms = 0.0f;
static int32_t   g_mean = 0;

/* The room's noise floor, the event level that follows from it, and how
 * many readings in a row have been at or above that level.  The floor is
 * negative until the first reading arrives, which is what sets it. */
static float     g_floor = -1.0f;
static float     g_level = 0.006f;
static uint8_t   g_loud_streak = 0;

static uint32_t  g_frames_sent = 0;
static uint32_t  g_frames_lost = 0;
static uint32_t  g_requests = 0;
static uint32_t  g_heartbeats = 0;

/* The hub's signal as the node hears it: the RSSI of the last frame the
 * hub sent.  Its heartbeat arrives every 1000 ms, so this is a live
 * reading, and it is the node's own view of the link -- the mirror of
 * the reading the hub takes of the node's telemetry.  Zero means no hub
 * frame has been heard yet. */
static int8_t     g_hub_rssi = 0;
static uint32_t   g_hub_rssi_ms = 0;   /* millis() of the last hub frame */
static uint32_t  g_freezes = 0;

static bool      g_dump_active = false;  /* suppress text while dumping */

```

```
/* Set from the radio callback, acted on in the loop.  The callback runs in
 * the WiFi task, and sending 286 fragments from there would block the
 * radio for seconds. */
static volatile bool     g_request_pending = false;
static volatile uint16_t g_request_seq = 0;
static volatile uint16_t g_request_first = 0;
static volatile uint16_t g_request_count = 0;

/* ------------------------------------------------------------------ crc16 -- */

static uint16_t crc16(const uint8_t *data, size_t len) {
    uint16_t crc = 0xFFFF;
    for (size_t i = 0; i < len; i++) {
        crc ^= (uint16_t)data[i] << 8;
        for (int bit = 0; bit < 8; bit++) {
            crc = (crc & 0x8000u) ? (uint16_t)((crc << 1) ^ 0x1021u)
                                  : (uint16_t)(crc << 1);
        }
    }
    return crc;
}

/* CRC-32 (the IEEE 802.3 one: reflected, polynomial 0xEDB88320, the
 * final XOR of 0xFFFFFFFF that zlib's crc32 applies).  The hub computes
 * the same value over the window it reassembles from the radio, so the two
 * agreeing proves the audio crossed the link unchanged. */
static uint32_t crc32(const uint8_t *data, size_t len) {
    uint32_t crc = 0xFFFFFFFFu;
    for (size_t i = 0; i < len; i++) {
        crc ^= data[i];
        for (int bit = 0; bit < 8; bit++) {
            crc = (crc & 1u) ? ((crc >> 1) ^ 0xEDB88320u) : (crc >> 1);
        }
    }
    return crc ^ 0xFFFFFFFFu;
}

/* ---------------------------------------------------------------- frames -- */

static uint8_t g_frame[256];

/* Build one frame into g_frame and return its total length. */
static size_t build_frame(uint8_t type, uint16_t seq, uint16_t frag,
                          uint16_t count, const uint8_t *payload, uint16_t plen) {
    g_frame[0] = 0xC3;
    g_frame[1] = 0x7E;
    g_frame[2] = type;
    g_frame[3] = g_node_id;
    g_frame[4] = (uint8_t)(seq & 0xFF);
    g_frame[5] = (uint8_t)(seq >> 8);
    g_frame[6] = (uint8_t)(frag & 0xFF);
    g_frame[7] = (uint8_t)(frag >> 8);
    g_frame[8] = (uint8_t)(count & 0xFF);
    g_frame[9] = (uint8_t)(count >> 8);
    g_frame[10] = (uint8_t)(plen & 0xFF);
    g_frame[11] = (uint8_t)(plen >> 8);
    uint16_t pcrc = (plen > 0) ? crc16(payload, plen) : 0;
    g_frame[12] = (uint8_t)(pcrc & 0xFF);
    g_frame[13] = (uint8_t)(pcrc >> 8);
    if (plen > 0) {
        memcpy(&g_frame[14], payload, plen);
    }
    size_t total = 14 + (size_t)plen;
    uint16_t fcrc = crc16(g_frame, total);
    g_frame[total] = (uint8_t)(fcrc & 0xFF);
    g_frame[total + 1] = (uint8_t)(fcrc >> 8);
    return total + 2;
}

/* ---------------------------------------------------------- the window -- */

/* Exact mean over the valid part of the live window.  The sum is an
 * integer: a float accumulator loses a few counts on 32000 samples and
 * the residual DC is what the front end's standardisation then amplifies
 * (this is exactly the bug the hub's on-device probes caught).
 *
 * The valid region is the newest `valid` samples, so it is walked
 * backwards from the write cursor.  That matters after a burst: the live
 * window is dropped and refilled, and until it is full again the newest
 * samples are not at the start of the array. */
static int32_t window_mean(uint32_t valid) {
    int64_t sum = 0;
    uint32_t idx = g_write;
    for (uint32_t k = 0; k < valid; k++) {
        if (idx == 0) {
```

```
            idx = WINDOW_SAMPLES;
        }
        idx--;
        sum += (int64_t)g_window[idx];
    }
    return (int32_t)(sum / (int64_t)valid);
}

/* RMS of the newest n samples, using the mean of the whole window so that
 * the number matches what the hub's front end will see for the same
 * window.  Returned in the same normalised units as the hub: the sample
 * divided by 32768 after the mean is removed. */
static float rms_of_last(int32_t mean, uint32_t n) {
    if (n == 0) {
        return 0.0f;
    }
    uint32_t idx = (g_write + WINDOW_SAMPLES - n) % WINDOW_SAMPLES;
    int64_t acc = 0;
    for (uint32_t k = 0; k < n; k++) {
        int32_t d = (int32_t)g_window[idx] - mean;
        acc += (int64_t)d * d;
        if (++idx == WINDOW_SAMPLES) {
            idx = 0;
        }
    }
    double m = (double)acc / (double)n;
    return (float)(sqrt(m) / 32768.0);
}

/* The RMS of a buffer, with its own mean removed — the same thing the
 * hub's front end does to a window before it makes the features out of
 * it.  Used on the frozen snapshot, so that the level the model is shown
 * can be compared with the level of the room at the time. */
static float buffer_rms(const int16_t *buf, uint32_t n) {
    if (n == 0) {
        return 0.0f;
    }
    int64_t sum = 0;
    for (uint32_t i = 0; i < n; i++) {
        sum += (int32_t)buf[i];
    }
    const int32_t mean = (int32_t)(sum / (int64_t)n);
    int64_t acc = 0;
    for (uint32_t i = 0; i < n; i++) {
        const int32_t d = (int32_t)buf[i] - mean;
        acc += (int64_t)d * d;
    }
    return (float)(sqrt((double)acc / (double)n) / 32768.0);
}

/* Copy the live window into the frozen snapshot, oldest sample first. */
static void freeze_window(void) {
    for (uint32_t i = 0; i < WINDOW_SAMPLES; i++) {
        uint32_t idx = g_write + i;
        if (idx >= WINDOW_SAMPLES) {
            idx -= WINDOW_SAMPLES;
        }
        g_frozen[i] = g_window[idx];
    }
    g_frozen_fill = WINDOW_SAMPLES;
    g_frozen_valid = true;
    g_frozen_served = false;
    g_frozen_ms = millis();
    /* The level of the window that is being offered.  The hub reports it
     * beside the live reading, and the two together say whether the window
     * that was judged was the loud one or a quiet stretch next to it. */
    g_frozen_rms = buffer_rms(g_frozen, WINDOW_SAMPLES);
    /* The checksum of the exact bytes that will be uploaded.  The hub
     * computes the same value over the window it reassembles, so the two
     * agreeing is the proof that the radio delivered the audio intact. */
    g_frozen_crc = crc32((const uint8_t *)g_frozen, BURST_BYTES);
    g_burst_seq++;
    g_freezes++;
}

/* Sample i of the upload target, oldest first. */
static int16_t burst_sample(uint16_t i, bool from_frozen) {
    if (from_frozen) {
        return g_frozen[i];
    }
    uint32_t idx = g_write + i;
    if (idx >= WINDOW_SAMPLES) {
        idx -= WINDOW_SAMPLES;
    }
    return g_window[idx];
```

```
}

/* ---------------------------------------------------------------- i2s -- */

static bool i2s_start(void) {
    g_i2s.setPins(PIN_SCK, PIN_WS, -1, PIN_DIN);
    g_i2s.setTimeout(500);
    /* STEREO with both slots — the peripheral must be given the whole
     * 64-bit frame.
     *
     * The INMP441 drives a 64-bit I2S frame of two 32-bit slots and puts
     * its 24-bit sample, left-justified, in the left one.  Asking for
     * I2S_SLOT_MODE_MONO halves the frame to 32 bits, the microphone's
     * framing breaks and the samples come out misaligned: measured on the
     * real board, -3500 counts with the full frame against -620 with the
     * halved one.  Asking for stereo also makes the Arduino library force
     * the both-slots mask whatever mask is given, so the DMA delivers the
     * left and the right slot interleaved either way; the right one is
     * dropped in i2s_pump().  node_diag prints both side by side. */
    if (!g_i2s.begin(I2S_MODE_STD, SAMPLE_RATE, I2S_DATA_BIT_WIDTH_32BIT,
                     I2S_SLOT_MODE_STEREO, I2S_STD_SLOT_BOTH)) {
        return false;
    }
    /* The INMP441's first samples are garbage: measured on the real board,
     * the first 100 ms window after begin() is a full-scale burst (the
     * capture node's RMS field reads 656 milli against the room's 6-11)
     * and it decays over about half a second.  Without this, every boot
     * would freeze a garbage window the moment the burst crossed the EVENT
     * level and hand the hub a spurious upload.  Discard 500 ms; the
     * slower DC drift that remains is removed by the mean. */
    static const uint32_t WARMUP_FRAMES = SAMPLE_RATE / 2;   /* 500 ms */
    uint32_t warmed = 0;
    const uint32_t started = millis();
    while (warmed < WARMUP_FRAMES) {
        if (millis() - started >= 3000u) {
            return false;
        }
        size_t got = g_i2s.readBytes((char *)g_raw, sizeof(g_raw));
        got -= got % 8;                 /* whole stereo frames only */
        warmed += got / 8;
    }
    return true;
}

/* Read whatever the microphone has produced and append it to the window.
 *
 * A read that comes back short means the microphone's buffer ran dry --
 * the driver waited and the samples did not arrive -- so the audio has a
 * hole in it and the live window can no longer be trusted.  That is
 * flagged here and acted on by the caller. */
static uint32_t g_loop_iters = 0;
static uint32_t g_loop_max_us = 0;
static uint64_t g_t_pump = 0, g_t_tick = 0, g_t_req = 0, g_t_rest = 0;
static uint32_t g_pump_calls = 0;
static uint32_t g_pump_empty = 0;   /* calls that returned no whole frame */
static uint32_t g_pump_frames = 0;
static uint32_t g_pump_max_us = 0;
static volatile bool     g_window_damaged = false;  /* the live window has a hole */
static volatile uint32_t g_i2s_restarts = 0;
static uint32_t g_zero_samples = 0;
static const uint32_t I2S_ZERO_LIMIT = SAMPLE_RATE * 5u; /* five seconds */
static size_t i2s_pump(void) {
    const int64_t t0 = esp_timer_get_time();
    size_t got = g_i2s.readBytes((char *)g_raw, sizeof(g_raw));
    const int64_t dt = esp_timer_get_time() - t0;
    g_t_pump += (uint64_t)dt;
    if (dt > (int64_t)g_pump_max_us) {
        g_pump_max_us = (uint32_t)dt;
    }
    g_pump_calls++;
    if (got < sizeof(g_raw)) {
        g_zero_samples = 0; /* missing frames are not consecutive zeros */
        /* The microphone's buffer ran dry: what is in the window is no
         * longer continuous audio. */
        g_window_damaged = true;
    }
    if (got < 8) {
        g_pump_empty++;
    }
    g_pump_frames += got / 8;
    got -= got % 8;                 /* whole stereo frames only */
    size_t frames = got / 8;
    for (size_t f = 0; f < frames; f++) {
        /* The left slot holds the microphone's 24-bit sample, left
         * justified, so the top 16 bits are the full-scale int16 value.
```

```
         * The right slot is a constant and is dropped. */
        int16_t s = (int16_t)(g_raw[f * 2] >> 16);
        if (s == 0) {
            if (g_zero_samples < I2S_ZERO_LIMIT) {
                g_zero_samples++;
            }
        } else if (g_zero_samples < I2S_ZERO_LIMIT) {
            g_zero_samples = 0;
        }
        g_window[g_write] = s;
        g_write++;
        if (g_write == WINDOW_SAMPLES) {
            g_write = 0;
        }
        if (g_filled < WINDOW_SAMPLES) {
            g_filled++;
        }
    }
    return frames;
}

/* The microphone's own task.
 *
 * The I2S read blocks until the samples are there, so reading the
 * microphone from the main loop stopped the main loop for as long as the
 * audio took to arrive -- sixteen milliseconds, every time.  That was
 * harmless while a burst took three and a half seconds, and it was not
 * harmless at all once the radio learned to carry a burst in a fraction
 * of a second: then the microphone's real-time rate, and not the radio's,
 * decided how long a burst took, and the system got slower than it had
 * ever been.
 *
 * So the microphone has a task of its own.  It wakes sixteen times a
 * second, takes what the hardware has produced, and puts it in the
 * window.  The main loop never waits for audio: the radio is fed as
 * fast as the radio can carry it, and the node keeps listening while it
 * is talking. */
static void capture_task(void *arg)
{
    (void)arg;
    for (;;) {
        i2s_pump();
        /* Exact zero samples for five seconds are not a quiet room: the
         * microphone normally has noise.  Restart I2S from its owning task;
         * invalidate the live ring, but keep any frozen upload intact. */
        if (g_zero_samples >= I2S_ZERO_LIMIT) {
            g_zero_samples = 0;
            g_window_damaged = true;
            g_i2s_restarts++;
            if (!g_i2s.end() || !i2s_start()) {
                /* A failed driver teardown or warmup cannot be repaired
                 * safely by reusing its handles.  Reboot instead. */
                ESP.restart();
            }
            g_window_damaged = true;
        }
        /* The read itself paces this task: it returns when the samples
         * have arrived. */
    }
}

/* ------------------------------------------------------------------ radio -- */

static uint8_t g_bcast[6] = {0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF};

/* The hub's address, learned from the request the node receives: the hub
 * broadcasts its requests, and the node replies to the address they came
 * from.  Unicast is used once the hub is known, because a broadcast
 * destination is filtered more aggressively by the receiver's radio. */
static uint8_t g_hub_mac[6] = {0};
static bool    g_hub_known = false;
/* Set from the receive callback, acted on in the loop: the callback
 * runs in the WiFi task, and esp_now_add_peer() waits for that very
 * task, which wedged the radio — the send callback stopped firing and
 * every radio_send burned its two-second cap.  The hub's node_link
 * carries the same rule: peers are added from a normal task. */
static volatile bool g_hub_peer_pending = false;
static uint8_t       g_pending_mac[6];

/* A resend request from the hub: the fragments of a burst that never
 * arrived.  The bitmap says which ones. */
static bool     g_resend_pending = false;
static uint16_t g_resend_seq = 0;
static uint8_t  g_resend_bitmap[RESEND_BITMAP_BYTES];

```

```
/* The request the node answered last, and when.  The hub sends its
 * request three times, because a broadcast is never acknowledged; the
 * node must answer it once.  A repeat of the request it is already
 * serving is dropped, or the node would send the same hundred and
 * twenty-eight fragments two or three times over and the hub would see
 * a burst that never settles. */
static uint16_t  g_resend_last_seq = 0;
static uint32_t  g_resend_last_ms = 0;
/* The bitmap of that request, so a repeat can be told apart from a
 * *new* round. */
static uint8_t   g_resend_last_bitmap[RESEND_BITMAP_BYTES];

/* How close together two requests for the same window have to be for
 * the second to count as a repeat of the first.
 *
 * This was 800 ms, and it was longer than the hub's own round of
 * resends -- the hub asks again about 670-770 ms after the last ask --
 * so the node threw away every round after the first as a "repeat" of
 * the one it was answering.  The consequence was fatal: a burst that
 * lost one fragment could never be completed.  The hub asked for the
 * missing fragment five times, the node answered none of the five
 * requests, the burst sat at 285 of 286 until it timed out, and the
 * whole two-second window -- the one that held the cry -- was thrown
 * away.  A burst that is one frame short is the *commonest* failure
 * on a weak link, so this was the single most expensive line in the
 * firmware.
 *
 * The rule is now two-sided: a request is a repeat only if it names
 * the same window AND the same fragments AND arrives within the window
 * below.  A round that asks for a *different* fragment is a new round
 * and is always answered. */
static const uint32_t RESEND_DEDUPE_MS = 250;

/* The radio's transmit callback.  It fires when a frame has actually left
 * the queue.  The sender counts the frames it has handed over against the
 * frames that have left, so it always knows how many the radio is still
 * holding.
 *
 * ESP-NOW's queue is ten frames deep.  A burst is 286 fragments, so they
 * cannot all be handed over at once: every frame past the tenth would be
 * lost for good.  The sender used to wait for each frame's own
 * acknowledgement before handing over the next one, and that cost about
 * twelve milliseconds a fragment -- three and a half seconds for a burst,
 * which was most of the time the system took to answer a cry.  It now
 * keeps a few frames in the radio at once and waits for a group of them,
 * which is what the queue is for. */
static volatile uint32_t g_tx_acks = 0;
static uint32_t g_tx_queued = 0;   /* handed over, not yet acknowledged */
/* How many frames the radio itself could not carry.  The callback's
 * status used to be thrown away, so this was always zero and the node
 * reported a clean sheet even when the radio was dropping its burst. */
static volatile uint32_t g_tx_failed = 0;

/* How many frames the radio may be holding at once.  The queue is ten
 * deep; four leaves room for the telemetry and the heartbeats that share
 * it, and keeps the far end's receive queue from filling up -- one
 * fragment in three hundred was being lost on the way to the hub, and
 * the far end's queue overflowing is the likeliest reason. */
#define TX_INFLIGHT_MAX 4

/* How many frames the radio is holding right now. */
static uint32_t tx_inflight(void)
{
    return g_tx_queued - g_tx_acks;
}

/* How long the sender waits for the radio to take each frame.  A frame
 * that is acknowledged in a millisecond was handed to the driver, not put
 * on the air; a frame that takes a hundred milliseconds was transmitted
 * and acknowledged. */
static uint32_t g_send_max_us = 0;   /* time inside esp_now_send() */
static uint32_t g_send_calls = 0;
static uint32_t g_tx_wait_us = 0;
static uint32_t g_tx_wait_n = 0;
static uint32_t g_tx_wait_max = 0;

/* How many times the radio had no room and the sender had to wait. */
static uint32_t g_tx_full = 0;

static void on_send(const esp_now_send_info_t *info, esp_now_send_status_t status) {
    (void)info;
    /* The frame is finished either way, so it leaves the in-flight
     * count.  But the status says whether the radio could carry it, and
     * it used to be thrown away: the node's own counters said "nothing
     * lost" no matter what the radio did, which is how a cot whose
```

```
     * fragments never arrive can still report a clean sheet. */
    g_tx_acks++;
    if (status != ESP_NOW_SEND_SUCCESS) {
        g_tx_failed++;
    }
}

static void on_recv(const esp_now_recv_info_t *info, const uint8_t *data, int len) {
    if (len < 12 || data[0] != 0xC3 || data[1] != 0x7E) {
        return;
    }
    /* The hub's signal as the node hears it: the RSSI of a frame the hub
     * sent.  Only the hub's own frame types count — a request, a
     * heartbeat, a resend request.  The telemetry and the window
     * fragments are the *other* cot's traffic, which the node also
     * hears because everything is broadcast, and reading the signal
     * strength of another cot's frame would report the loudest cot in
     * the room rather than the hub. */
    if (info != NULL && info->rx_ctrl != NULL &&
        (data[2] == LINK_TYPE_REQUEST || data[2] == LINK_TYPE_HEARTBEAT ||
         data[2] == LINK_TYPE_RESEND)) {
        g_hub_rssi = (int8_t)info->rx_ctrl->rssi;
        g_hub_rssi_ms = millis();
    }
    /* A heartbeat (type 4) is the hub keeping its radio awake.  It asks
     * for nothing, so the node's window is left alone. */
    /* Learn the hub's address from any frame it sends — requests and
     * heartbeats both — and add it as a peer so the reply can be
     * unicast to it.  The heartbeat arrives every 1000 ms, so the
     * address is learned within about a second of boot; waiting
     * for a request meant that a hub which had already heard the node
     * (and so had stopped probing) never told the node where to send. */
    if (!g_hub_known && !g_hub_peer_pending && info != NULL &&
        (data[2] == LINK_TYPE_REQUEST || data[2] == LINK_TYPE_HEARTBEAT ||
         data[2] == LINK_TYPE_RESEND)) {
        memcpy(g_pending_mac, info->src_addr, 6);
        g_hub_peer_pending = true;
    }
    /* A heartbeat (type 4) is the hub keeping its radio awake.  It asks
     * for nothing, so the node's window is left alone. */
    if (data[2] == LINK_TYPE_HEARTBEAT) {
        g_heartbeats++;
        return;
    }

    /* A resend request: the hub's list of the fragments of a burst that
     * never arrived.  The payload is a bitmap, one bit per fragment,
     * and the node answers by sending exactly the fragments whose bit
     * is set -- out of the same frozen window, with the same sequence
     * number, so the pieces that did arrive are not sent again. */
    if (data[2] == LINK_TYPE_RESEND) {
        const uint16_t plen = (uint16_t)data[10] | ((uint16_t)data[11] << 8);
        const size_t total = 14u + (size_t)plen;
        if (plen != RESEND_BITMAP_BYTES || len != (int)(total + 2u)) {
            g_frames_lost++;
            return;
        }
        /* The frame's own CRC first: the bitmap is only worth reading
         * if the frame is intact. */
        const uint16_t fcrc = (uint16_t)data[total] | ((uint16_t)data[total + 1] << 8);
        if (crc16(data, total) != fcrc) {
            g_frames_lost++;
            return;
        }
        const uint16_t pcrc = (uint16_t)data[12] | ((uint16_t)data[13] << 8);
        if (crc16(&data[14], plen) != pcrc) {
            g_frames_lost++;
            return;
        }
        if (data[3] != g_node_id && data[3] != 0xFF) {
            return;                       /* not for us */
        }
        const uint16_t seq = (uint16_t)data[4] | ((uint16_t)data[5] << 8);
        const uint32_t now = millis();
        /* A repeat is the same window, the same fragments, and close in
         * time.  A round that asks for a fragment the last round did
         * not ask for is a new round: it is answered, however soon it
         * follows the last one. */
        const bool same_round =
            (seq == g_resend_last_seq) &&
            (memcmp(&data[14], g_resend_last_bitmap, RESEND_BITMAP_BYTES) == 0);
        if (same_round && (now - g_resend_last_ms) < RESEND_DEDUPE_MS) {
            /* A repeat of the request already being answered. */
            return;
        }
```

```
        g_resend_last_seq = seq;
        g_resend_last_ms = now;
        memcpy(g_resend_last_bitmap, &data[14], RESEND_BITMAP_BYTES);
        g_resend_pending = true;
        g_resend_seq = seq;
        memcpy(g_resend_bitmap, &data[14], RESEND_BITMAP_BYTES);
        if (Serial) {
            /* How many fragments does the hub say it is missing?  The
             * bitmap is counted here rather than assumed, because a
             * hub that asks for 274 fragments and a node that reads
             * twelve bits out of the same request is a disagreement
             * that has to be visible in the log to be findable. */
            uint32_t asked = 0;
            for (uint16_t b = 0; b < RESEND_BITMAP_BYTES; b++) {
                uint8_t v = g_resend_bitmap[b];
                for (int bit = 0; bit < 8; bit++) {
                    if (v & (1u << bit)) {
                        asked++;
                    }
                }
            }
            Serial.printf("resend request: seq %u, %lu of %u fragments asked "
                          "for, bitmap %02X %02X %02X %02X\n",
                          (unsigned)g_resend_seq, (unsigned long)asked,
                          (unsigned)FRAG_COUNT,
                          g_resend_bitmap[0], g_resend_bitmap[1],
                          g_resend_bitmap[2], g_resend_bitmap[3]);
        }
        return;
    }

    if (data[2] != LINK_TYPE_REQUEST) {
        return;
    }
    uint16_t want = (uint16_t)data[10] | ((uint16_t)data[11] << 8);
    if (crc16(data, 10) != want) {
        g_frames_lost++;
        return;
    }
    if (data[3] != g_node_id && data[3] != 0xFF) {
        return;                       /* not for us */
    }
    g_request_pending = true;
    g_request_seq   = (uint16_t)data[4] | ((uint16_t)data[5] << 8);
    g_request_first = (uint16_t)data[6] | ((uint16_t)data[7] << 8);
    g_request_count = (uint16_t)data[8] | ((uint16_t)data[9] << 8);
    if (Serial) {
        Serial.printf("request: seq %u first %u count %u\n",
                      (unsigned)g_request_seq, (unsigned)g_request_first,
                  (unsigned)g_request_count);
    }
}

/* The channel the radio is actually on.  WiFi.channel() is no use here:
 * it reports the channel of the AP the station is associated with, and
 * this node never associates with one. */
static int link_channel(void) {
    uint8_t c = 0;
    wifi_second_chan_t second = WIFI_SECOND_CHAN_NONE;
    if (esp_wifi_get_channel(&c, &second) == ESP_OK) {
        return (int)c;
    }
    return -1;
}

static bool radio_start(void) {
    WiFi.mode(WIFI_STA);
    WiFi.setSleep(false);
    delay(100);

    /* The channel is set with esp_wifi_set_channel(), and the reason it
     * is not WiFi.channel(11) is worth writing down: the Arduino core
     * has no channel *setter*.  WiFi.channel(x) is the scan result's
     * channel — WiFiScanClass::channel(uint8_t i) — so the call
     * compiled, returned an int, and changed nothing at all.  The node
     * sat on channel 1 the whole time and the sends failed with "peer
     * channel is not equal to the home channel". */
    if (esp_wifi_set_channel(LINK_CHANNEL, WIFI_SECOND_CHAN_NONE) != ESP_OK) {
        if (Serial) {
            Serial.printf("the channel could not be set to %d\n",
                          (unsigned)LINK_CHANNEL);
        }
        return false;
    }
    delay(50);
```

```

    if (esp_now_init() != ESP_OK) {
        return false;
    }
    /* esp_now_init() starts the Wi-Fi driver, and the driver puts the
     * interface back on channel 1 when it starts — so the channel is
     * set again, after it. */
    esp_wifi_set_channel(LINK_CHANNEL, WIFI_SECOND_CHAN_NONE);
    delay(50);

    esp_now_register_recv_cb(on_recv);
    esp_now_register_send_cb(on_send);

    /* The transmitter's power, and why it is turned down.
     *
     * The INMP441 sits on the same board as the Wi-Fi antenna, a few
     * centimetres from it, and the radio's own transmission reaches the
     * microphone: with the radio at full power the node's windows
     * carried roughly five times the energy of the same room's windows
     * recorded with the radio off, and the extra energy had the shape
     * the model reads as a cry.  The node's own transmitter was putting
     * cry-shaped noise into the audio it was about to be judged on.
     *
     * 8.5 dBm is enough for a link of a couple of metres, which is what
     * a cot node is, and it is 11 dB less power into the room.  On a
     * battery node that is also the right thing to be doing. */
    esp_wifi_set_max_tx_power(WIFI_POWER_8_5dBm);

    esp_now_peer_info_t peer = {};
    memcpy(peer.peer_addr, g_bcast, 6);
    /* The channel is set explicitly rather than left at 0.  With 0 the
     * peer is supposed to follow the interface's current channel, but on
     * this driver the frame never left the node: the send callback fired
     * and nothing arrived at the hub, on any of the thirteen channels.
     * The hub's peer has always had the channel set, and the hub's
     * transmissions arrive perfectly. */
    peer.channel = LINK_CHANNEL;
    peer.ifidx   = WIFI_IF_STA;
    peer.encrypt = false;
    if (esp_now_add_peer(&peer) != ESP_OK) {
        return false;
    }
    return true;
}

/* Hand one frame to the radio.  The driver copies it into its queue and
 * returns at once; the frame goes on the air a moment later and its
 * acknowledgement arrives in the Wi-Fi task.
 *
 * Broadcast.  Unicast to the learned hub address wedged the send path:
 * the frame was accepted, never transmitted, and the send callback never
 * fired, so every send burned its two-second cap and the loop starved
 * the microphone.  Broadcast callbacks fire in milliseconds; revisit
 * unicast only with a passing dedicated test. */
static bool radio_queue(const uint8_t *data, size_t len)
{
    const uint8_t *dest = g_bcast;
    const int64_t st0 = esp_timer_get_time();
    const esp_err_t serr = esp_now_send(dest, data, len);
    const int64_t sdt = esp_timer_get_time() - st0;
    g_send_calls++;
    if (sdt > (int64_t)g_send_max_us) {
        g_send_max_us = (uint32_t)sdt;
    }
    if (serr != ESP_OK) {
        g_frames_lost++;
        return false;
    }
    g_frames_sent++;
    g_tx_queued++;
    return true;
}

/* Wait until the radio has room for another frame.
 *
 * The microphone is NOT read here, and that is the whole point.  An
 * earlier version of this code read it while the radio was busy, to
 * keep the node listening during a burst -- and the burst became
 * *slower* than it had ever been: a read of the microphone blocks until
 * the samples are there, so the burst was paced by the rate the
 * microphone produces audio (sixteen thousand samples a second, and a
 * read can only ever take what has arrived) rather than by the radio.
 * The burst measured 7.2 s that way, against 3.4 s before.
 *
 * The microphone has its own task now, so the audio is collected while
```

```
 * the radio is busy and this loop only has to wait for the radio. */
static void radio_wait_room(void)
{
    const int64_t start_us = esp_timer_get_time();
    while (tx_inflight() >= TX_INFLIGHT_MAX) {
        delay(1);
        if (esp_timer_get_time() - start_us > 5000000) {
            /* Far longer than the radio should ever take to take a frame;
             * give up rather than hang the node. */
            g_tx_full++;
            return;
        }
    }
}

/* Send one frame and wait for it to be acknowledged.  The telemetry uses
 * this: it is one frame every hundred milliseconds, and the wait is what
 * keeps the radio's queue tidy. */
static bool radio_send(const uint8_t *data, size_t len)
{
    radio_wait_room();
    const uint32_t before = g_tx_acks;
    if (!radio_queue(data, len)) {
        return false;
    }
    uint32_t waited = 0;
    const uint32_t t0 = micros();
    while (g_tx_acks == before && waited < 50) {
        delay(1);
        waited++;
    }
    const uint32_t waited_us = micros() - t0;
    g_tx_wait_us += waited_us;
    g_tx_wait_n++;
    if (waited_us > g_tx_wait_max) {
        g_tx_wait_max = waited_us;
    }
    return true;
}

/* Send one fragment of the upload target. */
static void send_fragment(uint16_t index, bool from_frozen, bool over_serial) {
    uint8_t payload[FRAG_PAYLOAD];
    uint16_t offset = (uint16_t)(index * FRAG_PAYLOAD);
    uint16_t len = (uint16_t)(BURST_BYTES - offset);
    if (len > FRAG_PAYLOAD) {
        len = FRAG_PAYLOAD;
    }
    for (uint16_t i = 0; i < len; i += 2) {
        int16_t s = burst_sample((uint16_t)(offset / 2 + i / 2), from_frozen);
        payload[i]     = (uint8_t)(s & 0xFF);
        payload[i + 1] = (uint8_t)((uint16_t)s >> 8);
    }
    size_t total = build_frame(2, g_burst_seq, index, FRAG_COUNT, payload, len);
    if (over_serial) {
        Serial.write(g_frame, total);
    } else {
        /* The radio takes a few frames at a time; the wait for room is
         * where the microphone is read, so the node keeps listening while
         * it talks.  A frame the driver refuses is tried again: the
         * queue was full, and the fragment is one the hub is waiting
         * for. */
        for (int tries = 0; tries < 3; tries++) {
            radio_wait_room();
            if (radio_queue(g_frame, total)) {
                break;
            }
            delay(2);
        }
    }
}

/* ------------------------------------------------------------ telemetry -- */

static void send_telemetry(void) {
    int32_t rms_milli = (int32_t)(g_rms * 1000.0f + (g_rms >= 0 ? 0.5f : -0.5f));
    uint8_t payload[32];
    payload[0] = (uint8_t)(rms_milli & 0xFF);
    payload[1] = (uint8_t)((rms_milli >> 8) & 0xFF);
    payload[2] = (uint8_t)((rms_milli >> 16) & 0xFF);
    payload[3] = (uint8_t)((rms_milli >> 24) & 0xFF);
    uint16_t win_ms = (uint16_t)(WINDOW_SAMPLES * 1000 / SAMPLE_RATE);
    payload[4] = (uint8_t)(win_ms & 0xFF);
    payload[5] = (uint8_t)(win_ms >> 8);
    uint8_t flags = 0;
```

```
    if (g_frozen_valid) {
        flags |= 0x01;              /* a burst is ready */
    }
    if (g_rms >= g_level) {
        flags |= 0x02;              /* the event level is being crossed */
    }
    if (g_filled == WINDOW_SAMPLES) {
        flags |= 0x04;              /* the live window is complete */
    }
    payload[6] = flags;
    /* The event level the node is using, in milli-RMS.  The node owns the
     * noise floor, so it owns the level that follows from it, and the hub
     * triggers on this number: the node is the one that freezes the
     * window, so the level that matters is the one it used.  Zero means
     * "not reported" and the hub falls back to its own fixed level. */
    uint16_t level_milli = (uint16_t)(g_level * 1000.0f + 0.5f);
    payload[7] = (uint8_t)(level_milli & 0xFF);
    payload[8] = (uint8_t)((level_milli >> 8) & 0xFF);
    /* The frozen window's own level, in milli-RMS: what the model will
     * actually be shown, as opposed to the live reading beside it. */
    uint16_t frozen_milli = (uint16_t)(g_frozen_rms * 1000.0f + 0.5f);
    payload[9] = (uint8_t)(frozen_milli & 0xFF);
    payload[10] = (uint8_t)((frozen_milli >> 8) & 0xFF);
    /* The noise floor itself, in milli-RMS, clamped to what one byte
     * holds. */
    uint16_t floor_milli = (uint16_t)(g_floor * 1000.0f + 0.5f);
    payload[11] = (uint8_t)(floor_milli > 255u ? 255u : floor_milli);
    /* The CRC-32 of the frozen window, so the hub can check the window it
     * reassembles against the one the node actually froze. */
    uint32_t crc = g_frozen_valid ? g_frozen_crc : 0u;
    payload[12] = (uint8_t)(crc & 0xFF);
    payload[13] = (uint8_t)((crc >> 8) & 0xFF);
    payload[14] = (uint8_t)((crc >> 16) & 0xFF);
    payload[15] = (uint8_t)((crc >> 24) & 0xFF);
    /* The radio's own counters.  They are the answer to the question a
     * stalled burst raises -- "did the node send them, or did the air
     * eat them?" -- and they cost the telemetry sixteen bytes.
     *
     * frames_sent is what the node handed to the radio, frames_acked is
     * what the radio says it put on the air, and frames_lost is what
     * the radio refused outright.  A node whose frames_sent climbs to
     * 286 while frames_acked stays at 30 has lost the rest in the air;
     * a node whose frames_lost climbs has a radio queue too small for
     * the burst.  The hub prints all four whenever a burst stalls, so
     * the difference is on the log instead of in a guess. */
    const uint32_t counters[4] = {
        (uint32_t)g_frames_sent,
        (uint32_t)g_frames_lost,
        (uint32_t)g_tx_acks,
        (uint32_t)g_tx_full,
    };
    /* The counters start at byte 16 now.  They used to start at byte
     * 18, behind the battery voltage that used to sit between them and
     * the CRC (D-83); the battery tier is gone and the counters moved
     * down to fill the gap. */
    for (int i = 0; i < 4; i++) {
        payload[16 + i * 4]     = (uint8_t)(counters[i] & 0xFF);
        payload[16 + i * 4 + 1] = (uint8_t)((counters[i] >> 8) & 0xFF);
        payload[16 + i * 4 + 2] = (uint8_t)((counters[i] >> 16) & 0xFF);
        payload[16 + i * 4 + 3] = (uint8_t)((counters[i] >> 24) & 0xFF);
    }

    size_t total = build_frame(1, g_burst_seq, 0, 0, payload, sizeof(payload));
    radio_send(g_frame, total);
    g_telemetry_seq++;
}

/* ------------------------------------------------------------------ console -- */

/* How many bars a link reading is worth, on the same scale the hub's
 * display uses: 2.4 GHz links in a house, strong above -60 dBm and
 * barely there below -90. */
static int rssi_bars(int8_t rssi) {
    if (rssi == 0) { return 0; }
    if (rssi >= -60) { return 4; }
    if (rssi >= -70) { return 3; }
    if (rssi >= -80) { return 2; }
    if (rssi >= -90) { return 1; }
    return 0;
}

/* The node's own view of the link to the hub, as a network bar. */
static void print_link_bar(void) {
    const int bars = rssi_bars(g_hub_rssi);
    Serial.print("link   [");
```

```
    for (int i = 0; i < 4; i++) {
        Serial.print(i < bars ? '#' : '.');
    }
    if (g_hub_rssi != 0) {
        Serial.printf("] %d dBm", (int)g_hub_rssi);
    } else {
        Serial.print("] --");
    }
    if (g_hub_rssi != 0) {
        const uint32_t age = millis() - g_hub_rssi_ms;
        Serial.printf("  (hub frame %lu ms ago)\n", (unsigned long)age);
    } else {
        Serial.println("  (no hub frame heard)");
    }
}

static void print_status(void) {
    /* The radio's own address, which the hub needs in its table of node
     * addresses.  WiFi.macAddress() reports a different one on this
     * board, so it is read from the chip directly. */
    uint8_t sta[6] = {0};
    esp_read_mac(sta, ESP_MAC_WIFI_STA);
    Serial.printf("node %u  mac %02X:%02X:%02X:%02X:%02X:%02X\n", g_node_id,
                  sta[0], sta[1], sta[2], sta[3], sta[4], sta[5]);
    Serial.printf("esp_now_send calls=%u max_us=%u\n",
                  (unsigned)g_send_calls, (unsigned)g_send_max_us);
    Serial.printf("time pump=%llu tick=%llu req=%llu rest=%llu ms\n",
                  (unsigned long long)(g_t_pump / 1000),
                  (unsigned long long)(g_t_tick / 1000),
                  (unsigned long long)(g_t_req / 1000),
                  (unsigned long long)(g_t_rest / 1000));
    Serial.printf("pump calls=%u empty=%u frames=%u max_us=%u loops=%u loop_max_us=%u\n",
                  (unsigned)g_pump_calls, (unsigned)g_pump_empty,
                  (unsigned)g_pump_frames, (unsigned)g_pump_max_us,
                  (unsigned)g_loop_iters, (unsigned)g_loop_max_us);
    Serial.printf("I2S zero watchdog restarts=%u\n", (unsigned)g_i2s_restarts);
    Serial.printf("window %u/%u samples   frozen=%u seq=%u\n", (unsigned)g_filled,
                  (unsigned)WINDOW_SAMPLES, g_frozen_valid ? 1 : 0,
                  (unsigned)g_burst_seq);
    if (g_frozen_valid) {
        Serial.printf("frozen crc32 0x%08X\n", (unsigned)g_frozen_crc);
    }
    Serial.printf("rms %.4f  mean %d  floor %.4f  level %.4f  fragments=%u\n",
                  g_rms, (int)g_mean, g_floor, g_level, (unsigned)FRAG_COUNT);
    Serial.printf("heap %u bytes\n", (unsigned)ESP.getFreeHeap());
    Serial.printf("channel %d\n", link_channel());
    uint32_t ver = 0;
    esp_now_get_version(&ver);
    Serial.printf("ESP-NOW version %u, hub %s\n", (unsigned)ver,
                  g_hub_known ? "known" : "unknown");
    print_link_bar();
}

static void print_stats(void) {
    Serial.printf("telemetry seq %u  frames sent %u  lost %u  requests %u  freezes %u\n",
                  (unsigned)g_telemetry_seq, (unsigned)g_frames_sent,
                  (unsigned)g_frames_lost, (unsigned)g_requests,
                  (unsigned)g_freezes);
    Serial.printf("tx wait: avg %u us, max %u us, over %u frames\n",
                  (unsigned)(g_tx_wait_n ? g_tx_wait_us / g_tx_wait_n : 0),
                  (unsigned)g_tx_wait_max, (unsigned)g_tx_wait_n);
    Serial.printf("radio: queued %u  acked %u  in flight %u  queue full %u times\n",
                  (unsigned)g_tx_queued, (unsigned)g_tx_acks,
                  (unsigned)tx_inflight(), (unsigned)g_tx_full);
    Serial.printf("radio: the driver could not carry %u frames\n",
                  (unsigned)g_tx_failed);
}

static void handle_command(char *line) {
    if (strcmp(line, "status") == 0) {
        print_status();
    } else if (strcmp(line, "stats") == 0) {
        print_stats();
    } else if (strcmp(line, "freeze") == 0) {
        if (g_filled < WINDOW_SAMPLES) {
            Serial.println("window not full yet");
        } else {
            freeze_window();
            Serial.printf("frozen seq=%u\n", g_burst_seq);
        }
    } else if (strcmp(line, "burst") == 0) {
        if (!g_frozen_valid) {
            Serial.println("no frozen window");
        } else {
            Serial.printf("BURST seq=%u fragments=%u bytes=%u\n", g_burst_seq,
```

```
                           FRAG_COUNT, (unsigned)BURST_BYTES);
            g_dump_active = true;
            for (uint16_t i = 0; i < FRAG_COUNT; i++) {
                send_fragment(i, true, true);
            }
            g_dump_active = false;
            Serial.println("BURST-END");
        }
    } else if (strncmp(line, "id ", 3) == 0) {
        int v = atoi(line + 3);
        /* This hub watches two cots, so a node's number is one of two.
         * A third number is a node the hub will not answer to. */
        if (v >= 1 && v <= CRYNET_COT_COUNT) {
            g_node_id = (uint8_t)v;
            g_prefs.putUChar("node_id", g_node_id);
            Serial.printf("node id %u (saved)\n", g_node_id);
        } else {
            Serial.printf("id must be 1..%u (this hub watches two cots)\n",
                          (unsigned)CRYNET_COT_COUNT);
        }
    } else if (strcmp(line, "help") == 0) {
        Serial.println("status | stats | freeze | burst | id <n> | help");
    } else {
        Serial.println("unknown command");
    }
}

/* ---------------------------------------------------------------- loop -- */

void setup() {
    Serial.begin(115200);
    /* A closed serial port must never cost the loop time: the HWCDC
     * write path waits up to two seconds per print when the host is not
     * reading, and an unattended node printing its rms every hundred
     * milliseconds starved its own loop into a crawl.  Zero timeout
     * makes prints drop instead of block. */
    Serial.setTxTimeoutMs(0);
    delay(200);

    uint8_t mac[6];
    WiFi.macAddress(mac);
    const uint8_t derived = (uint8_t)((mac[5] & 0x0F) + 1);

    /* A node id set over serial survives reboots in NVS; a board that was
     * never named keeps the MAC-derived default.  Identical Pro Minis can
     * derive the same number — the project really did end up with two
     * boards both calling themselves node 1. */
    g_prefs.begin("crynet", false);
    const uint8_t saved = g_prefs.getUChar("node_id", 0);
    g_node_id = saved ? saved : derived;

    Serial.println("\nCryNet node firmware — Stage 2b");
    if (saved) {
        Serial.printf("node id %u (saved; the MAC would have said %u)\n",
                      g_node_id, (unsigned)derived);
    } else {
        Serial.printf("node id %u (derived from the MAC; change it with \"id <n>\")\n",
                      g_node_id);
    }
    /* This hub watches two cots and ignores every other node number, so
     * a node that has been left on a MAC-derived number outside 1..2
     * would transmit into a hub that never answers it.  Say so plainly
     * rather than leaving the board looking dead. */
    if (g_node_id < 1 || g_node_id > CRYNET_COT_COUNT) {
        Serial.printf("WARNING: this hub watches cots 1 and %u only. "
                      "Set this board with \"id 1\" or \"id %u\".\n",
                      (unsigned)CRYNET_COT_COUNT, (unsigned)CRYNET_COT_COUNT);
    }

    if (!i2s_start()) {
        Serial.println("FATAL: I2S init failed");
        while (true) {
            delay(1000);
        }
    }
    Serial.println("I2S ready: 16 kHz, mono, 32-bit slots -> int16");
    /* The microphone gets a task of its own, so that the main loop never
     * waits for audio and the radio is fed as fast as the radio can
     * carry it.  The task owns the I2S peripheral from here on: nothing
     * else may read it. */
    if (xTaskCreate(capture_task, "capture", 4096, NULL, 3, NULL) != pdPASS) {
        Serial.println("FATAL: the capture task could not be started");
        while (true) {
            delay(1000);
        }
```

```
    }

    if (!radio_start()) {
        Serial.println("FATAL: ESP-NOW init failed");
        while (true) {
            delay(1000);
        }
    }
    uint32_t now_version = 0;
    if (esp_now_get_version(&now_version) == ESP_OK) {
        Serial.printf("ESP-NOW protocol version %u\n", (unsigned)now_version);
    }
    Serial.printf("ESP-NOW ready on channel %d\n", link_channel());

    Serial.println("commands: status | stats | freeze | burst | id <n> | help");
}

void loop() {
    const int64_t lt0 = esp_timer_get_time();
    g_loop_iters++;
    if (g_hub_peer_pending) {
        g_hub_peer_pending = false;
        esp_now_peer_info_t hub = {};
        memcpy(g_hub_mac, g_pending_mac, 6);
        memcpy(hub.peer_addr, g_hub_mac, 6);
        hub.channel = LINK_CHANNEL;
        hub.ifidx = WIFI_IF_STA;
        hub.encrypt = false;
        if (esp_now_add_peer(&hub) == ESP_OK) {
            g_hub_known = true;
            if (Serial)
                Serial.printf("hub is %02X:%02X:%02X:%02X:%02X:%02X\n",
                          g_hub_mac[0], g_hub_mac[1], g_hub_mac[2],
                          g_hub_mac[3], g_hub_mac[4], g_hub_mac[5]);
        }
    }
    /* The microphone is read by its own task; the main loop does not
     * wait for audio.  All it has to do here is notice when the audio
     * came through with a hole in it. */
    if (g_window_damaged) {
        g_filled = 0;
        g_window_damaged = false;
    }
    /* The window is only worth measuring once it holds a full 100 ms: the
     * hub's front end removes the mean of the whole two seconds, and a
     * partial window would not match it. */
    const int64_t tk0 = esp_timer_get_time();
    if (g_filled >= RMS_SAMPLES) {
        uint32_t now = millis();
        if (now - g_last_tick >= TELEMETRY_MS) {
            g_last_tick = now;
            g_mean = window_mean(g_filled);
            uint32_t n = (g_filled < RMS_SAMPLES) ? g_filled : RMS_SAMPLES;
            g_rms = rms_of_last(g_mean, n);

            /* The noise floor, and the event level that follows from it.
             * The floor falls at once when the room gets quieter and rises
             * slowly, so a cry that lasts a minute cannot raise the floor
             * to the cry's own level and turn the trigger off half way
             * through the episode.  The first reading sets the floor. */
            if (g_floor < 0.0f) {
                g_floor = g_rms;
            } else if (g_rms < g_floor) {
                g_floor = g_rms;
            } else {
                g_floor += (g_rms - g_floor) * FLOOR_RISE;
            }
            g_level = TRIGGER_MULT * g_floor;
            if (g_level < TRIGGER_MIN) {
                g_level = TRIGGER_MIN;
            } else if (g_level > TRIGGER_MAX) {
                g_level = TRIGGER_MAX;
            }

            /* Two readings in a row at or above the level: the same rule
             * the hub uses, and the same one the study assumed.  One lone
             * reading is the microphone's slow drift, not an event. */
            if (g_rms >= g_level) {
                if (g_loud_streak < 255u) {
                    g_loud_streak++;
                }
            } else {
                g_loud_streak = 0;
            }

```

```
            /* Freeze first, announce second.  The telemetry has to carry
             * the sequence number and the CRC-32 of the window the hub is
             * about to ask for; if the freeze happened after the frame
             * was sent, the hub had nothing to check the window it
             * reassembled against. */
            /* The hold on the uploaded window ends here.  Until then
             * the node keeps that snapshot -- it is the one the hub is
             * missing fragments of -- and does not freeze a new one.
             *
             * The hold does not end while the hub is still fetching
             * that window.  It used to, and in a noisy room the three
             * seconds it lasts were up long before the hub's resend
             * rounds had finished, so the node froze a new window in
             * the middle of a transfer and the pieces the hub was still
             * missing went with the old one.
             *
             * The test was also backwards: `g_frozen_hold_ms` is a
             * deadline (millis() + FROZEN_HOLD_MS), so `now - deadline`
             * is negative -- and as an unsigned integer that is a
             * number well past 0x80000000 -- for every moment of the
             * hold, and small only once the deadline has passed.  The
             * test as written therefore released the window on the
             * very next reading and never released it afterwards, the
             * opposite of the intent.  Measured on the bench, the
             * minute of 2026-11-04 19:26: node 3 was refused four
             * resends with "no frozen window for it" because it had
             * already frozen the next one, and none of its bursts ever
             * finished.  A signed difference is the deadline test.
             *
             * When the hold does end, the window is dropped outright.
             * It has served its purpose: the hub has had its three
             * seconds of resend rounds and the fragments are no longer
             * needed.  Leaving it valid would leave it advertised,
             * and the freeze rule below refuses to take a new window
             * while one is advertised -- so the node would never
             * freeze again. */
            const bool hub_is_fetching =
                (uint32_t)(now - g_transfer_ms) < TRANSFER_HOLD_MS;
            if (g_frozen_served && !hub_is_fetching &&
                (int32_t)(now - g_frozen_hold_ms) >= 0) {
                g_frozen_served = false;
                g_frozen_valid = false;
                g_peak_pending = false;
            }
            /* Cry mode: once a loud window is frozen, do not freeze
             * another until the room goes quiet.  A cry that lasts
             * thirty seconds would otherwise freeze six windows and the
             * hub would process them one by one, 30 s behind.  (D-69.)
             * The exit threshold is absolute (0.01 rms): the adaptive
             * level can be as low as 0.005, and a quiet room at 0.01
             * would never fall below it.  (D-70.) */
            if (g_cry_mode && g_rms < 0.01f && g_loud_streak == 0u) {
                g_cry_mode = false;
            }
            /* Preempt the hold for a loud event.  A cry is typically an
             * order of magnitude louder than the room, so a new event
             * that is much louder than the window already frozen is
             * worth an immediate freeze, even if the previous window is
             * still being served: the hub abandons the stale burst and
             * asks for this one instead.  (D-68.)
             *
             * The comparison is against the level of the window that
             * was frozen, NOT against the trigger level.  It used to be
             * the trigger level, and that made the node freeze a new
             * window every half second while a cry was going on -- the
             * cry is far above the trigger level, so the hold was
             * cancelled every time -- and the hub, still waiting for
             * the window before it, threw that burst away and asked
             * for the newest one instead.  The hub chased windows it
             * could never finish assembling, one after another.  The
             * rule now asks the question that was meant: is this a new
             * event, or the same one going on?
             *
             * And it does not take a window the hub is still fetching.
             * The rule used to be able to cancel the hold at any moment,
             * which in a noisy room was every second or two -- and each
             * time it fired, the fragments the hub was still missing were
             * thrown away with the old window.  The hub could never
             * finish a burst at such a cot: it asked for the pieces that
             * were missing and the node had no copy of them left.  A
             * window the hub is still asking about is now left alone
             * until TRANSFER_HOLD_MS has passed with no word from it.
             *
             * And it now covers a window the hub has not asked about
             * yet as well.  A frozen window that is still advertised
```

```
             * -- `g_frozen_valid`, whether or not it has been served --
             * is the only copy of the cry that cot heard, and the room
             * itself must not be allowed to take its place: measured
             * on the bench, the minute of 2026-11-04 19:26, node 1
             * froze a window of 0.1769 rms -- a genuine cry -- and then
             * replaced it with a 0.0104 rms window of ordinary room
             * sound three seconds later, because the hub had spent its
             * airtime on another cot's window and had not come back
             * for it.  Only a much louder event (the same three-times
             * rule as before) or the hold above may take the place of
             * an advertised window now. */
            if (g_frozen_valid && g_frozen_rms > 0.0f && !hub_is_fetching &&
                g_rms >= 3.0f * g_frozen_rms && g_loud_streak >= 2u) {
                g_frozen_served = false;
                g_frozen_valid = false;
            }
            /* In cry mode, only a louder event (preempt) may freeze.
             * And only an unadvertised window may be frozen at all:
             * while the node holds a window the hub has not taken, the
             * room must wait.  Nothing here is lost by that -- the
             * live ring keeps filling, and a cry that arrives while a
             * quiet window is held takes its place through the rule
             * above, three times as loud as it is. */
            bool allow_freeze = !g_cry_mode ||
                (g_frozen_rms > 0.0f && g_rms >= 3.0f * g_frozen_rms);
            if (allow_freeze && !g_frozen_valid && g_filled == WINDOW_SAMPLES) {
                if (!g_peak_pending && g_loud_streak >= 2u) {
                    /* A loud moment: remember it, and freeze a second
                     * later, when the ring holds the second that follows
                     * it as well.  The two seconds the hub classifies are
                     * then centred on the cry instead of ending at it. */
                    g_peak_pending = true;
                    g_peak_ms = now;
                } else if (g_peak_pending) {
                    const uint32_t age = now - g_peak_ms;
                    /* The cry stopped: the streak broke.  The window is
                     * frozen then, so that it holds the cry that just
                     * ended instead of the room behind it.  The two
                     * seconds after the peak are the cap for a cry
                     * that goes on, not a delay every cry has to sit
                     * through.  (research/simulate_freeze.py: the
                     * stop-capped rule catches as many of the corpus's
                     * cry episodes as the old peak-centred rule, and
                     * the hub announced no room window under any of
                     * the three.) */
                    const bool cry_stopped = (g_loud_streak < 2u);
                    if (age >= PEAK_DELAY_MS ||
                        (cry_stopped && age >= PEAK_STOP_DELAY_MS)) {
                        if (age < PEAK_EXPIRE_MS) {
                            freeze_window();
                            /* A loud window puts the node into cry mode:
                             * do not freeze again until the room goes
                             * quiet, so a sustained cry does not queue
                             * six windows ahead of itself.  (D-69.)
                             * The threshold is absolute (0.05 rms), not
                             * relative to the floor: a quiet room has a
                             * floor near 0.001, and 5x that is 0.005,
                             * which quiet windows exceed.  (D-70.) */
                            if (g_frozen_rms >= 0.05f) {
                                g_cry_mode = true;
                            }
                            if (!g_dump_active && Serial) {
                                Serial.printf("froze window, burst seq %u  "
                                              "level %.4f  win rms %.4f%s\n",
                                              g_burst_seq, g_level,
                                              g_frozen_rms,
                                              g_cry_mode ? "  (cry mode)" : "");
                            }
                        }
                        g_peak_pending = false;
                    }
                }
            }

            /* A frozen window that nobody asked for must not block the
             * next one.  The microphone's DC drift can cross the event
             * level for a single reading, and the hub's trigger needs two
             * of them, so that window would sit here unused and the next
             * real cry would be served the stale, quiet one.
             *
             * The age is measured from a fresh clock reading, not from
             * the `now` above: that one was taken before the freeze
             * above happened, so `now - g_frozen_ms` would underflow to
             * four billion and expire the window that was frozen a
             * moment earlier — which is exactly what happened, and the
```

```
             * node never once advertised a burst as ready. */
            if (g_frozen_valid &&
                (millis() - g_frozen_ms) > FROZEN_EXPIRE_MS) {
                g_frozen_valid = false;
                g_frozen_served = false;
                g_peak_pending = false;
            }

            send_telemetry();

            if (!g_dump_active && Serial) {
                /* Serial here is USB-CDC: with no host reading the
                 * port, a write blocks — and this print runs every
                 * hundred milliseconds, so an unattended node starved
                 * its own loop and stopped pumping the microphone. */
                Serial.printf("rms %.4f  floor %.4f  level %.4f  %s\n",
                              g_rms, g_floor, g_level,
                              g_loud_streak >= 2u ? "EVENT" : "");
            }
        }
    }
    g_t_tick += (uint64_t)(esp_timer_get_time() - tk0);

    /* A request from the hub is served from the frozen snapshot when it
     * matches, and from the live window otherwise. */
    const int64_t rq0 = esp_timer_get_time();
    if (g_request_pending) {
        g_request_pending = false;
        g_requests++;
        bool from_frozen = (g_request_seq == g_burst_seq) && g_frozen_valid;
        if (!from_frozen && g_filled < WINDOW_SAMPLES) {
            /* The live window is still refilling after the last upload,
             * so its chronological order is not known.  Wait. */
            from_frozen = false;
            if (!g_dump_active && Serial) {
                Serial.println("request ignored: the live window is not complete");
            }
        } else {
            uint16_t first = g_request_first;
            uint16_t count = g_request_count;
            if (count == 0 || count > FRAG_COUNT) {
                count = FRAG_COUNT;
            }
            if (first >= FRAG_COUNT) {
                first = 0;
            }
            uint16_t sent = 0;
            const int64_t burst_t0 = esp_timer_get_time();
            if (from_frozen) {
                /* The hub is fetching this window.  While it does, the
                 * window is not taken away by a louder moment: the
                 * fragments that did not arrive are only in this copy. */
                g_transfer_ms = millis();
            }
            for (uint16_t i = 0; i < count; i++) {
                uint16_t index = (uint16_t)(first + i);
                if (index >= FRAG_COUNT) {
                    break;
                }
                send_fragment(index, from_frozen, false);
                sent++;
                /* The window is being fetched.  The clock is refreshed
                 * for every fragment, so a burst that takes ten seconds
                 * to cross a bad link keeps its window for the whole
                 * of it. */
                if (from_frozen) {
                    g_transfer_ms = millis();
                }
                /* No fixed gap between the fragments.  The radio takes
                 * them a few at a time and acknowledges them in groups,
                 * and the wait for room is where the microphone is read,
                 * so the burst crosses the air as fast as the radio can
                 * carry it.
                 *
                 * The gap was ten milliseconds for a long time, and it
                 * was the whole of the burst's length: 286 fragments at
                 * thirteen milliseconds each is the 3.8 s the hub
                 * measures.  Then it was four, which still left the
                 * burst at 3.4 s, because the cost was in waiting for
                 * each frame's own acknowledgement rather than in the
                 * gap.  The radio now carries a group of frames at a
                 * time and the burst is a fraction of its old length. */
            }
            if (!g_dump_active && Serial) {
                const uint64_t burst_ms =
```

```
                    (uint64_t)(esp_timer_get_time() - burst_t0) / 1000;
                Serial.printf("served %u fragments of %u in %llu ms\n",
                              (unsigned)sent, (unsigned)count,
                              (unsigned long long)burst_ms);
            }
            /* The live window is only thrown away if the microphone
             * actually lost samples while the burst was on the air.  It
             * used to be dropped every time, because nothing read the
             * microphone during the burst and the audio it held had
             * holes in it -- which left the node deaf for the two
             * seconds the window took to refill, and the next cry at
             * this cot could not be seen until then. */
            if (count > 1) {
                if (g_window_damaged) {
                    g_filled = 0;
                    g_window_damaged = false;
                }
                /* The frozen snapshot stays.  The hub may have lost a
                 * few fragments of this very window in the air, and it
                 * can only get them back from a node that still has
                 * them.  g_frozen_served says "this one has been
                 * uploaded", which is what lets the node freeze the
                 * next one straight away. */
                g_frozen_served = true;
                g_frozen_hold_ms = millis() + FROZEN_HOLD_MS;
                g_peak_pending = false;
            }
        }
    }
    g_t_req += (uint64_t)(esp_timer_get_time() - rq0);

    /* A resend request is served from the frozen window, and only if the
     * frozen window is still the one the hub asked about.  If the node
     * has already frozen a newer window the fragments the hub wants are
     * gone, and sending newer ones under the old sequence number would
     * corrupt the burst rather than complete it -- so the request is
     * refused and the hub asks for the newer window instead. */
    if (g_resend_pending) {
        g_resend_pending = false;
        if (!g_frozen_valid || g_resend_seq != g_burst_seq) {
            if (Serial) {
                Serial.println("resend refused: no frozen window for it");
            }
        } else {
            uint16_t resent = 0;
            /* The hub is still fetching this window. */
            g_transfer_ms = millis();
            for (uint16_t i = 0; i < FRAG_COUNT; i++) {
                if ((g_resend_bitmap[i >> 3] & (1u << (i & 7u))) == 0) {
                    continue;
                }
                send_fragment(i, true, false);
                resent++;
                g_transfer_ms = millis();
            }
            if (Serial) {
                Serial.printf("resent %u fragments of seq %u\n",
                              (unsigned)resent, (unsigned)g_burst_seq);
            }
        }
    }

    /* The console. */
    static char line[64];
    static uint8_t len = 0;
    while (Serial.available() > 0) {
        char c = (char)Serial.read();
        if (c == '\n' || c == '\r') {
            line[len] = 0;
            if (len > 0) {
                handle_command(line);
            }
            len = 0;
        } else if (len < sizeof(line) - 1) {
            line[len++] = c;
        }
    }
    const int64_t ldt = esp_timer_get_time() - lt0;
    g_t_rest += (uint64_t)ldt;
    if (ldt > (int64_t)g_loop_max_us) {
        g_loop_max_us = (uint32_t)ldt;
    }
}
```

The wearable, and the motor's bench sketch

firmware/wearable/wearable_crynet/wearable_crynet.ino

750 lines · 31,153 bytes

```
/*
 * CryNet caregiver wearable — ESP32-C3 SuperMini + SSD1306 OLED.
 *
 * Stage 2d's first artifact: the BLE client side of the alert contract
 * (firmware/hub_infer/components/ble_alert — the same service a phone
 * connects to, verified in the alert sessions).  The wearable scans for
 * the hub, subscribes to the six-byte alert characteristic and shows the
 * alert on the OLED.  The eight-byte state characteristic is subscribed
 * to as well: the hub publishes it whenever it changes and at least once
 * a second, so a wearable that attaches in the middle of a cry still
 * sees it, and the idle face's crying mask stays honest for as long as
 * the link is up.  The alert face reads that mask too: the alert names
 * one cot, because the hub names the loudest and one cry must be one
 * buzz, but the screen lists every cot the hub says is crying.
 *
 * All BLE work — scanning, connecting, subscribing — runs on its own
 * FreeRTOS task, and loop() only renders: the library's scan start,
 * connect and characteristic reads all block their caller, so leaving
 * them in loop() froze the whole screen whenever the hub went away.
 *
 * The screen is set in the hub panel's house style (firmware/hub_infer/
 * components/hub_display), translated to one bit: a boot face with the
 * breathing wordmark and a filling progress bar, a title card carrying
 * the device name, the builder's name and the id number, then the
 * dashboard with the waveform along its bottom edge.  An alert inverts a
 * band across the top -- the closest a monochrome panel comes to the
 * hub's red -- and blinks it while the wrist buzzes.
 *
 * The wearable does not carry the full project title: it is ninety-seven
 * characters and this screen is eighty pixels wide, so it would crowd
 * out the name and the id number that matter more here.  The title
 * belongs on the hub's panel, which has room for it.
 *
 * The hub's contract, unchanged by this sketch:
 *
 *   device name   INFANT_MONITOR_HUB
 *   service       19b10000-e8f2-537e-4f6c-d104768a1214
 *   alert (notify) 19b10001-...  6 bytes:
 *       0  cot id
 *       1  alert type   1 = a cry, 2 = the cry has stopped
 *       2  confidence, percent, little-endian u16
 *       4  event id, little-endian u16
 *   state (read+notify) 19b10002-...  8 bytes:
 *       0  cot count   1  crying mask (bit 0 = cot 1, ...)
 *       2  reserved    3  confidence, percent
 *       4  started_at, seconds, little-endian u32
 *
 * The hub advertises only when no caregiver device is attached and no
 * ESP-NOW burst is in the air, so a disconnected wearable simply keeps
 * scanning and attaches in the gaps — that is the design, not a bug.
 *
 * Wiring — SSD1306 to ESP32-C3 SuperMini (SYSTEM_WIRING.md 5.3):
 *     VCC -> 3V3    GND -> GND    SDA -> GPIO5    SCL -> GPIO6
 *
 * The vibration motor hangs on a BC547 low-side switch that this
 * firmware drives: GPIO7 -> 2.2 k base resistor -> base, emitter to
 * ground, the motor in the collector with its own flyback diode
 * (page 5 of the wiring diagrams).  The 100 k base-emitter pulldown
 * keeps the transistor off while the C3 boots with the pin floating, so
 * GPIO7 is driven low before anything else happens here.
 *
 * Serial console — 115200 baud: alerts, links and errors print here.
 */

#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>
#include <BLEDevice.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

/* ------------------------------------------------------------------ pins -- */

static const uint8_t PIN_SDA = 5;   /* SSD1306 SDA (SYSTEM_WIRING.md 5.3) */
static const uint8_t PIN_SCL = 6;   /* SSD1306 SCL                        */
```

```
static const uint8_t PIN_MOTOR = 7; /* the BC547 haptic switch — wiring
                                       diagram page 5: GPIO7 -> 2.2 k ->
                                       base, motor in the collector     */

/* ------------------------------------------------------------ contract -- */

static const char *HUB_NAME     = "INFANT_MONITOR_HUB";
static const char *SERVICE_UUID = "19b10000-e8f2-537e-4f6c-d104768a1214";
static const char *ALERT_UUID   = "19b10001-e8f2-537e-4f6c-d104768a1214";
static const char *STATE_UUID   = "19b10002-e8f2-537e-4f6c-d104768a1214";

/* ------------------------------------------------------------------ state -- */

static Adafruit_SSD1306 display(128, 64, &Wire, -1);

static BLEAdvertisedDevice   *g_hub_device = nullptr;
static BLEClient             *g_client     = nullptr;
static BLERemoteCharacteristic *g_alert    = nullptr;
static BLERemoteCharacteristic *g_state    = nullptr;

static volatile bool g_do_connect = false;
static volatile bool g_connected  = false;

/* The last alert received.  Written from the BLE callback, read by the
 * renderer — set the dirty flag after each write. */
static volatile uint8_t  g_cot      = 0;
static volatile uint8_t  g_type     = 0;    /* 1 = cry, 2 = cancel */
static volatile uint16_t g_conf_pct = 0;
static volatile uint16_t g_event    = 0;
static volatile bool     g_have_alert = false;
static volatile uint32_t g_alert_ms   = 0;

static volatile uint8_t  g_cots = 0;
static volatile uint8_t  g_mask = 0;

/* The haptic drive.
 *
 * GPIO7 high turns the BC547 on and the motor spins; GPIO7 low lets it
 * stop.  A buzz is a 250 ms pulse, and it runs in step with the band
 * blinking across the top of the screen — the same rhythm, so the wrist
 * and the eye agree on what is happening.
 *
 * One buzz per cry: the hub re-announces a cry when a louder cot joins
 * it, and the caregiver must not be buzzed twice for one cry. */
static const uint32_t HAPTIC_MS       = 2500;  /* how long a buzz runs    */
static const uint32_t HAPTIC_PULSE_MS = 250;   /* one pulse of the motor */
static volatile uint32_t g_haptic_until = 0;   /* when the buzz stops    */
static volatile uint32_t g_haptic_start = 0;   /* when the buzz started  */

static bool g_dirty = true;      /* the screen needs a redraw */
static bool g_oled_ok = false;

/* ---------------------------------------------------------- haptics -- */

static void haptic_trigger(void)
{
    g_haptic_start = millis();
    g_haptic_until = g_haptic_start + HAPTIC_MS;
    Serial.printf("[haptic] alert %u — motor on GPIO7 for %u ms\n",
                  (unsigned)g_event, (unsigned)HAPTIC_MS);
}

static void haptic_tick(void)
{
    const uint32_t now = millis();
    bool on = false;
    if (now < g_haptic_until) {
        /* 250 ms on, 250 ms off. */
        on = (((now - g_haptic_start) / HAPTIC_PULSE_MS) % 2) == 0;
    }
    digitalWrite(PIN_MOTOR, on ? HIGH : LOW);
}

/* ------------------------------------------------------------ theme -- */

/* The hub's panel sets the house style, and this little screen borrows
 * it in one bit: the waveform that runs along the bottom edge, the
 * wordmark, the accent band across the top of the title card, and the
 * boot progress bar that fills as the device comes up.
 *
 * Colour is the one thing a monochrome panel cannot copy, so the accent
 * is drawn inverted, and an alert turns the whole screen upside down
 * for as long as the wrist is buzzing -- the loudest thing one bit can
 * say. */

```

```
#define WEAR_BOOT_MS   2400    /* the boot face, before the title card  */
#define WEAR_TITLE_MS  6400    /* how long the title card holds         */
#define ALERT_HOLD_MS  60000   /* how long an alert stays on the screen */

static uint32_t g_boot_ms      = 0;    /* millis() when the OLED came up */
static uint32_t g_last_draw_ms = 0;    /* the last frame put on the glass */

/* The classic font is six pixels wide per cell, at any text size, so
 * centring is arithmetic, not measurement. */
static uint8_t centre_x(const char *s, uint8_t size)
{
    return (uint8_t)((128 - (int)strlen(s) * 6 * size) / 2);
}

/* The waveform along the bottom edge -- the same two drifting sine
 * waves the panel's title card uses, drawn as filled bars.  It is the
 * one motif that says this screen and the hub's panel are the same
 * project. */
static void draw_waveform(uint8_t y_base, int frame)
{
    const float t = (float)frame * 0.18f;
    for (int i = 0; i < 21; i++) {
        const float p = t + i * 0.42f;
        const float a = 0.5f + 0.5f * sinf(p * 1.7f);
        const float b = 0.5f + 0.5f * sinf(p * 0.53f + 1.7f);
        const int bh = (int)(8.0f * (0.18f + 0.82f * a * b));
        display.fillRect(i * 6, y_base + 1 - bh, 4, bh, SSD1306_WHITE);
    }
}

/* The boot steps, in the panel's voice -- and in its capitalisation: a
 * sentence starts with a capital, the way the panel's own boot lines do
 * when they are read out. */
static const char *s_boot_steps[] = {
    "The display is up",
    "Bluetooth is awake",
    "Searching for the hub",
};

/* The device's name, on two lines because it is sixteen characters and
 * this screen is a hundred and twenty-eight pixels wide: at the panel's
 * headline size two, sixteen characters need a hundred and ninety-two
 * pixels.  Stacked, the two halves sit inside the screen and still read
 * as one name. */
#define DEVICE_NAME_TOP "AMAZING'S"
#define DEVICE_NAME_BOT "CRYNET"

static void draw_boot_face(uint32_t elapsed_ms, int frame)
{
    display.clearDisplay();
    display.setTextColor(SSD1306_WHITE);

    /* The wordmark -- the device's name, not the project's -- and under
     * it the breathing line that stands in for the panel's colour pulse:
     * a monochrome wordmark cannot brighten and dim, but the line under
     * it can swell and shrink. */
    display.setTextSize(2);
    display.setCursor(centre_x(DEVICE_NAME_TOP, 2), 3);
    display.print(DEVICE_NAME_TOP);
    display.setCursor(centre_x(DEVICE_NAME_BOT, 2), 19);
    display.print(DEVICE_NAME_BOT);
    display.setTextSize(1);
    display.setCursor(centre_x("INFANT CRY MONITOR", 1), 35);
    display.print("INFANT CRY MONITOR");

    const float pulse = 0.5f + 0.5f * sinf((float)frame * 0.12f);
    const int line_w = (int)(12.0f + 36.0f * pulse);
    display.fillRect(64 - line_w / 2, 44, line_w, 1, SSD1306_WHITE);

    /* What the wrist is doing as it comes up. */
    int step = (int)(elapsed_ms / (WEAR_BOOT_MS / 3));
    if (step > 2) { step = 2; }
    display.setCursor(centre_x(s_boot_steps[step], 1), 48);
    display.print(s_boot_steps[step]);

    /* The progress bar, same as the panel's. */
    const int bar_w = 80, bar_x = (128 - bar_w) / 2, bar_y = 57;
    const float prog = (float)elapsed_ms / (float)WEAR_BOOT_MS;
    const int filled = (int)(bar_w * (prog > 1.0f ? 1.0f : prog));
    display.drawRect(bar_x - 1, bar_y - 1, bar_w + 2, 6, SSD1306_WHITE);
    if (filled > 0) {
        display.fillRect(bar_x, bar_y, filled, 4, SSD1306_WHITE);
    }
    display.display();
}
```

```

/* The title card: what the device is, who built it, and the number it
 * is registered under -- the panel's own card, set for a screen eighty
 * pixels wide where the panel has three hundred and twenty.
 *
 * The full project title is ninety-seven characters and this font is
 * six pixels a cell, so the title cannot be laid out here the way it is
 * on the panel -- it would crowd out the name and the id number, which
 * matter more on a wrist.  The title stays on the hub's panel, which has
 * the room; this card carries the device's name and its owner's. */
static void draw_title_face(uint32_t now, int frame)
{
    (void)now;
    display.clearDisplay();
    display.setTextColor(SSD1306_WHITE);

    /* The accent band across the top, with its thicker leading edge. */
    display.fillRect(0, 0, 128, 2, SSD1306_WHITE);
    display.fillRect(0, 0, 34, 3, SSD1306_WHITE);

    /* The device's name -- sixteen characters, which at the panel's
     * headline size would need a hundred and ninety-two pixels, so it
     * sits on one line at the smaller size instead. */
    display.setTextSize(1);
    display.setCursor(centre_x(DEVICE_NAME_TOP " " DEVICE_NAME_BOT, 1), 9);
    display.print(DEVICE_NAME_TOP " " DEVICE_NAME_BOT);

    /* A rule, then the two things a panel has to say if the person
     * looking at it is the person who built it. */
    display.fillRect(28, 21, 72, 1, SSD1306_WHITE);

    display.setCursor(centre_x("AMAZING AYOMIKUN", 1), 27);
    display.print("AMAZING AYOMIKUN");
    display.setCursor(centre_x("ADELOWO", 1), 37);
    display.print("ADELOWO");
    display.setCursor(centre_x("22/05/04/0029", 1), 47);
    display.print("22/05/04/0029");

    /* The same waveform the panel runs along its edge, so the card is
     * alive in the same way the dashboard will be. */
    draw_waveform(62, frame * 2);
    display.display();
}

/* The dashboard, in the panel's language: a wordmark and a status pill
 * in the header, the news in the middle, and the waveform along the
 * bottom edge so the screen is visibly alive even when nothing is
 * happening. */
static void draw_dashboard(uint32_t now, int frame)
{
    const bool cry_now    = g_have_alert && g_type == 1 &&
                            (now - g_alert_ms) < ALERT_HOLD_MS;
    const bool deaf_now   = g_have_alert && g_type == 3 &&
                            (now - g_alert_ms) < ALERT_HOLD_MS;
    const bool cleared_now = g_have_alert && g_type == 2 &&
                            (now - g_alert_ms) < ALERT_HOLD_MS;

    display.clearDisplay();
    display.setTextColor(SSD1306_WHITE);
    display.setTextSize(1);

    /* ------------------------------------------------- the alert face */
    if (cry_now || deaf_now) {
        /* The band across the top, inverted -- the monochrome answer to
         * the panel's red.  It blinks while the wrist is buzzing, then
         * holds steady: urgency first, information after. */
        const bool band_on = (now - g_alert_ms) >= HAPTIC_MS ||
                             ((now / 250) % 2 == 0);
        if (band_on) {
            display.fillRect(0, 0, 128, 11, SSD1306_WHITE);
            display.setTextColor(SSD1306_BLACK);
        }
        display.setCursor(2, 2);
        display.print(cry_now ? "CRY" : "DEAF");

        /* Every cot that is crying, not only the one the alert names.
         * The alert names the loudest cot -- that is the hub's alert
         * policy, and it is what keeps one cry to one buzz -- while the
         * state characteristic carries the whole crying mask.  Two cots
         * hearing one cry, or two babies crying at once, both read as
         * the list here.
         *
         * The named cot is always in the list.  The alert is instant
         * and the mask can be a second behind it, or already clear
         * while this face still holds, so the mask alone could leave
```

```
         * the screen saying CRY with nobody named.
         *
         * A deaf cot is one cot by definition: it keeps the mask out. */
        char ids[16];
        uint8_t count = 0;
        if (cry_now) {
            const uint8_t limit = g_cots ? g_cots : 8;
            uint8_t mask = g_mask;
            if (g_cot >= 1 && g_cot <= 8) {
                mask |= (uint8_t)(1u << (g_cot - 1));
            }
            size_t used = 0;
            ids[0] = '\0';
            for (uint8_t bit = 0; bit < 8 && bit < limit; ++bit) {
                if (!(mask & (1u << bit))) {
                    continue;
                }
                ++count;
                if (used && used < sizeof ids - 1) {
                    ids[used++] = ' ';
                }
                if (used < sizeof ids - 1) {
                    ids[used++] = (char)('1' + bit);
                }
                ids[used] = '\0';
            }
        }
        const bool many = count > 1;

        char pill[8];
        if (many) {
            snprintf(pill, sizeof pill, "COTS");
        } else {
            snprintf(pill, sizeof pill, "COT %u", (unsigned)g_cot);
        }
        display.setCursor(126 - (int)strlen(pill) * 6, 2);
        display.print(pill);
        display.setTextColor(SSD1306_WHITE);

        /* Who, large -- read from across a room.  One cot keeps the
         * size-3 "COT n".  Both cots crying reads "COTS 1 2", which is
         * 144 px at size 3 on a 128 px glass, so a list drops to size 2
         * (96 px); the size falls again rather than let snprintf cut a
         * cot off the end of the string.  The loop still walks all eight
         * mask bits, because the state characteristic's mask byte is
         * eight bits wide even though this hub only ever sets two. */
        char big[24];
        if (many) {
            snprintf(big, sizeof big, "COTS %s", ids);
        } else {
            snprintf(big, sizeof big, "%s", pill);
        }
        uint8_t big_size = many ? 2 : 3;
        while (big_size > 1 && (int)strlen(big) * 6 * big_size > 126) {
            --big_size;
        }
        display.setTextSize(big_size);
        display.setCursor(centre_x(big, big_size), many ? 20 : 17);
        display.print(big);
        display.setTextSize(1);

        if (cry_now) {
            /* The confidence, on a bar: the panel's level bar, scaled
             * to what the model is sure of rather than to how loud the
             * room is -- a wrist has no microphone.  With several cots
             * crying this is the loudest cry's confidence, which is the
             * only figure the hub sends; it is not a per-cot number. */
            const int bar_w = 86, bar_y = 45;
            display.drawRect(-1, bar_y - 1, bar_w + 2, 8, SSD1306_WHITE);
            const int filled = (int)(bar_w * (g_conf_pct / 100.0f));
            if (filled > 0) {
                display.fillRect(0, bar_y, filled, 6, SSD1306_WHITE);
            }
            char pct[6];
            snprintf(pct, sizeof pct, "%u%%", (unsigned)g_conf_pct);
            display.setCursor(126 - (int)strlen(pct) * 6, 46);
            display.print(pct);
        } else {
            /* The cot has gone deaf: the hub cannot get its audio, so
             * nobody is watching that cot at all. */
            display.setCursor(centre_x("The hub cannot", 1), 44);
            display.print("The hub cannot");
            display.setCursor(centre_x("hear that cot", 1), 53);
            display.print("hear that cot");
            display.display();
```

```
            return;
        }

        char line[24];
        snprintf(line, sizeof line, "Event #%u  %us ago",
                 (unsigned)g_event,
                 (unsigned)((now - g_alert_ms) / 1000));
        display.setCursor(0, 57);
        display.print(line);
        display.display();
        return;
    }

    /* ------------------------------------------------- the header */
    display.setCursor(0, 0);
    display.print("CRYNET");
    /* The pill: solid when the hub is attached, hollow while it scans --
     * the panel's two-state pill, reduced to outline and fill. */
    if (g_connected) {
        display.fillRect(84, 1, 4, 4, SSD1306_WHITE);
        display.setCursor(92, 0);
        display.print("LINK");
    } else {
        display.drawRect(84, 1, 4, 4, SSD1306_WHITE);
        display.setCursor(92, 0);
        display.print("SCAN");
    }

    /* ------------------------------------------------ the middle news */
    if (cleared_now) {
        display.setTextSize(2);
        display.setCursor(centre_x("CLEARED", 2), 22);
        display.print("CLEARED");
        display.setTextSize(1);
    } else if (g_connected) {
        if (g_mask) {
            display.setTextSize(2);
            display.setCursor(0, 20);
            display.print("CRYING");
            display.setTextSize(1);
            display.setCursor(0, 42);
            for (uint8_t bit = 0; bit < 8 && bit < (g_cots ? g_cots : 8); ++bit) {
                if (g_mask & (1u << bit)) {
                    display.printf("COT %u ", (unsigned)(bit + 1));
                }
            }
        } else {
            display.setTextSize(2);
            display.setCursor(centre_x("ALL CLEAR", 2), 22);
            display.print("ALL CLEAR");
            display.setTextSize(1);
            display.setCursor(centre_x("Every cot is quiet", 1), 42);
            display.print("Every cot is quiet");
        }
    } else {
        display.setCursor(centre_x("Searching", 1), 22);
        display.print("Searching");
        display.setCursor(centre_x("for the hub...", 1), 34);
        display.print("for the hub...");
    }

    /* ------------------------------------------------------ the edge */
    draw_waveform(64, g_mask ? frame * 2 : frame);
    display.display();
}

/* The splash and the dashboard, whichever face the clock calls for. */
static void render(void)
{
    if (!g_oled_ok) {
        return;
    }
    const uint32_t now = millis();
    const uint32_t up  = now - g_boot_ms;

    if (up < WEAR_BOOT_MS) {
        draw_boot_face(up, (int)(now / 50));
        return;
    }
    if (up < WEAR_BOOT_MS + WEAR_TITLE_MS) {
        draw_title_face(now, (int)(now / 50));
        return;
    }
    draw_dashboard(now, (int)(now / 50));
}
```

```
501
502 /* ---------------------------------------------------------------- BLE -- */
503
504 static void alert_notify_cb(BLERemoteCharacteristic * /*chr*/,
505                             uint8_t *data, size_t len, bool /*isNotify*/)
506 {
507     if (len < 6) {
508         Serial.printf("[ble] short alert (%u bytes) ignored\n", (unsigned)len);
509         return;
510     }
511     /* What the wrist was told last, so one cry is one buzz. */
512     const uint8_t prev_type = g_type;
513     g_cot      = data[0];
514     g_type     = data[1];
515     g_conf_pct = (uint16_t)data[2] | ((uint16_t)data[3] << 8);
516     g_event    = (uint16_t)data[4] | ((uint16_t)data[5] << 8);
517     g_have_alert = true;
518     g_alert_ms   = millis();
519     g_dirty      = true;
520     Serial.printf("[ble] ALERT cot %u type %u conf %u%% event #%u\n",
521                   (unsigned)g_cot, (unsigned)g_type,
522                   (unsigned)g_conf_pct, (unsigned)g_event);
523     if (g_type == 1 && prev_type != 1) {
524         /* One buzz per cry, not one per announcement.  The hub names
525          * the loudest cot, and it re-issues the alert when a louder
526          * cot joins the cry -- the display follows the baby from cot
527          * to cot, but the wrist buzzes once. */
528         haptic_trigger();
529     } else if (g_type == 3 && prev_type != 3) {
530         /* That cot has gone deaf: the hub cannot get its audio, so it
531          * cannot announce a cry.  It is not being watched, and the
532          * caregiver has to hear that from the wrist, not only see it
533          * on the hub's panel. */
534         Serial.printf("[ble] cot %u is DEAF\n", (unsigned)g_cot);
535         haptic_trigger();
536     }
537 }
538
539 /* The state characteristic arrives on its own: the hub publishes it
540  * whenever it changes and at least once a second while a link is up, so
541  * the idle face's crying mask is never more than a second stale and
542  * nothing here ever blocks waiting for a read the hub may never answer. */
543 static void state_notify_cb(BLERemoteCharacteristic * /*chr*/,
544                             uint8_t *data, size_t len, bool /*isNotify*/)
545 {
546     if (len < 8) {
547         Serial.printf("[ble] short state (%u bytes)\n", (unsigned)len);
548         return;
549     }
550     g_cots = (uint8_t)data[0];
551     g_mask = (uint8_t)data[1];
552     /* data[3] carries the loudest cry's confidence; started_at is data[4..7]. */
553     g_dirty = true;
554     Serial.printf("[ble] state: %u cot(s), mask 0x%02X\n",
555                   (unsigned)g_cots, (unsigned)g_mask);
556 }
557
558 class HubClientCallbacks : public BLEClientCallbacks {
559     void onConnect(BLEClient *) override
560     {
561         Serial.println("[ble] hub connected");
562     }
563     void onDisconnect(BLEClient *) override
564     {
565         Serial.println("[ble] hub lost — rescanning");
566         g_connected = false;
567         g_alert     = nullptr;
568         g_state     = nullptr;
569         /* The mask describes the hub's cots as they were when the link
570          * was up.  The alert face outlives a drop by up to a minute, so
571          * a stale mask left here would name cots nobody is watching any
572          * more.  Cleared, the face falls back to the cot the alert
573          * named. */
574         g_mask      = 0;
575         g_cots      = 0;
576         g_dirty     = true;
577     }
578 };
579
580 static bool connect_to_hub(void)
581 {
582     Serial.printf("[ble] connecting to %s\n", g_hub_device->getAddress().toString().c_str());
583
584     /* Connect to the remote BLE Server. */
585     if (!g_client->connect(g_hub_device)) {
```

```
        Serial.println("[ble] connect failed");
        return false;
    }

    /* Fresh service discovery, every connection.  The library keeps the
     * previous connection's service handles across a disconnect —
     * m_haveServices survives it — so without this the second connect
     * hands back handles that point at a peer that no longer exists:
     * subscriptions go nowhere and the link looks alive but is deaf. */
    if (g_client->getServices()->empty()) {
        Serial.println("[ble] service discovery failed");
        g_client->disconnect();
        return false;
    }

    BLERemoteService *svc = g_client->getService(BLEUUID(SERVICE_UUID));
    if (!svc) {
        Serial.println("[ble] alert service not found");
        g_client->disconnect();
        return false;
    }
    g_alert = svc->getCharacteristic(BLEUUID(ALERT_UUID));
    if (!g_alert) {
        Serial.println("[ble] alert characteristic not found");
        g_alert = nullptr;
        g_client->disconnect();
        return false;
    }
    if (!g_alert->subscribe(true, alert_notify_cb, false)) {
        Serial.println("[ble] alert subscribe failed");
        g_alert = nullptr;
        g_client->disconnect();
        return false;
    }
    g_state = svc->getCharacteristic(BLEUUID(STATE_UUID));
    if (!g_state || !g_state->subscribe(true, state_notify_cb, false)) {
        Serial.println("[ble] state subscribe failed");
        g_state = nullptr;
        g_client->disconnect();
        return false;
    }

    if (!g_client->isConnected()) {
        /* The hub dropped us while the subscriptions were going up. */
        Serial.println("[ble] hub dropped during setup");
        return false;
    }

    g_connected = true;
    g_dirty     = true;
    Serial.println("[ble] subscribed to alerts + state");
    return true;
}

class HubScanCallbacks : public BLEAdvertisedDeviceCallbacks {
    void onResult(BLEAdvertisedDevice advertisedDevice) override
    {
        const bool name_ok = advertisedDevice.getName() == HUB_NAME;
        const bool uuid_ok = advertisedDevice.haveServiceUUID() &&
                             advertisedDevice.isAdvertisingService(BLEUUID(SERVICE_UUID));
        if (!name_ok && !uuid_ok) {
            return;
        }
        Serial.printf("[ble] hub found (%s)\n",
                      advertisedDevice.getAddress().toString().c_str());
        BLEDevice::getScan()->stop();
        delete g_hub_device;
        g_hub_device = new BLEAdvertisedDevice(advertisedDevice);
        g_do_connect = true;
    }
};

/* ---------------------------------------------------------------- main -- */

/* Defined below; setup() hands it to the scheduler. */
static void ble_task(void *arg);

void setup()
{
    Serial.begin(115200);
    /* Serial on the C3 is CDC: a write with no reader (a laptop asleep,
     * a cable in a charger) blocks until the buffer drains, and prints
     * made from BLE callbacks stall the radio that has to keep the
     * link alive.  Zero timeout — never wait on the console. */
    Serial.setTxTimeoutMs(0);
```

```
    delay(300);
    Serial.println("\nCryNet wearable — BLE alert client (stage 2d)");

    /* The motor's switch first, and low: the transistor must be off
     * from the first instruction, not from whenever setup() gets
     * round to it. */
    pinMode(PIN_MOTOR, OUTPUT);
    digitalWrite(PIN_MOTOR, LOW);

    Wire.begin(PIN_SDA, PIN_SCL);
    /* The waveform and the title ticker redraw ten times a second; at
     * the default 100 kHz each frame costs most of that budget, so the
     * bus runs at 400 kHz. */
    Wire.setClock(400000);
    g_oled_ok = display.begin(SSD1306_SWITCHCAPVCC, 0x3C);
    if (!g_oled_ok) {
        Serial.println("[oled] SSD1306 not found at 0x3C — check SDA 5 / SCL 6");
    } else {
        Serial.println("[oled] up");
    }
    g_boot_ms      = millis();   /* the splash starts here */
    g_last_draw_ms = g_boot_ms;
    g_dirty = true;

    BLEDevice::init("CRYNET_WEARABLE");
    g_client = BLEDevice::createClient();
    g_client->setClientCallbacks(new HubClientCallbacks());
    BLEScan *scan = BLEDevice::getScan();
    scan->setAdvertisedDeviceCallbacks(new HubScanCallbacks(), false);
    scan->setActiveScan(true);
    scan->setInterval(50);
    scan->setWindow(30);

    /* The BLE worker gets a generous stack: service discovery and the
     * library's own bookkeeping run on it.  Priority 1 — below nothing
     * the user sees, above the idle task. */
    xTaskCreate(ble_task, "ble", 6144, nullptr, 1, nullptr);
}

/* The BLE task.  Every library call here blocks its caller — the two-
 * argument scan start waits on a semaphore forever, connect can hold
 * for up to thirty seconds, and a characteristic read waits for a peer
 * that may never answer — so it gets its own task and the render loop
 * never waits on the radio.  Unplugging the hub can stall this task for
 * a while; the screen, the buzz and the animation keep going. */
static void ble_task(void * /*arg*/)
{
    for (;;) {
        if (g_do_connect) {
            g_do_connect = false;
            if (!connect_to_hub()) {
                g_connected = false;      /* the scan resumes; the client is reused */
                g_dirty     = true;
            }
        } else if (!g_connected) {
            BLEScan *scan = BLEDevice::getScan();
            scan->start(2, false);        /* scan in two-second windows */
            scan->clearResults();
        }
        vTaskDelay(pdMS_TO_TICKS(20));
    }
}

void loop()
{
    haptic_tick();

    /* The screen is animated — the waveform moves, the title ticker
     * crawls, the alert band blinks — so it redraws on a steady beat,
     * and at once whenever new information arrives.  The render itself
     * decides which face the clock calls for, so an alert also leaves
     * the screen on its own when its minute is up. */
    const uint32_t now = millis();
    if (g_dirty || now - g_last_draw_ms >= 100) {
        g_dirty        = false;
        g_last_draw_ms = now;
        render();
    }
    delay(10);
}
```

firmware/wearable/haptic_test/haptic_test.ino

70 lines · 2,655 bytes

```
/*
 * Haptic motor bench test -- the wearable's motor, on a breadboard.
 *
 * What it does: drives the BC547 low-side switch on GPIO7 exactly the
 * way the real wearable firmware does (D-77), so the motor runs for
 * 3 seconds, stops for 2 seconds, and repeats until the board is
 * unplugged.  Watch the motor, or watch the serial log at 115200 baud.
 *
 * Wiring (from SYSTEM_WIRING.md 5.3 / the page 5 diagram):
 *
 *   3V3 ---[22 ohm in the motor's + lead]--- motor --- collector  BC547
 *                                                          |
 *   GPIO7 ---[2.2 k]--- base                          emitter --- GND
 *                                                          |
 *                        1N5819 across the motor, cathode UP to the + net
 *                        100 k from base to emitter
 *
 * The motor's + leg must come from the 3V3 pin, the BC547's emitter to
 * GND, and the flyback diode's striped end (cathode) towards the + net.
 * A backwards diode does not just make the motor quiet -- it shorts the
 * motor and pushes the whole supply current through the BC547.
 *
 * This sketch replaces the wearable firmware on the board.  Put the real
 * one back afterwards with:
 *
 *   python firmware/tools/flash_node.py --port COM30 --sketch wearable
 *
 * (or flash the built image from firmware/prebuilt/wearable/).
 */

#include <Arduino.h>

/* The BC547 haptic switch (D-77): GPIO7 -> 2.2 k -> base.  HIGH turns the
 * transistor on and the motor spins; LOW lets it stop. */
static const uint8_t PIN_MOTOR = 7;

/* The cycle the owner asked for: three seconds on, two seconds off. */
static const uint32_t ON_MS  = 3000;
static const uint32_t OFF_MS = 2000;

void setup()
{
    /* The transistor must be off before anything else happens here: the
     * 100 k base-emitter pulldown holds it off at reset, and this makes
     * sure the motor never twitches while the sketch starts. */
    pinMode(PIN_MOTOR, OUTPUT);
    digitalWrite(PIN_MOTOR, LOW);

    Serial.begin(115200);
    unsigned long t0 = millis();
    while (!Serial && millis() - t0 < 2000) {
        delay(10);
    }

    Serial.println();
    Serial.println("[haptic-test] GPIO7 drives the BC547 low-side switch.");
    Serial.println("[haptic-test] motor ON for 3 s, OFF for 2 s, forever.");
    Serial.println("[haptic-test] ctrl-c stops the log; the board keeps going.");
}

void loop()
{
    digitalWrite(PIN_MOTOR, HIGH);
    Serial.println("[haptic-test] motor ON");
    delay(ON_MS);

    digitalWrite(PIN_MOTOR, LOW);
    Serial.println("[haptic-test] motor OFF");
    delay(OFF_MS);
}
```

The training pipeline

research/build_notebook.py

2293 lines · 101,809 bytes

```
"""Generates notebooks/crynet_training.ipynb -- the full infant-cry training pipeline.

Kept as a builder script (not raw JSON) so the notebook source is reviewable and
regeneratable.  Run:  python build_notebook.py
"""
import json
from pathlib import Path

ROOT = Path(__file__).resolve().parent.parent
OUT = ROOT / "notebooks" / "crynet_training.ipynb"

CELLS = []


def md(text):
    CELLS.append(("markdown", text.strip("\n")))


def code(text):
    CELLS.append(("code", text.strip("\n")))


# ============================================================================
md(r"""
# CryNet — Infant Cry Detection for ESP32-S3

**Binary cry / no-cry classifier, trained on 28.5k real recordings, exported as an INT8 TensorFlow Lite model for TFLite Micro on the ESP32-S3 Central Hub.**

---

## What this notebook does

| # | Stage | Output |
|---|-------|--------|
| 1 | **Acquire** — ICD corpus (28,478 clips), Baby-Cry-4, ESC-50, Snoring, GTZAN music | cached archives |
| 2 | **Frontend** — 16 kHz mono → 81×64 log-Mel (bit-reproducible with the C firmware) | `crynet_tables.h` |
| 3 | **Build** — 2 s windows, "any cry present" labelling, real-noise + RIR augmentation | feature matrix |
| 4 | **Train** — 3 CNN variants, pick the best on validation AUC | model |
| 5 | **Mine** — score the training set, duplicate the worst mistakes, fine-tune | better model |
| 6 | **Calibrate** — threshold sweep for ≥95 % recall on the cry class | alarm threshold |
| 7 | **Quantize** — full INT8 post-training quantization + numerical parity check | `crynet_int8.tflite` |
| 8 | **Export** — TFLite + C header + C reference DSP + TFLM glue + contract | `out/` |

---

## The data (why this is the right corpus)

**ICD — `hoyyu1/infant-cry-detection` (primary).** 28,478 clips, *all* 16 kHz mono / 5 s, pre-assigned 5 balanced folds.

| Class | n | Source |
|---|---|---|
| cry | 5,026 | EnesBabyCries |
| cry | 4,552 | CryCeleb2023 |
| cry | 3,752 | iFLYTEK 2020 (Xunfei) |
| non-cry | 4,963 | VoxCeleb **speech** |
| non-cry | 4,506 | DASEE **domestic** noise |
| non-cry | 2,076 | human non-speech |
| non-cry | 1,953 | **cat** meows |
| non-cry | 1,650 | silence |

Three independent cry corpora (hundreds of different infants, three recording chains) against speech, household noise, cat meows and silence — the exact confusion set that produces false alarms in a nursery.

**Supplementary corpora** (cry diversity + *held-out cross-corpus* test sets):
`karimeletriby/infant-cry-classification-4-classes` (4,871 real cries, 16 kHz, cross-dataset deduplicated), ESC-50 (1,960 non-cry + 40 real `crying_baby`), Snoring (1,000 adult-vocal hard negatives), **GTZAN (1,000 real music clips, ten genres)**, and **loud speech** (the speech classes of FSD50K at 16 kHz — `Speech`, `Child_speech_and_kid_speaking`, `Conversation`, `Chatter`, `Crowd`, `Human_voice`, laughter, singing — built into one zip by `research/loudspeech/build_speech_zip.py`).

> Every corpus is split **by source clip**, never by window, so no recording can appear in both train and test.

**Why music is in there.** The first false-alarm map over ESC-50 (2026-10-31) showed the model rejecting speech, household noise and human non-speech well (1.6 %) but had **never seen music at all** — no corpus in the mix contained any, and ESC-50 does not either. A nursery has a radio, a television and a mobile playing something, so music is a first-class confusion class here, not an
```

```
afterthought. GTZAN's ten genres give the model real music to reject and the cross-corpus split gives a clean "music the model was not trained on" measurement afterwards.

  70  **Why loud speech is in there.** The corpus has speech in it — 4,963 VoxCeleb clips that came in with the ICD negatives — but no *loud* speech and no children's voices, and the one measured conversation window in the room scored **0.9727**, which is below the saturation the hub announces on (0.9900) but not by much. A nursery is full of people talking, and a child's voice is the closest thing to a baby's cry that a real room contains. The speech is sourced online rather than recorded: the user has nobody around to read a sentence into a microphone.
  71
  72  ---
  73
  74  ## The firmware contract (read this before touching the C code)
  75
  76  The model is only as good as the front end it is fed. This notebook fixes the contract:
  77
  78  ```
  79  int16_t pcm[32000]                2.000 s @ 16 kHz, mono
  80     -> float x = pcm/32768.0
  81     -> subtract mean (DC removal; the INMP441 sits at ~+18000 counts)
  82     -> zero-pad 256 front / 512 rear
  83     -> 81 frames, 512-pt Hann window, hop 400
  84     -> |rfft|^2  (257 bins)
  85     -> 64 Mel bands (librosa slaney mel, 20 Hz - 8 kHz)
  86     -> 10*log10(max(P,1e-10)), floored at -80 dB
  87     -> per-window standardisation (mean 0, std 1)
  88     -> int8 quantise: q = round(v/scale) + zero_point
  89     -> TFLite Micro int8 inference -> P(cry)
  90  ```
  91
  92  `out/dsp_crynet.c` implements exactly this, and `out/crynet_tables.h` carries the *exact* Mel filterbank and Hann window used in training, so the firmware cannot drift from the model. A parity test (Python reference vs compiled C) runs at the end of this notebook if a C compiler is available.
  93
  94  ---
  95
  96  ## How to run (Kaggle)
  97
  98  1. **Nothing to attach.** The notebook fetches everything it needs: the cry corpora from Hugging Face and Kaggle, and the **loud-speech corpus** (6,000 clips of real speech — FSD50K's nursery classes: children's voices, laughter, chatter, singing; plus Common Voice) from its published URL. Import the notebook, press *Run All*, and it builds itself. If the speech corpus cannot be fetched the run **stops** rather than quietly training on the old mix.
  99  2. Accelerator: **GPU T4**.
 100  3. Internet: **On** — the notebook pulls **~5.8 GB** of audio (ICD + cry4 + ESC-50 + Snoring + GTZAN) and caches it under `./_cache`. The Hugging Face files are fetched with `huggingface_hub`, so an interrupted transfer resumes instead of starting over.
 101  4. `Run All` (or *Save Version* → *Save & Run All*).
 102  5. **Download `crynet_artifacts.zip` from the Output tab** of the saved version. The last cell packages everything the deployment needs: the INT8 model, the calibrated tables, the model as a C array, the metrics and the front-end sources.
 103
 104  Expected wall-clock on Kaggle (T4): ~10 min download + ~10 min feature extraction + **~30-45 min training** for the one variant the hub runs (`conv_cnn`).
 105
 106  Useful switches:
 107
 108  | variable | effect |
 109  |---|---|
 110  | `CRYNET_FAST=1` | subsample the corpora and train for 3 epochs — local smoke test |
 111  | `CRYNET_OFFLINE=1` | never download; use whatever is already in `CRYNET_CACHE` |
 112  | `CRYNET_KINDS` | which variants to train, e.g. `temporal_cnn` (default: `conv_cnn` — the one the hub runs) |
 113  | `CRYNET_EPOCHS` | training epochs (default 25) |
 114  | `CRYNET_CACHE` | where the archives live (default `./_cache`) |
 115  | `CRYNET_OUT` | where the artifacts are written (default `./out`) |
 116
 117  Local smoke test: `CRYNET_FAST=1 python run_notebook_local.py`.
 118  """)
 119
 120  # ============================================================================
 121  code(r"""
 122  # ============================================================================
 123  # CELL 1 — Environment, configuration, contract constants
 124  # ============================================================================
 125  import os
 126  import sys
 127  import io
 128  import csv
 129  import json
 130  import math
 131  import time
 132  import random
 133  import hashlib
 134  import shutil
 135  import zipfile
 136  import threading
 137  import warnings
 138  import subprocess
 139  from pathlib import Path
```

```
from concurrent.futures import ThreadPoolExecutor

warnings.filterwarnings("ignore")
os.environ.setdefault("TF_CPP_MIN_LOG_LEVEL", "2")

import numpy as np

FAST = os.environ.get("CRYNET_FAST", "0") == "1"          # local smoke-test mode

# On Kaggle the notebook is run from the UI, where there is nowhere to set
# an environment variable.  This run trains the one variant the hub runs
# (conv_cnn); the other two were explored earlier and are not deployed.
if "CRYNET_KINDS" not in os.environ:
    os.environ["CRYNET_KINDS"] = "conv_cnn"
CACHE = Path(os.environ.get("CRYNET_CACHE", "./_cache")).resolve()
WORK = Path(os.environ.get("CRYNET_WORK", "./_work")).resolve()
OUT = Path(os.environ.get("CRYNET_OUT", "./out")).resolve()
for _d in (CACHE, WORK, OUT):
    _d.mkdir(parents=True, exist_ok=True)

SEED = 1337
random.seed(SEED)
np.random.seed(SEED)

# ---------------------------------------------------------------------------
# SIGNAL CONTRACT — these numbers are mirrored into the C firmware.
# Change one and the C side is regenerated automatically, but the trained
# weights would no longer match, so treat them as frozen.
# ---------------------------------------------------------------------------
SR          = 16000        # Hz
WIN_SEC     = 2.0          # seconds per inference window
N_SAMPLES   = int(SR * WIN_SEC)      # 32000
N_FFT       = 512
HOP         = 400
N_MELS      = 64
FMIN        = 20.0
FMAX        = 8000.0
CENTER_PAD  = 256          # zero pad at the front (librosa center=True behaviour)
TAIL_PAD    = 512          # zero pad at the rear
N_FRAMES    = 1 + (N_SAMPLES + CENTER_PAD + TAIL_PAD - N_FFT) // HOP   # 81
DB_FLOOR    = -80.0
LOG_EPS     = 1e-10
CMVN        = True         # per-window mean/variance normalisation

# Inference gate: below this the window is (near) silent and the model is not run.
# The ICD recordings sit around RMS 0.006 (~-44 dBFS), and a distant whimper
# can be quieter, so the gate is deliberately low.  The model itself rejects
# near-silent windows; this gate only skips work.
SILENCE_RMS_GATE = 0.0015  # ~ -56 dBFS on a 16-bit normalised signal
TARGET_RECALL = 0.95       # alarm threshold is chosen to reach this on the cry class

assert N_FRAMES == 81, N_FRAMES

print("=" * 72)
print("CryNet — infant cry detection for ESP32-S3")
print("=" * 72)
print(f"  mode          : {'FAST (local smoke test)' if FAST else 'FULL'}")
print(f"  signal        : {SR} Hz mono, {WIN_SEC:.1f} s window ({N_SAMPLES} samples)")
print(f"  log-mel       : {N_FRAMES} frames x {N_MELS} bands, n_fft={N_FFT}, hop={HOP}")
print(f"  dB floor      : {DB_FLOOR} dB   CMVN: {CMVN}")
print(f"  cache / work  : {CACHE}  |  {WORK}")
print(f"  artifacts     : {OUT}")
print("=" * 72)
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 2 — Dependencies
# ============================================================================
def _pip(*pkgs):
    subprocess.run([sys.executable, "-m", "pip", "install", "-q", *pkgs],
                   check=False, capture_output=True)


for _p in ("librosa", "soundfile", "soxr", "scipy", "sklearn"):
    try:
        __import__(_p)
    except Exception:
        _pip(_p)

import soundfile as sf
import librosa
import scipy.signal as sps
import tensorflow as tf
```

```
print("tensorflow", tf.__version__, "| librosa", librosa.__version__)

try:
    gpus = tf.config.list_physical_devices("GPU")
    for g in gpus:
        tf.config.experimental.set_memory_growth(g, True)
    print("GPU:", gpus if gpus else "none (CPU mode — expect it to be slower)")
except Exception as e:  # pragma: no cover
    print("GPU probe failed:", e)

tf.random.set_seed(SEED)
""")

# ============================================================================
code(r"""
# ========================================================================
# CELL 3 — Dataset acquisition
#
#  HF     : hoyyu1/infant-cry-detection  (28,478 clips, 16 kHz, 5 s, 5 folds)
#           fetched with huggingface_hub (resumable + retried) — a 4.6 GB
#           transfer must not be lost to a dropped connection.
#  Kaggle : public download endpoint, no auth required:
#           https://www.kaggle.com/api/v1/datasets/download/{owner}/{slug}
#
#  A file attached to the notebook (/kaggle/input) is used as-is.
#  Everything is cached, so a re-run costs nothing.
#  Set CRYNET_OFFLINE=1 to require the archives to be present already.
# ========================================================================
HF_REPO = "hoyyu1/infant-cry-detection"
HF_URL = "https://huggingface.co/datasets/{repo}/resolve/main/{path}"
KAGGLE_URL = "https://www.kaggle.com/api/v1/datasets/download/{ref}"
# The loud-speech corpus is published once, so that this notebook is
# self-contained: import it, press Run All, and it fetches everything it
# needs.  (It used to have to be dragged into Kaggle by hand.)
SPEECH_URL = ("https://github.com/Amazo123/CryNet/releases/download/"
              "speech-corpus-v1/speech__fsd50k_speech.zip")
SPEECH_SHA256 = "eeea29323c3d02154cb058388b6ac727a2dc33571ed13e773b8c5659423e0ac5"
OFFLINE = os.environ.get("CRYNET_OFFLINE", "0") == "1"
KAGGLE_INPUT = Path("/kaggle/input") if Path("/kaggle/input").exists() else None

DATASETS = {
    # key        : (kind, ref, member, approx size in bytes)
    "icd_meta": ("hf",     HF_REPO, "metadata/all28479.csv", 1_700_000),
    "icd_pos":  ("hf",     HF_REPO, "audio/positive.zip",    2_160_000_000),
    "icd_neg":  ("hf",     HF_REPO, "audio/negative.zip",    2_480_000_000),
    "icd_rir":  ("hf",     HF_REPO, "rir.zip",               110_000_000),
    "cry4":    ("kaggle", "karimeletriby/infant-cry-classification-4-classes", None, 495_000_000),
    "esc50":   ("kaggle", "akaliutau/esc-50", None, 617_000_000),
    "snore":   ("kaggle", "tareqkhanemu/snoring", None, 118_000_000),
    # Music.  Nothing else in the corpus contains any: the model had never
    # been trained to reject a song, a radio or a television, and a nursery
    # has all three.  GTZAN is ten genres of real recorded music.
    "gtzan":   ("kaggle", "carlthome/gtzan-genre-collection", None, 1_226_000_000),
    # Loud speech.  The corpus has speech in it — 4,963 VoxCeleb clips that
    # came in with the ICD negatives — but no *loud* speech, and no
    # children's voices.  A nursery is full of people talking, and a
    # child's voice is the closest thing to a baby's cry that a real room
    # contains.  The clips are the speech classes of FSD50K at 16 kHz,
    # built into one zip by research/loudspeech/build_speech_zip.py and
    # published as a release asset, so the notebook fetches it by URL.
    # Unlike the corpora above this one is NOT optional: without it the
    # run would train on the old mix and quietly miss the whole point,
    # so a failure here stops the run instead of hiding.
    "speech":  ("url",   SPEECH_URL, "speech__fsd50k_speech.zip", 793_000_000),
}
OPTIONAL = {"icd_rir"}


def canonical_path(key):
    kind, ref, member, _sz = DATASETS[key]
    if kind == "hf":
        return CACHE / member
    if kind == "url":
        return CACHE / member
    if kind == "local":
        return CACHE / "speech__fsd50k_speech.zip"
    return CACHE / (key + "__" + ref.replace("/", "_") + ".zip")


def _attached(basename):
    '''A copy attached to the notebook (/kaggle/input) wins over downloading.'''
    if KAGGLE_INPUT is None:
        return None
    try:
```

```
        for f in KAGGLE_INPUT.rglob(basename):
            if f.is_file():
                return f
    except Exception:
        pass
    return None


def _curl(url, dest):
    dest.parent.mkdir(parents=True, exist_ok=True)
    tmp = dest.with_suffix(dest.suffix + ".part")
    with tmp.open("wb") as fh:
        subprocess.run(["curl", "-sSL", "--max-time", "5400", "-o", str(tmp), url], check=True)
    tmp.replace(dest)


def _hf_download(member, dest):
    '''Resumable download; falls back to curl if the hub library is unavailable.'''
    try:
        from huggingface_hub import hf_hub_download
        return Path(hf_hub_download(HF_REPO, member, repo_type="dataset",
                                    local_dir=str(CACHE)))
    except Exception as e:
        print(f"  huggingface_hub unavailable ({e}); using curl")
        _curl(HF_URL.format(repo=HF_REPO, path=member), dest)
        return dest


def acquire(key):
    kind, ref, member, size = DATASETS[key]
    dest = canonical_path(key)
    if kind == "local":
        # Built on this machine by research/loudspeech/build_speech_zip.py.
        # It is not downloaded, so there is nothing to fetch: it is either
        # there or it is not, and the caller says which.
        if dest.exists():
            print(f"  local      {key:9s} {dest.stat().st_size/1e6:8.1f} MB")
            return dest
        # On Kaggle the zip is attached as an input dataset instead of
        # being built here, and it is found by name under /kaggle/input.
        att = _attached(dest.name)
        if att is not None:
            print(f"  attached   {key:9s} {att}  ({att.stat().st_size/1e6:.1f} MB)")
            return att
        raise FileNotFoundError(
            f"{key} not built at {dest}"
            + (f" and no {dest.name} is attached to this notebook"
               if KAGGLE_INPUT is not None else ""))
    if kind == "url":
        # Published once, fetched like any other corpus.  A copy attached
        # to the notebook still wins, so a run that has it locally does
        # not pay for the download.
        if dest.exists() and dest.stat().st_size >= size * 0.98:
            print(f"  cached     {key:9s} {dest.stat().st_size/1e6:8.1f} MB")
            return dest
        if OFFLINE:
            raise FileNotFoundError(f"{key} missing at {dest} (CRYNET_OFFLINE=1)")
        att = _attached(dest.name)
        if att is not None:
            print(f"  attached   {key:9s} {att}  ({att.stat().st_size/1e6:.1f} MB)")
            return att
        t0 = time.time()
        _curl(ref, dest)
        got = dest.stat().st_size
        if got < size * 0.98:
            dest.unlink(missing_ok=True)
            raise IOError(f"{key}: the download is short ({got} bytes, "
                          f"expected about {size})")
        if key == "speech":
            # A truncated or corrupted archive would show up much later as
            # a quietly smaller corpus, so it is checked here.
            h = hashlib.sha256()
            with dest.open("rb") as fh:
                for blk in iter(lambda: fh.read(1 << 20), b""):
                    h.update(blk)
            if h.hexdigest() != SPEECH_SHA256:
                dest.unlink(missing_ok=True)
                raise IOError(f"{key}: the archive is not the one that was "
                              f"published (sha256 {h.hexdigest()})")
        print(f"  downloaded {key:9s} {dest.stat().st_size/1e6:8.1f} MB  "
              f"({time.time()-t0:.0f}s)")
        return dest
    if dest.exists() and (OFFLINE or dest.stat().st_size >= size * 0.98):
        print(f"  cached     {key:9s} {dest.stat().st_size/1e6:8.1f} MB")
        return dest
```

```
    if OFFLINE:
        raise FileNotFoundError(f"{key} missing at {dest} (CRYNET_OFFLINE=1)")

    att = _attached(Path(member).name if kind == "hf" else dest.name)
    t0 = time.time()
    if att is not None:
        dest.parent.mkdir(parents=True, exist_ok=True)
        shutil.copyfile(att, dest)
        print(f"  attached   {key:9s} {dest.stat().st_size/1e6:8.1f} MB")
    elif kind == "hf":
        _hf_download(member, dest)
        print(f"  downloaded {key:9s} {dest.stat().st_size/1e6:8.1f} MB  ({time.time()-t0:.0f}s)")
    else:
        _curl(KAGGLE_URL.format(ref=ref), dest)
        print(f"  downloaded {key:9s} {dest.stat().st_size/1e6:8.1f} MB  ({time.time()-t0:.0f}s)")
    return dest


print("Acquiring corpora ...")
PATHS = {}
for key in DATASETS:
    try:
        PATHS[key] = acquire(key)
    except Exception as e:
        if key in OPTIONAL:
            print(f"  [optional] {key} unavailable ({e}) — continuing without it")
        else:
            raise

with open(PATHS["icd_meta"], newline="", encoding="utf-8-sig") as fh:
    ICD_ROWS = list(csv.DictReader(fh))
print("ICD manifest:", len(ICD_ROWS), "clips")
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 4 — The audio front end: the single source of truth.
#
# The C firmware in out/dsp_crynet.c is generated from these same numbers and
# is verified against this implementation by a parity test at the end.
# ============================================================================
from numpy.lib.stride_tricks import sliding_window_view

MEL_BANK = librosa.filters.mel(sr=SR, n_fft=N_FFT, n_mels=N_MELS,
                               fmin=FMIN, fmax=FMAX, htk=False, norm="slaney"
                               ).astype(np.float32)          # (64, 257)
HANN = np.hanning(N_FFT).astype(np.float32)                 # (512,)


def log_mel(x):
    '''(32000,) float32 in [-1,1]  ->  (81, 64) float32 standardised log-Mel.'''
    x = np.asarray(x, dtype=np.float32)
    assert x.shape[0] == N_SAMPLES, x.shape
    x = x - x.mean()                                    # DC removal (INMP441 bias)
    xp = np.concatenate([np.zeros(CENTER_PAD, np.float32), x,
                         np.zeros(TAIL_PAD, np.float32)])
    frames = sliding_window_view(xp, N_FFT)[::HOP]      # (81, 512)
    spec = np.fft.rfft(frames * HANN, axis=-1)          # (81, 257) complex
    power = spec.real ** 2 + spec.imag ** 2             # (81, 257)
    mel = power @ MEL_BANK.T                            # (81, 64)
    db = 10.0 * np.log10(np.maximum(mel, LOG_EPS))
    db = np.maximum(db, DB_FLOOR)
    if CMVN:
        db = (db - db.mean()) / (db.std() + 1e-6)
    return db.astype(np.float32)


_probe = np.zeros(N_SAMPLES, np.float32)
_probe[8000:16000] = np.sin(2 * np.pi * 420 * np.arange(8000) / SR)
_m = log_mel(_probe)
print(f"log_mel shape : {_m.shape}  (expect ({N_FRAMES}, {N_MELS}))")
print(f"value range   : [{_m.min():.2f}, {_m.max():.2f}]")
print(f"mean / std    : {_m.mean():+.4f} / {_m.std():.4f}   (CMVN -> ~0 / ~1)")
print(f"mel bank      : {MEL_BANK.shape}, non-zero weights {int((MEL_BANK>0).sum())}")
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 5 — Archive readers (archives are read in place; nothing is unzipped)
# ============================================================================
class ZipReader:
    '''Thread-safe reader over a zip archive (one handle per thread).'''

```

```
    def __init__(self, path):
        self.path = str(path)
        self._local = threading.local()
        self._names = None
        self._index = None
        self._path_index = None

    def zf(self):
        z = getattr(self._local, "zf", None)
        if z is None:
            z = zipfile.ZipFile(self.path)
            self._local.zf = z
        return z

    def names(self):
        if self._names is None:
            self._names = [n for n in self.zf().namelist() if not n.endswith("/")]
        return self._names

    def read(self, member):
        return self.zf().read(member)

    def index_by_stem(self):
        if self._index is None:
            idx = {}
            for n in self.names():
                idx.setdefault(Path(n).stem.lower(), n)
            self._index = idx
        return self._index

    def index_by_path(self):
        '''Like index_by_stem, but keyed by the whole path.

        Some corpora put the same file name in many folders — GTZAN calls
        every clip 00000.wav .. 00099.wav, once per genre — so a stem
        index would silently keep only one of the ten genres.
        '''
        if self._path_index is None:
            idx = {}
            for n in self.names():
                idx.setdefault(str(Path(n).with_suffix("")).lower(), n)
            self._path_index = idx
        return self._path_index


def _read_au(raw):
    '''A Sun/NeXT .au, read without libsndfile.

    GTZAN ships as .au, and libsndfile's AU support is not present in
    every build — on Kaggle it decoded nothing, and a corpus that
    decodes to nothing is a corpus that is silently missing from the
    training set.  The format is simple enough to read directly:

        magic '.snd' | data offset | data size | encoding | rate | channels

    Encodings: 1 mu-law, 2 8-bit, 3 16-bit, 4 24-bit, 5 32-bit,
    6 float32, 7 float64.
    '''
    import struct
    if len(raw) < 28 or raw[:4] != b".snd":
        return None, None
    offset, size, enc, rate, ch = struct.unpack(">5I", raw[8:28])
    data = raw[offset:offset + size] if size else raw[offset:]
    if enc == 1:                                    # 8-bit mu-law
        b = np.frombuffer(data, dtype=np.uint8).astype(np.int32)
        u = np.bitwise_not(b) & 0xFF
        u = (u & 0x0F) << 3
        u = u + 0x84
        u = u << (((~u) & 0x70) >> 4)
        u = u - 0x84
        v = u.astype(np.float32) / 32768.0
        x = np.where((b & 0x80) != 0, -v, v)
    elif enc == 2:                                  # 8-bit linear
        b = np.frombuffer(data, dtype=np.uint8).astype(np.float32)
        x = (b - 128.0) / 128.0
    elif enc == 3:                                  # 16-bit linear
        x = np.frombuffer(data, dtype=">i2").astype(np.float32) / 32768.0
    elif enc == 4:                                  # 24-bit linear
        b = np.frombuffer(data, dtype=np.uint8).reshape(-1, 3).astype(np.int32)
        v = b[:, 0] | (b[:, 1] << 8) | (b[:, 2] << 16)
        v = np.where(v & 0x800000, v - (1 << 24), v)
        x = v.astype(np.float32) / 8388608.0
    elif enc == 5:                                  # 32-bit linear
        x = np.frombuffer(data, dtype=">i4").astype(np.float32) / 2147483648.0
    elif enc == 6:                                  # 32-bit float
```

```
        x = np.frombuffer(data, dtype=">f4").astype(np.float32)
    elif enc == 7:                                  # 64-bit float
        x = np.frombuffer(data, dtype=">f8").astype(np.float32)
    else:
        return None, None
    if ch > 1:
        x = x.reshape(-1, ch).mean(axis=1)
    return np.ascontiguousarray(x, np.float32), float(rate)


def decode(raw_bytes, target_sr=SR):
    '''bytes -> float32 mono at target_sr (None on unreadable data).'''
    try:
        x, sr = sf.read(io.BytesIO(raw_bytes), dtype="float32", always_2d=False)
    except Exception:
        # libsndfile's AU support is not in every build; read it here.
        x, sr = _read_au(raw_bytes)
        if x is None:
            return None
    if x.ndim > 1:
        x = x.mean(axis=1)
    if sr != target_sr:
        x = librosa.resample(x, orig_sr=sr, target_sr=target_sr, res_type="soxr_hq")
    x = np.asarray(x, dtype=np.float32)
    if x.size < SR // 2:
        return None
    return x


def take_windows(x, n, rng=None, mode="random"):
    '''Slice a clip into n windows of exactly N_SAMPLES.'''
    if len(x) <= N_SAMPLES:
        w = np.zeros(N_SAMPLES, np.float32)
        w[: len(x)] = x
        return [w]
    max_start = len(x) - N_SAMPLES
    if mode == "center":
        starts = [max_start // 2] if n == 1 else \
                 list(np.linspace(0, max_start, n).astype(int))
    else:
        if rng is None:
            starts = [0]
        else:
            starts = list(rng.integers(0, max_start + 1, size=n))
    return [np.array(x[s:s + N_SAMPLES], dtype=np.float32) for s in starts]


print("ZipReader ready.")
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 6 — Corpus manifests
#
#   ICD        : positive/negative, subclass, fold  (from the shipped manifest)
#   cry4       : every clip is a real infant cry            -> label 1
#   ESC-50     : crying_baby -> label 1, all other categories -> label 0
#   Snoring    : adult vocalisations                         -> label 0
#   GTZAN      : music, ten genres                           -> label 0
# ============================================================================
READERS = {
    "icd_pos": ZipReader(PATHS["icd_pos"]),
    "icd_neg": ZipReader(PATHS["icd_neg"]),
    "cry4":    ZipReader(PATHS["cry4"]),
    "esc50":   ZipReader(PATHS["esc50"]),
    "snore":   ZipReader(PATHS["snore"]),
    "gtzan":   ZipReader(PATHS["gtzan"]),
}
if "speech" in PATHS:
    READERS["speech"] = ZipReader(PATHS["speech"])

# --- ICD ---------------------------------------------------------------------
ICD_POS_IDX = READERS["icd_pos"].index_by_stem()
ICD_NEG_IDX = READERS["icd_neg"].index_by_stem()

icd_entries = []           # (stem, label, subclass, fold, reader_key)
for r in ICD_ROWS:
    stem = Path(r["wav"]).stem.lower()
    if r["class_string"] == "positive":
        if stem in ICD_POS_IDX:
            icd_entries.append((stem, 1, r.get("subclass", "cry"), r["fold"], "icd_pos"))
    else:
        if stem in ICD_NEG_IDX:
            icd_entries.append((stem, 0, r.get("subclass", "noncry"), r["fold"], "icd_neg"))
```

```
# --- cry4 ---------------------------------------------------------------------
cry4_entries = []
for member in READERS["cry4"].names():
    if not member.lower().endswith(".wav"):
        continue
    parts = Path(member).parts
    cls = parts[-2].lower() if len(parts) >= 2 else "cry4"
    cry4_entries.append((Path(member).stem.lower(), 1, f"cry4_{cls}", "x", "cry4"))

# --- ESC-50 -------------------------------------------------------------------
ESC50_META = {}
for member in READERS["esc50"].names():
    if member.lower().endswith("esc50.csv"):
        for row in csv.DictReader(io.StringIO(READERS["esc50"].read(member).decode("utf-8", "ignore"))):
            ESC50_META[Path(row["filename"]).stem.lower()] = row["category"].lower()
        break

esc50_idx = READERS["esc50"].index_by_stem()
esc50_entries = []
for stem, cat in ESC50_META.items():
    if stem not in esc50_idx:
        continue
    label = 1 if cat == "crying_baby" else 0
    esc50_entries.append((stem, label, f"esc50_{cat}", "x", "esc50"))

# --- snoring ------------------------------------------------------------------
snore_idx = READERS["snore"].index_by_stem()
snore_entries = [(s, 0, "snoring", "x", "snore") for s in snore_idx]

# --- GTZAN (music) ------------------------------------------------------------
# Every clip is music and every clip is a negative.  The genre is read off
# the folder above the file (genres/blues/blues.00000.au -> blues), so the
# cross-corpus breakdown can report the model per genre.
#
# Deliberately not filtered by extension: the archive ships Sun/NeXT .au
# today and WAV in other copies, and a corpus that is skipped because of
# its file extension is a corpus that is silently missing from the
# training set.  Anything inside a genre folder is offered to the reader;
# whatever it cannot decode is counted and reported.
gtzan_entries = []
for member in READERS["gtzan"].names():
    parts = Path(member).parts
    if len(parts) >= 2:
        genre = parts[-2].lower()
    else:
        # a flat archive: the genre is the prefix of the file name
        genre = Path(member).stem.split(".")[0].lower()
    if not genre or not genre.isalpha():
        continue
    # keyed by the whole path: every genre calls its clips 00000.wav...
    gtzan_entries.append((str(Path(member).with_suffix("")).lower(), 0,
                          f"gtzan_{genre}", "x", "gtzan"))

# --- loud speech (FSD50K and Common Voice, 16 kHz) ----------------------------
# The zip's file names carry the class first: speech/Speech__fsd12345.wav.
# Every clip is a negative — a person talking is not a baby crying — and
# the class becomes the subclass, so the report can say which kind of
# speech the model refuses.
speech_entries = []
if "speech" in READERS:
    for member in READERS["speech"].names():
        if not member.lower().endswith(".wav"):
            continue
        stem = Path(member).stem
        cls, _, cid = stem.partition("__")
        if not cid:
            cls, cid = "speech", stem
        speech_entries.append((stem.lower(), 0, f"speech_{cls.lower()}", "x", "speech"))

print(f"ICD      : {len(icd_entries):>6}  (cry={sum(e[1] for e in icd_entries)})")
print(f"cry4     : {len(cry4_entries):>6}  (cry={sum(e[1] for e in cry4_entries)})")
print(f"ESC-50   : {len(esc50_entries):>6}  (cry={sum(e[1] for e in esc50_entries)})")
print(f"snoring  : {len(snore_entries):>6}")
print(f"GTZAN    : {len(gtzan_entries):>6}  (music, all negative)")
print(f"speech   : {len(speech_entries):>6}  (loud speech, all negative)"
      + ("" if speech_entries else "   [corpus not built - skipped]"))
""")

# ================================================================================
code(r"""
# ============================================================================
# CELL 7 — Augmentation resources: a real-noise bank and room impulse responses
# ============================================================================
rng_global = np.random.default_rng(SEED)
```

```
735
736
737 def _load_noise_pool():
738     '''Real non-cry recordings (from the ICD negative pool) for SNR mixing.'''
739     out = []
740     step = max(1, len(icd_entries) // (1500 if not FAST else 300))
741     cand = [e for e in icd_entries if e[1] == 0][::step][:1500 if not FAST else 300]
742     for stem, _l, _s, _f, key in cand:
743         x = decode(READERS[key].read(READERS[key].index_by_stem()[stem]))
744         if x is None:
745             continue
746         out.append(take_windows(x, 1, mode="center")[0])
747     return out
748
749
750 def _load_rirs():
751     if "icd_rir" not in PATHS:
752         return []
753     try:
754         zr = ZipReader(PATHS["icd_rir"])
755         out = []
756         for m in zr.names():
757             if not m.lower().endswith((".wav", ".flac")):
758                 continue
759             x, sr = sf.read(io.BytesIO(zr.read(m)), dtype="float32", always_2d=False)
760             if x.ndim > 1:
761                 x = x.mean(axis=1)
762             if sr != SR:
763                 x = librosa.resample(x, orig_sr=sr, target_sr=SR, res_type="soxr_hq")
764             if 0.05 < len(x) < SR and np.max(np.abs(x)) > 1e-6:
765                 out.append(np.asarray(x, np.float32))
766             if len(out) >= 60:
767                 break
768         return out
769     except Exception as e:
770         print("  RIRs unavailable:", e)
771         return []
772
773
774 _t0 = time.time()
775 NOISE_POOL = _load_noise_pool()
776 RIRS = _load_rirs()
777 print(f"noise pool : {len(NOISE_POOL)} x {N_SAMPLES} samples")
778 print(f"RIRs       : {len(RIRS)}")
779 print(f"({time.time()-_t0:.1f}s)")
780 """)
781
782 # ============================================================================================
783 code(r"""
784 # ============================================================================
785 # CELL 8 — Augmentation in the waveform domain
786 # ============================================================================
787 def augment_window(x, rng):
788     '''Gain + optional room impulse response + optional real-noise mix.'''
789     y = x.copy()
790     y = y * float(10 ** (rng.uniform(-18.0, 6.0) / 20.0))       # random gain
791     peak = float(np.max(np.abs(y)))
792     if peak > 0.98:
793         y = y * (0.98 / peak)
794
795     if RIRS and rng.random() < 0.35:                              # reverberation
796         r = RIRS[rng.integers(len(RIRS))]
797         y = sps.fftconvolve(y, r)[:N_SAMPLES]
798         if len(y) < N_SAMPLES:
799             y = np.pad(y, (0, N_SAMPLES - len(y)))
800
801     if NOISE_POOL and rng.random() < 0.5:                         # real noise mix
802         n = NOISE_POOL[rng.integers(len(NOISE_POOL))]
803         snr_db = rng.uniform(3.0, 25.0)
804         ps, pn = float(np.mean(y ** 2)), float(np.mean(n ** 2))
805         if ps > 1e-12 and pn > 1e-12:
806             a = np.sqrt(ps / (pn * 10 ** (snr_db / 10.0)))
807             y = y + a * n
808             p2 = float(np.max(np.abs(y)))
809             if p2 > 0.99:
810                 y = y * (0.99 / p2)
811     return y.astype(np.float32)
812
813
814 print("augment_window ready")
815 """)
816
817 # ============================================================================================
818 code(r"""
819 # ============================================================================
```

```
# CELL 9 — Feature extraction
#
#   build_corpus(entries, n_base, n_aug, mode) ->
#       X      (N, 81, 64) float16
#       y      (N,) int8
#       meta   list of (subclass, fold)
#       group  (N,) int64   clip id, so splits can never leak across clips
# ============================================================================
# What each corpus produced, and why anything was dropped.  Read after
# every build_corpus call.
BUILD_STATS = {}


def build_corpus(entries, n_base=2, n_aug=1, mode="random", subsample=None, tag=""):
    import zlib
    rng = np.random.default_rng(SEED + zlib.crc32(tag.encode()) % 10000)
    items = list(entries)
    if subsample and len(items) > subsample:
        sel = rng.choice(len(items), subsample, replace=False)
        items = [items[i] for i in sorted(sel)]

    Xs, ys, ms, gs, srcs = [], [], [], [], []
    counter = [0]
    # Why an item produced nothing.  A corpus that comes back empty has to
    # say so, loudly and specifically: an empty corpus is the difference
    # between a model that rejects music and one that does not, and it
    # must never pass as a quiet success.
    stats = {"items": len(items), "no_member": 0, "unreadable": 0, "windows": 0}

    def _one(item):
        stem, label, sub, fold, key = item
        reader = READERS[key]
        member = reader.index_by_stem().get(stem)
        if member is None:
            # some corpora (GTZAN) reuse the same file name in every folder
            member = reader.index_by_path().get(stem)
        if member is None:
            stats["no_member"] += 1
            return None
        try:
            x = decode(reader.read(member))
        except Exception:
            stats["unreadable"] += 1
            return None
        if x is None:
            stats["unreadable"] += 1
            return None
        gid = counter[0]
        counter[0] += 1
        feats = []
        for w in take_windows(x, n_base, rng, mode=mode):
            feats.append(log_mel(w))
        for _ in range(n_aug):
            w = take_windows(x, 1, rng, mode="random")[0]
            feats.append(log_mel(augment_window(w, rng)))
        return feats, label, sub, fold, gid, item

    with ThreadPoolExecutor(max_workers=8) as ex:
        for res in ex.map(_one, items):
            if res is None:
                continue
            feats, label, sub, fold, gid, item = res
            for f in feats:
                Xs.append(f.astype(np.float16))
                ys.append(label)
                ms.append((sub, fold))
                gs.append(gid)
                srcs.append(item)
    stats["windows"] = len(Xs)
    BUILD_STATS[tag] = stats

    if not Xs:
        return (np.zeros((0, N_FRAMES, N_MELS), np.float16), np.zeros((0,), np.int8),
                [], np.zeros((0,), np.int64), [])
    return (np.stack(Xs), np.asarray(ys, np.int8), ms, np.asarray(gs, np.int64), srcs)


SUB = None if not FAST else 2500     # per-corpus subsample in FAST mode

t0 = time.time()
X_icd, y_icd, m_icd, g_icd, src_icd = build_corpus(icd_entries, n_base=2, n_aug=0,
                                        mode="random", subsample=SUB, tag="icd")
print(f"  ICD base windows : {X_icd.shape}  ({time.time()-t0:.0f}s)")
""")

```

```
# ================================================================================
code(r"""
# ============================================================================
# CELL 10 — Supplementary corpora
# ============================================================================
t0 = time.time()
X_cry4, y_cry4, m_cry4, g_cry4, src_cry4 = build_corpus(cry4_entries, n_base=1, n_aug=0,
                                                        mode="center", subsample=SUB, tag="cry4")
X_esc50, y_esc50, m_esc50, g_esc50, src_esc50 = build_corpus(esc50_entries, n_base=1, n_aug=0,
                                                             mode="center", subsample=SUB, tag="esc50")
X_snore, y_snore, m_snore, g_snore, src_snore = build_corpus(snore_entries, n_base=1, n_aug=0,
                                                             mode="center", subsample=SUB, tag="snore")
# Music: random windows, because a thirty-second track is not one sound —
# the part that plays in the cot matters, not the average of the track.
X_gtzan, y_gtzan, m_gtzan, g_gtzan, src_gtzan = build_corpus(gtzan_entries, n_base=2, n_aug=0,
                                                             mode="random", subsample=SUB, tag="gtzan")
# Loud speech: two base windows and one augmented view per clip.  Speech is
# short and variable, so the windows are spread over the clip rather than
# taken from one moment of it.
X_speech, y_speech, m_speech, g_speech, src_speech = build_corpus(
    speech_entries, n_base=2, n_aug=1, mode="random",
    subsample=(2500 if FAST else None), tag="speech")
print(f"  cry4   {X_cry4.shape}")
print(f"  esc50  {X_esc50.shape}")
print(f"  snore  {X_snore.shape}")
print(f"  gtzan  {X_gtzan.shape}")
print(f"  speech {X_speech.shape}")
print(f"  ({time.time()-t0:.0f}s)")
for _tag in ("cry4", "esc50", "snore", "gtzan", "speech"):
    print(f"  [{_tag}] {BUILD_STATS.get(_tag)}")

if X_gtzan.shape[0] == 0:
    # Stop here rather than go on and train a model that has never heard
    # music.  That model is exactly the one that false-alarms on it, and
    # it would be deployed believing the fix had been applied.  The
    # diagnostics below say which of the three possible failures it was:
    # the archive never arrived, the archive holds nothing the reader
    # recognises, or the audio could not be decoded.
    print("")
    print("!" * 72)
    print("GTZAN PRODUCED NO WINDOWS - the model would be trained without music.")
    print("!" * 72)
    _p = PATHS.get("gtzan")
    print(f"  archive path : {_p}")
    if _p and Path(_p).exists():
        print(f"  archive size : {Path(_p).stat().st_size:,} bytes")
    try:
        _names = READERS["gtzan"].names()
        print(f"  members      : {len(_names)}")
        print(f"  first three  : {_names[:3]}")
    except Exception as _e:
        print(f"  the archive could not be opened: {_e!r}")
    print(f"  stats        : {BUILD_STATS.get('gtzan')}")
    print("  (no_member = the clip was not in the archive; unreadable = it")
    print("   could not be decoded)")
    raise SystemExit("GTZAN is empty - fix the corpus before training.")
""")

# ================================================================================
code(r"""
# ============================================================================
# CELL 11 — Augmented training views for the ICD training portion
#
# We re-read the ICD training clips and produce one extra augmented view of
# each clip (random crop + gain + noise/reverb).  Augmented views exist only
# for the training folds.
# ============================================================================
ICD_FOLD_OF = np.array([str(m[1]) for m in m_icd])
TRAIN_FOLDS = {"1", "2", "3"}
VAL_FOLDS = {"4"}
TEST_FOLDS = {"5"}

icd_train_rows = np.where(np.isin(ICD_FOLD_OF, list(TRAIN_FOLDS)))[0]
# map feature row -> source entry (entries were consumed in order)
# build_corpus recorded the exact source entry for every emitted window, so the
# augmented views can never be attached to the wrong clip.

print(f"ICD training windows: {len(icd_train_rows)}")

t0 = time.time()
X_aug, y_aug, m_aug = [], [], []
_aug_rng = np.random.default_rng(SEED + 7)


def _aug_one(row):
```

```
    gi = int(row)
    entry = src_icd[gi]
    stem, label, sub, fold, key = entry
    reader = READERS[key]
    member = reader.index_by_stem().get(stem)
    if member is None:
        return None
    try:
        x = decode(reader.read(member))
    except Exception:
        return None
    if x is None:
        return None
    w = take_windows(x, 1, _aug_rng, mode="random")[0]
    return log_mel(augment_window(w, _aug_rng)).astype(np.float16), int(label), (sub, fold)


N_AUG = 1 if FAST else 2
# One representative row per source CLIP (not per window) so every clip gets
# exactly N_AUG augmented views instead of N_AUG x (windows per clip).
_seen, icd_train_clips = set(), []
for _gi in icd_train_rows:
    _gid = int(g_icd[_gi])
    if _gid not in _seen:
        _seen.add(_gid)
        icd_train_clips.append(int(_gi))
print(f"ICD training clips: {len(icd_train_clips)}  (windows: {len(icd_train_rows)})")

with ThreadPoolExecutor(max_workers=8) as ex:
    for _ in range(N_AUG):
        for res in ex.map(_aug_one, icd_train_clips):
            if res is None:
                continue
            X_aug.append(res[0])
            y_aug.append(res[1])
            m_aug.append(res[2])

X_aug = np.stack(X_aug) if X_aug else np.zeros((0, N_FRAMES, N_MELS), np.float16)
y_aug = np.asarray(y_aug, np.int8)
print(f"  augmented views : {X_aug.shape}  ({time.time()-t0:.0f}s)")
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 12 — Assemble the split
#
#   TRAIN      ICD folds 1-3 (clean + augmented) + 70% of cry4
#              + 75% of ESC-50 non-cry + 75% of Snoring + 75% of the music
#              + 75% of the loud speech
#   VAL        ICD fold 4
#   TEST       ICD fold 5
#   CROSS      the held-out 30% of cry4, 25% of ESC-50, 25% of Snoring,
#              25% of the music and 25% of the speech, plus ALL 40 ESC-50
#              crying_baby clips  -> unseen corpora, unseen babies
#
# Memory: this machine has 16 GB of RAM and NO page file, so an allocation
# that does not fit is a hard failure, not a slowdown.  The training matrix
# is ~900 MB, and the first version of this cell built it by concatenating
# copies — four live copies of it at once — so the run died with an
# allocation error before it ever reached the training.  The split is
# therefore assembled in place: one preallocated array, filled a chunk at a
# time, and every source matrix released the moment it falls out of use.
#
# The order of the parts is unchanged, so the de-duplication sees the same
# sequence of windows and keeps exactly the same ones.
# ============================================================================
def split_by_clip(X, y, groups, frac, seed):
    '''Split whole clips (never windows) into two parts.'''
    uids = np.unique(groups)
    if len(uids) == 0:
        # An empty corpus must not become a float array: `~` on one raises
        # "ufunc 'invert' not supported", which hides the real problem
        # (the corpus was empty) behind a confusing numpy error.
        return np.zeros((0,), dtype=bool)
    rng = np.random.default_rng(seed)
    rng.shuffle(uids)
    n_test = int(round(len(uids) * frac))
    held = set(uids[:n_test].tolist())
    mask = np.array([g in held for g in groups], dtype=bool)
    return mask


def fill_rows(dst, start, src, idx, chunk=2048):
    '''Copy the selected rows of `src` into `dst` from `start`, a chunk at a time.
```

```
    Fancy-indexing a whole part would make a temporary as large as the part
    itself; chunking keeps the temporary at a few megabytes.'''
    for s in range(0, len(idx), chunk):
        piece = idx[s:s + chunk]
        dst[start + s: start + s + len(piece)] = src[piece]
    return start + len(idx)


def hash_rows(src, idx):
    '''One content hash per selected row.

    The hash is over the float16 bytes: equal float16 windows hash equal
    and unequal ones hash unequal, exactly as with float32, but half the
    bytes go through MD5.'''
    return [hashlib.md5(np.ascontiguousarray(src[i]).tobytes()).hexdigest()
            for i in idx]


def dedupe_parts(parts):
    '''Drop rows whose window is byte-identical to an earlier row.

    `parts` is a list of (array, indices); the earlier part wins, so the
    kept windows are the same ones the single-array version kept — but the
    windows are never all in memory at once.  Returns the kept indices per
    part and the number of duplicates dropped.'''
    seen, keep_per_part, dropped = set(), [], 0
    for arr, idx in parts:
        keep = []
        for pos, h in enumerate(hash_rows(arr, idx)):
            if h in seen:
                dropped += 1
                continue
            seen.add(h)
            keep.append(idx[pos])
        keep_per_part.append(np.asarray(keep, np.int64))
    return keep_per_part, dropped


def rss_mb():
    '''Resident set size in MB — the number that matters on a box with no page file.'''
    try:
        import ctypes

        class _PMC(ctypes.Structure):
            _fields_ = [("cb", ctypes.c_ulong),
                        ("PageFaultCount", ctypes.c_ulong),
                        ("PeakWorkingSetSize", ctypes.c_size_t),
                        ("WorkingSetSize", ctypes.c_size_t),
                        ("QuotaPeakPagedPoolUsage", ctypes.c_size_t),
                        ("QuotaPagedPoolUsage", ctypes.c_size_t),
                        ("QuotaPeakNonPagedPoolUsage", ctypes.c_size_t),
                        ("QuotaNonPagedPoolUsage", ctypes.c_size_t),
                        ("PagefileUsage", ctypes.c_size_t),
                        ("PeakPagefileUsage", ctypes.c_size_t)]

        c = _PMC()
        c.cb = ctypes.sizeof(_PMC)
        # The pseudo-handle from GetCurrentProcess() has every bit set, so
        # it must be passed as a pointer: without the restype, ctypes
        # truncates it to 32 bits and the call fails silently.
        k32 = ctypes.windll.kernel32
        k32.GetCurrentProcess.restype = ctypes.c_void_p
        fn = getattr(ctypes.windll.psapi, "GetProcessMemoryInfo", None)
        if fn is None:
            fn = k32.K32GetProcessMemoryInfo
        fn.restype = ctypes.c_int
        fn.argtypes = [ctypes.c_void_p, ctypes.POINTER(_PMC), ctypes.c_ulong]
        if not fn(k32.GetCurrentProcess(), ctypes.byref(c), c.cb):
            return float("nan")
        return c.WorkingSetSize / (1024.0 * 1024.0)
    except Exception:
        return float("nan")


# ---- ICD rows ---------------------------------------------------------
icd_is_train = np.isin(ICD_FOLD_OF, list(TRAIN_FOLDS))
icd_is_val = np.isin(ICD_FOLD_OF, list(VAL_FOLDS))
icd_is_test = np.isin(ICD_FOLD_OF, list(TEST_FOLDS))

# ---- supplementary splits ---------------------------------------------
cry4_hold = split_by_clip(X_cry4, y_cry4, g_cry4, 0.30, SEED + 11)
esc50_hold = split_by_clip(X_esc50, y_esc50, g_esc50, 0.25, SEED + 12)
snore_hold = split_by_clip(X_snore, y_snore, g_snore, 0.25, SEED + 13)
gtzan_hold = split_by_clip(X_gtzan, y_gtzan, g_gtzan, 0.25, SEED + 14)
```

```
# Speech: three quarters into the training set, a quarter held out so the
# cross-corpus score can say what the model does on speech it never saw.
speech_hold = split_by_clip(X_speech, y_speech, g_speech, 0.25, SEED + 15)

# every ESC-50 crying_baby clip is held out -> pure cross-domain cry test
esc50_cry = np.array([m[0] == "esc50_crying_baby" for m in m_esc50])
esc50_keep = ~esc50_hold & ~esc50_cry

# ---- assemble ----------------------------------------------------------
# Each part is (name, feature matrix, label vector, rows to use).
train_parts = [
    ("icd",    X_icd,    y_icd,    np.where(icd_is_train)[0]),
    ("aug",    X_aug,    y_aug,    np.arange(len(X_aug))),
    ("cry4",   X_cry4,   y_cry4,   np.where(~cry4_hold)[0]),
    ("esc50",  X_esc50,  y_esc50,  np.where(esc50_keep)[0]),
    ("snore",  X_snore,  y_snore,  np.where(~snore_hold)[0]),
    ("gtzan",  X_gtzan,  y_gtzan,  np.where(~gtzan_hold)[0]),
    ("speech", X_speech, y_speech, np.where(~speech_hold)[0]),
]

keep_per_part, dropped = dedupe_parts([(a, i) for _n, a, _y, i in train_parts])
n_train = int(sum(len(k) for k in keep_per_part))
y_dtype = np.result_type(*[p[2].dtype for p in train_parts])

X_train = np.empty((n_train, N_FRAMES, N_MELS), np.float16)
y_train = np.empty((n_train,), y_dtype)

_pos = 0
for (_name, _arr, _ys, _idx), _kp in zip(train_parts, keep_per_part):
    _pos = fill_rows(X_train, _pos, _arr, _kp)
    y_train[_pos - len(_kp):_pos] = _ys[_kp]

# The augmented views are used by nothing else: let them go now rather than
# at the end of the cell, so the peak stays as low as possible.
train_parts.clear()
del _arr, _ys, _idx, _kp, X_aug, y_aug

# ---- the held-out halves -----------------------------------------------
X_val = np.empty((int(icd_is_val.sum()), N_FRAMES, N_MELS), np.float16)
fill_rows(X_val, 0, X_icd, np.where(icd_is_val)[0])
y_val = y_icd[icd_is_val]

X_test = np.empty((int(icd_is_test.sum()), N_FRAMES, N_MELS), np.float16)
fill_rows(X_test, 0, X_icd, np.where(icd_is_test)[0])
y_test = y_icd[icd_is_test]

# ICD is done with: only the per-clip metadata (m_icd) is still needed later.
del X_icd, y_icd

# ---- the cross-corpus set ----------------------------------------------
cross_parts = [
    ("cry4",   X_cry4,   y_cry4,   np.where(cry4_hold)[0]),
    ("esc50",  X_esc50,  y_esc50,  np.where(esc50_hold | esc50_cry)[0]),
    ("snore",  X_snore,  y_snore,  np.where(snore_hold)[0]),
    ("gtzan",  X_gtzan,  y_gtzan,  np.where(gtzan_hold)[0]),
    ("speech", X_speech, y_speech, np.where(speech_hold)[0]),
]

n_cross = int(sum(len(i) for _n, _a, _y, i in cross_parts))
X_cross = np.empty((n_cross, N_FRAMES, N_MELS), np.float16)
y_cross = np.empty((n_cross,), y_dtype)

_pos = 0
for (_name, _arr, _ys, _idx) in cross_parts:
    _pos = fill_rows(X_cross, _pos, _arr, _idx)
    y_cross[_pos - len(_idx):_pos] = _ys[_idx]

meta_cross = ([m_cry4[i] for i in np.where(cry4_hold)[0]] +
              [m_esc50[i] for i in np.where(esc50_hold | esc50_cry)[0]] +
              [m_snore[i] for i in np.where(snore_hold)[0]] +
              [m_gtzan[i] for i in np.where(gtzan_hold)[0]] +
              [m_speech[i] for i in np.where(speech_hold)[0]])

# Release the rest of the intermediates: only the final split matrices are
# needed from here on, and they are what the tf.data pipeline holds.
import gc
cross_parts.clear()
del _arr, _ys, _idx
for _v in ("X_cry4", "X_esc50", "X_snore", "X_gtzan", "X_speech",
           "y_cry4", "y_esc50", "y_snore", "y_gtzan", "y_speech"):
    if _v in globals():
        del globals()[_v]
gc.collect()

print(f"TRAIN  {str(X_train.shape):>18}  cry={int(y_train.sum())}  "
```

```
      f"non-cry={int((y_train==0).sum())}   (dropped {dropped} duplicate windows)")
print(f"VAL    {str(X_val.shape):>18}  cry={int(y_val.sum())}")
print(f"TEST   {str(X_test.shape):>18}  cry={int(y_test.sum())}")
print(f"CROSS  {str(X_cross.shape):>18}  cry={int(y_cross.sum())}   "
      f"[cry4 + ESC-50 + snoring + music + speech — unseen corpora, unseen babies]")

cw = {c: float(len(y_train) / (2.0 * max(int((y_train == c).sum()), 1))) for c in (0, 1)}
print("class weights:", cw)
print(f"peak RSS after the split: {rss_mb():.0f} MB")
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 13 — tf.data pipeline with SpecAugment
# ============================================================================
AUTOTUNE = tf.data.AUTOTUNE


def spec_augment(xb):
    '''SpecAugment: mask 2 random frequency bands and 2 random time spans.

    Implemented with pure TF ops so the tensor shape stays static (a
    tf.py_function would erase it and Keras could not infer the batch shape).
    Masks are drawn per sample.'''
    x = xb
    B = tf.shape(x)[0]
    T, M = xb.shape[1], xb.shape[2]
    for _ in range(2):
        f0 = tf.random.uniform([B, 1, 1], minval=0, maxval=M - 8, dtype=tf.int32)
        fw = tf.random.uniform([B, 1, 1], minval=1, maxval=9, dtype=tf.int32)
        cols = tf.range(M, dtype=tf.int32)[None, None, :]
        x = tf.where((cols >= f0) & (cols < f0 + fw), tf.zeros_like(x), x)
    for _ in range(2):
        t0 = tf.random.uniform([B, 1, 1], minval=0, maxval=T - 12, dtype=tf.int32)
        tw = tf.random.uniform([B, 1, 1], minval=1, maxval=13, dtype=tf.int32)
        rows = tf.range(T, dtype=tf.int32)[None, :, None]
        x = tf.where((rows >= t0) & (rows < t0 + tw), tf.zeros_like(x), x)
    return x


def make_ds(X, y, batch_size, train):
    ds = tf.data.Dataset.from_tensor_slices((X, y))
    if train:
        # A bounded shuffle buffer: shuffling the whole corpus would copy it.
        # Each buffered element is a whole 81x64 window, so a 32k buffer
        # would hold ~340 MB on its own; 8k shuffles just as well for a
        # corpus of this size and costs a tenth of the memory.
        buf = int(min(len(X), 8192))
        ds = ds.shuffle(buf, seed=SEED, reshuffle_each_iteration=True)
    ds = ds.batch(batch_size, drop_remainder=train)
    if train:
        ds = ds.map(lambda xb, yb: (spec_augment(tf.cast(xb, tf.float32)), yb),
                    num_parallel_calls=AUTOTUNE)
        ds = ds.repeat()
    else:
        ds = ds.map(lambda xb, yb: (tf.cast(xb, tf.float32), yb))
    return ds.prefetch(AUTOTUNE)


BATCH = 128 if not FAST else 64
ds_train = make_ds(X_train, y_train, BATCH, True)
ds_val = make_ds(X_val, y_val, BATCH, False)
ds_test = make_ds(X_test, y_test, BATCH, False)
ds_cross = make_ds(X_cross, y_cross, BATCH, False)

print("tf.data ready:", X_train.shape, "->", BATCH, "x", N_FRAMES, "x", N_MELS)
print(f"RSS with the datasets built: {rss_mb():.0f} MB")
print(f"training windows: {len(X_train)}  ({int(y_train.sum())} cry / "
      f"{int((y_train == 0).sum())} non-cry)")
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 14 — Model zoo
#
# Every block is TFLite-Micro friendly: Conv2D / DepthwiseConv2D / BatchNorm /
# ReLU / MaxPool / GlobalAveragePooling / Dense.  No attention, no RNN.
# ============================================================================
from tensorflow.keras import layers, models


def make_model(kind, seed=SEED):
    tf.keras.utils.set_random_seed(seed)
```

```
    inp = layers.Input(shape=(N_FRAMES, N_MELS, 1), name="mel")
    x = inp

    if kind == "conv_cnn":
        for f in (16, 32, 64, 64, 96):
            x = layers.Conv2D(f, 3, padding="same", use_bias=False)(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.MaxPool2D((2, 2))(x)
        x = layers.GlobalAveragePooling2D()(x)
        x = layers.Dense(64, activation="relu")(x)
        x = layers.Dropout(0.3)(x)

    elif kind == "sep_cnn":
        for f in (24, 48, 96, 128):
            x = layers.DepthwiseConv2D(3, padding="same", use_bias=False)(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.Conv2D(f, 1, use_bias=False)(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.MaxPool2D((2, 2))(x)
        x = layers.GlobalAveragePooling2D()(x)
        x = layers.Dense(64, activation="relu")(x)
        x = layers.Dropout(0.3)(x)

    elif kind == "bsc_cnn":
        for f in (32, 64, 128, 128):
            x = layers.Conv2D(f, 3, padding="same", use_bias=False)(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.MaxPool2D((2, 2))(x)
        x = layers.GlobalAveragePooling2D()(x)
        x = layers.Dense(96, activation="relu")(x)
        x = layers.Dropout(0.3)(x)

    elif kind == "temporal_cnn":
        x = layers.Conv2D(64, (5, 1), padding="same", use_bias=False)(x)
        x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)
        x = layers.MaxPool2D((2, 1))(x)
        for f in (96, 128):
            x = layers.Conv2D(f, (3, 1), padding="same", use_bias=False)(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.MaxPool2D((2, 1))(x)
        x = layers.GlobalAveragePooling2D()(x)
        x = layers.Dense(64, activation="relu")(x)
        x = layers.Dropout(0.3)(x)

    else:
        raise ValueError(kind)

    out = layers.Dense(2, activation="softmax", name="probs")(x)
    m = models.Model(inp, out, name=kind)
    m.compile(optimizer=tf.keras.optimizers.Adam(1e-3),
              loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return m


# Three variants, deliberately different in shape, so the comparison is meaningful:
#   conv_cnn      plain 2-D CNN over the log-Mel plane       (reference)
#   sep_cnn       depthwise-separable — the smallest / fastest on the ESP32
#   temporal_cnn  1-D convolutions over time, Mel bins as channels (strongest
#                 for keyword-style tasks)
# Add "bsc_cnn" to the list if you want a fourth candidate.
_KINDS_DEFAULT = "conv_cnn,sep_cnn,temporal_cnn"
KINDS = [k.strip() for k in os.environ.get("CRYNET_KINDS", _KINDS_DEFAULT).split(",")
         if k.strip()]
if FAST:
    KINDS = [k for k in KINDS if k in ("conv_cnn", "temporal_cnn")] or KINDS[:1]

for k in KINDS:
    mm = make_model(k)
    print(f"{k:14s} params={mm.count_params():>9,}  in={mm.input_shape}  out={mm.output_shape}")
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 15 — Train every variant, keep the best on validation AUC
# ============================================================================
EPOCHS = int(os.environ.get("CRYNET_EPOCHS", 3 if FAST else 35))

METRICS, MODELS = {}, {}
```

```
BEST_KIND, BEST_AUC = None, -1.0


def _auc(y, p):
    from sklearn.metrics import roc_auc_score
    return float(roc_auc_score(y, p)) if len(set(y.tolist())) > 1 else float("nan")


for kind in KINDS:
    print("\n" + "=" * 72)
    print(f"TRAINING {kind}   ({EPOCHS} epochs)")
    print("=" * 72)
    tf.keras.utils.set_random_seed(SEED)
    model = make_model(kind)

    callbacks = [
        tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss", factor=0.5,
                                             patience=3, min_lr=1e-5, verbose=0),
        tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=5,
                                         restore_best_weights=True, verbose=0),
    ]
    hist = model.fit(ds_train, steps_per_epoch=max(1, len(X_train) // BATCH),
                     epochs=EPOCHS, validation_data=ds_val, callbacks=callbacks,
                     class_weight=cw, verbose=2)

    m = {
        "kind": kind,
        "params": int(model.count_params()),
        "val_auc": _auc(y_val, model.predict(X_val, verbose=0)[:, 1]),
        "test_auc": _auc(y_test, model.predict(X_test, verbose=0)[:, 1]),
        "cross_auc": _auc(y_cross, model.predict(X_cross, verbose=0)[:, 1]),
    }
    METRICS[kind] = m
    MODELS[kind] = model
    print(f"  params={m['params']:,}  val_auc={m['val_auc']:.4f}  "
          f"test_auc={m['test_auc']:.4f}  cross_auc={m['cross_auc']:.4f}")
    if m["val_auc"] == m["val_auc"] and m["val_auc"] > BEST_AUC:
        BEST_KIND, BEST_AUC = kind, m["val_auc"]

BEST_MODEL = MODELS[BEST_KIND]
print("\n" + "=" * 72)
print(f"SELECTED: {BEST_KIND}   val AUC {BEST_AUC:.4f}   "
      f"{BEST_MODEL.count_params():,} parameters")
print("=" * 72)
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 16 — Hard-example mining (second round)
#
# The first round's mistakes are the most valuable training signal there is.
# A cough or a car horn that the model rates as a cry is exactly what turns a
# baby monitor into a nuisance, and the missed cries are exactly what a
# caregiver needs to hear about.  So: score the training set, take the
# worst-scoring negatives and the best-scoring positives, duplicate them, and
# fine-tune for a few epochs at a low learning rate.
#
# Only training rows are mined, so the held-out folds stay honest.
# ============================================================================
MINED_ROUNDS = int(os.environ.get("CRYNET_MINING_ROUNDS", 1))

if MINED_ROUNDS > 0:
    print("=" * 72)
    print("HARD-EXAMPLE MINING")
    print("=" * 72)
    # Score the training set in chunks.  Fancy-indexing every negative (or
    # every positive) at once copies most of the matrix; chunking does not.
    p_train = np.empty(len(X_train), np.float32)
    CHUNK = 4096
    for _a in range(0, len(X_train), CHUNK):
        _b = min(_a + CHUNK, len(X_train))
        p_train[_a:_b] = BEST_MODEL.predict(X_train[_a:_b], verbose=0)[:, 1]

    neg_idx = np.where(y_train == 0)[0]
    pos_idx = np.where(y_train == 1)[0]
    p_neg = p_train[neg_idx]
    p_pos = p_train[pos_idx]

    n_hard = int(min(len(neg_idx), 600))
    hard_neg = neg_idx[np.argsort(-p_neg)[:n_hard]]          # most cry-like
    hard_pos = pos_idx[np.argsort(p_pos)[:n_hard]]           # least cry-like
    print(f"  hardest negatives : {len(hard_neg)}  (P(cry) up to {p_neg.max():.3f})")
    print(f"  hardest positives : {len(hard_pos)}  (P(cry) down to {p_pos.min():.3f})")

```

```
    # The hard examples are mixed in rather than copied into a new matrix:
    # a second copy of the whole training set is ~900 MB, and this machine
    # has no page file.  The weight reproduces the old behaviour, where each
    # hard row appeared twice -- 1.4% of the rows were duplicated, so 1.4%
    # of the batches were drawn from them.
    hard_idx = np.concatenate([hard_neg, hard_pos])
    X_hard = X_train[hard_idx]
    y_hard = y_train[hard_idx]
    print(f"  fine-tuning set   : {len(X_train)} windows "
          f"(+{len(hard_idx)} hard examples mixed in)")

    ds_mine = tf.data.Dataset.sample_from_datasets(
        [ds_train, make_ds(X_hard, y_hard, BATCH, True)],
        weights=[1.0, 0.014],
        stop_on_empty_dataset=False)

    BEST_MODEL.compile(optimizer=tf.keras.optimizers.Adam(3e-4),
                       loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    mine_hist = BEST_MODEL.fit(
        ds_mine,
        steps_per_epoch=max(1, len(X_train) // BATCH),
        epochs=6,
        validation_data=ds_val,
        callbacks=[tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=3,
                                                    restore_best_weights=True, verbose=0)],
        verbose=2,
    )
    del X_hard, y_hard, ds_mine, p_train
    import gc
    gc.collect()
    print(f"  after mining: val_auc = "
          f"{_auc(y_val, BEST_MODEL.predict(X_val, verbose=0)[:, 1]):.4f}")
else:
    print("hard-example mining disabled (CRYNET_MINING_ROUNDS=0)")
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 17 — Metrics, threshold calibration, per-corpus breakdown
# ============================================================================
from sklearn.metrics import (roc_auc_score, average_precision_score,
                             confusion_matrix, f1_score,
                             precision_recall_fscore_support)


def metrics_from_probs(y, p, threshold=0.5):
    yh = (p >= threshold).astype(int)
    cm = confusion_matrix(y, yh, labels=[0, 1])
    tn, fp, fn, tp = cm.ravel()
    prec, rec, f1, _ = precision_recall_fscore_support(y, yh, average="binary",
                                                       zero_division=0)
    return {
        "n": int(len(y)), "cry": int(y.sum()),
        "auc": float(roc_auc_score(y, p)) if len(set(y.tolist())) > 1 else float("nan"),
        "ap": float(average_precision_score(y, p)) if y.sum() > 0 else float("nan"),
        "acc": float((yh == y).mean()),
        "precision": float(prec), "recall": float(rec), "f1": float(f1),
        "fpr": float(fp / max(fp + tn, 1)),
        "tn": int(tn), "fp": int(fp), "fn": int(fn), "tp": int(tp),
    }


def show(m, name):
    print(f"  {name:26s} n={m['n']:<6} cry={m['cry']:<6} AUC={m['auc']:.4f} "
          f"AP={m['ap']:.4f} acc={m['acc']:.4f} F1={m['f1']:.4f} "
          f"rec={m['recall']:.4f} prec={m['precision']:.4f} FPR={m['fpr']:.4f}")
    return m


p_val = BEST_MODEL.predict(X_val, verbose=0)[:, 1]

print("Threshold sweep on the ICD validation fold")
rows = []
for t in np.arange(0.05, 0.96, 0.01):
    yh = (p_val >= t).astype(int)
    rec = float((yh[y_val == 1] == 1).mean()) if (y_val == 1).any() else 0.0
    fpr = float((yh[y_val == 0] == 1).mean()) if (y_val == 0).any() else 0.0
    rows.append((float(t), rec, fpr))
ok = [r for r in rows if r[1] >= TARGET_RECALL]
THRESHOLD = float(max(r[0] for r in ok)) if ok else 0.5
print(f"  target recall {TARGET_RECALL:.0%}  ->  THRESHOLD = {THRESHOLD:.2f}")
for t, rec, fpr in rows[::5]:
    print(f"    t={t:.2f}  recall={rec:.4f}  FPR={fpr:.4f}")

```

```
print("\nPer-corpus detail at the selected threshold")
print("-" * 104)
res_val = show(metrics_from_probs(y_val, p_val, THRESHOLD), "ICD val (fold 4)")
res_test = show(metrics_from_probs(y_test, BEST_MODEL.predict(X_test, verbose=0)[:, 1], THRESHOLD),
                "ICD test (fold 5)")
res_cross = show(metrics_from_probs(y_cross, BEST_MODEL.predict(X_cross, verbose=0)[:, 1], THRESHOLD),
                 "cross-corpus (unseen)")

print("\nCross-corpus breakdown:")
mc = [m[0] for m in meta_cross]
for grp in sorted(set(mc)):
    sel = np.array([g == grp for g in mc])
    if sel.sum() == 0:
        continue
    show(metrics_from_probs(y_cross[sel],
                            BEST_MODEL.predict(X_cross[sel], verbose=0)[:, 1], THRESHOLD),
         "  " + grp)

print("\nPer-subclass on the ICD test fold:")
mt = [m[0] for m in m_icd]
test_rows = np.where(icd_is_test)[0]
for sub in sorted(set(mt[i] for i in test_rows)):
    sel = np.array([mt[i] == sub for i in test_rows])
    if sel.sum() < 20:
        continue
    p = BEST_MODEL.predict(X_test[sel], verbose=0)[:, 1]
    yt = y_test[sel]
    rec = float((p[yt == 1] >= THRESHOLD).mean()) if (yt == 1).any() else float("nan")
    print(f"  {sub:14s} n={int(sel.sum()):<6} cry={int(yt.sum()):<6} "
          f"recall={rec:.4f} meanP={p.mean():.4f}")
""")

# ============================================================================
code(r"""
# ========================================================================
# CELL 18 — INT8 quantization + numerical parity with the float model
# ========================================================================
_repr_idx = np.random.default_rng(SEED).choice(
    len(X_train), min(512, len(X_train)), replace=False)


def representative_gen():
    # The converter wants a LIST with one array per input tensor, and the shape
    # must match the model input exactly: (batch, frames, mels, 1)
    for i in _repr_idx:
        yield [X_train[i:i + 1, :, :, None].astype(np.float32)]


converter = tf.lite.TFLiteConverter.from_keras_model(BEST_MODEL)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
tflite_int8 = converter.convert()
(OUT / "crynet_int8.tflite").write_bytes(tflite_int8)

conv_f = tf.lite.TFLiteConverter.from_keras_model(BEST_MODEL)
conv_f.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_float = conv_f.convert()
(OUT / "crynet_float.tflite").write_bytes(tflite_float)

interp = tf.lite.Interpreter(model_content=tflite_int8)
interp.allocate_tensors()
inp_d = interp.get_input_details()[0]
out_d = interp.get_output_details()[0]
IN_SCALE, IN_ZP = float(inp_d["quantization"][0]), int(inp_d["quantization"][1])
OUT_SCALE, OUT_ZP = float(out_d["quantization"][0]), int(out_d["quantization"][1])
print("input :", inp_d["shape"], inp_d["dtype"].__name__,
      f"scale={IN_SCALE:.8f} zp={IN_ZP}")
print("output:", out_d["shape"], out_d["dtype"].__name__,
      f"scale={OUT_SCALE:.8f} zp={OUT_ZP}")


# --- guardrail: every op in the model must have a TFLite Micro kernel ---------
# (TFLM registers MEAN via kernels/reduce.cc, AVERAGE_POOL_2D via pooling.cc, ...)
TFLM_KERNELS = {
    "CONV_2D", "DEPTHWISE_CONV_2D", "AVERAGE_POOL_2D", "MAX_POOL_2D",
    "MEAN", "FULLY_CONNECTED", "SOFTMAX", "LOGISTIC", "RELU", "TANH",
    "RESHAPE", "PAD", "SQUEEZE", "LEAKY_RELU",
}
model_ops = sorted({d["op_name"] for d in interp._get_ops_details()})
# DELEGATE only appears because the host interpreter runs XNNPACK; TFLM has none
ops_for_tflm = [o for o in model_ops if o != "DELEGATE"]
missing_kernels = [o for o in ops_for_tflm if o not in TFLM_KERNELS]
```

```
print("model operators :", ops_for_tflm)
if missing_kernels:
    print(f"  WARNING — no TFLite Micro kernel for: {missing_kernels}")
    print("  The model will NOT run on the ESP32-S3 until this is fixed.")
else:
    print("  all operators have a TFLite Micro kernel")


def run_int8(interp, X):
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]
    out = np.zeros((len(X), 2), np.float32)
    for i in range(len(X)):
        q = np.clip(np.round(X[i].astype(np.float32) / s) + z, -128, 127).astype(np.int8)
        interp.set_tensor(i_d["index"], q[None, ..., None])
        interp.invoke()
        o = interp.get_tensor(o_d["index"])[0].astype(np.float32)
        out[i] = (o - o_d["quantization"][1]) * o_d["quantization"][0]
    return out


n_check = min(4000, len(X_val))
# NOTE: X_val is ordered by the ICD manifest, which lists every positive clip
# before every negative one, so the first n_check windows of the validation
# fold would be ALL cry (the fold's 2,665 positive clips alone contribute
# 5,330 windows).  A prefix is therefore not a sample: on the full run it
# produced a single-class slice, whose AUC is undefined, and the parity
# numbers were reported as NaN.  Draw a seeded random subset instead, so
# the slice always contains both classes.
_parity_rng = np.random.default_rng(SEED + 21)
_sel = _parity_rng.permutation(len(X_val))[:n_check]
p_int8 = run_int8(interp, X_val[_sel])[:, 1]
p_float = BEST_MODEL.predict(X_val[_sel], verbose=0)[:, 1]
agree = float(((p_int8 >= THRESHOLD) == (p_float >= THRESHOLD)).mean())
print(f"\nparity on {n_check} validation windows (random subset, "
      f"{int(y_val[_sel].sum())} cry):")
print(f"  max |delta p|        = {np.abs(p_int8 - p_float).max():.4f}")
print(f"  mean |delta p|       = {np.abs(p_int8 - p_float).mean():.5f}")
print(f"  decision agreement   = {agree:.4%}")

res_int8 = show(metrics_from_probs(y_val[_sel], p_int8, THRESHOLD),
                "INT8 (as deployed) val")
res_f32 = show(metrics_from_probs(y_val[_sel], p_float, THRESHOLD),
               "float32 reference val")
if np.isnan(res_int8["auc"]) or np.isnan(res_f32["auc"]):
    print("  WARNING — the parity slice contains a single class; its AUC is "
          "undefined. Draw the subset with _parity_rng.permutation instead of "
          "taking a prefix of the manifest order.")
print(f"\n  model size : int8 {len(tflite_int8)/1024:.1f} KB | "
      f"float32 {len(tflite_float)/1024:.1f} KB")
print(f"  AUC delta  : {res_int8['auc'] - res_f32['auc']:+.5f}")
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 19 — Export the deployment artifacts
# ============================================================================
BEST_MODEL.save(OUT / "crynet.keras")
print("saved the trained Keras model -> crynet.keras")


def to_c_array(data: bytes, name: str) -> str:
    out = [f"// {len(data)} bytes — generated by the CryNet notebook",
           f"const unsigned char {name}[] = {{"]
    for i in range(0, len(data), 16):
        out.append("  " + ", ".join(f"0x{b:02x}" for b in data[i:i + 16]) + ",")
    out.append("};")
    out.append(f"const unsigned int {name}_len = {len(data)};")
    return "\n".join(out)


(OUT / "model_data.h").write_text(
    "#ifndef CRYNET_MODEL_DATA_H\n#define CRYNET_MODEL_DATA_H\n"
    "#include <stdint.h>\n\n"
    "#ifdef __cplusplus\nextern \"C\" {\n#endif\n\n"
    + to_c_array(tflite_int8, "crynet_model")
    + "\n\n#ifdef __cplusplus\n}\n#endif\n#endif\n", encoding="utf-8")

# --- sparse Mel filterbank + Hann window -------------------------------------
mel_rows = []
for m in range(N_MELS):
    w = MEL_BANK[m]
    nz = np.nonzero(w > 0)[0]
```

```
    mel_rows.append((int(nz[0]), int(nz[-1]) + 1,
                     w[nz[0]:nz[-1] + 1].astype(np.float32)))
flat = np.concatenate([w for _, _, w in mel_rows])
offs, acc = [], 0
for _, _, w in mel_rows:
    offs.append(acc)
    acc += len(w)

L = [
    "#ifndef CRYNET_TABLES_H",
    "#define CRYNET_TABLES_H",
    "#include <stdint.h>",
    "",
    "/* Generated by the CryNet training notebook — do not edit by hand. */",
    f"#define CRYNET_SR            {SR}",
    f"#define CRYNET_N_SAMPLES     {N_SAMPLES}",
    f"#define CRYNET_N_FFT         {N_FFT}",
    f"#define CRYNET_HOP           {HOP}",
    f"#define CRYNET_N_MELS        {N_MELS}",
    f"#define CRYNET_N_FRAMES      {N_FRAMES}",
    f"#define CRYNET_CENTER_PAD    {CENTER_PAD}",
    f"#define CRYNET_TAIL_PAD      {TAIL_PAD}",
    f"#define CRYNET_DB_FLOOR      ({DB_FLOOR:.1f}f)",
    f"#define CRYNET_LOG_EPS       ({LOG_EPS:.10f}f)",
    f"#define CRYNET_CMVN          {1 if CMVN else 0}",
    f"#define CRYNET_IN_SCALE      {IN_SCALE:.9f}f",
    f"#define CRYNET_IN_ZP         {IN_ZP}",
    f"#define CRYNET_OUT_SCALE     {OUT_SCALE:.9f}f",
    f"#define CRYNET_OUT_ZP        {OUT_ZP}",
    f"#define CRYNET_THRESHOLD     {THRESHOLD:.4f}f",
    f"#define CRYNET_SILENCE_RMS   {SILENCE_RMS_GATE:.6f}f",
    "",
    "/* Sparse Mel filterbank: band m uses FFT bins",
    "   CRYNET_MEL_START[m] .. CRYNET_MEL_END[m]-1 with weights taken from",
    "   CRYNET_MEL_WEIGHTS[CRYNET_MEL_OFFSET[m] + (bin - CRYNET_MEL_START[m])]. */",
    "const int16_t CRYNET_MEL_START[CRYNET_N_MELS] = {",
    "  " + ", ".join(str(s) for s, _, _ in mel_rows),
    "};",
    "const int16_t CRYNET_MEL_END[CRYNET_N_MELS] = {",
    "  " + ", ".join(str(e) for _, e, _ in mel_rows),
    "};",
    "const int16_t CRYNET_MEL_OFFSET[CRYNET_N_MELS] = {",
    "  " + ", ".join(str(o) for o in offs),
    "};",
    f"const float CRYNET_MEL_WEIGHTS[{acc}] = {{",
]
for i in range(0, acc, 10):
    L.append("  " + ", ".join(f"{v:.8f}f" for v in flat[i:i + 10]) + ",")
L.append("};")
L.append("")
L.append(f"const float CRYNET_HANN[CRYNET_N_FFT] = {{")
for i in range(0, N_FFT, 10):
    L.append("  " + ", ".join(f"{v:.8f}f" for v in HANN[i:i + 10]) + ",")
L.append("};")
L.append("")
L.append("#endif /* CRYNET_TABLES_H */")
(OUT / "crynet_tables.h").write_text("\n".join(L), encoding="utf-8")

print(f"wrote {OUT/'model_data.h'}   ({len(tflite_int8)/1024:.1f} KB model)")
print(f"wrote {OUT/'crynet_tables.h'} ({acc} Mel weights)")
""")

# ============================================================================
code(r'''
# ============================================================================
# CELL 20 — The C reference implementation (port this to ESP-IDF)
# ============================================================================
C_SOURCE = r"""
/* ============================================================================
 * dsp_crynet.c — CryNet log-Mel front end, numerically identical to the
 *                Python implementation used for training.
 *
 *  Host parity test:
 *      gcc -O2 -DCRYNET_SELFTEST -o dsp_test dsp_crynet.c -lm
 *      ./dsp_test probe.s16le
 *
 *  On the ESP32-S3 replace fft512() with ESP-DSP:
 *      dsps_fft2r() + dsps_cmplx_mag()  (roughly 10x faster than the
 *      reference FFT below, which is kept so the file builds anywhere).
 * ============================================================================*/
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

```

```c
#include "crynet_tables.h"

typedef struct { float re, im; } cpx_t;

/* ---- iterative radix-2 FFT, N = 512, in place ---------------------------- */
static void fft512(cpx_t *a) {
    unsigned i, j, len;
    cpx_t t;
    for (i = 1, j = 0; i < 512; i++) {
        unsigned bit = 512 >> 1;
        for (; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if (i < j) { t = a[i]; a[i] = a[j]; a[j] = t; }
    }
    for (len = 2; len <= 512; len <<= 1) {
        float ang = -2.0f * 3.14159265358979f / (float)len;
        float wr = cosf(ang), wi = sinf(ang);
        for (i = 0; i < 512; i += len) {
            float cr = 1.0f, ci = 0.0f;
            for (j = 0; j < len / 2; j++) {
                cpx_t u = a[i + j];
                cpx_t v;
                v.re = a[i + j + len / 2].re * cr - a[i + j + len / 2].im * ci;
                v.im = a[i + j + len / 2].re * ci + a[i + j + len / 2].im * cr;
                a[i + j].re = u.re + v.re;  a[i + j].im = u.im + v.im;
                a[i + j + len / 2].re = u.re - v.re;
                a[i + j + len / 2].im = u.im - v.im;
                float nr = cr * wr - ci * wi;
                ci = cr * wi + ci * wr;
                cr = nr;
            }
        }
    }
}

/* ---- log-Mel front end --------------------------------------------------- */
/*  pcm     : CRYNET_N_SAMPLES int16 mono samples @ 16 kHz
 *  mel_out : CRYNET_N_FRAMES * CRYNET_N_MELS floats (standardised dB)     */
void crynet_log_mel(const int16_t *pcm, float *mel_out) {
    static float x[CRYNET_N_SAMPLES + CRYNET_CENTER_PAD + CRYNET_TAIL_PAD];
    static cpx_t spec[CRYNET_N_FFT];
    static float power[CRYNET_N_FFT / 2 + 1];
    static float mel[CRYNET_N_MELS];

    /* 1. DC removal + normalise to [-1, 1] */
    float mean = 0.0f;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) mean += (float)pcm[i];
    mean /= (float)CRYNET_N_SAMPLES;

    memset(x, 0, sizeof(float) * (CRYNET_N_SAMPLES + CRYNET_CENTER_PAD + CRYNET_TAIL_PAD));
    for (int i = 0; i < CRYNET_N_SAMPLES; i++)
        x[CRYNET_CENTER_PAD + i] = ((float)pcm[i] - mean) / 32768.0f;

    /* 2. frames -> FFT -> power -> mel -> dB */
    for (int fr = 0; fr < CRYNET_N_FRAMES; fr++) {
        const float *w = x + fr * CRYNET_HOP;
        for (int i = 0; i < CRYNET_N_FFT; i++) {
            spec[i].re = w[i] * CRYNET_HANN[i];
            spec[i].im = 0.0f;
        }
        fft512(spec);
        for (int i = 0; i <= CRYNET_N_FFT / 2; i++)
            power[i] = spec[i].re * spec[i].re + spec[i].im * spec[i].im;
        for (int m = 0; m < CRYNET_N_MELS; m++) {
            float acc = 0.0f;
            int start = CRYNET_MEL_START[m], end = CRYNET_MEL_END[m];
            const float *wts = &CRYNET_MEL_WEIGHTS[CRYNET_MEL_OFFSET[m]];
            for (int b = start; b < end; b++) acc += power[b] * wts[b - start];
            mel[m] = acc;
        }
        for (int m = 0; m < CRYNET_N_MELS; m++) {
            float p = mel[m];
            if (p < CRYNET_LOG_EPS) p = CRYNET_LOG_EPS;
            float db = 10.0f * log10f(p);
            if (db < CRYNET_DB_FLOOR) db = CRYNET_DB_FLOOR;
            mel_out[fr * CRYNET_N_MELS + m] = db;
        }
    }

#if CRYNET_CMVN
    /* 3. per-window standardisation */
    float mu = 0.0f;
    const int n = CRYNET_N_FRAMES * CRYNET_N_MELS;
    for (int i = 0; i < n; i++) mu += mel_out[i];
    mu /= (float)n;
```

```
    float var = 0.0f;
    for (int i = 0; i < n; i++) { float d = mel_out[i] - mu; var += d * d; }
    float sd = sqrtf(var / (float)n) + 1e-6f;
    for (int i = 0; i < n; i++) mel_out[i] = (mel_out[i] - mu) / sd;
#endif
}

/* ---- quantisation helpers ------------------------------------------------ */
void crynet_quantise(const float *mel, int8_t *q) {
    const int n = CRYNET_N_FRAMES * CRYNET_N_MELS;
    for (int i = 0; i < n; i++) {
        float v = nearbyintf(mel[i] / CRYNET_IN_SCALE) + (float)CRYNET_IN_ZP;
        if (v > 127.0f) v = 127.0f;
        if (v < -128.0f) v = -128.0f;
        q[i] = (int8_t)v;
    }
}

float crynet_dequantise(int8_t v) {
    return ((float)v - (float)CRYNET_OUT_ZP) * CRYNET_OUT_SCALE;
}

/* ---- silence gate -------------------------------------------------------- */
float crynet_window_rms(const int16_t *pcm) {
    float mean = 0.0f;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) mean += (float)pcm[i];
    mean /= (float)CRYNET_N_SAMPLES;
    float acc = 0.0f;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) {
        float d = ((float)pcm[i] - mean) / 32768.0f;
        acc += d * d;
    }
    return sqrtf(acc / (float)CRYNET_N_SAMPLES);
}

/* ---- host parity test ---------------------------------------------------- */
#ifdef CRYNET_SELFTEST
int main(int argc, char **argv) {
    if (argc < 2) { fprintf(stderr, "usage: %s raw_pcm_s16le.bin\n", argv[0]); return 2; }
    FILE *f = fopen(argv[1], "rb");
    if (!f) { perror("fopen"); return 2; }
    static int16_t pcm[CRYNET_N_SAMPLES];
    size_t got = fread(pcm, sizeof(int16_t), CRYNET_N_SAMPLES, f);
    fclose(f);
    if (got != CRYNET_N_SAMPLES) { fprintf(stderr, "short read\n"); return 2; }
    static float mel[CRYNET_N_FRAMES * CRYNET_N_MELS];
    crynet_log_mel(pcm, mel);
    FILE *o = fopen("mel_c.bin", "wb");
    fwrite(mel, sizeof(float), CRYNET_N_FRAMES * CRYNET_N_MELS, o);
    fclose(o);
    fprintf(stderr, "wrote mel_c.bin  rms=%.6f\n", crynet_window_rms(pcm));
    return 0;
}
#endif
"""

(OUT / "dsp_crynet.c").write_text(C_SOURCE, encoding="utf-8")
(OUT / "dsp_crynet.h").write_text(
    "#ifndef DSP_CRYNET_H\n#define DSP_CRYNET_H\n#include <stdint.h>\n"
    "#include \"crynet_tables.h\"\n"
    "#ifdef __cplusplus\nextern \"C\" {\n#endif\n"
    "void crynet_log_mel(const int16_t *pcm, float *mel_out);\n"
    "void crynet_quantise(const float *mel, int8_t *q);\n"
    "float crynet_dequantise(int8_t v);\n"
    "float crynet_window_rms(const int16_t *pcm);\n"
    "#ifdef __cplusplus\n}\n#endif\n#endif\n", encoding="utf-8")

# --- TFLite Micro glue: the missing link between the DSP and the model -------
TFLM_GLUE = r"""
/* ============================================================================
 * crynet_infer_example.c — glue between the DSP front end and TFLite Micro.
 *
 * Port this into the ESP32-S3 firmware.  It assumes:
 *   - model_data.h   (the INT8 model as a C array)
 *   - crynet_tables.h (Mel filterbank, Hann window, quantisation constants)
 *   - dsp_crynet.c   (the log-Mel front end)
 *
 * ESP-IDF: add the TensorFlowLite-ESP32 component and include
 * "tensorflow/lite/micro/micro_interpreter.h".
 * ==========================================================================*/
#include <stdint.h>
#include <string.h>

#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
```

```
#include "tensorflow/lite/schema/schema_generated.h"

#include "dsp_crynet.h"
#include "model_data.h"

#ifndef CRYNET_ARENA_SIZE
#define CRYNET_ARENA_SIZE 120000   /* bytes; raise if AllocateTensors fails */
#endif

/* The ops the exported model uses: CONV_2D, DEPTHWISE_CONV_2D, MAX_POOL_2D,
 * MEAN, FULLY_CONNECTED, SOFTMAX.  AddConv2D() etc. come from
 * tensorflow/lite/micro/kernels/kernel_registrar.h. */
static tflite::MicroMutableOpResolver<6> g_resolver;
static tflite::MicroInterpreter *g_interp = nullptr;
static uint8_t *g_arena = nullptr;

bool crynet_init() {
  g_arena = (uint8_t *)malloc(CRYNET_ARENA_SIZE);
  if (g_arena == nullptr) return false;
  g_resolver.AddConv2D();
  g_resolver.AddDepthwiseConv2D();
  g_resolver.AddMaxPool2D();
  g_resolver.AddMean();            /* GlobalAveragePooling2D lowers to MEAN */
  g_resolver.AddFullyConnected();
  g_resolver.AddSoftmax();

  const tflite::Model *model = tflite::GetModel(crynet_model);
  static tflite::MicroInterpreter interpreter(model, g_resolver, g_arena,
                                              CRYNET_ARENA_SIZE);
  if (interpreter.AllocateTensors() != kTfLiteOk) return false;
  g_interp = &interpreter;
  return true;
}

/* pcm: CRYNET_N_SAMPLES int16 samples @ 16 kHz mono.
 * Returns P(cry) in [0, 1]; 0.0 if the window is silent or inference failed. */
float crynet_probability(const int16_t *pcm) {
  if (g_interp == nullptr) return 0.0f;
  if (crynet_window_rms(pcm) < CRYNET_SILENCE_RMS) return 0.0f;   /* silence gate */

  static float mel[CRYNET_N_FRAMES * CRYNET_N_MELS];
  static int8_t quant[CRYNET_N_FRAMES * CRYNET_N_MELS];
  crynet_log_mel(pcm, mel);
  crynet_quantise(mel, quant);

  TfLiteTensor *in = g_interp->input(0);
  memcpy(in->data.raw, quant, sizeof(quant));
  in->bytes = sizeof(quant);
  if (g_interp->Invoke() != kTfLiteOk) return 0.0f;

  TfLiteTensor *out = g_interp->output(0);
  const int8_t *o = out->data.int8;
  /* output order is [P(no_cry), P(cry)] */
  return (float)(o[1] - out->params.zero_point) * out->params.scale;
}

/* Call once per 2 s window; returns true when the alarm should fire. */
bool crynet_is_cry(const int16_t *pcm, float *confidence) {
  const float p = crynet_probability(pcm);
  if (confidence) *confidence = p;
  return p >= CRYNET_THRESHOLD;
}
"""
(OUT / "crynet_infer_example.c").write_text(TFLM_GLUE, encoding="utf-8")

contract = {
    "model": "crynet_int8.tflite",
    "model_sha256": hashlib.sha256(tflite_int8).hexdigest(),
    "model_bytes": len(tflite_int8),
    "input": {
        "name": "mel", "shape": [1, N_FRAMES, N_MELS, 1], "dtype": "int8",
        "scale": IN_SCALE, "zero_point": IN_ZP,
        "preprocessing": [
            f"int16 PCM, {SR} Hz mono, {N_SAMPLES} samples ({WIN_SEC}s)",
            "subtract mean (DC removal)",
            f"zero-pad {CENTER_PAD} front / {TAIL_PAD} rear",
            f"{N_FRAMES} frames, {N_FFT}-pt Hann, hop {HOP}",
            f"{N_MELS} Mel bands (librosa slaney, {FMIN}-{FMAX} Hz)",
            f"10*log10(max(P, {LOG_EPS})) floored at {DB_FLOOR} dB",
            "per-window standardisation" if CMVN else "no CMVN",
            "q = round(v/scale) + zero_point",
        ],
    },
    "output": {"shape": [1, 2], "dtype": "int8", "scale": OUT_SCALE,
               "zero_point": OUT_ZP, "classes": ["no_cry", "cry"]},
```

```
    "operators": ops_for_tflm,
    "tflm_kernel_check": "ok" if not missing_kernels else missing_kernels,
    "decision": {
        "threshold": THRESHOLD,
        "silence_rms_gate": SILENCE_RMS_GATE,
        "rule": "P(cry) >= threshold  ->  ALERT",
    },
    "tflite_micro": {
        "arena_bytes_hint": 120000,
        "note": "put the tensor arena in PSRAM if it does not fit in internal SRAM",
    },
    "metrics": {
        "selected_model": BEST_KIND,
        "parameters": int(BEST_MODEL.count_params()),
        "val_auc": res_val["auc"], "test_auc": res_test["auc"], "cross_auc": res_cross["auc"],
        "val_f1": res_val["f1"], "test_f1": res_test["f1"], "cross_f1": res_cross["f1"],
        "val_recall": res_val["recall"], "val_precision": res_val["precision"],
        "int8_val_auc": res_int8["auc"], "float_val_auc": res_f32["auc"],
    },
}
(OUT / "deploy_config.json").write_text(json.dumps(contract, indent=2), encoding="utf-8")
(OUT / "metrics.json").write_text(json.dumps(
    {"selected": BEST_KIND, "threshold": THRESHOLD,
     "variants": {k: v for k, v in METRICS.items()}}, indent=2), encoding="utf-8")

print("artifacts written to", OUT)
for f in sorted(OUT.iterdir()):
    print(f"   {f.name:26s} {f.stat().st_size/1024:9.1f} KB")
''')

# ============================================================================================
code(r"""
# ============================================================================
# CELL 21 — C <-> Python numerical parity test (skipped when no compiler)
# ============================================================================
gcc = shutil.which("gcc") or shutil.which("cc") or shutil.which("clang")
CC_CMD = [gcc] if gcc else None
if CC_CMD is None:
    try:                                    # pip install ziglang -> a full C compiler
        import ziglang
        CC_CMD = [sys.executable, "-m", "ziglang", "cc"]
    except Exception:
        CC_CMD = None
parity = None

if CC_CMD is None:
    print("no C compiler found — skipping the parity test.")
    print("Compile it yourself:  gcc -O2 -DCRYNET_SELFTEST -o dsp_test "
          "out/dsp_crynet.c -lm")
else:
    print("compiling out/dsp_crynet.c with", " ".join(CC_CMD), "...")
    exe = WORK / ("dsp_test" + (".exe" if os.name == "nt" else ""))
    env = dict(os.environ)
    env.setdefault("ZIG_GLOBAL_CACHE_DIR", str(WORK / "zigcache"))
    r = subprocess.run(CC_CMD + ["-O2", "-DCRYNET_SELFTEST", "-I", str(OUT),
                                 "-o", str(exe), str(OUT / "dsp_crynet.c"), "-lm"],
                       capture_output=True, text=True, env=env)
    if r.returncode != 0:
        print("compile failed:\n", r.stderr[:2000])
    else:
        rs = np.random.default_rng(7)
        t = np.arange(N_SAMPLES) / SR
        pcm = (0.35 * np.sin(2 * np.pi * 420 * t)
               + 0.12 * np.sin(2 * np.pi * 1350 * t)
               + 0.02 * rs.standard_normal(N_SAMPLES)).astype(np.float32)
        pcm16 = (np.clip(pcm, -1, 1) * 32767).astype(np.int16)
        raw = WORK / "probe.s16le"
        raw.write_bytes(pcm16.tobytes())

        run = subprocess.run([str(exe), str(raw)], capture_output=True, text=True, cwd=str(WORK))
        if run.returncode != 0:
            print("parity binary failed:", run.stderr[:1000])
        else:
            mel_c = np.fromfile(WORK / "mel_c.bin", dtype=np.float32).reshape(N_FRAMES, N_MELS)
            mel_py = log_mel(pcm16.astype(np.float32) / 32768.0)
            diff = np.abs(mel_c - mel_py)
            parity = {"max_abs_diff": float(diff.max()),
                      "mean_abs_diff": float(diff.mean())}
            print(f"  max |C - Python|  = {diff.max():.3e}")
            print(f"  mean|C - Python|  = {diff.mean():.3e}")
            if diff.max() < 1e-2:
                print("  OK — the C front end matches training (float32 rounding only)")
            else:
                print("  WARNING: mismatch — do not ship this C code")
            (OUT / "c_parity.json").write_text(json.dumps(parity, indent=2), encoding="utf-8")
```

```
print(json.dumps({"parity": parity}, indent=2))
""")

# ============================================================================
code(r"""
# ============================================================================
# CELL 22 — Final report
# ============================================================================
print("=" * 78)
print("  CryNet — final report")
print("=" * 78)
print(f"  selected model     : {BEST_KIND}  ({BEST_MODEL.count_params():,} parameters)")
print(f"  int8 model size    : {len(tflite_int8)/1024:.1f} KB")
print(f"  input              : [1,{N_FRAMES},{N_MELS},1] int8  "
      f"(scale {IN_SCALE:.6f}, zp {IN_ZP})")
print(f"  output             : [1,2] int8 -> [P(no_cry), P(cry)]")
print(f"  alarm threshold    : {THRESHOLD:.2f}")
print(f"  silence RMS gate   : {SILENCE_RMS_GATE}")
print()
print(f"  ICD val  (fold 4)  : AUC {res_val['auc']:.4f}  F1 {res_val['f1']:.4f}  "
      f"recall {res_val['recall']:.4f}  FPR {res_val['fpr']:.4f}")
print(f"  ICD test (fold 5)  : AUC {res_test['auc']:.4f}  F1 {res_test['f1']:.4f}  "
      f"recall {res_test['recall']:.4f}  FPR {res_test['fpr']:.4f}")
print(f"  cross-corpus       : AUC {res_cross['auc']:.4f}  F1 {res_cross['f1']:.4f}  "
      f"recall {res_cross['recall']:.4f}  FPR {res_cross['fpr']:.4f}")
print(f"  int8 vs float32    : AUC {res_int8['auc']:.4f} vs {res_f32['auc']:.4f}  "
      f"(delta {res_int8['auc']-res_f32['auc']:+.5f})")
print()
print("  artifacts in", OUT)
for f in sorted(OUT.iterdir()):
    print(f"    {f.name:26s} {f.stat().st_size/1024:9.1f} KB")
print()
print("  next: flash model_data.h + crynet_tables.h + dsp_crynet.c into the")
print("        ESP32-S3 firmware, then wire crynet_log_mel() -> TFLite Micro.")
print("=" * 78)
""")

# ============================================================================
md(r"""
---

## Where to go next (checkpoint 2)

1. **Wire the front end into the firmware.** `out/dsp_crynet.c` is portable C99 — swap `fft512()` for ESP-DSP `dsps_fft2r()` + `dsps_cmplx_mag()` (≈10× faster) and copy `crynet_tables.h` and `model_data.h` next to it.
2. **Run TFLite Micro** with an INT8 input tensor `[1, 81, 64, 1]`; put the tensor arena in PSRAM if it does not fit in internal SRAM.
3. **Lower the trigger threshold.** The old hub fired an audio pull at RMS ≥ 0.18, so a *soft* cry never reached any classifier. Feed the model anything above ~0.03 RMS and let it reject the false triggers — rejecting false triggers is exactly what it was trained for (speech, cat meows, household noise).
4. **Record ~50 real clips from your own INMP441** (cries + room noise + speech + TV), then re-run this notebook with them as a small custom corpus. Matching your actual microphone, gain and room is the single biggest accuracy win left.
5. **Alert policy.** The existing BLE packet carries `cot_id`, `alert_type`, `confidence_pct`; the model output maps straight onto it (`alert_type = 0x01` when `P(cry) >= threshold`).
""")


# ============================================================================
code(r"""
# ============================================================================
# CELL 20 — Package the artifacts for download
#
# Everything the deployment needs is in `out/`.  On Kaggle the notebook's
# working directory is the output of the saved version, so the artifacts can
# be taken from there — but a single zip is easier to hand over than a dozen
# files, and it carries the exact bytes that were validated.
# ============================================================================
ZIP_NAME = "crynet_artifacts.zip"
ZIP_PATH = Path.cwd() / ZIP_NAME

_files = sorted(p for p in OUT.rglob("*") if p.is_file())
if not _files:
    raise SystemExit(f"nothing to package in {OUT}")

with zipfile.ZipFile(ZIP_PATH, "w", zipfile.ZIP_DEFLATED, compresslevel=1) as _z:
    for _p in _files:
        _z.write(_p, arcname=str(Path("out") / _p.relative_to(OUT)))

print("=" * 78)
print(f"packaged {len(_files)} files -> {ZIP_PATH}  ({ZIP_PATH.stat().st_size/1e6:.1f} MB)")
for _p in _files:
    print(f"    {str(_p.relative_to(OUT)):26s} {_p.stat().st_size/1024:9.1f} KB")
print()
print("Download it from the Output tab of the saved notebook version:")
```

```
print("  Kaggle: Save Version -> Save & Run All -> Output -> crynet_artifacts.zip")
print("=" * 78)
""")


# ============================================================================
def build():
    cells = []
    for kind, src in CELLS:
        lines = src.split("\n")
        if kind == "markdown":
            cell = {"cell_type": "markdown", "id": f"cell-{len(cells)}",
                    "metadata": {}, "source": lines}
        else:
            cell = {"cell_type": "code", "id": f"cell-{len(cells)}",
                    "execution_count": None, "metadata": {},
                    "outputs": [], "source": lines}
        cells.append(cell)

    nb = {
        "cells": cells,
        "metadata": {
            "kernelspec": {"display_name": "Python 3", "language": "python", "name": "python3"},
            "language_info": {"name": "python", "version": "3.11"},
        },
        "nbformat": 4,
        "nbformat_minor": 5,
    }
    OUT.parent.mkdir(parents=True, exist_ok=True)
    OUT.write_text(json.dumps(nb, indent=1), encoding="utf-8")
    print(f"wrote {OUT}  ({len(cells)} cells)")


if __name__ == "__main__":
    build()
```

notebooks/crynet_training_kaggle.py

2229 lines · 100,591 bytes

```
# ============================================================================
# CryNet - infant cry detection for ESP32-S3
#
# Generated from notebooks/crynet_training.ipynb by research/export_single_cell.py.
# Do not edit this file: edit research/build_notebook.py and regenerate.
#
# On Kaggle: attach the loud-speech corpus as an input dataset (the notebook
# finds speech__fsd50k_speech.zip under /kaggle/input by itself), set the
# accelerator to GPU T4 and the internet to On, then Run All.  The last cell
# writes crynet_artifacts.zip into the working directory; download it from
# the Output tab of the saved version.
# ============================================================================



# ----------------------------------------------------------------------------
#
# # CryNet — Infant Cry Detection for ESP32-S3
#
# **Binary cry / no-cry classifier, trained on 28.5k real recordings, exported as an INT8 TensorFlow Lite model for TFLite Micro on the ESP32-S3 Central Hub.**
#
# ---
#
# ## What this notebook does
#
# | # | Stage | Output |
# |---|-------|--------|
# | 1 | **Acquire** — ICD corpus (28,478 clips), Baby-Cry-4, ESC-50, Snoring, GTZAN music | cached archives |
# | 2 | **Frontend** — 16 kHz mono → 81×64 log-Mel (bit-reproducible with the C firmware) | `crynet_tables.h` |
# | 3 | **Build** — 2 s windows, "any cry present" labelling, real-noise + RIR augmentation | feature matrix |
# | 4 | **Train** — 3 CNN variants, pick the best on validation AUC | model |
# | 5 | **Mine** — score the training set, duplicate the worst mistakes, fine-tune | better model |
# | 6 | **Calibrate** — threshold sweep for ≥95 % recall on the cry class | alarm threshold |
# | 7 | **Quantize** — full INT8 post-training quantization + numerical parity check | `crynet_int8.tflite` |
# | 8 | **Export** — TFLite + C header + C reference DSP + TFLM glue + contract | `out/` |
#
# ---
#
# ## The data (why this is the right corpus)
#
# **ICD — `hoyyu1/infant-cry-detection` (primary).** 28,478 clips, *all* 16 kHz mono / 5 s, pre-assigned 5 balanced folds.
```

```
#
# | Class | n | Source |
# |---|---|---|
# | cry | 5,026 | EnesBabyCries |
# | cry | 4,552 | CryCeleb2023 |
# | cry | 3,752 | iFLYTEK 2020 (Xunfei) |
# | non-cry | 4,963 | VoxCeleb **speech** |
# | non-cry | 4,506 | DASEE **domestic** noise |
# | non-cry | 2,076 | human non-speech |
# | non-cry | 1,953 | **cat** meows |
# | non-cry | 1,650 | silence |
#
# Three independent cry corpora (hundreds of different infants, three recording chains) against speech, household noise, cat meows and silence — the exact confusion set that produces false alarms in a nursery.
#
# **Supplementary corpora** (cry diversity + *held-out cross-corpus* test sets):
# `karimeletriby/infant-cry-classification-4-classes` (4,871 real cries, 16 kHz, cross-dataset deduplicated), ESC-50 (1,960 non-cry + 40 real `crying_baby`), Snoring (1,000 adult-vocal hard negatives), **GTZAN (1,000 real music clips, ten genres)**, and **loud speech** (the speech classes of FSD50K at 16 kHz — `Speech`, `Child_speech_and_kid_speaking`, `Conversation`, `Chatter`, `Crowd`, `Human_voice`, laughter, singing — built into one zip by `research/loudspeech/build_speech_zip.py`).
#
# > Every corpus is split **by source clip**, never by window, so no recording can appear in both train and test.
#
# **Why music is in there.** The first false-alarm map over ESC-50 (2026-10-31) showed the model rejecting speech, household noise and human non-speech well (1.6 %) but had **never seen music at all** — no corpus in the mix contained any, and ESC-50 does not either. A nursery has a radio, a television and a mobile playing something, so music is a first-class confusion class here, not an afterthought. GTZAN's ten genres give the model real music to reject and the cross-corpus split gives a clean "music the model was not trained on" measurement afterwards.
#
# **Why loud speech is in there.** The corpus has speech in it — 4,963 VoxCeleb clips that came in with the ICD negatives — but no *loud* speech and no children's voices, and the one measured conversation window in the room scored **0.9727**, which is below the saturation the hub announces on (0.9900) but not by much. A nursery is full of people talking, and a child's voice is the closest thing to a baby's cry that a real room contains. The speech is sourced online rather than recorded: the user has nobody around to read a sentence into a microphone.
#
# ---
#
# ## The firmware contract (read this before touching the C code)
#
# The model is only as good as the front end it is fed. This notebook fixes the contract:
#
# ```
# int16_t pcm[32000]                2.000 s @ 16 kHz, mono
#    -> float x = pcm/32768.0
#    -> subtract mean (DC removal; the INMP441 sits at ~+18000 counts)
#    -> zero-pad 256 front / 512 rear
#    -> 81 frames, 512-pt Hann window, hop 400
#    -> |rfft|^2  (257 bins)
#    -> 64 Mel bands (librosa slaney mel, 20 Hz - 8 kHz)
#    -> 10*log10(max(P,1e-10)), floored at -80 dB
#    -> per-window standardisation (mean 0, std 1)
#    -> int8 quantise: q = round(v/scale) + zero_point
#    -> TFLite Micro int8 inference -> P(cry)
# ```
#
# `out/dsp_crynet.c` implements exactly this, and `out/crynet_tables.h` carries the *exact* Mel filterbank and Hann window used in training, so the firmware cannot drift from the model. A parity test (Python reference vs compiled C) runs at the end of this notebook if a C compiler is available.
#
# ---
#
# ## How to run (Kaggle)
#
# 1. **Nothing to attach.** The notebook fetches everything it needs: the cry corpora from Hugging Face and Kaggle, and the **loud-speech corpus** (6,000 clips of real speech — FSD50K's nursery classes: children's voices, laughter, chatter, singing; plus Common Voice) from its published URL. Import the notebook, press *Run All*, and it builds itself. If the speech corpus cannot be fetched the run **stops** rather than quietly training on the old mix.
# 2. Accelerator: **GPU T4**.
# 3. Internet: **On** — the notebook pulls **~5.8 GB** of audio (ICD + cry4 + ESC-50 + Snoring + GTZAN) and caches it under `./_cache`. The Hugging Face files are fetched with `huggingface_hub`, so an interrupted transfer resumes instead of starting over.
# 4. `Run All` (or *Save Version* → *Save & Run All*).
# 5. **Download `crynet_artifacts.zip` from the Output tab** of the saved version. The last cell packages everything the deployment needs: the INT8 model, the calibrated tables, the model as a C array, the metrics and the front-end sources.
#
# Expected wall-clock on Kaggle (T4): ~10 min download + ~10 min feature extraction + **~30-45 min training** for the one variant the hub runs (`conv_cnn`).
#
# Useful switches:
#
# | variable | effect |
# |---|---|
# | `CRYNET_FAST=1` | subsample the corpora and train for 3 epochs — local smoke test |
# | `CRYNET_OFFLINE=1` | never download; use whatever is already in `CRYNET_CACHE` |
# | `CRYNET_KINDS` | which variants to train, e.g. `temporal_cnn` (default: `conv_cnn` — the one the hub runs) |
# | `CRYNET_EPOCHS` | training epochs (default 25) |
```

```
# | `CRYNET_CACHE` | where the archives live (default `./_cache`) |
# | `CRYNET_OUT` | where the artifacts are written (default `./out`) |
#
# Local smoke test: `CRYNET_FAST=1 python run_notebook_local.py`.
# ------------------------------------------------------------------------

# ------------------------------------------------------------------------
# ========================================================================
# CELL 1 — Environment, configuration, contract constants
# ========================================================================
import os
import sys
import io
import csv
import json
import math
import time
import random
import hashlib
import shutil
import zipfile
import threading
import warnings
import subprocess
from pathlib import Path
from concurrent.futures import ThreadPoolExecutor

warnings.filterwarnings("ignore")
os.environ.setdefault("TF_CPP_MIN_LOG_LEVEL", "2")

import numpy as np

FAST = os.environ.get("CRYNET_FAST", "0") == "1"          # local smoke-test mode

# On Kaggle the notebook is run from the UI, where there is nowhere to set
# an environment variable.  This run trains the one variant the hub runs
# (conv_cnn); the other two were explored earlier and are not deployed.
if "CRYNET_KINDS" not in os.environ:
    os.environ["CRYNET_KINDS"] = "conv_cnn"
CACHE = Path(os.environ.get("CRYNET_CACHE", "./_cache")).resolve()
WORK = Path(os.environ.get("CRYNET_WORK", "./_work")).resolve()
OUT = Path(os.environ.get("CRYNET_OUT", "./out")).resolve()
for _d in (CACHE, WORK, OUT):
    _d.mkdir(parents=True, exist_ok=True)

SEED = 1337
random.seed(SEED)
np.random.seed(SEED)

# ------------------------------------------------------------------------
# SIGNAL CONTRACT — these numbers are mirrored into the C firmware.
# Change one and the C side is regenerated automatically, but the trained
# weights would no longer match, so treat them as frozen.
# ------------------------------------------------------------------------
SR          = 16000        # Hz
WIN_SEC     = 2.0          # seconds per inference window
N_SAMPLES   = int(SR * WIN_SEC)      # 32000
N_FFT       = 512
HOP         = 400
N_MELS      = 64
FMIN        = 20.0
FMAX        = 8000.0
CENTER_PAD  = 256          # zero pad at the front (librosa center=True behaviour)
TAIL_PAD    = 512          # zero pad at the rear
N_FRAMES    = 1 + (N_SAMPLES + CENTER_PAD + TAIL_PAD - N_FFT) // HOP   # 81
DB_FLOOR    = -80.0
LOG_EPS     = 1e-10
CMVN        = True         # per-window mean/variance normalisation

# Inference gate: below this the window is (near) silent and the model is not run.
# The ICD recordings sit around RMS 0.006 (~-44 dBFS), and a distant whimper
# can be quieter, so the gate is deliberately low.  The model itself rejects
# near-silent windows; this gate only skips work.
SILENCE_RMS_GATE = 0.0015  # ~ -56 dBFS on a 16-bit normalised signal
TARGET_RECALL = 0.95       # alarm threshold is chosen to reach this on the cry class

assert N_FRAMES == 81, N_FRAMES

print("=" * 72)
print("CryNet — infant cry detection for ESP32-S3")
print("=" * 72)
print(f"  mode          : {'FAST (local smoke test)' if FAST else 'FULL'}")
print(f"  signal        : {SR} Hz mono, {WIN_SEC:.1f} s window ({N_SAMPLES} samples)")
print(f"  log-mel       : {N_FRAMES} frames x {N_MELS} bands, n_fft={N_FFT}, hop={HOP}")
print(f"  dB floor      : {DB_FLOOR} dB   CMVN: {CMVN}")
```

```
print(f"  cache / work  : {CACHE}  |  {WORK}")
print(f"  artifacts     : {OUT}")
print("=" * 72)


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 2 — Dependencies
# ===========================================================================
def _pip(*pkgs):
    subprocess.run([sys.executable, "-m", "pip", "install", "-q", *pkgs],
                   check=False, capture_output=True)


for _p in ("librosa", "soundfile", "soxr", "scipy", "sklearn"):
    try:
        __import__(_p)
    except Exception:
        _pip(_p)

import soundfile as sf
import librosa
import scipy.signal as sps
import tensorflow as tf

print("tensorflow", tf.__version__, "| librosa", librosa.__version__)

try:
    gpus = tf.config.list_physical_devices("GPU")
    for g in gpus:
        tf.config.experimental.set_memory_growth(g, True)
    print("GPU:", gpus if gpus else "none (CPU mode — expect it to be slower)")
except Exception as e:  # pragma: no cover
    print("GPU probe failed:", e)

tf.random.set_seed(SEED)


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 3 — Dataset acquisition
#
#  HF     : hoyyu1/infant-cry-detection  (28,478 clips, 16 kHz, 5 s, 5 folds)
#           fetched with huggingface_hub (resumable + retried) — a 4.6 GB
#           transfer must not be lost to a dropped connection.
#  Kaggle : public download endpoint, no auth required:
#           https://www.kaggle.com/api/v1/datasets/download/{owner}/{slug}
#
#  A file attached to the notebook (/kaggle/input) is used as-is.
#  Everything is cached, so a re-run costs nothing.
#  Set CRYNET_OFFLINE=1 to require the archives to be present already.
# ===========================================================================
HF_REPO = "hoyyu1/infant-cry-detection"
HF_URL = "https://huggingface.co/datasets/{repo}/resolve/main/{path}"
KAGGLE_URL = "https://www.kaggle.com/api/v1/datasets/download/{ref}"
# The loud-speech corpus is published once, so that this notebook is
# self-contained: import it, press Run All, and it fetches everything it
# needs.  (It used to have to be dragged into Kaggle by hand.)
SPEECH_URL = ("https://github.com/Amazo123/CryNet/releases/download/"
              "speech-corpus-v1/speech__fsd50k_speech.zip")
SPEECH_SHA256 = "eeea29323c3d02154cb058388b6ac727a2dc33571ed13e773b8c5659423e0ac5"
OFFLINE = os.environ.get("CRYNET_OFFLINE", "0") == "1"
KAGGLE_INPUT = Path("/kaggle/input") if Path("/kaggle/input").exists() else None

DATASETS = {
    # key        : (kind, ref, member, approx size in bytes)
    "icd_meta": ("hf",     HF_REPO, "metadata/all28479.csv", 1_700_000),
    "icd_pos":  ("hf",     HF_REPO, "audio/positive.zip",    2_160_000_000),
    "icd_neg":  ("hf",     HF_REPO, "audio/negative.zip",    2_480_000_000),
    "icd_rir":  ("hf",     HF_REPO, "rir.zip",               110_000_000),
    "cry4":    ("kaggle", "karimeletriby/infant-cry-classification-4-classes", None, 495_000_000),
    "esc50":   ("kaggle", "akaliutau/esc-50", None, 617_000_000),
    "snore":   ("kaggle", "tareqkhanemu/snoring", None, 118_000_000),
    # Music.  Nothing else in the corpus contains any: the model had never
    # been trained to reject a song, a radio or a television, and a nursery
    # has all three.  GTZAN is ten genres of real recorded music.
    "gtzan":   ("kaggle", "carlthome/gtzan-genre-collection", None, 1_226_000_000),
    # Loud speech.  The corpus has speech in it — 4,963 VoxCeleb clips that
    # came in with the ICD negatives — but no *loud* speech, and no
    # children's voices.  A nursery is full of people talking, and a
    # child's voice is the closest thing to a baby's cry that a real room
    # contains.  The clips are the speech classes of FSD50K at 16 kHz,
    # built into one zip by research/loudspeech/build_speech_zip.py and
    # published as a release asset, so the notebook fetches it by URL.
    # Unlike the corpora above this one is NOT optional: without it the
```

```
    # run would train on the old mix and quietly miss the whole point,
    # so a failure here stops the run instead of hiding.
    "speech":  ("url",   SPEECH_URL, "speech__fsd50k_speech.zip", 793_000_000),
}
OPTIONAL = {"icd_rir"}


def canonical_path(key):
    kind, ref, member, _sz = DATASETS[key]
    if kind == "hf":
        return CACHE / member
    if kind == "url":
        return CACHE / member
    if kind == "local":
        return CACHE / "speech__fsd50k_speech.zip"
    return CACHE / (key + "__" + ref.replace("/", "_") + ".zip")


def _attached(basename):
    '''A copy attached to the notebook (/kaggle/input) wins over downloading.'''
    if KAGGLE_INPUT is None:
        return None
    try:
        for f in KAGGLE_INPUT.rglob(basename):
            if f.is_file():
                return f
    except Exception:
        pass
    return None


def _curl(url, dest):
    dest.parent.mkdir(parents=True, exist_ok=True)
    tmp = dest.with_suffix(dest.suffix + ".part")
    with tmp.open("wb") as fh:
        subprocess.run(["curl", "-sSL", "--max-time", "5400", "-o", str(tmp), url], check=True)
    tmp.replace(dest)


def _hf_download(member, dest):
    '''Resumable download; falls back to curl if the hub library is unavailable.'''
    try:
        from huggingface_hub import hf_hub_download
        return Path(hf_hub_download(HF_REPO, member, repo_type="dataset",
                                    local_dir=str(CACHE)))
    except Exception as e:
        print(f"  huggingface_hub unavailable ({e}); using curl")
        _curl(HF_URL.format(repo=HF_REPO, path=member), dest)
        return dest


def acquire(key):
    kind, ref, member, size = DATASETS[key]
    dest = canonical_path(key)
    if kind == "local":
        # Built on this machine by research/loudspeech/build_speech_zip.py.
        # It is not downloaded, so there is nothing to fetch: it is either
        # there or it is not, and the caller says which.
        if dest.exists():
            print(f"  local      {key:9s} {dest.stat().st_size/1e6:8.1f} MB")
            return dest
        # On Kaggle the zip is attached as an input dataset instead of
        # being built here, and it is found by name under /kaggle/input.
        att = _attached(dest.name)
        if att is not None:
            print(f"  attached   {key:9s} {att}  ({att.stat().st_size/1e6:.1f} MB)")
            return att
        raise FileNotFoundError(
            f"{key} not built at {dest}"
            + (f" and no {dest.name} is attached to this notebook"
               if KAGGLE_INPUT is not None else ""))
    if kind == "url":
        # Published once, fetched like any other corpus.  A copy attached
        # to the notebook still wins, so a run that has it locally does
        # not pay for the download.
        if dest.exists() and dest.stat().st_size >= size * 0.98:
            print(f"  cached     {key:9s} {dest.stat().st_size/1e6:8.1f} MB")
            return dest
        if OFFLINE:
            raise FileNotFoundError(f"{key} missing at {dest} (CRYNET_OFFLINE=1)")
        att = _attached(dest.name)
        if att is not None:
            print(f"  attached   {key:9s} {att}  ({att.stat().st_size/1e6:.1f} MB)")
            return att
        t0 = time.time()
```

```
        _curl(ref, dest)
        got = dest.stat().st_size
        if got < size * 0.98:
            dest.unlink(missing_ok=True)
            raise IOError(f"{key}: the download is short ({got} bytes, "
                          f"expected about {size})")
        if key == "speech":
            # A truncated or corrupted archive would show up much later as
            # a quietly smaller corpus, so it is checked here.
            h = hashlib.sha256()
            with dest.open("rb") as fh:
                for blk in iter(lambda: fh.read(1 << 20), b""):
                    h.update(blk)
            if h.hexdigest() != SPEECH_SHA256:
                dest.unlink(missing_ok=True)
                raise IOError(f"{key}: the archive is not the one that was "
                              f"published (sha256 {h.hexdigest()})")
        print(f"  downloaded {key:9s} {dest.stat().st_size/1e6:8.1f} MB  "
              f"({time.time()-t0:.0f}s)")
        return dest
    if dest.exists() and (OFFLINE or dest.stat().st_size >= size * 0.98):
        print(f"  cached     {key:9s} {dest.stat().st_size/1e6:8.1f} MB")
        return dest
    if OFFLINE:
        raise FileNotFoundError(f"{key} missing at {dest} (CRYNET_OFFLINE=1)")

    att = _attached(Path(member).name if kind == "hf" else dest.name)
    t0 = time.time()
    if att is not None:
        dest.parent.mkdir(parents=True, exist_ok=True)
        shutil.copyfile(att, dest)
        print(f"  attached   {key:9s} {dest.stat().st_size/1e6:8.1f} MB")
    elif kind == "hf":
        _hf_download(member, dest)
        print(f"  downloaded {key:9s} {dest.stat().st_size/1e6:8.1f} MB  ({time.time()-t0:.0f}s)")
    else:
        _curl(KAGGLE_URL.format(ref=ref), dest)
        print(f"  downloaded {key:9s} {dest.stat().st_size/1e6:8.1f} MB  ({time.time()-t0:.0f}s)")
    return dest


print("Acquiring corpora ...")
PATHS = {}
for key in DATASETS:
    try:
        PATHS[key] = acquire(key)
    except Exception as e:
        if key in OPTIONAL:
            print(f"  [optional] {key} unavailable ({e}) — continuing without it")
        else:
            raise

with open(PATHS["icd_meta"], newline="", encoding="utf-8-sig") as fh:
    ICD_ROWS = list(csv.DictReader(fh))
print("ICD manifest:", len(ICD_ROWS), "clips")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 4 — The audio front end: the single source of truth.
#
# The C firmware in out/dsp_crynet.c is generated from these same numbers and
# is verified against this implementation by a parity test at the end.
# ===========================================================================
from numpy.lib.stride_tricks import sliding_window_view

MEL_BANK = librosa.filters.mel(sr=SR, n_fft=N_FFT, n_mels=N_MELS,
                               fmin=FMIN, fmax=FMAX, htk=False, norm="slaney"
                               ).astype(np.float32)          # (64, 257)
HANN = np.hanning(N_FFT).astype(np.float32)                 # (512,)


def log_mel(x):
    '''(32000,) float32 in [-1,1]  ->  (81, 64) float32 standardised log-Mel.'''
    x = np.asarray(x, dtype=np.float32)
    assert x.shape[0] == N_SAMPLES, x.shape
    x = x - x.mean()                                    # DC removal (INMP441 bias)
    xp = np.concatenate([np.zeros(CENTER_PAD, np.float32), x,
                         np.zeros(TAIL_PAD, np.float32)])
    frames = sliding_window_view(xp, N_FFT)[::HOP]      # (81, 512)
    spec = np.fft.rfft(frames * HANN, axis=-1)          # (81, 257) complex
    power = spec.real ** 2 + spec.imag ** 2             # (81, 257)
    mel = power @ MEL_BANK.T                            # (81, 64)
    db = 10.0 * np.log10(np.maximum(mel, LOG_EPS))
    db = np.maximum(db, DB_FLOOR)
```

```
    if CMVN:
        db = (db - db.mean()) / (db.std() + 1e-6)
    return db.astype(np.float32)


_probe = np.zeros(N_SAMPLES, np.float32)
_probe[8000:16000] = np.sin(2 * np.pi * 420 * np.arange(8000) / SR)
_m = log_mel(_probe)
print(f"log_mel shape : {_m.shape}  (expect ({N_FRAMES}, {N_MELS}))")
print(f"value range   : [{_m.min():.2f}, {_m.max():.2f}]")
print(f"mean / std    : {_m.mean():+.4f} / {_m.std():.4f}   (CMVN -> ~0 / ~1)")
print(f"mel bank      : {MEL_BANK.shape}, non-zero weights {int((MEL_BANK>0).sum())}")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 5 — Archive readers (archives are read in place; nothing is unzipped)
# ===========================================================================
class ZipReader:
    '''Thread-safe reader over a zip archive (one handle per thread).'''

    def __init__(self, path):
        self.path = str(path)
        self._local = threading.local()
        self._names = None
        self._index = None
        self._path_index = None

    def zf(self):
        z = getattr(self._local, "zf", None)
        if z is None:
            z = zipfile.ZipFile(self.path)
            self._local.zf = z
        return z

    def names(self):
        if self._names is None:
            self._names = [n for n in self.zf().namelist() if not n.endswith("/")]
        return self._names

    def read(self, member):
        return self.zf().read(member)

    def index_by_stem(self):
        if self._index is None:
            idx = {}
            for n in self.names():
                idx.setdefault(Path(n).stem.lower(), n)
            self._index = idx
        return self._index

    def index_by_path(self):
        '''Like index_by_stem, but keyed by the whole path.

        Some corpora put the same file name in many folders — GTZAN calls
        every clip 00000.wav .. 00099.wav, once per genre — so a stem
        index would silently keep only one of the ten genres.
        '''
        if self._path_index is None:
            idx = {}
            for n in self.names():
                idx.setdefault(str(Path(n).with_suffix("")).lower(), n)
            self._path_index = idx
        return self._path_index


def _read_au(raw):
    '''A Sun/NeXT .au, read without libsndfile.

    GTZAN ships as .au, and libsndfile's AU support is not present in
    every build — on Kaggle it decoded nothing, and a corpus that
    decodes to nothing is a corpus that is silently missing from the
    training set.  The format is simple enough to read directly:

        magic '.snd' | data offset | data size | encoding | rate | channels

    Encodings: 1 mu-law, 2 8-bit, 3 16-bit, 4 24-bit, 5 32-bit,
    6 float32, 7 float64.
    '''
    import struct
    if len(raw) < 28 or raw[:4] != b".snd":
        return None, None
    offset, size, enc, rate, ch = struct.unpack(">5I", raw[8:28])
    data = raw[offset:offset + size] if size else raw[offset:]
    if enc == 1:                                    # 8-bit mu-law
```

```
        b = np.frombuffer(data, dtype=np.uint8).astype(np.int32)
        u = np.bitwise_not(b) & 0xFF
        u = (u & 0x0F) << 3
        u = u + 0x84
        u = u << (((~u) & 0x70) >> 4)
        u = u - 0x84
        v = u.astype(np.float32) / 32768.0
        x = np.where((b & 0x80) != 0, -v, v)
    elif enc == 2:                                # 8-bit linear
        b = np.frombuffer(data, dtype=np.uint8).astype(np.float32)
        x = (b - 128.0) / 128.0
    elif enc == 3:                                # 16-bit linear
        x = np.frombuffer(data, dtype=">i2").astype(np.float32) / 32768.0
    elif enc == 4:                                # 24-bit linear
        b = np.frombuffer(data, dtype=np.uint8).reshape(-1, 3).astype(np.int32)
        v = b[:, 0] | (b[:, 1] << 8) | (b[:, 2] << 16)
        v = np.where(v & 0x800000, v - (1 << 24), v)
        x = v.astype(np.float32) / 8388608.0
    elif enc == 5:                                # 32-bit linear
        x = np.frombuffer(data, dtype=">i4").astype(np.float32) / 2147483648.0
    elif enc == 6:                                # 32-bit float
        x = np.frombuffer(data, dtype=">f4").astype(np.float32)
    elif enc == 7:                                # 64-bit float
        x = np.frombuffer(data, dtype=">f8").astype(np.float32)
    else:
        return None, None
    if ch > 1:
        x = x.reshape(-1, ch).mean(axis=1)
    return np.ascontiguousarray(x, np.float32), float(rate)


def decode(raw_bytes, target_sr=SR):
    '''bytes -> float32 mono at target_sr (None on unreadable data).'''
    try:
        x, sr = sf.read(io.BytesIO(raw_bytes), dtype="float32", always_2d=False)
    except Exception:
        # libsndfile's AU support is not in every build; read it here.
        x, sr = _read_au(raw_bytes)
        if x is None:
            return None
    if x.ndim > 1:
        x = x.mean(axis=1)
    if sr != target_sr:
        x = librosa.resample(x, orig_sr=sr, target_sr=target_sr, res_type="soxr_hq")
    x = np.asarray(x, dtype=np.float32)
    if x.size < SR // 2:
        return None
    return x


def take_windows(x, n, rng=None, mode="random"):
    '''Slice a clip into n windows of exactly N_SAMPLES.'''
    if len(x) <= N_SAMPLES:
        w = np.zeros(N_SAMPLES, np.float32)
        w[: len(x)] = x
        return [w]
    max_start = len(x) - N_SAMPLES
    if mode == "center":
        starts = [max_start // 2] if n == 1 else \
                 list(np.linspace(0, max_start, n).astype(int))
    else:
        if rng is None:
            starts = [0]
        else:
            starts = list(rng.integers(0, max_start + 1, size=n))
    return [np.array(x[s:s + N_SAMPLES], dtype=np.float32) for s in starts]


print("ZipReader ready.")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 6 — Corpus manifests
#
#   ICD        : positive/negative, subclass, fold  (from the shipped manifest)
#   cry4       : every clip is a real infant cry            -> label 1
#   ESC-50     : crying_baby -> label 1, all other categories -> label 0
#   Snoring    : adult vocalisations                         -> label 0
#   GTZAN      : music, ten genres                           -> label 0
# ===========================================================================
READERS = {
    "icd_pos": ZipReader(PATHS["icd_pos"]),
    "icd_neg": ZipReader(PATHS["icd_neg"]),
    "cry4":    ZipReader(PATHS["cry4"]),
```

```
    "esc50":   ZipReader(PATHS["esc50"]),
    "snore":   ZipReader(PATHS["snore"]),
    "gtzan":   ZipReader(PATHS["gtzan"]),
}
if "speech" in PATHS:
    READERS["speech"] = ZipReader(PATHS["speech"])

# --- ICD ---------------------------------------------------------------------
ICD_POS_IDX = READERS["icd_pos"].index_by_stem()
ICD_NEG_IDX = READERS["icd_neg"].index_by_stem()

icd_entries = []           # (stem, label, subclass, fold, reader_key)
for r in ICD_ROWS:
    stem = Path(r["wav"]).stem.lower()
    if r["class_string"] == "positive":
        if stem in ICD_POS_IDX:
            icd_entries.append((stem, 1, r.get("subclass", "cry"), r["fold"], "icd_pos"))
    else:
        if stem in ICD_NEG_IDX:
            icd_entries.append((stem, 0, r.get("subclass", "noncry"), r["fold"], "icd_neg"))

# --- cry4 --------------------------------------------------------------------
cry4_entries = []
for member in READERS["cry4"].names():
    if not member.lower().endswith(".wav"):
        continue
    parts = Path(member).parts
    cls = parts[-2].lower() if len(parts) >= 2 else "cry4"
    cry4_entries.append((Path(member).stem.lower(), 1, f"cry4_{cls}", "x", "cry4"))

# --- ESC-50 ------------------------------------------------------------------
ESC50_META = {}
for member in READERS["esc50"].names():
    if member.lower().endswith("esc50.csv"):
        for row in csv.DictReader(io.StringIO(READERS["esc50"].read(member).decode("utf-8", "ignore"))):
            ESC50_META[Path(row["filename"]).stem.lower()] = row["category"].lower()
        break

esc50_idx = READERS["esc50"].index_by_stem()
esc50_entries = []
for stem, cat in ESC50_META.items():
    if stem not in esc50_idx:
        continue
    label = 1 if cat == "crying_baby" else 0
    esc50_entries.append((stem, label, f"esc50_{cat}", "x", "esc50"))

# --- snoring -----------------------------------------------------------------
snore_idx = READERS["snore"].index_by_stem()
snore_entries = [(s, 0, "snoring", "x", "snore") for s in snore_idx]

# --- GTZAN (music) -----------------------------------------------------------
# Every clip is music and every clip is a negative.  The genre is read off
# the folder above the file (genres/blues/blues.00000.au -> blues), so the
# cross-corpus breakdown can report the model per genre.
#
# Deliberately not filtered by extension: the archive ships Sun/NeXT .au
# today and WAV in other copies, and a corpus that is skipped because of
# its file extension is a corpus that is silently missing from the
# training set.  Anything inside a genre folder is offered to the reader;
# whatever it cannot decode is counted and reported.
gtzan_entries = []
for member in READERS["gtzan"].names():
    parts = Path(member).parts
    if len(parts) >= 2:
        genre = parts[-2].lower()
    else:
        # a flat archive: the genre is the prefix of the file name
        genre = Path(member).stem.split(".")[0].lower()
    if not genre or not genre.isalpha():
        continue
    # keyed by the whole path: every genre calls its clips 00000.wav...
    gtzan_entries.append((str(Path(member).with_suffix("")).lower(), 0,
                          f"gtzan_{genre}", "x", "gtzan"))

# --- loud speech (FSD50K and Common Voice, 16 kHz) ---------------------------
# The zip's file names carry the class first: speech/Speech__fsd12345.wav.
# Every clip is a negative — a person talking is not a baby crying — and
# the class becomes the subclass, so the report can say which kind of
# speech the model refuses.
speech_entries = []
if "speech" in READERS:
    for member in READERS["speech"].names():
        if not member.lower().endswith(".wav"):
            continue
        stem = Path(member).stem
```

```
        cls, _, cid = stem.partition("__")
        if not cid:
            cls, cid = "speech", stem
        speech_entries.append((stem.lower(), 0, f"speech_{cls.lower()}", "x", "speech"))

print(f"ICD      : {len(icd_entries):>6}  (cry={sum(e[1] for e in icd_entries)})")
print(f"cry4     : {len(cry4_entries):>6}  (cry={sum(e[1] for e in cry4_entries)})")
print(f"ESC-50   : {len(esc50_entries):>6}  (cry={sum(e[1] for e in esc50_entries)})")
print(f"snoring  : {len(snore_entries):>6}")
print(f"GTZAN    : {len(gtzan_entries):>6}  (music, all negative)")
print(f"speech   : {len(speech_entries):>6}  (loud speech, all negative)"
      + ("" if speech_entries else "   [corpus not built - skipped]"))


# ----------------------------------------------------------------------------
# ============================================================================
# CELL 7 — Augmentation resources: a real-noise bank and room impulse responses
# ============================================================================
rng_global = np.random.default_rng(SEED)


def _load_noise_pool():
    '''Real non-cry recordings (from the ICD negative pool) for SNR mixing.'''
    out = []
    step = max(1, len(icd_entries) // (1500 if not FAST else 300))
    cand = [e for e in icd_entries if e[1] == 0][::step][:1500 if not FAST else 300]
    for stem, _l, _s, _f, key in cand:
        x = decode(READERS[key].read(READERS[key].index_by_stem()[stem]))
        if x is None:
            continue
        out.append(take_windows(x, 1, mode="center")[0])
    return out


def _load_rirs():
    if "icd_rir" not in PATHS:
        return []
    try:
        zr = ZipReader(PATHS["icd_rir"])
        out = []
        for m in zr.names():
            if not m.lower().endswith((".wav", ".flac")):
                continue
            x, sr = sf.read(io.BytesIO(zr.read(m)), dtype="float32", always_2d=False)
            if x.ndim > 1:
                x = x.mean(axis=1)
            if sr != SR:
                x = librosa.resample(x, orig_sr=sr, target_sr=SR, res_type="soxr_hq")
            if 0.05 < len(x) < SR and np.max(np.abs(x)) > 1e-6:
                out.append(np.asarray(x, np.float32))
            if len(out) >= 60:
                break
        return out
    except Exception as e:
        print("  RIRs unavailable:", e)
        return []


_t0 = time.time()
NOISE_POOL = _load_noise_pool()
RIRS = _load_rirs()
print(f"noise pool : {len(NOISE_POOL)} x {N_SAMPLES} samples")
print(f"RIRs       : {len(RIRS)}")
print(f"({time.time()-_t0:.1f}s)")


# ----------------------------------------------------------------------------
# ============================================================================
# CELL 8 — Augmentation in the waveform domain
# ============================================================================
def augment_window(x, rng):
    '''Gain + optional room impulse response + optional real-noise mix.'''
    y = x.copy()
    y = y * float(10 ** (rng.uniform(-18.0, 6.0) / 20.0))       # random gain
    peak = float(np.max(np.abs(y)))
    if peak > 0.98:
        y = y * (0.98 / peak)

    if RIRS and rng.random() < 0.35:                              # reverberation
        r = RIRS[rng.integers(len(RIRS))]
        y = sps.fftconvolve(y, r)[:N_SAMPLES]
        if len(y) < N_SAMPLES:
            y = np.pad(y, (0, N_SAMPLES - len(y)))

    if NOISE_POOL and rng.random() < 0.5:                         # real noise mix
```

```
        n = NOISE_POOL[rng.integers(len(NOISE_POOL))]
        snr_db = rng.uniform(3.0, 25.0)
        ps, pn = float(np.mean(y ** 2)), float(np.mean(n ** 2))
        if ps > 1e-12 and pn > 1e-12:
            a = np.sqrt(ps / (pn * 10 ** (snr_db / 10.0)))
            y = y + a * n
            p2 = float(np.max(np.abs(y)))
            if p2 > 0.99:
                y = y * (0.99 / p2)
    return y.astype(np.float32)


print("augment_window ready")


# ------------------------------------------------------------------------
# ========================================================================
# CELL 9 — Feature extraction
#
#   build_corpus(entries, n_base, n_aug, mode) ->
#       X      (N, 81, 64) float16
#       y      (N,) int8
#       meta   list of (subclass, fold)
#       group  (N,) int64   clip id, so splits can never leak across clips
# ========================================================================
# What each corpus produced, and why anything was dropped.  Read after
# every build_corpus call.
BUILD_STATS = {}


def build_corpus(entries, n_base=2, n_aug=1, mode="random", subsample=None, tag=""):
    import zlib
    rng = np.random.default_rng(SEED + zlib.crc32(tag.encode()) % 10000)
    items = list(entries)
    if subsample and len(items) > subsample:
        sel = rng.choice(len(items), subsample, replace=False)
        items = [items[i] for i in sorted(sel)]

    Xs, ys, ms, gs, srcs = [], [], [], [], []
    counter = [0]
    # Why an item produced nothing.  A corpus that comes back empty has to
    # say so, loudly and specifically: an empty corpus is the difference
    # between a model that rejects music and one that does not, and it
    # must never pass as a quiet success.
    stats = {"items": len(items), "no_member": 0, "unreadable": 0, "windows": 0}

    def _one(item):
        stem, label, sub, fold, key = item
        reader = READERS[key]
        member = reader.index_by_stem().get(stem)
        if member is None:
            # some corpora (GTZAN) reuse the same file name in every folder
            member = reader.index_by_path().get(stem)
        if member is None:
            stats["no_member"] += 1
            return None
        try:
            x = decode(reader.read(member))
        except Exception:
            stats["unreadable"] += 1
            return None
        if x is None:
            stats["unreadable"] += 1
            return None
        gid = counter[0]
        counter[0] += 1
        feats = []
        for w in take_windows(x, n_base, rng, mode=mode):
            feats.append(log_mel(w))
        for _ in range(n_aug):
            w = take_windows(x, 1, rng, mode="random")[0]
            feats.append(log_mel(augment_window(w, rng)))
        return feats, label, sub, fold, gid, item

    with ThreadPoolExecutor(max_workers=8) as ex:
        for res in ex.map(_one, items):
            if res is None:
                continue
            feats, label, sub, fold, gid, item = res
            for f in feats:
                Xs.append(f.astype(np.float16))
                ys.append(label)
                ms.append((sub, fold))
                gs.append(gid)
                srcs.append(item)
```

```
    stats["windows"] = len(Xs)
    BUILD_STATS[tag] = stats

    if not Xs:
        return (np.zeros((0, N_FRAMES, N_MELS), np.float16), np.zeros((0,), np.int8),
                [], np.zeros((0,), np.int64), [])
    return (np.stack(Xs), np.asarray(ys, np.int8), ms, np.asarray(gs, np.int64), srcs)


SUB = None if not FAST else 2500     # per-corpus subsample in FAST mode

t0 = time.time()
X_icd, y_icd, m_icd, g_icd, src_icd = build_corpus(icd_entries, n_base=2, n_aug=0,
                                        mode="random", subsample=SUB, tag="icd")
print(f"  ICD base windows : {X_icd.shape}  ({time.time()-t0:.0f}s)")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 10 — Supplementary corpora
# ===========================================================================
t0 = time.time()
X_cry4, y_cry4, m_cry4, g_cry4, src_cry4 = build_corpus(cry4_entries, n_base=1, n_aug=0,
                                           mode="center", subsample=SUB, tag="cry4")
X_esc50, y_esc50, m_esc50, g_esc50, src_esc50 = build_corpus(esc50_entries, n_base=1, n_aug=0,
                                              mode="center", subsample=SUB, tag="esc50")
X_snore, y_snore, m_snore, g_snore, src_snore = build_corpus(snore_entries, n_base=1, n_aug=0,
                                              mode="center", subsample=SUB, tag="snore")
# Music: random windows, because a thirty-second track is not one sound —
# the part that plays in the cot matters, not the average of the track.
X_gtzan, y_gtzan, m_gtzan, g_gtzan, src_gtzan = build_corpus(gtzan_entries, n_base=2, n_aug=0,
                                             mode="random", subsample=SUB, tag="gtzan")
# Loud speech: two base windows and one augmented view per clip.  Speech is
# short and variable, so the windows are spread over the clip rather than
# taken from one moment of it.
X_speech, y_speech, m_speech, g_speech, src_speech = build_corpus(
    speech_entries, n_base=2, n_aug=1, mode="random",
    subsample=(2500 if FAST else None), tag="speech")
print(f"  cry4   {X_cry4.shape}")
print(f"  esc50  {X_esc50.shape}")
print(f"  snore  {X_snore.shape}")
print(f"  gtzan  {X_gtzan.shape}")
print(f"  speech {X_speech.shape}")
print(f"  ({time.time()-t0:.0f}s)")
for _tag in ("cry4", "esc50", "snore", "gtzan", "speech"):
    print(f"  [{_tag}] {BUILD_STATS.get(_tag)}")

if X_gtzan.shape[0] == 0:
    # Stop here rather than go on and train a model that has never heard
    # music.  That model is exactly the one that false-alarms on it, and
    # it would be deployed believing the fix had been applied.  The
    # diagnostics below say which of the three possible failures it was:
    # the archive never arrived, the archive holds nothing the reader
    # recognises, or the audio could not be decoded.
    print("")
    print("!" * 72)
    print("GTZAN PRODUCED NO WINDOWS - the model would be trained without music.")
    print("!" * 72)
    _p = PATHS.get("gtzan")
    print(f"  archive path : {_p}")
    if _p and Path(_p).exists():
        print(f"  archive size : {Path(_p).stat().st_size:,} bytes")
    try:
        _names = READERS["gtzan"].names()
        print(f"  members      : {len(_names)}")
        print(f"  first three  : {_names[:3]}")
    except Exception as _e:
        print(f"  the archive could not be opened: {_e!r}")
    print(f"  stats        : {BUILD_STATS.get('gtzan')}")
    print("  (no_member = the clip was not in the archive; unreadable = it")
    print("   could not be decoded)")
    raise SystemExit("GTZAN is empty - fix the corpus before training.")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 11 — Augmented training views for the ICD training portion
#
# We re-read the ICD training clips and produce one extra augmented view of
# each clip (random crop + gain + noise/reverb).  Augmented views exist only
# for the training folds.
# ===========================================================================
ICD_FOLD_OF = np.array([str(m[1]) for m in m_icd])
TRAIN_FOLDS = {"1", "2", "3"}
VAL_FOLDS = {"4"}
```

```
TEST_FOLDS = {"5"}

icd_train_rows = np.where(np.isin(ICD_FOLD_OF, list(TRAIN_FOLDS)))[0]
# map feature row -> source entry (entries were consumed in order)
# build_corpus recorded the exact source entry for every emitted window, so the
# augmented views can never be attached to the wrong clip.

print(f"ICD training windows: {len(icd_train_rows)}")

t0 = time.time()
X_aug, y_aug, m_aug = [], [], []
_aug_rng = np.random.default_rng(SEED + 7)


def _aug_one(row):
    gi = int(row)
    entry = src_icd[gi]
    stem, label, sub, fold, key = entry
    reader = READERS[key]
    member = reader.index_by_stem().get(stem)
    if member is None:
        return None
    try:
        x = decode(reader.read(member))
    except Exception:
        return None
    if x is None:
        return None
    w = take_windows(x, 1, _aug_rng, mode="random")[0]
    return log_mel(augment_window(w, _aug_rng)).astype(np.float16), int(label), (sub, fold)


N_AUG = 1 if FAST else 2
# One representative row per source CLIP (not per window) so every clip gets
# exactly N_AUG augmented views instead of N_AUG x (windows per clip).
_seen, icd_train_clips = set(), []
for _gi in icd_train_rows:
    _gid = int(g_icd[_gi])
    if _gid not in _seen:
        _seen.add(_gid)
        icd_train_clips.append(int(_gi))
print(f"ICD training clips: {len(icd_train_clips)}  (windows: {len(icd_train_rows)})")

with ThreadPoolExecutor(max_workers=8) as ex:
    for _ in range(N_AUG):
        for res in ex.map(_aug_one, icd_train_clips):
            if res is None:
                continue
            X_aug.append(res[0])
            y_aug.append(res[1])
            m_aug.append(res[2])

X_aug = np.stack(X_aug) if X_aug else np.zeros((0, N_FRAMES, N_MELS), np.float16)
y_aug = np.asarray(y_aug, np.int8)
print(f"  augmented views : {X_aug.shape}  ({time.time()-t0:.0f}s)")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 12 — Assemble the split
#
#   TRAIN      ICD folds 1-3 (clean + augmented) + 70% of cry4
#              + 75% of ESC-50 non-cry + 75% of Snoring + 75% of the music
#              + 75% of the loud speech
#   VAL        ICD fold 4
#   TEST       ICD fold 5
#   CROSS      the held-out 30% of cry4, 25% of ESC-50, 25% of Snoring,
#              25% of the music and 25% of the speech, plus ALL 40 ESC-50
#              crying_baby clips  -> unseen corpora, unseen babies
#
# Memory: this machine has 16 GB of RAM and NO page file, so an allocation
# that does not fit is a hard failure, not a slowdown.  The training matrix
# is ~900 MB, and the first version of this cell built it by concatenating
# copies — four live copies of it at once — so the run died with an
# allocation error before it ever reached the training.  The split is
# therefore assembled in place: one preallocated array, filled a chunk at a
# time, and every source matrix released the moment it falls out of use.
#
# The order of the parts is unchanged, so the de-duplication sees the same
# sequence of windows and keeps exactly the same ones.
# ===========================================================================
def split_by_clip(X, y, groups, frac, seed):
    '''Split whole clips (never windows) into two parts.'''
    uids = np.unique(groups)
    if len(uids) == 0:
```

```
        # An empty corpus must not become a float array: `~` on one raises
        # "ufunc 'invert' not supported", which hides the real problem
        # (the corpus was empty) behind a confusing numpy error.
        return np.zeros((0,), dtype=bool)
    rng = np.random.default_rng(seed)
    rng.shuffle(uids)
    n_test = int(round(len(uids) * frac))
    held = set(uids[:n_test].tolist())
    mask = np.array([g in held for g in groups], dtype=bool)
    return mask


def fill_rows(dst, start, src, idx, chunk=2048):
    '''Copy the selected rows of `src` into `dst` from `start`, a chunk at a time.

    Fancy-indexing a whole part would make a temporary as large as the part
    itself; chunking keeps the temporary at a few megabytes.'''
    for s in range(0, len(idx), chunk):
        piece = idx[s:s + chunk]
        dst[start + s: start + s + len(piece)] = src[piece]
    return start + len(idx)


def hash_rows(src, idx):
    '''One content hash per selected row.

    The hash is over the float16 bytes: equal float16 windows hash equal
    and unequal ones hash unequal, exactly as with float32, but half the
    bytes go through MD5.'''
    return [hashlib.md5(np.ascontiguousarray(src[i]).tobytes()).hexdigest()
            for i in idx]


def dedupe_parts(parts):
    '''Drop rows whose window is byte-identical to an earlier row.

    `parts` is a list of (array, indices); the earlier part wins, so the
    kept windows are the same ones the single-array version kept — but the
    windows are never all in memory at once.  Returns the kept indices per
    part and the number of duplicates dropped.'''
    seen, keep_per_part, dropped = set(), [], 0
    for arr, idx in parts:
        keep = []
        for pos, h in enumerate(hash_rows(arr, idx)):
            if h in seen:
                dropped += 1
                continue
            seen.add(h)
            keep.append(idx[pos])
        keep_per_part.append(np.asarray(keep, np.int64))
    return keep_per_part, dropped


def rss_mb():
    '''Resident set size in MB — the number that matters on a box with no page file.'''
    try:
        import ctypes

        class _PMC(ctypes.Structure):
            _fields_ = [("cb", ctypes.c_ulong),
                        ("PageFaultCount", ctypes.c_ulong),
                        ("PeakWorkingSetSize", ctypes.c_size_t),
                        ("WorkingSetSize", ctypes.c_size_t),
                        ("QuotaPeakPagedPoolUsage", ctypes.c_size_t),
                        ("QuotaPagedPoolUsage", ctypes.c_size_t),
                        ("QuotaPeakNonPagedPoolUsage", ctypes.c_size_t),
                        ("QuotaNonPagedPoolUsage", ctypes.c_size_t),
                        ("PagefileUsage", ctypes.c_size_t),
                        ("PeakPagefileUsage", ctypes.c_size_t)]

        c = _PMC()
        c.cb = ctypes.sizeof(_PMC)
        # The pseudo-handle from GetCurrentProcess() has every bit set, so
        # it must be passed as a pointer: without the restype, ctypes
        # truncates it to 32 bits and the call fails silently.
        k32 = ctypes.windll.kernel32
        k32.GetCurrentProcess.restype = ctypes.c_void_p
        fn = getattr(ctypes.windll.psapi, "GetProcessMemoryInfo", None)
        if fn is None:
            fn = k32.K32GetProcessMemoryInfo
        fn.restype = ctypes.c_int
        fn.argtypes = [ctypes.c_void_p, ctypes.POINTER(_PMC), ctypes.c_ulong]
        if not fn(k32.GetCurrentProcess(), ctypes.byref(c), c.cb):
            return float("nan")
        return c.WorkingSetSize / (1024.0 * 1024.0)
```

```
    except Exception:
        return float("nan")


# ---- ICD rows ---------------------------------------------------------
icd_is_train = np.isin(ICD_FOLD_OF, list(TRAIN_FOLDS))
icd_is_val = np.isin(ICD_FOLD_OF, list(VAL_FOLDS))
icd_is_test = np.isin(ICD_FOLD_OF, list(TEST_FOLDS))

# ---- supplementary splits ---------------------------------------------
cry4_hold = split_by_clip(X_cry4, y_cry4, g_cry4, 0.30, SEED + 11)
esc50_hold = split_by_clip(X_esc50, y_esc50, g_esc50, 0.25, SEED + 12)
snore_hold = split_by_clip(X_snore, y_snore, g_snore, 0.25, SEED + 13)
gtzan_hold = split_by_clip(X_gtzan, y_gtzan, g_gtzan, 0.25, SEED + 14)
# Speech: three quarters into the training set, a quarter held out so the
# cross-corpus score can say what the model does on speech it never saw.
speech_hold = split_by_clip(X_speech, y_speech, g_speech, 0.25, SEED + 15)

# every ESC-50 crying_baby clip is held out -> pure cross-domain cry test
esc50_cry = np.array([m[0] == "esc50_crying_baby" for m in m_esc50])
esc50_keep = ~esc50_hold & ~esc50_cry

# ---- assemble ---------------------------------------------------------
# Each part is (name, feature matrix, label vector, rows to use).
train_parts = [
    ("icd",    X_icd,    y_icd,    np.where(icd_is_train)[0]),
    ("aug",    X_aug,    y_aug,    np.arange(len(X_aug))),
    ("cry4",   X_cry4,   y_cry4,   np.where(~cry4_hold)[0]),
    ("esc50",  X_esc50,  y_esc50,  np.where(esc50_keep)[0]),
    ("snore",  X_snore,  y_snore,  np.where(~snore_hold)[0]),
    ("gtzan",  X_gtzan,  y_gtzan,  np.where(~gtzan_hold)[0]),
    ("speech", X_speech, y_speech, np.where(~speech_hold)[0]),
]

keep_per_part, dropped = dedupe_parts([(a, i) for _n, a, _y, i in train_parts])
n_train = int(sum(len(k) for k in keep_per_part))
y_dtype = np.result_type(*[p[2].dtype for p in train_parts])

X_train = np.empty((n_train, N_FRAMES, N_MELS), np.float16)
y_train = np.empty((n_train,), y_dtype)

_pos = 0
for (_name, _arr, _ys, _idx), _kp in zip(train_parts, keep_per_part):
    _pos = fill_rows(X_train, _pos, _arr, _kp)
    y_train[_pos - len(_kp):_pos] = _ys[_kp]

# The augmented views are used by nothing else: let them go now rather than
# at the end of the cell, so the peak stays as low as possible.
train_parts.clear()
del _arr, _ys, _idx, _kp, X_aug, y_aug

# ---- the held-out halves ----------------------------------------------
X_val = np.empty((int(icd_is_val.sum()), N_FRAMES, N_MELS), np.float16)
fill_rows(X_val, 0, X_icd, np.where(icd_is_val)[0])
y_val = y_icd[icd_is_val]

X_test = np.empty((int(icd_is_test.sum()), N_FRAMES, N_MELS), np.float16)
fill_rows(X_test, 0, X_icd, np.where(icd_is_test)[0])
y_test = y_icd[icd_is_test]

# ICD is done with: only the per-clip metadata (m_icd) is still needed later.
del X_icd, y_icd

# ---- the cross-corpus set ---------------------------------------------
cross_parts = [
    ("cry4",   X_cry4,   y_cry4,   np.where(cry4_hold)[0]),
    ("esc50",  X_esc50,  y_esc50,  np.where(esc50_hold | esc50_cry)[0]),
    ("snore",  X_snore,  y_snore,  np.where(snore_hold)[0]),
    ("gtzan",  X_gtzan,  y_gtzan,  np.where(gtzan_hold)[0]),
    ("speech", X_speech, y_speech, np.where(speech_hold)[0]),
]

n_cross = int(sum(len(i) for _n, _a, _y, i in cross_parts))
X_cross = np.empty((n_cross, N_FRAMES, N_MELS), np.float16)
y_cross = np.empty((n_cross,), y_dtype)

_pos = 0
for (_name, _arr, _ys, _idx) in cross_parts:
    _pos = fill_rows(X_cross, _pos, _arr, _idx)
    y_cross[_pos - len(_idx):_pos] = _ys[_idx]

meta_cross = ([m_cry4[i] for i in np.where(cry4_hold)[0]] +
              [m_esc50[i] for i in np.where(esc50_hold | esc50_cry)[0]] +
              [m_snore[i] for i in np.where(snore_hold)[0]] +
              [m_gtzan[i] for i in np.where(gtzan_hold)[0]] +
```

```
            [m_speech[i] for i in np.where(speech_hold)[0]])

# Release the rest of the intermediates: only the final split matrices are
# needed from here on, and they are what the tf.data pipeline holds.
import gc
cross_parts.clear()
del _arr, _ys, _idx
for _v in ("X_cry4", "X_esc50", "X_snore", "X_gtzan", "X_speech",
           "y_cry4", "y_esc50", "y_snore", "y_gtzan", "y_speech"):
    if _v in globals():
        del globals()[_v]
gc.collect()

print(f"TRAIN  {str(X_train.shape):>18}  cry={int(y_train.sum())}  "
      f"non-cry={int((y_train==0).sum())}   (dropped {dropped} duplicate windows)")
print(f"VAL    {str(X_val.shape):>18}  cry={int(y_val.sum())}")
print(f"TEST   {str(X_test.shape):>18}  cry={int(y_test.sum())}")
print(f"CROSS  {str(X_cross.shape):>18}  cry={int(y_cross.sum())}   "
      f"[cry4 + ESC-50 + snoring + music + speech — unseen corpora, unseen babies]")

cw = {c: float(len(y_train) / (2.0 * max(int((y_train == c).sum()), 1))) for c in (0, 1)}
print("class weights:", cw)
print(f"peak RSS after the split: {rss_mb():.0f} MB")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 13 — tf.data pipeline with SpecAugment
# ===========================================================================
AUTOTUNE = tf.data.AUTOTUNE


def spec_augment(xb):
    '''SpecAugment: mask 2 random frequency bands and 2 random time spans.

    Implemented with pure TF ops so the tensor shape stays static (a
    tf.py_function would erase it and Keras could not infer the batch shape).
    Masks are drawn per sample.'''
    x = xb
    B = tf.shape(x)[0]
    T, M = xb.shape[1], xb.shape[2]
    for _ in range(2):
        f0 = tf.random.uniform([B, 1, 1], minval=0, maxval=M - 8, dtype=tf.int32)
        fw = tf.random.uniform([B, 1, 1], minval=1, maxval=9, dtype=tf.int32)
        cols = tf.range(M, dtype=tf.int32)[None, None, :]
        x = tf.where((cols >= f0) & (cols < f0 + fw), tf.zeros_like(x), x)
    for _ in range(2):
        t0 = tf.random.uniform([B, 1, 1], minval=0, maxval=T - 12, dtype=tf.int32)
        tw = tf.random.uniform([B, 1, 1], minval=1, maxval=13, dtype=tf.int32)
        rows = tf.range(T, dtype=tf.int32)[None, :, None]
        x = tf.where((rows >= t0) & (rows < t0 + tw), tf.zeros_like(x), x)
    return x


def make_ds(X, y, batch_size, train):
    ds = tf.data.Dataset.from_tensor_slices((X, y))
    if train:
        # A bounded shuffle buffer: shuffling the whole corpus would copy it.
        # Each buffered element is a whole 81x64 window, so a 32k buffer
        # would hold ~340 MB on its own; 8k shuffles just as well for a
        # corpus of this size and costs a tenth of the memory.
        buf = int(min(len(X), 8192))
        ds = ds.shuffle(buf, seed=SEED, reshuffle_each_iteration=True)
    ds = ds.batch(batch_size, drop_remainder=train)
    if train:
        ds = ds.map(lambda xb, yb: (spec_augment(tf.cast(xb, tf.float32)), yb),
                    num_parallel_calls=AUTOTUNE)
        ds = ds.repeat()
    else:
        ds = ds.map(lambda xb, yb: (tf.cast(xb, tf.float32), yb))
    return ds.prefetch(AUTOTUNE)


BATCH = 128 if not FAST else 64
ds_train = make_ds(X_train, y_train, BATCH, True)
ds_val = make_ds(X_val, y_val, BATCH, False)
ds_test = make_ds(X_test, y_test, BATCH, False)
ds_cross = make_ds(X_cross, y_cross, BATCH, False)

print("tf.data ready:", X_train.shape, "->", BATCH, "x", N_FRAMES, "x", N_MELS)
print(f"RSS with the datasets built: {rss_mb():.0f} MB")
print(f"training windows: {len(X_train)}  ({int(y_train.sum())} cry / "
      f"{int((y_train == 0).sum())} non-cry)")
```

```
# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 14 — Model zoo
#
# Every block is TFLite-Micro friendly: Conv2D / DepthwiseConv2D / BatchNorm /
# ReLU / MaxPool / GlobalAveragePooling / Dense.  No attention, no RNN.
# ===========================================================================
from tensorflow.keras import layers, models


def make_model(kind, seed=SEED):
    tf.keras.utils.set_random_seed(seed)
    inp = layers.Input(shape=(N_FRAMES, N_MELS, 1), name="mel")
    x = inp

    if kind == "conv_cnn":
        for f in (16, 32, 64, 64, 96):
            x = layers.Conv2D(f, 3, padding="same", use_bias=False)(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.MaxPool2D((2, 2))(x)
        x = layers.GlobalAveragePooling2D()(x)
        x = layers.Dense(64, activation="relu")(x)
        x = layers.Dropout(0.3)(x)

    elif kind == "sep_cnn":
        for f in (24, 48, 96, 128):
            x = layers.DepthwiseConv2D(3, padding="same", use_bias=False)(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.Conv2D(f, 1, use_bias=False)(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.MaxPool2D((2, 2))(x)
        x = layers.GlobalAveragePooling2D()(x)
        x = layers.Dense(64, activation="relu")(x)
        x = layers.Dropout(0.3)(x)

    elif kind == "bsc_cnn":
        for f in (32, 64, 128, 128):
            x = layers.Conv2D(f, 3, padding="same", use_bias=False)(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.MaxPool2D((2, 2))(x)
        x = layers.GlobalAveragePooling2D()(x)
        x = layers.Dense(96, activation="relu")(x)
        x = layers.Dropout(0.3)(x)

    elif kind == "temporal_cnn":
        x = layers.Conv2D(64, (5, 1), padding="same", use_bias=False)(x)
        x = layers.BatchNormalization()(x)
        x = layers.ReLU()(x)
        x = layers.MaxPool2D((2, 1))(x)
        for f in (96, 128):
            x = layers.Conv2D(f, (3, 1), padding="same", use_bias=False)(x)
            x = layers.BatchNormalization()(x)
            x = layers.ReLU()(x)
            x = layers.MaxPool2D((2, 1))(x)
        x = layers.GlobalAveragePooling2D()(x)
        x = layers.Dense(64, activation="relu")(x)
        x = layers.Dropout(0.3)(x)

    else:
        raise ValueError(kind)

    out = layers.Dense(2, activation="softmax", name="probs")(x)
    m = models.Model(inp, out, name=kind)
    m.compile(optimizer=tf.keras.optimizers.Adam(1e-3),
              loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    return m


# Three variants, deliberately different in shape, so the comparison is meaningful:
#   conv_cnn      plain 2-D CNN over the log-Mel plane       (reference)
#   sep_cnn       depthwise-separable — the smallest / fastest on the ESP32
#   temporal_cnn  1-D convolutions over time, Mel bins as channels (strongest
#                 for keyword-style tasks)
# Add "bsc_cnn" to the list if you want a fourth candidate.
_KINDS_DEFAULT = "conv_cnn,sep_cnn,temporal_cnn"
KINDS = [k.strip() for k in os.environ.get("CRYNET_KINDS", _KINDS_DEFAULT).split(",")
         if k.strip()]
if FAST:
    KINDS = [k for k in KINDS if k in ("conv_cnn", "temporal_cnn")] or KINDS[:1]

for k in KINDS:
```

```
    mm = make_model(k)
    print(f"{k:14s} params={mm.count_params():>9,}  in={mm.input_shape}  out={mm.output_shape}")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 15 — Train every variant, keep the best on validation AUC
# ===========================================================================
EPOCHS = int(os.environ.get("CRYNET_EPOCHS", 3 if FAST else 35))

METRICS, MODELS = {}, {}
BEST_KIND, BEST_AUC = None, -1.0


def _auc(y, p):
    from sklearn.metrics import roc_auc_score
    return float(roc_auc_score(y, p)) if len(set(y.tolist())) > 1 else float("nan")


for kind in KINDS:
    print("\n" + "=" * 72)
    print(f"TRAINING {kind}   ({EPOCHS} epochs)")
    print("=" * 72)
    tf.keras.utils.set_random_seed(SEED)
    model = make_model(kind)

    callbacks = [
        tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss", factor=0.5,
                                             patience=3, min_lr=1e-5, verbose=0),
        tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=5,
                                         restore_best_weights=True, verbose=0),
    ]
    hist = model.fit(ds_train, steps_per_epoch=max(1, len(X_train) // BATCH),
                     epochs=EPOCHS, validation_data=ds_val, callbacks=callbacks,
                     class_weight=cw, verbose=2)

    m = {
        "kind": kind,
        "params": int(model.count_params()),
        "val_auc": _auc(y_val, model.predict(X_val, verbose=0)[:, 1]),
        "test_auc": _auc(y_test, model.predict(X_test, verbose=0)[:, 1]),
        "cross_auc": _auc(y_cross, model.predict(X_cross, verbose=0)[:, 1]),
    }
    METRICS[kind] = m
    MODELS[kind] = model
    print(f"  params={m['params']:,}  val_auc={m['val_auc']:.4f}  "
          f"test_auc={m['test_auc']:.4f}  cross_auc={m['cross_auc']:.4f}")
    if m["val_auc"] == m["val_auc"] and m["val_auc"] > BEST_AUC:
        BEST_KIND, BEST_AUC = kind, m["val_auc"]

BEST_MODEL = MODELS[BEST_KIND]
print("\n" + "=" * 72)
print(f"SELECTED: {BEST_KIND}   val AUC {BEST_AUC:.4f}   "
      f"{BEST_MODEL.count_params():,} parameters")
print("=" * 72)


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 16 — Hard-example mining (second round)
#
# The first round's mistakes are the most valuable training signal there is.
# A cough or a car horn that the model rates as a cry is exactly what turns a
# baby monitor into a nuisance, and the missed cries are exactly what a
# caregiver needs to hear about.  So: score the training set, take the
# worst-scoring negatives and the best-scoring positives, duplicate them, and
# fine-tune for a few epochs at a low learning rate.
#
# Only training rows are mined, so the held-out folds stay honest.
# ===========================================================================
MINED_ROUNDS = int(os.environ.get("CRYNET_MINING_ROUNDS", 1))

if MINED_ROUNDS > 0:
    print("=" * 72)
    print("HARD-EXAMPLE MINING")
    print("=" * 72)
    # Score the training set in chunks.  Fancy-indexing every negative (or
    # every positive) at once copies most of the matrix; chunking does not.
    p_train = np.empty(len(X_train), np.float32)
    CHUNK = 4096
    for _a in range(0, len(X_train), CHUNK):
        _b = min(_a + CHUNK, len(X_train))
        p_train[_a:_b] = BEST_MODEL.predict(X_train[_a:_b], verbose=0)[:, 1]

    neg_idx = np.where(y_train == 0)[0]
```

```
    pos_idx = np.where(y_train == 1)[0]
    p_neg = p_train[neg_idx]
    p_pos = p_train[pos_idx]

    n_hard = int(min(len(neg_idx), 600))
    hard_neg = neg_idx[np.argsort(-p_neg)[:n_hard]]          # most cry-like
    hard_pos = pos_idx[np.argsort(p_pos)[:n_hard]]           # least cry-like
    print(f"  hardest negatives : {len(hard_neg)}  (P(cry) up to {p_neg.max():.3f})")
    print(f"  hardest positives : {len(hard_pos)}  (P(cry) down to {p_pos.min():.3f})")

    # The hard examples are mixed in rather than copied into a new matrix:
    # a second copy of the whole training set is ~900 MB, and this machine
    # has no page file.  The weight reproduces the old behaviour, where each
    # hard row appeared twice -- 1.4% of the rows were duplicated, so 1.4%
    # of the batches were drawn from them.
    hard_idx = np.concatenate([hard_neg, hard_pos])
    X_hard = X_train[hard_idx]
    y_hard = y_train[hard_idx]
    print(f"  fine-tuning set   : {len(X_train)} windows "
          f"(+{len(hard_idx)} hard examples mixed in)")

    ds_mine = tf.data.Dataset.sample_from_datasets(
        [ds_train, make_ds(X_hard, y_hard, BATCH, True)],
        weights=[1.0, 0.014],
        stop_on_empty_dataset=False)

    BEST_MODEL.compile(optimizer=tf.keras.optimizers.Adam(3e-4),
                       loss="sparse_categorical_crossentropy", metrics=["accuracy"])
    mine_hist = BEST_MODEL.fit(
        ds_mine,
        steps_per_epoch=max(1, len(X_train) // BATCH),
        epochs=6,
        validation_data=ds_val,
        callbacks=[tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=3,
                                                    restore_best_weights=True, verbose=0)],
        verbose=2,
    )
    del X_hard, y_hard, ds_mine, p_train
    import gc
    gc.collect()
    print(f"  after mining: val_auc = "
          f"{_auc(y_val, BEST_MODEL.predict(X_val, verbose=0)[:, 1]):.4f}")
else:
    print("hard-example mining disabled (CRYNET_MINING_ROUNDS=0)")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 17 — Metrics, threshold calibration, per-corpus breakdown
# ===========================================================================
from sklearn.metrics import (roc_auc_score, average_precision_score,
                             confusion_matrix, f1_score,
                             precision_recall_fscore_support)


def metrics_from_probs(y, p, threshold=0.5):
    yh = (p >= threshold).astype(int)
    cm = confusion_matrix(y, yh, labels=[0, 1])
    tn, fp, fn, tp = cm.ravel()
    prec, rec, f1, _ = precision_recall_fscore_support(y, yh, average="binary",
                                                       zero_division=0)
    return {
        "n": int(len(y)), "cry": int(y.sum()),
        "auc": float(roc_auc_score(y, p)) if len(set(y.tolist())) > 1 else float("nan"),
        "ap": float(average_precision_score(y, p)) if y.sum() > 0 else float("nan"),
        "acc": float((yh == y).mean()),
        "precision": float(prec), "recall": float(rec), "f1": float(f1),
        "fpr": float(fp / max(fp + tn, 1)),
        "tn": int(tn), "fp": int(fp), "fn": int(fn), "tp": int(tp),
    }


def show(m, name):
    print(f"  {name:26s} n={m['n']:<6} cry={m['cry']:<6} AUC={m['auc']:.4f} "
          f"AP={m['ap']:.4f} acc={m['acc']:.4f} F1={m['f1']:.4f} "
          f"rec={m['recall']:.4f} prec={m['precision']:.4f} FPR={m['fpr']:.4f}")
    return m


p_val = BEST_MODEL.predict(X_val, verbose=0)[:, 1]

print("Threshold sweep on the ICD validation fold")
rows = []
for t in np.arange(0.05, 0.96, 0.01):
    yh = (p_val >= t).astype(int)
```

```
    rec = float((yh[y_val == 1] == 1).mean()) if (y_val == 1).any() else 0.0
    fpr = float((yh[y_val == 0] == 1).mean()) if (y_val == 0).any() else 0.0
    rows.append((float(t), rec, fpr))
ok = [r for r in rows if r[1] >= TARGET_RECALL]
THRESHOLD = float(max(r[0] for r in ok)) if ok else 0.5
print(f"  target recall {TARGET_RECALL:.0%}  ->  THRESHOLD = {THRESHOLD:.2f}")
for t, rec, fpr in rows[::5]:
    print(f"    t={t:.2f}  recall={rec:.4f}  FPR={fpr:.4f}")

print("\nPer-corpus detail at the selected threshold")
print("-" * 104)
res_val = show(metrics_from_probs(y_val, p_val, THRESHOLD), "ICD val (fold 4)")
res_test = show(metrics_from_probs(y_test, BEST_MODEL.predict(X_test, verbose=0)[:, 1], THRESHOLD),
                "ICD test (fold 5)")
res_cross = show(metrics_from_probs(y_cross, BEST_MODEL.predict(X_cross, verbose=0)[:, 1], THRESHOLD),
                 "cross-corpus (unseen)")

print("\nCross-corpus breakdown:")
mc = [m[0] for m in meta_cross]
for grp in sorted(set(mc)):
    sel = np.array([g == grp for g in mc])
    if sel.sum() == 0:
        continue
    show(metrics_from_probs(y_cross[sel],
                            BEST_MODEL.predict(X_cross[sel], verbose=0)[:, 1], THRESHOLD),
         "  " + grp)

print("\nPer-subclass on the ICD test fold:")
mt = [m[0] for m in m_icd]
test_rows = np.where(icd_is_test)[0]
for sub in sorted(set(mt[i] for i in test_rows)):
    sel = np.array([mt[i] == sub for i in test_rows])
    if sel.sum() < 20:
        continue
    p = BEST_MODEL.predict(X_test[sel], verbose=0)[:, 1]
    yt = y_test[sel]
    rec = float((p[yt == 1] >= THRESHOLD).mean()) if (yt == 1).any() else float("nan")
    print(f"  {sub:14s} n={int(sel.sum()):<6} cry={int(yt.sum()):<6} "
          f"recall={rec:.4f} meanP={p.mean():.4f}")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 18 — INT8 quantization + numerical parity with the float model
# ===========================================================================
_repr_idx = np.random.default_rng(SEED).choice(
    len(X_train), min(512, len(X_train)), replace=False)


def representative_gen():
    # The converter wants a LIST with one array per input tensor, and the shape
    # must match the model input exactly: (batch, frames, mels, 1)
    for i in _repr_idx:
        yield [X_train[i:i + 1, :, :, None].astype(np.float32)]


converter = tf.lite.TFLiteConverter.from_keras_model(BEST_MODEL)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_gen
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
tflite_int8 = converter.convert()
(OUT / "crynet_int8.tflite").write_bytes(tflite_int8)

conv_f = tf.lite.TFLiteConverter.from_keras_model(BEST_MODEL)
conv_f.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_float = conv_f.convert()
(OUT / "crynet_float.tflite").write_bytes(tflite_float)

interp = tf.lite.Interpreter(model_content=tflite_int8)
interp.allocate_tensors()
inp_d = interp.get_input_details()[0]
out_d = interp.get_output_details()[0]
IN_SCALE, IN_ZP = float(inp_d["quantization"][0]), int(inp_d["quantization"][1])
OUT_SCALE, OUT_ZP = float(out_d["quantization"][0]), int(out_d["quantization"][1])
print("input :", inp_d["shape"], inp_d["dtype"].__name__,
      f"scale={IN_SCALE:.8f} zp={IN_ZP}")
print("output:", out_d["shape"], out_d["dtype"].__name__,
      f"scale={OUT_SCALE:.8f} zp={OUT_ZP}")


# --- guardrail: every op in the model must have a TFLite Micro kernel ---------
# (TFLM registers MEAN via kernels/reduce.cc, AVERAGE_POOL_2D via pooling.cc, ...)
TFLM_KERNELS = {
```

```
    "CONV_2D", "DEPTHWISE_CONV_2D", "AVERAGE_POOL_2D", "MAX_POOL_2D",
    "MEAN", "FULLY_CONNECTED", "SOFTMAX", "LOGISTIC", "RELU", "TANH",
    "RESHAPE", "PAD", "SQUEEZE", "LEAKY_RELU",
}
model_ops = sorted({d["op_name"] for d in interp._get_ops_details()})
# DELEGATE only appears because the host interpreter runs XNNPACK; TFLM has none
ops_for_tflm = [o for o in model_ops if o != "DELEGATE"]
missing_kernels = [o for o in ops_for_tflm if o not in TFLM_KERNELS]
print("model operators :", ops_for_tflm)
if missing_kernels:
    print(f"  WARNING — no TFLite Micro kernel for: {missing_kernels}")
    print("  The model will NOT run on the ESP32-S3 until this is fixed.")
else:
    print("  all operators have a TFLite Micro kernel")


def run_int8(interp, X):
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]
    out = np.zeros((len(X), 2), np.float32)
    for i in range(len(X)):
        q = np.clip(np.round(X[i].astype(np.float32) / s) + z, -128, 127).astype(np.int8)
        interp.set_tensor(i_d["index"], q[None, ..., None])
        interp.invoke()
        o = interp.get_tensor(o_d["index"])[0].astype(np.float32)
        out[i] = (o - o_d["quantization"][1]) * o_d["quantization"][0]
    return out


n_check = min(4000, len(X_val))
# NOTE: X_val is ordered by the ICD manifest, which lists every positive clip
# before every negative one, so the first n_check windows of the validation
# fold would be ALL cry (the fold's 2,665 positive clips alone contribute
# 5,330 windows).  A prefix is therefore not a sample: on the full run it
# produced a single-class slice, whose AUC is undefined, and the parity
# numbers were reported as NaN.  Draw a seeded random subset instead, so
# the slice always contains both classes.
_parity_rng = np.random.default_rng(SEED + 21)
_sel = _parity_rng.permutation(len(X_val))[:n_check]
p_int8 = run_int8(interp, X_val[_sel])[:, 1]
p_float = BEST_MODEL.predict(X_val[_sel], verbose=0)[:, 1]
agree = float(((p_int8 >= THRESHOLD) == (p_float >= THRESHOLD)).mean())
print(f"\nparity on {n_check} validation windows (random subset, "
      f"{int(y_val[_sel].sum())} cry):")
print(f"  max |delta p|        = {np.abs(p_int8 - p_float).max():.4f}")
print(f"  mean |delta p|       = {np.abs(p_int8 - p_float).mean():.5f}")
print(f"  decision agreement   = {agree:.4%}")

res_int8 = show(metrics_from_probs(y_val[_sel], p_int8, THRESHOLD),
                "INT8 (as deployed) val")
res_f32 = show(metrics_from_probs(y_val[_sel], p_float, THRESHOLD),
               "float32 reference val")
if np.isnan(res_int8["auc"]) or np.isnan(res_f32["auc"]):
    print("  WARNING — the parity slice contains a single class; its AUC is "
          "undefined. Draw the subset with _parity_rng.permutation instead of "
          "taking a prefix of the manifest order.")
print(f"\n  model size : int8 {len(tflite_int8)/1024:.1f} KB | "
      f"float32 {len(tflite_float)/1024:.1f} KB")
print(f"  AUC delta  : {res_int8['auc'] - res_f32['auc']:+.5f}")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 19 — Export the deployment artifacts
# ===========================================================================
BEST_MODEL.save(OUT / "crynet.keras")
print("saved the trained Keras model -> crynet.keras")


def to_c_array(data: bytes, name: str) -> str:
    out = [f"// {len(data)} bytes — generated by the CryNet notebook",
           f"const unsigned char {name}[] = {{"]
    for i in range(0, len(data), 16):
        out.append("  " + ", ".join(f"0x{b:02x}" for b in data[i:i + 16]) + ",")
    out.append("};")
    out.append(f"const unsigned int {name}_len = {len(data)};")
    return "\n".join(out)


(OUT / "model_data.h").write_text(
    "#ifndef CRYNET_MODEL_DATA_H\n#define CRYNET_MODEL_DATA_H\n"
    "#include <stdint.h>\n\n"
    "#ifdef __cplusplus\nextern \"C\" {\n#endif\n\n"
    + to_c_array(tflite_int8, "crynet_model")
```

```
    + "\n\n#ifdef __cplusplus\n}\n#endif\n#endif\n", encoding="utf-8")

# --- sparse Mel filterbank + Hann window ----------------------------------
mel_rows = []
for m in range(N_MELS):
    w = MEL_BANK[m]
    nz = np.nonzero(w > 0)[0]
    mel_rows.append((int(nz[0]), int(nz[-1]) + 1,
                     w[nz[0]:nz[-1] + 1].astype(np.float32)))
flat = np.concatenate([w for _, _, w in mel_rows])
offs, acc = [], 0
for _, _, w in mel_rows:
    offs.append(acc)
    acc += len(w)

L = [
    "#ifndef CRYNET_TABLES_H",
    "#define CRYNET_TABLES_H",
    "#include <stdint.h>",
    "",
    "/* Generated by the CryNet training notebook — do not edit by hand. */",
    f"#define CRYNET_SR            {SR}",
    f"#define CRYNET_N_SAMPLES     {N_SAMPLES}",
    f"#define CRYNET_N_FFT         {N_FFT}",
    f"#define CRYNET_HOP           {HOP}",
    f"#define CRYNET_N_MELS        {N_MELS}",
    f"#define CRYNET_N_FRAMES      {N_FRAMES}",
    f"#define CRYNET_CENTER_PAD    {CENTER_PAD}",
    f"#define CRYNET_TAIL_PAD      {TAIL_PAD}",
    f"#define CRYNET_DB_FLOOR      ({DB_FLOOR:.1f}f)",
    f"#define CRYNET_LOG_EPS       ({LOG_EPS:.10f}f)",
    f"#define CRYNET_CMVN          {1 if CMVN else 0}",
    f"#define CRYNET_IN_SCALE      {IN_SCALE:.9f}f",
    f"#define CRYNET_IN_ZP         {IN_ZP}",
    f"#define CRYNET_OUT_SCALE     {OUT_SCALE:.9f}f",
    f"#define CRYNET_OUT_ZP        {OUT_ZP}",
    f"#define CRYNET_THRESHOLD     {THRESHOLD:.4f}f",
    f"#define CRYNET_SILENCE_RMS   {SILENCE_RMS_GATE:.6f}f",
    "",
    "/* Sparse Mel filterbank: band m uses FFT bins",
    "   CRYNET_MEL_START[m] .. CRYNET_MEL_END[m]-1 with weights taken from",
    "   CRYNET_MEL_WEIGHTS[CRYNET_MEL_OFFSET[m] + (bin - CRYNET_MEL_START[m])]. */",
    "const int16_t CRYNET_MEL_START[CRYNET_N_MELS] = {",
    "  " + ", ".join(str(s) for s, _, _ in mel_rows),
    "};",
    "const int16_t CRYNET_MEL_END[CRYNET_N_MELS] = {",
    "  " + ", ".join(str(e) for _, e, _ in mel_rows),
    "};",
    "const int16_t CRYNET_MEL_OFFSET[CRYNET_N_MELS] = {",
    "  " + ", ".join(str(o) for o in offs),
    "};",
    f"const float CRYNET_MEL_WEIGHTS[{acc}] = {{",
]
for i in range(0, acc, 10):
    L.append("  " + ", ".join(f"{v:.8f}f" for v in flat[i:i + 10]) + ",")
L.append("};")
L.append("")
L.append(f"const float CRYNET_HANN[CRYNET_N_FFT] = {{")
for i in range(0, N_FFT, 10):
    L.append("  " + ", ".join(f"{v:.8f}f" for v in HANN[i:i + 10]) + ",")
L.append("};")
L.append("")
L.append("#endif /* CRYNET_TABLES_H */")
(OUT / "crynet_tables.h").write_text("\n".join(L), encoding="utf-8")

print(f"wrote {OUT/'model_data.h'}   ({len(tflite_int8)/1024:.1f} KB model)")
print(f"wrote {OUT/'crynet_tables.h'} ({acc} Mel weights)")


# ---------------------------------------------------------------------------
# ============================================================================
# CELL 20 — The C reference implementation (port this to ESP-IDF)
# ============================================================================
C_SOURCE = r"""
/* ============================================================================
 * dsp_crynet.c — CryNet log-Mel front end, numerically identical to the
 *                Python implementation used for training.
 *
 *  Host parity test:
 *      gcc -O2 -DCRYNET_SELFTEST -o dsp_test dsp_crynet.c -lm
 *      ./dsp_test probe.s16le
 *
 *  On the ESP32-S3 replace fft512() with ESP-DSP:
 *      dsps_fft2r() + dsps_cmplx_mag()  (roughly 10x faster than the
 *      reference FFT below, which is kept so the file builds anywhere).
```

```c
 * ========================================================================*/
#include <math.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#include "crynet_tables.h"

typedef struct { float re, im; } cpx_t;

/* ---- iterative radix-2 FFT, N = 512, in place ---------------------------- */
static void fft512(cpx_t *a) {
    unsigned i, j, len;
    cpx_t t;
    for (i = 1, j = 0; i < 512; i++) {
        unsigned bit = 512 >> 1;
        for (; j & bit; bit >>= 1) j ^= bit;
        j ^= bit;
        if (i < j) { t = a[i]; a[i] = a[j]; a[j] = t; }
    }
    for (len = 2; len <= 512; len <<= 1) {
        float ang = -2.0f * 3.14159265358979f / (float)len;
        float wr = cosf(ang), wi = sinf(ang);
        for (i = 0; i < 512; i += len) {
            float cr = 1.0f, ci = 0.0f;
            for (j = 0; j < len / 2; j++) {
                cpx_t u = a[i + j];
                cpx_t v;
                v.re = a[i + j + len / 2].re * cr - a[i + j + len / 2].im * ci;
                v.im = a[i + j + len / 2].re * ci + a[i + j + len / 2].im * cr;
                a[i + j].re = u.re + v.re;  a[i + j].im = u.im + v.im;
                a[i + j + len / 2].re = u.re - v.re;
                a[i + j + len / 2].im = u.im - v.im;
                float nr = cr * wr - ci * wi;
                ci = cr * wi + ci * wr;
                cr = nr;
            }
        }
    }
}

/* ---- log-Mel front end --------------------------------------------------- */
/*  pcm     : CRYNET_N_SAMPLES int16 mono samples @ 16 kHz
 *  mel_out : CRYNET_N_FRAMES * CRYNET_N_MELS floats (standardised dB)     */
void crynet_log_mel(const int16_t *pcm, float *mel_out) {
    static float x[CRYNET_N_SAMPLES + CRYNET_CENTER_PAD + CRYNET_TAIL_PAD];
    static cpx_t spec[CRYNET_N_FFT];
    static float power[CRYNET_N_FFT / 2 + 1];
    static float mel[CRYNET_N_MELS];

    /* 1. DC removal + normalise to [-1, 1] */
    float mean = 0.0f;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) mean += (float)pcm[i];
    mean /= (float)CRYNET_N_SAMPLES;

    memset(x, 0, sizeof(float) * (CRYNET_N_SAMPLES + CRYNET_CENTER_PAD + CRYNET_TAIL_PAD));
    for (int i = 0; i < CRYNET_N_SAMPLES; i++)
        x[CRYNET_CENTER_PAD + i] = ((float)pcm[i] - mean) / 32768.0f;

    /* 2. frames -> FFT -> power -> mel -> dB */
    for (int fr = 0; fr < CRYNET_N_FRAMES; fr++) {
        const float *w = x + fr * CRYNET_HOP;
        for (int i = 0; i < CRYNET_N_FFT; i++) {
            spec[i].re = w[i] * CRYNET_HANN[i];
            spec[i].im = 0.0f;
        }
        fft512(spec);
        for (int i = 0; i <= CRYNET_N_FFT / 2; i++)
            power[i] = spec[i].re * spec[i].re + spec[i].im * spec[i].im;
        for (int m = 0; m < CRYNET_N_MELS; m++) {
            float acc = 0.0f;
            int start = CRYNET_MEL_START[m], end = CRYNET_MEL_END[m];
            const float *wts = &CRYNET_MEL_WEIGHTS[CRYNET_MEL_OFFSET[m]];
            for (int b = start; b < end; b++) acc += power[b] * wts[b - start];
            mel[m] = acc;
        }
        for (int m = 0; m < CRYNET_N_MELS; m++) {
            float p = mel[m];
            if (p < CRYNET_LOG_EPS) p = CRYNET_LOG_EPS;
            float db = 10.0f * log10f(p);
            if (db < CRYNET_DB_FLOOR) db = CRYNET_DB_FLOOR;
            mel_out[fr * CRYNET_N_MELS + m] = db;
        }
    }

```

```
#if CRYNET_CMVN
    /* 3. per-window standardisation */
    float mu = 0.0f;
    const int n = CRYNET_N_FRAMES * CRYNET_N_MELS;
    for (int i = 0; i < n; i++) mu += mel_out[i];
    mu /= (float)n;
    float var = 0.0f;
    for (int i = 0; i < n; i++) { float d = mel_out[i] - mu; var += d * d; }
    float sd = sqrtf(var / (float)n) + 1e-6f;
    for (int i = 0; i < n; i++) mel_out[i] = (mel_out[i] - mu) / sd;
#endif
}

/* ---- quantisation helpers ------------------------------------------------ */
void crynet_quantise(const float *mel, int8_t *q) {
    const int n = CRYNET_N_FRAMES * CRYNET_N_MELS;
    for (int i = 0; i < n; i++) {
        float v = nearbyintf(mel[i] / CRYNET_IN_SCALE) + (float)CRYNET_IN_ZP;
        if (v > 127.0f) v = 127.0f;
        if (v < -128.0f) v = -128.0f;
        q[i] = (int8_t)v;
    }
}

float crynet_dequantise(int8_t v) {
    return ((float)v - (float)CRYNET_OUT_ZP) * CRYNET_OUT_SCALE;
}

/* ---- silence gate -------------------------------------------------------- */
float crynet_window_rms(const int16_t *pcm) {
    float mean = 0.0f;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) mean += (float)pcm[i];
    mean /= (float)CRYNET_N_SAMPLES;
    float acc = 0.0f;
    for (int i = 0; i < CRYNET_N_SAMPLES; i++) {
        float d = ((float)pcm[i] - mean) / 32768.0f;
        acc += d * d;
    }
    return sqrtf(acc / (float)CRYNET_N_SAMPLES);
}

/* ---- host parity test ---------------------------------------------------- */
#ifdef CRYNET_SELFTEST
int main(int argc, char **argv) {
    if (argc < 2) { fprintf(stderr, "usage: %s raw_pcm_s16le.bin\n", argv[0]); return 2; }
    FILE *f = fopen(argv[1], "rb");
    if (!f) { perror("fopen"); return 2; }
    static int16_t pcm[CRYNET_N_SAMPLES];
    size_t got = fread(pcm, sizeof(int16_t), CRYNET_N_SAMPLES, f);
    fclose(f);
    if (got != CRYNET_N_SAMPLES) { fprintf(stderr, "short read\n"); return 2; }
    static float mel[CRYNET_N_FRAMES * CRYNET_N_MELS];
    crynet_log_mel(pcm, mel);
    FILE *o = fopen("mel_c.bin", "wb");
    fwrite(mel, sizeof(float), CRYNET_N_FRAMES * CRYNET_N_MELS, o);
    fclose(o);
    fprintf(stderr, "wrote mel_c.bin  rms=%.6f\n", crynet_window_rms(pcm));
    return 0;
}
#endif
"""

(OUT / "dsp_crynet.c").write_text(C_SOURCE, encoding="utf-8")
(OUT / "dsp_crynet.h").write_text(
    "#ifndef DSP_CRYNET_H\n#define DSP_CRYNET_H\n#include <stdint.h>\n"
    "#include \"crynet_tables.h\"\n"
    "#ifdef __cplusplus\nextern \"C\" {\n#endif\n"
    "void crynet_log_mel(const int16_t *pcm, float *mel_out);\n"
    "void crynet_quantise(const float *mel, int8_t *q);\n"
    "float crynet_dequantise(int8_t v);\n"
    "float crynet_window_rms(const int16_t *pcm);\n"
    "#ifdef __cplusplus\n}\n#endif\n#endif\n", encoding="utf-8")

# --- TFLite Micro glue: the missing link between the DSP and the model -------
TFLM_GLUE = r"""
/* ============================================================================
 * crynet_infer_example.c — glue between the DSP front end and TFLite Micro.
 *
 * Port this into the ESP32-S3 firmware.  It assumes:
 *   - model_data.h   (the INT8 model as a C array)
 *   - crynet_tables.h (Mel filterbank, Hann window, quantisation constants)
 *   - dsp_crynet.c   (the log-Mel front end)
 *
 * ESP-IDF: add the TensorFlowLite-ESP32 component and include
 * "tensorflow/lite/micro/micro_interpreter.h".
```

```
 * ========================================================================*/
#include <stdint.h>
#include <string.h>

#include "tensorflow/lite/micro/micro_interpreter.h"
#include "tensorflow/lite/micro/micro_mutable_op_resolver.h"
#include "tensorflow/lite/schema/schema_generated.h"

#include "dsp_crynet.h"
#include "model_data.h"

#ifndef CRYNET_ARENA_SIZE
#define CRYNET_ARENA_SIZE 120000   /* bytes; raise if AllocateTensors fails */
#endif

/* The ops the exported model uses: CONV_2D, DEPTHWISE_CONV_2D, MAX_POOL_2D,
 * MEAN, FULLY_CONNECTED, SOFTMAX.  AddConv2D() etc. come from
 * tensorflow/lite/micro/kernels/kernel_registrar.h. */
static tflite::MicroMutableOpResolver<6> g_resolver;
static tflite::MicroInterpreter *g_interp = nullptr;
static uint8_t *g_arena = nullptr;

bool crynet_init() {
  g_arena = (uint8_t *)malloc(CRYNET_ARENA_SIZE);
  if (g_arena == nullptr) return false;
  g_resolver.AddConv2D();
  g_resolver.AddDepthwiseConv2D();
  g_resolver.AddMaxPool2D();
  g_resolver.AddMean();            /* GlobalAveragePooling2D lowers to MEAN */
  g_resolver.AddFullyConnected();
  g_resolver.AddSoftmax();

  const tflite::Model *model = tflite::GetModel(crynet_model);
  static tflite::MicroInterpreter interpreter(model, g_resolver, g_arena,
                                              CRYNET_ARENA_SIZE);
  if (interpreter.AllocateTensors() != kTfLiteOk) return false;
  g_interp = &interpreter;
  return true;
}

/* pcm: CRYNET_N_SAMPLES int16 samples @ 16 kHz mono.
 * Returns P(cry) in [0, 1]; 0.0 if the window is silent or inference failed. */
float crynet_probability(const int16_t *pcm) {
  if (g_interp == nullptr) return 0.0f;
  if (crynet_window_rms(pcm) < CRYNET_SILENCE_RMS) return 0.0f;   /* silence gate */

  static float mel[CRYNET_N_FRAMES * CRYNET_N_MELS];
  static int8_t quant[CRYNET_N_FRAMES * CRYNET_N_MELS];
  crynet_log_mel(pcm, mel);
  crynet_quantise(mel, quant);

  TfLiteTensor *in = g_interp->input(0);
  memcpy(in->data.raw, quant, sizeof(quant));
  in->bytes = sizeof(quant);
  if (g_interp->Invoke() != kTfLiteOk) return 0.0f;

  TfLiteTensor *out = g_interp->output(0);
  const int8_t *o = out->data.int8;
  /* output order is [P(no_cry), P(cry)] */
  return (float)(o[1] - out->params.zero_point) * out->params.scale;
}

/* Call once per 2 s window; returns true when the alarm should fire. */
bool crynet_is_cry(const int16_t *pcm, float *confidence) {
  const float p = crynet_probability(pcm);
  if (confidence) *confidence = p;
  return p >= CRYNET_THRESHOLD;
}
"""
(OUT / "crynet_infer_example.c").write_text(TFLM_GLUE, encoding="utf-8")

contract = {
    "model": "crynet_int8.tflite",
    "model_sha256": hashlib.sha256(tflite_int8).hexdigest(),
    "model_bytes": len(tflite_int8),
    "input": {
        "name": "mel", "shape": [1, N_FRAMES, N_MELS, 1], "dtype": "int8",
        "scale": IN_SCALE, "zero_point": IN_ZP,
        "preprocessing": [
            f"int16 PCM, {SR} Hz mono, {N_SAMPLES} samples ({WIN_SEC}s)",
            "subtract mean (DC removal)",
            f"zero-pad {CENTER_PAD} front / {TAIL_PAD} rear",
            f"{N_FRAMES} frames, {N_FFT}-pt Hann, hop {HOP}",
            f"{N_MELS} Mel bands (librosa slaney, {FMIN}-{FMAX} Hz)",
            f"10*log10(max(P, {LOG_EPS})) floored at {DB_FLOOR} dB",
```

```
            "per-window standardisation" if CMVN else "no CMVN",
            "q = round(v/scale) + zero_point",
        ],
    },
    "output": {"shape": [1, 2], "dtype": "int8", "scale": OUT_SCALE,
               "zero_point": OUT_ZP, "classes": ["no_cry", "cry"]},
    "operators": ops_for_tflm,
    "tflm_kernel_check": "ok" if not missing_kernels else missing_kernels,
    "decision": {
        "threshold": THRESHOLD,
        "silence_rms_gate": SILENCE_RMS_GATE,
        "rule": "P(cry) >= threshold  ->  ALERT",
    },
    "tflite_micro": {
        "arena_bytes_hint": 120000,
        "note": "put the tensor arena in PSRAM if it does not fit in internal SRAM",
    },
    "metrics": {
        "selected_model": BEST_KIND,
        "parameters": int(BEST_MODEL.count_params()),
        "val_auc": res_val["auc"], "test_auc": res_test["auc"], "cross_auc": res_cross["auc"],
        "val_f1": res_val["f1"], "test_f1": res_test["f1"], "cross_f1": res_cross["f1"],
        "val_recall": res_val["recall"], "val_precision": res_val["precision"],
        "int8_val_auc": res_int8["auc"], "float_val_auc": res_f32["auc"],
    },
}
(OUT / "deploy_config.json").write_text(json.dumps(contract, indent=2), encoding="utf-8")
(OUT / "metrics.json").write_text(json.dumps(
    {"selected": BEST_KIND, "threshold": THRESHOLD,
     "variants": {k: v for k, v in METRICS.items()}}, indent=2), encoding="utf-8")

print("artifacts written to", OUT)
for f in sorted(OUT.iterdir()):
    print(f"   {f.name:26s} {f.stat().st_size/1024:9.1f} KB")


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 21 — C <-> Python numerical parity test (skipped when no compiler)
# ===========================================================================
gcc = shutil.which("gcc") or shutil.which("cc") or shutil.which("clang")
CC_CMD = [gcc] if gcc else None
if CC_CMD is None:
    try:                                    # pip install ziglang -> a full C compiler
        import ziglang
        CC_CMD = [sys.executable, "-m", "ziglang", "cc"]
    except Exception:
        CC_CMD = None
parity = None

if CC_CMD is None:
    print("no C compiler found — skipping the parity test.")
    print("Compile it yourself:  gcc -O2 -DCRYNET_SELFTEST -o dsp_test "
          "out/dsp_crynet.c -lm")
else:
    print("compiling out/dsp_crynet.c with", " ".join(CC_CMD), "...")
    exe = WORK / ("dsp_test" + (".exe" if os.name == "nt" else ""))
    env = dict(os.environ)
    env.setdefault("ZIG_GLOBAL_CACHE_DIR", str(WORK / "zigcache"))
    r = subprocess.run(CC_CMD + ["-O2", "-DCRYNET_SELFTEST", "-I", str(OUT),
                                 "-o", str(exe), str(OUT / "dsp_crynet.c"), "-lm"],
                       capture_output=True, text=True, env=env)
    if r.returncode != 0:
        print("compile failed:\n", r.stderr[:2000])
    else:
        rs = np.random.default_rng(7)
        t = np.arange(N_SAMPLES) / SR
        pcm = (0.35 * np.sin(2 * np.pi * 420 * t)
               + 0.12 * np.sin(2 * np.pi * 1350 * t)
               + 0.02 * rs.standard_normal(N_SAMPLES)).astype(np.float32)
        pcm16 = (np.clip(pcm, -1, 1) * 32767).astype(np.int16)
        raw = WORK / "probe.s16le"
        raw.write_bytes(pcm16.tobytes())

        run = subprocess.run([str(exe), str(raw)], capture_output=True, text=True, cwd=str(WORK))
        if run.returncode != 0:
            print("parity binary failed:", run.stderr[:1000])
        else:
            mel_c = np.fromfile(WORK / "mel_c.bin", dtype=np.float32).reshape(N_FRAMES, N_MELS)
            mel_py = log_mel(pcm16.astype(np.float32) / 32768.0)
            diff = np.abs(mel_c - mel_py)
            parity = {"max_abs_diff": float(diff.max()),
                      "mean_abs_diff": float(diff.mean())}
            print(f"  max |C - Python|  = {diff.max():.3e}")
            print(f"  mean|C - Python|  = {diff.mean():.3e}")
```

```
            if diff.max() < 1e-2:
                print("  OK — the C front end matches training (float32 rounding only)")
            else:
                print("  WARNING: mismatch — do not ship this C code")
            (OUT / "c_parity.json").write_text(json.dumps(parity, indent=2), encoding="utf-8")

print(json.dumps({"parity": parity}, indent=2))


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 22 — Final report
# ===========================================================================
print("=" * 78)
print("  CryNet — final report")
print("=" * 78)
print(f"  selected model     : {BEST_KIND}  ({BEST_MODEL.count_params():,} parameters)")
print(f"  int8 model size    : {len(tflite_int8)/1024:.1f} KB")
print(f"  input              : [1,{N_FRAMES},{N_MELS},1] int8  "
      f"(scale {IN_SCALE:.6f}, zp {IN_ZP})")
print(f"  output             : [1,2] int8 -> [P(no_cry), P(cry)]")
print(f"  alarm threshold    : {THRESHOLD:.2f}")
print(f"  silence RMS gate   : {SILENCE_RMS_GATE}")
print()
print(f"  ICD val  (fold 4)  : AUC {res_val['auc']:.4f}  F1 {res_val['f1']:.4f}  "
      f"recall {res_val['recall']:.4f}  FPR {res_val['fpr']:.4f}")
print(f"  ICD test (fold 5)  : AUC {res_test['auc']:.4f}  F1 {res_test['f1']:.4f}  "
      f"recall {res_test['recall']:.4f}  FPR {res_test['fpr']:.4f}")
print(f"  cross-corpus       : AUC {res_cross['auc']:.4f}  F1 {res_cross['f1']:.4f}  "
      f"recall {res_cross['recall']:.4f}  FPR {res_cross['fpr']:.4f}")
print(f"  int8 vs float32    : AUC {res_int8['auc']:.4f} vs {res_f32['auc']:.4f}  "
      f"(delta {res_int8['auc']-res_f32['auc']:+.5f})")
print()
print("  artifacts in", OUT)
for f in sorted(OUT.iterdir()):
    print(f"    {f.name:26s} {f.stat().st_size/1024:9.1f} KB")
print()
print("  next: flash model_data.h + crynet_tables.h + dsp_crynet.c into the")
print("        ESP32-S3 firmware, then wire crynet_log_mel() -> TFLite Micro.")
print("=" * 78)


# ---------------------------------------------------------------------------
#
# ---
#
# ## Where to go next (checkpoint 2)
#
# 1. **Wire the front end into the firmware.** `out/dsp_crynet.c` is portable C99 — swap `fft512()` for ESP-DSP `dsps_fft2r()` + `dsps_cmplx_mag()` (≈10× faster) and copy `crynet_tables.h` and `model_data.h` next to it.
# 2. **Run TFLite Micro** with an INT8 input tensor `[1, 81, 64, 1]`; put the tensor arena in PSRAM if it does not fit in internal SRAM.
# 3. **Lower the trigger threshold.** The old hub fired an audio pull at RMS ≥ 0.18, so a *soft* cry never reached any classifier. Feed the model anything above ~0.03 RMS and let it reject the false triggers — rejecting false triggers is exactly what it was trained for (speech, cat meows, household noise).
# 4. **Record ~50 real clips from your own INMP441** (cries + room noise + speech + TV), then re-run this notebook with them as a small custom corpus. Matching your actual microphone, gain and room is the single biggest accuracy win left.
# 5. **Alert policy.** The existing BLE packet carries `cot_id`, `alert_type`, `confidence_pct`; the model output maps straight onto it (`alert_type = 0x01` when `P(cry) >= threshold`).
# ---------------------------------------------------------------------------


# ---------------------------------------------------------------------------
# ===========================================================================
# CELL 20 — Package the artifacts for download
#
# Everything the deployment needs is in `out/`.  On Kaggle the notebook's
# working directory is the output of the saved version, so the artifacts can
# be taken from there — but a single zip is easier to hand over than a dozen
# files, and it carries the exact bytes that were validated.
# ===========================================================================
ZIP_NAME = "crynet_artifacts.zip"
ZIP_PATH = Path.cwd() / ZIP_NAME

_files = sorted(p for p in OUT.rglob("*") if p.is_file())
if not _files:
    raise SystemExit(f"nothing to package in {OUT}")

with zipfile.ZipFile(ZIP_PATH, "w", zipfile.ZIP_DEFLATED, compresslevel=1) as _z:
    for _p in _files:
        _z.write(_p, arcname=str(Path("out") / _p.relative_to(OUT)))

print("=" * 78)
print(f"packaged {len(_files)} files -> {ZIP_PATH}  ({ZIP_PATH.stat().st_size/1e6:.1f} MB)")
for _p in _files:
    print(f"    {str(_p.relative_to(OUT)):26s} {_p.stat().st_size/1024:9.1f} KB")
```

```
print()
print("Download it from the Output tab of the saved notebook version:")
print("  Kaggle: Save Version -> Save & Run All -> Output -> crynet_artifacts.zip")
print("=" * 78)
```

Evaluation and verification

research/eval_indomain.py

189 lines · 7,645 bytes

```
#!/usr/bin/env python3
"""Measure the domain gap: run the deployed model over the in-domain clips.

The deployed model was trained on Kaggle corpora.  This script runs the
*exported* INT8 model — the very file that is flashed onto the hub — over
the clips recorded from this room's microphone, using the notebook's own
front end, so the numbers are what the hub would actually produce.

It answers two questions:

  1. Would the hub even ask the model?  The hub only pulls a window when
     the node's telemetry RMS reaches CRYNET_TRIGGER_RMS (0.030), so a clip
     quieter than that never reaches the classifier at all.
  2. If it did, what would the model say?  AUC and the confusion matrix
     over the labelled clips, at the deployed threshold of 0.95.

Usage:
    python research/eval_indomain.py --session research/in_domain/session1
    python research/eval_indomain.py --session research/in_domain/session1 \
        --threshold 0.95 --window-seconds 2
"""

from __future__ import annotations

import argparse
import csv
import json
import sys
import wave
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parents[1]
OUT = ROOT / "research" / "out"

sys.path.insert(0, str(ROOT / "research"))
from verify_int8_parity import notebook_namespace  # noqa: E402


def read_wav(path: Path) -> np.ndarray:
    """A 16-bit PCM WAV as float32 in [-1, 1] — the scale soundfile gives."""
    with wave.open(str(path), "rb") as w:
        if w.getsampwidth() != 2 or w.getnchannels() != 1:
            raise ValueError(f"{path} is not 16-bit mono")
        raw = w.readframes(w.getnframes())
    return np.frombuffer(raw, dtype="<i2").astype(np.float32) / 32768.0


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--session", required=True)
    ap.add_argument("--threshold", type=float, default=0.95,
                    help="the deployed decision threshold")
    ap.add_argument("--trigger-rms", type=float, default=0.030,
                    help="the hub's trigger level (CRYNET_TRIGGER_RMS)")
    ap.add_argument("--windows-per-clip", type=int, default=2)
    ap.add_argument("--model", default=str(OUT / "crynet_int8.tflite"),
                    help="the INT8 model to score; point it at a candidate "
                         "to judge a retrain before it is deployed")
    ap.add_argument("--report", default=str(OUT / "eval_indomain.json"))
    args = ap.parse_args()

    session = Path(args.session)
    manifest_path = session / "manifest.csv"
    if not manifest_path.is_file():
        print(f"no manifest in {session} — run firmware/tools/label_session.py first")
        return 1
```

```
    rows = list(csv.DictReader(manifest_path.open(newline="")))
    if not rows:
        print("the manifest is empty")
        return 1

    ns = notebook_namespace()
    log_mel = ns["log_mel"]
    take_windows = ns["take_windows"]

    # ---- features ------------------------------------------------------------
    feats, labels, per_clip = [], [], []
    skipped = 0
    for row in rows:
        path = session / "clips_labeled" / row["file"]
        if not path.is_file():
            skipped += 1
            continue
        x = read_wav(path)
        label = 1 if row["label"] == "cry" else 0
        windows = take_windows(x, args.windows_per_clip, None, mode="center")
        f = np.stack([log_mel(w) for w in windows]).astype(np.float32)
        feats.append(f)
        labels.extend([label] * f.shape[0])
        per_clip.append((row["file"], label, float(row["rms_mean"]),
                         float(row["rms_peak"])))
    if skipped:
        print(f"warning: {skipped} clip(s) named in the manifest are missing")
    X = np.concatenate(feats, axis=0)
    y = np.asarray(labels, dtype=np.int32)
    print(f"\nclips: {len(per_clip)}  windows: {X.shape[0]}  "
          f"cry windows: {int(y.sum())}")

    # ---- the deployed INT8 model ---------------------------------------------
    import tensorflow as tf  # noqa: E402

    interp = tf.lite.Interpreter(model_path=str(args.model))
    interp.allocate_tensors()
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]

    p_int8 = np.zeros((len(X), 2), np.float32)
    for i in range(len(X)):
        q = np.clip(np.round(X[i].astype(np.float32) / s) + z, -128, 127).astype(np.int8)
        interp.set_tensor(i_d["index"], q[None, ..., None])
        interp.invoke()
        o = interp.get_tensor(o_d["index"])[0].astype(np.float32)
        p_int8[i] = (o - o_d["quantization"][1]) * o_d["quantization"][0]

    keras_model = tf.keras.models.load_model(str(OUT / "crynet.keras"))
    p_float = keras_model.predict(X, verbose=0).astype(np.float32)

    # ---- metrics -------------------------------------------------------------
    from sklearn.metrics import roc_auc_score  # noqa: E402

    def confusion(p):
        yh = (p >= args.threshold).astype(int)
        tp = int(((yh == 1) & (y == 1)).sum())
        fn = int(((yh == 0) & (y == 1)).sum())
        fp = int(((yh == 1) & (y == 0)).sum())
        tn = int(((yh == 0) & (y == 0)).sum())
        return {
            "tp": tp, "fn": fn, "fp": fp, "tn": tn,
            "accuracy": (tp + tn) / max(len(y), 1),
            "recall": tp / max(tp + fn, 1),
            "fpr": fp / max(fp + tn, 1),
        }

    auc = float(roc_auc_score(y, p_int8[:, 1])) if len(set(y.tolist())) > 1 else float("nan")
    conf_int8 = confusion(p_int8[:, 1])
    conf_float = confusion(p_float[:, 1])

    # ---- the trigger ---------------------------------------------------------
    cry_clips = [c for c in per_clip if c[1] == 1]
    quiet_clips = [c for c in per_clip if c[1] == 0]
    above = sum(1 for c in cry_clips if c[2] >= args.trigger_rms)
    below = sum(1 for c in quiet_clips if c[2] >= args.trigger_rms)

    report = {
        "source": "research/eval_indomain.py",
        "session": str(session),
        "clips": len(per_clip),
        "windows": int(X.shape[0]),
        "cry_windows": int(y.sum()),
```

```
        "threshold": args.threshold,
        "trigger_rms": args.trigger_rms,
        "auc_int8": auc,
        "int8": conf_int8,
        "float32": conf_float,
        "trigger": {
            "cry_clips_above_trigger": above,
            "cry_clips": len(cry_clips),
            "no_cry_clips_above_trigger": below,
            "no_cry_clips": len(quiet_clips),
        },
        "per_clip": [
            {"file": f, "label": l, "rms_mean": rm, "rms_peak": rp}
            for f, l, rm, rp in per_clip
        ],
    }
    Path(args.report).write_text(json.dumps(report, indent=2))

    print(f"\nAUC (INT8, the deployed model): {auc:.4f}")
    print(f"at P >= {args.threshold}:")
    print(f"  INT8    acc {conf_int8['accuracy']:.3f}  "
          f"recall {conf_int8['recall']:.3f}  fpr {conf_int8['fpr']:.3f}  "
          f"(tp {conf_int8['tp']} fn {conf_int8['fn']} fp {conf_int8['fp']} "
          f"tn {conf_int8['tn']})")
    print(f"  float32 acc {conf_float['accuracy']:.3f}  "
          f"recall {conf_float['recall']:.3f}  fpr {conf_float['fpr']:.3f}")
    print(f"\ntrigger at RMS >= {args.trigger_rms}:")
    print(f"  {above}/{len(cry_clips)} cry clips would reach the model")
    print(f"  {below}/{len(quiet_clips)} no-cry clips would also reach it "
          f"(false triggers)")
    print(f"\nwrote {args.report}")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

research/eval_esc50.py

282 lines · 11,493 bytes

```
#!/usr/bin/env python3
"""The false-alarm map: what does the *deployed* model say about real speech,
real music and real household noise?

The deployed INT8 model is run over ESC-50 — 2,000 real-world recordings
across 50 categories (speech, music, domestic sounds, animals, human
non-speech, and 40 real "crying_baby" clips).  For every clip the script
reports:

  * the peak RMS and whether the clip would even reach the classifier
    (the hub only pulls a window when the node's 100 ms RMS reaches
    CRYNET_TRIGGER_RMS, 0.030),
  * the model's P(cry),
  * whether the model would have raised an alarm (P >= the deployed
    threshold, 0.95).

and then rolls those up per ESC-50 category and per coarse group, so the
answer to "can it tell a cry from music or speech?" is a table rather
than an opinion.

The front end is the notebook's own, and the model is the very file that
is flashed onto the hub, so the numbers are what the hub produces.

Caveat, stated plainly: ESC-50's non-cry clips were part of the model's
training corpus, so the false-alarm rates here are optimistic for those
categories.  What this buys is the *category map* — which kinds of sound
the model finds cry-like and which it does not — and the end-to-end
number of alarms a room full of clatter would produce.  The honest
in-room number comes from the confusion session recorded with the
INMP441 (research/in_domain/), and it is a separate measurement.

Note also what ESC-50 does *not* contain: it has no music and no
speech, only environmental sound and human non-speech vocalisations.
The model's behaviour on music is therefore measured separately, by
research/eval_music.py.

Usage:
    python research/eval_esc50.py                # every clip
    python research/eval_esc50.py --max-clips 300
"""

```

```
from __future__ import annotations

import argparse
import csv
import io
import json
import sys
import time
import zipfile
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parents[1]
OUT = ROOT / "research" / "out"
CACHE = ROOT / "research" / "_cache"
ESC50_ZIP = CACHE / "esc50__akaliutau_esc-50.zip"

sys.path.insert(0, str(ROOT / "research"))
from verify_int8_parity import notebook_namespace  # noqa: E402

# ESC-50's own five groups (the paper's taxonomy).  Forty clips each,
# except crying_baby which is the cry class.  Note what is *not* here:
# ESC-50 has no music and no speech — it is environmental sound plus
# human non-speech vocalisations.  Speech and music have to come from
# somewhere else.
GROUPS = {
    "animal": [
        "dog", "rooster", "crow", "pig", "cow", "frog", "hen", "cat",
        "sheep", "insects",
    ],
    "nature": [
        "rain", "thunderstorm", "sea_waves", "crackling_fire", "crickets",
        "chirping_birds", "water_drops", "wind", "pouring_water",
        "toilet_flush",
    ],
    "human_non_speech": [
        "crying", "sneezing", "clapping", "breathing", "coughing",
        "footsteps", "laughing", "drinking_sipping",
    ],
    "domestic": [
        "door_wood_creaks", "mouse_click", "keyboard_typing",
        "door_wood_knock", "washing_dishes", "washing_machine",
        "vacuum_cleaner", "clock_alarm", "clock_tick", "glass_breaking",
        "can_opening", "brushing_teeth",
    ],
    "urban": [
        "helicopter", "chainsaw", "siren", "car_horn", "engine", "train",
        "church_bells", "airplane", "fireworks", "hand_saw",
    ],
}

# The categories that are a cry of one kind or another, by ESC-50's own
# names.  ESC-50's `crying_baby` is a stock recording, not an infant cry
# from a cot, and `crying` is an adult sob; both are reported separately
# from the ordinary negatives.
CRY_CATEGORIES = {"crying_baby", "crying"}


def group_of(category: str) -> str:
    if category in ("crying_baby", "crying"):
        return "cry-ish"
    for name, members in GROUPS.items():
        if category in members:
            return name
    return "other"


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--max-clips", type=int, default=0)
    ap.add_argument("--windows-per-clip", type=int, default=2)
    ap.add_argument("--threshold", type=float, default=0.95)
    ap.add_argument("--trigger-rms", type=float, default=0.030)
    ap.add_argument("--model", default=str(OUT / "crynet_int8.tflite"),
                    help="the INT8 model to score; point it at a candidate "
                         "to judge a retrain before it is deployed")
    ap.add_argument("--report", default=str(OUT / "eval_esc50.json"))
    args = ap.parse_args()

    if not ESC50_ZIP.is_file():
        print(f"no ESC-50 archive in the cache: {ESC50_ZIP}")
        return 1

    t_start = time.time()
```

```
    ns = notebook_namespace()
    log_mel = ns["log_mel"]
    take_windows = ns["take_windows"]
    decode = ns["decode"]

    with zipfile.ZipFile(ESC50_ZIP) as z:
        meta_name = next(n for n in z.namelist() if n.endswith("meta/esc50.csv"))
        rows = list(csv.DictReader(io.TextIOWrapper(z.open(meta_name),
                                                    encoding="utf-8")))
        print(f"\nESC-50 manifest: {len(rows)} clips")

        # ---- features -----------------------------------------------------
        feats, clips, skipped = [], [], 0
        for row in rows:
            if args.max_clips and len(clips) >= args.max_clips:
                break
            member = f"ESC-50-master/audio/{row['filename']}"
            try:
                raw = z.read(member)
            except KeyError:
                skipped += 1
                continue
            x = decode(raw)
            if x is None:
                skipped += 1
                continue
            windows = take_windows(x, args.windows_per_clip, None, mode="center")
            first = len(feats)
            for w in windows:
                feats.append(log_mel(w))
            clips.append({
                "category": row["category"],
                "filename": row["filename"],
                "rms": float(np.sqrt(np.mean(x.astype(np.float64) ** 2))),
                "windows": list(range(first, len(feats))),
            })
            if len(clips) % 200 == 0:
                print(f"  {len(clips)} clips", flush=True)
    if skipped:
        print(f"warning: {skipped} clip(s) skipped")

    X = np.stack(feats).astype(np.float32)
    print(f"windows: {X.shape[0]}  ({time.time() - t_start:.0f}s of features)")

    # ---- the deployed INT8 model ------------------------------------------
    import tensorflow as tf  # noqa: E402

    interp = tf.lite.Interpreter(model_path=str(args.model))
    interp.allocate_tensors()
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]

    p = np.zeros(len(X), np.float32)
    for i in range(len(X)):
        q = np.clip(np.round(X[i] / s) + z, -128, 127).astype(np.int8)
        interp.set_tensor(i_d["index"], q[None, ..., None])
        interp.invoke()
        o = interp.get_tensor(o_d["index"]).astype(np.float32).reshape(-1)
        oq = (o - o_d["quantization"][1]) * o_d["quantization"][0]
        p[i] = float(np.asarray(oq).reshape(-1)[1])

    # ---- per clip -----------------------------------------------------------
    # A clip is judged by its most confident window: the hub classifies
    # 2 s windows, so one loud moment inside a 5 s clip is enough to make
    # it raise an alarm.
    per_clip = []
    for c in clips:
        ps = [float(p[i]) for i in c["windows"]]
        pmax = max(ps)
        per_clip.append({
            "category": c["category"],
            "group": group_of(c["category"]),
            "rms": c["rms"],
            "p_cry": pmax,
            "p_cry_mean": float(np.mean(ps)),
            "alarm": bool(pmax >= args.threshold),
            "reaches_model": bool(c["rms"] >= args.trigger_rms),
        })

    # ---- roll-ups -----------------------------------------------------------
    def rollup(keyfn, subset):
        out = {}
        for key in sorted({keyfn(c) for c in subset}):
            rows_k = [c for c in subset if keyfn(c) == key]
```

```
            reach = [c for c in rows_k if c["reaches_model"]]
            alarms = [c for c in reach if c["alarm"]]
            out[key] = {
                "clips": len(rows_k),
                "mean_p_cry": float(np.mean([c["p_cry"] for c in rows_k])),
                "max_p_cry": float(np.max([c["p_cry"] for c in rows_k])),
                "clips_reaching_model": len(reach),
                "alarms": len(alarms),
                "alarm_rate_over_reached": (len(alarms) / len(reach)
                                            if reach else 0.0),
                "alarm_rate_over_all": len(alarms) / len(rows_k) if rows_k else 0.0,
            }
        return out

    by_group = rollup(lambda c: c["group"], per_clip)
    by_category = rollup(lambda c: c["category"], per_clip)

    negatives = [c for c in per_clip if c["category"] not in CRY_CATEGORIES]
    n_reach = sum(1 for c in negatives if c["reaches_model"])
    n_alarm = sum(1 for c in negatives
                  if c["reaches_model"] and c["alarm"])

    report = {
        "source": "research/eval_esc50.py",
        "corpus": "ESC-50 (2,000 real-world clips, 50 categories)",
        "caveat": ("ESC-50's non-cry clips were part of the training corpus, "
                   "so these false-alarm rates are optimistic for those "
                   "categories; this is the category map, not the in-room "
                   "number"),
        "threshold": args.threshold,
        "trigger_rms": args.trigger_rms,
        "windows_per_clip": args.windows_per_clip,
        "windows": int(X.shape[0]),
        "clips": len(per_clip),
        "negatives": {
            "clips": len(negatives),
            "reaching_model": n_reach,
            "alarms": n_alarm,
            "alarm_rate_over_reached": (n_alarm / n_reach) if n_reach else 0.0,
        },
        "by_group": by_group,
        "by_category": by_category,
        "per_clip": per_clip,
    }
    Path(args.report).write_text(json.dumps(report, indent=2))

    # ---- print ------------------------------------------------------------
    print(f"\n=== ESC-50, by kind of sound ===")
    print(f"{'group':<18}{'clips':>7}{'reached':>9}{'alarms':>8}"
          f"{'alarm/reached':>15}{'max P(cry)':>12}")
    for g, r in by_group.items():
        print(f"{g:<18}{r['clips']:>7}{r['clips_reaching_model']:>9}"
              f"{r['alarms']:>8}{r['alarm_rate_over_reached']:>15.3f}"
              f"{r['max_p_cry']:>12.4f}")

    print(f"\n=== the ten worst categories by P(cry) ===")
    worst = sorted(by_category.items(), key=lambda kv: -kv[1]["max_p_cry"])[:10]
    for c, r in worst:
        print(f"{c:<24}{r['clips']:>5} clips  reached {r['clips_reaching_model']:>4}"
              f"  alarms {r['alarms']:>3}  max P {r['max_p_cry']:.4f}")

    print(f"\nnegatives: {n_alarm} alarms out of {n_reach} that reached the "
          f"model "
          f"({(n_alarm / n_reach * 100) if n_reach else 0:.2f} %)")
    print(f"wrote {args.report}")
    print(f"total {time.time() - t_start:.0f}s")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

research/eval_music.py

225 lines · 8,541 bytes

```
#!/usr/bin/env python3
"""Does the deployed model mistake music for a cry?

The training corpus has no music in it at all: ICD brings cries, VoxCeleb
speech, DASEE domestic noise, human non-speech, cat meows and silence;
the supplementary sets are cries, ESC-50 and snoring.  ESC-50 itself has
```

```
no music either.  So the model's behaviour on music was neither trained
nor measured — and a nursery with a radio or a television is exactly the
case the demo will contain.

This script measures it.  It runs the *deployed* INT8 model — the very
file that is flashed onto the hub — over a music corpus (GTZAN, ten
genres), through the notebook's own frozen front end, and reports, per
genre and overall:

  * the peak RMS of each clip and whether it would reach the classifier
    (the hub only pulls a window when the node's 100 ms RMS reaches
    CRYNET_TRIGGER_RMS, 0.030),
  * the model's P(cry),
  * whether the model would have raised an alarm (P >= the deployed
    threshold, 0.95).

The same numbers, for the same reason, are the ones the hub produces.

Usage:
    python research/eval_music.py
    python research/eval_music.py --max-clips 200
"""

from __future__ import annotations

import argparse
import json
import sys
import time
import zipfile
from collections import Counter
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parents[1]
OUT = ROOT / "research" / "out"
CACHE = ROOT / "research" / "_cache"

DEFAULT_ZIPS = [
    CACHE / "gtzan__carlthome_gtzan-genre-collection.zip",
]

sys.path.insert(0, str(ROOT / "research"))
from verify_int8_parity import notebook_namespace  # noqa: E402


def find_corpus() -> Path | None:
    for p in DEFAULT_ZIPS:
        if p.is_file():
            return p
    return None


def members_of(path: Path):
    """Every clip in the archive, with the genre read off the folder name.

    GTZAN ships as Sun/NeXT AU (libsndfile reads it), some copies as WAV.
    """
    if path.suffix == ".zip":
        with zipfile.ZipFile(path) as z:
            for name in z.namelist():
                if name.lower().endswith((".wav", ".au")):
                    genre = Path(name).parent.name.lower()
                    yield name, genre, z.read(name)
    else:
        import tarfile
        with tarfile.open(path) as t:
            for m in t.getmembers():
                if m.isfile() and m.name.lower().endswith((".wav", ".au")):
                    f = t.extractfile(m)
                    if f is None:
                        continue
                    yield m.name, Path(m.name).parent.name.lower(), f.read()


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--corpus", help="the archive to read; auto-detected if omitted")
    ap.add_argument("--max-clips", type=int, default=0)
    ap.add_argument("--windows-per-clip", type=int, default=2)
    ap.add_argument("--threshold", type=float, default=0.95)
    ap.add_argument("--trigger-rms", type=float, default=0.030)
    ap.add_argument("--model", default=str(OUT / "crynet_int8.tflite"),
                    help="the INT8 model to score; point it at a candidate "
```

```
                        "to judge a retrain before it is deployed")
    ap.add_argument("--report", default=str(OUT / "eval_music.json"))
    args = ap.parse_args()

    corpus = Path(args.corpus) if args.corpus else find_corpus()
    if corpus is None or not corpus.is_file():
        print(f"no music corpus found; looked for {DEFAULT_ZIPS}")
        return 1

    t_start = time.time()
    ns = notebook_namespace()
    log_mel = ns["log_mel"]
    take_windows = ns["take_windows"]
    decode = ns["decode"]

    feats, clips, skipped = [], [], 0
    for name, genre, raw in members_of(corpus):
        if args.max_clips and len(clips) >= args.max_clips:
            break
        x = decode(raw)
        if x is None:
            skipped += 1
            continue
        windows = take_windows(x, args.windows_per_clip, None, mode="center")
        first = len(feats)
        for w in windows:
            feats.append(log_mel(w))
        clips.append({
            "genre": genre,
            "file": Path(name).name,
            "rms": float(np.sqrt(np.mean(x.astype(np.float64) ** 2))),
            "windows": list(range(first, len(feats))),
        })
        if len(clips) % 100 == 0:
            print(f"  {len(clips)} clips", flush=True)
    if skipped:
        print(f"warning: {skipped} clip(s) skipped")

    X = np.stack(feats).astype(np.float32)
    print(f"clips: {len(clips)}  windows: {X.shape[0]}  "
          f"({time.time() - t_start:.0f}s of features)")

    import tensorflow as tf  # noqa: E402

    interp = tf.lite.Interpreter(model_path=str(args.model))
    interp.allocate_tensors()
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]

    p = np.zeros(len(X), np.float32)
    for i in range(len(X)):
        q = np.clip(np.round(X[i] / s) + z, -128, 127).astype(np.int8)
        interp.set_tensor(i_d["index"], q[None, ..., None])
        interp.invoke()
        o = interp.get_tensor(o_d["index"]).astype(np.float32).reshape(-1)
        oq = (o - o_d["quantization"][1]) * o_d["quantization"][0]
        p[i] = float(np.asarray(oq).reshape(-1)[1])

    per_clip = []
    for c in clips:
        ps = [float(p[i]) for i in c["windows"]]
        pmax = max(ps)
        per_clip.append({
            "genre": c["genre"],
            "file": c["file"],
            "rms": c["rms"],
            "p_cry": pmax,
            "p_cry_mean": float(np.mean(ps)),
            "alarm": bool(pmax >= args.threshold),
            "reaches_model": bool(c["rms"] >= args.trigger_rms),
        })

    def stats(rows):
        reach = [c for c in rows if c["reaches_model"]]
        alarms = [c for c in reach if c["alarm"]]
        return {
            "clips": len(rows),
            "mean_p_cry": float(np.mean([c["p_cry"] for c in rows])) if rows else 0.0,
            "max_p_cry": float(np.max([c["p_cry"] for c in rows])) if rows else 0.0,
            "clips_reaching_model": len(reach),
            "alarms": len(alarms),
            "alarm_rate_over_reached": (len(alarms) / len(reach) if reach else 0.0),
            "alarm_rate_over_all": (len(alarms) / len(rows) if rows else 0.0),
        }
```

```
    by_genre = {}
    for genre in sorted({c["genre"] for c in per_clip}):
        by_genre[genre] = stats([c for c in per_clip if c["genre"] == genre])

    overall = stats(per_clip)
    # the loudest tenth of the corpus — the music that would sit above the
    # hub's trigger most easily
    order = sorted(per_clip, key=lambda c: -c["rms"])
    loud = order[: max(1, len(order) // 10)]
    loud_stats = stats(loud)

    report = {
        "source": "research/eval_music.py",
        "corpus": corpus.name,
        "caveat": ("no music was in the training corpus at all, so these "
                   "numbers are the model's behaviour on music it has never "
                   "seen — neither trained nor tuned for it"),
        "threshold": args.threshold,
        "trigger_rms": args.trigger_rms,
        "windows_per_clip": args.windows_per_clip,
        "clips": len(per_clip),
        "windows": int(X.shape[0]),
        "overall": overall,
        "loudest_decile": loud_stats,
        "by_genre": by_genre,
        "per_clip": per_clip,
    }
    Path(args.report).write_text(json.dumps(report, indent=2))

    print(f"\n=== music, by genre ===")
    print(f"{'genre':<18}{'clips':>7}{'reached':>9}{'alarms':>8}"
          f"{'alarm/reached':>15}{'max P(cry)':>12}")
    for g, r in by_genre.items():
        print(f"{g:<18}{r['clips']:>7}{r['clips_reaching_model']:>9}"
              f"{r['alarms']:>8}{r['alarm_rate_over_reached']:>15.3f}"
              f"{r['max_p_cry']:>12.4f}")
    print(f"\noverall: {overall['alarms']} alarms out of "
          f"{overall['clips_reaching_model']} that reached the model "
          f"({overall['alarm_rate_over_reached'] * 100:.2f} %)")
    print(f"loudest tenth: {loud_stats['alarms']}/{loud_stats['clips_reaching_model']} "
          f"({loud_stats['alarm_rate_over_reached'] * 100:.2f} %)")
    print(f"wrote {args.report}")
    print(f"total {time.time() - t_start:.0f}s")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

research/eval_speech.py

324 lines · 13,510 bytes

```
#!/usr/bin/env python3
"""Does the deployed model mistake a room full of talking for a cry?

The corpus the model was trained on holds speech — 4,963 VoxCeleb clips
that came in with the ICD negatives — but **no loud speech**: the room
where the bench lives is quiet, and the one measured conversation
window scored 0.9727 (see SYSTEM_UPDATE.md), which is below the
saturation the hub announces on (0.9900) but not by much.

The user has nobody around to record a conversation, so the speech is
**sourced online** instead:

  * **FSD50K (16 kHz mirror)** — real-world recordings with class labels.
    The speech classes are the point: `Speech`, `Child_speech_and_kid_
    speaking`, `Conversation`, `Chatter`, `Crowd`, `Human_voice`,
    `Male_speech_and_man_speaking`, `Female_speech_and_woman_speaking`,
    `Laughter`, `Giggle`, `Singing`, `Human_group_actions`.  A nursery has
    all of these, and a child's voice is the closest thing to a baby's
    cry that a real room contains.
  * **Common Voice (16 kHz)** — real read speech from thousands of
    speakers with accents, recorded close to the microphone in quiet
    rooms: the closest thing to a person talking in the next room.

Both are read from the **eval / validation / test** partitions, which the
model has never seen, so these numbers are the model's behaviour on
speech it has not been trained on.

This script runs the *deployed* INT8 model — the very file flashed onto
```

```
the hub — through the notebook's own frozen front end and reports:

  * the peak RMS of each clip and whether it would reach the classifier
    (the hub pulls a window when the node's 100 ms RMS reaches 0.030),
  * the model's P(cry) on the loudest window of the clip,
  * whether the model would have raised an alarm at the deployed
    threshold (0.95) and at the hub's saturation (0.99).

Usage:
    python research/eval_speech.py
    python research/eval_speech.py --max-clips 400 --windows-per-clip 3
"""

from __future__ import annotations

import os

# One thread for the parquet reads: this machine runs the bench, the
# downloads and the model at the same time, and pyarrow's worker pool
# fails with "resource unavailable" when the machine is busy.
os.environ.setdefault("ARROW_NUM_THREADS", "1")

import argparse
import io
import json
import sys
import time
from collections import Counter
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parents[1]
OUT = ROOT / "research" / "out"
RAW = ROOT / "research" / "loudspeech" / "raw"

sys.path.insert(0, str(ROOT / "research"))
from verify_int8_parity import notebook_namespace  # noqa: E402

sys.path.insert(0, str(ROOT / "research" / "loudspeech"))
from speech_common import (AMBIGUOUS_CLASSES, FSD_MEASURE_PARTITIONS,  # noqa: E402
                           SPEECH_CLASSES, cv_half, cv_id)


def fsd50k_rows(partitions):
    """Every clip of the given FSD50K partitions, with its class labels."""
    import pyarrow.parquet as pq

    rows = []
    for shard in sorted((RAW / "fsd").rglob("*.parquet")):
        tbl = pq.read_table(shard, columns=["audio", "labels", "duration_seconds",
                                            "source_partition", "freesound_id", "title"],
                            use_threads=False)
        labels = tbl.column("labels").to_pylist()
        parts = tbl.column("source_partition").to_pylist()
        audio = tbl.column("audio").to_pylist()
        dur = tbl.column("duration_seconds").to_pylist()
        fids = tbl.column("freesound_id").to_pylist()
        titles = tbl.column("title").to_pylist()
        for i, ls in enumerate(labels):
            if parts[i] not in partitions:
                continue
            keep = [l for l in ls if l in SPEECH_CLASSES or l in AMBIGUOUS_CLASSES]
            if not keep:
                continue
            blob = audio[i]["bytes"]
            if blob is None:
                continue
            rows.append({
                "source": "fsd50k",
                "raw": blob,
                "group": keep[0] if len(keep) == 1 else "+".join(sorted(keep)),
                "cls": keep[0],
                "partition": parts[i],
                "id": f"fsd{int(fids[i])}",
                "duration": float(dur[i]),
                "title": titles[i],
            })
    return rows


def common_voice_rows():
    """Every clip of the Common Voice **measurement** half.

    The mirror's test split is halved by a hash of the clip id: the other
```

```
    half is what the training corpus is built from, so nothing measured
    here was ever trained on.
    """
    import pyarrow.parquet as pq

    rows = []
    for shard in sorted((RAW / "cv").rglob("*.parquet")):
        tbl = pq.read_table(shard, use_threads=False)
        cols = tbl.column_names
        audio_col = "audio" if "audio" in cols else cols[0]
        accent_col = "accent" if "accent" in cols else None
        audio = tbl.column(audio_col).to_pylist()
        accents = tbl.column(accent_col).to_pylist() if accent_col else [""] * len(audio)
        for i, a in enumerate(audio):
            blob = a["bytes"] if isinstance(a, dict) else a
            if blob is None:
                continue
            cid = cv_id(shard.name, i)
            if cv_half(cid) != "measure":
                continue
            acc = accents[i] or "unspecified"
            rows.append({
                "source": "common_voice",
                "raw": blob,
                "group": f"cv_{acc.lower().replace(' ', '_')}",
                "cls": "cv_speech",
                "partition": "measure",
                "id": cid,
                "duration": 0.0,
                "title": "",
            })
    return rows


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--max-clips", type=int, default=0, help="0 = all of them")
    ap.add_argument("--windows-per-clip", type=int, default=3)
    ap.add_argument("--threshold", type=float, default=0.95, help="the deployed threshold")
    ap.add_argument("--saturation", type=float, default=0.99, help="the hub's announcement rule")
    ap.add_argument("--trigger-rms", type=float, default=0.030)
    ap.add_argument("--model", default=str(OUT / "crynet_int8.tflite"),
                    help="the INT8 model to score; point it at a candidate "
                         "to judge a retrain before it is deployed")
    ap.add_argument("--report", default=str(OUT / "eval_speech.json"))
    args = ap.parse_args()

    t_start = time.time()
    ns = notebook_namespace()
    log_mel = ns["log_mel"]
    take_windows = ns["take_windows"]
    decode = ns["decode"]

    rows = []
    for part in sorted(FSD_MEASURE_PARTITIONS):
        rows += fsd50k_rows({part})
    print(f"FSD50K speech-ish clips: {len(rows)}", flush=True)
    rows += common_voice_rows()
    print(f"plus Common Voice: {len(rows)} clips in total", flush=True)

    if args.max_clips:
        rows = rows[: args.max_clips]

    import tensorflow as tf  # noqa: E402

    interp = tf.lite.Interpreter(model_path=str(args.model))
    interp.allocate_tensors()
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]
    o_scale, o_zp = o_d["quantization"]

    def score(w):
        """One 2 s window -> P(cry)."""
        q = np.clip(np.round(log_mel(w) / s) + z, -128, 127).astype(np.int8)
        interp.set_tensor(i_d["index"], q[None, ..., None])
        interp.invoke()
        o = interp.get_tensor(o_d["index"]).astype(np.float32).reshape(-1)
        return float(((o - o_zp) * o_scale).reshape(-1)[1])

    # Each window is scored the moment it is computed: the feature matrix
    # of a whole corpus does not fit in this machine's memory beside the
    # downloads, and the features are only needed once.
    clips, skipped, n_windows = [], 0, 0
    for r in rows:
```

```
        x = decode(r["raw"])
        if x is None:
            skipped += 1
            continue
        # the loudest window of the clip is the fairest test: a false alarm
        # is judged by the loudest moment of the speech, not a quiet one.
        # "center" with n windows starts them at the beginning, the middle
        # and the end of the clip.
        wins = take_windows(x, args.windows_per_clip, None, mode="center")
        ps = [score(w) for w in wins]
        n_windows += len(ps)
        clips.append({
            "source": r["source"],
            "group": r["group"],
            "cls": r["cls"],
            "id": r["id"],
            "partition": r["partition"],
            "rms": float(np.sqrt(np.mean(x.astype(np.float64) ** 2))),
            "peak": float(np.max(np.abs(x))),
            "clipped": bool(np.max(np.abs(x)) > 0.999),
            "p_cry": max(ps),
            "p_cry_mean": float(np.mean(ps)),
        })
        if len(clips) % 500 == 0:
            print(f"  {len(clips)} clips", flush=True)
    if skipped:
        print(f"warning: {skipped} clip(s) skipped")

    print(f"clips: {len(clips)}  windows: {n_windows}", flush=True)

    per_clip = []
    for c in clips:
        pmax = c["p_cry"]
        per_clip.append({
            **c,
            "alarm": bool(pmax >= args.threshold),
            "announces": bool(pmax >= args.saturation),
            "reaches_model": bool(c["rms"] >= args.trigger_rms),
            "ambiguous": c["cls"] in AMBIGUOUS_CLASSES,
        })

    def stats(rows_):
        reach = [c for c in rows_ if c["reaches_model"]]
        alarms = [c for c in reach if c["alarm"]]
        announce = [c for c in reach if c["announces"]]
        return {
            "clips": len(rows_),
            "mean_p_cry": float(np.mean([c["p_cry"] for c in rows_])) if rows_ else 0.0,
            "max_p_cry": float(np.max([c["p_cry"] for c in rows_])) if rows_ else 0.0,
            "clips_reaching_model": len(reach),
            "alarms_at_threshold": len(alarms),
            "announces_at_saturation": len(announce),
            "alarm_rate_over_reached": (len(alarms) / len(reach) if reach else 0.0),
            "announce_rate_over_reached": (len(announce) / len(reach) if reach else 0.0),
        }

    clear = [c for c in per_clip if not c["ambiguous"]]
    by_group = {}
    for g in sorted({c["group"] for c in clear}):
        by_group[g] = stats([c for c in clear if c["group"] == g])

    by_source = {}
    for s_ in sorted({c["source"] for c in clear}):
        by_source[s_] = stats([c for c in clear if c["source"] == s_])

    ambiguous = [c for c in per_clip if c["ambiguous"]]
    overall = stats(clear)
    order = sorted(clear, key=lambda c: -c["rms"])
    loud = order[: max(1, len(order) // 10)]

    report = {
        "source": "research/eval_speech.py",
        "corpora": {
            "fsd50k": "quinnlue/FSD50K-16k (eval + validation partitions, CC-BY)",
            "common_voice": "DTU54DL/common-voice-test16k (test split, MIT)",
        },
        "threshold": args.threshold,
        "saturation": args.saturation,
        "trigger_rms": args.trigger_rms,
        "windows_per_clip": args.windows_per_clip,
        "clips": len(per_clip),
        "windows": int(n_windows),
        "overall": overall,
        "by_source": by_source,
        "loudest_decile": stats(loud),
```

```
        "ambiguous_cry_adjacent": stats(ambiguous),
        "ambiguous_classes": sorted(AMBIGUOUS_CLASSES),
        "by_group": by_group,
        "per_clip": per_clip,
    }
    Path(args.report).write_text(json.dumps(report, indent=2))

    print(f"\n=== speech, by source ===")
    print(f"{'source':<16}{'clips':>7}{'reached':>9}{'alarms':>8}{'announces':>11}"
          f"{'alarm/reached':>15}{'max P(cry)':>12}")
    for s_, r in by_source.items():
        print(f"{s_:<16}{r['clips']:>7}{r['clips_reaching_model']:>9}"
              f"{r['alarms_at_threshold']:>8}{r['announces_at_saturation']:>11}"
              f"{r['alarm_rate_over_reached']:>15.3f}{r['max_p_cry']:>12.4f}")
    print(f"\noverall (clear negatives): {overall['alarms_at_threshold']} alarms "
          f"and {overall['announces_at_saturation']} announcements out of "
          f"{overall['clips_reaching_model']} clips that reached the model "
          f"({overall['alarm_rate_over_reached']*100:.2f} % / "
          f"{overall['announce_rate_over_reached']*100:.2f} %)")
    print(f"loudest tenth: {stats(loud)['alarms_at_threshold']}/"
          f"{stats(loud)['clips_reaching_model']} "
          f"({stats(loud)['alarm_rate_over_reached']*100:.2f} %)")
    if ambiguous:
        print(f"cry-adjacent (reported separately): "
              f"{stats(ambiguous)['clips']} clips, "
              f"{stats(ambiguous)['alarms_at_threshold']} alarms, "
              f"max P(cry) {stats(ambiguous)['max_p_cry']:.4f}")

    print(f"\n=== per group (the worst 15 by max P(cry)) ===")
    worst = sorted(clear, key=lambda c: -c["p_cry"])[:15]
    print(f"{'group':<44}{'rms':>9}{'P(cry)':>9}{'alarm':>7}")
    for c in worst:
        print(f"{c['group'][:43]:<44}{c['rms']:>9.4f}{c['p_cry']:>9.4f}"
              f"{str(c['alarm']):>7}")
    print(f"\nwrote {args.report}")
    print(f"total {time.time() - t_start:.0f}s")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

research/test_c_parity.py

140 lines · 5,083 bytes

```
"""Standalone validation of the notebook's C front end.

Executes the real notebook cells (config + frontend + export + C source) with
stubbed training outputs, compiles out/dsp_crynet.c and compares the C log-Mel
against the Python log-Mel on a synthetic probe.

    python test_c_parity.py
"""
import json
import os
import subprocess
import sys
import types
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parent.parent
NB = ROOT / "notebooks" / "crynet_training.ipynb"
WORK = ROOT / "research" / "_parity"
OUT = WORK / "out"
WORK.mkdir(parents=True, exist_ok=True)
OUT.mkdir(parents=True, exist_ok=True)

MARKERS = {
    "config": "CELL 1 — Environment",
    "frontend": "CELL 4 — The audio front end",
    "export": "CELL 19 — Export the deployment artifacts",
    "csource": "CELL 20 — The C reference implementation",
}

nb = json.loads(NB.read_text(encoding="utf-8"))
srcs = {}
for c in nb["cells"]:
    if c["cell_type"] != "code":
        continue
```

```
    s = "\n".join(c["source"])
    for k, m in MARKERS.items():
        if m in s:
            srcs[k] = s

missing = set(MARKERS) - set(srcs)
if missing:
    raise SystemExit(f"markers not found in notebook: {missing}")

ns = {"__name__": "__main__"}
import librosa  # noqa: E402  (the notebook imports it in its dependency cell)
ns["librosa"] = librosa

# the notebook's config cell resolves OUT/WORK from the environment, so point it
# at our sandbox before executing it
os.environ["CRYNET_OUT"] = str(OUT)
os.environ["CRYNET_WORK"] = str(WORK)
os.environ["CRYNET_CACHE"] = str(WORK / "cache")

# ---- stub the training outputs so the export cells run stand-alone ------------
ns["tflite_int8"] = bytes(range(256)) * 4
ns["IN_SCALE"], ns["IN_ZP"] = 0.012345678, 3
ns["OUT_SCALE"], ns["OUT_ZP"] = 0.00390625, -128
ns["THRESHOLD"] = 0.62
ns["BEST_KIND"] = "stub"


class _Model:
    def count_params(self):
        return 12345


ns["BEST_MODEL"] = _Model()
ns["res_val"] = {"auc": 0.9, "test_auc": 0.9, "cross_auc": 0.9, "f1": 0.9,
                 "test_f1": 0.9, "cross_f1": 0.9, "recall": 0.9, "precision": 0.9}
ns["res_test"] = dict(ns["res_val"])
ns["res_cross"] = dict(ns["res_val"])
ns["res_int8"] = dict(ns["res_val"])
ns["res_f32"] = dict(ns["res_val"])
ns["METRICS"] = {"stub": {"kind": "stub", "params": 12345, "val_auc": 0.9,
                          "test_auc": 0.9, "cross_auc": 0.9}}

ns["OUT"] = OUT
ns["WORK"] = WORK

for key in ("config", "frontend", "export", "csource"):
    print(f"\n>>> executing notebook cell: {key}")
    exec(compile(srcs[key], f"<{key}>", "exec"), ns)

N_SAMPLES = ns["N_SAMPLES"]
N_FRAMES = ns["N_FRAMES"]
N_MELS = ns["N_MELS"]
log_mel = ns["log_mel"]

# ---- compile the C ------------------------------------------------------------
CC = [sys.executable, "-m", "ziglang", "cc"]
exe = WORK / "dsp_test.exe"
cmd = CC + ["-O2", "-DCRYNET_SELFTEST", "-I", str(OUT), "-o", str(exe),
            str(OUT / "dsp_crynet.c"), "-lm"]
print("\n>>> compiling:", " ".join(cmd))
env = dict(os.environ)
env.setdefault("ZIG_GLOBAL_CACHE_DIR", str(WORK / "zigcache"))
env.setdefault("ZIG_LOCAL_CACHE_DIR", str(WORK / "zigcache_local"))
r = subprocess.run(cmd, capture_output=True, text=True, env=env)
if r.returncode != 0:
    print(r.stdout[-4000:])
    print(r.stderr[-4000:])
    raise SystemExit("COMPILE FAILED")
print("compiled OK")

# ---- parity on several probes -------------------------------------------------
rs = np.random.default_rng(7)
t = np.arange(N_SAMPLES) / ns["SR"]
probes = {
    "sine+harmonic": (0.35 * np.sin(2 * np.pi * 420 * t)
                      + 0.12 * np.sin(2 * np.pi * 1350 * t)
                      + 0.02 * rs.standard_normal(N_SAMPLES)),
    "noise": 0.05 * rs.standard_normal(N_SAMPLES),
    "dc_offset_cry": (0.3 * np.sin(2 * np.pi * 300 * t) + 0.05 * rs.standard_normal(N_SAMPLES)),
    "silence": np.zeros(N_SAMPLES),
}

worst = 0.0
for name, sig in probes.items():
    pcm16 = (np.clip(sig, -1, 1) * 32767).astype(np.int16)
```

```
    if name == "dc_offset_cry":
        pcm16 = (pcm16 + 18000).astype(np.int16)      # emulate the INMP441 bias
    raw = WORK / "probe.s16le"
    raw.write_bytes(pcm16.tobytes())
    r = subprocess.run([str(exe), str(raw)], capture_output=True, text=True, cwd=str(WORK))
    if r.returncode != 0:
        print(r.stderr[:800])
        raise SystemExit("parity binary failed")
    mel_c = np.fromfile(WORK / "mel_c.bin", dtype=np.float32).reshape(N_FRAMES, N_MELS)
    mel_py = log_mel(pcm16.astype(np.float32) / 32768.0)
    d = float(np.abs(mel_c - mel_py).max())
    worst = max(worst, d)
    print(f"  {name:16s} max|C-Python| = {d:.3e}   "
          f"mel range C [{mel_c.min():+.3f},{mel_c.max():+.3f}] "
          f"Py [{mel_py.min():+.3f},{mel_py.max():+.3f}]")

print(f"\nworst deviation over all probes: {worst:.3e}")
print("VERDICT:", "PASS — C front end matches training"
      if worst < 1e-2 else "FAIL — C and Python disagree")
```

research/verify_int8_parity.py

241 lines · 9,071 bytes

```
#!/usr/bin/env python3
"""Independently verify the deployed CryNet artifacts on the host.

The full Kaggle run reported `int8_val_auc` and `float_val_auc` as NaN,
because the parity cell took a *prefix* of the validation fold. The ICD
manifest lists every positive clip before every negative one, and the
validation fold's positive clips alone contribute more windows than the
4000-window cap, so that prefix contained only cry audio and its AUC was
undefined. The model itself is unaffected — every headline metric is
computed over the whole split — but the INT8-versus-float numbers for
these weights were never produced.

This script closes that gap without a 40-minute Kaggle re-run. It:

  * rebuilds the ICD validation fold (fold 4) straight from the cached
    archives and the shipped manifest;
  * extracts features with the notebook's own front end;
  * runs BOTH the exported INT8 TFLite model and the float32 Keras model
    over them;
  * reports the AUC of each, their difference, and the decision agreement
    at the calibrated threshold.

It deliberately uses its own centre-cropped windows rather than the
notebook's random ones, so the check is independent of that run.

Usage:  python research/verify_int8_parity.py [--max-clips N]
"""

from __future__ import annotations

import argparse
import json
import os
import sys
import time
from concurrent.futures import ThreadPoolExecutor
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parents[1]
NB = ROOT / "notebooks" / "crynet_training.ipynb"
OUT = ROOT / "research" / "out"
CACHE = ROOT / "research" / "_cache"
REPORT = OUT / "verify_int8_parity.json"

CELL_MARKERS = [
    "CELL 1 — Environment",
    "CELL 2 — Dependencies",
    "CELL 3 — Dataset acquisition",
    "CELL 4 — The audio front end",
    "CELL 5 — Archive readers",
    "CELL 6 — Corpus manifests",
]

VAL_FOLD = "4"


```

```
def notebook_namespace():
    nb = json.loads(NB.read_text(encoding="utf-8"))
    srcs = {}
    for cell in nb["cells"]:
        if cell.get("cell_type") != "code":
            continue
        text = "\n".join(cell["source"])
        for marker in CELL_MARKERS:
            if marker in text:
                srcs[marker] = text

    missing = [m for m in CELL_MARKERS if m not in srcs]
    if missing:
        raise SystemExit(f"markers not found in the notebook: {missing}")

    import librosa  # noqa: E402  (the notebook imports it in its dependency cell)

    ns = {"__name__": "__main__", "librosa": librosa, "np": np}

    # The configuration cell resolves its directories from the environment.
    os.environ["CRYNET_CACHE"] = str(CACHE)
    os.environ["CRYNET_WORK"] = str(ROOT / "research" / "_verify_work")
    os.environ["CRYNET_OUT"] = str(OUT)
    for sub in ("_verify_work",):
        (ROOT / "research" / sub).mkdir(parents=True, exist_ok=True)

    for marker in CELL_MARKERS:
        print(f">>> {marker}")
        exec(compile(srcs[marker], f"<{marker}>", "exec"), ns)
    return ns


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--max-clips", type=int, default=0,
                    help="limit the number of clips (0 = the whole fold)")
    args = ap.parse_args()

    t_start = time.time()
    ns = notebook_namespace()

    log_mel = ns["log_mel"]
    take_windows = ns["take_windows"]
    decode = ns["decode"]
    READERS = ns["READERS"]
    icd_entries = ns["icd_entries"]

    val_entries = [e for e in icd_entries if str(e[3]) == VAL_FOLD]
    if args.max_clips:
        val_entries = val_entries[: args.max_clips]
    n_cry = sum(e[1] for e in val_entries)
    print(f"\nvalidation fold {VAL_FOLD}: {len(val_entries)} clips "
          f"({n_cry} cry, {len(val_entries) - n_cry} non-cry)")

    # ---- feature extraction ------------------------------------------------
    failures = {"missing": 0, "decode": 0, "ok": 0}

    def features_for(entry):
        stem, label, sub, fold, key = entry
        reader = READERS[key]
        member = reader.index_by_stem().get(stem)
        if member is None:
            failures["missing"] += 1
            return None
        try:
            x = decode(reader.read(member))
        except Exception:
            x = None
        if x is None:
            failures["decode"] += 1
            return None
        feats = [log_mel(w) for w in take_windows(x, 2, None, mode="center")]
        failures["ok"] += 1
        return np.stack(feats).astype(np.float32), label

    t0 = time.time()
    X, y = [], []
    with ThreadPoolExecutor(max_workers=8) as ex:
        for res in ex.map(features_for, val_entries):
            if res is None:
                continue
            feats, label = res
            X.append(feats)
            y.extend([label] * feats.shape[0])

```

```
    if not X:
        raise SystemExit(
            f"no features were extracted: {failures['missing']} clips were not "
            f"found in the archives and {failures['decode']} failed to decode. "
            f"Check that the cached archives match the manifest."
        )
    X = np.concatenate(X, axis=0)
    y = np.asarray(y, dtype=np.int32)
    print(f"features: {X.shape}  ({time.time() - t0:.0f}s)  "
          f"[ok {failures['ok']}, missing {failures['missing']}, "
          f"undecodable {failures['decode']}]")

    # ---- models ------------------------------------------------------------
    import tensorflow as tf  # noqa: E402

    threshold = float(ns.get("THRESHOLD", 0.95))

    interp = tf.lite.Interpreter(model_path=str(OUT / "crynet_int8.tflite"))
    interp.allocate_tensors()
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]

    p_int8 = np.zeros((len(X), 2), np.float32)
    for i in range(len(X)):
        q = np.clip(np.round(X[i].astype(np.float32) / s) + z, -128, 127).astype(np.int8)
        interp.set_tensor(i_d["index"], q[None, ..., None])
        interp.invoke()
        o = interp.get_tensor(o_d["index"])[0].astype(np.float32)
        p_int8[i] = (o - o_d["quantization"][1]) * o_d["quantization"][0]

    keras_model = tf.keras.models.load_model(str(OUT / "crynet.keras"))
    p_float = keras_model.predict(X, verbose=0).astype(np.float32)

    # ---- comparison --------------------------------------------------------
    from sklearn.metrics import roc_auc_score  # noqa: E402

    a_int8 = float(roc_auc_score(y, p_int8[:, 1]))
    a_float = float(roc_auc_score(y, p_float[:, 1]))
    delta_p = np.abs(p_int8[:, 1] - p_float[:, 1])
    agree = float(((p_int8[:, 1] >= threshold) == (p_float[:, 1] >= threshold)).mean())

    def at_threshold(p):
        yh = (p >= threshold).astype(int)
        tp = int(((yh == 1) & (y == 1)).sum())
        fn = int(((yh == 0) & (y == 1)).sum())
        fp = int(((yh == 1) & (y == 0)).sum())
        tn = int(((yh == 0) & (y == 0)).sum())
        recall = tp / max(tp + fn, 1)
        fpr = fp / max(fp + tn, 1)
        return {"tp": tp, "fn": fn, "fp": fp, "tn": tn,
                "recall": recall, "fpr": fpr,
                "precision": tp / max(tp + fp, 1)}

    m_int8 = at_threshold(p_int8[:, 1])
    m_float = at_threshold(p_float[:, 1])

    report = {
        "source": "research/verify_int8_parity.py",
        "model_sha256_note": "the SHA-256 of crynet_int8.tflite is checked by "
                             "firmware/tools/check_artifact_drift.py",
        "fold": VAL_FOLD,
        "clips": int(len(val_entries)),
        "windows": int(len(X)),
        "cry_windows": int(y.sum()),
        "threshold": threshold,
        "auc_int8": a_int8,
        "auc_float32": a_float,
        "auc_delta": a_int8 - a_float,
        "max_abs_delta_p": float(delta_p.max()),
        "mean_abs_delta_p": float(delta_p.mean()),
        "decision_agreement": agree,
        "int8_at_threshold": m_int8,
        "float32_at_threshold": m_float,
        "elapsed_s": time.time() - t_start,
    }

    OUT.mkdir(parents=True, exist_ok=True)
    REPORT.write_text(json.dumps(report, indent=2), encoding="utf-8")

    print("\n" + "=" * 66)
    print("INT8 vs float32 on the ICD validation fold")
    print("=" * 66)
    print(f"  windows              : {len(X)} ({int(y.sum())} cry)")
    print(f"  AUC   int8           : {a_int8:.6f}")
```

```
    print(f"  AUC   float32        : {a_float:.6f}")
    print(f"  AUC   delta          : {a_int8 - a_float:+.6f}")
    print(f"  max |delta p|        : {delta_p.max():.6f}")
    print(f"  mean |delta p|       : {delta_p.mean():.6f}")
    print(f"  decision agreement   : {agree:.4%}")
    print(f"  int8    recall {m_int8['recall']:.4f}  FPR {m_int8['fpr']:.4f}")
    print(f"  float32 recall {m_float['recall']:.4f}  FPR {m_float['fpr']:.4f}")
    print(f"\n  wrote {REPORT.relative_to(ROOT)}")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

research/adaptive_trigger_study.py

205 lines · 7,221 bytes

```
#!/usr/bin/env python3
"""Choose the adaptive trigger's multiple from the recorded sessions.

The hub's trigger is a fixed RMS level (0.030), and the in-room confusion
session (D-60) found the consequence: a cry played at one metre from the
node reaches a median RMS of 0.0070 and never crosses 0.030, so it is never
offered to the model at all.

The remedy is an adaptive trigger: the hub watches each node's own noise
floor and triggers at a multiple of it.  The multiple is a number, and this
script picks it from the data already on disk rather than by taste.

For every recorded session it reports the RMS distribution, and for a grid
of thresholds it reports how many *trigger events* the fixed rule would
produce - an event being two consecutive readings above the level, which is
what the firmware counts (`loud_streak >= 2`).

The two numbers to compare are:

  * the number of trigger events in a non-cry session (the false alarms the
    model would have been asked about), and
  * the number of trigger events in a cry session (the chances the model
    would have been given the cry at all).

Usage:
    python research/adaptive_trigger_study.py
"""

from __future__ import annotations

import csv
import sys
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
SESSIONS = ROOT / "research" / "in_domain"

# What each session was, from the session notes.
TRUTH = {
    "session5": "quiet room",
    "session6": "speech, desk distance",
    "session6b": "speech, 30 cm",
    "session7": "music, 1 m",
    "session7b": "music, 30 cm",
    "session8": "film, 1 m",
    "session8b": "film, 30 cm",
    "session9": "phone cry, 15 cm",
    "session10": "phone cry, 1 m",
}
CRY_SESSIONS = {"session9", "session10"}


def read_rms(name: str) -> list[float]:
    path = SESSIONS / name / "records.csv"
    if not path.is_file():
        return []
    out = []
    with path.open(newline="", encoding="utf-8") as fh:
        for row in csv.DictReader(fh):
            try:
                out.append(float(row["rms"]))
            except (KeyError, ValueError):
                continue
    return out
```

```

def percentile(values: list[float], q: float) -> float:
    if not values:
        return float("nan")
    s = sorted(values)
    k = (len(s) - 1) * q
    lo = int(k)
    hi = min(lo + 1, len(s) - 1)
    return s[lo] + (s[hi] - s[lo]) * (k - lo)


def trigger_events(values: list[float], level: float) -> int:
    """How many times the firmware's rule would fire: two consecutive
    readings at or above the level start an event, and the event ends when
    a reading falls back below it."""
    events = 0
    streak = 0
    for v in values:
        if v >= level:
            streak += 1
            if streak == 2:
                events += 1
        else:
            streak = 0
    return events


def simulate(values: list[float], mult: float, rise: float,
             lo: float, hi: float) -> tuple[int, float]:
    """Run the adaptive rule over a session's readings.

    The floor tracker is the one the firmware will use: it falls at once
    when the room gets quieter and rises very slowly, so that a long cry
    does not drag the floor up to the cry's own level and switch the
    trigger off in the middle of an episode.

    Returns the number of trigger events and the median trigger level the
    session ran at.
    """
    if not values:
        return 0, 0.0
    floor = values[0]
    events = 0
    streak = 0
    levels = []
    for v in values:
        if v < floor:
            floor = v
        else:
            floor += (v - floor) * rise
        level = max(lo, min(hi, mult * floor))
        levels.append(level)
        if v >= level:
            streak += 1
            if streak == 2:
                events += 1
        else:
            streak = 0
    return events, percentile(levels, 0.50)


def main() -> int:
    names = [n for n in sorted(TRUTH) if (SESSIONS / n / "records.csv").is_file()]
    if not names:
        print("no recorded sessions found", file=sys.stderr)
        return 1

    data = {n: read_rms(n) for n in names}

    print("=== the RMS of every recorded session (the node's own telemetry)\n")
    print(f"{'session':<10} {'what it was':<24} {'n':>5} {'p10':>8} {'p25':>8} "
          f"{'median':>8} {'p90':>8} {'max':>8}")
    for n in names:
        v = data[n]
        print(f"{n:<10} {TRUTH[n]:<24} {len(v):>5} "
              f"{percentile(v, 0.10):>8.4f} {percentile(v, 0.25):>8.4f} "
              f"{percentile(v, 0.50):>8.4f} {percentile(v, 0.90):>8.4f} "
              f"{max(v):>8.4f}")

    # The quiet room is the reference: it is the floor the adaptive rule
    # would measure.
    quiet = data.get("session5") or []
    floor = percentile(quiet, 0.10) if quiet else 0.002

```

```
    print(f"\n=== the noise floor of the quiet room: {floor:.4f} (p10)\n")
    print("The adaptive rule would trigger at 3x that, i.e. "
          f"{3 * floor:.4f}.\n")

    levels = [0.004, 0.005, 0.006, 0.007, 0.008, 0.010, 0.012, 0.015,
              0.020, 0.025, 0.030]

    print("=== trigger events per session at each level "
          "(two consecutive readings above it)\n")
    header = f"{'level':>7} " + " ".join(f"{n.replace('session', 's'):>6}"
                                          for n in names)
    print(header)
    for lv in levels:
        row = f"{lv:>7.3f} "
        for n in names:
            row += f"{trigger_events(data[n], lv):>7} "
        print(row)

    print("\n=== the same, as the number of non-cry events and cry events\n")
    print(f"{'level':>7} {'non-cry events':>16} {'cry events':>12} "
          f"{'cry s9':>8} {'cry s10':>8}")
    for lv in levels:
        noncry = sum(trigger_events(data[n], lv) for n in names
                     if n not in CRY_SESSIONS)
        cry = sum(trigger_events(data[n], lv) for n in names
                  if n in CRY_SESSIONS)
        s9 = trigger_events(data.get("session9", []), lv)
        s10 = trigger_events(data.get("session10", []), lv)
        print(f"{lv:>7.3f} {noncry:>16} {cry:>12} {s9:>8} {s10:>8}")

    # The adaptive rule, simulated over each session's own readings.
    print("\n=== the adaptive rule, simulated over each session "
          "(the floor tracker, then the trigger)\n")
    for rise in (0.002, 0.005):
        print(f"--- the floor rises by {rise:.3f} of the way to the reading "
              f"each time (a {1.0 / (rise * 10):.0f} s time constant)")
        for mult in (2.0, 3.0, 4.0, 5.0):
            row = f"  {mult:.0f}x floor: "
            noncry = 0
            cry = 0
            for n in names:
                ev, lv = simulate(data[n], mult, rise, 0.005, 0.030)
                row += f"{n.replace('session', 's')}={ev:>3}"
                if n in CRY_SESSIONS:
                    cry += ev
                else:
                    noncry += ev
            row += f"   non-cry {noncry:>4}  cry {cry:>4}"
            print(row)
        print()

    return 0


if __name__ == "__main__":
    sys.exit(main())
```

research/analyze_gates.py

111 lines · 4,239 bytes

```
#!/usr/bin/env python3
"""Which gate is stopping the hub from pulling a cot's window?

The hub pulls a window when six conditions all hold at once.  This reads
the hub's log and, for every *distinct* window a cot offered, works out
which condition was the one that failed.  The answer is the difference
between "the radio was busy" and "the hub never wanted this window".
"""

from __future__ import annotations

import re
import sys
from collections import Counter
from pathlib import Path

TEL = re.compile(
    r"node (\d+): rms ([\d.]+)\s+seq (\d+)\s+flags 0x([0-9A-F]+)\s+crc 0x([0-9A-F]+)"
    r"\s+level ([\d.]+)\s+streak (\d+)\s+win ([\d.]+)")
PULL = re.compile(r"node (\d+): asking for the window \(seq (\d+)")
WINDOW = re.compile(r"node (\d+): window rms ([\d.]+)\s+mean (-?\d+)\s+peak (\d+)\s+crest ([\d.]+)")
```

```

FLAG_BURST_READY = 0x01
FLAG_EVENT = 0x02
FLAG_WINDOW_FULL = 0x04

PULL_GATE = 0.8 * 0.020     # worth_pulling
ANNOUNCE_MIN = 0.020        # CRYNET_MAYBE_MIN_RMS
QUIET_CREST = 3.5           # CRYNET_QUIET_CREST


def main() -> int:
    path = Path(sys.argv[1] if len(sys.argv) > 1 else "/tmp/hub_now.log")
    text = path.read_text(errors="replace")

    # The last telemetry line for each (node, seq) — that is the state
    # the cot was actually offering when the hub decided.
    offered: dict[tuple[int, int], dict] = {}
    order: list[tuple[int, int]] = []
    pulled: set[tuple[int, int]] = set()
    windows: dict[tuple[int, int], dict] = {}

    for line in text.splitlines():
        m = TEL.search(line)
        if m:
            key = (int(m.group(1)), int(m.group(3)))
            if key not in offered:
                order.append(key)
            offered[key] = {
                "rms": float(m.group(2)), "flags": int(m.group(4), 16),
                "level": float(m.group(6)), "streak": int(m.group(7)),
                "win": float(m.group(8)),
            }
            continue
        m = PULL.search(line)
        if m:
            pulled.add((int(m.group(1)), int(m.group(2))))
            continue
        m = WINDOW.search(line)
        if m:
            windows[(int(m.group(1)), 0)] = {
                "rms": float(m.group(2)), "crest": float(m.group(5))}

    print(f"{path}")
    print(f"distinct windows offered: {len(offered)}   pulled: {len(pulled)}")
    print()

    blockers = Counter()
    for key in order:
        if key in pulled:
            blockers["pulled"] += 1
            continue
        o = offered[key]
        why = []
        if not (o["flags"] & FLAG_BURST_READY):
            why.append("no frozen window")
        if not (o["flags"] & FLAG_WINDOW_FULL):
            why.append("live window incomplete")
        if o["win"] < PULL_GATE:
            why.append(f"frozen level {o['win']:.4f} < {PULL_GATE:.3f}")
        if not ((o["streak"] >= 2) or (o["rms"] >= 3.0 * o["level"])):
            why.append("not live-loud")
        blockers[", ".join(why) or "eligible (radio busy or starved)"] += 1

    print("why each offered window was not pulled:")
    for why, n in blockers.most_common():
        print(f"  {n:5d}  {why}")
    print()

    # The waste: windows the hub DID pull that the announce rule could
    # never have announced.
    pulled_keys = [k for k in order if k in pulled]
    print(f"the pull gate is {PULL_GATE:.3f}; a window needs {ANNOUNCE_MIN:.3f} rms")
    print(f"(or crest {QUIET_CREST}) to be announced:")
    waste = [k for k in pulled_keys if offered[k]["win"] < ANNOUNCE_MIN]
    print(f"  windows pulled below the announce floor: {len(waste)} of {len(pulled_keys)}")
    for k in waste:
        print(f"    node {k[0]} seq {k[1]}: frozen level {offered[k]['win']:.4f}")
    print()

    # What the hub actually measured on the wire.
    print("the window as the hub measured it:")
    for (node, _), w in windows.items():
        announce = (w["rms"] >= ANNOUNCE_MIN) or (w["crest"] >= QUIET_CREST)
        print(f"  node {node}: rms {w['rms']:.4f}  crest {w['crest']:.1f}  "
              f"-> {'announceable' if announce else 'REFUSED (too quiet)'}")
```

```
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

research/screen_clips.py

249 lines · 9,495 bytes

```
#!/usr/bin/env python3
"""Which recorded clips will the deployed model actually announce?

The demo failed on 2026-11-03 because the clip played was one the model
does not accept: it scored P(cry)=0.148 on its own, clean, and the
model is judged per 2 s window, so a clip is only as good as its best
window.

This screens every clip in the in-domain sessions against the *deployed*
INT8 model with the *notebook's own* front end, so the numbers here are
the numbers the hub will print.  For each clip it reports, per 2 s
window, the model's P(cry) as it stands and with the low-frequency
rumble taken out (a 100 Hz high-pass), plus the level and crest the
hub's gates will see.

Output:
    research/out/clip_screening.csv    one row per window
    research/out/clip_screening.md     the ranked, human-usable summary

Usage:
    python research/screen_clips.py
    python research/screen_clips.py --sessions session9 session10
"""

from __future__ import annotations

import argparse
import csv
import sys
import wave
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parents[1]
OUT = ROOT / "research" / "out"
CLIPS = ROOT / "research" / "in_domain"

# The notebook's frozen front end (build_notebook.py, CELL 1 and CELL 4).
SR = 16000
WIN_SEC = 2.0
N_SAMPLES = int(SR * WIN_SEC)
N_FFT = 512
HOP = 400
N_MELS = 64
FMIN = 20.0
FMAX = 8000.0
CENTER_PAD = 256
TAIL_PAD = 512
DB_FLOOR = -80.0
LOG_EPS = 1e-10
CMVN = True

ANNOUNCE = 0.99       # one window may announce on its own
MAYBE = 0.95          # a maybe must be confirmed by the next window
NEAR = 0.60           # "the model can see it but will not announce it"


def log_mel(x: np.ndarray) -> np.ndarray:
    from numpy.lib.stride_tricks import sliding_window_view

    x = np.asarray(x, dtype=np.float32)
    assert x.shape[0] == N_SAMPLES, x.shape
    x = x - x.mean()
    xp = np.concatenate([np.zeros(CENTER_PAD, np.float32), x,
                         np.zeros(TAIL_PAD, np.float32)])
    frames = sliding_window_view(xp, N_FFT)[::HOP]
    hann = np.hanning(N_FFT).astype(np.float32)
    spec = np.fft.rfft(frames * hann, axis=-1)
    power = spec.real ** 2 + spec.imag ** 2
    mel = power @ MEL_BANK.T
    db = 10.0 * np.log10(np.maximum(mel, LOG_EPS))
```

```
    db = np.maximum(db, DB_FLOOR)
    if CMVN:
        db = (db - db.mean()) / (db.std() + 1e-6)
    return db.astype(np.float32)


def high_pass(x: np.ndarray, cutoff: float = 100.0) -> np.ndarray:
    from scipy.signal import firwin, lfilter

    taps = firwin(513, cutoff, pass_zero=False, fs=SR)
    return lfilter(taps, 1.0, x)


def read_wav(path: Path) -> np.ndarray:
    with wave.open(str(path), "rb") as f:
        if f.getframerate() != SR or f.getnchannels() != 1:
            raise ValueError(f"{path}: {f.getframerate()} Hz, {f.getnchannels()} ch")
        raw = f.readframes(f.getnframes())
    return np.frombuffer(raw, dtype="<i2").astype(np.float32) / 32768.0


def build_scorer():
    import librosa
    import tensorflow as tf

    global MEL_BANK
    MEL_BANK = librosa.filters.mel(sr=SR, n_fft=N_FFT, n_mels=N_MELS,
                                   fmin=FMIN, fmax=FMAX, htk=False,
                                   norm="slaney").astype(np.float32)

    interp = tf.lite.Interpreter(model_path=str(OUT / "crynet_int8.tflite"))
    interp.allocate_tensors()
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]

    def score(sig: np.ndarray) -> float:
        f = log_mel(sig.astype(np.float32))[None, ...]
        q = np.clip(np.round(f / s) + z, -128, 127).astype(np.int8)
        interp.set_tensor(i_d["index"], q[..., None])
        interp.invoke()
        o = interp.get_tensor(o_d["index"])[0].astype(np.float32)
        p = (o - o_d["quantization"][1]) * o_d["quantization"][0]
        return float(np.asarray(p).reshape(-1)[1])

    return score


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--sessions", nargs="*", default=None,
                    help="only these session directories")
    ap.add_argument("--hop-sec", type=float, default=1.0,
                    help="how far the 2 s analysis window slides along a clip")
    args = ap.parse_args()

    score = build_scorer()

    sessions = sorted(p for p in CLIPS.glob("session*") if p.is_dir())
    if args.sessions:
        wanted = set(args.sessions)
        sessions = [p for p in sessions if p.name in wanted]

    rows = []
    for session in sessions:
        for clip in sorted((session / "clips").glob("*.wav")):
            x = read_wav(clip)
            if len(x) < N_SAMPLES:
                print(f"{clip}: too short ({len(x) / SR:.2f} s) — skipped",
                      file=sys.stderr)
                continue
            step = max(1, int(args.hop_sec * SR))
            t = 0
            wi = 0
            while t + N_SAMPLES <= len(x):
                w = x[t:t + N_SAMPLES]
                centred = w - w.mean()
                rms = float(np.sqrt((centred ** 2).mean()))
                peak = float(np.abs(w).max())
                crest = peak / (rms or 1.0)
                p = score(centred)
                p_hp = score(high_pass(centred))
                rows.append({
                    "session": session.name,
                    "clip": clip.name,
```

```
                    "window": wi,
                    "start_sec": round(t / SR, 2),
                    "rms": round(rms, 6),
                    "peak": round(peak, 6),
                    "crest": round(crest, 2),
                    "p_cry": round(p, 4),
                    "p_cry_highpassed": round(p_hp, 4),
                    "hub_would_announce": int(p >= ANNOUNCE),
                })
                wi += 1
                t += step

    OUT.mkdir(parents=True, exist_ok=True)
    csv_path = OUT / "clip_screening.csv"
    with csv_path.open("w", newline="", encoding="utf-8") as f:
        w = csv.DictWriter(f, fieldnames=list(rows[0].keys()))
        w.writeheader()
        w.writerows(rows)

    # ---- the ranked summary -------------------------------------------------
    by_clip: dict[tuple[str, str], list[dict]] = {}
    for r in rows:
        by_clip.setdefault((r["session"], r["clip"]), []).append(r)

    def best(r: dict) -> float:
        return max(r["p_cry"], r["p_cry_highpassed"])

    ranked = []
    for (session, clip), rs in by_clip.items():
        top = max(rs, key=best)
        ranked.append({
            "session": session,
            "clip": clip,
            "windows": len(rs),
            "best_p": max(r["p_cry"] for r in rs),
            "best_p_hp": max(r["p_cry_highpassed"] for r in rs),
            "best_p_any": best(top),
            "n_announce": sum(1 for r in rs if r["p_cry"] >= ANNOUNCE),
            "n_maybe": sum(1 for r in rs if MAYBE <= r["p_cry"] < ANNOUNCE),
            "rms": top["rms"],
            "crest": top["crest"],
        })
    ranked.sort(key=lambda d: -d["best_p_any"])

    md = OUT / "clip_screening.md"
    lines = [
        "# Clip screening — what the deployed model will announce",
        "",
        "Scored with the deployed INT8 model and the notebook's own front end.",
        "A clip is **demo-ready** when one of its 2 s windows reaches "
        f"P(cry) >= {ANNOUNCE} — that is the hub's single-window announce bar.",
        "",
        f"| session | clip | windows | best P | best P (100 Hz HP) | announce | maybe | rms | crest |",
        "|---|---|---|---|---|---|---|---|---|",
    ]
    for d in ranked:
        mark = "**" if d["n_announce"] else ""
        lines.append(
            f"| {d['session']} | {mark}{d['clip']}{mark} | {d['windows']} | "
            f"{d['best_p']:.4f} | {d['best_p_hp']:.4f} | {d['n_announce']} | "
            f"{d['n_maybe']} | {d['rms']:.4f} | {d['crest']:.1f} |")
    lines.append("")
    ready = [d for d in ranked if d["n_announce"]]
    near = [d for d in ranked if not d["n_announce"] and d["best_p_any"] >= NEAR]
    lines.append(f"## demo-ready clips (P >= {ANNOUNCE}): {len(ready)} of {len(ranked)}")
    for d in ready:
        lines.append(f"- `{d['session']}/clips/{d['clip']}` — P {d['best_p']:.4f}, "
                     f"{d['n_announce']} announcing window(s)")
    lines.append("")
    lines.append(f"## seen but not announceable (P >= {NEAR} only): {len(near)}")
    for d in near:
        lines.append(f"- `{d['session']}/clips/{d['clip']}` — P {d['best_p']:.4f} "
                     f"(HP {d['best_p_hp']:.4f})")
    md.write_text("\n".join(lines) + "\n", encoding="utf-8")

    print(f"windows scored : {len(rows)}")
    print(f"clips          : {len(ranked)}")
    print(f"demo-ready     : {len(ready)} (P >= {ANNOUNCE} in at least one window)")
    print(f"near-miss      : {len(near)} (P >= {NEAR} but < {ANNOUNCE})")
    print(f"wrote {csv_path}")
    print(f"wrote {md}")
    print()
    for d in ranked[:25]:
        flag = "ANNOUNCE" if d["n_announce"] else ("maybe   " if d["n_maybe"] else "        ")
        print(f"  {flag}  {d['session']:<12} {d['clip']:<16} "
```

```
            f"P {d['best_p']:.4f}  HP {d['best_p_hp']:.4f}  "
            f"rms {d['rms']:.4f}  crest {d['crest']:.1f}")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

research/score_demo_clips.py

76 lines · 2,711 bytes

```
#!/usr/bin/env python3
"""Score the demo kit's clips with the deployed INT8 model on the host.

The bench run announced nothing.  This asks the question that decides
where the fault is: does the deployed model, on the host, still call
these clips a cry?  If it does, the clip is good and the fault is in
the link; if it does not, the clip itself is not what the model was
trained on.

Every two-second window of the clip is scored, the way the hub scores
the window it pulls off the air.
"""
from __future__ import annotations

import os
import sys
import wave

import numpy as np

ROOT = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
OUT = os.path.join(ROOT, "research", "out")
SR = 16000
N = 32000  # two seconds at 16 kHz, the node's window


def read_wav(path: str) -> np.ndarray:
    with wave.open(path, "rb") as f:
        raw = f.readframes(f.getnframes())
    return np.frombuffer(raw, dtype="<i2").astype(np.float32) / 32768.0


def main(argv: list[str]) -> int:
    paths = argv[1:] or [os.path.join(ROOT, "research", "out", "demo_kit",
                                      "session9_clip_0005.wav")]
    sys.path.insert(0, os.path.join(ROOT, "research"))
    from eval_indomain import notebook_namespace  # type: ignore
    import tensorflow as tf

    ns = notebook_namespace()
    log_mel = ns["log_mel"]
    interp = tf.lite.Interpreter(model_path=os.path.join(OUT, "crynet_int8.tflite"))
    interp.allocate_tensors()
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]

    def score(sig: np.ndarray) -> float:
        f = log_mel(sig.astype(np.float32))[None, ...]
        q = np.clip(np.round(f / s) + z, -128, 127).astype(np.int8)
        interp.set_tensor(i_d["index"], q[..., None])
        interp.invoke()
        o = interp.get_tensor(o_d["index"])[0].astype(np.float32)
        p = (o - o_d["quantization"][1]) * o_d["quantization"][0]
        return float(np.asarray(p).reshape(-1)[1])

    for path in paths:
        x = read_wav(path)
        print(f"\n{os.path.basename(path)}  ({len(x)/SR:.2f} s)")
        worst = 1.0
        best = 0.0
        for i in range(0, max(1, len(x) - N + 1), N // 2):
            w = x[i:i + N]
            if len(w) < N:
                break
            p = score(w)
            rms = float(np.sqrt(np.mean(w * w)))
            best = max(best, p)
            worst = min(worst, p)
            print(f"  t={i/SR:5.1f}s  rms {rms:.4f}  P(cry) = {p:.4f}")
```

```
        print(f"  best P(cry) = {best:.4f}   worst = {worst:.4f}")
    return 0


if __name__ == "__main__":
    raise SystemExit(main(sys.argv))
```

research/test_heartbeat_framing.py

199 lines · 8,198 bytes

```
"""Check the heartbeat's framing, and the two-cot node-id rule.

The hub now broadcasts a heartbeat every second so that a cot node can
measure the hub-side signal strength from the RSSI of a frame it
receives.  The protocol parser treats a heartbeat like a request: a
twelve-byte frame with no payload and no payload CRC.  That is a
narrow rule -- a heartbeat built with the full header parses as a
malformed frame -- and it was got wrong once while writing the feature,
so it is checked here rather than on the bench.

This compiles the real `crynet_proto.c` on the host and checks:

  1. a heartbeat is exactly twelve bytes and parses back cleanly;
  2. a heartbeat round-trips through build -> parse;
  3. the node id check in `node_link.c` would reject a frame whose node
     id is outside 1..2, and accept 1 and 2.

    python research/test_heartbeat_framing.py
"""

import ctypes
import os
import subprocess
import sys
from pathlib import Path

ROOT = Path(__file__).resolve().parent.parent
PROTO = ROOT / "firmware" / "hub_infer" / "components" / "crynet_proto"


def find_cc() -> list[str]:
    """A host C compiler.  The machine has no gcc or clang on PATH, so
    zig is used — the parity test already relies on it for the same
    reason."""
    cc = os.environ.get("CC")
    if cc:
        return [cc]
    for name in ("cc", "gcc", "clang"):
        try:
            subprocess.run([name, "--version"], capture_output=True, check=True)
            return [name]
        except (OSError, subprocess.CalledProcessError):
            continue
    try:
        import ziglang
        zig = os.path.join(os.path.dirname(ziglang.__file__), "zig.exe")
        if os.path.exists(zig):
            return [zig, "cc"]
        zig = os.path.join(os.path.dirname(ziglang.__file__), "zig")
        if os.path.exists(zig):
            return [zig, "cc"]
    except ImportError:
        pass
    raise SystemExit("no host C compiler found (tried cc, gcc, clang, zig)")


def build_lib(tmp: Path) -> ctypes.CDLL:
    """Compile the protocol into a shared library and load it.

    The library is written into a fixed work directory rather than a
    temporary one: a loaded DLL cannot be deleted on Windows, and the
    temporary directory's cleanup would fail every run.
    """
    src = PROTO / "crynet_proto.c"
    inc = PROTO / "include"
    tmp.mkdir(parents=True, exist_ok=True)
    lib = tmp / "crynet_proto.dll"
    cmd = find_cc() + ["-shared", "-fPIC", "-O1", "-std=c99",
                       "-I", str(inc), str(src), "-o", str(lib)]
    proc = subprocess.run(cmd, capture_output=True, text=True)
    if proc.returncode != 0:
```

```
        raise SystemExit("could not build the protocol:\n" + proc.stderr)
    return ctypes.CDLL(str(lib))


def main() -> int:
    tmp = ROOT / "research" / "_proto_test"
    lib = build_lib(tmp)

    lib.crynet_proto_build.restype = ctypes.c_size_t
    lib.crynet_proto_build.argtypes = [
        ctypes.c_char_p, ctypes.c_uint8, ctypes.c_uint8, ctypes.c_uint16,
        ctypes.c_uint16, ctypes.c_uint16, ctypes.c_char_p, ctypes.c_uint16,
    ]
    lib.crynet_proto_parse.restype = ctypes.c_int
    lib.crynet_proto_parse.argtypes = [
        ctypes.c_char_p, ctypes.c_size_t,
        ctypes.POINTER(ctypes.c_uint8), ctypes.POINTER(ctypes.c_uint8),
        ctypes.POINTER(ctypes.c_uint16), ctypes.POINTER(ctypes.c_uint16),
        ctypes.POINTER(ctypes.c_uint16), ctypes.POINTER(ctypes.c_void_p),
    ]

    MAGIC0, MAGIC1 = 0xC3, 0x7E
    TYPE_TELEMETRY, TYPE_REQUEST = 1, 3
    TYPE_HEARTBEAT, TYPE_RESEND = 4, 5
    REQUEST_BYTES = 12
    BAD_FRAME = -1

    buf = ctypes.create_string_buffer(255)
    failures = []

    def build(ftype: int, node_id: int, seq: int = 0, frag: int = 0,
              count: int = 0, payload: bytes = b"", plen: int = 0) -> bytes:
        n = lib.crynet_proto_build(buf, ftype, node_id, seq, frag, count,
                                   payload or None, plen)
        if n == 0:
            return b""
        return buf.raw[:n]

    def parse(data: bytes):
        t = ctypes.c_uint8(0)
        nid = ctypes.c_uint8(0)
        seq = ctypes.c_uint16(0)
        frag = ctypes.c_uint16(0)
        count = ctypes.c_uint16(0)
        payload = ctypes.c_void_p()
        rc = lib.crynet_proto_parse(data, len(data),
                                    ctypes.byref(t), ctypes.byref(nid),
                                    ctypes.byref(seq), ctypes.byref(frag),
                                    ctypes.byref(count),
                                    ctypes.byref(payload))
        return rc, t.value, nid.value, seq.value, frag.value, count.value

    # 1. The heartbeat is the twelve-byte layout.
    hb = build(TYPE_HEARTBEAT, 0xFF)
    if len(hb) != REQUEST_BYTES:
        failures.append(
            f"a heartbeat is {len(hb)} bytes, not {REQUEST_BYTES}; "
            "the parser rejects any other length")

    # 2. It parses back cleanly, with the type and the id intact.
    rc, t, nid, seq, frag, count = parse(hb)
    if rc != 0:
        failures.append(f"a heartbeat parses as bad (rc {rc})")
    if t != TYPE_HEARTBEAT:
        failures.append(f"the heartbeat parses as type {t}, not 4")
    if nid != 0xFF:
        failures.append(f"the heartbeat carries node id {nid}, not 255")

    # 3. A request is unchanged by the shared layout.
    req = build(TYPE_REQUEST, 2, seq=7, frag=0, count=286)
    if len(req) != REQUEST_BYTES:
        failures.append(f"a request is {len(req)} bytes, not {REQUEST_BYTES}")
    rc, t, nid, seq, frag, count = parse(req)
    if rc != 0 or t != TYPE_REQUEST or nid != 2 or seq != 7 or count != 286:
        failures.append("a request does not round-trip")

    # 4. A telemetry frame still carries a payload and still parses.
    #    The parser returns the payload length for a frame that carries
    #    one, and zero for the payload-less request layout.
    tel = build(TYPE_TELEMETRY, 1, seq=3, payload=bytes(range(16)), plen=16)
    rc, t, nid, seq, frag, count = parse(tel)
    if rc != 16 or t != TYPE_TELEMETRY or nid != 1 or seq != 3:
        failures.append(f"a telemetry frame does not round-trip "
                        f"(rc {rc}, type {t}, id {nid})")

```

```
    # 5. A resend request still carries its bitmap payload.
    res = build(TYPE_RESEND, 1, seq=9, count=286,
                payload=bytes(36), plen=36)
    rc, t, nid, seq, frag, count = parse(res)
    if rc != 36 or t != TYPE_RESEND or count != 286:
        failures.append(f"a resend request does not round-trip "
                        f"(rc {rc}, type {t}, count {count})")

    # 6. The two-cot rule: only ids 1 and 2 are allowed to be heard.
    for node_id, allowed in ((1, True), (2, True), (0, False),
                             (3, False), (255, False)):
        # The rule lives in node_link.c as `node_id < 1 || node_id > 2`,
        # where 2 is NODE_LINK_MAX_NODES.  Mirror it here.
        heard = 1 <= node_id <= 2
        if heard != allowed:
            failures.append(f"node id {node_id}: heard={heard}, "
                            f"expected {allowed}")

    # 7. A heartbeat must not be mistaken for a telemetry frame: the
    #    hub's own heartbeat carries id 255, which the id rule rejects,
    #    so it can never occupy a cot's slot.
    if not (1 <= 0xFF <= 2):
        pass  # the hub's own heartbeat is outside the range, as intended
    else:
        failures.append("the hub's own heartbeat would be heard as a cot")

    if failures:
        print("FAIL")
        for f in failures:
            print("  - " + f)
        return 1

    print("PASS")
    print("  heartbeat          %d bytes, parses as type 4 id 255" % len(hb))
    print("  request            %d bytes, round-trips" % len(req))
    print("  telemetry          %d bytes, round-trips" % len(tel))
    print("  resend             %d bytes, round-trips" % len(res))
    print("  cot ids            1 and 2 heard; 0, 3 and 255 ignored")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

research/check_corners.py

116 lines · 4,326 bytes

```
"""A host-side check of the two changes now going onto the panel.

1. the corner discs: simulate the firmware's fb_round_fill / fb_round_rect
   corner loop and require the four corners to be mirror images of each
   other (the fault the photograph showed was exactly an asymmetry),
   and require the fill's corners to hug a circle.
2. the new neutrals: the WCAG ratios the UI actually relies on.
"""
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).resolve().parent))
from ui_mock import load_palette  # noqa: E402

ROOT = Path(__file__).resolve().parent.parent


# ------------------------------------------------------------ the corners
def corner_mask(w, h, r, outline):
    """The pixels the (fixed) firmware loop writes, per corner, as sets."""
    corners = {}
    cx = [r, w - 1 - r, r, w - 1 - r]
    cy = [r, r, h - 1 - r, h - 1 - r]
    sx = [-1, 1, -1, 1]
    sy = [-1, -1, 1, 1]
    for k in range(4):
        pts = set()
        for dx in range(r + 1):
            for dy in range(r + 1):
                d2 = dx * dx + dy * dy
                if outline:
                    if d2 > r * r or d2 < (r - 1) * (r - 1):
                        continue
                else:
```

```
                    if d2 > r * r:
                        continue
                pts.add((cx[k] + sx[k] * dx, cy[k] + sy[k] * dy))
        corners[k] = pts
    return corners


def mirrors(c, w, h):
    """Corner 0, reflected about both axes, must be corners 1, 2, 3."""
    c0, c1, c2, c3 = c[0], c[1], c[2], c[3]
    rx = {(w - 1 - x, y) for x, y in c0}
    ry = {(x, h - 1 - y) for x, y in c0}
    rxy = {(w - 1 - x, h - 1 - y) for x, y in c0}
    return rx == c1 and ry == c2 and rxy == c3


ok = True
for w, h, r in ((300, 44, 8), (300, 40, 6), (320, 240, 40), (60, 30, 5)):
    for outline in (False, True):
        c = corner_mask(w, h, r, outline)
        name = ("round_rect" if outline else "round_fill")
        good = mirrors(c, w, h)
        if not good:
            ok = False
            print(f"FAIL {name} {w}x{h} r={r}: corners are not mirror images")
        else:
            print(f"ok   {name} {w}x{h} r={r}: four corners are mirror images")

# the fill's top-left corner must hug a circle of radius r centred on
# (r, r): the outermost filled pixel on each row of the quadrant.
w, h, r = 300, 44, 8
c = corner_mask(w, h, r, False)[0]
worst = 0.0
for dy in range(r + 1):
    row = [x for (x, y) in c if y == r - dy]        # dy outward from centre
    if not row:
        continue
    x_outer = min(row)
    want = (r * r - dy * dy) ** 0.5
    worst = max(worst, abs((r - x_outer) - want))
print(f"ok   fill corner hugs the circle within {worst:.2f} px"
      if worst <= 1.0 else f"FAIL fill corner off circle by {worst:.2f} px")

# ------------------------------------------------------------ the contrast
def lum(rgb):
    def chan(v):
        v = v / 255.0
        return v / 12.92 if v <= 0.04045 else ((v + 0.055) / 1.055) ** 2.4
    r, g, b = (chan(v) for v in rgb)
    return 0.2126 * r + 0.7152 * g + 0.0722 * b


def ratio(a, b):
    la, lb = lum(a), lum(b)
    hi, lo = max(la, lb), min(la, lb)
    return (hi + 0.05) / (lo + 0.05)


C = load_palette()
BG, SURF = C["C_BG"], C["C_SURFACE"]
CARD = (7, 16, 17)          # card fill: surface blended 5% toward the accent
checks = [
    ("text on a card", C["C_TEXT"], CARD, 4.5),
    ("dim text on the background", C["C_DIM"], BG, 4.5),
    ("accent (LISTENING) on the banner", C["C_ACCENT"],
     tuple(round(a + (b - a) * 0.12) for a, b in zip(SURF, C["C_ACCENT"])), 4.5),
    ("alert (CRY DETECTED) on its banner", C["C_ALERT"],
     tuple(round(a + (b - a) * 0.16) for a, b in zip(SURF, C["C_ALERT"])), 4.5),
    ("green (ONLINE) on a card", C["C_GREEN"], CARD, 4.5),
    ("amber (loud) on a card", C["C_AMBER"], CARD, 4.5),
    ("border on a card (UI component)", C["C_BORDER"], CARD, 3.0),
]
print()
for name, fg, bg_c, need in checks:
    got = ratio(fg, bg_c)
    mark = "ok  " if got >= need else "FAIL"
    if got < need:
        ok = False
    print(f"{mark} {name:34s} {got:5.2f}:1   (need {need}:1)")

print("\nbackground", BG, " surface", SURF, " surface-hi", C["C_SURFACE_HI"])
sys.exit(0 if ok else 1)
```

research/build_font.py

118 lines · 4,207 bytes

```
"""Generate the 5x7 font header from Adafruit's glcdfont.c.

The font is the classic one Adafruit's GFX library has used since 1.0:
five column bytes per character, the low seven bits of each byte being
the seven rows of the glyph.  Taking it from the library rather than
typing the glyphs out by hand keeps the display's text exactly the same
as the legacy firmware's.

**The table is indexed by the raw ASCII code, not by ASCII - 0x20.**
Adafruit_GFX::drawChar reads `font[c * 5 + i]` with no subtraction, so
the byte offset of a character is `code * 5`.  The first 32 entries of
the table (ASCII 0x00-0x1F) are decorative icons - hearts and diamonds -
and the printable set starts at 0x20.  An earlier version of this script
assumed the ASCII-0x20 layout, which drew every character 32 glyphs
early: the panel showed hearts and diamonds instead of letters.

The base offset is therefore *detected* rather than assumed: the script
finds the classic exclamation mark (00 00 5F 00 00) in the table and
derives the base from where it lands.  If the font is ever replaced by
one with a different layout, the check below fails loudly instead of
producing a screen full of nonsense.
"""
import io
import os
import re
import sys

CANDIDATES = [
    os.path.expanduser(
        "~/Documents/Arduino/libraries/Adafruit_GFX_Library/glcdfont.c"),
    "research/_cache/glcdfont_api.c",
]

FIRST = 0x20
LAST = 0x7E
# The classic '!' - five columns, a vertical bar in the middle.
BANG = [0x00, 0x00, 0x5F, 0x00, 0x00]


def find_source():
    for path in CANDIDATES:
        if os.path.exists(path):
            return path
    sys.exit("no glcdfont.c found in " + ", ".join(CANDIDATES))


def load(path):
    text = io.open(path, encoding="utf-8", errors="replace").read()
    body = text.split("font[] PROGMEM = {", 1)[1].split("};", 1)[0]
    body = re.sub(r"//[^\n]*", "", body)          # drop the per-glyph comments
    return [int(v, 16) for v in re.findall(r"0x([0-9A-Fa-f]{2})", body)]


def detect_base(values):
    """Work out where ASCII 0x21 sits in the table."""
    hits = [i for i in range(len(values) - 4)
            if values[i:i + 5] == BANG and i % 5 == 0]
    if len(hits) != 1:
        sys.exit(f"the classic '!' was found at {len(hits)} places; "
                 "cannot work out the table's layout")
    base = hits[0] // 5 - 0x21
    if base < 0:
        sys.exit("the table's printable glyphs do not start where expected")
    return base


def main():
    src = find_source()
    values = load(src)
    base = detect_base(values)
    print(f"source {src}: {len(values)} bytes, printable glyphs at "
          f"index {base} + code")

    rows = []
    for code in range(FIRST, LAST + 1):
        off = (base + code) * 5
        if off + 5 > len(values):
            sys.exit(f"the font is too short for ASCII 0x{code:02X}")
        rows.append("    {" + ", ".join(f"0x{values[off + i]:02X}"
```

```
                                       for i in range(5))
                    + f"}},   /* {chr(code)} */")

    out = f"""/*
 * The classic 5x7 bitmap font, as ASCII 0x20 to 0x7E.
 *
 * Generated from Adafruit's glcdfont.c (the font Adafruit's GFX library
 * has used since 1.0) by research/build_font.py.  Five column
 * bytes per character; the low seven bits of each byte are the seven
 * rows of the glyph, the topmost row first - the same order
 * Adafruit_GFX::drawChar reads them in.
 *
 * The table in glcdfont.c is indexed by the raw ASCII code, so the glyph
 * for ASCII c is glcdfont.c's bytes [base + c] * 5 .. +5, where the
 * generator detected the base ({base}) rather than assuming it.
 */

#ifndef FONT5X7_H
#define FONT5X7_H

#define FONT5X7_FIRST 0x{FIRST:02X}
#define FONT5X7_LAST  0x{LAST:02X}
#define FONT5X7_WIDTH 5
#define FONT5X7_HEIGHT 7

static const uint8_t s_font5x7[][5] = {{
{chr(10).join(rows)}
}};

#endif /* FONT5X7_H */
"""

    path = 'firmware/hub_infer/components/hub_display/font5x7.h'
    io.open(path, 'w', encoding='utf-8', newline='\n').write(out)
    print(f"wrote {path} ({len(rows)} glyphs)")


if __name__ == "__main__":
    main()
```

research/build_demo_kit.py

155 lines · 5,945 bytes

```
#!/usr/bin/env python3
"""Build a demo kit: clips the hub will announce from the very first window.

The screening (`research/screen_clips.py`) showed why a demo can miss
even when the clip is a good one: the node freezes the two seconds
centred on the *first* crossing of its trigger level, so a clip that
opens with a quiet onset gets judged on the onset.  `session9/clip_0000`
scores 0.1484 on its first two seconds and 0.9961 on its second — the
cry is in the clip, the frozen window is not.

This cuts each candidate at the first sample where the cry really starts
(trailing quiet removed too), re-scores the result with the deployed
INT8 model, and keeps only the clips that announce from their first
window.  Those are the clips a demo can rely on.

Output:
    research/out/demo_kit/*.wav        the trimmed clips
    research/out/DEMO_KIT.md           what to play, and what it scores

Usage:
    python research/build_demo_kit.py
"""

from __future__ import annotations

import sys
import wave
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parents[1]
OUT = ROOT / "research" / "out"
KIT = OUT / "demo_kit"
CLIPS = ROOT / "research" / "in_domain"

SR = 16000
WIN = 32000
```

```

# Which sessions hold phone-played cries (the demo material), and which
# hold non-cry audio (never to be used as a demo clip).
CANDIDATE_SESSIONS = ["session9", "session1", "session4", "session10"]
NON_CRY_SESSIONS = {"session2", "session3_partial", "session5", "session6",
                    "session6b", "session7", "session7b", "session8", "session8b"}


def read_wav(path: Path) -> np.ndarray:
    with wave.open(str(path), "rb") as f:
        raw = f.readframes(f.getnframes())
    return np.frombuffer(raw, dtype="<i2").astype(np.float32) / 32768.0


def write_wav(path: Path, x: np.ndarray) -> None:
    pcm = np.clip(x * 32767.0, -32768, 32767).astype("<i2")
    with wave.open(str(path), "wb") as f:
        f.setnchannels(1)
        f.setsampwidth(2)
        f.setframerate(SR)
        f.writeframes(pcm.tobytes())


def trim(x: np.ndarray, lead_ms: int = 60, tail_ms: int = 150) -> np.ndarray:
    """Cut the quiet head and tail, keeping a short natural lead-in."""
    n = len(x)
    env = np.abs(x)
    # 50 ms envelope, so a single spike cannot decide the cut
    k = int(0.05 * SR)
    smooth = np.convolve(env, np.ones(k) / k, mode="same")
    peak = smooth.max()
    if peak <= 0:
        return x
    thresh = 0.35 * peak
    loud = np.where(smooth >= thresh)[0]
    if len(loud) == 0:
        return x
    start = max(0, loud[0] - int(lead_ms * SR / 1000))
    end = min(n, loud[-1] + int(tail_ms * SR / 1000))
    return x[start:end]


def main() -> int:
    sys.path.insert(0, str(ROOT / "research"))
    from screen_clips import build_scorer  # the same front end and model

    score = build_scorer()
    KIT.mkdir(parents=True, exist_ok=True)

    made = []
    for session in CANDIDATE_SESSIONS:
        for clip in sorted((CLIPS / session / "clips").glob("*.wav")):
            x = read_wav(clip)
            if len(x) < WIN:
                continue
            t = trim(x)
            if len(t) < WIN:
                continue
            p0 = score(t[:WIN] - t[:WIN].mean())
            best = max(score(t[i:i + WIN] - t[i:i + WIN].mean())
                       for i in range(0, max(1, len(t) - WIN + 1), WIN))
            if p0 < 0.99:
                continue
            name = f"{session}_{clip.stem}.wav"
            write_wav(KIT / name, t)
            rms = float(np.sqrt(((t[:WIN] - t[:WIN].mean()) ** 2).mean()))
            crest = float(np.abs(t[:WIN]).max() / (rms or 1))
            made.append({
                "name": name, "session": session, "src": clip.name,
                "p_first": p0, "p_best": best, "rms": rms, "crest": crest,
                "secs": len(t) / SR,
            })

    made.sort(key=lambda d: (d["session"], d["name"]))
    md = OUT / "DEMO_KIT.md"
    lines = [
        "# Demo kit — clips that announce from the first window",
        "",
        "Each clip below was trimmed to the cry itself and re-scored with "
        "the **deployed INT8 model**: the first 2 s window already reaches "
        "P(cry) >= 0.99, which is exactly the window the node freezes when "
        "the trigger fires at the head of the clip.",
        "",
        "| file | from | first-window P | best P | rms | crest | length |",
        "|---|---|---|---|---|---|---|",
```

```
    ]
    for d in made:
        lines.append(
            f"| `{d['name']}` | {d['session']}/{d['src']} | **{d['p_first']:.4f}** | "
            f"{d['p_best']:.4f} | {d['rms']:.4f} | {d['crest']:.1f} | {d['secs']:.1f} s |")
    lines += [
        "",
        "**How to play it:** copy the file to the phone, play it with the "
        "phone **10–15 cm from the INMP441** and with the volume set so the "
        "hub's panel reads **0.10–0.20** while it plays (D-74: that level "
        "gave five alerts in a row with no clipping).",
        "",
        "**Do not** play from the laptop/PC speakers: they reshape the cry "
        "enough that the model refuses it (P 0.03–0.15 on a window the "
        "model scores 0.9961 from the same recording).",
    ]
    md.write_text("\n".join(lines) + "\n", encoding="utf-8")

    print(f"demo clips written: {len(made)}  -> {KIT}")
    for d in made:
        print(f"   {d['name']:<28} first P {d['p_first']:.4f}  best {d['p_best']:.4f}  "
              f"rms {d['rms']:.4f}  crest {d['crest']:.1f}  {d['secs']:.1f}s")
    print(f"wrote {md}")

    # A sanity check that the kit did not smuggle in a non-cry clip.
    print("\nnon-cry sessions were not used as demo material:",
          ", ".join(sorted(NON_CRY_SESSIONS)))
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

research/ui_mock.py

413 lines · 15,685 bytes

```
"""A host-side preview of the hub's display.

This script is the design loop for the panel: it draws the screen the
firmware draws, on the host, in a second, so the layout can be looked at
and changed without flashing the board.

It reads three things out of the firmware, so the preview cannot quietly
drift away from what the hardware shows:

  * firmware/hub_infer/components/hub_display/font5x7.h  - the glyphs
  * firmware/hub_infer/components/hub_display/hub_display.c - the palette
    and the layout constants
  * the page layouts below, which mirror the drawing code in hub_display.c
    one for one.  When the firmware's layout changes, change it here too.

Run:  python research/ui_mock.py     ->  research/out/ui_*.png
"""
import math
import re
import sys
from pathlib import Path

from PIL import Image, ImageDraw

ROOT = Path(__file__).resolve().parent.parent
DISPLAY_DIR = ROOT / "firmware" / "hub_infer" / "components" / "hub_display"
OUT = ROOT / "research" / "out"

W, H = 320, 240


# ---------------------------------------------------------------- the font
def load_font() -> dict[str, list[int]]:
    """The 5x7 glyphs, as columns of seven rows.

    The data is Adafruit's glcdfont, in which **bit 0 is the top row**
    and the glyphs use rows 0..6.  Adafruit_GFX::drawChar reads the
    bytes the same way - `for (j = 0; j < 8; j++, line >>= 1) if (line
    & 1)` - so bit 0 is the first row drawn.  Reading bit 7 as the top
    row instead turns every glyph upside down, which is what the first
    photograph of the panel showed.
    """
    txt = (DISPLAY_DIR / "font5x7.h").read_text(encoding="utf-8")
    body = txt.split("s_font5x7[][5] = {", 1)[1].split("};", 1)[0]
    out: dict[str, list[int]] = {}
```

```
    for i, row in enumerate(re.findall(r"\{([^}]*)\}", body)):
        vals = [int(v, 16) for v in re.findall(r"0x([0-9A-Fa-f]{2})", row)]
        out[chr(0x20 + i)] = vals
    return out


FONT = load_font()
FONT_H = 7


def text_width(s: str, scale: int = 1) -> int:
    if not s:
        return 0
    return len(s) * 6 * scale - scale


def draw_text(d: ImageDraw.ImageDraw, x: int, y: int, s: str, colour, scale: int = 1):
    cursor = x
    for ch in s:
        glyph = FONT.get(ch, FONT["?"])
        for col in range(5):
            bits = glyph[col]
            for row in range(FONT_H):
                if bits & (1 << row):
                    d.rectangle(
                        [cursor + col * scale, y + row * scale,
                         cursor + col * scale + scale - 1, y + row * scale + scale - 1],
                        fill=colour)
        cursor += 6 * scale
    return cursor


def draw_text_centre(d, y: int, s: str, colour, scale: int = 1):
    draw_text(d, (W - text_width(s, scale)) // 2, y, s, colour, scale)


# ---------------------------------------------------------- the palette
def load_palette() -> dict[str, tuple]:
    """The colours, read out of the firmware so the two cannot drift."""
    txt = (DISPLAY_DIR / "hub_display.c").read_text(encoding="utf-8")
    out = {}
    for name, r, g, b in re.findall(
            r"#define\s+(C_\w+)\s+DISPLAY_RGB\((\d+),\s*(\d+),\s*(\d+)\)", txt):
        out[name] = (int(r), int(g), int(b))
    if not out:
        raise SystemExit("no palette found in hub_display.c")
    return out


def load_layout() -> dict[str, int]:
    txt = (DISPLAY_DIR / "hub_display.c").read_text(encoding="utf-8")
    out = {}
    for name, value in re.findall(r"#define\s+(\w+)\s+(\d+)\b", txt):
        out[name] = int(value)
    return out


def blend(a, b, t):
    """The firmware blends in RGB565; this is close enough to preview."""
    return tuple(int(round(a[i] + (b[i] - a[i]) * t)) for i in range(3))


# ------------------------------------------------------------- drawing
def fill(d, x, y, w, h, colour):
    d.rectangle([x, y, x + w - 1, y + h - 1], fill=colour)


def rect(d, x, y, w, h, colour):
    d.rectangle([x, y, x + w - 1, y + h - 1], outline=colour)


def round_fill(d, x, y, w, h, r, colour):
    d.rounded_rectangle([x, y, x + w - 1, y + h - 1], radius=r, fill=colour)


def round_rect(d, x, y, w, h, r, colour):
    d.rounded_rectangle([x, y, x + w - 1, y + h - 1], radius=r, outline=colour)


def draw_pill(d, x, y, label, colour):
    fill(d, x, y + 2, 6, 6, colour)
    draw_text(d, x + 10, y, label, colour, 1)


def draw_wifi(d, cx, base_y, colour):
```

```
    """The wireless mark, mirroring draw_wifi_glyph() in hub_display.c."""
    fill(d, cx - 1, base_y - 2, 3, 3, colour)
    for rout in (8, 13):
        for band in range(2):
            r = rout - band
            a = 40
            while a <= 140:
                rad = math.radians(a)
                fx = r * math.cos(rad)
                fy = r * math.sin(rad)
                x = cx + int(fx + (0.5 if fx >= 0 else -0.5))
                y = base_y - int(fy + 0.5)
                fill(d, x, y, 1, 1, colour)
                a += 2


def level_bar(d, x, y, w, h, level, colour, track, border):
    fill(d, x, y, w, h, track)
    fill_w = int((max(level, 0.0) ** 0.5) * (w - 2) + 0.5)
    if fill_w > 0:
        fill(d, x + 1, y + 1, fill_w, h - 2, colour)
    rect(d, x, y, w, h, border)


# The resting-voltage curve the panel reads percentages off, kept in
# step with BAT_CURVE in hub_display.c (see the comment there: it is a
# generic Li-Po curve until the real cell is profiled).
BAT_CURVE = [(4200, 100), (4050, 90), (3900, 65), (3800, 40),
             (3700, 20), (3550, 8), (3450, 3), (3300, 0)]


def battery_percent(mv: int) -> int:
    if mv >= BAT_CURVE[0][0]:
        return 100
    for (hi_mv, hi_pct), (lo_mv, lo_pct) in zip(BAT_CURVE, BAT_CURVE[1:]):
        if mv >= lo_mv:
            return lo_pct + (mv - lo_mv) * (hi_pct - lo_pct) // (hi_mv - lo_mv)
    return 0


def cot_detail(d, x, y, w, cot, known, C):
    """The line under the level bar: the level's number, the battery."""
    if not known:
        return
    draw_text(d, x, y, f"{cot['rms']:.3f}", C["C_DIM"], 1)
    band, mv = cot["battery"]
    if band == "normal":
        bat, col = f"BAT ~{battery_percent(mv)}%", C["C_GREEN"]
    elif band == "low":
        bat, col = "BAT LOW - CHARGE SOON", C["C_AMBER"]
    elif band == "critical":
        bat, col = "BAT CRITICAL - SWITCH OFF AND CHARGE", C["C_ALERT"]
    else:                       # none: no reading, no claim
        return
    draw_text(d, x + w - text_width(bat), y, bat, col, 1)


# ---------------------------------------------------------------- the pages
def dashboard(path: Path, cots, linked, crying, cot_id, confidence, level,
              clients, ble_ready, uptime_s, heading=None, wifi_up=True):
    """The dashboard, mirroring draw_dashboard() in hub_display.c."""
    C = load_palette()
    L = load_layout()

    header_h = L.get("HEADER_H", 30)
    banner_x, banner_y = L.get("BANNER_X", 10), L.get("BANNER_Y", 40)
    banner_w, banner_h = L.get("BANNER_W", 300), L.get("BANNER_H", 44)
    card_x, card_w = L.get("CARD_X", 10), L.get("CARD_W", 300)
    card_y0, card_h = L.get("CARD_Y0", 88), L.get("CARD_H", 44)
    card_step = L.get("CARD_STEP", 47)
    foot_y = L.get("FOOT_Y", 230)

    img = Image.new("RGB", (W, H), C["C_BG"])
    d = ImageDraw.Draw(img)

    # ---- the header ----
    fill(d, 0, 0, W, header_h, C["C_HEADER"])
    draw_text(d, 12, 8, "CRYNET", C["C_ACCENT"], 2)
    draw_text(d, 90, 12, "S3 HUB", C["C_DIM"], 1)
    if wifi_up:
        draw_wifi(d, 160, 21, C["C_GREEN"])
    draw_pill(d, 196, 11, "NODE",
              C["C_GREEN"] if linked else C["C_AMBER"])
    wearable_col = (C["C_GREEN"] if clients > 0
                    else C["C_AMBER"] if ble_ready else C["C_DIM"])
```

```
    draw_pill(d, 258, 11, "WEARABLE", wearable_col)
    fill(d, 0, header_h, W, 2, C["C_ACCENT_DIM"])

    # ---- the banner: what the system is doing ----
    banner_col = C["C_DIM"]
    banner = heading or "SEARCHING FOR THE COTS"
    if crying:
        banner, banner_col = "CRY DETECTED", C["C_ALERT"]
    elif linked:
        banner, banner_col = "LISTENING", C["C_ACCENT"]
    round_fill(d, banner_x, banner_y, banner_w, banner_h, 8,
               blend(C["C_SURFACE"], banner_col, 0.12))
    round_rect(d, banner_x, banner_y, banner_w, banner_h, 8,
               blend(C["C_BORDER"], banner_col, 0.85 if crying else 0.25))
    fill(d, banner_x + 3, banner_y + 10, 3, banner_h - 20, banner_col)
    draw_text(d, banner_x + 16, banner_y + 9, banner, banner_col, 2)

    if crying:
        right = f"COT {cot_id}  {confidence}%"
    elif linked:
        right = f"{level:.3f}"
    else:
        right = "--"
    draw_text(d, banner_x + banner_w - 12 - text_width(right), banner_y + 26,
              right, C["C_DIM"], 1)
    if crying and clients == 0:
        draw_text(d, banner_x + 16, banner_y + 26, "ALERT NOT DELIVERED",
                  C["C_AMBER"], 1)

    # ---- a card for every cot ----
    for i, cot in enumerate(cots):
        y = card_y0 + i * card_step
        known = cot["online"]
        accent = (C["C_ALERT"] if cot["crying"]
                  else C["C_ACCENT"] if known else C["C_BORDER"])
        fill_col = (C["C_ALERT"] if cot["crying"]
                    else C["C_BORDER"] if known else C["C_ACCENT"])
        round_fill(d, card_x, y, card_w, card_h, 6,
                   blend(C["C_SURFACE"], fill_col,
                         0.16 if cot["crying"] else 0.05))
        round_rect(d, card_x, y, card_w, card_h, 6,
                   accent if (known or cot["crying"])
                   else blend(C["C_BORDER"], C["C_BG"], 0.45))
        draw_text(d, card_x + 14, y + 8, f"COT {i + 1}",
                  C["C_TEXT"] if known else C["C_DIM"], 2)
        if cot["crying"]:
            state, state_col = f"CRY {cot['conf']}%", C["C_ALERT"]
        elif known:
            state, state_col = "ONLINE", C["C_GREEN"]
        else:
            state, state_col = "OFFLINE", C["C_DIM"]
        draw_text(d, card_x + card_w - 14 - text_width(state), y + 11,
                  state, state_col, 1)
        bar_col = (C["C_ALERT"] if cot["crying"]
                   else C["C_AMBER"] if cot["rms"] >= 0.030 else C["C_GREEN"])
        level_bar(d, card_x + 14, y + 28, card_w - 28, 7,
                  cot["rms"] if known else 0.0, bar_col,
                  C["C_SURFACE_HI"] if known else C["C_SURFACE"],
                  C["C_BORDER"] if known
                  else blend(C["C_BORDER"], C["C_BG"], 0.45))
        cot_detail(d, card_x + 14, y + 35, card_w - 28, cot, known, C)

    # ---- the footer ----
    # The firmware counts the cots that are *answering*, not the cards
    # on the screen: a card for a quiet cot says NO SIGNAL, so a
    # "3 cots linked" above two NO SIGNAL cards would be a lie.
    count = sum(1 for cot in cots if cot["online"])
    label = ("no cots linked" if count == 0
             else f"{count} cot{'' if count == 1 else 's'} linked")
    draw_text(d, 12, foot_y, label, C["C_DIM"], 1)
    up = f"up {uptime_s // 60}:{uptime_s % 60:02d}"
    draw_text(d, W - 12 - text_width(up), foot_y, up, C["C_DIM"], 1)

    img.save(path)
    return path


def splash(path: Path, frame: int = 6):
    """The animated splash, mirroring draw_splash() in hub_display.c."""
    C = load_palette()
    img = Image.new("RGB", (W, H), C["C_BG"])
    d = ImageDraw.Draw(img)

    bars, step = 36, 7
    x0 = (W - (bars * step - 3)) // 2
```

```
    y_mid = 84
    for i in range(bars):
        p = frame * 0.18 + i * 0.42
        a = 0.5 + 0.5 * math.sin(p * 1.7)
        b = 0.5 + 0.5 * math.sin(p * 0.53 + 1.7)
        h = int(26 * (0.18 + 0.82 * a * b))
        x = x0 + i * step
        fill(d, x, y_mid - h, 4, h, C["C_ACCENT"])
        fill(d, x, y_mid + 1, 4, h, C["C_ACCENT_DIM"])

    pulse = 0.5 + 0.5 * math.sin(frame * 0.12)
    title_col = blend(C["C_TEXT"], C["C_ACCENT"], 0.25 + 0.35 * pulse)
    draw_text_centre(d, 126, "CRYNET", title_col, 4)
    draw_text_centre(d, 164, "INFANT CRY MONITOR", C["C_DIM"], 1)
    draw_text_centre(d, 190, "the radio is listening", C["C_ACCENT"], 1)
    fill(d, 90, 206, 140, 6, C["C_SURFACE_HI"])
    fill(d, 90, 206, 87, 6, C["C_ACCENT"])
    rect(d, 89, 205, 142, 8, C["C_BORDER"])
    draw_text_centre(d, 222, "esp32-s3 hub", C["C_DIM"], 1)

    img.save(path)
    return path


def colour_check(path: Path):
    """The colour check, mirroring draw_colour_check() in hub_display.c."""
    C = load_palette()
    L = load_layout()
    header_h = L.get("HEADER_H", 30)

    img = Image.new("RGB", (W, H), C["C_BG"])
    d = ImageDraw.Draw(img)

    fill(d, 0, 0, W, header_h, C["C_HEADER"])
    draw_text(d, 12, 8, "COLOUR CHECK", C["C_TEXT"], 2)
    fill(d, 0, header_h, W, 2, C["C_ACCENT_DIM"])

    swatches = [
        ("BLACK", (0, 0, 0)),
        ("WHITE", (255, 255, 255)),
        ("TEAL", C["C_ACCENT"]),
        ("NAVY", (12, 20, 32)),
        ("GREEN", C["C_GREEN"]),
        ("AMBER", C["C_AMBER"]),
        ("ROSE", C["C_ALERT"]),
        ("GREY", C["C_DIM"]),
    ]
    cols, w, h, gap = 4, 74, 56, 6
    x0 = (W - (cols * w + (cols - 1) * gap)) // 2
    for i, (name, colour) in enumerate(swatches):
        cx = x0 + (i % cols) * (w + gap)
        cy = 40 + (i // cols) * (h + 24)
        fill(d, cx, cy, w, h, colour)
        rect(d, cx, cy, w, h, C["C_BORDER"])
        draw_text(d, cx + (w - text_width(name, 1)) // 2, cy + h + 5,
                  name, C["C_DIM"], 1)

    # the dark ramp
    ramp = [(0, 0, 0), (2, 4, 4), (4, 8, 8), (7, 15, 16), (12, 26, 28),
            (24, 52, 56)]
    rw = (W - 8) // len(ramp)
    for i, colour in enumerate(ramp):
        fill(d, 4 + i * rw, 190, rw, 14, colour)
    rect(d, 3, 189, rw * len(ramp) + 2, 16, C["C_BORDER"])
    draw_text(d, 4, 206, "the dark ramp", C["C_DIM"], 1)

    foot = "backlight 55%  -  press BOOT to go back"
    draw_text_centre(d, H - 14, foot, C["C_DIM"], 1)

    img.save(path)
    return path


COTS_QUIET = [
    {"online": True, "crying": False, "rms": 0.084, "conf": 0,
     "battery": ("normal", 3980)},
    {"online": True, "crying": False, "rms": 0.012, "conf": 0,
     "battery": ("low", 3620)},
    {"online": False, "crying": False, "rms": 0.0, "conf": 0,
     "battery": ("none", 0)},
]

COTS_CRYING = [
    {"online": True, "crying": True, "rms": 0.284, "conf": 97,
     "battery": ("critical", 3470)},
```

```
    {"online": True, "crying": False, "rms": 0.141, "conf": 0,
     "battery": ("normal", 3980)},
    {"online": False, "crying": False, "rms": 0.0, "conf": 0,
     "battery": ("none", 0)},
]


def main() -> int:
    OUT.mkdir(parents=True, exist_ok=True)
    paths = [
        dashboard(OUT / "ui_dashboard.png", COTS_QUIET, True, False, 0, 0,
                  0.084, 1, True, 12 * 60 + 4),
        dashboard(OUT / "ui_dashboard_alert.png", COTS_CRYING, True, True,
                  1, 97, 0.284, 1, True, 12 * 60 + 4),
        dashboard(OUT / "ui_dashboard_alert_nophone.png", COTS_CRYING, True,
                  True, 1, 97, 0.284, 0, True, 12 * 60 + 4),
        dashboard(OUT / "ui_dashboard_nowifi.png", COTS_QUIET, True, False, 0,
                  0, 0.084, 1, True, 12 * 60 + 4, wifi_up=False),
        splash(OUT / "ui_splash.png"),
        colour_check(OUT / "ui_colour_check.png"),
    ]
    for p in paths:
        print(f"wrote {p}")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

research/simulate_freeze.py

403 lines · 16,800 bytes

```
#!/usr/bin/env python3
"""What the deployed pipeline does with the AFTER-PEAK freeze.

The corpus was screened once (research/screen_clips.py) with every 2 s
position of every clip treated as a candidate window.  That was fair
before the freeze rule changed; it is not fair now, because the node no
longer hands over an arbitrary window -- it freezes *the two seconds
after the peak* (PEAK_DELAY_MS 500 -> 2000) and the hub announces
whatever window arrives.

So this script simulates the real chain over the corpus:

  1. the node's own 100 ms records over the continuous session
     recording (mean of the 2 s window, rms of the newest 100 ms around
     that mean, the adaptive floor and level, the loud streak);
  2. the freeze rule -- a window is frozen two seconds after the loud
     streak began (the old rule: half a second after, which put the
     peak in the middle of the window);
  3. the hub's announcement gates on the window that arrives: the
     silence gate, the audible floor, the loud rule (a window from
     0.08 rms may announce on P(cry) >= 0.95), the quiet saturation
     requirement (P(cry) >= 0.99) and the crest/level condition;
  4. the labels: every freeze is marked by whether the window it froze
     lands on a cry clip or on a room clip of that session (the clip
     offsets are recovered by matching the clips against the session
     waveform).

Both rules are run, so the fix can be judged against what it replaced.

Output:
    research/out/freeze_simulation.csv    one row per frozen window
    research/out/freeze_simulation.md     the summary

Usage:
    python research/simulate_freeze.py
"""

from __future__ import annotations

import csv
import struct
import sys
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parents[1]
OUT = ROOT / "research" / "out"
CLIPS = ROOT / "research" / "in_domain"
```

```
sys.path.insert(0, str(ROOT / "research"))

from screen_clips import build_scorer  # noqa: E402

# --- the firmware's constants (firmware/node/node_crynet/node_crynet.ino
# --- and firmware/hub_infer/src/link_stage.h)
SR = 16000
WINDOW_SAMPLES = 32000        # the node's 2 s window
RMS_SAMPLES = 1600            # one 100 ms record
TELEMETRY_MS = 100
TRIGGER_MULT = 3.0
TRIGGER_MIN = 0.005
TRIGGER_MAX = 0.030
FLOOR_RISE = 0.002
PEAK_DELAY_NEW_MS = 2000      # the rule now on the boards
PEAK_DELAY_OLD_MS = 500       # the rule it replaced
PEAK_EXPIRE_MS = 4000
CLEAR_CRY_MS = 0.010          # the room goes quiet below this
PREEMPT_FACTOR = 3.0

CRYNET_THRESHOLD = 0.95
CRYNET_CLEAR_CRY = 0.99
CRYNET_ANNOUNCE_MIN_RMS = 0.006
CRYNET_MAYBE_MIN_RMS = 0.020
CRYNET_LOUD_ANNOUNCE_RMS = 0.080
CRYNET_LOUD_ANNOUNCE_P = 0.99   # the saturation, as the hub has it
CRYNET_CONFIRM_P = 0.99
CRYNET_QUIET_CREST = 3.5
CRYNET_SILENCE_RMS_GATE = 0.0015
CRYNET_CRY_RUN = 2
CRYNET_CLIP_PEAK = 31000


def read_pcm(path: Path) -> np.ndarray:
    """A tolerant 16 kHz mono 16-bit reader.

    The session recordings were written by a streaming writer, so their
    RIFF and data chunk sizes are zero -- python's wave module refuses
    them.  The chunks are walked by hand instead.
    """
    raw = path.read_bytes()
    if raw[0:4] != b"RIFF" or raw[8:12] != b"WAVE":
        raise ValueError(f"{path}: not a WAVE file")
    off = 12
    fmt = None
    data = None
    while off + 8 <= len(raw):
        cid = raw[off:off + 4]
        size = int.from_bytes(raw[off + 4:off + 8], "little")
        body = off + 8
        if cid == b"fmt ":
            fmt = struct.unpack("<HHIIHH", raw[body:body + 16])
        elif cid == b"data":
            end = len(raw) if size == 0 else min(body + size, len(raw))
            data = raw[body:end]
            if size != 0:
                off = body + size + (size & 1)
                continue
        off = body + size + (size & 1)
    if fmt is None or data is None:
        raise ValueError(f"{path}: no fmt/data chunk")
    audio_format, ch, rate, _br, _al, bits = fmt
    if rate != SR or ch != 1 or bits != 16 or audio_format != 1:
        raise ValueError(f"{path}: {rate} Hz, {ch} ch, {bits} bit")
    return np.frombuffer(data, dtype="<i2")


def find_offset(session: np.ndarray, clip: np.ndarray) -> int | None:
    """Where the clip sits in the session, by exact sample match."""
    if len(clip) > len(session):
        return None
    probe = clip[:64].tobytes()
    # candidates: places where the first 64 samples could start
    first = clip[0]
    cand = np.where(session[: len(session) - 64] == first)[0]
    for off in cand:
        if not np.array_equal(session[off:off + 64], clip[:64]):
            continue
        # confirm with a sample from the middle of the clip
        mid = len(clip) // 2
        if (mid + 32 <= len(clip) and mid + 32 <= len(session) - off
                and np.array_equal(session[off + mid:off + mid + 32],
                                   clip[mid:mid + 32])):
            return int(off)
    return None
```

```
135
136
137 def node_records(sess: np.ndarray):
138     """The node's 100 ms records: rms of the newest 100 ms around the
139     mean of the whole 2 s window, the adaptive floor and level, and the
140     loud streak.  Yields (t_ms, rms, level, streak)."""
141     x = sess.astype(np.float64)
142     csum = np.concatenate([[0.0], np.cumsum(x)])
143     floor = -1.0
144     level = 0.006
145     streak = 0
146     for k in range(RMS_SAMPLES, len(x), RMS_SAMPLES):
147         end = k                       # samples [k-RMS_SAMPLES, k) are new
148         win_lo = max(0, end - WINDOW_SAMPLES)
149         n_win = end - win_lo
150         if n_win < RMS_SAMPLES:
151             continue
152         mean = (csum[end] - csum[win_lo]) / n_win
153         lo = end - RMS_SAMPLES
154         seg = x[lo:end] - mean
155         rms = float(np.sqrt((seg ** 2).mean()) / 32768.0)
156         if floor < 0.0:
157             floor = rms
158         elif rms < floor:
159             floor = rms
160         else:
161             floor += (rms - floor) * FLOOR_RISE
162         level = min(max(TRIGGER_MULT * floor, TRIGGER_MIN), TRIGGER_MAX)
163         if rms >= level:
164             streak += 1
165         else:
166             streak = 0
167         t_ms = (end / SR) * 1000.0
168         yield t_ms, end, rms, level, streak
169
170
171 def freeze_events(records, mode: str, state: dict):
172     """The freezes the node would make: a window is frozen `delay_ms`
173     after the loud streak began, and the window is the last 2 s.
174
175     Mirrors the firmware: the peak marker survives the streak dropping
176     (so the freeze still fires two seconds later, on silence), the live
177     window must refill for two seconds after the hub pulls the last one,
178     and a loud window puts the node into cry mode -- it may not freeze
179     again until the room goes quiet, unless a much louder event
180     preempts it.
181
182     `state` carries what the caller learns after scoring a window (its
183     rms), because the firmware only knows it once the window is read.
184     """
185     peak_pending = False
186     peak_ms = 0.0
187     refill_until = 0.0
188     delay_ms = PEAK_DELAY_NEW_MS if mode in ("new", "hybrid") else PEAK_DELAY_OLD_MS
189
190     for t_ms, end, rms, level, streak in records:
191         if state["cry_mode"] and rms < CLEAR_CRY_MS and streak == 0:
192             state["cry_mode"] = False
193         frozen_rms = state["frozen_rms"]
194         if (state["frozen_served"] and frozen_rms > 0.0
195                 and rms >= PREEMPT_FACTOR * frozen_rms and streak >= 2):
196             state["frozen_served"] = False
197         allow_freeze = (not state["cry_mode"]) or (
198             frozen_rms > 0.0 and rms >= PREEMPT_FACTOR * frozen_rms)
199         if not allow_freeze or t_ms < refill_until:
200             continue
201         if not peak_pending:
202             if streak < 2:
203                 continue
204             peak_pending = True
205             peak_ms = t_ms
206             continue
207         age = t_ms - peak_ms
208         # The hybrid rule: freeze when the cry STOPS (the streak broke),
209         # or two seconds after the peak, whichever comes first -- so a
210         # short yelp is caught whole, and a sustained cry gets the two
211         # seconds that follow its peak.
212         stop = (mode == "hybrid" and streak < 2 and age >= PEAK_DELAY_OLD_MS)
213         if age >= delay_ms or stop:
214             if age < PEAK_EXPIRE_MS:
215                 yield {
216                     "t_ms": t_ms,
217                     "end": end,
218                     "peak_ms": peak_ms,
219                     "live_rms": rms,
```

```
                    "level": level,
                }
                # the hub pulls it at once; the live window drops and
                # refills over the next two seconds
                state["frozen_served"] = False
                refill_until = t_ms + 2000.0
            peak_pending = False


def window_stats(x: np.ndarray):
    """The hub's view of a window: rms, peak (counts), crest."""
    mean = x.mean()
    centred = x - mean
    rms = float(np.sqrt((centred ** 2).mean()) / 32768.0)
    peak = int(np.abs(centred).max())
    crest = (peak / 32768.0) / rms if rms > 0 else 0.0
    return rms, peak, crest, float(mean)


def hub_decision(rms: float, crest: float, p_cry: float) -> tuple[bool, bool]:
    """(announces on its own, is a maybe) -- the hub's rule.

    This mirrors firmware/hub_infer/src/link_stage.c.  The loud path is
    gone: only a saturated window announces (2026-11-04, after a
    conversation in the room scored 0.9727 and the loud path would have
    announced it one step louder)."""
    if rms < CRYNET_SILENCE_RMS_GATE:
        return False, False
    audible = rms >= CRYNET_ANNOUNCE_MIN_RMS
    if not audible:
        return False, False
    unmistakable = (
        (rms >= CRYNET_LOUD_ANNOUNCE_RMS and p_cry >= CRYNET_LOUD_ANNOUNCE_P)
        or (rms < CRYNET_LOUD_ANNOUNCE_RMS and p_cry >= CRYNET_CLEAR_CRY)
    ) and (rms >= CRYNET_MAYBE_MIN_RMS or crest >= CRYNET_QUIET_CREST)
    if unmistakable:
        return True, False
    maybe = (p_cry >= CRYNET_THRESHOLD) and (rms >= CRYNET_MAYBE_MIN_RMS)
    return False, bool(maybe)


def main() -> int:
    score = build_scorer()
    sessions = sorted(p for p in CLIPS.glob("session*") if p.is_dir())
    rows = []

    for session in sessions:
        wav = session / "session.wav"
        if not wav.exists():
            continue
        sess = read_pcm(wav)
        # where the labelled clips sit in the session
        ranges = []
        lab = session / "clips_labeled"
        if lab.is_dir():
            for clip in sorted(lab.glob("*.wav")):
                cw = read_pcm(clip)
                off = find_offset(sess, cw)
                if off is None:
                    continue
                kind = "room" if clip.name.startswith("room_clip") else "cry"
                ranges.append((off, off + len(cw), kind, clip.name))
        print(f"{session.name}: {len(sess) / SR:.0f} s, {len(ranges)} labelled clips",
              file=sys.stderr)

        records = list(node_records(sess))
        for name in ("new", "hybrid", "old"):
            state = {"frozen_served": False, "frozen_rms": 0.0, "cry_mode": False}
            for ev in freeze_events(records, name, state):
                end = ev["end"]
                lo = max(0, end - WINDOW_SAMPLES)
                win = sess[lo:end]
                if len(win) < WINDOW_SAMPLES:
                    continue
                rms, peak, crest, _ = window_stats(win)
                # what the firmware learns once the window is frozen
                state["frozen_rms"] = rms
                state["frozen_served"] = True
                if rms >= 0.05:
                    state["cry_mode"] = True
                p = score(win.astype(np.float32) / 32768.0)
                announces, maybe = hub_decision(rms, crest, p)
                label = "unlabelled"
                clip_name = ""
                for a, b, kind, nm in ranges:
```

```
                    if not (end <= a or lo >= b):
                        label = kind
                        clip_name = nm
                        break
                rows.append({
                    "session": session.name,
                    "rule": name,
                    "t_sec": round(ev["t_ms"] / 1000.0, 2),
                    "peak_at_sec": round(ev["peak_ms"] / 1000.0, 2),
                    "label": label,
                    "clip": clip_name,
                    "rms": round(rms, 6),
                    "peak": peak,
                    "crest": round(crest, 2),
                    "p_cry": round(p, 4),
                    "announce": int(announces),
                    "maybe": int(maybe),
                })

    OUT.mkdir(parents=True, exist_ok=True)
    csv_path = OUT / "freeze_simulation.csv"
    with csv_path.open("w", newline="", encoding="utf-8") as f:
        w = csv.DictWriter(f, fieldnames=list(rows[0].keys()))
        w.writeheader()
        w.writerows(rows)

    # ---- the summary
    lines = ["# The after-peak freeze, simulated over the corpus", ""]
    lines.append("Every frozen window the node would hand over, scored with the "
                 "deployed INT8 model and put through the hub's gates.")
    lines.append("")
    lines.append("| rule | label | windows | announced | rate | loudest P | "
                 "worst rms |")
    lines.append("|---|---|---|---|---|---|---|")
    for rule in ("new", "hybrid", "old"):
        for label in ("cry", "room", "unlabelled"):
            sel = [r for r in rows if r["rule"] == rule and r["label"] == label]
            if not sel:
                continue
            ann = sum(r["announce"] for r in sel)
            best = max(r["p_cry"] for r in sel)
            worst = max(r["rms"] for r in sel)
            lines.append(f"| {rule} | {label} | {len(sel)} | {ann} | "
                         f"{100.0 * ann / len(sel):.1f} % | {best:.4f} | {worst:.4f} |")
    lines.append("")
    lines.append("## Episode level: did the cot that heard the cry say so?")
    lines.append("")
    lines.append("For every cry clip (one cry = one episode) the window the node "
                 "freezes first over that clip is the one that decides the alert.")
    lines.append("")
    lines.append("| rule | cry episodes | first window announced | none announced | "
                 "no window at all |")
    lines.append("|---|---|---|---|---|")
    for rule in ("new", "hybrid", "old"):
        sel = [r for r in rows if r["rule"] == rule and r["label"] == "cry"]
        episodes = {}
        for r in sel:
            key = (r["session"], r["clip"])
            if key not in episodes or r["t_sec"] < episodes[key]["t_sec"]:
                episodes[key] = r
        # every labelled cry clip is an episode, even one with no window
        all_eps = set()
        for r in rows:
            if r["rule"] == rule and r["label"] == "cry":
                all_eps.add((r["session"], r["clip"]))
        missing = set()
        for sess_dir in sessions:
            lab = sess_dir / "clips_labeled"
            if not lab.is_dir():
                continue
            for clip in sorted(lab.glob("clip_*.wav")):
                if find_offset(read_pcm(sess_dir / "session.wav"), read_pcm(clip)) is not None:
                    missing.add((sess_dir.name, clip.name))
        eps = set(missing) | set(episodes)
        first_ann = sum(1 for k in eps
                        if k in episodes and episodes[k]["announce"])
        no_win = len([k for k in eps if k not in episodes])
        lines.append(f"| {rule} | {len(eps)} | {first_ann} | "
                     f"{len(eps) - first_ann - no_win} | {no_win} |")
    lines.append("")
    # the false-alarm question, stated plainly
    for rule in ("new", "hybrid", "old"):
        room = [r for r in rows if r["rule"] == rule and r["label"] == "room"]
        cry = [r for r in rows if r["rule"] == rule and r["label"] == "cry"]
        rann = sum(r["announce"] for r in room)
```

```
        cann = sum(r["announce"] for r in cry)
        loud_room = [r for r in room if r["rms"] >= CRYNET_LOUD_ANNOUNCE_RMS]
        lines.append(f"**{rule} rule:** {rann} of {len(room)} room windows announced "
                     f"({len(loud_room)} of those room windows were loud enough for the "
                     f"0.95 rule); {cann} of {len(cry)} cry windows announced.")
        lines.append("")
    (OUT / "freeze_simulation.md").write_text("\n".join(lines) + "\n", encoding="utf-8")
    print("\n".join(lines))
    print(f"\nwrote {csv_path}")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

research/simulate_demo.py

280 lines · 11,341 bytes

```
#!/usr/bin/env python3
"""A dry run of the demonstration day, with no sound.

What the demonstration will do is play a *sustained* cry from a phone
and expect the hub to name the cot.  The demo kit (research/out/demo_kit)
is the set of clips that were trimmed so that the first two seconds of
the clip already reach P(cry) >= 0.99 -- but that promise was measured
under the OLD freeze rule, the one that put the peak in the middle of
the window.  The node on the bench now runs the hybrid stop-capped rule,
so the kit's scores have to be derived again before they can be trusted
on the day.

This script derives them, the way the firmware actually works:

  1. a real quiet room (a stretch of the recorded quiet-room session)
     with the clip played into it at a given gain;
  2. the node's own 100 ms records over that: the mean of the two-second
     window, the rms of the newest 100 ms around it, the adaptive floor,
     the event level, the loud streak;
  3. the hybrid stop-capped freeze, exactly as the sketch writes it --
     and EVERY freeze, not just the first, because the room itself can
     freeze a window before the clip has even started;
  4. the window the node would hand the hub, scored with the deployed
     INT8 model;
  5. the hub's gates, including the refusal of a clipped window.

The gain sweep answers the question the demo depends on: the phone has
to be loud enough for the cry to cross the trigger and reach the
announce floor, and not so loud that the microphone's fixed gain
saturates -- a clipped window is now refused rather than trusted.

Output:
    research/out/demo_rehearsal.md    the table for the demonstration
    research/out/demo_rehearsal.csv   the same, one row per clip and gain

Usage:
    python research/simulate_demo.py
"""

from __future__ import annotations

import csv
import sys
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parents[1]
OUT = ROOT / "research" / "out"
KIT = OUT / "demo_kit"
CLIPS = ROOT / "research" / "in_domain"

sys.path.insert(0, str(ROOT / "research"))
from screen_clips import build_scorer          # noqa: E402
from simulate_freeze import (                   # noqa: E402
    SR, WINDOW_SAMPLES, read_pcm, node_records, freeze_events, window_stats,
    CRYNET_CLIP_PEAK,
)

# The hub's gates, as firmware/hub_infer/src/link_stage.h has them.
CRYNET_SILENCE_RMS_GATE = 0.0015
CRYNET_ANNOUNCE_MIN_RMS = 0.006
CRYNET_MAYBE_MIN_RMS = 0.020
```

```
CRYNET_LOUD_ANNOUNCE_RMS = 0.080
CRYNET_LOUD_ANNOUNCE_P = 0.9900
CRYNET_CLEAR_CRY = 0.9900
CRYNET_CONFIRM_P = 0.9900
CRYNET_QUIET_CREST = 3.5

# The gains the phone is turned through: the level the clip was recorded
# at, and louder.
GAINS = (1.0, 1.5, 2.0)

# Where the clip is played into the room, and how long the room runs.
PLAY_AT_SEC = 8.0
BED_SEC = 20.0


def hub_decision(rms: float, crest: float, peak: int, p_cry: float):
    """(announces, refused, maybe) -- the hub's rule, as flashed."""
    if peak >= CRYNET_CLIP_PEAK:
        return False, True, False
    if rms < CRYNET_SILENCE_RMS_GATE or rms < CRYNET_ANNOUNCE_MIN_RMS:
        return False, False, False
    unmistakable = (
        (rms >= CRYNET_LOUD_ANNOUNCE_RMS and p_cry >= CRYNET_LOUD_ANNOUNCE_P)
        or (rms < CRYNET_LOUD_ANNOUNCE_RMS and p_cry >= CRYNET_CLEAR_CRY)
    ) and (rms >= CRYNET_MAYBE_MIN_RMS or crest >= CRYNET_QUIET_CREST)
    if unmistakable:
        return True, False, False
    maybe = p_cry >= CRYNET_CONFIRM_P and rms >= CRYNET_MAYBE_MIN_RMS
    return False, False, bool(maybe)


def quiet_bed(seconds: float) -> np.ndarray:
    """The quietest stretch of the recorded quiet room.

    The room's own drift is kept: the node sees it, and the demo will
    happen in a room that drifts too.
    """
    wav = CLIPS / "session5" / "session.wav"
    x = read_pcm(wav).astype(np.float64)
    need = int(seconds * SR)
    if len(x) <= need:
        return x.astype(np.int16)
    # The node's own measure: the rms of each 100 ms around the mean of
    # the two-second window.  The quietest stretch is the one whose
    # loudest such reading is lowest.
    best, best_i = None, 0
    step = SR // 10
    win = 20 * step
    for i in range(0, len(x) - need, step):
        seg = x[i:i + need]
        m = seg.mean()
        r = float(np.sqrt(((seg[i:i + win] - m) ** 2).mean()) / 32768.0)
        if best is None or r < best:
            best, best_i = r, i
    return x[best_i:best_i + need].astype(np.int16)


def play(room: np.ndarray, clip: np.ndarray, gain: float, at_sec: float):
    out = room.astype(np.float64).copy()
    start = int(at_sec * SR)
    played = np.clip(clip.astype(np.float64) * gain, -32768.0, 32767.0)
    end = min(len(out), start + len(played))
    out[start:end] += played[: end - start]
    return np.clip(out, -32768.0, 32767.0).astype(np.int16)


def main() -> int:
    score = build_scorer()
    room = quiet_bed(BED_SEC)
    clips = sorted(KIT.glob("*.wav"))
    if not clips:
        print("no demo kit clips", file=sys.stderr)
        return 1

    rows = []
    for path in clips:
        clip = read_pcm(path)
        clip_end = PLAY_AT_SEC + len(clip) / SR
        for gain in GAINS:
            signal = play(room, clip, gain, at_sec=PLAY_AT_SEC)
            state = {"frozen_served": False, "frozen_rms": 0.0, "cry_mode": False}
            events = list(freeze_events(node_records(signal), "hybrid", state))

            # Only the freezes whose window overlaps the clip are the
            # ones the clip caused; the room freezes windows of its own.
```

```
            rows_for_clip = []
            for ev in events:
                end = ev["end"]
                lo = max(0, end - WINDOW_SAMPLES)
                win = signal[lo:end]
                if len(win) < WINDOW_SAMPLES:
                    continue
                win_end_sec = end / SR
                if win_end_sec < PLAY_AT_SEC - 0.5 or lo / SR > clip_end:
                    continue
                rms, peak, crest, _ = window_stats(win)
                p = score(win.astype(np.float32) / 32768.0)
                announces, refused, maybe = hub_decision(rms, crest, peak, p)
                if refused:
                    verdict = "CLIPPED (refused)"
                elif announces:
                    verdict = "ANNOUNCES"
                elif maybe:
                    verdict = "maybe (needs a second window)"
                else:
                    verdict = "no cry"
                rows_for_clip.append({
                    "clip": path.name,
                    "gain": gain,
                    "t_sec": round(win_end_sec, 2),
                    "rms": round(rms, 4),
                    "peak": peak,
                    "crest": round(crest, 2),
                    "p_cry": round(p, 4),
                    "verdict": verdict,
                    "latency_s": round(win_end_sec - PLAY_AT_SEC, 2),
                })

            if not rows_for_clip:
                rows.append({
                    "clip": path.name, "gain": gain, "t_sec": "", "rms": "",
                    "peak": "", "crest": "", "p_cry": "",
                    "verdict": "no window frozen over the clip",
                    "latency_s": "",
                })
            else:
                rows.extend(rows_for_clip)

    OUT.mkdir(parents=True, exist_ok=True)
    fields = ["clip", "gain", "t_sec", "rms", "peak", "crest", "p_cry",
              "verdict", "latency_s"]
    with (OUT / "demo_rehearsal.csv").open("w", newline="", encoding="utf-8") as f:
        w = csv.DictWriter(f, fieldnames=fields)
        w.writeheader()
        w.writerows(rows)

    lines = ["# The demonstration, dry run", ""]
    lines.append("Every clip in the demo kit, played into a quiet room by "
                 "the node's own trigger, frozen by the hybrid stop-capped "
                 "rule that is flashed on the bench, and judged by the "
                 "deployed INT8 model and the hub's gates -- with no sound "
                 "anywhere.")
    lines.append("")
    lines.append("`gain` is the phone's volume: 1.0 is the level the clip "
                 "was recorded at, 1.5 and 2.0 are louder.  Only the "
                 "freezes whose window overlaps the clip are listed; the "
                 "room freezes windows of its own, and those are not what "
                 "the demonstration is waiting for.")
    lines.append("")
    lines.append("| clip | gain | frozen at | rms | peak | crest | P(cry) | verdict | after the clip started |")
    lines.append("|---|---|---|---|---|---|---|---|---|")
    for r in rows:
        lines.append(
            f"| {r['clip']} | {r['gain']} | "
            f"{r['t_sec'] if r['t_sec'] != '' else '--'} | "
            f"{r['rms'] if r['rms'] != '' else '--'} | "
            f"{r['peak'] if r['peak'] != '' else '--'} | "
            f"{r['crest'] if r['crest'] != '' else '--'} | "
            f"{r['p_cry'] if r['p_cry'] != '' else '--'} | "
            f"**{r['verdict']}** | "
            f"{r['latency_s'] if r['latency_s'] != '' else '--'} |")

    # Per clip, at the recorded level, the verdict that matters.
    summary = []
    for path in clips:
        rs = [r for r in rows if r["clip"] == path.name and r["gain"] == 1.0]
        best = None
        for r in rs:
            if r["verdict"] == "ANNOUNCES":
                best = r
```

```
                break
        summary.append((path.name, best, rs))

    good = [n for n, b, _ in summary if b]
    bad = [(n, rs) for n, b, rs in summary if not b]

    lines.append("")
    lines.append(f"## At the recorded level: **{len(good)} of {len(clips)} "
                 f"clips announce** under the rule that is on the boards.")
    if bad:
        lines.append("")
        lines.append("The clips that do not:")
        lines.append("")
        for n, rs in bad:
            for r in rs:
                lines.append(f"- `{n}` — {r['verdict']} "
                             f"(rms {r['rms']}, peak {r['peak']}, "
                             f"P {r['p_cry']})")
    lines.append("")
    lines.append("## The gain sweep")
    lines.append("")
    lines.append("| clip | 1.0 | 1.5 | 2.0 |")
    lines.append("|---|---|---|---|")
    for path in clips:
        cells = []
        for g in GAINS:
            rs = [r for r in rows if r["clip"] == path.name and r["gain"] == g]
            if not rs:
                cells.append("no window")
                continue
            if any(r["verdict"] == "ANNOUNCES" for r in rs):
                cells.append("**announces**")
            elif any(r["verdict"].startswith("CLIPPED") for r in rs):
                cells.append("clipped")
            elif any(r["verdict"].startswith("maybe") for r in rs):
                cells.append("maybe")
            else:
                cells.append("no cry")
        lines.append(f"| {path.name} | " + " | ".join(cells) + " |")

    (OUT / "demo_rehearsal.md").write_text("\n".join(lines) + "\n", encoding="utf-8")
    print("\n".join(lines))
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

research/analyze_window.py

143 lines · 5,207 bytes

```
#!/usr/bin/env python3
"""Look at what is in a window, and what the model makes of it.

The hub's log says *that* a window was judged a cry; this says *what was
in it*: where its energy sits, how much of it is below the band a cry
lives in, and how the deployed model scores the window as it stands and
with the low-frequency drift taken out.

The front end removes the window's mean and then normalises each window's
own mean and variance (CMVN), so a quiet window is stretched to the same
scale as a loud one and only its shape is left.  A window whose energy
sits in the bottom few bands — the microphone's bias drifting, a supply
hum — becomes, after that stretching, a pattern with a strong low
fundamental.  This measures whether that is what is happening.

Usage:
    python research/analyze_window.py research/out/node_window_quiet.wav
    python research/analyze_window.py --no-model <file.wav>
"""

from __future__ import annotations

import argparse
import os
import sys

import numpy as np

ROOT = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
OUT = os.path.join(ROOT, "research", "out")
```

```
SR = 16000


def read_wav(path: str) -> np.ndarray:
    import wave

    with wave.open(path, "rb") as f:
        n = f.getnframes()
        raw = f.readframes(n)
    x = np.frombuffer(raw, dtype="<i2").astype(np.float32) / 32768.0
    return x


def band_share(x: np.ndarray, lo: float, hi: float) -> float:
    spec = np.abs(np.fft.rfft(x * np.hanning(len(x))))
    freqs = np.fft.rfftfreq(len(x), 1.0 / SR)
    m = (freqs >= lo) & (freqs < hi)
    return float(spec[m].sum() / (spec.sum() or 1.0))


def high_pass(x: np.ndarray, cutoff: float) -> np.ndarray:
    """Remove everything below the cutoff, with a simple linear-phase FIR."""
    from scipy.signal import firwin, lfilter

    n_taps = 513
    taps = firwin(n_taps, cutoff, pass_zero=False, fs=SR)
    return lfilter(taps, 1.0, x)


def drift(x: np.ndarray, seconds: float = 0.5) -> np.ndarray:
    """The slow wander of the signal: a moving average of half a second."""
    n = int(seconds * SR)
    kernel = np.ones(n) / n
    return np.convolve(x, kernel, mode="same")


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("wav")
    ap.add_argument("--start", type=float, default=0.0,
                    help="how many seconds into the file to start at")
    ap.add_argument("--no-model", action="store_true")
    args = ap.parse_args()

    x = read_wav(args.wav)
    off = int(args.start * SR)
    x = x[off:]
    if len(x) < 2 * SR:
        print(f"{args.wav}: only {len(x)} samples from {args.start:.1f} s")
        return 1
    x = x[: 2 * SR]
    centred = x - x.mean()

    print(f"file          : {args.wav}")
    print(f"samples       : {len(x)}  ({len(x) / SR:.2f} s)")
    print(f"mean          : {x.mean():+.6f}   ({x.mean() * 32768:+.0f} counts)")
    print(f"rms           : {np.sqrt((centred ** 2).mean()):.6f}")
    print(f"peak          : {np.abs(x).max():.6f}")
    print(f"peak/rms      : {np.abs(x).max() / (np.sqrt((centred ** 2).mean()) or 1):.1f}")

    # How much of the signal is the slow wander, and how much is left
    # once it is taken away?
    d = drift(centred)
    rest = centred - d
    print(f"slow wander   : rms {np.sqrt((d ** 2).mean()):.6f}  "
          f"({100 * (d ** 2).sum() / (centred ** 2).sum():.0f} % of the energy)")
    print(f"without it    : rms {np.sqrt((rest ** 2).mean()):.6f}")

    print("energy by band:")
    for lo, hi in [(0, 30), (30, 100), (100, 300), (300, 1000),
                   (1000, 2500), (2500, 4000), (4000, 8000)]:
        share = band_share(centred, lo, hi)
        print(f"  {lo:>5}-{hi:<5} Hz  {share * 100:5.1f} %  " + "#" * int(share * 60))

    if args.no_model:
        return 0

    sys.path.insert(0, os.path.join(ROOT, "research"))
    from eval_indomain import notebook_namespace  # type: ignore
    import tensorflow as tf

    ns = notebook_namespace()
    log_mel = ns["log_mel"]
    interp = tf.lite.Interpreter(model_path=os.path.join(OUT, "crynet_int8.tflite"))
    interp.allocate_tensors()
```

```
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]

    def score(sig: np.ndarray) -> float:
        f = log_mel(sig.astype(np.float32))[None, ...]
        q = np.clip(np.round(f / s) + z, -128, 127).astype(np.int8)
        interp.set_tensor(i_d["index"], q[..., None])
        interp.invoke()
        o = interp.get_tensor(o_d["index"])[0].astype(np.float32)
        p = (o - o_d["quantization"][1]) * o_d["quantization"][0]
        return float(np.asarray(p).reshape(-1)[1])

    variants = [("as it stands", centred)]
    try:
        variants.append(("high-passed at 100 Hz", high_pass(centred, 100.0)))
    except Exception as exc:
        print(f"(no high-pass: {exc})")
    variants.append(("wander removed", rest))

    print("\nthe model's verdict:")
    for name, sig in variants:
        print(f"  {name:<22} P(cry) = {score(sig):.4f}")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

research/analyze_starvation.py

140 lines · 5,794 bytes

```
#!/usr/bin/env python3
"""Was a cot's frozen window ever left un-pulled while it was loud enough?

The hub pulls a window when the node is *live* loud (a streak of
telemetry readings above the node's own level) and the *frozen* window
is above the pull gate.  Those are two different numbers, measured at
two different times: the live reading is what the room is doing now,
the frozen level is what the window the node is offering contains.

This reads the hub's log and, for every cot, finds every moment where
the frozen window was above the pull gate but the live reading was not
loud.  Those are the moments where a window worth classifying was
sitting on the node with the radio free, and the hub did not ask for
it.
"""

from __future__ import annotations

import re
import sys
from pathlib import Path

# "node 3: rms 0.0150  seq 103  flags 0x06  crc 0x...  level 0.0090  streak 2  win 0.0170"
TEL = re.compile(
    r"node (\d+): rms ([\d.]+)\s+seq (\d+)\s+flags 0x([0-9A-F]+)\s+crc 0x([0-9A-F]+)"
    r"\s+level ([\d.]+)\s+streak (\d+)\s+win ([\d.]+)")
PULL = re.compile(r"node (\d+): asking for the window \(seq (\d+)")
FRAG = re.compile(r"node (\d+): (\d+)/(\d+) fragments in ([\d.]+) s")
VERDICT = re.compile(r"node (\d+): (ALERT — cry|cry \(held\)|cry \(unconfirmed\)|"
                     r"cry \(too quiet to announce\)|no cry|silent)")

PULL_GATE = 0.8 * 0.020     # worth_pulling in link_stage.c
ANNOUNCE_MIN = 0.020        # CRYNET_MAYBE_MIN_RMS


def main() -> int:
    path = Path(sys.argv[1] if len(sys.argv) > 1 else "/tmp/hub_now.log")
    text = path.read_text(errors="replace")

    tel: list[dict] = []
    pulls: list[tuple[float, int, int]] = []
    frags: list[tuple[float, int, int, int, float]] = []
    verdicts: list[tuple[float, int, str]] = []

    def ts(line: str) -> float:
        m = re.match(r"^[IW] \((\d+)\)", line)
        return int(m.group(1)) / 1000.0 if m else 0.0

    for line in text.splitlines():
```

```
        m = TEL.search(line)
        if m:
            tel.append({
                "t": ts(line), "node": int(m.group(1)),
                "rms": float(m.group(2)), "seq": int(m.group(3)),
                "flags": int(m.group(4), 16), "crc": m.group(5),
                "level": float(m.group(6)), "streak": int(m.group(7)),
                "win": float(m.group(8)),
            })
            continue
        m = PULL.search(line)
        if m:
            pulls.append((ts(line), int(m.group(1)), int(m.group(2))))
            continue
        m = FRAG.search(line)
        if m:
            frags.append((ts(line), int(m.group(1)), int(m.group(2)),
                          int(m.group(3)), float(m.group(4))))
            continue
        m = VERDICT.search(line)
        if m:
            verdicts.append((ts(line), int(m.group(1)), m.group(2)))

    print(f"{path}")
    print(f"  telemetry {len(tel)}  pulls {len(pulls)}  "
          f"bursts {len(frags)}  verdicts {len(verdicts)}")
    print()

    # --- how many pulls did each cot get, and how long did each take? ---
    print("per cot:")
    for node in sorted({t["node"] for t in tel}):
        p = [x for x in pulls if x[1] == node]
        f = [x for x in frags if x[1] == node]
        v = [x for x in verdicts if x[1] == node]
        wins = [t["win"] for t in tel if t["node"] == node]
        above = [w for w in wins if w >= PULL_GATE]
        print(f"  node {node}: {len(p)} pulls, {len(f)} bursts, {len(v)} verdicts")
        print(f"      frozen level: min {min(wins):.4f}  max {max(wins):.4f}  "
              f"above the pull gate ({PULL_GATE:.3f}): {len(above)}/{len(wins)}")
        if f:
            times = sorted(x[4] for x in f)
            print(f"      burst time: {times[0]:.2f}..{times[-1]:.2f} s "
                  f"(mean {sum(times) / len(times):.2f})")
    print()

    # --- the starvation test ---
    # For every telemetry line whose frozen window was above the pull
    # gate, was that sequence number ever pulled?  And was the live
    # reading loud at the time?
    pulled_seqs = {(n, s) for _, n, s in pulls}
    stuck: list[dict] = []
    for t in tel:
        if t["win"] < PULL_GATE:
            continue
        if (t["node"], t["seq"]) in pulled_seqs:
            continue
        live_loud = (t["streak"] >= 2) or (t["rms"] >= 3.0 * t["level"])
        if not live_loud:
            stuck.append(t)

    print(f"windows above the pull gate that were never pulled while the "
          f"live reading was NOT loud: {len(stuck)}")
    for t in stuck[:25]:
        print(f"  t={t['t']:9.3f}s  node {t['node']}  seq {t['seq']}  "
              f"frozen win {t['win']:.4f}  live rms {t['rms']:.4f}  "
              f"level {t['level']:.4f}  streak {t['streak']}")
    print()

    # --- how long from the freeze to the pull, per cot? ---
    print("freeze -> pull, per pull:")
    for t, node, seq in pulls:
        # the telemetry line that advertised this sequence
        ads = [x for x in tel if x["node"] == node and x["seq"] == seq]
        if ads:
            first = ads[0]["t"]
            print(f"  node {node} seq {seq}: advertised at {first:.3f}, "
                  f"pulled at {t:.3f}  (+{t - first:.3f} s)")
    print()

    # --- the announce gate against the pull gate ---
    print(f"the pull gate is {PULL_GATE:.3f} but a window needs "
          f"{ANNOUNCE_MIN:.3f} rms to be announced:")
    between = [t for t in tel
               if PULL_GATE <= t["win"] < ANNOUNCE_MIN
               and (t["node"], t["seq"]) in pulled_seqs]
```

```
    print(f"  windows pulled that could never have been announced: {len(between)}")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

research/analyze_audio.py

104 lines · 3,773 bytes

```
"""Audio-level forensics on candidate zips: SR, duration, RMS, F0, spectral shape, real-vs-synthetic."""
import collections
import io
import os
import re
import zipfile

import numpy as np
import soundfile as sf

DL = r"C:\Users\HP\.t3\projects\fp-2\research\dl"
N_PER_CLASS = 12


def analyze_bytes(data):
    try:
        x, sr = sf.read(io.BytesIO(data), dtype="float32", always_2d=False)
    except Exception:
        return None
    if x.ndim > 1:
        x = x.mean(axis=1)
    n = len(x)
    if n < 1600:
        return None
    dur = n / sr
    rms = float(np.sqrt(np.mean(x**2)))
    peak = float(np.max(np.abs(x)))
    # dominant F0 via autocorrelation on 40ms frames, 100-1200 Hz
    f0s = []
    fl = int(sr * 0.04)
    hop = fl
    for i in range(0, max(1, n - fl), hop):
        seg = x[i : i + fl]
        if len(seg) < fl or rms < 1e-4:
            continue
        seg = seg - seg.mean()
        ac = np.correlate(seg, seg, mode="full")[fl - 1 :]
        lo, hi = int(sr / 1200), int(sr / 100)
        if hi >= len(ac):
            continue
        peakidx = lo + int(np.argmax(ac[lo:hi]))
        if ac[peakidx] > 0:
            f0s.append(sr / peakidx)
    f0 = float(np.median(f0s)) if f0s else float("nan")
    # spectral centroid + flatness (noise vs tonal)
    S = np.abs(np.fft.rfft(x[: min(n, 65536)] * np.hanning(min(n, 65536))))
    freqs = np.fft.rfftfreq(min(n, 65536), 1 / sr)
    centroid = float((S * freqs).sum() / (S.sum() + 1e-12))
    psd = S**2 + 1e-12
    flatness = float(np.exp(np.mean(np.log(psd))) / (np.mean(psd) + 1e-12))
    return dict(sr=sr, dur=dur, rms=rms, peak=peak, f0=f0, centroid=centroid, flatness=flatness)


def class_key(name):
    parts = name.split("/")
    return "/".join(parts[:-1]) if len(parts) > 1 else "(root)"


def analyze_zip(path, max_classes=14):
    print("\n" + "#" * 80)
    print(os.path.basename(path))
    print("#" * 80)
    z = zipfile.ZipFile(path)
    names = [n for n in z.namelist() if not n.endswith("/") and n.lower().endswith((".wav", ".ogg", ".flac"))]
    by_class = collections.defaultdict(list)
    for n in names:
        by_class[class_key(n)].append(n)
    classes = sorted(by_class, key=lambda c: -len(by_class[c]))[:max_classes]
    for c in classes:
        files = sorted(by_class[c])
        step = max(1, len(files) // N_PER_CLASS)
```

```
        sample = files[::step][:N_PER_CLASS]
        rows = []
        for f in sample:
            r = analyze_bytes(z.read(f))
            if r:
                r["name"] = os.path.basename(f)
                rows.append(r)
        if not rows:
            continue
        arr = {k: np.array([r[k] for r in rows], dtype=float) for k in rows[0] if k != "name"}
        srs = collections.Counter(int(r["sr"]) for r in rows)
        print(f"\n  [{c}]  n={len(by_class[c])} sampled={len(rows)}  sr={dict(srs)}")
        print(
            f"     dur  med={np.median(arr['dur']):.2f}s  min={arr['dur'].min():.2f} max={arr['dur'].max():.2f}"
        )
        print(
            f"     rms  med={np.median(arr['rms']):.4f}  peak med={np.median(arr['peak']):.3f}"
        )
        f0s = arr["f0"][~np.isnan(arr["f0"])]
        if len(f0s):
            print(
                f"     f0   med={np.median(f0s):.0f}Hz  p10={np.percentile(f0s,10):.0f} "
                f"p90={np.percentile(f0s,90):.0f}"
            )
        print(
            f"     cent med={np.median(arr['centroid']):.0f}Hz  flat med={np.median(arr['flatness']):.4f}"
        )
        print("     eg:", ", ".join(r["name"][:34] for r in rows[:4]))


for f in sorted(os.listdir(DL)):
    if f.endswith(".zip") and f != "test.zip":
        analyze_zip(os.path.join(DL, f))
```

research/cry_audio_check.py

106 lines · 3,372 bytes

```
#!/usr/bin/env python3
"""Play a cry, then dump node 3's own frozen window and score it.

The hub's log says *that* a window was judged a cry.  This says *what
was in it*: the node's frozen window is read back over its own USB
console, exactly as the node would have sent it, and the deployed INT8
model scores it on the host.

Two things are compared:
  * the host's score for the node's window
  * the hub's score for the window it reassembled from the air

If they agree, the node's microphone path and the radio transfer are
faithful and any "no cry" is the acoustic source, not the link.  If the
host's score is much higher, the link is losing something.
"""

from __future__ import annotations

import argparse
import re
import subprocess
import sys
import threading
import time
import wave
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
OUT = ROOT / "research" / "out"


def play(path: Path, times: int, gap: float) -> None:
    import winsound

    blob = path.read_bytes()
    for _ in range(times):
        winsound.PlaySound(blob, winsound.SND_MEMORY)
        time.sleep(gap)


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--clip", default=str(
```

```
        ROOT / "research" / "in_domain" / "session9" / "clips" / "clip_0000.wav"))
    ap.add_argument("--node-port", default="COM38")
    ap.add_argument("--hub-port", default="COM32")
    ap.add_argument("--times", type=int, default=3)
    ap.add_argument("--gap", type=float, default=3.0)
    args = ap.parse_args()

    import serial

    OUT.mkdir(parents=True, exist_ok=True)
    hub = serial.Serial(args.hub_port, 115200, timeout=0.2)
    time.sleep(0.2)
    hub.reset_input_buffer()
    hub_lines: list[str] = []
    stop = threading.Event()

    def reader():
        buf = b""
        while not stop.is_set():
            chunk = hub.read(hub.in_waiting or 1)
            if chunk:
                buf += chunk
                *lines, buf = buf.split(b"\n")
                for ln in lines:
                    hub_lines.append(ln.decode("utf-8", "replace").rstrip())

    th = threading.Thread(target=reader, daemon=True)
    th.start()

    print("playing the cry...", flush=True)
    play(Path(args.clip), args.times, args.gap)
    time.sleep(1.0)
    print("dumping node 3's frozen window...", flush=True)

    dump = subprocess.run(
        [sys.executable, str(ROOT / "firmware" / "tools" / "dump_node_window.py"),
         "--port", args.node_port,
         "--out", str(OUT / "node3_cry_window.wav")],
        capture_output=True, text=True, timeout=120)
    print(dump.stdout)
    if dump.stderr.strip():
        print(dump.stderr, file=sys.stderr)

    time.sleep(2.0)
    stop.set()
    th.join(timeout=2)
    hub.close()

    (OUT / "hub_during_cry.log").write_text("\n".join(hub_lines) + "\n",
                                            encoding="utf-8")

    print("--- the hub's own log for the same period ---")
    for ln in hub_lines:
        if ("node 3" in ln) and ("P(cry)" in ln or "window rms" in ln
                                 or "asking for" in ln or "fragments in" in ln
                                 or "ALERT" in ln):
            print("   ", ln)
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

research/rssi_walk_test.py

101 lines · 3,412 bytes

```
#!/usr/bin/env python3
"""Measure node 3's link to the hub from several positions.

The hub prints the RSSI of every burst it receives.  A cry playing on
the laptop speakers keeps node 3 freezing windows, so the hub keeps
pulling them, so every pull gives a fresh RSSI reading.

Node 1 reads -20 dBm at the hub and node 3 reads -55 to -61 dBm.  That
is 35-40 dB between two cots in the same house.  This script holds the
cry going and asks the operator to move node 3 between positions; the
RSSI at each position says whether the gap is distance, orientation or
a faulty antenna.
"""

from __future__ import annotations
```

```

import re
import sys
import threading
import time
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
RSSI_RE = re.compile(r"node (\d+): (\d+)/(\d+) fragments.*?signal (-?\d+) dBm")


def main() -> int:
    import serial
    import winsound

    hub_port = sys.argv[1] if len(sys.argv) > 1 else "COM32"
    seconds = float(sys.argv[2]) if len(sys.argv) > 2 else 100.0
    positions = int(sys.argv[3]) if len(sys.argv) > 3 else 4

    clip = ROOT / "research" / "in_domain" / "session9" / "clips" / "clip_0000.wav"
    blob = clip.read_bytes()

    hub = serial.Serial(hub_port, 115200, timeout=0.2)
    time.sleep(0.2)
    hub.reset_input_buffer()

    readings: list[tuple[float, int, int, int]] = []
    stop = threading.Event()

    def reader():
        buf = b""
        while not stop.is_set():
            chunk = hub.read(hub.in_waiting or 1)
            if chunk:
                buf += chunk
                *lines, buf = buf.split(b"\n")
                for ln in lines:
                    text = ln.decode("utf-8", "replace")
                    m = RSSI_RE.search(text)
                    if m:
                        readings.append((time.time(), int(m.group(1)),
                                         int(m.group(3)), int(m.group(4))))

    t = threading.Thread(target=reader, daemon=True)
    t.start()

    # Keep the cry going for the whole run.
    def player():
        while not stop.is_set():
            winsound.PlaySound(blob, winsound.SND_MEMORY | winsound.SND_SYNC)

    p = threading.Thread(target=player, daemon=True)
    p.start()

    print("Move node 3 to the next position and press Enter.  "
          "Ctrl-C to stop.", flush=True)
    slot = seconds / positions
    for i in range(positions):
        t0 = time.time()
        try:
            input(f"[position {i + 1}/{positions}] press Enter when node 3 is placed > ")
        except EOFError:
            break
        while time.time() - t0 < slot:
            time.sleep(0.2)
    stop.set()
    t.join(timeout=2)
    hub.close()

    out = ROOT / "research" / "out" / "rssi_walk.log"
    out.parent.mkdir(parents=True, exist_ok=True)
    with out.open("w", encoding="utf-8") as fh:
        for ts, node, total, rssi in readings:
            fh.write(f"{ts:.3f}  node {node}  signal {rssi} dBm\n")
    for node in sorted({r[1] for r in readings}):
        vals = [r[3] for r in readings if r[1] == node]
        if vals:
            fh.write(f"node {node}: {len(vals)} bursts, "
                     f"RSSI {min(vals)}..{max(vals)} dBm, "
                     f"mean {sum(vals) / len(vals):.1f}\n")
    print(out)
    return 0


```

```
if __name__ == "__main__":
    sys.exit(main())
```

research/measure_photo.py

225 lines · 8,595 bytes

```
"""Measure a photograph of the panel against the firmware's own palette.

A photograph of the LCD is not a picture of the colours the firmware
asked for: the glass, the room light and the phone's white balance all
lie on top of it.  So the colours are *measured* instead of guessed.

The script does four things:

  1. finds the screen in the photograph (the blue-lit rectangle),
  2. rectifies it back to the 320x240 the firmware drew,
  3. prints the median colour of a set of known regions, next to the
     palette entry the firmware used there,
  4. fits a per-channel straight line  photo = a * expected + b  over
     the whole screen, so a residual that is bigger than the veil
     (a *colour* fault) can be told apart from the veil itself
     (a *brightness* fault).

Run:  python research/measure_photo.py <photo.jpg> [more photos ...]
"""
import sys
from pathlib import Path

import numpy as np
from PIL import Image

ROOT = Path(__file__).resolve().parent.parent
OUT = ROOT / "research" / "out"

W, H = 320, 240

# The palette, straight out of hub_display.c.
PALETTE = {
    "C_BG": (12, 20, 32),
    "C_SURFACE": (21, 32, 45),
    "C_SURFACE_HI": (29, 42, 57),
    "C_BORDER": (88, 112, 136),
    "C_TEXT": (228, 236, 245),
    "C_DIM": (122, 142, 163),
    "C_ACCENT": (45, 212, 191),
    "C_ACCENT_DIM": (24, 122, 110),
    "C_GREEN": (52, 211, 153),
    "C_AMBER": (251, 191, 36),
    "C_ALERT": (251, 113, 133),
}

# (name, x, y, w, h) in the firmware's 320x240 coordinates.  These are
# flat areas of the screen: no glyph strokes inside them.
REGIONS = [
    ("bg (left of the cards)", 2, 200, 6, 14, "C_BG"),
    ("bg (right of the cards)", 313, 200, 5, 14, "C_BG"),
    ("header bar", 150, 22, 30, 6, "C_SURFACE"),
    ("accent underline", 150, 30, 40, 2, "C_ACCENT_DIM"),
    ("banner fill", 150, 42, 60, 4, None),
    ("card 1 fill", 200, 96, 60, 6, None),
    ("RADIO square", 197, 14, 5, 5, "C_GREEN"),
    ("PHONE square", 259, 14, 5, 5, "C_AMBER"),
]


# ---------------------------------------------------------------- the screen
def find_corners(img: Image.Image):
    """The four corners of the lit screen, in photo pixels.

    The screen is the big blue-lit region.  It is found by its colour
    (blue, and brighter than the desk) and then reduced to a
    quadrilateral by taking the extremes along the diagonals of its own
    principal axes - for a rectangle, those extremes *are* the corners.
    """
    # the search runs on a copy scaled down to 600 px: the blob walk is
    # a plain Python flood fill, and a 2000x1500 image would take
    # minutes.
    scale = 600.0 / max(img.size)
    small = img.convert("RGB").resize(
        (max(1, int(img.width * scale)), max(1, int(img.height * scale))),
        Image.LANCZOS)
```

```
    a = np.asarray(small).astype(np.int16)
    r, g, b = a[:, :, 0], a[:, :, 1], a[:, :, 2]

    mask = (b - r > 25) & (b > 55) & (g < b - 10)

    # Keep only the biggest blob: the blue PCB next to the screen is
    # blue too, and it is not the screen.
    mask = biggest_blob(mask)

    ys, xs = np.nonzero(mask)
    if len(xs) < 500:
        raise SystemExit("no screen found in the photograph")

    pts = np.stack([xs, ys], axis=1).astype(np.float64)
    centre = pts.mean(axis=0)
    centred = pts - centre
    # principal axes
    cov = centred.T @ centred / len(centred)
    vals, vecs = np.linalg.eigh(cov)
    u = vecs[:, np.argmax(vals)]
    v = vecs[:, np.argmin(vals)]
    a_ = centred @ u
    b_ = centred @ v
    corners = []
    for sa, sb in ((1, 1), (1, -1), (-1, -1), (-1, 1)):
        score = sa * a_ + sb * b_
        k = int(np.argmax(score))
        corners.append(pts[k])
    corners = np.array(corners)

    # the panel is landscape, so the left-most corner pair is the top
    tl = corners[np.argmin(corners[:, 0] + corners[:, 1])]
    br = corners[np.argmax(corners[:, 0] + corners[:, 1])]
    tr = corners[np.argmin(corners[:, 1] - corners[:, 0])]
    bl = corners[np.argmax(corners[:, 1] - corners[:, 0])]
    out = np.array([tl, tr, br, bl], dtype=np.float64)
    return out / scale          # back to the photo's own pixels


def biggest_blob(mask: np.ndarray) -> np.ndarray:
    """The largest connected region of True in the mask."""
    h, w = mask.shape
    labels = np.zeros((h, w), dtype=np.int32)
    current = 0
    best, best_size = 0, 0
    for y0 in range(h):
        for x0 in range(w):
            if mask[y0, x0] and labels[y0, x0] == 0:
                current += 1
                stack = [(y0, x0)]
                labels[y0, x0] = current
                size = 0
                while stack:
                    y, x = stack.pop()
                    size += 1
                    for dy, dx in ((1, 0), (-1, 0), (0, 1), (0, -1)):
                        ny, nx = y + dy, x + dx
                        if 0 <= ny < h and 0 <= nx < w and mask[ny, nx] \
                                and labels[ny, nx] == 0:
                            labels[ny, nx] = current
                            stack.append((ny, nx))
                if size > best_size:
                    best_size, best = size, current
    return labels == best


def rectify(img: Image.Image, corners: np.ndarray) -> np.ndarray:
    """The photograph, mapped back onto the 320x240 the firmware drew."""
    a = np.asarray(img.convert("RGB")).astype(np.float64)
    tl, tr, br, bl = corners
    sx = np.linspace(0, W - 1, W)
    sy = np.linspace(0, H - 1, H)
    u, v = np.meshgrid(sx, sy)
    u = u / (W - 1)
    v = v / (H - 1)
    top = tl + u[..., None] * (tr - tl)
    bot = bl + u[..., None] * (br - bl)
    pts = top + v[..., None] * (bot - top)
    x = np.clip(pts[:, :, 0], 0, a.shape[1] - 1)
    y = np.clip(pts[:, :, 1], 0, a.shape[0] - 1)
    x0 = np.floor(x).astype(int)
    y0 = np.floor(y).astype(int)
    x1 = np.clip(x0 + 1, 0, a.shape[1] - 1)
    y1 = np.clip(y0 + 1, 0, a.shape[0] - 1)
    fx = x - x0
```

```
    fy = y - y0
    out = (a[y0, x0] * (1 - fx)[..., None] * (1 - fy)[..., None]
           + a[y0, x1] * fx[..., None] * (1 - fy)[..., None]
           + a[y1, x0] * (1 - fx)[..., None] * fy[..., None]
           + a[y1, x1] * fx[..., None] * fy[..., None])
    return out


def median_of(rect: np.ndarray, x, y, w, h):
    patch = rect[y:y + h, x:x + w]
    return np.median(patch.reshape(-1, 3), axis=0)


def measure(path: Path):
    img = Image.open(path)
    corners = find_corners(img)
    rect = rectify(img, corners)
    print(f"\n=== {path.name}  ({img.width} x {img.height}) ===")
    print("corners:", np.round(corners).astype(int).tolist())

    OUT.mkdir(parents=True, exist_ok=True)
    Image.fromarray(np.clip(rect, 0, 255).astype(np.uint8)).save(
        OUT / f"photo_{path.stem[:8]}.png")

    measured = {}
    print(f"{'region':26s} {'measured (r,g,b)':>22s}   {'palette':>16s}")
    for name, x, y, w, h, pal in REGIONS:
        m = median_of(rect, x, y, w, h)
        measured[name] = (m, pal)
        want = f"{PALETTE[pal]}" if pal else "-"
        print(f"{name:26s} {str(tuple(int(v) for v in m)):>22s}   {want:>16s}")

    # ---- the veil: a per-channel straight line over the flat areas ----
    pairs = [(m, PALETTE[p]) for m, p in measured.values() if p]
    if len(pairs) >= 3:
        print("\nper-channel fit  photo = a * expected + b  (flat areas)")
        for ch, cname in enumerate("rgb"):
            xs = np.array([p[1][ch] for p in pairs], dtype=np.float64)
            ys = np.array([p[0][ch] for p in pairs], dtype=np.float64)
            a, b = np.polyfit(xs, ys, 1)
            print(f"  {cname}: a = {a:5.3f}   b = {b:7.1f}")
        # the same fit with red and blue exchanged in the *expected*
        print("\nsame fit, but with the expected red and blue exchanged")
        for ch, cname in enumerate("rgb"):
            other = [2, 1, 0][ch]
            xs = np.array([p[1][other] for p in pairs], dtype=np.float64)
            ys = np.array([p[0][ch] for p in pairs], dtype=np.float64)
            a, b = np.polyfit(xs, ys, 1)
            resid = ys - (a * xs + b)
            print(f"  {cname}: a = {a:5.3f}   b = {b:7.1f}"
                  f"   rms residual {np.sqrt((resid ** 2).mean()):6.1f}")
    return rect


def main(argv):
    if len(argv) < 2:
        print(__doc__)
        return 1
    for arg in argv[1:]:
        measure(Path(arg))
    return 0


if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

research/compare_photo.py

121 lines · 4,709 bytes

```
"""Compare a photograph of the panel with the picture the firmware drew.

`measure_photo.py` says *what* colour the glass shows in a few places.
This asks the harder question: *which* colour is it showing, given that
the photograph is veiled and blurred and a few pixels out of place.

It renders the expected screen with `ui_mock.py` (the same layout the
firmware draws), aligns it to the rectified photograph, and then asks,
for every palette entry, what the photograph shows where the firmware
put that entry.  It also fits the two rival hypotheses - the colours as
drawn, and the colours with red and blue exchanged - and reports the
residual of each.
```

```
13
14  Run:  python research/compare_photo.py <photo.jpg>
15  """
16  import sys
17  from pathlib import Path
18
19  import numpy as np
20  from PIL import Image
21
22  sys.path.insert(0, str(Path(__file__).resolve().parent))
23  import ui_mock                                      # noqa: E402
24  from measure_photo import PALETTE, find_corners, rectify   # noqa: E402
25
26  ROOT = Path(__file__).resolve().parent.parent
27  OUT = ROOT / "research" / "out"
28
29
30  def expected_screen(path: Path) -> np.ndarray:
31      """The screen the firmware drew for the state in the photograph.
32
33      One cot answering, two quiet, the radio up, and no caregiver
34      phone attached - which is why the PHONE square is amber.
35      """
36      cots = [
37          {"online": True, "crying": False, "rms": 0.010, "conf": 0},
38          {"online": False, "crying": False, "rms": 0.0, "conf": 0},
39          {"online": False, "crying": False, "rms": 0.0, "conf": 0},
40      ]
41      ui_mock.dashboard(path, cots, True, False, 0, 0, 0.010, 0, True, 23)
42      return np.asarray(Image.open(path).convert("RGB")).astype(np.float64)
43
44
45  def align(photo: np.ndarray, expected: np.ndarray) -> tuple[np.ndarray, int, int]:
46      """Slide the expected screen over the photograph to the best fit."""
47      best = None
48      for dy in range(-8, 9):
49          for dx in range(-8, 9):
50              ys = slice(max(0, dy), 240 + min(0, dy))
51              xs = slice(max(0, dx), 320 + min(0, dx))
52              p = photo[ys, xs]
53              e = expected[max(0, -dy):240 - max(0, dy),
54                           max(0, -dx):320 - max(0, dx)]
55              if p.shape != e.shape:
56                  continue
57              score = -np.abs(p - e).mean()
58              if best is None or score < best[0]:
59                  best = (score, dx, dy)
60      _, dx, dy = best
61      out = np.zeros_like(photo)
62      ys = slice(max(0, dy), 240 + min(0, dy))
63      xs = slice(max(0, dx), 320 + min(0, dx))
64      out[ys, xs] = expected[max(0, -dy):240 - max(0, dy),
65                             max(0, -dx):320 - max(0, dx)]
66      return out, dx, dy
67
68
69  def main(argv):
70      photo_path = Path(argv[1])
71      img = Image.open(photo_path)
72      corners = find_corners(img)
73      photo = rectify(img, corners)
74
75      exp_path = OUT / "_expected.png"
76      exp_path.parent.mkdir(parents=True, exist_ok=True)
77      expected = expected_screen(exp_path)
78      aligned, dx, dy = align(photo, expected)
79      print(f"aligned the expected screen at dx={dx}, dy={dy}")
80
81      print("\nwhat the glass shows where the firmware put each colour")
82      print(f"{'palette entry':12s} {'wanted (r,g,b)':>20s} {'measured':>20s}"
83            f"   px")
84      for name, (r, g, b) in PALETTE.items():
85          want = np.array([r, g, b], dtype=np.float64)
86          d = np.abs(aligned - want).max(axis=2)
87          mask = d < 12
88          n = int(mask.sum())
89          if n < 30:
90              continue
91          got = photo[mask].mean(axis=0)
92          print(f"{name:12s} {str((r, g, b)):>20s} "
93                f"{str(tuple(int(v) for v in got)):>20s}   {n}")
94
95      print("\nhypothesis tests over every pixel")
96      variants = {
97          "as drawn (RGB)": expected,
```

```
        "red and blue exchanged (BGR)": expected[:, :, ::-1],
    }
    for label, variant in variants.items():
        rms = []
        for ch in range(3):
            x = variant[:, :, ch].ravel()
            y = photo[:, :, ch].ravel()
            a, b = np.polyfit(x, y, 1)
            rms.append(float(np.sqrt(((y - (a * x + b)) ** 2).mean())))
        print(f"  {label:30s} rms residual  r {rms[0]:6.1f}  "
              f"g {rms[1]:6.1f}  b {rms[2]:6.1f}   (mean {np.mean(rms):5.1f})")

    # ---- the veil, fitted with the colours as drawn ----
    print("\nthe veil, per channel, with the colours as drawn")
    for ch, cname in enumerate("rgb"):
        x = expected[:, :, ch].ravel()
        y = photo[:, :, ch].ravel()
        a, b = np.polyfit(x, y, 1)
        print(f"  {cname}: photo = {a:5.3f} * expected + {b:6.1f}")
    return 0


if __name__ == "__main__":
    sys.exit(main(sys.argv))
```

Host tools

firmware/tools/flash_hub.py

179 lines · 6,722 bytes

```
#!/usr/bin/env python3
"""Flash the CryNet hub to an ESP32-S3 and print its boot log.

Everything this needs is already on the machine — there is nothing to
download and nothing to install. It finds the board by itself, writes the
three images at the offsets the partition table expects, and then prints
the verification report the firmware prints on boot.

The images must exist; build them first with:

    cd firmware/hub_infer && python -m platformio run

With --merged the three images are combined into a single file first and
written at 0x0, which is what a graphical flasher such as the Arduino
IDE's "ESP32 Flash Tools" extension expects. The combination MUST use
DIO flash mode: the ROM on this board cannot read the flash in QIO
before the bootloader has set up its own timing, and a QIO image makes
the chip watchdog-reset in a loop right after power-on. The bootloader
switches the chip to QIO by itself once it is running, so nothing is
lost.

Usage:
    python firmware/tools/flash_hub.py                # flash, then show the log
    python firmware/tools/flash_hub.py --merged        # one combined file
    python firmware/tools/flash_hub.py --flash-only
    python firmware/tools/flash_hub.py --log-only --seconds 30
    python firmware/tools/flash_hub.py --port COM32
"""

from __future__ import annotations

import argparse
import os
import shutil
import subprocess
import sys
import time
from pathlib import Path

ROOT = Path(__file__).resolve().parents[2]
BUILD = ROOT / "firmware" / "hub_infer" / ".pio" / "build" / "esp32s3"

# (file, offset) — the offsets come from firmware/hub_infer/partitions_16mb.csv
IMAGES = [
    (BUILD / "bootloader.bin", 0x0),
    (BUILD / "partitions.bin", 0x8000),
    (BUILD / "firmware.bin", 0x10000),
]
```

```
49
50 ESP32_VID_PIDS = {
51     (0x1A86, 0x55D3): "CH343 (CH343P/CH9102 USB-serial, common on ESP32-S3 boards)",
52     (0x1A86, 0x7523): "CH340",
53     (0x10C4, 0xEA60): "Silicon Labs CP210x",
54     (0x0403, 0x6001): "FTDI FT232",
55     (0x303A, 0x1001): "Espressif native USB",
56 }
57
58
59 def find_esptool() -> str | None:
60     """Prefer the esptool that ships with PlatformIO; fall back to PATH."""
61     candidates = [
62         Path.home() / ".platformio" / "packages" / "tool-esptool" / "esptool.exe",
63         Path.home() / ".platformio" / "packages" / "tool-esptool" / "esptool",
64     ]
65     for c in candidates:
66         if c.is_file():
67             return str(c)
68     for name in ("esptool", "esptool.exe"):
69         found = shutil.which(name)
70         if found:
71             return found
72     return None
73
74
75 def find_port() -> str | None:
76     try:
77         import serial.tools.list_ports as list_ports
78     except ImportError:
79         return None
80     for p in list_ports.comports():
81         if (p.vid, p.pid) in ESP32_VID_PIDS:
82             print(f"  found {p.device}: {p.description}")
83             return p.device
84     return None
85
86
87 def main() -> int:
88     ap = argparse.ArgumentParser()
89     ap.add_argument("--port", help="serial port; auto-detected if omitted")
90     ap.add_argument("--baud", type=int, default=921600, help="flashing baud")
91     ap.add_argument("--seconds", type=float, default=20.0, help="how long to listen")
92     ap.add_argument("--flash-only", action="store_true")
93     ap.add_argument("--log-only", action="store_true")
94     ap.add_argument("--merged", action="store_true",
95                     help="flash one combined file at 0x0 instead of three files")
96     args = ap.parse_args()
97
98     missing = [str(p) for p, _ in IMAGES if not p.is_file()]
99     if missing and not args.log_only:
100         print("Missing build artefacts:", file=sys.stderr)
101         for m in missing:
102             print("  ", m, file=sys.stderr)
103         print("\nBuild them first:  cd firmware/hub_infer && python -m platformio run",
104               file=sys.stderr)
105         return 1
106
107     port = args.port or find_port()
108     if not port:
109         print("No ESP32 serial port found. Plug the board in, or pass --port COMx.",
110               file=sys.stderr)
111         return 1
112
113     if not args.log_only:
114         esptool = find_esptool()
115         if esptool is None:
116             print("esptool not found (looked in ~/.platformio/packages/tool-esptool "
117                   "and on PATH).", file=sys.stderr)
118             return 1
119
120         print(f"Flashing {port} at {args.baud} baud:")
121         cmd = [esptool, "--chip", "esp32s3", "--port", port, "--baud", str(args.baud),
122                "write-flash"]
123         if args.merged:
124             merged = BUILD / "crynet_hub_merged.bin"
125             if not merged.is_file():
126                 print("  building the combined image ...")
127                 subprocess.run([esptool, "--chip", "esp32s3", "merge-bin",
128                                 "-o", str(merged),
129                                 "--flash-mode", "dio", "--flash-freq", "80m",
130                                 "--flash-size", "16MB",
131                                 "0x0", str(IMAGES[0][0]),
132                                 "0x8000", str(IMAGES[1][0]),
133                                 "0x10000", str(IMAGES[2][0])], check=True)
```

```
            print(f"   0x00000000  {merged.name}  (one file)")
            cmd += ["0x0", str(merged)]
        else:
            for path, offset in IMAGES:
                print(f"   {offset:#08x}  {path.name}")
                cmd += [f"{offset:#x}", str(path)]
        result = subprocess.run(cmd)
        if result.returncode != 0:
            print("Flashing failed.", file=sys.stderr)
            return result.returncode
        print("Flash OK.\n")
        if args.flash_only:
            return 0
        time.sleep(1.0)

    # --- capture the boot log ------------------------------------------
    try:
        import serial
    except ImportError:
        print("pyserial is not installed: pip install pyserial", file=sys.stderr)
        return 1

    print(f"Listening on {port} for {args.seconds:.0f} s ...")
    try:
        ser = serial.Serial(port, 115200, timeout=0.2)
    except Exception as exc:  # noqa: BLE001
        print(f"cannot open {port}: {exc}", file=sys.stderr)
        return 1

    with ser:
        ser.dtr = False
        ser.rts = True       # pulse EN so the boot log is captured
        time.sleep(0.15)
        ser.rts = False
        deadline = time.time() + args.seconds
        while time.time() < deadline:
            chunk = ser.read(4096)
            if chunk:
                sys.stdout.write(chunk.decode("utf-8", errors="replace"))
                sys.stdout.flush()
    print("\n--- done ---")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/flash_node.py

124 lines · 4,832 bytes

```
#!/usr/bin/env python3
"""Build the CryNet node sketch and flash it onto a node over USB.

The node is an ESP32-C3 on a WeAct Studio board, and it is flashed with
`arduino-cli` -- the copy inside the Arduino IDE, which is the only one
installed on this machine.  The command is the one in DECISIONS.md 352:

    arduino-cli compile --fqbn esp32:esp32:weact_studio_esp32c3 <sketch>
    arduino-cli upload  -p <port> --fqbn esp32:esp32:weact_studio_esp32c3 <sketch>

This script does both, finds `arduino-cli` wherever it lives, and takes
the sketch from the repository if it is there and from the Arduino
sketchbook if it is not -- so it works on the second laptop, where the
repository may not be checked out.

The wearable is flashed the same way, with `--sketch wearable`: it is
the same board and the same toolchain, and the two sketches live side by
side in the repository.  `--sketch haptic_test` flashes the bench test for
the wearable's vibration motor (GPIO7, 3 s on / 2 s off); it is a throwaway
image -- flash `--sketch wearable` again afterwards to put the real firmware
back.

Usage:
    python firmware/tools/flash_node.py --port COM38
    python firmware/tools/flash_node.py --port COM38 --dry-run
    python firmware/tools/flash_node.py --port COM30 --sketch wearable
    python firmware/tools/flash_node.py --port COM30 --sketch haptic_test
"""

from __future__ import annotations

```

```
import argparse
import os
import shutil
import subprocess
import sys
from pathlib import Path

ROOT = Path(__file__).resolve().parents[2]
SKETCH_NAME = "node_crynet"
FQBN = "esp32:esp32:weact_studio_esp32c3"
# The sketches this tool can flash, and where they live in the tree.
SKETCHES = {
    "node": ("node", "node_crynet"),
    "wearable": ("wearable", "wearable_crynet"),
    "haptic_test": ("wearable", "haptic_test"),
}


def find_arduino_cli() -> str | None:
    """The arduino-cli to use: the environment first, then the IDE's."""
    env = os.environ.get("ARDUINO_CLI")
    if env and Path(env).exists():
        return env
    found = shutil.which("arduino-cli") or shutil.which("arduino-cli.exe")
    if found:
        return found
    candidates = [
        Path.home() / "AppData/Local/Programs/Arduino IDE/resources/app/lib/"
                      "backend/resources/arduino-cli.exe",
        Path("/Applications/Arduino IDE.app/Contents/Resources/app/lib/backend/"
             "resources/arduino-cli"),
    ]
    for c in candidates:
        if c.exists():
            return str(c)
    return None


def find_sketch(which: str = "node") -> Path:
    """The sketch to build: the repository's, or the sketchbook's copy."""
    folder, name = SKETCHES[which]
    repo = ROOT / "firmware" / folder / name
    if (repo / f"{name}.ino").exists():
        return repo
    book = Path.home() / "Documents/Arduino" / name
    if (book / f"{name}.ino").exists():
        return book
    raise SystemExit(f"no {name} sketch found in {repo} or {book}")


def main() -> int:
    ap = argparse.ArgumentParser(description=__doc__,
                                 formatter_class=argparse.RawDescriptionHelpFormatter)
    ap.add_argument("--port", required=True, help="the node's serial port")
    ap.add_argument("--fqbn", default=FQBN, help="the board to build for")
    ap.add_argument("--sketch", default="node", choices=sorted(SKETCHES),
                    help="which sketch to build and flash")
    ap.add_argument("--dry-run", action="store_true",
                    help="build and upload without erasing the app partition")
    args = ap.parse_args()

    cli = find_arduino_cli()
    if cli is None:
        print("arduino-cli not found: install the Arduino IDE, or set "
              "ARDUINO_CLI to the binary", file=sys.stderr)
        return 1
    sketch = find_sketch(args.sketch)
    print(f"arduino-cli: {cli}")
    print(f"sketch:      {sketch}")

    build = subprocess.run(
        [cli, "compile", "--fqbn", args.fqbn, str(sketch)],
        capture_output=True, text=True)
    sys.stdout.write(build.stdout)
    sys.stderr.write(build.stderr)
    if build.returncode != 0:
        print("the build failed; nothing was flashed", file=sys.stderr)
        return build.returncode

    upload = [cli, "upload", "-p", args.port, "--fqbn", args.fqbn, str(sketch)]
    print("flashing:", " ".join(upload))
    up = subprocess.run(upload, capture_output=True, text=True)
    sys.stdout.write(up.stdout)
    sys.stderr.write(up.stderr)
    if up.returncode != 0:
```

```
        print(f"the flash to {args.port} failed", file=sys.stderr)
        return up.returncode
    print(f"{args.sketch} on {args.port} is running the new firmware")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/flash_unit.py

318 lines · 11,021 bytes

```
#!/usr/bin/env python3
"""Flash a CryNet unit from the prebuilt images in firmware/prebuilt/.

This is the offline path.  It needs exactly one thing on the machine: the
`esptool` executable.  No Arduino core, no PlatformIO, no compiler, no
network -- so a new hub, a new cot node or a new wearable can be brought
up on a laptop that has never seen this project before.

The images are written region by region, exactly where the partition table
says they go, and **NVS is never written**.  That matters: the node's id
lives in NVS, so this keeps the id that is already on the board (and on a
brand new board leaves it untouched, ready to be set with `id <n>`).

    python firmware/tools/flash_unit.py --role hub     --port COM32
    python firmware/tools/flash_unit.py --role node    --port COM31
    python firmware/tools/flash_unit.py --role wearable --port COM30

    python firmware/tools/flash_unit.py --check
    python firmware/tools/flash_unit.py --list
    python firmware/tools/flash_unit.py --role node --port COM31 --backup

If you change the firmware, rebuild the images with `make_prebuilt.py`
(or the compiler commands in FLASHING.md) and run `--check` again: it
compares the SHA-256 of every image against firmware/prebuilt/SHA256SUMS.

Usage:
    python firmware/tools/flash_unit.py --role hub --port COM32 [--backup]
    python firmware/tools/flash_unit.py --role node --port COM31 [--log 20]
"""

from __future__ import annotations

import argparse
import hashlib
import os
import shutil
import subprocess
import sys
import time
from pathlib import Path

ROOT = Path(__file__).resolve().parents[2]
PREBUILT = ROOT / "firmware" / "prebuilt"

# role -> the chip it is, and the images to write, in the order they go.
# (file under firmware/prebuilt/<role>/, offset in flash)
ROLES = {
    "hub": {
        "chip": "esp32s3",
        "baud": 921600,
        "flash_size": 0x1000000,       # 16 MB
        "images": [
            ("bootloader.bin", 0x0),
            ("partitions.bin", 0x8000),
            ("firmware.bin", 0x10000),
        ],
    },
    "node": {
        "chip": "esp32c3",
        "baud": 921600,
        "flash_size": 0x400000,         # 4 MB
        "images": [
            ("bootloader.bin", 0x0),
            ("partitions.bin", 0x8000),
            ("boot_app0.bin", 0xe000),
            ("app.bin", 0x10000),
        ],
    },
    "wearable": {
```

```
        "chip": "esp32c3",
        "baud": 921600,
        "flash_size": 0x400000,         # 4 MB
        "images": [
            ("bootloader.bin", 0x0),
            ("partitions.bin", 0x8000),
            ("boot_app0.bin", 0xe000),
            ("app.bin", 0x10000),
        ],
    },
}


def find_esptool() -> str | None:
    """The esptool to use: PlatformIO's copy first, then PATH."""
    env = os.environ.get("ESPTOOL")
    if env and Path(env).exists():
        return env
    candidates = [
        Path.home() / ".platformio" / "packages" / "tool-esptool" / "esptool.exe",
        Path.home() / ".platformio" / "packages" / "tool-esptool" / "esptool",
        Path.home() / ".platformio" / "packages" / "tool-esptool-py" / "esptool.exe",
    ]
    for c in candidates:
        if c.is_file():
            return str(c)
    for name in ("esptool", "esptool.exe"):
        found = shutil.which(name)
        if found:
            return found
    return None


def sha256(path: Path) -> str:
    h = hashlib.sha256()
    with path.open("rb") as f:
        for chunk in iter(lambda: f.read(1 << 20), b""):
            h.update(chunk)
    return h.hexdigest()


def sums_path() -> Path:
    return PREBUILT / "SHA256SUMS"


def read_sums() -> dict[str, str]:
    """The expected hashes, keyed by the path as written in SHA256SUMS."""
    p = sums_path()
    if not p.is_file():
        return {}
    out = {}
    for line in p.read_text().splitlines():
        line = line.strip()
        if not line or line.startswith("#"):
            continue
        parts = line.split(None, 1)
        if len(parts) == 2:
            out[parts[1].strip().lstrip("*")] = parts[0].lower()
    return out


def role_dir(role: str) -> Path:
    return PREBUILT / role


def role_images(role: str) -> list[tuple[Path, int]]:
    spec = ROLES[role]
    out = []
    for name, offset in spec["images"]:
        out.append((role_dir(role) / name, offset))
    return out


def check(role: str | None) -> int:
    """Everything the offline path needs, and nothing else."""
    print("== toolchain ==")
    esptool = find_esptool()
    if esptool:
        print(f"  esptool     OK   {esptool}")
    else:
        print("  esptool     MISSING  (looked in ~/.platformio/packages/"
              "tool-esptool and on PATH; set ESPTOOL to override)")
    try:
        import serial  # noqa: F401
        print("  pyserial    OK")
```

```
    except ImportError:
        print("  pyserial    MISSING  (pip install pyserial)")

    roles = [role] if role else list(ROLES)
    print("\n== prebuilt images ==")
    expected = read_sums()
    missing = False
    for r in roles:
        print(f"  {r}:")
        for path, offset in role_images(r):
            if not path.is_file():
                print(f"    MISSING  {path.relative_to(ROOT)}")
                missing = True
                continue
            digest = sha256(path)
            want = expected.get(str(path.relative_to(PREBUILT)).replace("\\", "/"))
            if want is None:
                state = "no hash on file"
            elif want == digest:
                state = "hash OK"
            else:
                state = "HASH MISMATCH"
                missing = True
            print(f"    {offset:#08x}  {path.name:<16} {path.stat().st_size:>9} B"
                  f"  {digest[:16]}...  {state}")
    if not expected:
        print("\n  (no SHA256SUMS file -- the images are unverified)")
    return 1 if missing else 0


def list_ports() -> int:
    try:
        import serial.tools.list_ports as list_ports
    except ImportError:
        print("pyserial is not installed: pip install pyserial", file=sys.stderr)
        return 1
    found = False
    for p in list_ports.comports():
        found = True
        print(f"{p.device:<8} {p.description:<40} vid={p.vid} pid={p.pid}")
    if not found:
        print("no serial ports found")
    return 0


def backup(esptool: str, role: str, port: str) -> int:
    spec = ROLES[role]
    dest = Path(os.environ.get("TEMP", ".")) / f"crynet_backup_{role}_{port}.bin"
    cmd = [esptool, "--chip", spec["chip"], "--port", port,
           "read-flash", "0x0", hex(spec["flash_size"]), str(dest)]
    print("backing up:", " ".join(cmd))
    if subprocess.run(cmd).returncode != 0:
        print("the backup failed; not flashing anything", file=sys.stderr)
        return 1
    print(f"backup written to {dest}")
    return 0


def capture_log(port: str, seconds: float) -> None:
    """Reset the board over RTS/EN and print what it says at boot."""
    try:
        import serial
    except ImportError:
        print("pyserial is not installed: cannot read the boot log", file=sys.stderr)
        return
    print(f"\nListening on {port} for {seconds:.0f} s ...")
    try:
        ser = serial.Serial(port, 115200, timeout=0.2)
    except Exception as exc:  # noqa: BLE001
        print(f"cannot open {port}: {exc}", file=sys.stderr)
        return
    with ser:
        ser.dtr = False
        ser.rts = True          # pulse EN so the boot log is captured
        time.sleep(0.15)
        ser.rts = False
        deadline = time.time() + seconds
        while time.time() < deadline:
            chunk = ser.read(4096)
            if chunk:
                sys.stdout.write(chunk.decode("utf-8", errors="replace"))
                sys.stdout.flush()
    print("\n--- done ---")
```

```
def main() -> int:
    ap = argparse.ArgumentParser(
        description=__doc__, formatter_class=argparse.RawDescriptionHelpFormatter)
    ap.add_argument("--role", choices=sorted(ROLES),
                    help="which unit to flash")
    ap.add_argument("--port", help="the unit's serial port, e.g. COM32")
    ap.add_argument("--baud", type=int, default=None, help="flashing baud")
    ap.add_argument("--backup", action="store_true",
                    help="read the whole flash to a file before writing")
    ap.add_argument("--log", type=float, default=0.0, metavar="SECONDS",
                    help="read the boot log for this long after flashing")
    ap.add_argument("--check", action="store_true",
                    help="check the toolchain and the prebuilt images, flash nothing")
    ap.add_argument("--list", action="store_true",
                    help="list the serial ports and exit")
    ap.add_argument("--dry-run", action="store_true",
                    help="print the esptool command and flash nothing")
    args = ap.parse_args()

    if args.list:
        return list_ports()
    if args.check:
        return check(args.role)
    if not args.role or not args.port:
        ap.error("--role and --port are both required to flash "
                 "(a COM number is not an identity -- check the ports with --list)")

    esptool = find_esptool()
    if esptool is None:
        print("esptool not found (looked in ~/.platformio/packages/tool-esptool "
              "and on PATH; set ESPTOOL to override)", file=sys.stderr)
        return 1

    spec = ROLES[args.role]
    images = role_images(args.role)
    for path, _ in images:
        if not path.is_file():
            print(f"missing image {path} -- run --check for the details",
                  file=sys.stderr)
            return 1

    rc = check(args.role)
    print()
    if rc != 0:
        print("the prebuilt images did not check out; not flashing anything",
              file=sys.stderr)
        return 1

    if args.backup:
        rc = backup(esptool, args.role, args.port)
        if rc != 0:
            return rc

    baud = args.baud or spec["baud"]
    cmd = [esptool, "--chip", spec["chip"], "--port", args.port,
           "--baud", str(baud), "write-flash"]
    for path, offset in images:
        cmd += [hex(offset), str(path)]

    print("flashing the", args.role, "on", args.port)
    for path, offset in images:
        print(f"   {offset:#08x}  {path.name}")
    if args.dry_run:
        print("\ndry run -- the command that would run is:\n")
        print("  " + " ".join(cmd))
        return 0
    result = subprocess.run(cmd)
    if result.returncode != 0:
        print("the flash failed", file=sys.stderr)
        return result.returncode
    print("\nflash OK")
    if args.log > 0:
        time.sleep(1.0)
        capture_log(args.port, args.log)
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/install_sketches.py

79 lines · 2,756 bytes

```
#!/usr/bin/env python3
"""Copy the CryNet sketches into the Arduino sketchbook so the IDE can open them.

The Arduino IDE builds a sketch from its own folder, and the sketches live
in the repository, so they have to be copied into the sketchbook (the
"node_crynet" and "node_capture" folders) before they can be compiled and
uploaded. This copies them and prints what to do next.

Usage:
    python firmware/tools/install_sketches.py
    python firmware/tools/install_sketches.py --list
"""

from __future__ import annotations

import argparse
import shutil
import sys
from pathlib import Path

ROOT = Path(__file__).resolve().parents[2]
# (subdirectory under firmware/, sketch name) — the wearable lives outside
# firmware/node but installs into the sketchbook the same way.
SKETCHES = [("node", "node_crynet"), ("node", "node_capture"),
            ("wearable", "wearable_crynet")]


def sketchbook() -> Path:
    return Path.home() / "Documents" / "Arduino"


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--list", action="store_true", help="list the sketches and exit")
    ap.add_argument("--remove", action="store_true",
                    help="remove the sketches from the sketchbook instead")
    args = ap.parse_args()

    dst_root = sketchbook()

    if args.list:
        for subdir, name in SKETCHES:
            src = ROOT / "firmware" / subdir / name
            dst = dst_root / name
            print(f"{name:<18} {'present' if (src / (name + '.ino')).is_file() else 'MISSING'}"
                  f"  ->  {dst}{' (installed)' if dst.is_dir() else ''}")
        return 0

    if not dst_root.is_dir():
        print(f"The sketchbook {dst_root} does not exist. Open the Arduino IDE "
              f"once so it creates it.", file=sys.stderr)
        return 1

    for subdir, name in SKETCHES:
        src = ROOT / "firmware" / subdir / name
        dst = dst_root / name
        if not (src / (name + ".ino")).is_file():
            print(f"missing {src / (name + '.ino')}", file=sys.stderr)
            return 1
        if args.remove:
            if dst.is_dir():
                shutil.rmtree(dst)
            print(f"removed {dst}")
            continue
        if dst.is_dir():
            shutil.rmtree(dst)
        shutil.copytree(src, dst)
        print(f"installed {dst}")

    print("\nIn the Arduino IDE: File > Open, pick the sketch, then set\n"
          "  Tools > Board            WeAct Studio ESP32C3\n"
          "  Tools > Port             the node's port (COM30)\n"
          "  Tools > USB CDC On Boot  Enabled\n"
          "and press Upload.")
    return 0


```

```
if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/check_artifact_drift.py

160 lines · 6,667 bytes

```
#!/usr/bin/env python3
"""Check that the firmware and the training artifacts agree.

Run this after dropping a new set of Kaggle outputs into research/out/ and
before building the firmware. It catches the failure mode that is easy to
miss: the notebook re-calibrates the threshold and re-derives the
quantisation scales on every run, so the constants mirrored into the
firmware headers go stale silently.

Checks
  1. research/out/crynet_int8.tflite SHA-256 == deploy_config.json model_sha256
  2. model_data.h embeds exactly model_bytes bytes
  3. crynet_tables.h defines == the macros mirrored into crynet_model.h
  4. deploy_config.json agrees with crynet_tables.h on scale/zero-point/threshold
  5. the staged copies under firmware/hub_infer/components are byte-identical
     to the ones in research/out/

Exit code 0 = consistent, 1 = drift found (the message says what moved).
"""

from __future__ import annotations

import hashlib
import json
import re
import sys
from pathlib import Path

ROOT = Path(__file__).resolve().parents[2]
OUT = ROOT / "research" / "out"
FW = ROOT / "firmware" / "hub_infer" / "components"

TABLES_H = OUT / "crynet_tables.h"
MODEL_H = OUT / "model_data.h"
TFLITE = OUT / "crynet_int8.tflite"
CONFIG = OUT / "deploy_config.json"

STAGED = {
    FW / "crynet_model" / "include" / "crynet_tables.h": TABLES_H,
    FW / "crynet_model" / "include" / "model_data.h": MODEL_H,
    FW / "crynet_model" / "include" / "deploy_config.json": CONFIG,
    FW / "crynet_dsp" / "reference" / "dsp_crynet.c": OUT / "dsp_crynet.c",
    FW / "crynet_dsp" / "reference" / "dsp_crynet.h": OUT / "dsp_crynet.h",
}

# macro in crynet_tables.h  ->  macro in crynet_model.h
MIRRORED = {
    "CRYNET_THRESHOLD": "CRYNET_THRESHOLD",
    "CRYNET_SILENCE_RMS": "CRYNET_SILENCE_RMS_GATE",
    "CRYNET_IN_SCALE": "CRYNET_INPUT_SCALE",
    "CRYNET_IN_ZP": "CRYNET_INPUT_ZERO_POINT",
    "CRYNET_OUT_SCALE": "CRYNET_OUTPUT_SCALE",
    "CRYNET_OUT_ZP": "CRYNET_OUTPUT_ZERO_POINT",
    "CRYNET_N_SAMPLES": "CRYNET_N_SAMPLES",
    "CRYNET_N_FRAMES": "CRYNET_N_FRAMES",
    "CRYNET_N_MELS": "CRYNET_N_MELS",
}


def fail(problems: list[str], message: str) -> None:
    problems.append(message)


def defines(path: Path) -> dict[str, str]:
    text = path.read_text(encoding="utf-8")
    out: dict[str, str] = {}
    for name, value in re.findall(r"#define\s+(CRYNET_\w+)\s+([^\r\n/]+)", text):
        out[name] = value.strip().rstrip(")").strip() if False else value.strip()
    return out


def numeric(value: str) -> float:
    """Evaluate a C constant expression that is just a number (or a negation)."""
    v = value.strip().rstrip("f").strip()
    if v.startswith("(") and v.endswith(")"):
```

```
        v = v[1:-1]
    return float(v)


def main() -> int:
    problems: list[str] = []

    for p in (TABLES_H, MODEL_H, TFLITE, CONFIG):
        if not p.exists():
            print(f"FAIL: missing {p}")
            return 1

    config = json.loads(CONFIG.read_text(encoding="utf-8"))
    tables = defines(TABLES_H)
    model_header = (FW / "crynet_model" / "include" / "crynet_model.h").read_text(
        encoding="utf-8"
    )
    mirrored = defines(FW / "crynet_model" / "include" / "crynet_model.h")

    # 1. model hash
    digest = hashlib.sha256(TFLITE.read_bytes()).hexdigest()
    if digest != config["model_sha256"]:
        fail(problems, f"crynet_int8.tflite sha256 {digest} != deploy_config {config['model_sha256']}")

    # 2. embedded byte count
    blob = MODEL_H.read_text(encoding="utf-8")
    body = blob.split("crynet_model[] = {", 1)[-1].split("};", 1)[0]
    n_bytes = len(re.findall(r"0x[0-9a-fA-F]{2}", body))
    if n_bytes != config["model_bytes"]:
        fail(problems, f"model_data.h holds {n_bytes} bytes, deploy_config says {config['model_bytes']}")
    if n_bytes != TFLITE.stat().st_size:
        fail(problems, f"model_data.h holds {n_bytes} bytes, tflite file is {TFLITE.stat().st_size}")

    # 3. mirrored macros
    for src, dst in MIRRORED.items():
        if src not in tables:
            fail(problems, f"crynet_tables.h does not define {src}")
            continue
        if dst not in mirrored:
            fail(problems, f"crynet_model.h does not define {dst}")
            continue
        a, b = numeric(tables[src]), numeric(mirrored[dst])
        if abs(a - b) > 1e-9:
            fail(problems, f"{src} = {a} in crynet_tables.h but {dst} = {b} in crynet_model.h")

    # 4. deploy_config vs tables
    inp = config["input"]
    if abs(float(inp["scale"]) - numeric(tables["CRYNET_IN_SCALE"])) > 1e-8:
        fail(problems, "deploy_config input scale disagrees with crynet_tables.h")
    if int(inp["zero_point"]) != int(numeric(tables["CRYNET_IN_ZP"])):
        fail(problems, "deploy_config input zero_point disagrees with crynet_tables.h")
    if abs(float(config["decision"]["threshold"]) - numeric(tables["CRYNET_THRESHOLD"])) > 1e-6:
        fail(problems, "deploy_config threshold disagrees with crynet_tables.h")
    if abs(float(config["decision"]["silence_rms_gate"]) - numeric(tables["CRYNET_SILENCE_RMS"])) > 1e-9:
        fail(problems, "deploy_config silence gate disagrees with crynet_tables.h")
    if list(inp["shape"]) != [1, 81, 64, 1]:
        fail(problems, f"unexpected input shape {inp['shape']}")
    if config["tflm_kernel_check"] != "ok":
        fail(problems, "deploy_config does not confirm the TFLite Micro kernel check")

    # 5. staged copies are current
    for staged, source in STAGED.items():
        if not staged.exists():
            fail(problems, f"staged copy missing: {staged}")
        elif staged.read_bytes() != source.read_bytes():
            fail(problems, f"staged copy is stale: {staged.relative_to(ROOT)} != {source.relative_to(ROOT)}")

    if problems:
        print("ARTIFACT DRIFT DETECTED")
        for p in problems:
            print(f"  - {p}")
        print("\nFix by re-staging research/out/ into firmware/hub_infer/components/ and "
              "updating the mirrored macros in crynet_model.h.")
        return 1

    print("OK  artifacts, firmware constants and deploy_config all agree")
    print(f"    model      : {n_bytes} bytes  sha256 {digest[:16]}...")
    print(f"    threshold  : {numeric(tables['CRYNET_THRESHOLD'])}")
    print(f"    input quant: scale {numeric(tables['CRYNET_IN_SCALE'])} zp {int(numeric(tables['CRYNET_IN_ZP']))}")
    print(f"    staged     : {len(STAGED)} files identical to research/out/")
    return 0


```

```
if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/check_prebuilt.py

194 lines · 7,386 bytes

```
#!/usr/bin/env python3
"""Prove that the offline flash images still match the firmware source.

The images in `firmware/prebuilt/` are what a unit is flashed with, offline,
with nothing but `esptool`.  Nothing forces them to stay in step with the
source -- a `.ino` can be edited and committed while the `.bin` files stay
old -- and the result would be a board running code that is not in the
repository.  This is the check that catches it.

Two things are checked:

  1. the images match `firmware/prebuilt/SHA256SUMS` (they are intact);
  2. the source that produced them is still the source in the tree
     (they are up to date).

The second one works from the record that `make_prebuilt.py` wrote:
`firmware/prebuilt/BUILD_INFO.json` holds, for each unit, the hash of
every source file that went into it and the rules used to find those
files.  The check recomputes them and compares.  It is the same on every
machine -- text files are hashed with their line endings normalised, so a
clone that checked out CRLF gives the same answer as one that did not.

    python firmware/tools/check_prebuilt.py        # check
    python firmware/tools/check_prebuilt.py --quiet

Exit code 0 = everything is in step, 1 = something is not.

If it says the images are stale, rebuild them and commit the result:

    python firmware/tools/make_prebuilt.py
"""

from __future__ import annotations

import argparse
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).resolve().parent))

from prebuilt_lib import (  # noqa: E402
    BUILD_INFO,
    PREBUILT,
    ROOT,
    file_hashes,
    fingerprint,
    image_hashes,
    read_build_info,
    sha256_file,
)

ROLES = ("hub", "node", "wearable")


def read_sums() -> dict[str, str]:
    """The expected hashes, keyed by the path as written in SHA256SUMS."""
    sums = PREBUILT / "SHA256SUMS"
    if not sums.is_file():
        return {}
    out: dict[str, str] = {}
    for line in sums.read_text(encoding="utf-8").splitlines():
        line = line.strip()
        if not line or line.startswith("#"):
            continue
        parts = line.split(None, 1)
        if len(parts) != 2:
            continue
        out[parts[1].strip().replace("\\", "/")] = parts[0].strip().lower()
    return out


def check_images(quiet: bool) -> list[str]:
    """Every image on disk matches SHA256SUMS, and vice versa."""
    problems: list[str] = []
    expected = read_sums()
```

```
    if not expected:
        return ["SHA256SUMS is missing or empty - the images cannot be verified"]

    on_disk = {p.relative_to(PREBUILT).as_posix() for p in PREBUILT.rglob("*.bin")}
    for rel in sorted(set(expected) - on_disk):
        problems.append(f"missing image: {rel}")
    for rel in sorted(on_disk - set(expected)):
        problems.append(f"image on disk but not in SHA256SUMS: {rel}")

    for rel, want in sorted(expected.items()):
        path = PREBUILT / rel
        if not path.is_file():
            continue
        got = sha256_file(path)
        if got != want:
            problems.append(f"damaged image: {rel}\n"
                            f"    expected {want}\n"
                            f"    got      {got}")
    if not problems and not quiet:
        print(f"  images: {len(expected)} files, all match SHA256SUMS")
    return problems


def check_source(quiet: bool) -> list[str]:
    """Each unit's source is the source its images were built from."""
    problems: list[str] = []
    info = read_build_info()
    if not info:
        return ["BUILD_INFO.json is missing - the images do not say which "
                "source they came from.\n"
                "    rebuild them:  python firmware/tools/make_prebuilt.py"]

    rules = info.get("rules")
    recorded = info.get("roles", {})
    for role in ROLES:
        entry = recorded.get(role)
        if not isinstance(entry, dict):
            problems.append(f"{role}: BUILD_INFO.json has no record of this unit")
            continue

        # The source may not be here.  The prebuilt folder is meant to be
        # copyable on its own -- onto a laptop with just esptool on it --
        # and then there is nothing to compare against.  That is not a
        # fault; it just means this check cannot be made.
        roots = [ROOT / r for r in (entry.get("roots") or [])]
        if not roots or not all(r.is_dir() for r in roots):
            if not quiet:
                print(f"  {role:<9} source not here - freshness not checked")
            continue

        want = entry.get("source_fingerprint")
        hashes = file_hashes(role, rules)
        got = fingerprint(hashes)
        if want != got:
            # name the files that moved, so the message says what to look at
            old = entry.get("files") or {}
            added = sorted(set(hashes) - set(old))
            removed = sorted(set(old) - set(hashes))
            changed = sorted(rel for rel in set(old) & set(hashes)
                             if old[rel] != hashes[rel])
            detail = []
            for rel in changed:
                detail.append(f"      changed  {rel}")
            for rel in added:
                detail.append(f"      added    {rel}")
            for rel in removed:
                detail.append(f"      removed  {rel}")
            problems.append(
                f"{role}: the source has changed since these images were built\n"
                f"    built from {str(want)[:16]}... over {len(old)} files\n"
                f"    source now  {got[:16]}... over {len(hashes)} files\n"
                + ("\n".join(detail) + "\n" if detail else "")
                + "    rebuild:     python firmware/tools/make_prebuilt.py")
            continue

        # the recorded image hashes must still be the images on disk
        for name, h in (entry.get("images") or {}).items():
            path = PREBUILT / role / name
            if not path.is_file():
                problems.append(f"{role}: {name} is gone")
            elif sha256_file(path) != h:
                problems.append(f"{role}: {name} has changed since it was built")
        if not quiet:
            print(f"  {role:<9} source fingerprint {got[:16]}... over "
                  f"{len(hashes)} files - up to date")
```

```
    return problems


def main() -> int:
    ap = argparse.ArgumentParser(
        description=__doc__, formatter_class=argparse.RawDescriptionHelpFormatter)
    ap.add_argument("--quiet", action="store_true",
                    help="only print problems")
    args = ap.parse_args()

    if not args.quiet:
        print("== the offline images vs the source ==")

    problems = check_images(args.quiet) + check_source(args.quiet)

    if problems:
        print()
        print("NOT IN STEP")
        for p in problems:
            print(f"  - {p}")
        print()
        print("Fix it by rebuilding the images and committing them:")
        print("    python firmware/tools/make_prebuilt.py")
        return 1

    if not args.quiet:
        print()
        print(f"OK  the offline images are up to date with the source "
              f"({BUILD_INFO.relative_to(ROOT)} records the build)")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/make_prebuilt.py

263 lines · 9,657 bytes

```
#!/usr/bin/env python3
"""Rebuild the prebuilt flash images in firmware/prebuilt/ from the source.

You only need this when you change the firmware.  Flashing a unit never
needs it -- the images are in the repository, so `flash_unit.py` works on
a machine that has never compiled anything.

What it does, per role:

  hub       PlatformIO / ESP-IDF build of firmware/hub_infer
            (needs the vendored esp-tflite-micro and esp-nn components --
             `bash firmware/tools/fetch_deps.sh` once, which needs network)

  node      arduino-cli compile of firmware/node/node_crynet
  wearable  arduino-cli compile of firmware/wearable/wearable_crynet
            (needs the ESP32 core, already installed)

It then writes firmware/prebuilt/SHA256SUMS, which `flash_unit.py --check`
verifies before every flash, and firmware/prebuilt/BUILD_INFO.json, which
records which source each unit was built from -- that is what
`check_prebuilt.py` uses to prove the shipped images are still up to date.

    python firmware/tools/make_prebuilt.py                # everything
    python firmware/tools/make_prebuilt.py --role node
"""

from __future__ import annotations

import argparse
import hashlib
import os
import shutil
import subprocess
import sys
from pathlib import Path

sys.path.insert(0, str(Path(__file__).resolve().parent))

import prebuilt_lib  # noqa: E402

ROOT = Path(__file__).resolve().parents[2]
PREBUILT = ROOT / "firmware" / "prebuilt"
BUILD = ROOT / "firmware" / ".build"
```

```

FQBN = "esp32:esp32:weact_studio_esp32c3"

SKETCHES = {
    "node": ("node", "node_crynet"),
    "wearable": ("wearable", "wearable_crynet"),
}


def find_arduino_cli() -> str | None:
    env = os.environ.get("ARDUINO_CLI")
    if env and Path(env).exists():
        return env
    found = shutil.which("arduino-cli") or shutil.which("arduino-cli.exe")
    if found:
        return found
    for c in [Path.home() / "AppData/Local/Programs/Arduino IDE/resources/app/"
                            "lib/backend/resources/arduino-cli.exe",
              Path("/Applications/Arduino IDE.app/Contents/Resources/app/lib/"
                   "backend/resources/arduino-cli")]:
        if c.exists():
            return str(c)
    return None


def find_platformio() -> str | None:
    env = os.environ.get("PLATFORMIO")
    if env and Path(env).exists():
        return env
    found = shutil.which("platformio") or shutil.which("platformio.exe")
    if found:
        return found
    for c in [Path.home() / ".platformio/penv/Scripts/platformio.exe",
              Path.home() / ".platformio/penv/bin/platformio"]:
        if c.exists():
            return str(c)
    return None


def sha256(path: Path) -> str:
    h = hashlib.sha256()
    with path.open("rb") as f:
        for chunk in iter(lambda: f.read(1 << 20), b""):
            h.update(chunk)
    return h.hexdigest()


def git_describe() -> str | None:
    """The version string the build should have baked into its banner.

    ESP-IDF bakes the app version and the compile time at configure time,
    so an incremental build can wear a banner from an older git state
    (D-106).  None means git is unavailable -- the check then skips.
    """
    try:
        out = subprocess.run(
            ["git", "describe", "--always", "--tags", "--dirty"],
            cwd=ROOT, capture_output=True, timeout=30, check=False)
    except (OSError, subprocess.SubprocessError):
        return None
    if out.returncode != 0:
        return None
    return out.stdout.decode().strip() or None


def build_hub() -> int:
    pio = find_platformio()
    if pio is None:
        print("platformio not found (looked on PATH and in "
              "~/.platformio/penv/Scripts)", file=sys.stderr)
        return 1
    project = ROOT / "firmware" / "hub_infer"
    for dep in ("esp-tflite-micro", "esp-nn"):
        if not (project / "components" / dep).is_dir():
            print(f"missing components/{dep} -- run "
                  f"`bash firmware/tools/fetch_deps.sh` once (needs network)",
                  file=sys.stderr)
            return 1
    # ESP-IDF bakes the app version and the compile time at configure time,
    # and an incremental build happily reuses a configure from an older git
    # state -- the unit then wears a banner that lies about which build it
    # runs (D-106).  Touching CMakeLists forces the reconfigure that
    # regenerates the stamp; the check below refuses an image that does not
    # carry it.
    cmake = project / "CMakeLists.txt"
```

```
    if cmake.is_file():
        os.utime(cmake, None)
    print("building the hub with PlatformIO (this takes a few minutes) ...")
    rc = subprocess.run([pio, "run"], cwd=project).returncode
    if rc != 0:
        return rc
    built = project / ".pio" / "build" / "esp32s3"
    want = git_describe()
    if want:
        base = want.removesuffix("-dirty")
        if base.encode() not in (built / "firmware.bin").read_bytes():
            print("the built firmware.bin does not carry the current app "
                  f"version ({want}) -- the build kept a stale configure's "
                  "banner.  Delete firmware/hub_infer/.pio/build, build "
                  "again, then package.", file=sys.stderr)
            return 1
    out = PREBUILT / "hub"
    out.mkdir(parents=True, exist_ok=True)
    for name in ("bootloader.bin", "partitions.bin", "firmware.bin"):
        src = built / name
        if not src.is_file():
            print(f"the build did not produce {name}", file=sys.stderr)
            return 1
        shutil.copy2(src, out / name)
        print(f"  {name}")
    return 0


def build_sketch(role: str) -> int:
    cli = find_arduino_cli()
    if cli is None:
        print("arduino-cli not found (install the Arduino IDE, or set "
              "ARDUINO_CLI to the binary)", file=sys.stderr)
        return 1
    folder, name = SKETCHES[role]
    sketch = ROOT / "firmware" / folder / name
    if not (sketch / f"{name}.ino").is_file():
        print(f"missing {sketch / (name + '.ino')}", file=sys.stderr)
        return 1
    build_path = BUILD / role
    build_path.mkdir(parents=True, exist_ok=True)
    # -j 1: the parallel build runs out of memory on a 16 GB machine.
    print(f"building the {role} with arduino-cli ...")
    cmd = [cli, "compile", "-j", "1", "--fqbn", FQBN,
           "--build-path", str(build_path), str(sketch)]
    rc = subprocess.run(cmd).returncode
    if rc != 0:
        return rc
    out = PREBUILT / role
    out.mkdir(parents=True, exist_ok=True)
    wanted = {
        "bootloader.bin": build_path / f"{name}.ino.bootloader.bin",
        "partitions.bin": build_path / f"{name}.ino.partitions.bin",
        "boot_app0.bin": build_path / "boot_app0.bin",
        "app.bin": build_path / f"{name}.ino.bin",
    }
    for out_name, src in wanted.items():
        if not src.is_file():
            print(f"the build did not produce {src.name}", file=sys.stderr)
            return 1
        shutil.copy2(src, out / out_name)
        print(f"  {out_name}")
    return 0


def tool_versions() -> dict[str, str]:
    """What built these images, so the record says."""
    versions: dict[str, str] = {}
    pio = find_platformio()
    if pio:
        try:
            out = subprocess.run([pio, "--version"], capture_output=True,
                                 timeout=60, check=False)
            if out.returncode == 0:
                versions["platformio"] = out.stdout.decode().strip().splitlines()[0]
        except (OSError, subprocess.SubprocessError):
            pass
    cli = find_arduino_cli()
    if cli:
        try:
            out = subprocess.run([cli, "version"], capture_output=True,
                                 timeout=60, check=False)
            if out.returncode == 0:
                versions["arduino-cli"] = out.stdout.decode().strip().splitlines()[0]
        except (OSError, subprocess.SubprocessError):
```

```
            pass
    versions["fqbn"] = FQBN
    return versions


def write_sums() -> None:
    lines = [
        "# SHA-256 of the prebuilt flash images.",
        "# Verify them with: python firmware/tools/flash_unit.py --check",
        "",
    ]
    for path in sorted(PREBUILT.rglob("*.bin")):
        rel = path.relative_to(PREBUILT).as_posix()
        lines.append(f"{sha256(path)}  {rel}")
    (PREBUILT / "SHA256SUMS").write_text("\n".join(lines) + "\n")
    print(f"wrote {PREBUILT / 'SHA256SUMS'}")


def write_build_info() -> None:
    """Record which source each unit was built from.

    All three units are recorded, not only the ones just built: a partial
    rebuild must not drop the record of the others.
    """
    path = prebuilt_lib.write_build_info(tools=tool_versions())
    print(f"wrote {path}")


def main() -> int:
    ap = argparse.ArgumentParser(
        description=__doc__, formatter_class=argparse.RawDescriptionHelpFormatter)
    ap.add_argument("--role", choices=["hub", "node", "wearable"],
                    help="build only this one (default: all three)")
    args = ap.parse_args()

    roles = [args.role] if args.role else ["hub", "node", "wearable"]
    for role in roles:
        print(f"== {role} ==")
        rc = build_hub() if role == "hub" else build_sketch(role)
        if rc != 0:
            return rc
        print()

    write_sums()
    write_build_info()
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/prebuilt_lib.py

310 lines · 11,291 bytes

```
"""The rules that decide what goes into each prebuilt image.

`firmware/prebuilt/BUILD_INFO.json` is written when the images are built.
It records, for each unit:

  * which source files went into it, and the SHA-256 of each one,
  * the rules used to find those files, so the same question can be asked
    later by another tool (the PowerShell check, or CI) without this
    module;
  * the SHA-256 of every image that was produced.

The rules are written into the file on purpose.  A check that has to know
the rules must be able to read them from somewhere, and reading them from
the record it is checking is the only way to be sure the two agree.

Everything here is deliberately machine-independent:

  * only files that belong to the unit are included (no build output);
  * text files are hashed with their line endings normalised to LF, so a
    clone that checked out CRLF gives the same answer as one that did not;
  * binary files are hashed as they are.
"""

from __future__ import annotations

import fnmatch
import hashlib
```

```
import json
import subprocess
from datetime import datetime
from pathlib import Path

ROOT = Path(__file__).resolve().parents[2]
PREBUILT = ROOT / "firmware" / "prebuilt"
BUILD_INFO = PREBUILT / "BUILD_INFO.json"

# Where the build was made and by what.  Kept out of BUILD_INFO.json on
# purpose: that file must be the same on every machine, so that CI can
# rebuild and compare it byte for byte.  This one is local only and is
# not committed.
BUILD_STATE = PREBUILT / ".build-state.json"

# role -> (directories that hold its source, extra files it also depends on)
ROLE_SOURCES: dict[str, tuple[list[Path], list[Path]]] = {
    # The hub is ESP-IDF.  The two vendored components (esp-tflite-micro,
    # esp-nn) are not in the repository -- they are pinned by revision in
    # fetch_deps.sh, so that file is part of the hub's fingerprint.
    "hub": (
        [ROOT / "firmware" / "hub_infer"],
        [ROOT / "firmware" / "tools" / "fetch_deps.sh",
         ROOT / "firmware" / "tools" / "patch_deps.py"],
    ),
    "node": (
        [ROOT / "firmware" / "node" / "node_crynet"],
        [],
    ),
    "wearable": (
        [ROOT / "firmware" / "wearable" / "wearable_crynet"],
        [],
    ),
}

# The rules, in a form that another tool can read straight out of the
# record.  Keep them here and in one place.
RULES: dict[str, object] = {
    # Directories that hold build output or vendored clones, not source.
    "exclude_dirs": [
        ".pio", "build", "managed_components", ".git", "__pycache__",
        "esp-tflite-micro", "esp-nn",
    ],
    # Files the IDEs write into the tree.  They are git-ignored, so a git
    # checkout never sees them; but a copy unpacked from a zip does, and
    # they must not change the answer.
    "exclude_names": [
        "c_cpp_properties.json", "launch.json", "ipch",
    ],
    # Name patterns to skip ...
    "exclude_globs": [
        "sdkconfig*", ".browse.c_cpp.db*",
    ],
    # ... except these, which are part of the build.
    "keep_globs": [
        "sdkconfig.defaults",
    ],
}


def sha256_bytes(data: bytes) -> str:
    return hashlib.sha256(data).hexdigest()


def sha256_file(path: Path) -> str:
    h = hashlib.sha256()
    with path.open("rb") as f:
        for chunk in iter(lambda: f.read(1 << 20), b""):
            h.update(chunk)
    return h.hexdigest()


def normalise(data: bytes) -> bytes:
    """Line-ending-normalise text; leave binary alone."""
    if b"\x00" in data:
        return data
    return data.replace(b"\r\n", b"\n")


def file_hash(path: Path) -> str:
    """The hash of a source file, the same on every machine."""
    return sha256_bytes(normalise(path.read_bytes()))


def _git_tracked(paths: list[Path]) -> set[str] | None:
```

```
    """The tracked files under `paths`, as repo-relative POSIX paths.

    Returns None when git cannot answer (not a repository, git missing),
    so the caller falls back to walking the tree.
    """
    rels = []
    for p in paths:
        try:
            rels.append(p.relative_to(ROOT).as_posix())
        except ValueError:
            return None
    try:
        out = subprocess.run(
            ["git", "-C", str(ROOT), "ls-files", "-z", "--", *rels],
            capture_output=True, timeout=60, check=False,
        )
    except (OSError, subprocess.SubprocessError):
        return None
    if out.returncode != 0:
        return None
    files = {n for n in out.stdout.decode("utf-8", "replace").split("\0") if n}
    return files or None


def _git_ignored(paths: list[Path]) -> set[str] | None:
    """The files under `paths` that git ignores.

    Untracked-but-not-ignored files are *source*: a component that has
    been written but not yet committed is still part of the build, and
    the guarantee that the shipped image matches the source has to
    cover it.  So it is git's ignored files that are left out, not its
    untracked ones.

    Returns None when git cannot answer, so the caller falls back to
    what it knows.
    """
    rels = []
    for p in paths:
        try:
            rels.append(p.relative_to(ROOT).as_posix())
        except ValueError:
            return None
    try:
        out = subprocess.run(
            ["git", "-C", str(ROOT), "ls-files", "-z", "--others",
             "--ignored", "--exclude-standard", "--", *rels],
            capture_output=True, timeout=60, check=False,
        )
    except (OSError, subprocess.SubprocessError):
        return None
    if out.returncode != 0:
        return None
    return {n for n in out.stdout.decode("utf-8", "replace").split("\0") if n}


def _excluded(path: Path, base: Path, rules: dict) -> bool:
    rel_parts = path.relative_to(base).parts
    if any(part in rules["exclude_dirs"] for part in rel_parts[:-1]):
        return True
    name = path.name
    if name in rules["exclude_names"]:
        return True
    for pattern in rules["exclude_globs"]:
        if fnmatch.fnmatch(name, pattern):
            for keep in rules["keep_globs"]:
                if fnmatch.fnmatch(name, keep):
                    return False
            return True
    return False


def source_files(role: str, rules: dict | None = None) -> list[str]:
    """The repo-relative files that make up a unit, sorted."""
    rules = rules or RULES
    dirs, extras = ROLE_SOURCES[role]
    tracked = _git_tracked(dirs)
    ignored = _git_ignored(dirs)
    found: set[str] = set()

    for base in dirs:
        if not base.is_dir():
            continue
        for path in base.rglob("*"):
            if not path.is_file():
                continue
```

```
            if _excluded(path, base, rules):
                continue
            rel = path.relative_to(ROOT).as_posix()
            if tracked is not None:
                # Untracked is not the same as ignored.  A file git
                # ignores is build output or an IDE's scratch file; a
                # file git merely has not been told about is source
                # that has not been committed yet, and the shipped
                # image has to be checked against it.
                if rel not in tracked and (ignored is None or rel in ignored):
                    continue
            found.add(rel)

    for extra in extras:
        if extra.is_file():
            found.add(extra.relative_to(ROOT).as_posix())

    return sorted(found)


def file_hashes(role: str, rules: dict | None = None) -> dict[str, str]:
    """The hash of every source file of a unit, keyed by repo-relative path."""
    return {rel: file_hash(ROOT / rel)
            for rel in source_files(role, rules)}


def fingerprint(hashes: dict[str, str]) -> str:
    """One hash for a whole unit's source."""
    h = hashlib.sha256()
    for rel, digest in sorted(hashes.items()):
        h.update(rel.encode("utf-8"))
        h.update(b"\x00")
        h.update(digest.encode("ascii"))
        h.update(b"\n")
    return h.hexdigest()


def git_state() -> dict[str, str]:
    """Which commit the working tree is at, and whether it is dirty."""
    state: dict[str, str] = {"commit": "unknown", "branch": "unknown", "dirty": "?"}
    try:
        out = subprocess.run(["git", "-C", str(ROOT), "rev-parse", "--short", "HEAD"],
                             capture_output=True, timeout=30, check=False)
        if out.returncode == 0:
            state["commit"] = out.stdout.decode().strip()
        out = subprocess.run(["git", "-C", str(ROOT), "rev-parse", "--abbrev-ref", "HEAD"],
                             capture_output=True, timeout=30, check=False)
        if out.returncode == 0:
            state["branch"] = out.stdout.decode().strip()
        out = subprocess.run(["git", "-C", str(ROOT), "status", "--porcelain",
                              "--", "firmware"],
                             capture_output=True, timeout=60, check=False)
        if out.returncode == 0:
            state["dirty"] = "yes" if out.stdout.strip() else "no"
    except (OSError, subprocess.SubprocessError):
        pass
    return state


def image_hashes(role: str) -> dict[str, str]:
    """Hash of every built image of a unit, keyed by file name."""
    folder = PREBUILT / role
    if not folder.is_dir():
        return {}
    return {p.name: sha256_file(p) for p in sorted(folder.glob("*.bin"))}


def write_build_info(roles: list[str] | None = None,
                     tools: dict[str, str] | None = None) -> Path:
    """Write BUILD_INFO.json for the units that were just built.

    All three units are recorded, not only the ones just built: a partial
    rebuild must not drop the record of the others.

    BUILD_INFO.json holds nothing that varies from machine to machine, so
    that CI can rebuild and compare it byte for byte.  Where the build
    happened and with which tools goes in .build-state.json, which is
    local only.
    """
    roles = sorted(roles or ROLE_SOURCES)
    per_role: dict[str, object] = {}
    for role in roles:
        hashes = file_hashes(role)
        per_role[role] = {
            "roots": [p.relative_to(ROOT).as_posix() for p in ROLE_SOURCES[role][0]],
```

```
            "extras": [p.relative_to(ROOT).as_posix() for p in ROLE_SOURCES[role][1]],
            "files": hashes,
            "source_fingerprint": fingerprint(hashes),
            "images": image_hashes(role),
        }
    info: dict[str, object] = {
        "schema": 2,
        "rules": RULES,
        "roles": per_role,
    }
    BUILD_INFO.write_text(json.dumps(info, indent=2) + "\n", encoding="utf-8")

    state = {
        "git": git_state(),
        "tools": tools or {},
        "built_at": datetime.now().astimezone().isoformat(timespec="seconds"),
    }
    BUILD_STATE.write_text(json.dumps(state, indent=2) + "\n", encoding="utf-8")
    return BUILD_INFO


def read_build_info() -> dict:
    if not BUILD_INFO.is_file():
        return {}
    try:
        return json.loads(BUILD_INFO.read_text(encoding="utf-8"))
    except (OSError, ValueError):
        return {}
```

firmware/tools/hub_console.py

195 lines · 7,496 bytes

```
#!/usr/bin/env python3
"""Talk to the CryNet hub's serial console.

The hub listens for one-line commands at 115200 baud and prints its state
in reply:

    help          the list of commands
    status        the cots and what they are hearing
    alert <cot>   publish a test alert for that cot
    clear         take the test alert back
    history [n]   the last n records, newest first
    log           how much is recorded and how much is unsent
    log clear     throw the log away
    net           the network and what it has sent
    net set <url> where to post the records
    net ssid <s> <p>  which Wi-Fi to join
    net name <name>   this hub's name on the dashboard
    net token <t>     the dashboard's shared secret
    upload        send the unsent records now
    logtest       prove, on this chip, that out-of-order records are not lost

Two things about the board make this fiddly, and the tool handles both.

**The board may reboot when the port opens.**  The USB-serial bridge holds
its enable line, so opening the port can reset the hub -- and a command
written into a board that is still booting is a command thrown away,
which looks exactly like a command that was ignored.  So the tool waits
for the console's own banner ("the console is listening") before it
types anything.

**But the banner only appears on a boot.**  If the hub is already
running, it printed the banner before the port was opened and the tool
will never see it.  In that case the tool waits for the board to show
signs of life instead, and then it knows a command landed by the echo
the console prints for every command it receives ("console: << net").
If a command is not echoed, it is sent again -- which is what makes this
safe on a board that was still booting: a command typed too early is
simply echoed late, and only one of them is ever run.

Usage:
    python firmware/tools/hub_console.py --port COM32 --send status
    python firmware/tools/hub_console.py --port COM32 --send log --send history 5
    python firmware/tools/hub_console.py --port COM32 --send "net set https://x/api/v1/events"
    python firmware/tools/hub_console.py --list
"""

from __future__ import annotations

import argparse
```

```
import sys
import time

try:
    import serial
    import serial.tools.list_ports as list_ports
except ImportError:  # pragma: no cover
    print("pyserial is not installed: pip install pyserial", file=sys.stderr)
    sys.exit(1)


# The console says this when it is ready for commands.
READY = "the console is listening"

# The console echoes every command it receives.
ECHO = "console: <<"


def list_serial_ports() -> int:
    ports = list(list_ports.comports())
    if not ports:
        print("no serial ports found")
        return 0
    for p in ports:
        print(f"{p.device:<10} {p.description}")
    return 0


def read_for(port, seconds: float, sink=None):
    """Read the port for a while, returning (text, saw_ready)."""
    end = time.time() + seconds
    text = ""
    ready = False
    while time.time() < end:
        line = port.readline()
        if not line:
            continue
        chunk = line.decode("utf-8", "replace")
        text += chunk
        if sink is not None:
            sink.write(chunk)
            sink.flush()
        if READY in chunk:
            ready = True
    return text, ready


def send_and_wait_echo(port, command: str, settle: float) -> bool:
    """Send a command and wait for the console to echo it.

    Returns True when the echo was seen.  The command is sent again if
    it was not: on a board that is still booting, the first attempt is
    swallowed and the second lands.
    """
    for attempt in range(3):
        port.write((command + "\n").encode("ascii", "replace"))
        port.flush()
        end = time.time() + settle
        while time.time() < end:
            line = port.readline()
            if not line:
                continue
            text = line.decode("utf-8", "replace")
            sys.stdout.write(text)
            sys.stdout.flush()
            if ECHO in text and command in text:
                return True
    return False


def main() -> int:
    ap = argparse.ArgumentParser(description=__doc__,
                                 formatter_class=argparse.RawDescriptionHelpFormatter)
    ap.add_argument("--port", default="COM32", help="the hub's serial port")
    ap.add_argument("--baud", type=int, default=115200)
    ap.add_argument("--send", action="append", default=[],
                    help="a console command to send (repeatable)")
    ap.add_argument("--settle", type=float, default=2.5,
                    help="seconds to listen after each command")
    ap.add_argument("--wait", type=float, default=20.0,
                    help="seconds to wait for the console to come up")
    ap.add_argument("--list", action="store_true", help="list the serial ports")
    args = ap.parse_args()

    if args.list:
```

```
        return list_serial_ports()

    try:
        port = serial.Serial(args.port, args.baud, timeout=0.2)
    except serial.SerialException as exc:
        print(f"cannot open {args.port}: {exc}", file=sys.stderr)
        return 1

    with port:
        # The board may be booting now.  Wait for the console to say it
        # is ready, so that nothing is typed into a board that is not
        # listening yet.
        deadline = time.time() + args.wait
        ready = False
        alive = False
        while time.time() < deadline:
            line = port.readline()
            if not line:
                continue
            text = line.decode("utf-8", "replace")
            sys.stdout.write(text)
            sys.stdout.flush()
            if READY in text:
                ready = True
                break
            alive = True   # the board is talking, so it is past the boot

        if not ready:
            if not alive:
                print(f"\n[hub-console] the console did not come up on "
                      f"{args.port} within {args.wait:.0f}s", file=sys.stderr)
                return 1
            # The board was already running: its banner was printed
            # before the port opened.  It is alive, so it is listening.
            print("\n[hub-console] the board was already running; "
                  "sending the commands now", file=sys.stderr)

        for command in args.send:
            time.sleep(0.3)
            print(f"> {command}")
            if not send_and_wait_echo(port, command, args.settle):
                print(f"\n[hub-console] {args.port} did not echo "
                      f"{command!r}; it may not have taken it", file=sys.stderr)
                return 1
            # Listen while the board answers, but not for ever: a busy
            # board prints something every second or two, so "until it
            # goes quiet" would never end.
            end = time.time() + 1.0
            hard_end = time.time() + args.settle + 5.0
            while time.time() < end and time.time() < hard_end:
                line = port.readline()
                if line:
                    sys.stdout.write(line.decode("utf-8", "replace"))
                    sys.stdout.flush()
                    end = time.time() + 1.0   # more is coming

    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/gen_probe_refs.py

212 lines · 8,171 bytes

```
#!/usr/bin/env python3
"""Generate the on-device verification references for the CryNet front end.

Runs the *notebook's own* log-Mel front end (extracted from
notebooks/crynet_training.ipynb) over four closed-form probes, quantises
the result exactly as the firmware does, and writes

    firmware/hub_infer/components/crynet_dsp/include/crynet_probe_refs.h

The firmware then reproduces the same probes on the ESP32-S3 and compares
its quantised features against these. This is the on-device half of the
host parity test in research/test_c_parity.py.

The probe definitions here must stay in step with firmware/hub_infer/src/
probes.c — the generated header records the PCM CRC of each probe so a
mismatch is visible immediately.
```

```

Usage:  python firmware/tools/gen_probe_refs.py
"""

from __future__ import annotations

import json
import math
import re
import struct
import sys
import zlib
from pathlib import Path

import numpy as np

ROOT = Path(__file__).resolve().parents[2]
NB = ROOT / "notebooks" / "crynet_training.ipynb"
TABLES = ROOT / "research" / "out" / "crynet_tables.h"
DEST = ROOT / "firmware" / "hub_infer" / "components" / "crynet_dsp" / "include" / "crynet_probe_refs.h"

SR = 16000
N_SAMPLES = 32000
N_FRAMES = 81
N_MELS = 64

PROBE_NAMES = ["silence", "dc_bias", "tone", "noise"]


def load_front_end():
    """Pull the front-end cell out of the notebook and return its log_mel."""
    if not NB.exists():
        raise SystemExit(f"notebook not found: {NB}")

    nb = json.loads(NB.read_text(encoding="utf-8"))
    markers = {
        "config": "CELL 1 — Environment",
        "frontend": "CELL 4 — The audio front end",
    }
    srcs = {}
    for cell in nb["cells"]:
        if cell.get("cell_type") != "code":
            continue
        text = "\n".join(cell["source"])
        for key, marker in markers.items():
            if marker in text:
                srcs[key] = text
    missing = set(markers) - set(srcs)
    if missing:
        raise SystemExit(f"markers not found in {NB.name}: {missing}")

    import librosa  # noqa: E402  (the notebook imports it in its dependency cell)

    ns = {"__name__": "__main__", "librosa": librosa, "np": np}

    # The configuration cell resolves its directories from the environment,
    # so point it at a scratch directory before executing it.
    import os

    work = ROOT / "research" / "_probe"
    work.mkdir(parents=True, exist_ok=True)
    os.environ["CRYNET_OUT"] = str(work / "out")
    os.environ["CRYNET_WORK"] = str(work / "work")
    os.environ["CRYNET_CACHE"] = str(work / "cache")
    for sub in ("out", "work", "cache"):
        (work / sub).mkdir(parents=True, exist_ok=True)

    for key in ("config", "frontend"):
        exec(compile(srcs[key], f"<{key}>", "exec"), ns)

    if "log_mel" not in ns:
        raise SystemExit("the notebook's front end does not define log_mel()")
    return ns["log_mel"], ns


def quantisation():
    """Read the input scale / zero point from the generated tables."""
    text = TABLES.read_text(encoding="utf-8")
    if not TABLES.exists():
        raise SystemExit(f"missing {TABLES} — run the notebook and place its out/ here")
    scale = float(re.search(r"#define\s+CRYNET_IN_SCALE\s+([0-9.eE+-]+)", text).group(1))
    zp = int(re.search(r"#define\s+CRYNET_IN_ZP\s+(-?\d+)", text).group(1))
    return scale, zp


```

```
def clip(v):
    return max(-1.0, min(1.0, v))


def to_pcm(sig):
    """float in [-1, 1] -> int16, truncating toward zero (matches the C)."""
    pcm = (np.clip(sig, -1.0, 1.0) * 32767.0).astype(np.int16)
    return pcm


def probe_pcm(index):
    if index == 0:
        return np.zeros(N_SAMPLES, dtype=np.int16)
    if index == 1:
        return np.full(N_SAMPLES, 18000, dtype=np.int16)
    if index == 2:
        t = np.arange(N_SAMPLES) / SR
        x = 0.35 * np.sin(2.0 * math.pi * 420.0 * t) + 0.12 * np.sin(2.0 * math.pi * 1350.0 * t)
        return to_pcm(x)
    if index == 3:
        # 32-bit LCG, 24 high bits -> uniform [0, 1)
        state = np.uint32(12345)
        out = np.empty(N_SAMPLES, dtype=np.float64)
        for i in range(N_SAMPLES):
            state = np.uint32(1103515245) * state + np.uint32(12345)
            out[i] = (float((state >> np.uint32(8)) & np.uint32(0x00FFFFFF))) / 16777216.0
        return to_pcm(0.05 * (2.0 * out - 1.0))
    raise ValueError(index)


def main() -> int:
    log_mel, ns = load_front_end()
    in_scale, in_zp = quantisation()

    n_frames = int(ns.get("N_FRAMES", N_FRAMES))
    n_mels = int(ns.get("N_MELS", N_MELS))
    if (n_frames, n_mels) != (N_FRAMES, N_MELS):
        raise SystemExit(f"unexpected feature geometry {n_frames}x{n_mels}")

    lines = []
    lines.append("/*")
    lines.append(" * CryNet — expected probe features (GENERATED, do not edit).")
    lines.append(" *")
    lines.append(" * Produced by firmware/tools/gen_probe_refs.py from the training")
    lines.append(" * notebook's own log-Mel front end, quantised exactly as the")
    lines.append(" * firmware quantises. The ESP32-S3 reproduces the same probes and")
    lines.append(" * compares against these bytes, which is the on-device half of the")
    lines.append(" * host parity test.")
    lines.append(" *")
    lines.append(f" * input quantisation: scale {in_scale!r}, zero point {in_zp}")
    lines.append(" */")
    lines.append("")
    lines.append("#ifndef CRYNET_PROBE_REFS_H")
    lines.append("#define CRYNET_PROBE_REFS_H")
    lines.append("")
    lines.append("#include <stdint.h>")
    lines.append("")
    lines.append(f"#define CRYNET_PROBE_REF_COUNT      {len(PROBE_NAMES)}")
    lines.append(f"#define CRYNET_PROBE_REF_BYTES      {N_FRAMES * N_MELS}")
    lines.append("")

    crc_rows = []
    for idx, name in enumerate(PROBE_NAMES):
        pcm = probe_pcm(idx)
        pcm_crc = zlib.crc32(pcm.tobytes()) & 0xFFFFFFFF

        mel = np.asarray(log_mel(pcm.astype(np.float32) / 32768.0), dtype=np.float32)
        if mel.shape != (N_FRAMES, N_MELS):
            raise SystemExit(f"probe {name}: unexpected feature shape {mel.shape}")

        q = np.round(mel / in_scale) + in_zp
        q = np.clip(q, -128, 127).astype(np.int8).reshape(-1)
        feat_crc = zlib.crc32(q.tobytes()) & 0xFFFFFFFF
        crc_rows.append((name, pcm_crc, feat_crc, float(mel.min()), float(mel.max())))

        values = [str(int(v)) for v in q]
        lines.append(f"/* {name}: pcm_crc32 0x{pcm_crc:08x}, feature_crc32 0x{feat_crc:08x} */")
        lines.append(f"static const int8_t crynet_probe_ref_{name}[CRYNET_PROBE_REF_BYTES] = {{")
        for i in range(0, len(values), 64):
            lines.append("    " + ", ".join(values[i:i + 64]) + ",")
        lines.append("};")
        lines.append("")

    lines.append("static const int8_t *const crynet_probe_refs[CRYNET_PROBE_REF_COUNT] = {")
    for name in PROBE_NAMES:
```

```
        lines.append(f"    crynet_probe_ref_{name},")
    lines.append("};")
    lines.append("")
    lines.append("static const uint32_t crynet_probe_ref_pcm_crc[CRYNET_PROBE_REF_COUNT] = {")
    lines.append("    " + ", ".join(f"0x{crc:08x}" for _, crc, _, _, _ in crc_rows))
    lines.append("};")
    lines.append("")
    lines.append("static const uint32_t crynet_probe_ref_feature_crc[CRYNET_PROBE_REF_COUNT] = {")
    lines.append("    " + ", ".join(f"0x{crc:08x}" for _, _, crc, _, _ in crc_rows))
    lines.append("};")
    lines.append("")
    lines.append("#endif /* CRYNET_PROBE_REFS_H */")

    DEST.parent.mkdir(parents=True, exist_ok=True)
    DEST.write_text("\n".join(lines) + "\n", encoding="utf-8")

    print(f"wrote {DEST.relative_to(ROOT)}  ({DEST.stat().st_size} bytes)")
    print(f"  input quantisation: scale {in_scale}, zero point {in_zp}")
    for name, pcm_crc, feat_crc, lo, hi in crc_rows:
        print(f"  {name:8s} pcm_crc 0x{pcm_crc:08x}  feature_crc 0x{feat_crc:08x}  "
              f"mel [{lo:+.2f}, {hi:+.2f}]")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/listen_hub.py

124 lines · 3,898 bytes

```
#!/usr/bin/env python3
"""Listen to the hub and report what it does with a sound.

This script plays nothing.  It only reads the hub's log on COM32 and
prints the chain, so that a sound the user makes by hand - a cry from a
phone next to the node - can be watched as it travels:

  asking for the window   the trigger fired
  286/286 fragments in X  the burst arrived
  P(cry)=...              the model's answer
  ALERT - cry             the cry was confirmed

Usage:
    python firmware/tools/listen_hub.py --seconds 90
    python firmware/tools/listen_hub.py --seconds 90 --warmup 10
"""

from __future__ import annotations

import argparse
import os
import re
import sys
import threading
import time

LINE_RE = re.compile(r"^I \((\d+)\) (\w+): (.*)$")

# The lines that matter, in the order they matter.
MARKERS = (
    ("asking for the window", "ask"),
    ("fragments in", "burst"),
    ("window rms", "window"),
    ("front end", "model"),
    ("P(cry)=", "model"),
    ("ALERT", "verdict"),
    ("no cry", "verdict"),
    ("silent", "verdict"),
    ("cry (", "maybe"),
    ("stalled", "problem"),
    ("timed out", "problem"),
    ("missing", "problem"),
    ("bad crc", "problem"),
    ("resend", "problem"),
)


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--port", default="COM32")
    ap.add_argument("--seconds", type=float, default=90.0,
```

```
                    help="how long to listen for")
    ap.add_argument("--warmup", type=float, default=10.0,
                    help="seconds of quiet before the sound is expected")
    ap.add_argument("--reset", action="store_true",
                    help="reset the hub before listening")
    args = ap.parse_args()

    import serial

    print(f"=== listening on {args.port} for {args.seconds:.0f} s "
          f"(the sound is expected after {args.warmup:.0f} s)", flush=True)

    ser = serial.Serial(args.port, 115200, timeout=0.3)
    if args.reset:
        ser.dtr = False
        ser.rts = True
        time.sleep(0.15)
        ser.rts = False

    marks: list[tuple[str, int, str]] = []
    start = time.time()

    def reader():
        end = start + args.seconds
        while time.time() < end:
            try:
                raw = ser.readline()
            except Exception:
                return
            if not raw:
                continue
            line = raw.decode("utf-8", "replace").rstrip()
            m = LINE_RE.match(line)
            if not m:
                continue
            ts = int(m.group(1))
            body = m.group(3)
            for needle, kind in MARKERS:
                if needle in body:
                    marks.append((kind, ts, body))
                    print(f"  [{time.time() - start:6.2f}s] {body[:90]}", flush=True)
                    break

    t = threading.Thread(target=reader, daemon=True)
    t.start()

    if args.warmup > 0:
        print(f"--- quiet for {args.warmup:.0f} s: **play the cry now** ---",
              flush=True)
        time.sleep(args.warmup)
        print("--- the sound is expected from here ---", flush=True)

    t.join(timeout=args.seconds + 5)
    try:
        ser.close()
    except Exception:
        pass

    verdicts = [m for m in marks if m[0] == "verdict"]
    cries = [m for m in verdicts if "ALERT" in m[2] or "cry (held)" in m[2]]
    print(f"\n=== {len(marks)} marks, {len(verdicts)} verdicts, "
          f"{len(cries)} cry verdicts", flush=True)
    for kind, ts, body in verdicts:
        print(f"  {body[:100]}")
    sys.stdout.flush()
    # The reader thread is still blocked in a read when the interpreter
    # wants to shut down; that prints a traceback that has nothing to do
    # with the listening.
    os._exit(0)


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/listen_node.py

91 lines · 2,583 bytes

```
#!/usr/bin/env python3
"""Listen to the node and report what it hears.

The node owns the noise floor and the trigger level (D-63).  Its log is
```

```
the record of *why* it froze a window, so it has to be read alongside
the hub's log - otherwise a "no cry" verdict has nothing to explain it.

  rms 0.0063  floor 0.0028  level 0.0083     the room, and the level
  froze window, burst seq 15  level 0.0120  win rms 0.0100

Usage:
    python firmware/tools/listen_node.py --seconds 90
"""

from __future__ import annotations

import argparse
import os
import sys
import threading
import time

MARKERS = (
    ("froze window", "freeze"),
    ("rms ", "level"),
    ("request: seq", "request"),
    ("resend", "problem"),
    ("served ", "served"),
    ("FATAL", "problem"),
)


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--port", default="COM30")
    ap.add_argument("--seconds", type=float, default=90.0)
    ap.add_argument("--warmup", type=float, default=0.0)
    args = ap.parse_args()

    import serial

    print(f"=== node on {args.port} for {args.seconds:.0f} s", flush=True)
    ser = serial.Serial(args.port, 115200, timeout=0.3)
    time.sleep(0.2)
    ser.reset_input_buffer()

    marks: list[tuple[str, str]] = []
    start = time.time()

    def reader():
        end = start + args.seconds
        while time.time() < end:
            try:
                raw = ser.readline()
            except Exception:
                return
            if not raw:
                continue
            line = raw.decode("utf-8", "replace").rstrip()
            if not line:
                continue
            for needle, kind in MARKERS:
                if needle in line:
                    marks.append((kind, line))
                    print(f"  [{time.time() - start:6.2f}s] {line[:100]}", flush=True)
                    break

    t = threading.Thread(target=reader, daemon=True)
    t.start()

    if args.warmup > 0:
        print(f"--- quiet for {args.warmup:.0f} s ---", flush=True)
        time.sleep(args.warmup)

    t.join(timeout=args.seconds + 5)
    try:
        ser.close()
    except Exception:
        pass

    freezes = [m for m in marks if m[0] == "freeze"]
    print(f"\n=== {len(marks)} marks, {len(freezes)} frozen windows", flush=True)
    for kind, line in marks:
        if kind in ("freeze", "problem"):
            print(f"  {line[:110]}")
    sys.stdout.flush()
    os._exit(0)


```

```
if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/node_console.py

92 lines · 3,227 bytes

```
#!/usr/bin/env python3
"""Talk to the CryNet node's serial console.

The node firmware (firmware/node/node_crynet) listens for one-line
commands at 115200 baud and prints its state in reply:

    status   the window, the RMS, the mean and the frozen window's CRC-32
    stats    the counters (telemetry, frames sent and lost, requests)
    freeze   freeze the live window now, so it becomes the upload target
    burst    dump the frozen window in the radio's framing
    id <n>   set the node id
    help     the list of commands

`freeze` is the one that matters for testing the radio: it makes the node
hold a specific two-second window and advertise its CRC-32 in the
telemetry, so the hub can check that the audio it reassembles from ESP-NOW
is the audio the node froze.

Usage:
    python firmware/tools/node_console.py --port COM30 --send status
    python firmware/tools/node_console.py --port COM30 --send freeze --seconds 15
    python firmware/tools/node_console.py --list
"""

from __future__ import annotations

import argparse
import sys
import time

try:
    import serial
    import serial.tools.list_ports as list_ports
except ImportError:  # pragma: no cover
    print("pyserial is not installed: pip install pyserial", file=sys.stderr)
    sys.exit(1)


def list_serial_ports() -> int:
    ports = list(list_ports.comports())
    if not ports:
        print("no serial ports found")
        return 0
    for p in ports:
        print(f"{p.device:<10} {p.description}")
    return 0


def main() -> int:
    ap = argparse.ArgumentParser(description=__doc__,
                                 formatter_class=argparse.RawDescriptionHelpFormatter)
    ap.add_argument("--port", default="COM30", help="the node's serial port")
    ap.add_argument("--baud", type=int, default=115200)
    ap.add_argument("--send", action="append", default=[],
                    help="a console command to send (repeatable)")
    ap.add_argument("--seconds", type=float, default=10.0,
                    help="how long to keep reading")
    ap.add_argument("--list", action="store_true", help="list the serial ports")
    args = ap.parse_args()

    if args.list:
        return list_serial_ports()

    try:
        port = serial.Serial(args.port, args.baud, timeout=0.2)
    except serial.SerialException as exc:
        print(f"cannot open {args.port}: {exc}", file=sys.stderr)
        return 1

    # The node prints its banner when it boots, but it does not need a
    # reset: open the port, send whatever was asked for, and read.
    deadline = time.time() + args.seconds
    pending = list(args.send)

    with port:
```

```
        while time.time() < deadline:
            if pending and time.time() > deadline - args.seconds + 0.5:
                command = pending.pop(0)
                print(f"> {command}")
                port.write((command + "\n").encode("ascii", "replace"))
                port.flush()

            line = port.readline()
            if line:
                sys.stdout.write(line.decode("utf-8", "replace"))
                sys.stdout.flush()

    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/serial_monitor.py

87 lines · 2,954 bytes

```
#!/usr/bin/env python3
"""Watch what a board says over its USB serial port.

This is the CryNet equivalent of the "Serial Monitor" in the Arduino IDE,
so that you can check a board's boot log without having the IDE installed.

It resets the board over RTS/EN and then prints everything the board says,
at 115200 baud, until you press Ctrl+C.

Usage:
    python firmware/tools/serial_monitor.py --port COM31
    python firmware/tools/serial_monitor.py --port COM31 --no-reset
    python firmware/tools/serial_monitor.py --list
"""

from __future__ import annotations

import argparse
import sys
import time

try:
    import serial
    import serial.tools.list_ports as list_ports
except ImportError:  # pragma: no cover
    print("pyserial is not installed: pip install pyserial", file=sys.stderr)
    sys.exit(1)


def show_ports() -> int:
    ports = list(list_ports.comports())
    if not ports:
        print("no serial ports found")
        return 0
    for p in ports:
        print(f"{p.device:<8} {p.description}")
    return 0


def main() -> int:
    ap = argparse.ArgumentParser(
        description=__doc__, formatter_class=argparse.RawDescriptionHelpFormatter)
    ap.add_argument("--port", help="the port to watch, e.g. COM31")
    ap.add_argument("--baud", type=int, default=115200)
    ap.add_argument("--seconds", type=float, default=0.0,
                    help="stop after this long (0 = until Ctrl+C)")
    ap.add_argument("--no-reset", action="store_true",
                    help="do not reset the board first")
    ap.add_argument("--list", action="store_true", help="list the ports and exit")
    args = ap.parse_args()

    if args.list:
        return show_ports()
    if not args.port:
        ap.error("--port is required (use --list to see them)")

    try:
        ser = serial.Serial(args.port, args.baud, timeout=0.2)
    except Exception as exc:  # noqa: BLE001
        print(f"cannot open {args.port}: {exc}", file=sys.stderr)
```

```
        return 1

    print(f"watching {args.port} at {args.baud} baud -- Ctrl+C to stop")
    deadline = time.time() + args.seconds if args.seconds else None
    with ser:
        if not args.no_reset:
            time.sleep(0.3)
            ser.reset_input_buffer()
            # On an ESP32 the RTS line drives EN, so pulsing it resets the
            # board; DTR drives GPIO0 and is left alone.
            ser.dtr = False
            ser.rts = True
            time.sleep(0.15)
            ser.rts = False
        try:
            while deadline is None or time.time() < deadline:
                chunk = ser.read(4096)
                if chunk:
                    sys.stdout.write(chunk.decode("utf-8", errors="replace"))
                    sys.stdout.flush()
        except KeyboardInterrupt:
            print("\n--- stopped ---")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/capture_serial.py

104 lines · 3,478 bytes

```
#!/usr/bin/env python3
"""Read the CryNet hub's serial log and optionally reset the board first.

The Stage 2a firmware prints its verification report over UART at 115200
baud: the platform summary, the four probe comparisons against the host
reference, P(cry), the front-end and inference latency, and the tensor
arena usage. This script opens the port, optionally pulses EN to restart
the board so the whole boot sequence is captured, and prints everything
it receives.

Usage:
    python firmware/tools/capture_serial.py --port COM32
    python firmware/tools/capture_serial.py --port COM32 --reset --seconds 30
    python firmware/tools/capture_serial.py --list
"""

from __future__ import annotations

import argparse
import sys
import time

try:
    import serial
    import serial.tools.list_ports as list_ports
except ImportError:  # pragma: no cover
    print("pyserial is not installed: pip install pyserial", file=sys.stderr)
    sys.exit(1)


def list_serial_ports() -> int:
    ports = list(list_ports.comports())
    if not ports:
        print("no serial ports found")
        return 0
    for p in ports:
        print(f"{p.device:<10} {p.description}")
    return 0


def pulse_reset(ser: serial.Serial) -> None:
    """Reset the board into the application.

    On the ESP32-S3 DevKitC the auto-reset circuit drives EN from RTS and
    GPIO0 from DTR, both through inverting transistors: RTS high pulls EN
    low (reset), and DTR low leaves GPIO0 high (boot the application, not
    the ROM download mode).
    """
    ser.dtr = False       # GPIO0 high — do not enter the bootloader
    ser.rts = True        # EN low — reset
```

```
    time.sleep(0.15)
    ser.rts = False       # EN high — release the reset
    time.sleep(0.05)


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--port", help="serial port, e.g. COM32")
    ap.add_argument("--baud", type=int, default=115200)
    ap.add_argument("--seconds", type=float, default=20.0)
    ap.add_argument("--reset", action="store_true",
                    help="pulse EN first so the boot log is captured")
    ap.add_argument("--list", action="store_true", help="list serial ports and exit")
    args = ap.parse_args()

    if args.list:
        return list_serial_ports()

    if not args.port:
        print("no --port given; available ports:", file=sys.stderr)
        list_serial_ports()
        return 2

    try:
        ser = serial.Serial(args.port, args.baud, timeout=0.2)
    except Exception as exc:  # noqa: BLE001
        print(f"cannot open {args.port}: {exc}", file=sys.stderr)
        return 1

    with ser:
        if args.reset:
            pulse_reset(ser)
        else:
            # Opening the port with DTR/RTS asserted would hold the chip in
            # reset; make sure both lines are released.
            ser.dtr = False
            ser.rts = False

        deadline = time.time() + args.seconds
        received = 0
        while time.time() < deadline:
            chunk = ser.read(4096)
            if chunk:
                received += len(chunk)
                sys.stdout.write(chunk.decode("utf-8", errors="replace"))
                sys.stdout.flush()

    print(f"\n--- captured {received} bytes from {args.port} "
          f"at {args.baud} baud ---")
    return 0 if received else 1


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/capture_multi.py

240 lines · 8,913 bytes

```
#!/usr/bin/env python3
"""Watch several boards at once.

The bench has the hub and all three cot nodes on one machine now, so the
questions that matter are the ones that need every board at the same
time: does the hub hear all three cots, does one board's transmit
starve the others, does the wearable stay connected.  The tools that
existed could only hold one port open at a time, which cannot answer
any of those.

This opens every port at once -- one thread each -- and writes every
line it sees to its own file, stamped with the time since the capture
started.  A command can be sent to a port after the boards have settled
(`--send-after`), which is how the node counters are read.

Opening a port resets an ESP32-C3 (its USB Serial/JTAG does the reset
on the DTR/RTS edge), so every board reboots when the capture starts.
That is deliberate here: the capture wants to see the boot banners, and
the hub's radio comes back up by itself.

Usage:
    python firmware/tools/capture_multi.py --ports COM31 COM32 COM37 COM38
    python firmware/tools/capture_multi.py --ports COM31 COM37 COM38 \\
```

```
        --labels node1 node2 node3 --send-after 10 COM31:stats COM37:stats
"""

from __future__ import annotations

import argparse
import os
import re
import sys
import threading
import time
from datetime import datetime
from pathlib import Path

try:
    import serial
except ImportError:  # pragma: no cover
    print("pyserial is not installed: pip install pyserial", file=sys.stderr)
    sys.exit(1)

ROOT = Path(__file__).resolve().parents[2]

# The lines worth knowing about without reading the whole file.
INTERESTING = re.compile(
    r"ALERT|alert|Brownout|brownout|panic|abort|rst:|Guru|watchdog|"
    r"cry|cot |node |hub |win\(|P\(cry\)|decoded|BLE|ble\]|state:",
    re.IGNORECASE,
)


class Port:
    """One board on one port."""

    def __init__(self, port: str, label: str, out_dir: Path):
        self.port = port
        self.label = label
        self.path = out_dir / f"{label}_{port}.log"
        self.fh = None
        self.serial = None
        self.lines = 0
        self.interesting: list[str] = []
        self.lock = threading.Lock()
        self.error: str | None = None

    def open(self) -> None:
        # Build the port by hand so that DTR and RTS are never asserted:
        # on the S3 hub those lines reach EN and GPIO0, and a stray
        # assertion would put the board in the bootloader.
        s = serial.Serial()
        s.port = self.port
        s.baudrate = 115200
        s.timeout = 0.2
        s.dtr = False
        s.rts = False
        s.open()
        try:
            s.dtr = False
            s.rts = False
        except Exception:  # noqa: BLE001 -- not every driver has them
            pass
        self.serial = s
        if self.fh is None:
            # A fresh file for the first open; a reopen after a board
            # reset appends, so the earlier part of the run survives.
            self.fh = self.path.open("w", encoding="utf-8", errors="replace")

    def write(self, line: str, t0: float) -> None:
        stamp = f"{time.time() - t0:8.2f}"
        with self.lock:
            self.lines += 1
            if self.fh is None:
                self.fh = self.path.open("w", encoding="utf-8", errors="replace")
            self.fh.write(f"[{stamp}] {line}")
            self.fh.flush()
            if INTERESTING.search(line):
                self.interesting.append(line.rstrip())

    def close(self) -> None:
        if self.serial is not None:
            try:
                self.serial.close()
            except Exception:  # noqa: BLE001
                pass
        if self.fh is not None:
            self.fh.close()
```

```

def reader(p: Port, t0: float, stop: threading.Event) -> None:
    """Read one port until the capture ends."""
    try:
        p.open()
    except Exception as exc:  # noqa: BLE001
        p.error = str(exc)
        return
    buf = bytearray()
    try:
        while not stop.is_set():
            try:
                chunk = p.serial.read(p.serial.in_waiting or 1)
            except Exception as exc:  # noqa: BLE001
                # The C3's own USB re-enumerates when the board resets,
                # and ClearCommError then fails on every read.  The
                # wearable is the one that resets (its console is the
                # same port the flasher uses), so the capture reopens
                # the port and carries on rather than losing the rest
                # of the run's worth of alerts.
                try:
                    p.serial.close()
                except Exception:  # noqa: BLE001
                    pass
                for _ in range(20):
                    if stop.is_set():
                        break
                    time.sleep(0.5)
                    try:
                        p.open()
                    except Exception:  # noqa: BLE001
                        continue
                    p.write(f"<< port {p.label} reopened after: {exc} >>\n", t0)
                    buf = bytearray()
                    break
                else:
                    p.error = str(exc)
                    break
                continue
            if not chunk:
                continue
            buf += chunk
            while b"\n" in buf:
                raw, _, rest = bytes(buf).partition(b"\n")
                buf = bytearray(rest)
                p.write(raw.decode("utf-8", "replace").rstrip() + "\n", t0)
    finally:
        p.close()


def send(p: Port, command: str, t0: float) -> None:
    """Send one console command to a port."""
    if p.serial is None:
        # The port never opened -- usually another capture already has
        # it, or the board is gone.  Say so instead of crashing.
        print(f"port {p.label} ({p.port}) is not open: {p.error}",
              file=sys.stderr)
        return
    try:
        p.serial.write((command + "\n").encode("ascii", "replace"))
        p.serial.flush()
        p.write(f"<< sent: {command} >>\n", t0)
    except Exception as exc:  # noqa: BLE001
        p.write(f"<< could not send {command}: {exc} >>\n", t0)


def main() -> int:
    ap = argparse.ArgumentParser(description=__doc__,
                                 formatter_class=argparse.RawDescriptionHelpFormatter)
    ap.add_argument("--ports", nargs="+", required=True,
                    help="the serial ports to watch")
    ap.add_argument("--labels", nargs="+", default=None,
                    help="a name for each port (default: the port name)")
    ap.add_argument("--seconds", type=float, default=120.0,
                    help="how long to watch")
    ap.add_argument("--outdir", default=None,
                    help="where to put the logs (default: research/out/capture_*)")
    ap.add_argument("--send-after", type=float, default=0.0,
                    help="seconds to wait before sending the commands")
    ap.add_argument("--send", nargs="*", default=[],
                    help="PORT:command pairs, sent after --send-after seconds")
    args = ap.parse_args()

    labels = args.labels or [p.replace("COM", "port") for p in args.ports]
```

```
    if len(labels) != len(args.ports):
        print("--labels must have one name per port", file=sys.stderr)
        return 1

    stamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    out_dir = Path(args.outdir) if args.outdir else \
        ROOT / "research" / "out" / f"capture_{stamp}"
    out_dir.mkdir(parents=True, exist_ok=True)

    ports = [Port(p, l, out_dir) for p, l in zip(args.ports, labels)]
    stop = threading.Event()
    t0 = time.time()
    threads = []
    for p in ports:
        th = threading.Thread(target=reader, args=(p, t0, stop), daemon=True)
        th.start()
        threads.append(th)

    # let the boards boot before anything is said to them
    pending = list(args.send)
    if pending:
        time.sleep(max(0.0, args.send_after))
        for item in pending:
            port, _, command = item.partition(":")
            if not command:
                command = port
                port = None
            for p in ports:
                if port is None or p.port == port or p.label == port:
                    send(p, command, t0)

    time.sleep(max(0.0, args.seconds - (time.time() - t0)))
    stop.set()
    for th in threads:
        th.join(timeout=3.0)

    print(f"logs in {out_dir}")
    for p in ports:
        note = f"  ({p.error})" if p.error else ""
        print(f"  {p.label:<10} {p.port:<8} {p.lines:>6} lines{note}")
        for line in p.interesting[-12:]:
            print(f"      {line}")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

firmware/tools/capture_hub_wait.py

45 lines · 1,358 bytes

```
"""Wait for the hub to be plugged in, then raw-log its serial for a window.

Writes research/out/hub_node1_fullchain.log with [t] timestamped lines.
Exits 0 when the capture finished, 1 when the wait timed out.
"""
import sys, time
sys.path.insert(0, r"C:\Users\HP\.t3\projects\fp-2\firmware\tools")
import serial
from serial.tools import list_ports

OUT = r"C:\Users\HP\.t3\projects\fp-2\research\out\hub_node1_fullchain.log"
CAPTURE_S = 420
WAIT_LIMIT_S = 900


def hub_port():
    for p in list_ports.comports():
        if (p.vid == 0x1A86) or p.device == "COM32":  # CH343, the S3 hub
            return p.device
    return None


t0 = time.time()
port = None
while time.time() - t0 < WAIT_LIMIT_S:
    port = hub_port()
    if port:
        break
    time.sleep(2)

```

```
if not port:
    print("TIMEOUT: hub never appeared")
    sys.exit(1)

print(f"hub found on {port}, capturing {CAPTURE_S}s")
with open(OUT, "w", encoding="utf-8", errors="replace") as f, \
     serial.Serial(port, 115200, timeout=1) as s:
    s.setDTR(False); s.setRTS(False)
    start = time.time()
    while time.time() - start < CAPTURE_S:
        line = s.readline().decode("utf-8", "replace").rstrip()
        if line:
            f.write(f"[{time.time() - start:7.2f}s] {line}\n")
            f.flush()
print("capture complete:", OUT)
```

firmware/tools/dump_node_window.py

246 lines · 8,751 bytes

```
#!/usr/bin/env python3
"""Dump the node's frozen window over its serial console and look at it.

The node's `burst` console command writes the frozen two-second window as
fragment frames in the same wire format the radio uses, so this reads them
back, reassembles the audio, writes it as a WAV, and says what is in it:
the level, the largest peaks, and how the energy sits across the spectrum.

This is the instrument for the question the hub's log cannot answer: the
hub says *that* a window was judged a cry, not *what was in it*.

Usage:
    python firmware/tools/dump_node_window.py --out research/out/node_window.wav
    python firmware/tools/dump_node_window.py --port COM30 --score
"""

from __future__ import annotations

import argparse
import os
import struct
import sys
import time

import numpy as np
import serial

MAGIC = b"\xC3\x7E"
HEADER_BYTES = 14
FRAME_TYPE_FRAGMENT = 2
SAMPLE_RATE = 16000
WINDOW_SAMPLES = 32000
WINDOW_BYTES = WINDOW_SAMPLES * 2


def crc16_ccitt(data: bytes) -> int:
    """The same CRC the firmware uses: CCITT-FALSE, poly 0x1021, init 0xFFFF."""
    crc = 0xFFFF
    for b in data:
        crc ^= b << 8
        for _ in range(8):
            crc = ((crc << 1) ^ 0x1021) & 0xFFFF if crc & 0x8000 else (crc << 1) & 0xFFFF
    return crc


def read_until(port: serial.Serial, marker: bytes, timeout: float) -> bytes:
    """Drain the port as fast as it arrives.

    The dump is 68 kB at 115200 baud — about six seconds — and the
    driver's own buffer is 4 kB, so a reader that waits for a full block
    loses the frames that arrive in between.  Take whatever is there.
    """
    buf = bytearray()
    end = time.time() + timeout
    while time.time() < end:
        waiting = port.in_waiting
        if waiting:
            buf += port.read(waiting)
            if marker in buf:
                return bytes(buf)
        else:
            time.sleep(0.002)
```

```
    return bytes(buf)


def parse_frames(buf: bytes, start: int) -> list[tuple[int, bytes]]:
    """Every fragment frame in the buffer, in order."""
    frames = []
    i = start
    while i + HEADER_BYTES <= len(buf):
        if buf[i:i + 2] != MAGIC:
            i += 1
            continue
        (_type, _node, seq, frag, count, plen) = struct.unpack_from("<BBHHHH", buf, i + 2)
        total = HEADER_BYTES + plen + 2
        if i + total > len(buf):
            break
        payload = buf[i + HEADER_BYTES:i + HEADER_BYTES + plen]
        fcrc = struct.unpack_from("<H", buf, i + HEADER_BYTES + plen)[0]
        if crc16_ccitt(buf[i:i + HEADER_BYTES + plen]) != fcrc:
            print(f"frame {frag}: the frame CRC failed", file=sys.stderr)
            i += total
            continue
        if _type == FRAME_TYPE_FRAGMENT:
            frames.append((frag, payload))
        i += total
    return frames


def spectrum_report(x: np.ndarray) -> None:
    """Where the energy is, and whether it is periodic."""
    win = np.hanning(len(x))
    spec = np.abs(np.fft.rfft(x * win))
    freqs = np.fft.rfftfreq(len(x), 1.0 / SAMPLE_RATE)
    total = spec.sum() or 1.0

    # The energy in the bands a cry lives in, against the rest.
    bands = [(0, 300), (300, 1000), (1000, 2500), (2500, 4000), (4000, 8000)]
    print("  band energy:")
    for lo, hi in bands:
        m = (freqs >= lo) & (freqs < hi)
        share = float(spec[m].sum() / total)
        bar = "#" * int(round(share * 40))
        print(f"    {lo:>5}-{hi:<5} Hz  {share * 100:5.1f}%  {bar}")

    # The strongest partials: a voiced cry has a clear harmonic series.
    top = np.argsort(spec)[::-1][:400]
    peaks = []
    for idx in top:
        f = float(freqs[idx])
        if any(abs(f - p) < 30 for p in peaks):
            continue
        peaks.append(f)
        if len(peaks) >= 6:
            break
    peaks.sort()
    if peaks:
        print("  strongest partials: " + ", ".join(f"{p:.0f} Hz" for p in peaks))
        if len(peaks) >= 2:
            gaps = [b - a for a, b in zip(peaks, peaks[1:]) if b - a > 0]
            if gaps:
                print(f"  spacing between them: {['%.0f' % g for g in gaps]} Hz")


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--port", default="COM30")
    ap.add_argument("--out", default="research/out/node_window.wav")
    ap.add_argument("--timeout", type=float, default=20.0)
    ap.add_argument("--score", action="store_true",
                    help="also score the window with the deployed INT8 model")
    args = ap.parse_args()

    port = serial.Serial(args.port, 115200, timeout=0.2)
    try:
        port.reset_input_buffer()
        port.write(b"burst\n")
        port.flush()
        buf = read_until(port, b"BURST-END", args.timeout)
    finally:
        port.close()

    marker = buf.find(b"BURST seq=")
    if marker < 0:
        print("the node did not send a burst", file=sys.stderr)
        print(buf[-400:].decode("utf-8", "replace"), file=sys.stderr)
        return 1
```

```
148
149     # The sequence number is the node's own name for the window, and it is
150     # the number the hub's log uses, so a dump and a verdict can be
151     # matched to each other.
152     head = buf[marker:marker + 40].decode("utf-8", "replace")
153     seq = None
154     try:
155         seq = int(head.split("seq=")[1].split()[0])
156     except (IndexError, ValueError):
157         pass
158
159     frames = parse_frames(buf, marker)
160     if not frames:
161         print("no fragment frames came back", file=sys.stderr)
162         return 1
163
164     # The USB driver drops the odd frame from a 68 kB burst.  A fragment
165     # is 224 bytes of 64000, so one lost fragment is a hole in the audio
166     # and not a reason to throw the window away: fill the hole with
167     # silence and say which one it was.
168     by_index = {frag: payload for frag, payload in frames}
169     count = max(by_index) + 1
170     missing = [i for i in range(count) if i not in by_index]
171     data = bytearray()
172     for i in range(count):
173         data += by_index.get(i, b"\x00" * 224)
174     if missing:
175         print(f"missing fragment(s): {missing} — filled with silence")
176
177     if len(data) < WINDOW_BYTES:
178         print(f"only {len(data)} of {WINDOW_BYTES} bytes came back "
179               f"({len(frames)} fragments)", file=sys.stderr)
180         return 1
181
182     samples = np.frombuffer(bytes(data[:WINDOW_BYTES]), dtype="<i2").astype(np.float32) / 32768.0
183     mean = float(samples.mean())
184     centred = samples - mean
185     rms = float(np.sqrt((centred ** 2).mean()))
186     peak = float(np.abs(samples).max())
187     print(f"fragments: {len(frames)}  bytes: {len(data)}  seq: {seq}")
188     print(f"mean {mean:+.6f}  rms {rms:.6f}  peak {peak:.6f}  "
189           f"peak/rms {(peak / rms if rms else 0):.1f}")
190     spectrum_report(samples)
191
192     out = os.path.abspath(args.out)
193     os.makedirs(os.path.dirname(out), exist_ok=True)
194     wave_out(out, samples)
195     print(f"wrote {out}")
196
197     if args.score:
198         return score(out)
199     return 0
200
201
202 def wave_out(path: str, samples: np.ndarray):
203     """Write 16 kHz mono int16 WAV."""
204     import wave
205
206     pcm = np.clip(samples * 32768.0, -32768, 32767).astype("<i2")
207     f = wave.open(path, "wb")
208     f.setnchannels(1)
209     f.setsampwidth(2)
210     f.setframerate(SAMPLE_RATE)
211     f.writeframes(pcm.tobytes())
212     f.close()
213     return path
214
215
216 def score(path: str) -> int:
217     """Score the window with the deployed INT8 model, the same way the hub does."""
218     sys.path.insert(0, os.path.join(os.path.dirname(__file__), "..", "..", "research"))
219     try:
220         from eval_indomain import notebook_namespace, read_wav  # type: ignore
221     except Exception as exc:  # pragma: no cover
222         print(f"could not load the notebook helpers: {exc}", file=sys.stderr)
223         return 1
224
225     import tensorflow as tf
226
227     ns = notebook_namespace()
228     x = read_wav(path)
229     f = ns["log_mel"](x)[None, ...].astype(np.float32)
230     out_dir = os.path.join(os.path.dirname(__file__), "..", "..", "research", "out")
231     interp = tf.lite.Interpreter(model_path=os.path.join(out_dir, "crynet_int8.tflite"))
232     interp.allocate_tensors()
```

```
    i_d = interp.get_input_details()[0]
    o_d = interp.get_output_details()[0]
    s, z = i_d["quantization"]
    q = np.clip(np.round(f / s) + z, -128, 127).astype(np.int8)
    interp.set_tensor(i_d["index"], q[..., None])
    interp.invoke()
    o = interp.get_tensor(o_d["index"])[0].astype(np.float32)
    p = (o - o_d["quantization"][1]) * o_d["quantization"][0]
    print(f"model: P(no_cry)={p[0]:.4f}  P(cry)={p[1]:.4f}")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/collect_clips.py

271 lines · 10,116 bytes

```
#!/usr/bin/env python3
"""Collect in-domain audio from the CryNet node over the USB serial link.

The deployed model was trained on corpora that this room's microphone has
never produced.  This tool runs the `node_capture` sketch, writes the
stream to a continuous WAV, records the per-record RMS so the loud parts
can be found, and cuts the session into fixed-length clips that match the
training format (16 kHz, mono, 16-bit).

Usage:
    python firmware/tools/collect_clips.py --out research/in_domain/session1
    python firmware/tools/collect_clips.py --port COM30 --seconds 60 \
        --out research/in_domain/session1 --note "cot, baby crying"
    python firmware/tools/collect_clips.py --list

The node prints `CRY-CAPTURE-LIVE` when it is ready, but the tool does not
depend on seeing it: a USB CDC port can enumerate after the sketch has
already printed its banner, and every record carries its own magic, so the
stream resynchronises on its own.
"""

from __future__ import annotations

import argparse
import csv
import struct
import sys
import time
from pathlib import Path

MAGIC = b"CRY1"
HEADER = struct.Struct("<4sHHIH")
SAMPLE_RATE = 16000


def crc16(data: bytes) -> int:
    crc = 0xFFFF
    for byte in data:
        crc ^= byte << 8
        for _ in range(8):
            crc = ((crc << 1) ^ 0x1021) & 0xFFFF if crc & 0x8000 else (crc << 1) & 0xFFFF
    return crc

ESP32_VID_PIDS = {
    (0x1A86, 0x55D3): "CH343",
    (0x1A86, 0x7523): "CH340",
    (0x10C4, 0xEA60): "CP210x",
    (0x0403, 0x6001): "FTDI",
    (0x303A, 0x1001): "Espressif native USB (the C3 SuperMini's own USB)",
}


class RecordParser:
    """Pulls capture records out of the byte stream.

    The stream is the sketch's binary records, possibly preceded by its text
    banner.  Each record carries its own magic and its own length, so the
    parser resynchronises after the banner and after a lost byte.
    """

    def __init__(self) -> None:
        self.buf = bytearray()
        self.bad_crc = 0
```

```
    def feed(self, data: bytes) -> tuple[list[tuple[int, int, int, bytes]], str]:
        """Return the complete records and any text that was skipped."""
        self.buf += data
        records: list[tuple[int, int, int, bytes]] = []
        text = ""
        while True:
            start = self.buf.find(MAGIC)
            if start < 0:
                # Keep the last 3 bytes: a magic may straddle two reads.
                if len(self.buf) > 3:
                    text += self.buf[:-3].decode("utf-8", "replace")
                    del self.buf[:-3]
                break
            if start:
                text += self.buf[:start].decode("utf-8", "replace")
                del self.buf[:start]
            if len(self.buf) < HEADER.size:
                break
            header = bytes(self.buf[:HEADER.size])
            magic, seq, nsamples, rms_milli, pcrc = HEADER.unpack(header)
            if magic != MAGIC or nsamples == 0 or nsamples > 4096:
                del self.buf[:1]
                continue
            need = HEADER.size + nsamples * 2
            if len(self.buf) < need:
                break
            payload = bytes(self.buf[HEADER.size:need])
            del self.buf[:need]
            # The checksum covers the first 12 header bytes and the payload.
            if crc16(header[:12] + payload) != pcrc:
                self.bad_crc += 1
                continue
            records.append((seq, nsamples, rms_milli, payload))
        return records, text


def find_port() -> str | None:
    import serial.tools.list_ports as list_ports

    for p in list_ports.comports():
        if (p.vid, p.pid) in ESP32_VID_PIDS:
            print(f"  found {p.device}: {p.description} "
                  f"({ESP32_VID_PIDS[(p.vid, p.pid)]})")
            return p.device
    return None


def write_wav_header(fh, data_bytes: int) -> bytes:
    """A 44-byte canonical WAV header with the sizes left at zero."""
    hdr = b"RIFF" + struct.pack("<I", 0) + b"WAVE"
    hdr += b"fmt " + struct.pack("<IHHIIHH", 16, 1, 1, SAMPLE_RATE,
                                 SAMPLE_RATE * 2, 2, 16)
    hdr += b"data" + struct.pack("<I", 0)
    return hdr


def patch_wav_size(path: Path) -> None:
    data = path.stat().st_size - 44
    if data < 0:
        return
    with path.open("r+b") as fh:
        fh.seek(4)
        fh.write(struct.pack("<I", 36 + data))
        fh.seek(40)
        fh.write(struct.pack("<I", data))


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--port", help="serial port; auto-detected if omitted")
    ap.add_argument("--out", help="output directory")
    ap.add_argument("--seconds", type=float, default=0.0,
                    help="stop after this many seconds (0 = until Ctrl-C)")
    ap.add_argument("--clip-seconds", type=float, default=5.0,
                    help="length of the cut clips, in seconds")
    ap.add_argument("--note", default="", help="note stored with the session")
    ap.add_argument("--baud", type=int, default=460800)
    ap.add_argument("--list", action="store_true")
    args = ap.parse_args()

    try:
        import serial
    except ImportError:
        print("pyserial is not installed: pip install pyserial", file=sys.stderr)
```

```
        return 1

    if args.list:
        import serial.tools.list_ports as list_ports
        for p in list_ports.comports():
            print(f"{p.device:<10} {p.description}")
        return 0

    if not args.out:
        ap.error("--out is required (it is the directory the session is "
                 "written to)")

    port = args.port or find_port()
    if not port:
        print("No ESP32 serial port found.", file=sys.stderr)
        return 1

    out = Path(args.out)
    out.mkdir(parents=True, exist_ok=True)
    clips_dir = out / "clips"
    clips_dir.mkdir(exist_ok=True)

    wav_path = out / "session.wav"
    csv_path = out / "records.csv"
    meta_path = out / "session.txt"

    print(f"Listening on {port} at {args.baud} baud ...")
    ser = serial.Serial(port, args.baud, timeout=0.1)
    ser.dtr = False
    ser.rts = False

    parser = RecordParser()
    total_samples = 0
    total_bytes = 0
    clip_index = 0
    clip_buf = bytearray()
    clip_samples = 0
    clip_target = int(args.clip_seconds * SAMPLE_RATE)
    records = 0
    started = time.time()
    live = False

    with wav_path.open("wb") as wav, csv_path.open("w", newline="") as csvf:
        wav.write(write_wav_header(wav, 0))
        writer = csv.writer(csvf)
        writer.writerow(["index", "seconds", "rms", "rms_milli"])

        try:
            while True:
                # Drain everything the node has sent.  Waiting for a fixed
                # block size lets the device's small CDC transmit buffer
                # overflow, and the bytes are lost for good.
                waiting = ser.in_waiting
                chunk = ser.read(waiting if waiting else 1)
                if chunk:
                    new_records, text = parser.feed(chunk)
                    if "CRY-CAPTURE-LIVE" in text and not live:
                        live = True
                        print("node is live")
                    if "FATAL" in text and not live:
                        print("the node reported a fatal error:\n" + text,
                              file=sys.stderr)
                        return 1

                    for seq, nsamples, rms_milli, payload in new_records:
                        wav.write(payload)
                        total_bytes += len(payload)
                        total_samples += nsamples
                        records += 1
                        writer.writerow([seq, f"{total_samples / SAMPLE_RATE:.3f}",
                                         f"{rms_milli / 1000.0:.5f}", rms_milli])

                        clip_buf += payload
                        clip_samples += nsamples
                        while clip_samples >= clip_target:
                            take = clip_target * 2
                            chunk_bytes = bytes(clip_buf[:take])
                            del clip_buf[:take]
                            clip_samples -= clip_target
                            clip_path = clips_dir / f"clip_{clip_index:04d}.wav"
                            with clip_path.open("wb") as cf:
                                cf.write(write_wav_header(cf, 0))
                                cf.write(chunk_bytes)
                            patch_wav_size(clip_path)
                            clip_index += 1
```

```
                    if records and records % 50 == 0:
                        wav.flush()
                        print(f"  {total_samples / SAMPLE_RATE:7.1f} s "
                              f"({total_bytes / 1024:.0f} KiB, {clip_index} clips)")

                if args.seconds and (time.time() - started) > args.seconds:
                    break
        except KeyboardInterrupt:
            print("\nstopped")
        finally:
            ser.close()

    patch_wav_size(wav_path)
    meta_path.write_text(
        f"port: {port}\n"
        f"sample_rate: {SAMPLE_RATE}\n"
        f"channels: 1\n"
        f"bits: 16\n"
        f"seconds: {total_samples / SAMPLE_RATE:.3f}\n"
        f"records: {records}\n"
        f"clips: {clip_index} x {args.clip_seconds} s\n"
        f"note: {args.note}\n"
        f"recorded: {time.strftime('%Y-%m-%d %H:%M:%S')}\n"
    )

    print(f"\nwrote {wav_path} ({total_samples / SAMPLE_RATE:.1f} s)")
    print(f"wrote {csv_path} ({records} records)")
    print(f"wrote {clips_dir} ({clip_index} clips of {args.clip_seconds} s)")
    print(f"wrote {meta_path}")
    if parser.bad_crc:
        print(f"\n{parser.bad_crc} record(s) failed their CRC and were dropped. "
              f"The audio around them is still good.")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/label_session.py

263 lines · 10,828 bytes

```
#!/usr/bin/env python3
"""Turn a captured session into labelled 5-second clips.

`collect_clips.py` records a session and cuts it into fixed-length clips,
but a fixed cut is not a label: a clip that happens to straddle the start
of a cry is neither "cry" nor "not cry".  This tool cuts the session on a
fixed grid and labels each clip by *what the clip contains*, measured from
the per-100 ms RMS that the capture sketch reports (records.csv) — or, if
that file is missing or truncated, from the audio itself.

The rules are deliberately simple and auditable:

  * the quiet floor is the 10th percentile of the record RMS;
  * a record counts as "signal" when its RMS is at or above the threshold,
    which defaults to max(3x the quiet floor, --threshold);
  * a clip is "cry" when at least --cry-share of its records are signal,
    "no_cry" when at most --quiet-share are, and anything in between is
    "uncertain" — reported, but not used for training.

Usage:
    python firmware/tools/label_session.py --session research/in_domain/session1
    python firmware/tools/label_session.py --session research/in_domain/session1 \
        --threshold 0.02 --clip-seconds 5.0 --stride 2.5

Outputs, inside the session directory:
    clips_labeled/    the cut clips (16 kHz mono 16-bit, the training format)
    manifest.csv      one row per clip: file, label, start, end, rms stats
    summary.txt       the thresholds, the percentiles and a per-10-second
                      timeline of the session, so the levels can be checked
                      by eye without listening.
"""

from __future__ import annotations

import argparse
import csv
import math
import statistics
import struct
```

```
import wave
from pathlib import Path

import numpy as np

SAMPLE_RATE = 16000
RECORD_SECONDS = 0.1
PER_RECORD = int(RECORD_SECONDS * SAMPLE_RATE)   # samples in one record


def percentile(values: list[float], q: float) -> float:
    """The q-th percentile (0..1) of a list, by linear interpolation."""
    if not values:
        return 0.0
    s = sorted(values)
    if len(s) == 1:
        return s[0]
    pos = q * (len(s) - 1)
    lo = int(math.floor(pos))
    hi = min(lo + 1, len(s) - 1)
    frac = pos - lo
    return s[lo] * (1 - frac) + s[hi] * frac


def read_wav_samples(path: Path) -> tuple[np.ndarray, int]:
    """The samples of a 16-bit PCM WAV, parsed by hand.

    The collector patches the WAV header's sizes when it finishes, so a run
    that is interrupted leaves a header that claims zero bytes.  Parsing
    the chunks directly means a partial recording is still usable.
    """
    raw = path.read_bytes()
    if raw[:4] != b"RIFF" or raw[8:12] != b"WAVE":
        raise ValueError(f"{path} is not a WAV")
    pos = 12
    rate = SAMPLE_RATE
    data = None
    while pos + 8 <= len(raw):
        cid = raw[pos:pos + 4]
        size = struct.unpack("<I", raw[pos + 4:pos + 8])[0]
        body = raw[pos + 8:pos + 8 + size]
        if cid == b"fmt " and len(body) >= 8:
            rate = struct.unpack("<I", body[4:8])[0]
        elif cid == b"data":
            # A truncated file, or a header whose sizes were never
            # patched: keep the bytes that are actually there.
            remaining = len(raw) - (pos + 8)
            data = raw[pos + 8:] if (size == 0 or remaining < size) else body
            break
        pos += 8 + size + (size & 1)
    if data is None:
        raise ValueError(f"{path} has no data chunk")
    x = np.frombuffer(data[:len(data) // 2 * 2], dtype="<i2").astype(np.float32) / 32768.0
    return x, rate


def envelope(x: np.ndarray, block: int = PER_RECORD) -> list[float]:
    """The RMS of every 100 ms block, each block's own mean removed — the
    same quantity the capture sketch reports."""
    out: list[float] = []
    for i in range(0, len(x) - block + 1, block):
        seg = x[i:i + block]
        d = seg - seg.mean()
        out.append(float(np.sqrt(np.mean(d * d))))
    return out


def read_records(path: Path) -> list[float]:
    """The per-record RMS in the order the records were captured."""
    out: list[float] = []
    with path.open(newline="") as fh:
        for row in csv.DictReader(fh):
            out.append(float(row["rms"]))
    return out


def write_clip(samples: bytes, path: Path, rate: int = SAMPLE_RATE) -> None:
    with wave.open(str(path), "wb") as out:
        out.setnchannels(1)
        out.setsampwidth(2)
        out.setframerate(rate)
        out.writeframes(samples)


def main() -> int:
```

```
    ap = argparse.ArgumentParser()
    ap.add_argument("--session", required=True, help="the session directory")
    ap.add_argument("--threshold", type=float, default=0.012,
                    help="RMS above which a record counts as signal")
    ap.add_argument("--clip-seconds", type=float, default=5.0)
    ap.add_argument("--stride", type=float, default=2.5,
                    help="distance between the starts of two clips")
    ap.add_argument("--cry-share", type=float, default=0.25,
                    help="the share of loud records that makes a clip a cry")
    ap.add_argument("--quiet-share", type=float, default=0.05,
                    help="the share of loud records that makes a clip quiet")
    ap.add_argument("--name", default="", help="a label prefix, e.g. 'cry_near'")
    ap.add_argument("--truth", choices=("auto", "cry", "no_cry"), default="auto",
                    help="'auto' labels a clip by its loudness (the default); "
                         "'cry' or 'no_cry' labels every clip with the ground "
                         "truth of the session, for a session in which one "
                         "sound was made for the whole minute")
    args = ap.parse_args()

    session = Path(args.session)
    wav_path = session / "session.wav"
    csv_path = session / "records.csv"
    if not wav_path.is_file():
        print(f"{session} does not look like a session directory "
              f"(session.wav is required)", flush=True)
        return 1

    records = read_records(csv_path) if csv_path.is_file() else []
    x, rate = read_wav_samples(wav_path)
    if not records:
        print("records.csv is missing or empty — the RMS envelope is "
              "computed from the audio instead", flush=True)
        records = envelope(x)
    if not records:
        print("the session has no audio at all", flush=True)
        return 1

    floor = percentile(records, 0.10)
    threshold = max(args.threshold, floor * 3.0)

    clip_samples = int(args.clip_seconds * SAMPLE_RATE)
    stride_samples = int(args.stride * SAMPLE_RATE)

    out_dir = session / "clips_labeled"
    out_dir.mkdir(exist_ok=True)
    for old in out_dir.glob("*.wav"):
        old.unlink()

    manifest: list[list] = []
    index = 0
    start = 0
    while start + clip_samples <= len(x):
        first = start
        last = start + clip_samples
        rec_from = first // PER_RECORD
        rec_to = min(len(records), last // PER_RECORD)
        window = records[rec_from:rec_to]
        share = (sum(1 for r in window if r >= threshold) / len(window)
                 if window else 0.0)
        if args.truth != "auto":
            # The session's ground truth, given by hand.  A loudness rule
            # cannot label this session: in the music and movie takes the
            # loud clips are exactly the ones that are NOT a cry, so the
            # rule would call them cries and the confusion matrix would
            # measure nothing.  One sound for the whole minute has one
            # truth, and the script of the session knows it.
            label = args.truth
        elif share >= args.cry_share:
            label = "cry"
        elif share <= args.quiet_share:
            label = "no_cry"
        else:
            label = "uncertain"
        payload = np.clip(np.round(x[first:last] * 32768.0), -32768, 32767
                          ).astype("<i2").tobytes()
        name = f"{args.name + '_' if args.name else ''}clip_{index:04d}.wav"
        write_clip(payload, out_dir / name, rate)
        manifest.append([
            name, label,
            f"{first / SAMPLE_RATE:.1f}", f"{last / SAMPLE_RATE:.1f}",
            f"{statistics.fmean(window):.5f}" if window else "0",
            f"{max(window):.5f}" if window else "0",
            f"{share:.2f}",
        ])
        index += 1
```

```
        start += stride_samples

    with (session / "manifest.csv").open("w", newline="") as fh:
        writer = csv.writer(fh)
        writer.writerow(["file", "label", "start_s", "end_s",
                         "rms_mean", "rms_peak", "loud_share"])
        writer.writerows(manifest)

    # The summary: the numbers, and a coarse timeline so the levels can be
    # checked without listening to the audio.
    counts = {"cry": 0, "no_cry": 0, "uncertain": 0}
    for m in manifest:
        counts[m[1]] += 1
    total_s = len(records) * RECORD_SECONDS
    labels_line = ("ground truth of the session (given by hand)"
                   if args.truth != "auto"
                   else "by loudness (auto)")
    lines = [
        f"session: {session}",
        f"labels: {labels_line}",
        f"records: {len(records)}  ({total_s:.1f} s)",
        f"threshold: {threshold:.5f}  (floor p10 = {floor:.5f}, "
        f"max = {max(records):.5f})",
        f"percentiles: p10 {percentile(records, 0.10):.5f}  "
        f"p50 {percentile(records, 0.50):.5f}  "
        f"p90 {percentile(records, 0.90):.5f}  "
        f"p99 {percentile(records, 0.99):.5f}",
        f"clips: {len(manifest)}  "
        f"({counts['cry']} cry, {counts['no_cry']} no_cry, "
        f"{counts['uncertain']} uncertain)",
        f"clip length: {args.clip_seconds:.1f} s, stride {args.stride:.1f} s",
        "",
        "timeline (every 10 s: mean RMS, peak RMS, [S]ignal share)",
    ]
    per = int(10 / RECORD_SECONDS)
    for i in range(0, len(records), per):
        window = records[i:i + per]
        if not window:
            continue
        share = sum(1 for r in window if r >= threshold) / len(window)
        mark = "S" if share > 0.5 else ("s" if share > 0 else ".")
        lines.append(f"  {i * RECORD_SECONDS:6.0f}s  mean {statistics.fmean(window):.4f}"
                     f"  peak {max(window):.4f}  [{mark}]")
    (session / "summary.txt").write_text("\n".join(lines) + "\n")

    print("\n".join(lines[:7]))
    print(f"\nwrote {len(manifest)} clips to {out_dir}")
    print(f"wrote {session / 'manifest.csv'}")
    print(f"wrote {session / 'summary.txt'}")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

firmware/tools/measure_burst.py

140 lines · 4,793 bytes

```
#!/usr/bin/env python3
"""Measure the hub's burst time, on the real boards.

The window takes 5.7-7.6 s to arrive at the hub, and the panel shows a
cry about eight seconds after it happened.  The cause is the hub's own
Bluetooth advertising: the ESP32-S3 has one radio and it carries both
Bluetooth and Wi-Fi, so every advertisement is a moment when the Wi-Fi
receiver cannot hear the fragments of the burst.

This script measures the burst the way the system experiences it:

  * it reads the hub's log on COM32,
  * it plays the recorded cry through the laptop's speakers, near the
    node, so the node's telemetry crosses the trigger and the hub asks
    for the window,
  * it reports how long the burst took and how many fragments arrived.

Usage:
    python firmware/tools/measure_burst.py
    python firmware/tools/measure_burst.py --label "after the BLE change"
"""

from __future__ import annotations
```

```

import argparse
import re
import sys
import threading
import time
import wave
from pathlib import Path

ROOT = Path(__file__).resolve().parents[2]
CRY_WAV = ROOT / "research" / "in_domain" / "session9" / "session.wav"

# "node 1: 286/286 fragments in 5.71 s"
BURST_RE = re.compile(r"node (\d+): (\d+)/(\d+) fragments in ([0-9.]+) s")
# "node 1: missing fragments:" / "the hub asked for the window"
ASK_RE = re.compile(r"asking for the window \(seq (\d+)")
RESEND_RE = re.compile(r"resend|round", re.I)


def play_cry(path: Path, seconds: float) -> None:
    """Play the recording on the laptop's speakers."""
    import winsound

    if not path.is_file():
        print(f"no recording at {path}", file=sys.stderr)
        return
    print(f"playing {path.name} for {seconds:.0f} s on the laptop's speakers")
    winsound.PlaySound(str(path), winsound.SND_FILENAME | winsound.SND_ASYNC)
    time.sleep(seconds)
    winsound.PlaySound(None, winsound.SND_PURGE)


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--port", default="COM32")
    ap.add_argument("--seconds", type=float, default=45.0,
                    help="how long to listen for the burst")
    ap.add_argument("--warmup", type=float, default=8.0,
                    help="seconds of listening before the sound starts")
    ap.add_argument("--sound-seconds", type=float, default=14.0)
    ap.add_argument("--label", default="")
    args = ap.parse_args()

    import serial

    label = args.label or time.strftime("%H:%M:%S")
    print(f"=== burst measurement: {label}")

    ser = serial.Serial(args.port, 115200, timeout=0.3)
    # The hub prints its banner only at boot; reset it so the log is
    # complete and the radio starts from a known state.
    ser.dtr = False
    ser.rts = True
    time.sleep(0.15)
    ser.rts = False

    lines: list[str] = []

    def reader():
        end = time.time() + args.seconds
        while time.time() < end:
            try:
                raw = ser.readline()
            except Exception:
                break
            if raw:
                lines.append(raw.decode("utf-8", "replace").rstrip())

    t = threading.Thread(target=reader, daemon=True)
    t.start()

    time.sleep(args.warmup)
    player = threading.Thread(target=play_cry,
                              args=(CRY_WAV, args.sound_seconds), daemon=True)
    player.start()
    player.join()
    time.sleep(max(args.seconds - args.warmup - args.sound_seconds, 5.0))
    t.join(timeout=3.0)
    try:
        ser.close()
    except Exception:
        pass

    bursts = []
    asks = 0
```

```
    for line in lines:
        m = BURST_RE.search(line)
        if m:
            bursts.append((int(m.group(1)), int(m.group(2)), int(m.group(3)),
                           float(m.group(4))))
        if ASK_RE.search(line):
            asks += 1
        if any(k in line.lower() for k in
               ("missing", "stalled", "timed out", "resend", "bad crc")):
            print("  !", line)

    print(f"\nwindows asked for: {asks}")
    if not bursts:
        print("no burst completed - the sound never crossed the trigger")
        for line in lines[-25:]:
            print("  ", line)
        return 1

    print(f"\nbursts completed: {len(bursts)}")
    for node, have, want, secs in bursts:
        print(f"  node {node}: {have}/{want} fragments in {secs:.2f} s")
    times = [b[3] for b in bursts]
    times_sorted = sorted(times)
    print(f"\nburst time: min {min(times):.2f} s  "
          f"median {times_sorted[len(times) // 2]:.2f} s  max {max(times):.2f} s")
    print(f"fragments complete: {sum(1 for b in bursts if b[1] == b[2])}/"
          f"{len(bursts)}")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/measure_latency.py

156 lines · 5,682 bytes

```
#!/usr/bin/env python3
"""Measure the hub's end-to-end latency, and whether it calls the cry.

The chain, from the log:

  asking for the window   the trigger fired
  286/286 fragments in X  the burst arrived
  P(cry)=...              the model's answer
  ALERT - cry             the cry was confirmed

The script plays the recorded cry through the laptop's speakers near the
node and prints the chain, so the latency from the sound to the verdict
is visible rather than assumed.

Usage:
    python firmware/tools/measure_latency.py --runs 3
"""

from __future__ import annotations

import argparse
import os
import re
import sys
import threading
import time
from pathlib import Path

ROOT = Path(__file__).resolve().parents[2]
CRY_WAV = ROOT / "research" / "in_domain" / "session9" / "session.wav"

LINE_RE = re.compile(r"^I \((\d+)\) (\w+): (.*)$")


def one_run(port: str, warmup: float, sound: float, tail: float):
    import serial
    import winsound

    ser = serial.Serial(port, 115200, timeout=0.3)
    lines: list[tuple[float, int, str]] = []

    def reader():
        end = time.time() + warmup + sound + tail
        while time.time() < end:
            try:
```

```
                raw = ser.readline()
            except Exception:
                return
            if raw:
                body = raw.decode("utf-8", "replace").rstrip()
                m = LINE_RE.match(body)
                if m:
                    lines.append((time.monotonic(), int(m.group(1)), m.group(3)))

    t = threading.Thread(target=reader, daemon=True)
    t.start()
    time.sleep(warmup)
    sound_at = time.monotonic()
    winsound.PlaySound(str(CRY_WAV), winsound.SND_FILENAME | winsound.SND_ASYNC)
    time.sleep(sound)
    winsound.PlaySound(None, winsound.SND_PURGE)
    time.sleep(tail)
    t.join(timeout=3)
    try:
        ser.close()
    except Exception:
        pass

    out = []
    for wall, ts, body in lines:
        if "asking for the window" in body:
            out.append(("ask", ts, body))
        elif "fragments in" in body:
            out.append(("burst", ts, body))
        elif "P(cry)=" in body:
            out.append(("model", ts, body))
        elif re.search(r"node \d+: (ALERT|cry \(held\)|no cry|silent)", body):
            out.append(("verdict", ts, body))
        elif "stalled" in body or "timed out" in body:
            out.append(("stall", ts, body))
    return out, sound_at, lines


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--port", default="COM32")
    ap.add_argument("--runs", type=int, default=3)
    ap.add_argument("--warmup", type=float, default=6.0)
    ap.add_argument("--sound", type=float, default=16.0)
    ap.add_argument("--tail", type=float, default=18.0)
    args = ap.parse_args()

    print(f"=== {args.runs} run(s): the cry played from the laptop's speakers\n")
    totals = []
    detection = []
    for i in range(args.runs):
        marks, sound_at, lines = one_run(args.port, args.warmup, args.sound, args.tail)
        print(f"--- run {i + 1}")

        # The hub's own clock starts at its boot; the laptop's clock starts
        # now.  One logged line is enough to tie the two together, and the
        # sound is played at a known moment on the laptop's clock.
        sound_ts = None
        if lines:
            wall0, ts0, _ = lines[0]
            sound_ts = ts0 + (sound_at - wall0) * 1000.0

        ask = burst = model = verdict = None
        for kind, ts, body in marks:
            if kind == "ask":
                ask = ts
                print(f"  {ts / 1000:9.3f}s  {body[:64]}")
            elif kind == "burst":
                burst = ts
                secs = re.search(r"in ([0-9.]+) s", body)
                print(f"  {ts / 1000:9.3f}s  {body[:64]}")
                if ask and secs:
                    print(f"                 (the burst took {secs.group(1)} s)")
            elif kind == "model":
                model = ts
                print(f"  {ts / 1000:9.3f}s  {body[:64]}")
            elif kind == "verdict":
                verdict = ts
                print(f"  {ts / 1000:9.3f}s  {body[:64]}")
            elif kind == "stall":
                print(f"  {ts / 1000:9.3f}s  !! {body[:64]}")
        if ask and verdict:
            total = (verdict - ask) / 1000.0
            totals.append(total)
            print(f"                 ==> request to verdict: {total:.2f} s")
```

```
        if burst and model:
            print(f"                ==> burst to model: {(model - burst) / 1000.0:.2f} s")
        if sound_ts is not None and verdict:
            whole = (verdict - sound_ts) / 1000.0
            print(f"                ==> sound to verdict: {whole:.2f} s")
            if ask:
                det = (ask - sound_ts) / 1000.0
                detection.append(det)
                print(f"                ==> sound to request: {det:.2f} s")
        print()

    if totals:
        print(f"request to verdict: min {min(totals):.2f} s  "
              f"max {max(totals):.2f} s  mean {sum(totals) / len(totals):.2f} s")
    if detection:
        print(f"sound to request:   min {min(detection):.2f} s  "
              f"max {max(detection):.2f} s  mean {sum(detection) / len(detection):.2f} s")
    sys.stdout.flush()
    # The reader thread is a daemon and is still blocked in a read when
    # the interpreter wants to shut down; leaving it that way prints a
    # traceback that has nothing to do with the measurement.
    os._exit(0)


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/verify_burst.py

280 lines · 9,782 bytes

```
#!/usr/bin/env python3
"""Check the CryNet node's audio path end to end, on the real hardware.

The node can dump its frozen two-second window over the serial console in
exactly the framing the radio uses.  This tool asks for that dump, checks
every frame's CRC, reassembles the 64,000 bytes, and writes them as a WAV.

If this passes, then the microphone, the 32-bit-slot to int16 conversion,
the circular window, the framing and the CRCs are all correct on the
device — which is the part that cannot be checked on the host.

Usage:
    python firmware/tools/verify_burst.py --port COM30
    python firmware/tools/verify_burst.py --port COM30 --out burst.wav
"""

from __future__ import annotations

import argparse
import struct
import sys
import time
from pathlib import Path

MAGIC = bytes((0xC3, 0x7E))
FIXED = struct.Struct("<BBBBHHHH")   # magic0 magic1 type id seq frag count plen pcrc
TRAILER = struct.Struct("<H")
BURST_BYTES = 64000
SAMPLE_RATE = 16000

ESP32_VID_PIDS = {
    (0x1A86, 0x55D3): "CH343",
    (0x1A86, 0x7523): "CH340",
    (0x10C4, 0xEA60): "CP210x",
    (0x0403, 0x6001): "FTDI",
    (0x303A, 0x1001): "Espressif native USB (the C3 SuperMini's own USB)",
}


def crc16(data: bytes) -> int:
    crc = 0xFFFF
    for byte in data:
        crc ^= byte << 8
        for _ in range(8):
            crc = ((crc << 1) ^ 0x1021) & 0xFFFF if crc & 0x8000 else (crc << 1) & 0xFFFF
    return crc


def build_frame(node_id: int, ftype: int, seq: int, frag: int, count: int,
                payload: bytes) -> bytes:
    """The same layout the firmware's build_frame() produces."""
```

```
    frame = bytearray(14 + len(payload) + 2)
    frame[0], frame[1], frame[2], frame[3] = 0xC3, 0x7E, ftype, node_id
    frame[4:6] = struct.pack("<H", seq)
    frame[6:8] = struct.pack("<H", frag)
    frame[8:10] = struct.pack("<H", count)
    frame[10:12] = struct.pack("<H", len(payload))
    frame[12:14] = struct.pack("<H", crc16(payload) if payload else 0)
    frame[14:14 + len(payload)] = payload
    frame[14 + len(payload):] = struct.pack(
        "<H", crc16(bytes(frame[:14 + len(payload)])))
    return bytes(frame)


class FrameParser:
    """Pulls burst fragments out of a byte stream, checking every CRC.

    The node's console mixes text and binary frames, so the parser has to
    resynchronise on the magic.  A frame that fails its CRC is skipped
    one byte at a time, which is what lets it recover from a dropped
    byte instead of giving up.
    """

    def __init__(self) -> None:
        self.buf = bytearray()
        self.fragments: dict[int, bytes] = {}
        self.frag_count = None
        self.burst_seq = None
        self.node_id = None
        self.bad_crc = 0

    def feed(self, data: bytes) -> str:
        """Add bytes and return whatever text was in between the frames."""
        self.buf += data
        out = ""
        while True:
            start = self.buf.find(MAGIC)
            if start < 0:
                # Keep the last byte: a magic may straddle two reads.
                if len(self.buf) > 1:
                    out += self.buf[:-1].decode("utf-8", "replace")
                    del self.buf[:-1]
                break
            if start:
                out += self.buf[:start].decode("utf-8", "replace")
                del self.buf[:start]
            if len(self.buf) < FIXED.size:
                break

            _, _, ftype, fid, seq, frag, count, plen, pcrc = FIXED.unpack(
                self.buf[:FIXED.size])
            if ftype != 2 or plen > 4096:
                del self.buf[:1]
                continue
            total = FIXED.size + plen + TRAILER.size
            if len(self.buf) < total:
                break

            payload = bytes(self.buf[FIXED.size:FIXED.size + plen])
            fcrc = TRAILER.unpack(self.buf[FIXED.size + plen:total])[0]
            if (crc16(bytes(self.buf[:FIXED.size + plen])) != fcrc
                    or crc16(payload) != pcrc):
                self.bad_crc += 1
                del self.buf[:1]
                continue

            del self.buf[:total]
            self.frag_count = count
            self.burst_seq = seq
            self.node_id = fid
            self.fragments[frag] = payload
        return out

    def done(self) -> bool:
        return (self.frag_count is not None
                and len(self.fragments) >= self.frag_count)

    def assemble(self) -> bytes:
        if self.frag_count is None:
            return b""
        return b"".join(self.fragments[i] for i in range(self.frag_count)
                        if i in self.fragments)


def find_port() -> str | None:
    import serial.tools.list_ports as list_ports
```

```
137
138     for p in list_ports.comports():
139         if (p.vid, p.pid) in ESP32_VID_PIDS:
140             print(f"  found {p.device}: {p.description}")
141             return p.device
142     return None
143
144
145 def write_wav(path: Path, pcm: bytes) -> None:
146     with path.open("wb") as fh:
147         fh.write(b"RIFF" + struct.pack("<I", 36 + len(pcm)) + b"WAVE")
148         fh.write(b"fmt " + struct.pack("<IHHIIHH", 16, 1, 1, SAMPLE_RATE,
149                                        SAMPLE_RATE * 2, 2, 16))
150         fh.write(b"data" + struct.pack("<I", len(pcm)))
151         fh.write(pcm)
152
153
154 def main() -> int:
155     ap = argparse.ArgumentParser()
156     ap.add_argument("--port", help="serial port; auto-detected if omitted")
157     ap.add_argument("--out", default="burst.wav", help="where to write the WAV")
158     ap.add_argument("--baud", type=int, default=115200)
159     ap.add_argument("--timeout", type=float, default=90.0)
160     ap.add_argument("--settle", type=float, default=5.0,
161                     help="seconds to wait before asking for the window")
162     args = ap.parse_args()
163
164     try:
165         import serial
166     except ImportError:
167         print("pyserial is not installed: pip install pyserial", file=sys.stderr)
168         return 1
169
170     port = args.port or find_port()
171     if not port:
172         print("No ESP32 serial port found.", file=sys.stderr)
173         return 1
174
175     ser = serial.Serial(port, args.baud, timeout=0.2)
176     ser.dtr = False
177     ser.rts = False
178
179     parser = FrameParser()
180
181     def drain() -> str:
182         """Read whatever the device has sent, without ever waiting long.
183
184         The node's CDC transmit buffer is small, so a reader that waits
185         for a fixed block size will let the device overflow it and lose
186         bytes.  Asking for exactly what has arrived keeps the link drained.
187         """
188         waiting = ser.in_waiting
189         chunk = ser.read(waiting if waiting else 1)
190         return parser.feed(chunk) if chunk else ""
191
192     def pump(seconds: float) -> str:
193         deadline = time.time() + seconds
194         out = ""
195         while time.time() < deadline:
196             out += drain()
197         return out
198
199     print(f"Connected to {port}. Waiting {args.settle:.0f} s for the node to boot "
200           f"and its window to fill ...")
201     pump(args.settle)
202
203     # The node refuses to freeze until its two-second window is complete,
204     # so ask again until it answers.
205     frozen = False
206     for _ in range(12):
207         ser.write(b"freeze\n")
208         text = pump(1.0)
209         if text.strip():
210             print(text.strip()[-300:])
211         if "frozen seq=" in text:
212             frozen = True
213             break
214     if not frozen:
215         print("The node never froze its window — is it running node_crynet?",
216               file=sys.stderr)
217
218     ser.write(b"burst\n")
219     print("Requested the burst; reading ...")
220
221     deadline = time.time() + args.timeout
```

```
    while time.time() < deadline:
        text = drain()
        # The node prints BURST-END once it has sent every fragment, so
        # there is no point waiting for the timeout after that.
        if "BURST-END" in text:
            break
        if parser.done():
            break

    ser.close()

    if parser.frag_count is None:
        print("FAIL: no burst fragment received.", file=sys.stderr)
        return 1

    missing = [i for i in range(parser.frag_count) if i not in parser.fragments]
    pcm = parser.assemble()

    print()
    print(f"node id      {parser.node_id}")
    print(f"burst seq    {parser.burst_seq}")
    print(f"fragments    {len(parser.fragments)}/{parser.frag_count}  "
          f"(bad CRC {parser.bad_crc})")
    print(f"payload      {len(pcm)} bytes (expected {BURST_BYTES})")

    if missing:
        print(f"FAIL: missing fragments {missing[:20]}", file=sys.stderr)
        return 1
    if parser.bad_crc:
        print(f"FAIL: {parser.bad_crc} frames failed their CRC", file=sys.stderr)
        return 1
    if len(pcm) != BURST_BYTES:
        print(f"FAIL: reassembled {len(pcm)} bytes, expected {BURST_BYTES}",
              file=sys.stderr)
        return 1

    samples = struct.unpack(f"<{len(pcm) // 2}h", pcm)
    mean = sum(samples) / len(samples)
    rms = (sum((s - mean) ** 2 for s in samples) / len(samples)) ** 0.5 / 32768.0
    peak = max(abs(s) for s in samples)

    out = Path(args.out)
    write_wav(out, pcm)
    print(f"mean         {mean:.1f} counts (the INMP441's DC offset)")
    print(f"rms          {rms:.5f}")
    print(f"peak         {peak} counts")
    print(f"wrote        {out}  ({len(pcm) / 2 / SAMPLE_RATE:.2f} s)")

    if peak == 0:
        print("FAIL: the window is silent — check the microphone wiring.",
              file=sys.stderr)
        return 1

    print("\nPASS: the node's capture, framing and CRC are correct on the device.")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/ble_alert_test.py

111 lines · 3,908 bytes

```
#!/usr/bin/env python3
"""Talk to the CryNet hub's BLE alert server from a PC.

The hub is a GATT server with the legacy's UUIDs: the service
19b10000-e8f2-537e-4f6c-d104768a1214 and the alert characteristic
19b10001-e8f2-537e-4f6c-d104768a1214, plus a state characteristic
19b10002-e8f2-537e-4f6c-d104768a1214.  The alert value is the legacy's
six-byte packet:

    cot id, alert type (1 = cry, 2 = the cry has stopped),
    confidence in percent, event id.

Usage:
    python firmware/tools/ble_alert_test.py                # scan only
    python firmware/tools/ble_alert_test.py --listen       # scan, connect,
                                                           # subscribe, wait
"""

```

```python
from __future__ import annotations

import argparse
import asyncio
import struct
import sys

DEVICE_NAME = "INFANT_MONITOR_HUB"
SERVICE_UUID = "19b10000-e8f2-537e-4f6c-d104768a1214"
ALERT_UUID = "19b10001-e8f2-537e-4f6c-d104768a1214"
STATE_UUID = "19b10002-e8f2-537e-4f6c-d104768a1214"


def decode_alert(data: bytes) -> dict:
    if len(data) < 6:
        return {"raw": data.hex()}
    cot, kind, confidence, event = struct.unpack("<BBHH", data[:6])
    return {
        "cot": cot,
        "alert": "cry" if kind == 1 else ("cleared" if kind == 2 else kind),
        "confidence_pct": confidence,
        "event_id": event,
    }


def decode_state(data: bytes) -> dict:
    if len(data) < 8:
        return {"raw": data.hex()}
    cots, mask, _res, confidence, started = struct.unpack("<BBBBI", data[:8])
    return {
        "cots": cots,
        "crying_mask": mask,
        "confidence_pct": confidence,
        "started_at_s": started,
    }


async def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--listen", action="store_true",
                    help="connect and subscribe, instead of only scanning")
    ap.add_argument("--seconds", type=float, default=120.0)
    ap.add_argument("--name", default=DEVICE_NAME)
    args = ap.parse_args()

    try:
        from bleak import BleakClient, BleakScanner
    except ImportError:
        print("bleak is not installed: pip install bleak", file=sys.stderr)
        return 1

    print(f"scanning for \"{args.name}\" ...")
    device = await BleakScanner.find_device_by_name(args.name, timeout=20.0)
    if device is None:
        print("the hub was not found. Is it powered and advertising?",
              file=sys.stderr)
        return 1
    print(f"found {device.name} at {device.address}")

    if not args.listen:
        return 0

    async def on_alert(_char, data: bytearray):
        print("ALERT", decode_alert(bytes(data)), flush=True)

    async def on_state(_char, data: bytearray):
        print("STATE", decode_state(bytes(data)), flush=True)

    async with BleakClient(device) as client:
        print("connected")
        for service in client.services:
            for char in service.characteristics:
                print(f"  service {service.uuid}  characteristic {char.uuid} "
                      f"{char.properties}")

        try:
            await client.start_notify(ALERT_UUID, on_alert)
            print("subscribed to the alert characteristic", flush=True)
        except Exception as exc:                        # noqa: BLE001
            print(f"the alert characteristic could not be subscribed to: {exc}")
        try:
            await client.start_notify(STATE_UUID, on_state)
            print("subscribed to the state characteristic", flush=True)
        except Exception as exc:                        # noqa: BLE001
            print(f"the state characteristic could not be subscribed to: {exc}")
```

```
104
105         print(f"listening for {args.seconds:.0f} s ...")
106         await asyncio.sleep(args.seconds)
107     return 0
108
109
110 if __name__ == "__main__":
111     sys.exit(asyncio.run(main()))
```

firmware/tools/serve_alert_page.py

134 lines · 4,938 bytes

```
1  #!/usr/bin/env python3
2  """Serve the CryNet alert page over HTTPS, so a phone can use Web Bluetooth.
3
4  Web Bluetooth only works in a *secure context*, and on a phone that
5  means HTTPS: a plain http:// page from the PC's address is not secure
6  enough, and neither is a file:// page.  So this serves the page with a
7  self-signed certificate; the phone shows a certificate warning once, and
8  "Advanced -> Proceed to..." makes the page a secure context, after which
9  the browser will offer the hub in its device chooser.
10
11 The page itself is research/web/alert.html.  Nothing is sent anywhere:
12 the server only hands out the one file, and the browser talks to the hub
13 over Bluetooth.
14
15 Usage:
16     python firmware/tools/serve_alert_page.py
17     python firmware/tools/serve_alert_page.py --port 8443 --no-tls
18 """
19
20 from __future__ import annotations
21
22 import argparse
23 import http.server
24 import os
25 import socket
26 import shutil
27 import ssl
28 import subprocess
29 import sys
30 from pathlib import Path
31
32 ROOT = Path(__file__).resolve().parents[2]
33 PAGE = ROOT / "research" / "web" / "alert.html"
34 CERT_DIR = ROOT / "research" / "web" / ".cert"
35
36
37 def local_addresses() -> list[str]:
38     """The addresses a phone on the same network could reach."""
39     out = []
40     try:
41         s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
42         s.connect(("8.8.8.8", 80))
43         out.append(s.getsockname()[0])
44         s.close()
45     except OSError:
46         pass
47     return out
48
49
50 def make_certificate() -> tuple[Path, Path]:
51     """A self-signed certificate for this machine, made once."""
52     CERT_DIR.mkdir(parents=True, exist_ok=True)
53     cert = CERT_DIR / "cert.pem"
54     key = CERT_DIR / "key.pem"
55     if cert.is_file() and key.is_file():
56         return cert, key
57     # openssl ships with Git for Windows; it is the simplest way to make a
58     # certificate without another dependency.  It is not on the Windows
59     # PATH, though, so the usual places are tried as well.
60     openssl = shutil.which("openssl")
61     if openssl is None:
62         for candidate in (r"C:\Program Files\Git\mingw64\bin\openssl.exe",
63                           r"C:\Program Files\Git\usr\bin\openssl.exe",
64                           "/mingw64/bin/openssl", "/usr/bin/openssl"):
65             if Path(candidate).is_file():
66                 openssl = candidate
67                 break
68     if openssl is None:
69         print("openssl was not found", file=sys.stderr)
```

```
        sys.exit(1)
    cmd = [
        openssl, "req", "-x509", "-newkey", "rsa:2048", "-nodes",
        "-keyout", str(key), "-out", str(cert), "-days", "365",
        "-subj", "/CN=crynet-hub-alert",
    ]
    # openssl takes one -addext per name, so the addresses go into a
    # single comma-separated list.
    addresses = ["127.0.0.1"] + local_addresses()
    cmd += ["-addext",
            "subjectAltName=" + ",".join(f"IP:{a}" for a in addresses)]
    try:
        subprocess.run(cmd, check=True, capture_output=True)
    except (OSError, subprocess.CalledProcessError) as exc:
        print(f"the certificate could not be made: {exc}", file=sys.stderr)
        print("openssl is needed for the HTTPS mode; --no-tls serves it "
              "plainly, which the PC can use but a phone cannot (Web "
              "Bluetooth needs a secure context).", file=sys.stderr)
        sys.exit(1)
    return cert, key


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--port", type=int, default=8443)
    ap.add_argument("--no-tls", action="store_true",
                    help="serve plain HTTP (the PC can use it; a phone cannot)")
    ap.add_argument("--bind", default="0.0.0.0")
    args = ap.parse_args()

    if not PAGE.is_file():
        print(f"{PAGE} is missing", file=sys.stderr)
        return 1

    class Handler(http.server.SimpleHTTPRequestHandler):
        def __init__(self, *a, **kw):
            super().__init__(*a, directory=str(PAGE.parent), **kw)

        def log_message(self, fmt, *a):
            pass   # quiet

    httpd = http.server.ThreadingHTTPServer((args.bind, args.port), Handler)
    scheme = "http"
    if not args.no_tls:
        cert, key = make_certificate()
        context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)
        context.load_cert_chain(cert, key)
        httpd.socket = context.wrap_socket(httpd.socket, server_side=True)
        scheme = "https"

    print(f"serving {PAGE.name} on {scheme}://{args.bind}:{args.port}")
    for address in local_addresses():
        print(f"  on the phone: {scheme}://{address}:{args.port}/")
    print("  the certificate is self-signed, so the phone will warn once: "
          "Advanced -> Proceed to ...")
    print("Ctrl-C to stop.")
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("\nstopped")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/who_is_who.py

60 lines · 1,717 bytes

```
#!/usr/bin/env python3
"""Say which node is on which port.

Node id is kept in NVS, so it survives a reflash -- but only if the
flash tool does not erase the partition.  This script does not flash at
all: it opens each port, resets the board over RTS/EN, and reads the
banner the firmware prints at boot, which names the node and its MAC.

Usage:
    python firmware/tools/who_is_who.py COM37 COM38
"""

```

```
import sys
import time

import serial


def read_banner(port: str, settle: float = 0.6, listen: float = 4.0) -> str:
    """Open the port, reset the board, and return what it says at boot."""
    s = serial.Serial(port, 115200, timeout=0.2)
    try:
        # The board may be mid-boot; drain whatever is buffered.
        time.sleep(0.3)
        s.reset_input_buffer()
        # ESP32: RTS drives EN, DTR drives GPIO0.  RTS alone resets.
        s.dtr = False
        s.rts = True
        time.sleep(0.15)
        s.rts = False
        time.sleep(settle)

        out = []
        deadline = time.time() + listen
        while time.time() < deadline:
            chunk = s.read(s.in_waiting or 1)
            if chunk:
                out.append(chunk.decode("utf-8", "replace"))
        return "".join(out)
    finally:
        s.close()


def main() -> int:
    ports = sys.argv[1:]
    if not ports:
        print(__doc__)
        return 1
    for p in ports:
        print(f"===== {p} =====")
        try:
            print(read_banner(p))
        except Exception as exc:  # noqa: BLE001 -- report, do not crash
            print(f"could not read {p}: {exc}")
        print()
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

firmware/tools/patch_deps.py

124 lines · 4,425 bytes

```
#!/usr/bin/env python3
"""Make the vendored TFLite Micro port build against the vendored esp-nn.

The Espressif TFLite Micro port declares its ESP-NN dependency in
`idf_component.yml`:

    dependencies:
      idf:
        version: ">=5.1"
      espressif/esp-nn:
        version: ">=1.3.0"

That declaration makes the IDF **component manager** download esp-nn from
the component registry. We vendor esp-nn instead, pinned to a commit in
`firmware/tools/fetch_deps.sh`, for three reasons:

  * the build stays offline and reproducible — no registry version drift;
  * the exact revision is visible in the repository rather than hidden in a
    lock file;
  * ESP-NN is worth having: it is the difference between roughly 40 ms and
    roughly 2.3 s for a convnet invoke on an ESP32-S3.

A local component named `esp-nn` and a registry-installed `esp-nn` in the
same build is a duplicate-component error, and the manager's injected
requirement would be redundant. So this script:

  1. removes the `espressif/esp-nn` entry from the port's
     `idf_component.yml`, leaving the `idf` version requirement intact;
  2. adds `esp-nn` to the port's own component requirements, which is what
```

```
     actually puts `esp_nn.h` on the include path and links the library.

Both edits are idempotent: running the script twice changes nothing the
second time. `fetch_deps.sh` calls it after every checkout.

Usage:  python firmware/tools/patch_deps.py [--check]
"""

from __future__ import annotations

import argparse
import sys
from pathlib import Path

ROOT = Path(__file__).resolve().parents[1]
# The vendored components live inside the ESP-IDF project itself (see the
# note in firmware/hub_infer/CMakeLists.txt about why that matters).
TFLM = ROOT / "hub_infer" / "components" / "esp-tflite-micro"

YML_OLD = """  espressif/esp-nn:
    # >=1.3.0: grouped-conv support the conv wrapper now relies on
    version: ">=1.3.0"
"""

YML_NEW = """  # espressif/esp-nn is vendored alongside this port (see
  # firmware/tools/fetch_deps.sh) and added by firmware/tools/patch_deps.py,
  # so it is deliberately not requested from the component registry here.
"""

CMAKE_OLD = "set(priv_req)\n"
CMAKE_NEW = "set(priv_req esp-nn)\n"


def patch_yml(text: str) -> tuple[str, bool]:
    if YML_NEW in text:
        return text, False
    if YML_OLD not in text:
        raise SystemExit(
            "patch_deps: the ESP-NN block in esp-tflite-micro/idf_component.yml "
            "is not the expected one — refusing to guess."
        )
    return text.replace(YML_OLD, YML_NEW), True


def patch_cmake(text: str) -> tuple[str, bool]:
    if CMAKE_NEW in text:
        return text, False
    if CMAKE_OLD not in text:
        raise SystemExit(
            "patch_deps: `set(priv_req)` not found in "
            "esp-tflite-micro/CMakeLists.txt — refusing to guess."
        )
    return text.replace(CMAKE_OLD, CMAKE_NEW, 1), True


def main() -> int:
    ap = argparse.ArgumentParser()
    ap.add_argument("--check", action="store_true",
                    help="report whether the patches are applied; change nothing")
    args = ap.parse_args()

    yml = TFLM / "idf_component.yml"
    cmake = TFLM / "CMakeLists.txt"
    for f in (yml, cmake):
        if not f.is_file():
            raise SystemExit(
                f"patch_deps: {f} is missing — run firmware/tools/fetch_deps.sh first."
            )

    yml_text = yml.read_text(encoding="utf-8")
    cmake_text = cmake.read_text(encoding="utf-8")

    new_yml, yml_changed = patch_yml(yml_text)
    new_cmake, cmake_changed = patch_cmake(cmake_text)

    if args.check:
        ok = not yml_changed and not cmake_changed
        print("patch_deps: patches applied" if ok else
              "patch_deps: patches MISSING — run firmware/tools/fetch_deps.sh")
        return 0 if ok else 1

    if yml_changed:
        yml.write_text(new_yml, encoding="utf-8")
        print("patch_deps: esp-nn registry dependency removed from idf_component.yml")
    if cmake_changed:
```

```
        cmake.write_text(new_cmake, encoding="utf-8")
        print("patch_deps: esp-nn added to the port's component requirements")

    if not (yml_changed or cmake_changed):
        print("patch_deps: already patched")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/fetch_deps.sh

44 lines · 1,486 bytes

```
#!/usr/bin/env bash
# Fetch the vendored firmware dependencies at their pinned revisions.
#
# The CryNet firmware builds against exactly these commits; they are not
# tracked in this repository so the history stays readable.
#
#   bash firmware/tools/fetch_deps.sh
#
set -euo pipefail

HERE="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
DEST="${HERE}/../hub_infer/components"

TFLM_REPO="https://github.com/espressif/esp-tflite-micro.git"
TFLM_REV="dd4085950a119d9e33a6de1d7be1fc7fe29e0cfc"

NN_REPO="https://github.com/espressif/esp-nn.git"
NN_REV="744d48deae6aa4a4d7b93b6a80f548d5298813a4"

clone_at() {
    local repo="$1" rev="$2" dir="$3"
    if [ -d "${dir}/.git" ]; then
        echo "==> ${dir} already present"
    else
        echo "==> cloning ${repo} -> ${dir}"
        git clone --filter=blob:none "${repo}" "${dir}"
    fi
    echo "==> checking out ${rev}"
    git -C "${dir}" fetch --depth 1 origin "${rev}" || true
    git -C "${dir}" checkout "${rev}"
}

mkdir -p "${DEST}"
clone_at "${TFLM_REPO}" "${TFLM_REV}" "${DEST}/esp-tflite-micro"
clone_at "${NN_REPO}" "${NN_REV}" "${DEST}/esp-nn"

# Point the port at the vendored esp-nn instead of the component registry.
# See firmware/tools/patch_deps.py for why.
python "${HERE}/patch_deps.py"

echo
echo "Dependencies ready:"
echo "  esp-tflite-micro  $(git -C "${DEST}/esp-tflite-micro" rev-parse HEAD)"
echo "  esp-nn            $(git -C "${DEST}/esp-nn" rev-parse HEAD)"
```

The record's proofs, and the dashboard's contract

firmware/tools/test_log_order.py

323 lines · 11,509 bytes

```
"""The cry log's ordering, proved on the host.

The hub numbers a record when the episode *starts* and writes it when
the episode *ends*.  Two cots crying at once can end in the opposite
order to the one they started in -- cot 1's long cry holds its number
while cot 2's short one finishes first -- so the file of records is
**not** in number order.

The old code found the records still to send with a binary search over
```

```
the file, which is only valid if the file is sorted.  It was not, and
the binary search answered wrongly in exactly the case that matters:
asked for the first record above a number, it could land *past* a
record whose number was lower.  That record was then never sent, and
the hub said it had sent everything.  The dashboard simply never showed
that cry.

The log now walks the file in order instead, and the number it claims
as sent stops below two things: the lowest number that did not fit in
the batch, and the lowest number an episode still holds while it is
being measured.

This file models that algorithm exactly and proves the property that
matters:

    every record that was written is eventually sent.

It is a model, not the firmware: the real proof is the hub's own
`logtest` command, which runs the same scenario against the real
component on the real chip (see crynet_log.c).  This one runs
anywhere, with no hardware, and it also demonstrates the bug it
guards, so the old behaviour cannot be reintroduced by accident.

    python firmware/tools/test_log_order.py
"""

from __future__ import annotations

import random
import sys


class Log:
    """The cry log's bookkeeping, as the firmware keeps it."""

    def __init__(self) -> None:
        # The file: records in the order they were written.
        self.file: list[int] = []
        self.next_seq = 1
        self.uploaded_seq = 0
        # Episodes that have started and not yet ended: their numbers.
        self.open: list[int] = []

    # ---- writing ------------------------------------------------------

    def reserve(self) -> int:
        seq = self.next_seq
        self.next_seq += 1
        self.open.append(seq)
        return seq

    def commit(self, seq: int) -> None:
        self.file.append(seq)
        self.open.remove(seq)

    # ---- the current algorithm ----------------------------------------

    def _collect(self, after: int, max_records: int) -> tuple[list[int], int, int]:
        """The records above `after`: the batch, the highest number in
        it, and the lowest number that did not fit.

        The batch is the *lowest* numbers, not the first ones in the
        file: the file is in the order the records were written, which
        is not the order they were numbered, and taking the first ones
        in the file could pick a high number while a lower one sits
        behind it -- and then that lower one would be sent over and
        over and the log would never move on.
        """
        qualifying = sorted(seq for seq in self.file if seq > after)
        batch = qualifying[:max_records]
        last_seq = batch[-1] if batch else 0
        stuck = qualifying[max_records] if len(qualifying) > max_records else 0
        return batch, last_seq, stuck

    def _watermark(self, last_seq: int, stuck: int) -> int:
        w = last_seq
        if stuck != 0 and (w == 0 or stuck - 1 < w):
            w = stuck - 1
        for seq in self.open:
            if seq - 1 < w:
                w = seq - 1
        if w < self.uploaded_seq:
            w = self.uploaded_seq
        return w

```

```
    def pending(self, max_records: int = 32) -> tuple[list[int], int]:
        batch, last_seq, stuck = self._collect(self.uploaded_seq, max_records)
        return batch, self._watermark(last_seq, stuck)

    def mark_uploaded(self, watermark: int) -> None:
        if watermark > self.uploaded_seq:
            self.uploaded_seq = watermark

    # ---- the old algorithm, kept so the bug stays visible --------------

    def _first_above_old(self, seq: int) -> int:
        """The old binary search.  Valid only on a sorted file."""
        lo, hi = 0, len(self.file)
        while lo < hi:
            mid = lo + (hi - lo) // 2
            if self.file[mid] > seq:
                hi = mid
            else:
                lo = mid + 1
        return lo

    def pending_old(self, max_records: int = 32) -> list[int]:
        start = self._first_above_old(self.uploaded_seq)
        return self.file[start:start + max_records]

    def mark_uploaded_old(self, last: int) -> None:
        if last > self.uploaded_seq:
            self.uploaded_seq = last


def drain(log: Log, batch: int = 32, sent: list[int] | None = None) -> list[int]:
    """Send everything, the way the uploader does, and return what went."""
    if sent is None:
        sent = []
    for _ in range(1000):
        records, watermark = log.pending(batch)
        if not records:
            break
        sent.extend(records)
        log.mark_uploaded(watermark)
    return sent


def drain_old(log: Log, batch: int = 32, sent: list[int] | None = None) -> list[int]:
    if sent is None:
        sent = []
    for _ in range(1000):
        records = log.pending_old(batch)
        if not records:
            break
        sent.extend(records)
        log.mark_uploaded_old(records[-1])
    return sent


# ---------------------------------------------------------------------
# The bug, and the case that caused it.
# ---------------------------------------------------------------------

def test_two_cots_out_of_order() -> None:
    """Cot 1's long cry starts first, cot 2's short one finishes first.

    The record of the short cry is written first and holds the higher
    number, so the file is out of order.  The uploader runs while the
    long cry is still going -- that is the ordinary case, the hub is on
    the network all night -- and both records must still be sent.
    """
    log = Log()
    long_cry = log.reserve()     # cot 1, starts first
    short_cry = log.reserve()    # cot 2, starts second
    log.commit(short_cry)        # ... but finishes first

    sent = drain(log)
    assert short_cry in sent, "the short cry was never sent"

    log.commit(long_cry)         # cot 1's cry ends later
    sent = drain(log)
    assert long_cry in sent, "the long cry was never sent"


def test_the_old_algorithm_loses_the_record() -> None:
    """The bug this guards, shown rather than described.

    The old code searched the file for the first record above the
    number it had sent.  With the file out of order it landed past a
```

```
    record that had never been sent, and that record was never sent at
    all.
    """
    log = Log()
    long_cry = log.reserve()
    short_cry = log.reserve()
    log.commit(short_cry)

    # The uploader runs while the long cry is still going.
    first = drain_old(log)
    assert short_cry in first, "the old code did not send the short cry"

    log.commit(long_cry)
    second = drain_old(log)

    assert long_cry not in first + second, (
        "the old code is expected to lose the long cry; if it does not, "
        "this test is no longer describing the bug it guards")


def test_a_batch_that_does_not_fit() -> None:
    """More records than one batch holds, with an episode still open."""
    log = Log()
    seqs = [log.reserve() for _ in range(40)]
    for seq in seqs:
        log.commit(seq)
    still_open = log.reserve()          # starts, does not finish yet

    sent = drain(log, batch=32)
    assert sorted(set(sent)) == sorted(seqs), "a record was lost"
    assert still_open not in sent

    # Now the open episode finishes and must be sent too.
    log.commit(still_open)
    sent = drain(log, batch=32)
    assert still_open in sent, "the episode that was still open was lost"


def test_an_episode_open_across_the_whole_upload() -> None:
    """The awkward case: one episode holds its number throughout."""
    log = Log()
    held = log.reserve()
    other = log.reserve()
    log.commit(other)

    sent = drain(log)
    assert other in sent, "the finished episode was not sent"
    assert held not in sent, "the open episode was sent before it finished"

    log.commit(held)
    sent = drain(log)
    assert held in sent, "the episode that finished late was never sent"


def test_soak() -> None:
    """Ten thousand random interleavings, every record sent.

    Episodes are reserved and committed in a random order, the uploader
    runs at random moments in between, in batches of a random size, and
    an episode is sometimes left open across the whole upload.
    """
    rng = random.Random(20261124)
    for _ in range(10000):
        log = Log()
        episodes = [log.reserve() for _ in range(rng.randint(1, 12))]
        rng.shuffle(episodes)
        batch = rng.choice([1, 2, 3, 8, 32])

        sent: list[int] = []
        written: list[int] = []
        for seq in episodes:
            log.commit(seq)
            written.append(seq)
            if rng.random() < 0.4:
                drain(log, batch=batch, sent=sent)
        drain(log, batch=batch, sent=sent)

        # Anything still open finishes now and must go out too.
        for seq in list(log.open):
            log.commit(seq)
            written.append(seq)
        drain(log, batch=batch, sent=sent)

        assert set(written) <= set(sent), (
            f"lost {sorted(set(written) - set(sent))} "
```

```
            f"(batch {batch}, order {episodes})")


def test_the_soak_catches_the_old_algorithm() -> None:
    """The soak is only worth running if it fails on the old code.

    It is run here against the old algorithm, and it is expected to
    lose records.  If it ever stops losing them, the soak has stopped
    testing what it was written to test.
    """
    rng = random.Random(20261124)
    lost_once = False
    for _ in range(2000):
        log = Log()
        episodes = [log.reserve() for _ in range(rng.randint(1, 12))]
        rng.shuffle(episodes)
        batch = rng.choice([1, 2, 3, 8, 32])

        sent: list[int] = []
        written: list[int] = []
        for seq in episodes:
            log.commit(seq)
            written.append(seq)
            if rng.random() < 0.4:
                drain_old(log, batch=batch, sent=sent)
        drain_old(log, batch=batch, sent=sent)
        for seq in list(log.open):
            log.commit(seq)
            written.append(seq)
        drain_old(log, batch=batch, sent=sent)

        if not set(written) <= set(sent):
            lost_once = True
            break

    assert lost_once, (
        "the old algorithm lost nothing in two thousand runs; the soak is "
        "no longer testing the thing it was written to test")


def main() -> int:
    tests = [
        test_two_cots_out_of_order,
        test_the_old_algorithm_loses_the_record,
        test_a_batch_that_does_not_fit,
        test_an_episode_open_across_the_whole_upload,
        test_soak,
        test_the_soak_catches_the_old_algorithm,
    ]
    for test in tests:
        test()
        print(f"  ok  {test.__name__}")
    print()
    print("OK  every record the log writes is eventually sent")
    return 0


if __name__ == "__main__":
    sys.exit(main())
```

firmware/tools/test_dashboard_contract.py

300 lines · 13,000 bytes

```
#!/usr/bin/env python3
"""Prove the hub and the dashboard agree, without any hardware.

The hub builds its upload body by hand with snprintf (ESP-IDF ships no
JSON library).  This script builds the *same* body, byte for byte, from
the same fields, posts it to a running dashboard, and checks that what
comes back out of the API is what went in.

That is the one contract in the system that no compiler checks: the
firmware and the page are written in different languages and a field
renamed on either side would fail silently -- the hub would report
success and the dashboard would show nothing.

    python firmware/tools/test_dashboard_contract.py
    python firmware/tools/test_dashboard_contract.py --port 8787 --token SECRET

It starts its own throwaway server on a free port unless you point it at
one, and it never writes inside the repository.
```

```
"""

from __future__ import annotations

import argparse
import json
import os
import socket
import sqlite3
import subprocess
import sys
import tempfile
import time
import urllib.error
import urllib.request
from pathlib import Path

ROOT = Path(__file__).resolve().parents[2]
SERVER = ROOT / "dashboard" / "server.py"

FW_VERSION = "2.0-dashboard"       # CRYNET_FW_VERSION in crynet_net.c


def hub_body(device: str, name: str, records: list[dict]) -> str:
    """Exactly what build_batch() in crynet_net.c produces.

    Keep this in step with that function.  The formatting is deliberate:
    %.4f for the levels, integers everywhere else, no spaces.
    """
    parts = []
    for r in records:
        dur_s = (r["duration_ms"] + 500) // 1000
        end = r["epoch_s"] + dur_s if r["epoch_s"] else 0
        parts.append(
            '{"seq":%d,"kind":"%s","cot":%d,'
            '"start":%d,"end":%d,"duration_s":%d,'
            '"peak_conf":%d,"windows":%d,'
            '"peak_rms":%.4f,"mean_rms":%.4f,'
            '"boot":%d,"uptime_ms":%d}'
            % (r["seq"], r["kind"], r["cot"], r["epoch_s"], end, dur_s,
               r["peak_conf"], r["windows"],
               r["peak_rms_milli"] / 1000.0, r["mean_rms_milli"] / 1000.0,
               r["boot_id"], r["uptime_ms"])
        )
    return ('{"device":"%s","hub":{"fw":"%s","name":"%s"},"events":[%s]}'
            % (device, FW_VERSION, name, ",".join(parts)))


def post(url: str, body: str, token: str) -> dict:
    req = urllib.request.Request(
        url, data=body.encode("utf-8"), method="POST",
        headers={"Content-Type": "application/json"})
    if token:
        req.headers["X-CryNet-Token"] = token
    with urllib.request.urlopen(req, timeout=10) as r:
        return json.loads(r.read().decode("utf-8"))


def get(url: str) -> dict:
    with urllib.request.urlopen(url, timeout=10) as r:
        return json.loads(r.read().decode("utf-8"))


def free_port() -> int:
    s = socket.socket()
    s.bind(("127.0.0.1", 0))
    p = s.getsockname()[1]
    s.close()
    return p


def wait_up(base: str, proc=None, seconds: float = 20.0) -> None:
    deadline = time.time() + seconds
    while time.time() < deadline:
        if proc is not None and proc.poll() is not None:
            raise SystemExit("the dashboard exited during startup")
        try:
            get(base + "/api/v1/health")
            return
        except (urllib.error.URLError, OSError):
            time.sleep(0.25)
    raise SystemExit("the dashboard did not come up")


def main() -> int:
```

```
    ap = argparse.ArgumentParser(description=__doc__)
    ap.add_argument("--port", type=int, default=0,
                    help="an already-running dashboard; 0 starts a throwaway one")
    ap.add_argument("--token", default="")
    args = ap.parse_args()

    tmp = None
    proc = None
    port = args.port
    if port == 0:
        port = free_port()
        tmp = tempfile.TemporaryDirectory(prefix="crynet-contract-")
        db = Path(tmp.name) / "contract.db"
        cmd = [sys.executable, "-I", str(SERVER), "--port", str(port),
               "--host", "127.0.0.1", "--db", str(db)]
        if args.token:
            cmd += ["--token", args.token]
        proc = subprocess.Popen(cmd, stdout=subprocess.DEVNULL,
                                stderr=subprocess.DEVNULL)
    base = f"http://127.0.0.1:{port}"
    failures = []

    try:
        wait_up(base, proc)

        # A night's worth of records, in the hub's own units.  Two cots,
        # a boot, a cot going deaf and coming back, and one record with
        # no clock at all -- the case the hub is in before it reaches a
        # time server.
        now = int(time.time())
        evening = now - (now % 86400) - 3600      # late yesterday evening
        recs = [
            dict(seq=1, kind="boot", cot=0, epoch_s=evening - 600,
                 duration_ms=0, peak_conf=0, windows=0,
                 peak_rms_milli=0, mean_rms_milli=0, boot_id=7,
                 uptime_ms=1200),
            dict(seq=2, kind="cry", cot=1, epoch_s=evening,
                 duration_ms=23400, peak_conf=100, windows=11,
                 peak_rms_milli=412, mean_rms_milli=187, boot_id=7,
                 uptime_ms=612000),
            dict(seq=3, kind="cry", cot=2, epoch_s=evening + 1800,
                 duration_ms=8600, peak_conf=99, windows=4,
                 peak_rms_milli=233, mean_rms_milli=140, boot_id=7,
                 uptime_ms=2412000),
            dict(seq=4, kind="deaf", cot=2, epoch_s=evening + 2400,
                 duration_ms=0, peak_conf=0, windows=0,
                 peak_rms_milli=0, mean_rms_milli=0, boot_id=7,
                 uptime_ms=3012000),
            dict(seq=5, kind="heard", cot=2, epoch_s=evening + 2700,
                 duration_ms=0, peak_conf=0, windows=0,
                 peak_rms_milli=0, mean_rms_milli=0, boot_id=7,
                 uptime_ms=3312000),
            dict(seq=6, kind="cry", cot=1, epoch_s=0,      # no clock yet
                 duration_ms=15000, peak_conf=100, windows=7,
                 peak_rms_milli=305, mean_rms_milli=190, boot_id=8,
                 uptime_ms=45000),
        ]
        device = "hub-7c9ebd12"
        body = hub_body(device, "Nursery hub", recs)

        # The body must be valid JSON before anything else is worth saying.
        try:
            json.loads(body)
        except ValueError as e:
            raise SystemExit(f"FAIL  the hub's body is not valid JSON: {e}")

        ans = post(base + "/api/v1/events", body, args.token)
        if ans.get("received") != len(recs):
            failures.append(f"the server received {ans.get('received')} "
                            f"of {len(recs)} records")
        print(f"  posted {len(recs)} records -> {ans}")

        # Posting the same batch again is what a retry after a dropped
        # connection looks like.  It must not double-count.
        ans2 = post(base + "/api/v1/events", body, args.token)
        if ans2.get("accepted") != 0:
            failures.append(f"a repeat batch added {ans2.get('accepted')} "
                            f"records; it should add none")
        print(f"  posted the same batch again -> {ans2}")

        s = get(base + "/api/v1/summary?days=90")
        k = s["kpis"]
        if k["total_cries"] != 3:
            failures.append(f"summary counts {k['total_cries']} cries, "
                            f"expected 3 (two timed, one untimed)")
```

```
        cots = {c["cot"]: c for c in s["per_cot"]}
        if set(cots) != {1, 2}:
            failures.append(f"per_cot has {sorted(cots)}, expected [1, 2]")
        if 1 in cots and cots[1]["cries"] != 2:
            failures.append(f"cot 1 shows {cots[1]['cries']} cries, expected 2")

        # The untimed record must be counted but never placed on a time
        # axis.  This is the honesty rule (D-90).
        placed = sum(d["cries"] for d in s["per_day"])
        if placed != 2:
            failures.append(f"{placed} cries placed on the day axis, "
                            f"expected 2 -- the untimed record must not be placed")
        hour_total = sum(h["cries"] for h in s["per_hour"])
        if hour_total != 2:
            failures.append(f"{hour_total} cries placed on the hour axis, "
                            f"expected 2")

        a = get(base + "/api/v1/analytics?days=90")
        if "duration_hist" not in a or "insights" not in a:
            failures.append("analytics is missing duration_hist or insights")

        ev = get(base + "/api/v1/events?limit=50")
        rows = ev["events"] if isinstance(ev, dict) else ev
        kinds = {r["kind"] for r in rows}
        if not {"cry", "deaf", "heard", "boot"} <= kinds:
            failures.append(f"the event list is missing kinds: {sorted(kinds)}")
        by_seq = {r["seq"]: r for r in rows if r["device"] == device}
        if 2 in by_seq:
            r = by_seq[2]
            if r["duration_s"] != 23:
                failures.append(f"record 2 lasted {r['duration_s']}s, "
                                f"expected 23 (23400 ms rounded)")
            if abs(r["peak_rms"] - 0.412) > 1e-6:
                failures.append(f"record 2 peak_rms {r['peak_rms']}, "
                                f"expected 0.412")
        else:
            failures.append("record 2 is not in the event list")

        d = get(base + "/api/v1/devices")
        names = d["devices"] if isinstance(d, dict) else d
        if not any((x.get("device") if isinstance(x, dict) else x) == device
                   for x in names):
            failures.append(f"{device} is not in the device list")

        # The hub's other post: the presence, which says who is here.
        # The body is build_presence() in crynet_net.c, byte for byte,
        # the same way hub_body() above is build_batch().
        pres = ('{"device":"%s","hub":{"fw":"%s","name":"%s"},'
                '"nodes":[{"cot":1,"online":true,"deaf":false,"rssi":-58},'
                '{"cot":2,"online":true,"deaf":false,"rssi":-61}]}'
                % (device, FW_VERSION, "Nursery hub"))
        json.loads(pres)                    # as the record body was checked
        ans = post(base + "/api/v1/presence", pres, args.token)
        if ans.get("cots") != 2:
            failures.append(f"the presence post stored {ans.get('cots')} of 2 cots")
        print(f"  posted presence -> {ans}")

        p = get(base + "/api/v1/presence")
        mine = next((h for h in p.get("hubs", []) if h["device"] == device), {})
        cots = {n["cot"]: n for n in p.get("nodes", []) if n["device"] == device}
        if not mine.get("online"):
            failures.append("the hub that just posted reads offline")
        online = sorted(c for c in cots if cots[c]["online"])
        if online != [1, 2]:
            failures.append(f"the cots the hub just reported online read "
                            f"{online}, expected [1, 2]")
        print("  read back -> hub online, cot 1 and cot 2 online with it")

        # Then the power is pulled the only way a test can pull it: the
        # clock is moved past the staleness window.  The hub must read as
        # gone -- and so must its cots, which before D-108 stayed online
        # for ever on the last post the hub ever made.  (Needs the
        # throwaway server; a dashboard we were pointed at is left alone.)
        if tmp is not None:
            conn = sqlite3.connect(str(db), timeout=5)
            with conn:
                conn.execute("UPDATE devices SET last_seen = last_seen - 600")
                conn.execute("UPDATE presence SET at = at - 600")
            conn.close()
            p = get(base + "/api/v1/presence")
            mine = next((h for h in p.get("hubs", []) if h["device"] == device), {})
            cots = {n["cot"]: n for n in p.get("nodes", []) if n["device"] == device}
            if mine.get("online"):
                failures.append("the hub still reads online ten minutes after "
                                "its last word")
```

```
            still = sorted(c for c in cots if cots[c]["online"])
            if still:
                failures.append(f"cots {still} read online long after the hub "
                                f"that sees them stopped speaking (D-108)")
            print("  aged 10 min -> hub offline, and both cots offline with it")

    finally:
        if proc is not None:
            proc.terminate()
            try:
                proc.wait(timeout=10)
            except subprocess.TimeoutExpired:
                proc.kill()
        if tmp is not None:
            tmp.cleanup()

    print()
    if failures:
        for f in failures:
            print(f"FAIL  {f}")
        return 1
    print("OK  the hub's upload body and the dashboard's API agree")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

The dashboard: the offline copy, the page, and the hosted Worker

dashboard/server.py

1024 lines · 40,246 bytes

```
#!/usr/bin/env python3
"""CryNet dashboard — the caregiver's record of the baby's cries.

The hub always writes every cry it hears to its own flash, whether or not
there is a network.  When the hub can reach this service it uploads what
it has; this service stores it, and the web dashboard in ``web/`` turns it
into the record a caregiver actually wants: how many times the baby cried,
in which cot, at what hour of the night, and whether that is changing.

It is deliberately plain: the Python standard library and SQLite, no
framework, no build step.  A caregiver's machine can run it with

    python dashboard/server.py

and that is the whole installation.  It binds every interface, so a phone
on the same network reaches it at http://<this machine's address>:8787.

The API is documented in API.md.  The short version:

    POST /api/v1/events        the hub uploads a batch of cry records
    GET  /api/v1/summary       the numbers and the charts for the dashboard
    GET  /api/v1/analytics     the patterns and the plain-language findings
    GET  /api/v1/events        the raw record, filtered
    GET  /api/v1/devices       the hubs that have reported in
    GET  /api/v1/health        is it alive

Usage:
    python dashboard/server.py
    python dashboard/server.py --port 8787 --db dashboard/data/crynet.db
    python dashboard/server.py --demo      # seed a week of made-up cries
"""

from __future__ import annotations

import argparse
import json
import os
import random
import socket
import sqlite3
import statistics
import threading
import time
from datetime import datetime, timedelta
from http.server import BaseHTTPRequestHandler, ThreadingHTTPServer
```

```
from pathlib import Path
from urllib.parse import parse_qs, urlparse

try:
    from zoneinfo import ZoneInfo
except ImportError:                                 # Python < 3.9
    ZoneInfo = None

HERE = Path(__file__).resolve().parent
WEB = HERE / "web"

# The kinds of record the hub sends.  "cry" is the one that matters; the
# others are the system's own health, which a caregiver wants to see
# beside the cry data (a cot that went deaf explains a silent night).
KINDS = ("cry", "deaf", "heard", "boot", "test")

# The night runs from eight in the evening until six in the morning.  The
# caregiver's question is usually "how many times was the baby up last
# night", so the night is counted as one bucket rather than as two
# calendar days.
NIGHT_START_HOUR = 20
NIGHT_END_HOUR = 6

# Duration buckets for the histogram, in seconds.  A short cry is a
# complaint and a long one is pain; the buckets are chosen around that
# difference rather than at even intervals.
DURATION_BUCKETS = (5, 15, 30, 60, 120, 300, 600)


# ---------------------------------------------------------------------------
# storage
# ---------------------------------------------------------------------------

SCHEMA = """
CREATE TABLE IF NOT EXISTS devices (
    device     TEXT PRIMARY KEY,
    name       TEXT,
    fw         TEXT,
    first_seen INTEGER,
    last_seen  INTEGER
);

CREATE TABLE IF NOT EXISTS events (
    device      TEXT NOT NULL,
    seq         INTEGER NOT NULL,
    kind        TEXT NOT NULL,
    cot         INTEGER,
    start_at    INTEGER,   -- epoch seconds, 0 = the hub did not know the time
    end_at      INTEGER,
    duration_s  INTEGER,
    peak_conf   INTEGER,
    windows     INTEGER,
    peak_rms    REAL,
    mean_rms    REAL,
    boot        INTEGER,
    uptime_ms   INTEGER,
    received_at INTEGER,
    PRIMARY KEY (device, seq)
);

CREATE INDEX IF NOT EXISTS idx_events_start ON events (start_at);
CREATE INDEX IF NOT EXISTS idx_events_kind  ON events (kind);

CREATE TABLE IF NOT EXISTS presence (
    device  TEXT NOT NULL,
    cot     INTEGER NOT NULL,
    online  INTEGER NOT NULL,
    deaf    INTEGER NOT NULL DEFAULT 0,
    rssi    INTEGER,
    at      INTEGER,
    PRIMARY KEY (device, cot)
);
"""


class Store:
    """The SQLite record.  One connection per thread."""

    def __init__(self, path: Path):
        path.parent.mkdir(parents=True, exist_ok=True)
        self.path = path
        self._local = threading.local()
        with self.connect() as db:
            db.executescript(SCHEMA)

```

```
    def connect(self) -> sqlite3.Connection:
        db = getattr(self._local, "db", None)
        if db is None:
            db = sqlite3.connect(self.path, timeout=30)
            db.row_factory = sqlite3.Row
            db.execute("PRAGMA journal_mode=WAL")
            self._local.db = db
        return db

    # -- writing ------------------------------------------------------

    def record(self, device: str, ev: dict, fw: str, name: str) -> bool:
        """Store one event.  Returns True if it was new.

        The hub sends the same record more than once if a reply is lost, so
        the store has to be idempotent: the pair (device, seq) is the key,
        and a record that is already there is simply acknowledged.
        """
        db = self.connect()
        now = int(time.time())
        with db:
            cur = db.execute(
                "INSERT OR IGNORE INTO events (device, seq, kind, cot, start_at,"
                " end_at, duration_s, peak_conf, windows, peak_rms, mean_rms,"
                " boot, uptime_ms, received_at)"
                " VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?)",
                (
                    device,
                    int(ev.get("seq", 0)),
                    str(ev.get("kind", "cry")),
                    _int_or_none(ev.get("cot")),
                    _int_or_none(ev.get("start")),
                    _int_or_none(ev.get("end")),
                    _int_or_none(ev.get("duration_s")),
                    _int_or_none(ev.get("peak_conf")),
                    _int_or_none(ev.get("windows")),
                    _float_or_none(ev.get("peak_rms")),
                    _float_or_none(ev.get("mean_rms")),
                    _int_or_none(ev.get("boot")),
                    _int_or_none(ev.get("uptime_ms")),
                    now,
                ),
            )
            inserted = cur.rowcount > 0
            db.execute(
                "INSERT INTO devices (device, name, fw, first_seen, last_seen)"
                " VALUES (?,?,?,?,?)"
                " ON CONFLICT(device) DO UPDATE SET"
                "   name=COALESCE(excluded.name, devices.name),"
                "   fw=COALESCE(excluded.fw, devices.fw),"
                "   last_seen=excluded.last_seen",
                (device, name, fw, now, now),
            )
        return inserted

    def device_seen(self, device: str, fw: str = "", name: str = "") -> None:
        db = self.connect()
        now = int(time.time())
        with db:
            db.execute(
                "INSERT INTO devices (device, name, fw, first_seen, last_seen)"
                " VALUES (?,?,?,?,?)"
                " ON CONFLICT(device) DO UPDATE SET"
                "   last_seen=excluded.last_seen",
                (device, name or None, fw or None, now, now),
            )

    def presence(self, device: str, nodes: list[dict], fw: str = "",
                 name: str = "") -> int:
        """Say who is here.

        The cots have no network of their own -- they talk to the hub
        over its radio -- so the hub is the only thing that can answer
        for them, and this is the whole of the hub's own report.  One
        row per cot, replaced each time the hub speaks, so what is
        stored is always the last thing the hub knew.

        The hub's own last_seen moves too: a hub that is talking is a
        hub that is there, whatever it has to say.
        """
        db = self.connect()
        now = int(time.time())
        written = 0
        with db:
            db.execute(
```

```
                "INSERT INTO devices (device, name, fw, first_seen, last_seen)"
                " VALUES (?,?,?,?,?)"
                " ON CONFLICT(device) DO UPDATE SET"
                "   name=COALESCE(excluded.name, devices.name),"
                "   fw=COALESCE(excluded.fw, devices.fw),"
                "   last_seen=excluded.last_seen",
                (device, name or None, fw or None, now, now),
            )
            for n in nodes:
                cot = _int_or_none(n.get("cot"))
                if cot is None:
                    continue
                db.execute(
                    "INSERT INTO presence (device, cot, online, deaf, rssi, at)"
                    " VALUES (?,?,?,?,?,?)"
                    " ON CONFLICT(device, cot) DO UPDATE SET"
                    "   online=excluded.online,"
                    "   deaf=excluded.deaf,"
                    "   rssi=excluded.rssi,"
                    "   at=excluded.at",
                    (device, cot,
                     1 if n.get("online") else 0,
                     1 if n.get("deaf") else 0,
                     _int_or_none(n.get("rssi")), now),
                )
                written += 1
        return written

    def presence_rows(self) -> list[dict]:
        rows = self.connect().execute(
            "SELECT * FROM presence ORDER BY device, cot"
        ).fetchall()
        return [dict(r) for r in rows]

    # -- reading ------------------------------------------------------

    def devices(self) -> list[dict]:
        rows = self.connect().execute(
            "SELECT * FROM devices ORDER BY last_seen DESC"
        ).fetchall()
        return [dict(r) for r in rows]

    def events(self, since: int = 0, until: int = 0, cot: int = 0,
               kind: str = "", limit: int = 500) -> list[dict]:
        """The records in a window, including the ones with no time.

        A record with `start_at = 0` is one the hub made before it knew
        the time (D-90).  It is evidence that a cry happened, so it is
        returned and counted; it is simply never placed on a time axis.
        For windowing it falls back to `received_at` -- we cannot know
        when the cry was, but we do know when we learned about it, and
        that is the only honest bound available.
        """
        when = "(CASE WHEN start_at > 0 THEN start_at ELSE received_at END)"
        sql = "SELECT * FROM events WHERE 1=1"
        args: list = []
        if since:
            sql += f" AND {when} >= ?"
            args.append(since)
        if until:
            sql += f" AND {when} <= ?"
            args.append(until)
        if cot:
            sql += " AND cot = ?"
            args.append(cot)
        if kind:
            sql += " AND kind = ?"
            args.append(kind)
        sql += f" ORDER BY {when} DESC, seq DESC LIMIT ?"
        args.append(limit)
        rows = self.connect().execute(sql, args).fetchall()
        return [dict(r) for r in rows]

    def all_cry_events(self, since: int = 0, until: int = 0) -> list[dict]:
        """Every cry, oldest first, for the analytics."""
        sql = "SELECT * FROM events WHERE kind IN ('cry','test')"
        args: list = []
        if since:
            sql += " AND start_at >= ?"
            args.append(since)
        if until:
            sql += " AND start_at <= ?"
            args.append(until)
        sql += " ORDER BY start_at ASC"
        return [dict(r) for r in self.connect().execute(sql, args).fetchall()]
```

```
301
302
303 def _int_or_none(v):
304     if v is None or v == "":
305         return None
306     try:
307         return int(v)
308     except (TypeError, ValueError):
309         return None
310
311
312 def _float_or_none(v):
313     if v is None or v == "":
314         return None
315     try:
316         return float(v)
317     except (TypeError, ValueError):
318         return None
319
320
321 # ---------------------------------------------------------------------------
322 # analytics
323 # ---------------------------------------------------------------------------
324
325 def make_local(tz: str = ""):
326     """Turn epoch seconds into a datetime in the caregiver's timezone.
327
328     The page sends its zone with every request (web/app.js) because the
329     hosted copy of this service runs in UTC and the caregiver does not.
330     An unknown or missing zone falls back to this machine's local time,
331     which is what the service has always done when it ran on the
332     caregiver's own computer.
333     """
334     if tz and ZoneInfo is not None:
335         try:
336             zone = ZoneInfo(tz)
337         except Exception:
338             zone = None
339         if zone is not None:
340             return lambda ts: datetime.fromtimestamp(ts, zone)
341     return datetime.fromtimestamp
342
343
344 def night_key(ts: int, lt=make_local()) -> str:
345     """The night a timestamp belongs to.
346
347     A cry at 02:00 on the 12th happened during the night of the 11th --
348     the evening of the 11th into the morning of the 12th -- so the night
349     is named by the day it started.  That is how a caregiver counts a
350     night, and it is the only way the numbers add up.
351     """
352     d = lt(ts)
353     if d.hour >= NIGHT_START_HOUR:
354         return d.date().isoformat()
355     if d.hour < NIGHT_END_HOUR:
356         return (d - timedelta(days=1)).date().isoformat()
357     return d.date().isoformat()
358
359
360 def local_date(ts: int, lt=make_local()) -> str:
361     return lt(ts).date().isoformat()
362
363
364 # A hub is taken to be there if it has said something in the last
365 # ninety seconds.  The hub says every thirty seconds, so three is three
366 # chances missed before the dashboard calls it gone -- long enough that
367 # one dropped request does not make the caregiver think the hub has
368 # died, short enough that a hub that has been switched off is gone
369 # before anyone has looked away.  The owner first asked for the
370 # connection to be checked "as often as possible" and then set the
371 # constraint that actually decides it: nothing may cost the cots
372 # anything, and they share the hub's radio.  So the hub posts half as
373 # often as the fastest figure considered (30 s, not 15 s), and the
374 # service's window is three of those.  The chain -- the hub speaks
375 # every 30 s, this waits 90 s, the page asks every 10 s -- means a hub
376 # that is switched off reads as gone in about two minutes.
377 PRESENCE_STALE_S = 90
378
379
380 def build_presence(store: Store) -> dict:
381     """Who is here: the hubs, and the cots each hub can see.
382
383     This is separate from the cry record on purpose.  A record says
384     what happened; this says whether the thing that watches is working.
385     A dashboard that only learns of a hub when a baby cries cannot tell
```

```
    the caregiver whether a quiet week means a quiet baby or a dead hub,
    and that is the question a person actually opens the dashboard to
    ask.
    """
    now = int(time.time())
    hubs = []
    demo = set()
    for d in store.devices():
        seen = int(d.get("last_seen") or 0)
        is_demo = (d.get("fw") or "") == "demo"
        if is_demo:
            demo.add(d.get("device"))
        hubs.append({
            "device": d.get("device"),
            "name": d.get("name"),
            "fw": d.get("fw"),
            "at": seen,
            "age_s": (now - seen) if seen else None,
            # The demo hub is not a thing that can go offline -- it is a
            # made-up hub with made-up records and nothing posting it.
            # Holding it offline would show the panel working exactly
            # when it is not, which is the one thing a demo must not do.
            "online": is_demo or (bool(seen) and (now - seen) <= PRESENCE_STALE_S),
            "demo": is_demo,
        })
    nodes = []
    for p in store.presence_rows():
        is_demo = p.get("device") in demo
        # A cot is online only while the hub that sees it still is: the
        # row is the hub's last word, and `at` says when it was said.  On
        # the same clock the hub itself is judged by, a row the hub has
        # stopped refreshing goes stale -- otherwise the post a hub made
        # just before a power cut would hold its cots "online" for ever
        # (D-108).
        at = int(p.get("at") or 0)
        fresh = at > 0 and (now - at) <= PRESENCE_STALE_S
        nodes.append({
            "device": p.get("device"),
            "cot": p.get("cot"),
            "online": is_demo or (bool(p.get("online")) and fresh),
            "deaf": bool(p.get("deaf")),
            "rssi": p.get("rssi"),
            "at": p.get("at"),
        })
    return {"now": now, "hubs": hubs, "nodes": nodes}


def build_summary(store: Store, days: int, cot: int = 0, tz: str = "") -> dict:
    """The numbers and the series the dashboard draws."""
    lt = make_local(tz)
    now = int(time.time())
    since = now - days * 86400
    events = store.events(since=since, until=now + 86400, cot=cot,
                          kind="cry", limit=100000)
    events = [e for e in events if e["kind"] in ("cry", "test")]
    events.sort(key=lambda e: e["start_at"] or 0)

    durations = [int(e["duration_s"] or 0) for e in events]
    total_s = sum(durations)

    per_cot: dict[int, dict] = {}
    per_day: dict[str, dict] = {}
    per_hour = [0] * 24
    per_hour_s = [0] * 24
    per_weekday = [0] * 7
    nights: dict[str, dict] = {}
    heat_counts: dict[int, list[int]] = {}
    heat_seconds: dict[int, list[int]] = {}

    untimed = 0
    for e in events:
        ts = int(e["start_at"] or 0)
        c = int(e["cot"] or 0)
        dur = int(e["duration_s"] or 0)

        # Which cot cried, how often and how loudly are facts whether or
        # not the hub knew the date, so they are counted for every
        # record.  Only `last_at` needs a time.
        bucket = per_cot.setdefault(c, {"cot": c, "cries": 0, "seconds": 0,
                                        "max_seconds": 0, "last_at": 0,
                                        "confidences": []})
        bucket["cries"] += 1
        bucket["seconds"] += dur
        bucket["max_seconds"] = max(bucket["max_seconds"], dur)
        if e["peak_conf"] is not None:
```

```
            bucket["confidences"].append(int(e["peak_conf"]))
        if not ts:
            untimed += 1
            continue
        bucket["last_at"] = max(bucket["last_at"], ts)
        local = lt(ts)

        day = per_day.setdefault(local_date(ts, lt), {"day": local_date(ts, lt),
                                                      "cries": 0, "seconds": 0,
                                                      "cots": set()})
        day["cries"] += 1
        day["seconds"] += dur
        if c:
            day["cots"].add(c)

        per_hour[local.hour] += 1
        per_hour_s[local.hour] += dur
        per_weekday[local.weekday()] += 1

        nk = night_key(ts, lt)
        night = nights.setdefault(nk, {"night": nk, "cries": 0, "seconds": 0,
                                       "cots": set()})
        night["cries"] += 1
        night["seconds"] += dur
        if c:
            night["cots"].add(c)

        if c:
            hc = heat_counts.setdefault(c, [0] * 24)
            hs = heat_seconds.setdefault(c, [0] * 24)
            hc[local.hour] += 1
            hs[local.hour] += dur

    for b in per_day.values():
        b["cots"] = sorted(b["cots"])
    for b in nights.values():
        b["cots"] = sorted(b["cots"])

    night_list = sorted(nights.values(), key=lambda n: n["night"])
    gaps = []
    for a, b in zip(events, events[1:]):
        if a["start_at"] and b["start_at"]:
            gaps.append(b["start_at"] - a["start_at"])

    cots_sorted = sorted(per_cot.values(), key=lambda b: b["cot"])
    for b in cots_sorted:
        confs = b.pop("confidences")
        b["avg_conf"] = round(sum(confs) / len(confs), 1) if confs else None

    heat_cots = sorted(heat_counts.keys())
    devices = store.devices()

    return {
        "device": devices[0] if devices else None,
        "devices": devices,
        "range": {"from": since, "to": now, "days": days},
        "kpis": {
            "total_cries": len(events),
            "timed_cries": len(events) - untimed,
            "untimed_cries": untimed,
            "cries_per_day": round(len(events) / days, 2) if days else 0,
            "total_cry_seconds": total_s,
            "avg_cry_seconds": round(total_s / len(events), 1) if events else 0,
            "longest_cry_seconds": max(durations) if durations else 0,
            "median_cry_seconds": round(statistics.median(durations), 1)
            if durations else 0,
            "cots_crying": len(per_cot),
            "nights_affected": len(nights),
            "nights_per_week": round(len(nights) / (days / 7.0), 2)
            if days >= 7 else 0,
            # A per-night average must divide like by like: a record with
            # no time cannot be attributed to a night.
            "avg_cries_per_night": round((len(events) - untimed) / len(nights), 2)
            if nights else 0,
            "last_cry_at": max((int(e["start_at"] or 0) for e in events),
                               default=0),
            "median_gap_min": round(statistics.median(gaps) / 60.0, 1)
            if gaps else None,
        },
        "per_cot": cots_sorted,
        "per_day": [per_day[k] for k in sorted(per_day)],
        "per_night": night_list,
        "per_hour": [{"hour": h, "cries": per_hour[h], "seconds": per_hour_s[h]}
                     for h in range(24)],
        "per_weekday": [{"weekday": w, "cries": per_weekday[w]}
```

```
                        for w in range(7)],
        "heatmap": {
            "cots": heat_cots,
            "counts": [heat_counts[c] for c in heat_cots],
            "seconds": [heat_seconds[c] for c in heat_cots],
        },
        "recent": [
            {
                "seq": e["seq"], "cot": e["cot"], "kind": e["kind"],
                "start": e["start_at"], "end": e["end_at"],
                "duration_s": e["duration_s"], "peak_conf": e["peak_conf"],
                "windows": e["windows"], "peak_rms": e["peak_rms"],
                "mean_rms": e["mean_rms"],
            }
            for e in events[-400:]
        ],
    }


def build_analytics(store: Store, days: int, tz: str = "") -> dict:
    """The patterns: distributions, gaps, nights, and what they mean."""
    lt = make_local(tz)
    now = int(time.time())
    since = now - days * 86400
    events = [e for e in store.all_cry_events(since=since, until=now + 86400)
              if e["start_at"]]
    untimed = sum(
        1 for e in store.events(since=since, until=now + 86400, kind="cry",
                                limit=100000)
        if not e["start_at"])

    durations = [int(e["duration_s"] or 0) for e in events]
    confs = [int(e["peak_conf"]) for e in events if e["peak_conf"] is not None]

    hist = [{"label": _bucket_label(lo, hi), "count": 0} for lo, hi in
            _bucket_pairs(DURATION_BUCKETS)]
    for d in durations:
        for i, (lo, hi) in enumerate(_bucket_pairs(DURATION_BUCKETS)):
            if lo <= d < hi:
                hist[i]["count"] += 1
                break

    gaps = []
    for a, b in zip(events, events[1:]):
        gaps.append(b["start_at"] - a["start_at"])

    nights: dict[str, list[dict]] = {}
    for e in events:
        nights.setdefault(night_key(int(e["start_at"]), lt), []).append(e)

    # The hour each cot is most often heard in, and how sharp that peak
    # is -- a caregiver wants to know "cot 2 always wakes at 03:00".
    per_cot = []
    for e_cot in sorted({int(e["cot"] or 0) for e in events}):
        sub = [e for e in events if int(e["cot"] or 0) == e_cot]
        by_hour = [0] * 24
        for e in sub:
            by_hour[lt(int(e["start_at"])).hour] += 1
        peak_hour = max(range(24), key=lambda h: by_hour[h])
        total = sum(by_hour)
        share = round(100.0 * by_hour[peak_hour] / total, 1) if total else 0
        secs = sum(int(e["duration_s"] or 0) for e in sub)
        per_cot.append({
            "cot": e_cot,
            "cries": len(sub),
            "seconds": secs,
            "avg_seconds": round(secs / len(sub), 1) if sub else 0,
            "max_seconds": max(int(e["duration_s"] or 0) for e in sub) if sub else 0,
            "median_seconds": round(statistics.median(
                [int(e["duration_s"] or 0) for e in sub]), 1) if sub else 0,
            "by_hour": by_hour,
            "peak_hour": peak_hour,
            "peak_share": share,
            "night_cries": sum(1 for e in sub
                               if lt(int(e["start_at"])).hour
                               >= NIGHT_START_HOUR
                               or lt(int(e["start_at"])).hour
                               < NIGHT_END_HOUR),
        })

    return {
        "range": {"from": since, "to": now, "days": days},
        "duration_hist": hist,
        "confidence": {
            "min": min(confs) if confs else None,
```

```
            "median": round(statistics.median(confs), 1) if confs else None,
            "max": max(confs) if confs else None,
        },
        "gaps": {
            "count": len(gaps),
            "median_min": round(statistics.median(gaps) / 60.0, 1) if gaps else None,
            "max_min": round(max(gaps) / 60.0, 1) if gaps else None,
            "min_min": round(min(gaps) / 60.0, 1) if gaps else None,
        },
        "nights": [
            {
                "night": k,
                "cries": len(v),
                "seconds": sum(int(e["duration_s"] or 0) for e in v),
                "cots": sorted({int(e["cot"] or 0) for e in v}),
                "first": min(int(e["start_at"]) for e in v),
                "last": max(int(e["start_at"]) for e in v),
            }
            for k, v in sorted(nights.items())
        ],
        "per_cot": per_cot,
        # Every pattern on this page is a statement about *when*.  A
        # record with no time cannot take part in one, so it is counted
        # and reported rather than quietly folded in.
        "untimed_cries": untimed,
        "insights": build_insights(events, per_cot, nights, gaps),
    }


def _bucket_pairs(edges):
    pairs = []
    lo = 0
    for hi in edges:
        pairs.append((lo, hi))
        lo = hi
    pairs.append((lo, 10 ** 9))
    return pairs


def _bucket_label(lo, hi):
    if hi >= 10 ** 9:
        return f"{lo / 60:.0f}+ min"
    return f"{lo}-{hi}s" if hi - lo <= 60 else f"{lo // 60}-{hi // 60} min"


def build_insights(events, per_cot, nights, gaps) -> list[str]:
    """The plain-language findings a caregiver can act on.

    These are read by a person, not a machine, so they are written as
    sentences and they only say what the data actually shows.
    """
    out: list[str] = []
    if not events:
        return ["No cries have been recorded yet."]

    total = len(events)
    out.append(f"{total} cry episode{'s' if total != 1 else ''} recorded.")

    for c in per_cot:
        if c["cries"] < 3:
            continue
        hour = c["peak_hour"]
        if c["peak_share"] >= 40:
            out.append(
                f"Cot {c['cot']} is usually heard around {hour:02d}:00 "
                f"({c['peak_share']}% of its cries fall in that hour)."
            )
        if c["night_cries"] / c["cries"] > 0.6:
            out.append(
                f"Cot {c['cot']} cries are mostly at night "
                f"({c['night_cries']} of {c['cries']})."
            )

    if nights:
        counts = [len(v) for v in nights.values()]
        avg = sum(counts) / len(counts)
        out.append(
            f"On average {avg:.1f} cry episodes per night across "
            f"{len(nights)} night{'s' if len(nights) != 1 else ''}."
        )
        worst = max(nights.items(), key=lambda kv: len(kv[1]))
        if len(counts) > 1 and len(worst[1]) > avg * 1.5:
            out.append(
                f"The night of {worst[0]} was the heaviest, with "
                f"{len(worst[1])} episodes."
```

```
            )

    if gaps:
        med = statistics.median(gaps) / 60.0
        if med < 30:
            out.append(
                f"Cries come roughly every {med:.0f} minutes on average, "
                f"which is a dense pattern."
            )
        elif med > 240:
            out.append(
                f"Cries are well spread out -- about every {med / 60:.1f} hours."
            )

    durs = [int(e["duration_s"] or 0) for e in events]
    if durs:
        med_d = statistics.median(durs)
        if med_d >= 60:
            out.append(
                f"The median cry lasts {med_d / 60:.1f} minutes, which is "
                f"long for settling complaints."
            )
        elif med_d <= 15:
            out.append(
                f"The median cry lasts {med_d:.0f} seconds -- short, "
                f"settling sounds."
            )
    return out


# ---------------------------------------------------------------------------
# the demo data
# ---------------------------------------------------------------------------

def seed_demo(store: Store, days: int = 7) -> None:
    """A week of invented cries, so the dashboard can be seen working
    before a hub has ever uploaded anything."""
    now = datetime.now()
    rng = random.Random(20261123)
    device = "demo-hub"
    store.device_seen(device, fw="demo", name="Demo hub")
    seq = 0
    for day in range(days, 0, -1):
        d = now - timedelta(days=day)
        # More cries at night than in the day: that is what a baby's
        # record looks like, and it is what the dashboard is for.
        for cot, n in ((1, rng.randint(0, 3)), (2, rng.randint(1, 5))):
            for _ in range(n):
                seq += 1
                hour = rng.choice([2, 3, 4, 13, 14, 18, 19, 20, 21])
                minute = rng.randint(0, 59)
                start = int(d.replace(hour=hour, minute=minute,
                                      second=0, microsecond=0).timestamp())
                dur = rng.randint(8, 150)
                store.record(device, {
                    "seq": seq, "kind": "cry", "cot": cot,
                    "start": start, "end": start + dur,
                    "duration_s": dur,
                    "peak_conf": rng.randint(94, 100),
                    "windows": rng.randint(1, 6),
                    "peak_rms": round(rng.uniform(0.01, 0.22), 4),
                    "mean_rms": round(rng.uniform(0.005, 0.10), 4),
                    "boot": 1, "uptime_ms": 0,
                }, fw="demo", name="Demo hub")

    # The demo hub also says who is here, so the presence panel has
    # something to draw.  Both cots are answering: one of them well,
    # one of them on the edge, which is what a real room looks like.
    store.presence(device, [
        {"cot": 1, "online": True, "deaf": False, "rssi": -47},
        {"cot": 2, "online": True, "deaf": False, "rssi": -63},
    ], fw="demo", name="Demo hub")


# ---------------------------------------------------------------------------
# the HTTP service
# ---------------------------------------------------------------------------

class Handler(BaseHTTPRequestHandler):
    server_version = "CryNetDashboard/1.0"
    store: Store
    token: str = ""

    def log_message(self, fmt, *args):
        print(f"[dashboard] {self.address_string()} {fmt % args}")
```

```
811
812     # -- helpers ------------------------------------------------------------
813
814     def _send_json(self, obj, status: int = 200):
815         body = json.dumps(obj).encode("utf-8")
816         self.send_response(status)
817         self.send_header("Content-Type", "application/json")
818         self.send_header("Content-Length", str(len(body)))
819         self.send_header("Cache-Control", "no-store")
820         self.end_headers()
821         self.wfile.write(body)
822
823     def _send_file(self, path: Path, ctype: str):
824         if not path.is_file():
825             self._send_json({"error": "not found"}, 404)
826             return
827         body = path.read_bytes()
828         self.send_response(200)
829         self.send_header("Content-Type", ctype)
830         self.send_header("Content-Length", str(len(body)))
831         self.send_header("Cache-Control", "no-cache")
832         self.end_headers()
833         self.wfile.write(body)
834
835     def _authorised(self) -> bool:
836         if not self.token:
837             return True
838         got = self.headers.get("X-CryNet-Token", "")
839         return got == self.token
840
841     # -- routes -------------------------------------------------------------
842
843     def do_GET(self):
844         route = urlparse(self.path)
845         query = parse_qs(route.query)
846         path = route.path
847
848         if path in ("/", "/index.html"):
849             return self._send_file(WEB / "index.html", "text/html; charset=utf-8")
850         if path == "/app.js":
851             return self._send_file(WEB / "app.js",
852                                    "application/javascript; charset=utf-8")
853         if path == "/style.css":
854             return self._send_file(WEB / "style.css", "text/css; charset=utf-8")
855
856         if path == "/api/v1/health":
857             return self._send_json({
858                 "ok": True,
859                 "time": int(time.time()),
860                 "devices": len(self.store.devices()),
861             })
862
863         if path == "/api/v1/devices":
864             return self._send_json({"devices": self.store.devices()})
865
866         if path == "/api/v1/presence":
867             return self._send_json(build_presence(self.store))
868
869         if path == "/api/v1/summary":
870             days = int(_first(query, "days", 14))
871             cot = int(_first(query, "cot", 0) or 0)
872             tz = _first(query, "tz", "") or ""
873             return self._send_json(build_summary(self.store, days, cot, tz))
874
875         if path == "/api/v1/analytics":
876             days = int(_first(query, "days", 30))
877             tz = _first(query, "tz", "") or ""
878             return self._send_json(build_analytics(self.store, days, tz))
879
880         if path == "/api/v1/events":
881             limit = min(int(_first(query, "limit", 200)), 2000)
882             since = int(_first(query, "from", 0) or 0)
883             until = int(_first(query, "to", 0) or 0)
884             cot = int(_first(query, "cot", 0) or 0)
885             kind = _first(query, "kind", "") or ""
886             rows = self.store.events(since=since, until=until, cot=cot,
887                                      kind=kind, limit=limit)
888             return self._send_json({"events": rows})
889
890         return self._send_json({"error": "not found"}, 404)
891
892     def do_POST(self):
893         route = urlparse(self.path)
894         path = route.path
895         length = int(self.headers.get("Content-Length", "0") or 0)
```

```
        raw = self.rfile.read(length) if length else b"{}"
        try:
            body = json.loads(raw.decode("utf-8") or "{}")
        except (UnicodeDecodeError, json.JSONDecodeError):
            return self._send_json({"error": "bad json"}, 400)

        if path == "/api/v1/events":
            if not self._authorised():
                return self._send_json({"error": "unauthorised"}, 401)
            device = str(body.get("device") or "unknown")
            hub = body.get("hub") or {}
            self.store.device_seen(device, str(hub.get("fw", "")),
                                   str(hub.get("name", "")))
            events = body.get("events") or []
            accepted = 0
            for ev in events:
                if self.store.record(device, ev, str(hub.get("fw", "")),
                                     str(hub.get("name", ""))):
                    accepted += 1
            print(f"[dashboard] {device}: {len(events)} record(s), "
                  f"{accepted} new")
            return self._send_json({
                "accepted": accepted,
                "received": len(events),
                "server_time": int(time.time()),
            })

        if path == "/api/v1/presence":
            if not self._authorised():
                return self._send_json({"error": "unauthorised"}, 401)
            device = str(body.get("device") or "unknown")
            hub = body.get("hub") or {}
            nodes = body.get("nodes") or []
            if not isinstance(nodes, list):
                nodes = []
            written = self.store.presence(device, nodes,
                                          str(hub.get("fw", "")),
                                          str(hub.get("name", "")))
            print(f"[dashboard] {device}: presence, {written} cot(s)")
            return self._send_json({
                "ok": True,
                "cots": written,
                "server_time": int(time.time()),
            })

        if path == "/api/v1/admin/seed":
            if not self._authorised():
                return self._send_json({"error": "unauthorised"}, 401)
            seed_demo(self.store)
            return self._send_json({"ok": True})

        return self._send_json({"error": "not found"}, 404)


def _first(query: dict, key: str, default):
    v = query.get(key)
    if not v:
        return default
    try:
        return v[0]
    except (IndexError, TypeError):
        return default


def local_addresses() -> list[str]:
    out = []
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect(("8.8.8.8", 80))
        out.append(s.getsockname()[0])
        s.close()
    except OSError:
        pass
    return out


def main() -> int:
    ap = argparse.ArgumentParser(description="the CryNet caregiver dashboard")
    ap.add_argument("--port", type=int, default=8787)
    ap.add_argument("--db", default=str(HERE / "data" / "crynet.db"))
    ap.add_argument("--host", default="0.0.0.0")
    ap.add_argument("--token", default=os.environ.get("CRYNET_TOKEN", ""),
                    help="shared secret the hub must send (X-CryNet-Token)")
    ap.add_argument("--demo", action="store_true",
                    help="seed a week of invented cries and exit")
```

```
    args = ap.parse_args()

    store = Store(Path(args.db))
    if args.demo:
        seed_demo(store)
        print(f"[dashboard] demo data written to {args.db}")
        return 0

    Handler.store = store
    Handler.token = args.token
    httpd = ThreadingHTTPServer((args.host, args.port), Handler)
    httpd.daemon_threads = True

    addrs = local_addresses()
    shown = addrs[0] if addrs else "localhost"
    print(f"[dashboard] serving on http://{args.host}:{args.port}")
    print(f"[dashboard]   the caregiver's browser: http://{shown}:{args.port}")
    print("[dashboard]")
    print("[dashboard] To point the hub here, type this on the hub's own")
    print("[dashboard] serial console (the same port its log goes to):")
    print("[dashboard]")
    for a in addrs or ["<this machine's address>"]:
        print(f"[dashboard]     net set http://{a}:{args.port}/api/v1/events")
    if args.token:
        print(f"[dashboard]     net token {args.token}")
        print("[dashboard]   (a token is required on every upload)")
    print("[dashboard]")
    print("[dashboard] The hub already knows the Wi-Fi it was built with.")
    print("[dashboard] To change it:  net ssid <name> <password>")
    print("[dashboard] The router's 2.4 GHz channel MUST be 6 -- the cots")
    print("[dashboard] talk on channel 6 and the hub has only one radio.")
    print("[dashboard] Check with:    net        Send now with:  upload")
    print("[dashboard] Nothing above changes how the hub answers a cry.")
    print("[dashboard] With no network it records to its own flash, as ever.")
    print("[dashboard] Ctrl-C to stop")
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        print("\n[dashboard] stopped")
    return 0


if __name__ == "__main__":
    raise SystemExit(main())
```

dashboard/web/index.html

124 lines · 5,865 bytes

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1">
<meta name="color-scheme" content="light dark">
<title>Amazing's CryNet</title>
<link rel="stylesheet" href="style.css">
<script>
  /* Stamp the theme before first paint: the saved choice, else the OS
   * setting.  app.js keeps it in step afterwards. */
  (function () {
    var t = null;
    try { t = localStorage.getItem('crynet-theme'); } catch (e) { /* private mode */ }
    if (t !== 'light' && t !== 'dark') {
      t = (window.matchMedia && window.matchMedia('(prefers-color-scheme: dark)').matches)
        ? 'dark' : 'light';
    }
    document.documentElement.setAttribute('data-theme', t);
  })();
</script>
</head>
<body>
<div class="app">

  <header class="topbar">
    <div class="brand"><span class="mark" aria-hidden="true"></span>Amazing's CryNet</div>
    <div class="hubmeta" id="hubmeta" data-state="none">
      <span class="hubmeta-dot" aria-hidden="true"></span><span id="hubmeta-text"></span>
    </div>
    <div class="conn" id="conn" data-state="loading" role="status" aria-live="polite">
      <span class="dot" aria-hidden="true"></span><span id="conn-text">Connecting</span>
    </div>
```

```
    <button class="theme-toggle" id="theme" type="button" aria-label="Switch to dark theme">
      <svg class="ico ico-moon" viewBox="0 0 16 16" fill="none" stroke="currentColor" stroke-width="1.5" stroke-linecap="round"
stroke-linejoin="round" aria-hidden="true">
        <path d="M13.4 9.6A5.6 5.6 0 0 1 6.4 2.6a5.6 5.6 0 1 0 7 7z"/>
      </svg>
      <svg class="ico ico-sun" viewBox="0 0 16 16" fill="none" stroke="currentColor" stroke-width="1.5" stroke-linecap="round"
stroke-linejoin="round" aria-hidden="true">
        <circle cx="8" cy="8" r="3.1"/>
        <path d="M8 1.4v1.7M8 12.9v1.7M1.4 8h1.7M12.9 8h1.7M3.3 3.3l1.2 1.2M11.5 11.5l1.2 1.2M12.7 3.3l-1.2 1.2M4.5 11.5l-1.2
1.2"/>
      </svg>
    </button>
  </header>

  <nav class="tabs" role="tablist" aria-label="Views">
    <button class="tab" id="tab-overview" role="tab" aria-selected="true" aria-controls="view-overview"
type="button">Overview</button>
    <button class="tab" id="tab-patterns" role="tab" aria-selected="false" aria-controls="view-patterns"
type="button">Patterns</button>
  </nav>

  <div class="controls">
    <div class="control">
      <span class="control-label" id="range-label">Range</span>
      <div class="segmented" id="range" role="group" aria-labelledby="range-label">
        <button type="button" data-days="7" aria-pressed="false">7 days</button>
        <button type="button" data-days="14" aria-pressed="true">14 days</button>
        <button type="button" data-days="30" aria-pressed="false">30 days</button>
        <button type="button" data-days="90" aria-pressed="false">90 days</button>
      </div>
    </div>
    <div class="control">
      <label for="cot">Cot</label>
      <select id="cot"><option value="0">All cots</option></select>
    </div>
    <div class="control" id="device-control" hidden>
      <label for="device">Hub</label>
      <select id="device"><option value="">All hubs</option></select>
    </div>
  </div>

  <p class="scope-note" id="scope-note" hidden></p>

  <main>
    <section id="view-overview" role="tabpanel" aria-labelledby="tab-overview" tabindex="-1">
      <!-- Who is here.  This strip is deliberately outside #overview-body:
           the body is hidden when there is nothing to show, and "is the
           watcher alive" is the question you ask exactly when there is
           nothing to show. -->
      <div class="presence-strip" id="presence-strip" hidden></div>
      <div class="empty" id="empty-overview" hidden>
        <svg width="72" height="56" viewBox="0 0 72 56" aria-hidden="true">
          <rect x="8" y="22" width="56" height="24" rx="6" fill="none" class="art-line" stroke-width="2"/>
          <path d="M18 22V14M30 22V14M42 22V14M54 22V14" class="art-soft" stroke-width="2" stroke-linecap="round"/>
          <path d="M14 46v6M58 46v6" class="art-line" stroke-width="2" stroke-linecap="round"/>
          <circle cx="36" cy="34" r="4" class="art-dot"/>
        </svg>
        <h2>No records yet</h2>
        <p id="empty-overview-detail">The hub has not reached the dashboard. It keeps every cry in its own flash and uploads it
when it can.</p>
      </div>
      <div id="overview-body" hidden>
        <div class="kpis" id="kpis"></div>
        <p class="card-note" id="untimed-note" hidden></p>
        <div class="system-strip" id="system-strip" hidden></div>
        <div class="grid" id="overview-grid"></div>
      </div>
    </section>

    <section id="view-patterns" role="tabpanel" aria-labelledby="tab-patterns" tabindex="-1" hidden>
      <div class="empty" id="empty-patterns" hidden>
        <svg width="72" height="56" viewBox="0 0 72 56" aria-hidden="true">
          <path d="M10 42h52" class="art-line" stroke-width="2" stroke-linecap="round"/>
          <path d="M18 42V30M30 42V34M42 42V26M54 42V36" class="art-soft" stroke-width="4" stroke-linecap="round"/>
        </svg>
        <h2>No patterns yet</h2>
        <p>Durations, nights and gaps appear once the hub has uploaded a few episodes.</p>
      </div>
      <div id="patterns-body" hidden>
        <p class="scope-note" id="patterns-scope" hidden></p>
        <div class="grid" id="patterns-grid"></div>
      </div>
    </section>
  </main>

```

```
  <footer class="footer">
    <span id="footer-range"></span>
    <span id="footer-updated"></span>
    <span>Times are shown in this device's own time zone.</span>
  </footer>
</div>

<div class="tooltip" id="tooltip" role="tooltip" hidden></div>

<script src="app.js"></script>
</body>
</html>
```

dashboard/web/style.css

825 lines · 21,469 bytes

```
/* Amazing's CryNet — the caregiver's record.
 *
 * Two themes, one identity: the teal of the hub's own glass.  Light mode
 * keeps the warm nursery surface; dark mode is the hub panel's near-black
 * teal, its exact values from hub_display.c (C_BG, C_SURFACE, C_SURFACE_HI,
 * C_TEXT, C_DIM, C_ACCENT).  The page is deliberately quiet so the data is
 * the only loud thing on it.
 *
 * Colour roles come from the validated reference palette.  Slot 1 and the
 * sequential ramp are teal — the hub's accent hue — in place of the
 * reference blue; the other seven slots are unchanged.  Both modes were
 * re-run through the palette validator against their own surfaces:
 *   light (#fcfcfb): lightness PASS · chroma PASS · adjacent CVD PASS
 *     (worst 9.1) · normal-vision PASS (worst 19.6) · contrast WARN on
 *     aqua/yellow/magenta -> the relief channel is shipped: every chart
 *     has a table twin and every cot mark carries a written "Cot n" label.
 *   dark (#070f10): all PASS — worst adjacent CVD 8.4, normal-vision 19.3.
 * Dark slot 1 is #23a893, the nearest passing step to the hub accent
 * #2dd4bf, which is too light for the data band; the hub accent itself is
 * kept for chrome (brand mark, tab underline, focus ring), where the band
 * does not apply.  The theme is stamped on <html data-theme> before first
 * paint by the inline script in index.html, so there is no flash.
 */

:root {
  --radius: 14px;
  --radius-sm: 9px;
  --gap: 16px;
  --sans: system-ui, -apple-system, "Segoe UI", Roboto, sans-serif;
}

:root,
[data-theme="light"] {
  color-scheme: light;

  /* surfaces & ink */
  --plane: #f9f9f7;
  --surface-1: #fcfcfb;
  --surface-2: #f0efec;
  --text-primary: #0b0b0b;
  --text-secondary: #52514e;
  --text-muted: #898781;
  --grid: #e1e0d9;
  --axis: #c3c2b7;
  --border: rgba(11, 11, 11, 0.10);
  --border-strong: rgba(11, 11, 11, 0.18);
  --wash: rgba(11, 11, 11, 0.045);

  /* categorical slots — fixed order, assigned by cot, never re-ranked */
  --series-1: #0d9488;
  --series-2: #eb6834;
  --series-3: #1baf7a;
  --series-4: #eda100;
  --series-5: #e87ba4;
  --series-6: #008300;
  --series-7: #6250d6;
  --series-8: #e34948;
  --series-other: #898781;

  /* sequential ramp — one hue (teal), light -> dark (magnitude only) */
  --seq-100: #bce1de;
  --seq-150: #a4d6d2;
  --seq-200: #8bcbc7;
  --seq-250: #73c0bb;
  --seq-300: #5bb4af;
```

```
  --seq-350: #43a9a3;
  --seq-400: #2b9e97;
  --seq-450: #0d9488;
  --seq-500: #0b7e74;
  --seq-550: #09685f;

  /* status — reserved, always shipped with an icon and a word */
  --good: #0ca30c;
  --warning: #fab219;
  --serious: #ec835a;
  --critical: #d03b3b;

  /* chrome accent — the hub's teal, a touch deeper on the light surface */
  --accent: #0d9488;
}

[data-theme="dark"] {
  color-scheme: dark;

  /* the hub panel's own values (hub_display.c) */
  --plane: #020404;          /* C_BG        (2, 4, 4)   */
  --surface-1: #070f10;      /* C_SURFACE   (7, 15, 16) */
  --surface-2: #0c1a1c;      /* C_SURFACE_HI (12, 26, 28) */
  --text-primary: #e4f0ec;   /* C_TEXT      (228, 240, 236) */
  --text-secondary: #7a9692; /* C_DIM       (122, 150, 146) */
  --text-muted: #5f7a76;
  --grid: #182626;
  --axis: #344a48;
  --border: rgba(228, 240, 236, 0.12);
  --border-strong: rgba(228, 240, 236, 0.22);
  --wash: rgba(228, 240, 236, 0.05);

  /* the same eight hues, stepped for the dark surface (validated) */
  --series-1: #23a893;
  --series-2: #d95926;
  --series-3: #199e70;
  --series-4: #c98500;
  --series-5: #d55181;
  --series-6: #008300;
  --series-7: #9085e9;
  --series-8: #e66767;
  --series-other: #7a9692;

  /* sequential ramp — one hue (teal), dim -> bright on the dark ground;
   * the top step is the hub accent itself */
  --seq-100: #0f2f2c;
  --seq-150: #12413c;
  --seq-200: #16544d;
  --seq-250: #19665d;
  --seq-300: #1c786d;
  --seq-350: #208a7e;
  --seq-400: #239d8e;
  --seq-450: #26af9e;
  --seq-500: #2ac1af;
  --seq-550: #2dd4bf;

  /* status — the same four steps; all clear 3:1 on this darker surface */
  --good: #0ca30c;
  --warning: #fab219;
  --serious: #ec835a;
  --critical: #d03b3b;

  /* chrome accent — the hub accent, exactly (C_ACCENT 45, 212, 191) */
  --accent: #2dd4bf;
}

* { box-sizing: border-box; }

/* The `hidden` attribute has to beat every `display` set below it.
 * A class rule like `.control { display: flex }` outranks the browser's
 * own `[hidden] { display: none }`, so anything toggled with `hidden`
 * would stay on the page.  This one line makes `hidden` mean hidden
 * everywhere. */
[hidden] { display: none !important; }

html, body {
  margin: 0;
  padding: 0;
  background: var(--plane);
  color: var(--text-primary);
  font-family: var(--sans);
  font-size: 15px;
  line-height: 1.5;
  -webkit-text-size-adjust: 100%;
}
```

```
.app {
  max-width: 1180px;
  margin: 0 auto;
  padding: 22px 20px 56px;
}

/* ---------------------------------------------------------------- header */

.topbar {
  display: flex;
  flex-wrap: wrap;
  align-items: baseline;
  gap: 10px 16px;
  margin-bottom: 18px;
}

.brand {
  display: flex;
  align-items: baseline;
  gap: 9px;
  font-size: 19px;
  font-weight: 650;
  letter-spacing: -0.01em;
}

.brand .mark {
  width: 10px;
  height: 10px;
  border-radius: 50%;
  background: var(--accent);
  align-self: center;
  flex: none;
}

.brand .sub {
  font-size: 14px;
  font-weight: 400;
  color: var(--text-secondary);
}

.hubmeta {
  display: inline-flex;
  align-items: center;
  gap: 8px;
  color: var(--text-secondary);
  font-size: 13px;
  flex: 1 1 auto;
  min-width: 0;
}

/* The hub's own state: green while it is talking to the dashboard,
 * amber once it has gone quiet.  The word beside it carries the state
 * too, so nothing here depends on the colour being seen. */
.hubmeta-dot {
  width: 8px;
  height: 8px;
  border-radius: 50%;
  background: var(--text-muted);
  flex: none;
}

.hubmeta[data-state="on"] .hubmeta-dot { background: var(--good); }
.hubmeta[data-state="off"] .hubmeta-dot { background: var(--warning); }

.conn {
  display: inline-flex;
  align-items: center;
  gap: 7px;
  font-size: 12.5px;
  color: var(--text-secondary);
  background: var(--surface-1);
  border: 1px solid var(--border);
  border-radius: 999px;
  padding: 4px 11px 4px 9px;
  white-space: nowrap;
}

.conn .dot {
  width: 8px;
  height: 8px;
  border-radius: 50%;
  background: var(--good);
  flex: none;
}
```

```
.conn[data-state="stale"] .dot { background: var(--warning); }
.conn[data-state="loading"] .dot { background: var(--axis); }

/* ---------------------------------------------------------- tabs & controls */

.tabs {
  display: flex;
  gap: 4px;
  border-bottom: 1px solid var(--border);
  margin-bottom: 14px;
}

.tab {
  appearance: none;
  background: none;
  border: 0;
  border-bottom: 2px solid transparent;
  margin-bottom: -1px;
  padding: 9px 14px;
  font: inherit;
  font-weight: 550;
  color: var(--text-secondary);
  cursor: pointer;
  border-radius: 8px 8px 0 0;
}

.tab:hover { background: var(--wash); }
.tab[aria-selected="true"] {
  color: var(--text-primary);
  border-bottom-color: var(--accent);
}

.controls {
  display: flex;
  flex-wrap: wrap;
  align-items: center;
  gap: 10px 14px;
  margin-bottom: 18px;
}

.control { display: flex; align-items: center; gap: 8px; }

.control > label,
.control-label {
  font-size: 12.5px;
  color: var(--text-secondary);
}

.segmented {
  display: inline-flex;
  background: var(--surface-1);
  border: 1px solid var(--border);
  border-radius: 999px;
  padding: 2px;
  gap: 2px;
}

.segmented button {
  appearance: none;
  border: 0;
  background: none;
  font: inherit;
  font-size: 13px;
  color: var(--text-secondary);
  padding: 4px 12px;
  border-radius: 999px;
  cursor: pointer;
  white-space: nowrap;
}

.segmented button:hover { background: var(--wash); }
.segmented button[aria-pressed="true"] {
  background: var(--surface-2);
  color: var(--text-primary);
  font-weight: 600;
}

select {
  appearance: none;
  font: inherit;
  font-size: 13px;
  color: var(--text-primary);
  background: var(--surface-1);
  border: 1px solid var(--border);
```

```
  border-radius: 999px;
  padding: 5px 30px 5px 13px;
  cursor: pointer;
  background-image: linear-gradient(45deg, transparent 50%, var(--text-secondary) 50%),
                    linear-gradient(135deg, var(--text-secondary) 50%, transparent 50%);
  background-position: calc(100% - 16px) 51%, calc(100% - 11px) 51%;
  background-size: 5px 5px, 5px 5px;
  background-repeat: no-repeat;
}

button:focus-visible,
select:focus-visible,
svg:focus-visible,
a:focus-visible,
[tabindex]:focus-visible {
  outline: 2px solid var(--accent);
  outline-offset: 2px;
}

/* The theme toggle: moon while light (click for dark), sun while dark. */
.theme-toggle {
  appearance: none;
  display: inline-flex;
  align-items: center;
  justify-content: center;
  width: 30px;
  height: 30px;
  padding: 0;
  background: var(--surface-1);
  border: 1px solid var(--border);
  border-radius: 999px;
  color: var(--text-secondary);
  cursor: pointer;
  flex: none;
  align-self: center;
}

.theme-toggle:hover { color: var(--accent); background: var(--wash); }
.theme-toggle .ico { width: 15px; height: 15px; display: block; }
.theme-toggle .ico-sun { display: none; }
[data-theme="dark"] .theme-toggle .ico-sun { display: block; }
[data-theme="dark"] .theme-toggle .ico-moon { display: none; }

.scope-note {
  font-size: 12.5px;
  color: var(--text-secondary);
  margin: 0 0 14px;
}

/* ------------------------------------------------------------ stat tiles */

.kpis {
  display: grid;
  grid-template-columns: repeat(auto-fit, minmax(158px, 1fr));
  gap: var(--gap);
  margin-bottom: var(--gap);
}

.tile {
  background: var(--surface-1);
  border: 1px solid var(--border);
  border-radius: var(--radius);
  padding: 14px 16px 13px;
  min-width: 0;
}

.tile .label {
  font-size: 12.5px;
  color: var(--text-secondary);
  margin-bottom: 3px;
}

.tile .value {
  font-size: 26px;
  font-weight: 620;
  letter-spacing: -0.015em;
  line-height: 1.15;
  overflow-wrap: anywhere;
}

.tile .sub {
  font-size: 12px;
  color: var(--text-secondary);
  margin-top: 3px;
}
```

```
406
407 .tile.hero { grid-column: span 2; }
408 .tile.hero .value { font-size: 48px; letter-spacing: -0.02em; }
409 .tile.hero .spark { margin-top: 8px; display: block; }
410
411 @media (max-width: 520px) {
412   .tile.hero { grid-column: span 2; }
413   .tile.hero .value { font-size: 40px; }
414   .kpis { grid-template-columns: repeat(2, 1fr); gap: 10px; }
415   .tile { padding: 12px 13px; }
416   .tile .value { font-size: 22px; }
417 }
418
419 /* small figures inside a card (gaps, confidence) — not nested cards */
420 .figures {
421   display: grid;
422   grid-template-columns: repeat(auto-fit, minmax(112px, 1fr));
423   gap: 12px 14px;
424 }
425 .figure .label { font-size: 12.5px; color: var(--text-secondary); }
426 .figure .value { font-size: 22px; font-weight: 620; letter-spacing: -0.01em; }
427 .figure .sub { font-size: 12px; color: var(--text-secondary); }
428
429 /* ------------------------------------------------------------------ system */
430
431 .system-strip {
432   display: flex;
433   flex-wrap: wrap;
434   align-items: center;
435   gap: 8px 10px;
436   background: var(--surface-1);
437   border: 1px solid var(--border);
438   border-radius: var(--radius);
439   padding: 10px 14px;
440   margin-bottom: var(--gap);
441   font-size: 13px;
442 }
443
444 .system-strip .strip-label {
445   color: var(--text-secondary);
446   font-size: 12.5px;
447   margin-right: 2px;
448 }
449
450 .health-chip {
451   display: inline-flex;
452   align-items: center;
453   gap: 6px;
454   border: 1px solid var(--border);
455   border-radius: 999px;
456   padding: 3px 10px 3px 8px;
457   background: var(--plane);
458   color: var(--text-primary);
459   white-space: nowrap;
460 }
461
462 .health-chip .icon {
463   width: 14px;
464   height: 14px;
465   flex: none;
466   display: block;
467 }
468
469 .health-chip .when { color: var(--text-secondary); font-size: 12px; }
470
471 .linkish {
472   appearance: none;
473   background: none;
474   border: 0;
475   padding: 0;
476   font: inherit;
477   font-size: 12.5px;
478   color: var(--text-secondary);
479   text-decoration: underline;
480   text-underline-offset: 2px;
481   cursor: pointer;
482   margin-left: auto;
483 }
484
485 .linkish:hover { color: var(--text-primary); }
486
487 /* ------------------------------------------------------------------ cards */
488
489 .grid {
490   display: grid;
```

```
  grid-template-columns: repeat(2, minmax(0, 1fr));
  gap: var(--gap);
  align-items: start;
}

.card {
  background: var(--surface-1);
  border: 1px solid var(--border);
  border-radius: var(--radius);
  padding: 15px 17px 16px;
  min-width: 0;
}

.card.span2 { grid-column: 1 / -1; }

.card-head {
  display: flex;
  align-items: flex-start;
  gap: 12px;
  margin-bottom: 4px;
}

.card-title {
  font-size: 14.5px;
  font-weight: 620;
  margin: 0;
  letter-spacing: -0.005em;
}

.card-sub {
  font-size: 12.5px;
  color: var(--text-secondary);
  margin: 2px 0 0;
}

.card-head .spacer { flex: 1 1 auto; }

.toggle {
  appearance: none;
  font: inherit;
  font-size: 12px;
  color: var(--text-secondary);
  background: var(--plane);
  border: 1px solid var(--border);
  border-radius: 999px;
  padding: 3px 11px;
  cursor: pointer;
  flex: none;
}

.toggle:hover { background: var(--wash); color: var(--text-primary); }
.toggle[aria-pressed="true"] { background: var(--surface-2); color: var(--text-primary); }

.card-body { margin-top: 10px; }
.chart-host { display: block; }
.chart-host svg { display: block; width: 100%; height: auto; overflow: visible; }

/* ------------------------------------------------------------ presence */

/* Who is here.  The hub, and each cot it can see, as pills on a strip
 * of their own -- the same shape as the system strip below it, because
 * they answer the same kind of question: what is the state of the
 * machine right now. */
.presence-strip {
  display: flex;
  flex-wrap: wrap;
  align-items: center;
  gap: 8px 10px;
  background: var(--surface-1);
  border: 1px solid var(--border);
  border-radius: var(--radius);
  padding: 10px 14px;
  margin-bottom: var(--gap);
  font-size: 13px;
}

.presence-strip .strip-label {
  color: var(--text-secondary);
  font-size: 12.5px;
  margin-right: 2px;
}

.presence-chip {
  display: inline-flex;
  align-items: center;
```

```
  gap: 7px;
  border: 1px solid var(--border);
  border-radius: 999px;
  padding: 3px 10px 3px 8px;
  background: var(--plane);
  color: var(--text-primary);
  white-space: nowrap;
}

.presence-chip .presence-dot {
  width: 8px;
  height: 8px;
  border-radius: 50%;
  background: var(--text-muted);
  flex: none;
  display: block;
}

.presence-chip[data-state="on"] .presence-dot { background: var(--good); }
.presence-chip[data-state="off"] .presence-dot { background: var(--text-muted); }
.presence-chip[data-state="deaf"] .presence-dot { background: var(--warning); }

.presence-chip .p-word { color: var(--text-secondary); }
.presence-chip .p-detail { color: var(--text-muted); font-size: 12px; }

.legend {
  display: flex;
  flex-wrap: wrap;
  gap: 6px 14px;
  margin-top: 10px;
  font-size: 12.5px;
  color: var(--text-secondary);
}

.legend .item { display: inline-flex; align-items: center; gap: 6px; }
.legend .swatch { width: 10px; height: 10px; border-radius: 3px; flex: none; }

.scale-legend {
  display: flex;
  align-items: center;
  gap: 8px;
  margin-top: 12px;
  font-size: 12px;
  color: var(--text-secondary);
}

.scale-legend .ramp { display: flex; gap: 2px; }
.scale-legend .ramp i { width: 20px; height: 9px; border-radius: 2px; display: block; }

.card-note {
  font-size: 12px;
  color: var(--text-secondary);
  margin: 10px 0 0;
}

/* ---------------------------------------------------------------- tables */

.table-wrap { overflow-x: auto; }

table.data {
  border-collapse: collapse;
  width: 100%;
  font-size: 13px;
}

table.data th,
table.data td {
  text-align: left;
  padding: 7px 10px 7px 0;
  border-bottom: 1px solid var(--grid);
  white-space: nowrap;
}

table.data th {
  font-weight: 550;
  color: var(--text-secondary);
  font-size: 12px;
  border-bottom: 1px solid var(--axis);
}

table.data tr:last-child td { border-bottom: 0; }
table.data td.n, table.data th.n { text-align: right; font-variant-numeric: tabular-nums; }
table.data td .sub { display: block; font-size: 11.5px; color: var(--text-secondary); }

.cot-chip { display: inline-flex; align-items: center; gap: 6px; }
```

```
.cot-chip i { width: 9px; height: 9px; border-radius: 3px; flex: none; display: block; }

.kind-tag {
  display: inline-flex;
  align-items: center;
  gap: 5px;
  font-size: 12px;
  color: var(--text-secondary);
}
.kind-tag .icon { width: 13px; height: 13px; flex: none; }

.meter {
  display: inline-block;
  vertical-align: middle;
  width: 62px;
  height: 7px;
  border-radius: 4px;
  background: var(--seq-100);
  overflow: hidden;
  margin-right: 8px;
}
.meter i { display: block; height: 100%; background: var(--seq-450); border-radius: 4px; }

.table-foot {
  display: flex;
  align-items: center;
  gap: 12px;
  margin-top: 10px;
  font-size: 12px;
  color: var(--text-secondary);
}

.row-toggle { margin-bottom: 10px; }

/* ------------------------------------------------------------- insights */

.insights { list-style: none; margin: 0; padding: 0; }

.insights li {
  display: flex;
  gap: 10px;
  padding: 9px 0;
  border-bottom: 1px solid var(--grid);
  font-size: 14px;
  line-height: 1.45;
}

.insights li:last-child { border-bottom: 0; }

.insights .bullet {
  width: 7px;
  height: 7px;
  border-radius: 50%;
  background: var(--seq-300);
  flex: none;
  margin-top: 7px;
}

/* ------------------------------------------------------------- empty state */

.empty {
  background: var(--surface-1);
  border: 1px solid var(--border);
  border-radius: var(--radius);
  padding: 44px 26px 40px;
  text-align: center;
}

.empty svg { display: block; margin: 0 auto 16px; }
.empty .art-line { stroke: var(--axis); }
.empty .art-soft { stroke: var(--grid); }
.empty .art-dot { fill: var(--seq-100); }
.empty h2 { font-size: 18px; font-weight: 620; margin: 0 0 8px; }
.empty p {
  margin: 0 auto 6px;
  max-width: 46ch;
  color: var(--text-secondary);
  font-size: 14px;
}
.empty code {
  font-family: ui-monospace, SFMono-Regular, Consolas, monospace;
  font-size: 12.5px;
  background: var(--surface-2);
  border-radius: 5px;
  padding: 1px 6px;
```

```
  color: var(--text-primary);
}

.mini-empty {
  color: var(--text-secondary);
  font-size: 13px;
  padding: 18px 0 10px;
}

/* Heatmap cell numbers wear ink, never the cell's colour: white on the
 * deep end of the light ramp, near-black on the bright end of the dark
 * ramp (which runs the other way). */
.heat-lo { fill: var(--text-primary); }
.heat-hi { fill: #ffffff; }
[data-theme="dark"] .heat-hi { fill: #020404; }

/* ---------------------------------------------------------------- tooltip */

.tooltip {
  position: fixed;
  z-index: 60;
  pointer-events: none;
  background: var(--surface-1);
  border: 1px solid var(--border-strong);
  border-radius: 10px;
  box-shadow: 0 6px 22px rgba(11, 11, 11, 0.10);
  padding: 8px 11px 9px;
  font-size: 12.5px;
  max-width: 260px;
  color: var(--text-primary);
}

.tooltip[hidden] { display: none; }
[data-theme="dark"] .tooltip { box-shadow: 0 6px 22px rgba(0, 0, 0, 0.5); }
.tooltip .tt-title { color: var(--text-secondary); font-size: 11.5px; margin-bottom: 5px; }
.tooltip .tt-row { display: flex; align-items: center; gap: 7px; margin-top: 3px; }
.tooltip .tt-key { width: 12px; height: 3px; border-radius: 2px; flex: none; }
.tooltip .tt-val { font-weight: 640; font-variant-numeric: tabular-nums; }
.tooltip .tt-name { color: var(--text-secondary); }
.tooltip .tt-total { margin-top: 6px; padding-top: 5px; border-top: 1px solid var(--grid); }

/* ---------------------------------------------------------------- refresh */

.refreshing { opacity: 0.62; transition: opacity 140ms ease; }
#view-overview, #view-patterns { transition: opacity 140ms ease; }

.footer {
  margin-top: 26px;
  font-size: 12px;
  color: var(--text-secondary);
  display: flex;
  flex-wrap: wrap;
  gap: 4px 16px;
}

.sr-only {
  position: absolute;
  width: 1px; height: 1px;
  padding: 0; margin: -1px;
  overflow: hidden;
  clip: rect(0 0 0 0);
  white-space: nowrap;
  border: 0;
}

/* ---------------------------------------------------------------- responsive */

@media (max-width: 720px) {
  .app { padding: 16px 13px 44px; }
  .grid { grid-template-columns: minmax(0, 1fr); }
  .card.span2 { grid-column: auto; }
  .topbar { margin-bottom: 12px; }
  .hubmeta { flex-basis: 100%; order: 3; }
  .controls { gap: 8px 10px; }
  .card { padding: 13px 13px 14px; }
}

@media (prefers-reduced-motion: reduce) {
  * { transition: none !important; animation: none !important; }
}
```

dashboard/web/app.js

2140 lines · 83,869 bytes

```
/* Amazing's CryNet — the caregiver's record.
 *
 * Plain ES2019-ish vanilla JS.  No modules, no dependencies, no build step;
 * it is loaded with a plain <script src="app.js">.
 *
 * The rules this file lives by:
 *
 *   1. A record with start == 0 happened, but the hub did not know WHEN.
 *      It is never placed on a time axis.  Every series builder filters on
 *      a positive timestamp before it touches a date, and the episodes
 *      table prints "Time unknown" rather than inventing 1970.
 *   2. Nothing is ever NaN, undefined or "0/0".  A missing number is "—".
 *   3. An empty dashboard is a normal first run, not an error.
 *   4. A cot that went deaf was NOT being watched.  That is surfaced at
 *      the top of the page, because a quiet stretch during a deaf spell
 *      means nothing at all.
 *
 * Field names come from server.py's build_summary() / build_analytics(),
 * not from prose.  Note that /api/v1/events returns raw store columns
 * (start_at / end_at / received_at), while summary.recent uses start / end.
 */
(function () {
  'use strict';

  var API = '/api/v1';
  var POLL_MS = 30000;
  var PRESENCE_POLL_MS = 10000;
  var NS = 'http://www.w3.org/2000/svg';
  var NIGHT_START = 20;          // server.py: NIGHT_START_HOUR
  var NIGHT_END = 6;             // server.py: NIGHT_END_HOUR
  var RECENT_PAGE = 12;
  var NIGHTS_PAGE = 12;
  var SYSTEM_KINDS = ['deaf', 'heard', 'boot', 'test'];

  /* Fixed categorical order, assigned by cot number and never re-ranked. */
  var SERIES = ['--series-1', '--series-2', '--series-3', '--series-4',
                '--series-5', '--series-6', '--series-7', '--series-8'];
  /* Sequential ramp for magnitude only (heatmap). */
  var RAMP = ['--seq-100', '--seq-200', '--seq-300',
              '--seq-400', '--seq-500', '--seq-550'];

  function v(name) { return 'var(' + name + ')'; }

  /* ------------------------------------------------------------------ DOM */

  function apply(node, attrs) {
    if (!attrs) return node;
    for (var k in attrs) {
      if (!Object.prototype.hasOwnProperty.call(attrs, k)) continue;
      var val = attrs[k];
      if (val === null || val === undefined || val === false) continue;
      if (k === 'text') node.appendChild(document.createTextNode(String(val)));
      else if (k === 'style') { for (var p in val) node.style[p] = val[p]; }
      else if (val === true) node.setAttribute(k, '');
      else node.setAttribute(k, String(val));
    }
    return node;
  }

  function kids(node, list) {
    if (list === null || list === undefined || list === false) return node;
    if (Array.isArray(list)) {
      for (var i = 0; i < list.length; i++) kids(node, list[i]);
      return node;
    }
    if (typeof list === 'string' || typeof list === 'number') {
      node.appendChild(document.createTextNode(String(list)));
      return node;
    }
    node.appendChild(list);
    return node;
  }

  function h(tag, attrs, content) {
    var n = apply(document.createElement(tag), attrs);
    return kids(n, content);
  }

  function svgEl(tag, attrs, content) {
```

```
  var n = apply(document.createElementNS(NS, tag), attrs);
  return kids(n, content);
}

function clear(node) { while (node && node.firstChild) node.removeChild(node.firstChild); }
function show(node, on) { if (node) node.hidden = !on; }
function text(node, str) { if (node) node.textContent = str; }

/* ------------------------------------------------------------ formatting */

function isNum(x) { return typeof x === 'number' && isFinite(x); }

function num(x) {
  if (x === null || x === undefined || x === '') return '—';
  if (!isNum(x)) { var n = Number(x); if (!isFinite(n)) return '—'; x = n; }
  return x.toLocaleString();
}

function dec(x, places) {
  if (x === null || x === undefined || x === '') return '—';
  if (!isNum(x)) { var n = Number(x); if (!isFinite(n)) return '—'; x = n; }
  return x.toFixed(places === undefined ? 1 : places);
}

function pct(x) {
  if (x === null || x === undefined || !isFinite(Number(x))) return '—';
  return Number(x).toFixed(1) + '%';
}

/* A duration in seconds, written the way a person reads it. */
function dur(sec) {
  if (sec === null || sec === undefined || !isFinite(Number(sec))) return '—';
  var t = Math.round(Number(sec));
  if (t < 0) return '—';
  if (t < 60) return t + ' s';
  var m = Math.floor(t / 60), r = t % 60;
  if (m < 60) return r ? m + ' min ' + r + ' s' : m + ' min';
  var hh = Math.floor(m / 60), mm = m % 60;
  return mm ? hh + ' h ' + mm + ' min' : hh + ' h';
}

/* Compact form for axis ticks, where "1 min 15 s" is too long. */
function durAxis(sec) {
  var t = Number(sec);
  if (!isFinite(t)) return '';
  if (t >= 3600) return (t / 3600).toFixed(t % 3600 ? 1 : 0) + 'h';
  if (t >= 60) return Math.round(t / 60) + 'm';
  return Math.round(t) + 's';
}

function minutes(min) {
  if (min === null || min === undefined || !isFinite(Number(min))) return '—';
  return dur(Number(min) * 60);
}

function ts(ms) { return new Date(ms * 1000); }

function clockAt(sec) {
  if (!sec) return '—';
  return ts(sec).toLocaleTimeString(undefined, { hour: '2-digit', minute: '2-digit' });
}

function dateAt(sec) {
  if (!sec) return '—';
  return ts(sec).toLocaleDateString(undefined, { day: 'numeric', month: 'short', year: 'numeric' });
}

function dateTimeAt(sec) {
  if (!sec) return '—';
  return ts(sec).toLocaleString(undefined, {
    day: 'numeric', month: 'short', hour: '2-digit', minute: '2-digit'
  });
}

function ago(sec) {
  if (!sec) return '—';
  var d = Math.round(Date.now() / 1000) - sec;
  if (d < 0) return 'just now';
  if (d < 60) return 'just now';
  if (d < 3600) return Math.round(d / 60) + ' min ago';
  if (d < 86400) return Math.round(d / 3600) + ' h ago';
  var days = Math.round(d / 86400);
  return days + (days === 1 ? ' day ago' : ' days ago');
}

```

```
/* "2026-11-20" is a LOCAL calendar day from the server.  Parsing it with
 * new Date(iso) would read it as UTC and shift it a day west of Greenwich,
 * so it is taken apart by hand. */
function dayDate(iso) {
  var p = String(iso || '').split('-');
  if (p.length !== 3) return null;
  var d = new Date(Number(p[0]), Number(p[1]) - 1, Number(p[2]), 12, 0, 0, 0);
  return isNaN(d.getTime()) ? null : d;
}

function isoDay(d) {
  var m = d.getMonth() + 1, day = d.getDate();
  return d.getFullYear() + '-' + (m < 10 ? '0' + m : m) + '-' + (day < 10 ? '0' + day : day);
}

function dayShort(iso) {
  var d = dayDate(iso);
  if (!d) return String(iso || '—');
  return d.toLocaleDateString(undefined, { weekday: 'short', day: 'numeric', month: 'short' });
}

function dayLong(iso) {
  var d = dayDate(iso);
  if (!d) return String(iso || '—');
  return d.toLocaleDateString(undefined, {
    weekday: 'long', day: 'numeric', month: 'long', year: 'numeric'
  });
}

/* A night is named by the evening it starts, exactly as server.py's
 * night_key() does: 20:00 on the 20th through 06:00 on the 21st is
 * "the night of the 20th". */
function nightShort(iso) { return dayShort(iso); }
function nightLong(iso) {
  var d = dayDate(iso);
  if (!d) return String(iso || '—');
  var next = new Date(d.getTime());
  next.setDate(next.getDate() + 1);
  var f = function (x) {
    return x.toLocaleDateString(undefined, { weekday: 'short', day: 'numeric', month: 'short' });
  };
  return 'Night of ' + f(d) + ' into ' + f(next);
}

var WEEKDAYS = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'];
function weekdayName(w) { return WEEKDAYS[w] || ('—'); }
function hourLabel(hh) { return (hh < 10 ? '0' + hh : String(hh)) + ':00'; }
function isNightHour(hh) { return hh >= NIGHT_START || hh < NIGHT_END; }

function cotName(c) { return c ? 'Cot ' + c : 'Cot unassigned'; }
function cotColor(c) { return c ? v(SERIES[(c - 1) % SERIES.length]) : v('--series-other'); }
function cotList(arr) {
  if (!arr || !arr.length) return '—';
  return arr.map(cotName).join(', ');
}

function hubName(d) {
  if (!d) return '';
  return d.name || d.device || 'Hub';
}

/* ------------------------------------------------------------- state */

var ids = {};
var ID_LIST = ['hubmeta', 'hubmeta-text', 'conn', 'conn-text', 'theme',
  'tab-overview', 'tab-patterns',
  'range', 'cot', 'device', 'device-control', 'scope-note', 'view-overview',
  'view-patterns', 'empty-overview', 'empty-overview-detail', 'overview-body',
  'kpis', 'untimed-note', 'presence-strip', 'system-strip', 'overview-grid',
  'empty-patterns',
  'patterns-body', 'patterns-scope', 'patterns-grid', 'footer-range',
  'footer-updated', 'tooltip'];

var state = {
  days: 14,
  cot: 0,
  device: '',
  tab: 'overview',
  summary: null,
  analytics: null,
  devices: [],
  presence: null,
  system: [],
  systemOk: true,
  counts: null,
```

```
    knownCots: [],
    tables: {},
    recentAll: false,
    nightsAll: false,
    systemOpen: false,
    lastGood: 0,
    failures: 0,
    busy: false,
    busySince: 0,
    presenceBusy: false
  };

  var charts = [];      /* { host, draw } — redrawn on resize and tab change */
  var timer = null;
  var presenceTimer = null;
  var pristine = {};    /* original empty-state wording, restored after errors */

  /* ------------------------------------------------------------- fetch */

  /* Days, hours and nights are bucketed on the server, and the server may
     not be in the caregiver's timezone (the hosted copy runs in UTC), so
     every request says which zone the numbers should be counted in. */
  var TZ = '';
  try { TZ = Intl.DateTimeFormat().resolvedOptions().timeZone || ''; } catch (e) { /* keep '' */ }

  /* A read that never comes back must not freeze the page.  The busy
   * flag below is only a guard against overlapping reads; if a request
   * hangs, the flag would stay set for ever and the page would sit on
   * its last answer -- which is how a hub that has gone offline keeps
   * reading as online.  So every read is given a deadline, and the
   * guard has a deadline of its own. */
  var FETCH_TIMEOUT_MS = 15000;
  var BUSY_STUCK_MS = 60000;

  function getJSON(path) {
    if (TZ) path += (path.indexOf('?') >= 0 ? '&' : '?') + 'tz=' + encodeURIComponent(TZ);
    var ctl = (typeof AbortController !== 'undefined') ? new AbortController() : null;
    var timer = null;
    if (ctl) timer = setTimeout(function () { ctl.abort(); }, FETCH_TIMEOUT_MS);
    function done(fn, arg) {
      if (timer) { clearTimeout(timer); timer = null; }
      return fn(arg);
    }
    return fetch(API + path, {
      cache: 'no-store',
      credentials: 'same-origin',
      signal: ctl ? ctl.signal : undefined
    })
      .then(function (r) {
        if (!r.ok) throw new Error('HTTP ' + r.status);
        return r.json();
      })
      .then(function (v) { return done(function () { return v; }); },
            function (e) { return done(function () { throw e; }); });
  }

  /* Never rejects: resolves to null so one bad endpoint cannot blank the page. */
  function soft(path) {
    return getJSON(path).catch(function () { return null; });
  }

  function refresh() {
    if (state.busy) {
      /* A read that has been running for a minute is a read that is
       * never going to finish; let this one go through. */
      if (Date.now() - state.busySince < BUSY_STUCK_MS) return;
      state.busy = false;
    }
    state.busy = true;
    state.busySince = Date.now();
    if (state.summary) {
      ids['view-overview'].classList.add('refreshing');
      ids['view-patterns'].classList.add('refreshing');
    }

    var now = Math.floor(Date.now() / 1000);
    var from = now - state.days * 86400;
    var to = now + 86400;
    var q = '?days=' + state.days + '&cot=' + state.cot;

    var jobs = [
      soft('/summary' + q),
      soft('/analytics?days=' + state.days),
      soft('/devices'),
      soft('/health'),
```

```
      soft('/presence')
    ].concat(SYSTEM_KINDS.map(function (k) {
      return soft('/events?kind=' + k + '&from=' + from + '&to=' + to + '&limit=300');
    }));

    Promise.all(jobs).then(function (res) {
      var summary = res[0], analytics = res[1], devices = res[2], health = res[3];
      var presence = res[4];
      var sysLists = res.slice(5);

      var sys = [], sysOk = true;
      sysLists.forEach(function (r) {
        if (!r || !Array.isArray(r.events)) { sysOk = false; return; }
        sys = sys.concat(r.events);
      });
      sys.sort(function (a, b) { return (b.start_at || 0) - (a.start_at || 0); });

      state.system = sys;
      state.systemOk = sysOk;

      if (devices && Array.isArray(devices.devices)) state.devices = devices.devices;
      else if (summary && Array.isArray(summary.devices)) state.devices = summary.devices;

      /* The presence is separate from the record: it says whether the
       * thing that watches is working, which is a different question
       * from what it has heard.  A service that does not have the
       * endpoint yet simply leaves it null. */
      setPresence(presence);

      if (analytics) state.analytics = analytics;

      if (summary) {
        state.summary = summary;
        state.lastGood = Date.now();
        state.failures = 0;
        rememberCots(summary, analytics);
        setConn(health && health.ok ? 'ok' : 'stale',
          health && health.ok ? 'Live · updated ' + clockAt(now)
                              : 'Service replied, but reports a problem');
        if (!sysOk) setConn('stale', 'Live · some system records unavailable');
      } else {
        state.failures++;
        setConn('stale', state.lastGood
          ? 'Offline · retrying (last read ' + clockAt(Math.round(state.lastGood / 1000)) + ')'
          : 'Offline · retrying every 30 s');
      }

      /* More than one hub: the API cannot split the totals, so at least
       * count what each hub uploaded and say so plainly. */
      if (state.devices.length > 1) {
        return soft('/events?kind=cry&from=' + from + '&to=' + to + '&limit=2000')
          .then(function (r) {
            if (r && Array.isArray(r.events)) {
              var counts = {};
              r.events.forEach(function (e) {
                var k = e.device || 'unknown';
                counts[k] = (counts[k] || 0) + 1;
              });
              state.counts = { counts: counts, capped: r.events.length >= 2000 };
            } else {
              state.counts = null;
            }
          });
      }
      state.counts = null;
      return null;
    }).catch(function () {
      state.failures++;
      setConn('stale', 'Offline · retrying every 30 s');
    }).then(function () {
      state.busy = false;
      ids['view-overview'].classList.remove('refreshing');
      ids['view-patterns'].classList.remove('refreshing');
      try {
        render();
      } catch (err) {
        setConn('stale', 'Could not draw the record · retrying');
        if (window.console && console.error) console.error(err);
      }
      schedule();
    });
  }

  function schedule() {
    if (timer) clearTimeout(timer);
```

```
    timer = setTimeout(refresh, POLL_MS);
  }

  /* The presence has its own, faster clock.
   *
   * The page asks the service every thirty seconds for the record, and
   * that is right: cries do not arrive every second, and the read is
   * not cheap.  But "is the hub still there" is a different question,
   * it is the one the owner opens the dashboard to ask, and it is
   * answered by one small endpoint.  So it is asked on its own, every
   * ten seconds, without dragging the rest of the page along with it.
   * The three links in the chain -- the hub speaks every 30 s, the
   * service calls a hub gone after 90 s, the page asks every 10 s --
   * add up to about two minutes between switching the hub off and the
   * page saying so.  The hub could speak every 15 s and this could be
   * shorter still, but the cots share the hub's radio and the owner
   * set the order: nothing costs the cots anything. */
  function refreshPresence() {
    if (state.presenceBusy) return;
    state.presenceBusy = true;
    soft('/presence').then(function (p) {
      state.presenceBusy = false;
      if (setPresence(p)) {
        try {
          renderChrome();
          renderPresence();
        } catch (err) {
          if (window.console && console.error) console.error(err);
        }
      }
      schedulePresence();
    });
  }

  /* Take the newest answer.  The fast poll and the full read both ask
   * for the presence, and they can arrive out of order; an older
   * answer must not replace a newer one, or the page would say the hub
   * is back when it has just gone. */
  function setPresence(p) {
    if (!p || !Array.isArray(p.hubs)) return false;
    if (state.presence && typeof p.now === 'number' &&
        typeof state.presence.now === 'number' &&
        p.now < state.presence.now) return false;
    state.presence = p;
    return true;
  }

  function schedulePresence() {
    if (presenceTimer) clearTimeout(presenceTimer);
    presenceTimer = setTimeout(refreshPresence, PRESENCE_POLL_MS);
  }

  function setConn(kind, msg) {
    if (!ids.conn) return;
    ids.conn.setAttribute('data-state', kind);
    text(ids['conn-text'], msg);
  }

  /* The cot list must survive a cot filter: summary.per_cot shrinks to the
   * chosen cot, but analytics.per_cot always covers every cot. */
  function rememberCots(summary, analytics) {
    var set = {};
    state.knownCots.forEach(function (c) { set[c] = true; });
    var add = function (c) { if (isNum(Number(c))) set[Number(c)] = true; };
    (summary && summary.per_cot || []).forEach(function (b) { add(b.cot); });
    (summary && summary.heatmap && summary.heatmap.cots || []).forEach(add);
    (analytics && analytics.per_cot || []).forEach(function (b) { add(b.cot); });
    state.knownCots = Object.keys(set).map(Number).sort(function (a, b) { return a - b; });
  }

  /* ------------------------------------------------------------- tooltip */

  function tipShow(x, y, data) {
    var el = ids.tooltip;
    if (!el || !data) return;
    clear(el);
    if (data.title) el.appendChild(h('div', { class: 'tt-title', text: data.title }));
    (data.rows || []).forEach(function (r) {
      el.appendChild(h('div', { class: 'tt-row' + (r.total ? ' tt-total' : '') }, [
        r.color ? h('span', { class: 'tt-key', style: { background: r.color } }) : null,
        h('span', { class: 'tt-name', text: r.name }),
        h('span', { class: 'tt-val', text: r.value })
      ]));
    });
    el.hidden = false;
```

```
  tipPlace(x, y);
}

function tipPlace(x, y) {
  var el = ids.tooltip;
  if (!el || el.hidden) return;
  var w = el.offsetWidth, hh = el.offsetHeight;
  var left = x + 14, top = y + 16;
  if (left + w > window.innerWidth - 8) left = x - w - 14;
  if (left < 8) left = 8;
  if (top + hh > window.innerHeight - 8) top = y - hh - 12;
  if (top < 8) top = 8;
  el.style.left = left + 'px';
  el.style.top = top + 'px';
}

function tipHide() { if (ids.tooltip) ids.tooltip.hidden = true; }

function tipify(node, build, highlight) {
  node.addEventListener('pointerenter', function (ev) {
    var d = build();
    if (d) tipShow(ev.clientX, ev.clientY, d);
    if (highlight) highlight(true);
  });
  node.addEventListener('pointermove', function (ev) { tipPlace(ev.clientX, ev.clientY); });
  node.addEventListener('pointerleave', function () {
    tipHide();
    if (highlight) highlight(false);
  });
  return node;
}

/* ------------------------------------------------------------------ icons */

var ICON_PATHS = {
  alert: ['M8 2.3 1.7 13.5h12.6z', 'M8 6.5v3.1', 'M8 11.7h.01'],
  check: ['M8 1.9a6.1 6.1 0 1 0 0 12.2A6.1 6.1 0 0 0 8 1.9z', 'M5.2 8.1 7.2 10.1 10.8 6.1'],
  power: ['M8 2.2v5.1', 'M4.7 4.6a4.7 4.7 0 1 0 6.6 0'],
  test: ['M6 2.3h4', 'M6.7 2.3v4.3L4.3 12a1.3 1.3 0 0 0 1.2 1.9h5a1.3 1.3 0 0 0 1.2-1.9L9.3 6.6V2.3'],
  info: ['M8 1.9a6.1 6.1 0 1 0 0 12.2A6.1 6.1 0 0 0 8 1.9z', 'M8 7.3v3.6', 'M8 5.1h.01']
};

function icon(name, colorVar, cls) {
  var paths = ICON_PATHS[name] || ICON_PATHS.info;
  return svgEl('svg', {
    class: cls || 'icon', viewBox: '0 0 16 16', fill: 'none',
    stroke: v(colorVar), 'stroke-width': '1.5',
    'stroke-linecap': 'round', 'stroke-linejoin': 'round', 'aria-hidden': 'true'
  }, paths.map(function (d) { return svgEl('path', { d: d }); }));
}

/* ------------------------------------------------------------ chart basics */

function niceMax(m) {
  if (!(m > 0)) return 1;
  var exp = Math.pow(10, Math.floor(Math.log(m) / Math.LN10));
  var f = m / exp;
  var n = f <= 1 ? 1 : f <= 2 ? 2 : f <= 5 ? 5 : 10;
  return n * exp;
}

/* Counts get an even top so both gridline labels stay whole numbers. */
function evenMax(m) {
  if (!(m > 0)) return 2;
  if (m <= 2) return 2;
  if (m <= 10) return Math.ceil(m / 2) * 2;
  return niceMax(m);
}

function hostWidth(host) {
  return Math.round(host.clientWidth || host.getBoundingClientRect().width || 0);
}

/* A smooth curve through a set of points.
 *
 * A straight line between two counts draws a corner at every point,
 * and a corner in a chart of daily counts says something that is not
 * there: that the count jumped between two days and then stayed put.
 * So the line is curved.
 *
 * The curve must not overshoot.  A plain Catmull-Rom spline through
 * these points would dip below the baseline between two quiet days,
 * and a chart of counts that draws a negative count is lying.  The
 * tangents are chosen the Fritsch-Carlson way instead: each point's
 * slope is limited by its neighbours', so the curve turns at a point
```

```
   * and never past it, and it never crosses the line the points sit
   * on.
   *
   * pts is [[x, y], ...] with the x values rising. */
  function smoothPath(pts) {
    var n = pts.length;
    if (n < 2) return '';
    var slopes = new Array(n - 1);
    for (var i = 0; i < n - 1; i++) {
      var dx = pts[i + 1][0] - pts[i][0];
      slopes[i] = dx === 0 ? 0 : (pts[i + 1][1] - pts[i][1]) / dx;
    }
    var m = new Array(n);
    m[0] = slopes[0];
    m[n - 1] = slopes[n - 2];
    for (var j = 1; j < n - 1; j++) {
      var a = slopes[j - 1], b = slopes[j];
      if (a * b <= 0) {
        m[j] = 0;                       /* a peak or a trough: flat here */
      } else {
        var mid = (a + b) / 2;
        var lim = 3 * Math.min(Math.abs(a), Math.abs(b));
        m[j] = Math.abs(mid) > lim ? (mid < 0 ? -lim : lim) : mid;
      }
    }
    var d = [];
    for (var k = 0; k < n - 1; k++) {
      var x0 = pts[k][0], y0 = pts[k][1];
      var x1 = pts[k + 1][0], y1 = pts[k + 1][1];
      var h = (x1 - x0) / 3;
      if (k === 0) {
        d.push('M' + x0.toFixed(2) + ',' + y0.toFixed(2));
      }
      d.push('C' + (x0 + h).toFixed(2) + ',' + (y0 + m[k] * h).toFixed(2) +
             ' ' + (x1 - h).toFixed(2) + ',' + (y1 - m[k + 1] * h).toFixed(2) +
             ' ' + x1.toFixed(2) + ',' + y1.toFixed(2));
    }
    return d.join(' ');
  }

  /* A bar with a rounded top and a square foot.
   *
   * The foot stands on the axis, so it has to be square: a bar that
   * lifted off the line would read as floating above it.  The top is
   * rounded because a hard corner on a wide bar is a slab, and the
   * curve softens the shape without turning it into a blob. */
  function barPath(x, y, w, h, r) {
    r = Math.max(0, Math.min(r, w / 2, h));
    var f = function (n) { return Number(n).toFixed(2); };
    return 'M' + f(x) + ',' + f(y + h) +
           'V' + f(y + r) +
           'A' + f(r) + ',' + f(r) + ' 0 0 1 ' + f(x + r) + ',' + f(y) +
           'H' + f(x + w - r) +
           'A' + f(r) + ',' + f(r) + ' 0 0 1 ' + f(x + w) + ',' + f(y + r) +
           'V' + f(y + h) + 'Z';
  }

  function registerChart(host, draw) {
    charts.push({ host: host, draw: draw });
  }

  function drawCharts() {
    for (var i = 0; i < charts.length; i++) {
      var c = charts[i];
      if (!c.host.isConnected) continue;
      if (!hostWidth(c.host)) continue;      /* hidden tab: draw it when shown */
      try { c.draw(); } catch (err) { if (window.console) console.error(err); }
    }
  }

  /* A hand-built bar chart.  spec:
   *   items   [{ label, value, tip:{title,rows}, }]
   *   color(i) -> css colour,  tick(i) -> string|null
   *   bands   [{ from, to, label }] index ranges shaded behind the bars
   *   fmt     value formatter for tooltips,  axisFmt for the y axis
   */
  function barChart(host, spec) {
    registerChart(host, function () { drawBars(host, spec); });
  }

  function drawBars(host, spec) {
    var W = hostWidth(host);
    if (!W) return;
    var items = spec.items || [];
    var n = items.length;
```

```
    clear(host);
    if (!n) return;

    var H = spec.height || 168;
    var vals = items.map(function (it) { return Math.max(0, Number(it.value) || 0); });
    var maxV = Math.max.apply(null, vals);
    var top = spec.integer === false ? niceMax(maxV) : evenMax(maxV);
    if (!(top > 0)) top = 1;

    var axisFmt = spec.axisFmt || function (x) { return String(Math.round(x)); };
    var labels = [0, top / 2, top].map(axisFmt);
    var padL = 12;
    labels.forEach(function (t) { padL = Math.max(padL, String(t).length * 6.3 + 12); });
    var padR = 6, padT = 12, padB = spec.padB || 26;
    var plotW = Math.max(40, W - padL - padR);
    var plotH = Math.max(30, H - padT - padB);
    var band = plotW / n;
    var barW = Math.max(2, Math.min(band * 0.68, spec.maxBar || 30));

    var svg = svgEl('svg', {
      viewBox: '0 0 ' + W + ' ' + H, width: W, height: H,
      role: 'img', 'aria-label': spec.aria || ''
    }, [svgEl('title', { text: spec.aria || '' })]);

    (spec.bands || []).forEach(function (b) {
      var x0 = padL + band * b.from;
      var w = band * (b.to - b.from + 1);
      svg.appendChild(svgEl('rect', {
        x: x0, y: padT, width: w, height: plotH, fill: v('--wash')
      }));
      if (b.label && w > 44) {
        svg.appendChild(svgEl('text', {
          x: x0 + w / 2, y: padT + 10, 'text-anchor': 'middle',
          'font-size': '9.5', fill: v('--text-muted'), text: b.label
        }));
      }
    });

    for (var g = 0; g < 3; g++) {
      var y = padT + plotH * (1 - g / 2);
      svg.appendChild(svgEl('line', {
        x1: padL, x2: W - padR, y1: y, y2: y,
        stroke: g === 0 ? v('--axis') : v('--grid'), 'stroke-width': '1'
      }));
      svg.appendChild(svgEl('text', {
        x: padL - 6, y: y + 3.5, 'text-anchor': 'end',
        'font-size': '10.5', fill: v('--text-muted'), text: String(labels[g])
      }));
    }

    var tickEvery = spec.tickEvery || Math.max(1, Math.ceil(n / Math.max(3, Math.floor(plotW / 62))));

    items.forEach(function (it, i) {
      var val = vals[i];
      var cx = padL + band * i + band / 2;
      var bar = null;
      if (val > 0) {
        var bh = Math.max(1.5, plotH * (val / top));
        /* The top curve scales with the bar: a narrow bar cannot carry
         * a wide curve, and a tall one can carry more of it. */
        var r = Math.min(barW * 0.24, 6, bh);
        bar = svgEl('path', {
          d: barPath(cx - barW / 2, padT + plotH - bh, barW, bh, r),
          fill: spec.color ? spec.color(i) : v('--series-1'),
          /* An edge of its own, so a bar standing on a pale grid still
           * has a boundary the eye can find. */
          stroke: v('--axis'), 'stroke-width': '1', 'stroke-linejoin': 'round'
        });
        svg.appendChild(bar);
      }
      var lbl = spec.tick ? spec.tick(i) : (i % tickEvery === 0 ? it.label : null);
      if (lbl) {
        svg.appendChild(svgEl('text', {
          x: cx, y: H - padB + 15, 'text-anchor': 'middle',
          'font-size': '10.5', fill: v('--text-muted'), text: String(lbl)
        }));
      }
      var hit = svgEl('rect', {
        x: padL + band * i, y: padT, width: band, height: plotH,
        fill: 'transparent', 'pointer-events': 'all'
      }, [svgEl('title', { text: it.plain || (it.label + ': ' + (spec.fmt ? spec.fmt(val) : val)) })]);
      tipify(hit, function () { return it.tip; }, function (on) {
        if (bar) bar.setAttribute('fill-opacity', on ? '0.72' : '1');
      });
      svg.appendChild(hit);
```

```
  });

  host.appendChild(svg);
}

/* ---------------------------------------------------------------- cards */

function makeCard(o) {
  var head = h('div', { class: 'card-head' }, [
    h('div', {}, [
      h('h3', { class: 'card-title', text: o.title }),
      o.sub ? h('p', { class: 'card-sub', text: o.sub }) : null
    ]),
    h('div', { class: 'spacer' })
  ]);
  var body = h('div', { class: 'card-body' });
  var root = h('div', { class: 'card' + (o.span2 ? ' span2' : '') }, [head, body]);
  return { root: root, head: head, body: body };
}

function note(msg) { return h('p', { class: 'card-note', text: msg }); }
function miniEmpty(msg) { return h('p', { class: 'mini-empty', text: msg }); }

function dataTable(headers, rows, caption) {
  var thead = h('tr', {}, headers.map(function (hd) {
    return h('th', { class: hd.n ? 'n' : null, scope: 'col', text: hd.t });
  }));
  var body = rows.map(function (cells) {
    return h('tr', {}, cells.map(function (c, i) {
      var attrs = { class: headers[i] && headers[i].n ? 'n' : null };
      if (typeof c === 'string' || typeof c === 'number') attrs.text = String(c);
      return h('td', attrs, (typeof c === 'string' || typeof c === 'number') ? null : c);
    }));
  });
  return h('table', { class: 'data' }, [
    caption ? h('caption', { class: 'sr-only', text: caption }) : null,
    h('thead', {}, thead), h('tbody', {}, body)
  ]);
}

function tableWrap(table, hidden) {
  return h('div', { class: 'table-wrap' + (hidden ? ' sr-only' : '') }, table);
}

/* Every chart ships a table twin.  It is always in the document for
 * screen readers (.sr-only) and the toggle makes it visible to everyone. */
function chartWithTwin(card, key, headers, rows, caption, drawFn) {
  var host = h('div', { class: 'chart-host' });
  var wrap = tableWrap(dataTable(headers, rows, caption), !state.tables[key]);
  var btn = h('button', {
    class: 'toggle', type: 'button', 'aria-pressed': state.tables[key] ? 'true' : 'false',
    text: 'Table'
  });
  btn.addEventListener('click', function () {
    state.tables[key] = !state.tables[key];
    btn.setAttribute('aria-pressed', state.tables[key] ? 'true' : 'false');
    wrap.className = 'table-wrap' + (state.tables[key] ? '' : ' sr-only');
  });
  card.head.appendChild(btn);
  card.body.appendChild(host);
  card.body.appendChild(wrap);
  drawFn(host);
  return host;
}

function cotChip(c) {
  return h('span', { class: 'cot-chip' }, [
    h('i', { style: { background: cotColor(c) } }), h('span', { text: cotName(c) })
  ]);
}

function legendItem(color, label) {
  return h('span', { class: 'item' }, [
    h('span', { class: 'swatch', style: { background: color } }), h('span', { text: label })
  ]);
}

function figure(label, value, sub) {
  return h('div', { class: 'figure' }, [
    h('div', { class: 'label', text: label }),
    h('div', { class: 'value', text: value }),
    sub ? h('div', { class: 'sub', text: sub }) : null
  ]);
}

```

```
/* ------------------------------------------------------ series builders */

/* Zero-fill a calendar span so a quiet day is drawn as a quiet day and
 * not silently dropped.  Records are keyed by the server's own local
 * day / night string; keys outside the span are kept, not discarded. */
function spanSeries(range, records, keyName) {
  var map = {};
  (records || []).forEach(function (r) {
    var k = r[keyName];
    if (k) map[k] = r;
  });
  var out = [];
  var a = ts(range && range.from ? range.from : Math.floor(Date.now() / 1000));
  var b = ts(range && range.to ? range.to : Math.floor(Date.now() / 1000));
  a.setHours(12, 0, 0, 0);
  b.setHours(12, 0, 0, 0);
  var d = new Date(a.getTime()), guard = 0;
  while (d.getTime() <= b.getTime() && guard < 400) {
    var k = isoDay(d);
    out.push({ key: k, rec: map[k] || null });
    delete map[k];
    d.setDate(d.getDate() + 1);
    guard++;
  }
  Object.keys(map).forEach(function (k) { out.push({ key: k, rec: map[k] }); });
  out.sort(function (x, y) { return x.key < y.key ? -1 : x.key > y.key ? 1 : 0; });
  return out;
}

/* ------------------------------------------------------------ overview */

function renderKpis(s) {
  var k = s.kpis || {};
  var host = ids.kpis;
  clear(host);

  var days = (s.range && s.range.days) || state.days;
  var scope = state.cot ? cotName(state.cot) : 'all cots';

  var hero = h('div', { class: 'tile hero' }, [
    h('div', { class: 'label', text: 'Cry episodes' }),
    h('div', { class: 'value', text: num(k.total_cries) }),
    h('div', { class: 'sub', text: 'in the last ' + days + ' days · ' + scope })
  ]);
  var spark = sparkline(s);
  if (spark) hero.appendChild(spark);
  host.appendChild(hero);

  host.appendChild(tile('Per day', dec(k.cries_per_day, 1),
    'average over ' + days + ' days'));

  host.appendChild(tile('Time crying', dur(k.total_cry_seconds),
    k.total_cries ? 'across ' + num(k.total_cries) + ' episodes' : 'no episodes in this range'));

  host.appendChild(tile('Typical cry', k.total_cries ? dur(k.median_cry_seconds) : '—',
    k.total_cries ? 'median · longest ' + dur(k.longest_cry_seconds)
                  : 'nothing recorded yet'));

  var nightSub;
  if (!k.nights_affected) nightSub = 'no night cries in this range';
  else if (days >= 7) nightSub = dec(k.nights_per_week, 1) + ' nights a week · ' +
    dec(k.avg_cries_per_night, 1) + ' episodes a night';
  else nightSub = 'nights with at least one cry';
  host.appendChild(tile('Nights affected', num(k.nights_affected), nightSub));
}

function tile(label, value, sub) {
  return h('div', { class: 'tile' }, [
    h('div', { class: 'label', text: label }),
    h('div', { class: 'value', text: value }),
    h('div', { class: 'sub', text: sub })
  ]);
}

function sparkline(s) {
  var series = spanSeries(s.range, s.per_day, 'day');
  if (series.length < 2) return null;
  var vals = series.map(function (x) { return x.rec ? x.rec.cries : 0; });
  var max = Math.max.apply(null, vals);
  if (!(max > 0)) return null;                /* no empty chart frame */
  var W = 100, H = 34, pad = 3;
  var step = W / (vals.length - 1);
  var pts = vals.map(function (val, i) {
    return [i * step, H - pad - (H - pad * 2) * (val / max)];
  });
```

```
  var line = smoothPath(pts);
  return svgEl('svg', {
    class: 'spark', viewBox: '0 0 ' + W + ' ' + H, width: '100%', height: H,
    preserveAspectRatio: 'none', role: 'img',
    'aria-label': 'Daily cry counts across the range, peaking at ' + max + ' in one day.'
  }, [
    svgEl('title', { text: 'Cries per day across the range' }),
    svgEl('path', {
      d: line + ' L' + W + ',' + H + ' L0,' + H + ' Z',
      fill: v('--seq-100')
    }),
    svgEl('path', {
      d: line, fill: 'none', stroke: v('--series-1'),
      'stroke-width': '1.5', 'stroke-linejoin': 'round', 'stroke-linecap': 'round',
      'vector-effect': 'non-scaling-stroke'
    })
  ]);
}

/* The honesty rule.  A record with start == 0 happened at an unknown
 * time; it belongs in the count and nowhere on a time axis. */
function untimedRecords(s) {
  return (s && s.recent || []).filter(function (r) { return !r.start; });
}

function renderUntimed(s) {
  var list = untimedRecords(s);
  var el = ids['untimed-note'];
  if (!list.length) { show(el, false); return; }
  var secs = list.reduce(function (a, r) { return a + (Number(r.duration_s) || 0); }, 0);
  text(el, list.length + (list.length === 1 ? ' episode has' : ' episodes have') +
    ' no known time — the hub had not reached a time server (' + dur(secs) +
    ' of crying). ' + (list.length === 1 ? 'It is' : 'They are') +
    ' counted in the totals, but not placed on any chart, and ' +
    (list.length === 1 ? 'it appears' : 'they appear') +
    ' as "Time unknown" in the episode list.');
  show(el, true);
}

/* ------------------------------------------------------------ the strip */

function listeningState() {
  var byCot = {};
  state.system.forEach(function (e) {
    if (e.kind !== 'deaf' && e.kind !== 'heard') return;
    if (!e.start_at) return;                    /* untimed: cannot order it */
    var c = Number(e.cot) || 0;
    if (!byCot[c] || e.start_at > byCot[c].start_at) byCot[c] = e;
  });
  return byCot;
}

function deafNow() {
  var m = listeningState(), out = [];
  Object.keys(m).forEach(function (c) {
    if (m[c].kind === 'deaf') out.push({ cot: Number(c), since: m[c].start_at });
  });
  out.sort(function (a, b) { return a.cot - b.cot; });
  return out;
}

function renderSystemStrip() {
  var el = ids['system-strip'];
  clear(el);
  var sys = state.system;

  el.appendChild(h('span', { class: 'strip-label', text: 'System' }));

  var listening = listeningState();
  var cots = Object.keys(listening).map(Number).sort(function (a, b) { return a - b; });
  cots.forEach(function (c) {
    var e = listening[c];
    var deaf = e.kind === 'deaf';
    el.appendChild(chip(
      deaf ? 'alert' : 'check',
      deaf ? '--critical' : '--good',
      cotName(c) + (deaf ? ' not being heard' : ' hearing again'),
      (deaf ? 'since ' : '') + clockAt(e.start_at) + ' · ' + ago(e.start_at)
    ));
  });

  var deafCount = sys.filter(function (e) { return e.kind === 'deaf'; }).length;
  if (deafCount) {
    el.appendChild(chip('alert', '--serious',
      deafCount + (deafCount === 1 ? ' deaf spell' : ' deaf spells'), 'in this range'));
```

```
    }

    var boots = sys.filter(function (e) { return e.kind === 'boot' && e.start_at; });
    if (boots.length) {
      el.appendChild(chip('power', '--text-secondary',
        'Hub restarted' + (boots.length > 1 ? ' ×' + boots.length : ''),
        ago(boots[0].start_at)));
    }

    var tests = sys.filter(function (e) { return e.kind === 'test'; }).length;
    if (tests) {
      el.appendChild(chip('test', '--text-secondary',
        tests + (tests === 1 ? ' test record' : ' test records'), 'in this range'));
    }

    if (!sys.length) {
      el.appendChild(chip('info', '--text-secondary',
        state.systemOk ? 'No deaf spells, restarts or tests uploaded'
                       : 'System records could not be read',
        state.systemOk ? 'in this range' : 'retrying'));
    }

    var btn = h('button', {
      class: 'linkish', type: 'button',
      text: state.systemOpen ? 'Hide system records' : 'Show system records'
    });
    btn.addEventListener('click', function () {
      state.systemOpen = !state.systemOpen;
      render();
    });
    el.appendChild(btn);
    show(el, true);
  }

  function chip(iconName, colorVar, label, when) {
    return h('span', { class: 'health-chip' }, [
      icon(iconName, colorVar),
      h('span', { text: label }),
      when ? h('span', { class: 'when', text: when }) : null
    ]);
  }

  function systemCard() {
    var card = makeCard({
      title: 'System records', span2: true,
      sub: 'When the hub was listening, and when it stopped.'
    });
    var rows = state.system.filter(function (e) { return e.start_at; });
    if (!rows.length) {
      card.body.appendChild(miniEmpty('No system records have been uploaded for this range.'));
      return card.root;
    }
    var headers = [{ t: 'When' }, { t: 'Record' }, { t: 'Cot' }, { t: 'Lasted', n: true }, { t: 'Hub' }];
    var body = rows.slice(0, 200).map(function (e) {
      var kindText = { deaf: 'Stopped hearing', heard: 'Hearing again', boot: 'Hub restarted', test: 'Test' }[e.kind] || e.kind;
      var kindIcon = { deaf: 'alert', heard: 'check', boot: 'power', test: 'test' }[e.kind] || 'info';
      var kindColor = e.kind === 'deaf' ? '--critical' : e.kind === 'heard' ? '--good' : '--text-secondary';
      return [
        h('span', {}, [dateTimeAt(e.start_at), h('span', { class: 'sub', text: ago(e.start_at) })]),
        h('span', { class: 'kind-tag' }, [icon(kindIcon, kindColor), h('span', { text: kindText })]),
        e.cot ? cotChip(e.cot) : '—',
        e.duration_s ? dur(e.duration_s) : '—',
        e.device || '—'
      ];
    });
    card.body.appendChild(tableWrap(dataTable(headers, body, 'System records, newest first'), false));
    card.body.appendChild(note('A deaf spell means that cot was not being watched.' +
      (rows.length > 200 ? ' Showing the most recent 200 of ' + rows.length + '.' : '')));
    return card.root;
  }

  /* ------------------------------------------------------ overview cards */

  function renderOverviewGrid(s) {
    var grid = ids['overview-grid'];
    clear(grid);
    if (state.systemOpen) grid.appendChild(systemCard());
    grid.appendChild(dayCard(s));
    grid.appendChild(nightCard(s));
    grid.appendChild(hourCard(s));
    grid.appendChild(weekdayCard(s));
    grid.appendChild(cotCard(s));
    grid.appendChild(recentCard(s));
  }

```

```
  function dayCard(s) {
    var series = spanSeries(s.range, s.per_day, 'day');
    var total = series.reduce(function (a, x) { return a + (x.rec ? x.rec.cries : 0); }, 0);
    var card = makeCard({
      title: 'Cries by day', span2: true,
      sub: 'Every day in the range.'
    });
    if (!total) {
      card.body.appendChild(miniEmpty('No cry episodes fall in this range yet.'));
      return card.root;
    }
    var items = series.map(function (x) {
      var c = x.rec ? x.rec.cries : 0;
      return {
        label: dayShort(x.key), value: c,
        plain: dayLong(x.key) + ': ' + c + ' cries',
        tip: {
          title: dayLong(x.key),
          rows: [
            { name: 'Cries', value: num(c) },
            { name: 'Time crying', value: dur(x.rec ? x.rec.seconds : 0) },
            { name: 'Cots', value: x.rec ? cotList(x.rec.cots) : '—' }
          ]
        }
      };
    });
    var rows = series.map(function (x) {
      return [dayShort(x.key), num(x.rec ? x.rec.cries : 0),
        dur(x.rec ? x.rec.seconds : 0), x.rec ? cotList(x.rec.cots) : '—'];
    });
    chartWithTwin(card, 'day', [{ t: 'Day' }, { t: 'Cries', n: true },
      { t: 'Time crying', n: true }, { t: 'Cots' }], rows,
      'Cry episodes per day', function (host) {
        barChart(host, {
          items: items, height: 190, integer: true,
          color: function () { return v('--series-1'); },
          aria: 'Bar chart of cry episodes per day over ' + series.length + ' days, ' +
                total + ' in total. The same numbers are in the table below.',
          fmt: function (x) { return num(x) + ' cries'; }
        });
      });
    card.body.appendChild(note('A day with no bar had no uploaded cries.'));
    return card.root;
  }

  function nightCard(s) {
    var series = spanSeries(s.range, s.per_night, 'night');
    var total = series.reduce(function (a, x) { return a + (x.rec ? x.rec.cries : 0); }, 0);
    var k = s.kpis || {};
    var card = makeCard({
      title: 'Night by night', span2: true,
      sub: '20:00 to 06:00, named by the evening it starts.'
    });
    if (!total) {
      card.body.appendChild(miniEmpty('No night cries in this range.'));
      return card.root;
    }
    var items = series.map(function (x) {
      var c = x.rec ? x.rec.cries : 0;
      return {
        label: nightShort(x.key), value: c,
        plain: nightLong(x.key) + ': ' + c + ' cries',
        tip: {
          title: nightLong(x.key),
          rows: [
            { name: 'Cries', value: num(c) },
            { name: 'Time crying', value: dur(x.rec ? x.rec.seconds : 0) },
            { name: 'Cots', value: x.rec ? cotList(x.rec.cots) : '—' }
          ]
        }
      };
    });
    var rows = series.map(function (x) {
      return [nightShort(x.key), num(x.rec ? x.rec.cries : 0),
        dur(x.rec ? x.rec.seconds : 0), x.rec ? cotList(x.rec.cots) : '—'];
    });
    chartWithTwin(card, 'night', [{ t: 'Night of' }, { t: 'Cries', n: true },
      { t: 'Time crying', n: true }, { t: 'Cots' }], rows,
      'Cry episodes per night, 20:00 to 06:00', function (host) {
        barChart(host, {
          items: items, height: 190, integer: true,
          color: function () { return v('--series-1'); },
          aria: 'Bar chart of cry episodes per night over ' + series.length +
                ' nights, ' + total + ' in total. The same numbers are in the table below.',
          fmt: function (x) { return num(x) + ' cries'; }
```

```
        });
      });
    if (k.nights_affected) {
      card.body.appendChild(note(num(k.nights_affected) + ' of ' + series.length +
        ' nights had at least one cry, averaging ' + dec(k.avg_cries_per_night, 1) +
        ' episodes on those nights. Nights with no bar were quiet.'));
    }
    return card.root;
  }

  function hourCard(s) {
    var hours = s.per_hour || [];
    var total = hours.reduce(function (a, x) { return a + (x.cries || 0); }, 0);
    var card = makeCard({
      title: 'When in the day',
      sub: 'Local hour. Shaded hours are night.'
    });
    if (!total) {
      card.body.appendChild(miniEmpty('No cry episodes to place on a clock yet.'));
      return card.root;
    }
    var items = hours.map(function (x) {
      return {
        label: hourLabel(x.hour), value: x.cries,
        plain: hourLabel(x.hour) + ': ' + x.cries + ' cries',
        tip: {
          title: hourLabel(x.hour) + ' · ' + (isNightHour(x.hour) ? 'night' : 'daytime'),
          rows: [
            { name: 'Cries', value: num(x.cries) },
            { name: 'Time crying', value: dur(x.seconds) },
            { name: 'Share', value: pct(100 * x.cries / total) }
          ]
        }
      };
    });
    var rows = hours.map(function (x) {
      return [hourLabel(x.hour), isNightHour(x.hour) ? 'night' : 'day',
        num(x.cries), dur(x.seconds)];
    });
    chartWithTwin(card, 'hour', [{ t: 'Hour' }, { t: 'Part of day' },
      { t: 'Cries', n: true }, { t: 'Time crying', n: true }], rows,
      'Cry episodes by local hour of day', function (host) {
        barChart(host, {
          items: items, height: 178, integer: true, maxBar: 22,
          bands: [{ from: 0, to: NIGHT_END - 1, label: 'night' },
                  { from: NIGHT_START, to: 23, label: 'night' }],
          tick: function (i) { return i % 3 === 0 ? hourLabel(i) : null; },
          color: function (i) { return isNightHour(i) ? v('--series-1') : v('--seq-250'); },
          aria: 'Bar chart of cry episodes by hour of the day. ' + total +
                ' episodes in total. The same numbers are in the table below.',
          fmt: function (x) { return num(x) + ' cries'; }
        });
      });
    card.body.appendChild(h('div', { class: 'legend' }, [
      legendItem(v('--series-1'), 'Night (20:00–06:00)'),
      legendItem(v('--seq-250'), 'Daytime (06:00–20:00)')
    ]));
    return card.root;
  }

  function weekdayCard(s) {
    var wd = s.per_weekday || [];
    var total = wd.reduce(function (a, x) { return a + (x.cries || 0); }, 0);
    var card = makeCard({ title: 'Day of the week' });
    if (!total) {
      card.body.appendChild(miniEmpty('No cry episodes to spread across the week yet.'));
      return card.root;
    }
    var items = wd.map(function (x) {
      return {
        label: weekdayName(x.weekday).slice(0, 3), value: x.cries,
        plain: weekdayName(x.weekday) + ': ' + x.cries + ' cries',
        tip: {
          title: weekdayName(x.weekday),
          rows: [{ name: 'Cries', value: num(x.cries) },
                 { name: 'Share', value: pct(100 * x.cries / total) }]
        }
      };
    });
    var rows = wd.map(function (x) { return [weekdayName(x.weekday), num(x.cries)]; });
    chartWithTwin(card, 'weekday', [{ t: 'Weekday' }, { t: 'Cries', n: true }], rows,
      'Cry episodes by day of the week', function (host) {
        barChart(host, {
          items: items, height: 178, integer: true, maxBar: 34,
          color: function () { return v('--series-1'); },
```

```
          aria: 'Bar chart of cry episodes by weekday, ' + total +
                ' in total. The same numbers are in the table below.',
          fmt: function (x) { return num(x) + ' cries'; }
        });
      });
    return card.root;
  }

  function cotCard(s) {
    var list = s.per_cot || [];
    var card = makeCard({
      title: 'By cot', span2: true,
      sub: 'Each cot keeps its own colour.'
    });
    if (!list.length) {
      card.body.appendChild(miniEmpty('No cot has a recorded cry in this range.'));
      return card.root;
    }
    var maxCri = Math.max.apply(null, list.map(function (b) { return b.cries || 0; })) || 1;
    var headers = [{ t: 'Cot' }, { t: 'Cries', n: true }, { t: 'Share' },
      { t: 'Time crying', n: true }, { t: 'Average', n: true },
      { t: 'Longest', n: true }, { t: 'Confidence', n: true }, { t: 'Last heard' }];
    var rows = list.map(function (b) {
      var avg = b.cries ? b.seconds / b.cries : null;
      return [
        cotChip(b.cot),
        num(b.cries),
        h('span', {}, [
          h('span', { class: 'meter' }, [
            h('i', { style: { width: (100 * (b.cries || 0) / maxCri).toFixed(1) + '%' } })
          ]),
          h('span', { text: pct(100 * (b.cries || 0) / (s.kpis.total_cries || 1)) })
        ]),
        dur(b.seconds),
        dur(avg),
        dur(b.max_seconds),
        b.avg_conf === null || b.avg_conf === undefined ? '—' : dec(b.avg_conf, 1) + '%',
        b.last_at ? h('span', {}, [dateTimeAt(b.last_at),
          h('span', { class: 'sub', text: ago(b.last_at) })]) : '—'
      ];
    });
    card.body.appendChild(tableWrap(dataTable(headers, rows, 'Cry record by cot'), false));
    return card.root;
  }

  function recentCard(s) {
    var all = (s.recent || []).slice().reverse();   /* newest first */
    var card = makeCard({
      title: 'The last episodes', span2: true,
      sub: 'Newest first.'
    });
    if (!all.length) {
      card.body.appendChild(miniEmpty('No episodes have been uploaded for this range.'));
      return card.root;
    }
    var shown = state.recentAll ? all : all.slice(0, RECENT_PAGE);
    var maxRms = Math.max.apply(null, all.map(function (r) { return Number(r.peak_rms) || 0; }));

    var toggle = h('button', {
      class: 'toggle', type: 'button', 'aria-pressed': state.recentAll ? 'true' : 'false',
      text: state.recentAll ? 'Show fewer' : 'Show all ' + all.length
    });
    toggle.addEventListener('click', function () {
      state.recentAll = !state.recentAll;
      render();
    });
    card.head.appendChild(toggle);

    var headers = [{ t: 'When' }, { t: 'Cot' }, { t: 'Length', n: true },
      { t: 'Confidence', n: true }, { t: 'Loudness' }];
    var rows = shown.map(function (r) {
      var when;
      if (r.start) {
        when = h('span', {}, [
          dateTimeAt(r.start),
          h('span', { class: 'sub', text: ago(r.start) +
            (r.kind === 'test' ? ' · test record' : '') })
        ]);
      } else {
        when = h('span', {}, [
          'Time unknown',
          h('span', { class: 'sub', text: 'no clock when it happened · record #' + r.seq })
        ]);
      }
      var rms = Number(r.peak_rms);
```

```
      var loud = isFinite(rms) && maxRms > 0
        ? h('span', {}, [
            h('span', { class: 'meter' }, [
              h('i', { style: { width: (100 * rms / maxRms).toFixed(1) + '%' } })
            ]),
            h('span', { text: rms.toFixed(3) })
          ])
        : '—';
      return [
        when,
        cotChip(r.cot),
        h('span', {}, [dur(r.duration_s),
          h('span', { class: 'sub', text: (r.windows ? r.windows + ' windows' : 'record #' + r.seq) })]),
        r.peak_conf === null || r.peak_conf === undefined ? '—' : r.peak_conf + '%',
        loud
      ];
    });
    card.body.appendChild(tableWrap(dataTable(headers, rows, 'The most recent cry episodes'), false));

    var untimed = all.filter(function (r) { return !r.start; }).length;
    var foot = 'Showing ' + shown.length + ' of ' + all.length + ' episodes held on this page.';
    if (all.length >= 400) foot += ' The API returns the last 400; the charts above use every record in the range.';
    if (untimed) foot += ' ' + untimed + ' of them have no known time and are listed last.';
    card.body.appendChild(h('div', { class: 'table-foot' }, [h('span', { text: foot })]));
    return card.root;
  }

  /* ------------------------------------------------------------ patterns */

  function renderPatternsGrid(a) {
    var grid = ids['patterns-grid'];
    clear(grid);
    grid.appendChild(durationCard(a));
    grid.appendChild(gapCard(a));
    grid.appendChild(heatmapCard(a));
    grid.appendChild(cotPatternCard(a));
    grid.appendChild(nightsCard(a));
    grid.appendChild(insightCard(a));
  }

  function durationCard(a) {
    var hist = a.duration_hist || [];
    var total = hist.reduce(function (x, b) { return x + (b.count || 0); }, 0);
    var card = makeCard({
      title: 'How long the cries last',
      sub: 'Short on the left, long on the right.'
    });
    if (!total) {
      card.body.appendChild(miniEmpty('No timed episodes to measure yet.'));
      return card.root;
    }
    var items = hist.map(function (b) {
      return {
        label: b.label, value: b.count,
        plain: b.label + ': ' + b.count + ' episodes',
        tip: {
          title: 'Lasted ' + b.label,
          rows: [{ name: 'Episodes', value: num(b.count) },
                 { name: 'Share', value: pct(100 * b.count / total) }]
        }
      };
    });
    var rows = hist.map(function (b) {
      return [b.label, num(b.count), pct(100 * b.count / total)];
    });
    chartWithTwin(card, 'dur', [{ t: 'Length' }, { t: 'Episodes', n: true },
      { t: 'Share', n: true }], rows, 'Episode counts by duration bucket', function (host) {
        barChart(host, {
          items: items, height: 186, integer: true, maxBar: 34,
          color: function () { return v('--series-1'); },
          aria: 'Bar chart of episode counts by how long the cry lasted, ' + total +
                ' episodes in total. The same numbers are in the table below.',
          fmt: function (x) { return num(x) + ' episodes'; }
        });
      });
    return card.root;
  }

  function gapCard(a) {
    var g = a.gaps || {}, c = a.confidence || {};
    var card = makeCard({
      title: 'Between cries, and how sure the model was',
      sub: 'Gaps are measured between consecutive timed episodes.'
    });
    var figs = h('div', { class: 'figures' }, [
```

```
      figure('Typical gap', minutes(g.median_min),
        g.count ? 'median of ' + num(g.count) + ' gaps' : 'needs at least two timed episodes'),
      figure('Shortest gap', minutes(g.min_min), g.count ? 'closest pair' : '—'),
      figure('Longest gap', minutes(g.max_min), g.count ? 'longest quiet stretch' : '—'),
      figure('Confidence', c.median === null || c.median === undefined ? '—' : dec(c.median, 1) + '%',
        c.min === null || c.min === undefined ? 'no confidence values yet'
          : 'median · range ' + c.min + '–' + c.max + '%')
    ]);
    card.body.appendChild(figs);
    card.body.appendChild(note('Confidence is the model’s own peak score for an episode, ' +
      'not a measure of how distressed the baby was. A gap only counts between two ' +
      'episodes the hub could put a time on.'));
    return card.root;
  }

  function heatmapCard(a) {
    var cots = (a.per_cot || []).slice();
    var card = makeCard({
      title: 'Hour by cot', span2: true,
      sub: 'Every cell prints its own count.'
    });
    var max = 0;
    cots.forEach(function (c) {
      (c.by_hour || []).forEach(function (x) { if (x > max) max = x; });
    });
    if (!cots.length || !max) {
      card.body.appendChild(miniEmpty('No timed episodes to place on a clock yet.'));
      return card.root;
    }

    var rows = [];
    cots.forEach(function (c) {
      var line = [cotName(c.cot)];
      for (var hh = 0; hh < 24; hh++) line.push(num((c.by_hour || [])[hh] || 0));
      rows.push(line);
    });
    var headers = [{ t: 'Cot' }];
    for (var i = 0; i < 24; i++) headers.push({ t: hourLabel(i), n: true });
    chartWithTwin(card, 'heat', headers, rows, 'Cry episodes by cot and local hour',
      function (host) {
        registerChart(host, function () { drawHeatmap(host, cots, max); });
      });

    var ramp = h('div', { class: 'ramp' }, RAMP.map(function (name) {
      return h('i', { style: { background: v(name) } });
    }));
    card.body.appendChild(h('div', { class: 'scale-legend' }, [
      h('span', { text: '0' }), ramp, h('span', { text: num(max) + ' cries in an hour' })
    ]));
    card.body.appendChild(note('Columns run 00:00 to 23:00 in this device’s time zone. ' +
      'The hours 20:00–06:00 are the night.'));
    return card.root;
  }

  function rampLevel(value, max) {
    if (!(value > 0)) return -1;
    var idx = Math.ceil((value / max) * RAMP.length) - 1;
    return Math.max(0, Math.min(RAMP.length - 1, idx));
  }

  function drawHeatmap(host, cots, max) {
    var W = hostWidth(host);
    if (!W) return;
    clear(host);
    var darkTheme = document.documentElement.getAttribute('data-theme') === 'dark';
    var padL = 62, padR = 6, padT = 16, padB = 22;
    cots.forEach(function (c) {
      padL = Math.max(padL, cotName(c.cot).length * 6.6 + 12);
    });
    var rowH = 26, gap = 3;
    var H = padT + cots.length * rowH + padB;
    var cellW = (W - padL - padR) / 24;

    var svg = svgEl('svg', {
      viewBox: '0 0 ' + W + ' ' + H, width: W, height: H, role: 'img',
      'aria-label': 'Grid of cry counts by cot and hour of day. Peak ' + max +
                    ' in one hour. The same numbers are in the table below.'
    }, [svgEl('title', { text: 'Cry episodes by cot and local hour' })]);

    /* Mark the night columns with a written bracket, not colour alone. */
    [[0, NIGHT_END - 1], [NIGHT_START, 23]].forEach(function (span) {
      var x0 = padL + cellW * span[0], w = cellW * (span[1] - span[0] + 1);
      svg.appendChild(svgEl('line', {
        x1: x0 + 1, x2: x0 + w - 1, y1: padT - 8, y2: padT - 8,
        stroke: v('--axis'), 'stroke-width': '1'
```

```
      }));
      if (w > 40) {
        svg.appendChild(svgEl('text', {
          x: x0 + w / 2, y: padT - 11, 'text-anchor': 'middle',
          'font-size': '9.5', fill: v('--text-muted'), text: 'night'
        }));
      }
    });

    cots.forEach(function (c, r) {
      var y = padT + r * rowH;
      svg.appendChild(svgEl('text', {
        x: padL - 8, y: y + rowH / 2 + 3.5, 'text-anchor': 'end',
        'font-size': '11.5', fill: v('--text-primary'), text: cotName(c.cot)
      }));
      for (var hh = 0; hh < 24; hh++) {
        var val = (c.by_hour || [])[hh] || 0;
        var lvl = rampLevel(val, max);
        var x = padL + cellW * hh;
        var fill = lvl < 0 ? v('--surface-2') : v(RAMP[lvl]);
        svg.appendChild(svgEl('rect', {
          x: x + 0.5, y: y + 0.5, width: Math.max(1, cellW - 1), height: rowH - gap,
          fill: fill, stroke: v('--grid'), 'stroke-width': '0.5', rx: 2
        }));
        if (val > 0 && cellW >= 17) {
          /* The number wears ink, not the cell's colour: white on the deep
           * end of the light ramp, near-black on the bright end of the dark
           * ramp.  The dark ramp runs the other way, so its switch sits one
           * step earlier. */
          var hi = lvl >= (darkTheme ? 3 : 4);
          svg.appendChild(svgEl('text', {
            x: x + cellW / 2, y: y + rowH / 2 + 3, 'text-anchor': 'middle',
            'font-size': '10.5', 'font-weight': '600',
            class: hi ? 'heat-hi' : 'heat-lo', text: String(val)
          }));
        }
        var hit = svgEl('rect', {
          x: x, y: y, width: cellW, height: rowH - gap + 1,
          fill: 'transparent', 'pointer-events': 'all'
        }, [svgEl('title', { text: cotName(c.cot) + ' at ' + hourLabel(hh) + ': ' + val + ' cries' })]);
        (function (cot, hour, value, cotTotal) {
          tipify(hit, function () {
            return {
              title: cotName(cot) + ' · ' + hourLabel(hour) +
                     ' · ' + (isNightHour(hour) ? 'night' : 'daytime'),
              rows: [
                { name: 'Cries', value: num(value), color: cotColor(cot) },
                { name: 'Of this cot', value: cotTotal ? pct(100 * value / cotTotal) : '—' }
              ]
            };
          });
        })(c.cot, hh, val, c.cries);
        svg.appendChild(hit);
      }
    });

    for (var hh2 = 0; hh2 < 24; hh2 += 3) {
      svg.appendChild(svgEl('text', {
        x: padL + cellW * hh2 + cellW / 2, y: H - 6, 'text-anchor': 'middle',
        'font-size': '10.5', fill: v('--text-muted'), text: hourLabel(hh2)
      }));
    }
    host.appendChild(svg);
  }

  function cotPatternCard(a) {
    var list = a.per_cot || [];
    var card = makeCard({
      title: 'Each cot's pattern', span2: true,
      sub: 'When each cot is usually heard.'
    });
    if (!list.length) {
      card.body.appendChild(miniEmpty('No cot has enough records for a pattern yet.'));
      return card.root;
    }
    var headers = [{ t: 'Cot' }, { t: 'Cries', n: true }, { t: 'Median', n: true },
      { t: 'Longest', n: true }, { t: 'Usual hour' }, { t: 'At night' }];
    var rows = list.map(function (c) {
      var share = c.cries ? 100 * c.night_cries / c.cries : 0;
      return [
        cotChip(c.cot),
        num(c.cries),
        dur(c.median_seconds),
        dur(c.max_seconds),
        h('span', {}, [hourLabel(c.peak_hour),
```

```
          h('span', { class: 'sub', text: pct(c.peak_share) + ' of its cries' })]),
        h('span', {}, [num(c.night_cries) + ' of ' + num(c.cries),
          h('span', { class: 'sub', text: pct(share) + ' between 20:00 and 06:00' })])
      ];
    });
    card.body.appendChild(tableWrap(dataTable(headers, rows, 'Pattern by cot'), false));
    return card.root;
  }

  function nightsCard(a) {
    var list = (a.nights || []).slice().reverse();
    var card = makeCard({
      title: 'The nights', span2: true,
      sub: 'Newest first.'
    });
    if (!list.length) {
      card.body.appendChild(miniEmpty('No night has a recorded cry in this range.'));
      return card.root;
    }
    var shown = state.nightsAll ? list : list.slice(0, NIGHTS_PAGE);
    var toggle = h('button', {
      class: 'toggle', type: 'button', 'aria-pressed': state.nightsAll ? 'true' : 'false',
      text: state.nightsAll ? 'Show fewer' : 'Show all ' + list.length
    });
    toggle.addEventListener('click', function () {
      state.nightsAll = !state.nightsAll;
      render();
    });
    card.head.appendChild(toggle);

    var headers = [{ t: 'Night of' }, { t: 'Cries', n: true }, { t: 'Time crying', n: true },
      { t: 'First cry' }, { t: 'Last cry' }, { t: 'Cots' }];
    var rows = shown.map(function (nn) {
      return [
        nightShort(nn.night),
        num(nn.cries),
        dur(nn.seconds),
        clockAt(nn.first),
        clockAt(nn.last),
        cotList(nn.cots)
      ];
    });
    card.body.appendChild(tableWrap(dataTable(headers, rows, 'Cry record night by night'), false));
    card.body.appendChild(h('div', { class: 'table-foot' }, [
      h('span', { text: 'Showing ' + shown.length + ' of ' + list.length +
        ' nights with at least one cry. A night that is not listed was quiet.' })
    ]));
    return card.root;
  }

  function insightCard(a) {
    var list = a.insights || [];
    var card = makeCard({
      title: 'What the record says', span2: true,
      sub: 'From the numbers above.'
    });
    if (!list.length) {
      card.body.appendChild(miniEmpty('Nothing to report yet.'));
      return card.root;
    }
    card.body.appendChild(h('ul', { class: 'insights' }, list.map(function (line) {
      return h('li', {}, [h('span', { class: 'bullet' }), h('span', { text: String(line) })]);
    })));
    return card.root;
  }

  /* ---------------------------------------------------------------- chrome */

  function renderControls() {
    var sel = ids.cot;
    var want = ['0'].concat(state.knownCots.map(String));
    var have = Array.prototype.map.call(sel.options, function (o) { return o.value; });
    if (want.join('|') !== have.join('|')) {
      clear(sel);
      sel.appendChild(h('option', { value: '0', text: 'All cots' }));
      state.knownCots.forEach(function (c) {
        sel.appendChild(h('option', { value: String(c), text: cotName(c) }));
      });
    }
    sel.value = String(state.cot);
    if (sel.value !== String(state.cot)) { state.cot = 0; sel.value = '0'; }

    var ds = ids.device;
    var dwant = [''].concat(state.devices.map(function (d) { return d.device; }));
    var dhave = Array.prototype.map.call(ds.options, function (o) { return o.value; });
```

```
  if (dwant.join('|') !== dhave.join('|')) {
    clear(ds);
    ds.appendChild(h('option', { value: '', text: 'All hubs' }));
    state.devices.forEach(function (d) {
      ds.appendChild(h('option', { value: d.device, text: hubName(d) }));
    });
  }
  ds.value = state.device;
  if (ds.value !== state.device) { state.device = ''; ds.value = ''; }
  show(ids['device-control'], state.devices.length > 1);

  Array.prototype.forEach.call(ids.range.querySelectorAll('button'), function (b) {
    b.setAttribute('aria-pressed',
      Number(b.getAttribute('data-days')) === state.days ? 'true' : 'false');
  });
}

function currentHub() {
  if (state.device) {
    for (var i = 0; i < state.devices.length; i++) {
      if (state.devices[i].device === state.device) return state.devices[i];
    }
  }
  if (state.devices.length === 1) return state.devices[0];
  return (state.summary && state.summary.device) || state.devices[0] || null;
}

/* ------------------------------------------------------------------ presence */

/* Who is here.
 *
 * The hub reports in once a minute whether or not there is anything to
 * record, so this is what answers the question a caregiver opens the
 * dashboard with after a quiet week: was the baby quiet, or did the
 * hub stop?  A record cannot answer that -- it only says what was
 * heard. */
function presenceHub() {
  var p = state.presence;
  if (!p || !p.hubs || !p.hubs.length) return null;
  var dev = state.device;
  if (!dev) {
    var cur = currentHub();
    dev = cur && cur.device ? cur.device : null;
  }
  if (dev) {
    for (var i = 0; i < p.hubs.length; i++) {
      if (p.hubs[i].device === dev) return p.hubs[i];
    }
  }
  return p.hubs[0];
}

function presenceNodes(device) {
  var p = state.presence;
  if (!p || !p.nodes) return [];
  return p.nodes.filter(function (n) {
    return !device || n.device === device;
  });
}

function presenceRow(name, word, dotState, detail) {
  return h('span', { class: 'presence-chip', 'data-state': dotState }, [
    h('span', { class: 'presence-dot', 'aria-hidden': 'true' }),
    h('span', { class: 'p-name', text: name }),
    h('span', { class: 'p-word', text: word }),
    detail ? h('span', { class: 'p-detail', text: detail }) : null
  ]);
}

/* Who is here, as a strip under the numbers.
 *
 * The hub reports in once a minute whether or not there is anything
 * to record, so this is what answers the question a caregiver opens
 * the dashboard with after a quiet week: was the baby quiet, or did
 * the hub stop?  A record cannot answer that -- it only says what
 * was heard. */
function renderPresence() {
  var el = ids['presence-strip'];
  if (!el) return;
  clear(el);
  var ph = presenceHub();
  if (!ph) {
    show(el, false);
    return;
  }
```

```
    el.appendChild(h('span', { class: 'strip-label', text: 'Here' }));

    var hubBits = [];
    if (ph.fw && ph.fw !== 'demo') hubBits.push('firmware ' + ph.fw);
    if (ph.online) {
      if (!ph.demo) hubBits.push('reported ' + ago(ph.at));
    } else {
      hubBits.push('last report ' + ago(ph.at));
    }
    el.appendChild(presenceRow(
      hubName(ph), ph.online ? 'online' : 'offline',
      ph.online ? 'on' : 'off', hubBits.join(' · ')));

    presenceNodes(ph.device).forEach(function (n) {
      var word, dot;
      if (!n.online) {
        word = 'offline';
        dot = 'off';
      } else if (n.deaf) {
        word = 'not heard';
        dot = 'deaf';
      } else {
        word = 'online';
        dot = 'on';
      }
      var bits = [];
      if (n.online && n.rssi) bits.push(n.rssi + ' dBm');
      if (n.deaf) bits.push('the hub cannot get a window from it');
      if (!n.online) bits.push('not answering');
      el.appendChild(presenceRow(cotName(n.cot), word, dot, bits.join(' · ')));
    });

    show(el, true);
  }

  function renderChrome() {
    var s = state.summary;
    var days = (s && s.range && s.range.days) || state.days;
    var hub = currentHub();

    /* hubmeta */
    var meta, dotState = 'none';
    var ph = presenceHub();
    if (ph) {
      dotState = ph.online ? 'on' : 'off';
      var bits = [hubName(ph), ph.online ? 'online' : 'offline'];
      var pnodes = presenceNodes(ph.device);
      if (pnodes.length) {
        var up = pnodes.filter(function (n) { return n.online; }).length;
        bits.push(up === pnodes.length
          ? pnodes.length + (pnodes.length === 1 ? ' cot online' : ' cots online')
          : up + ' of ' + pnodes.length + ' cots online');
      }
      meta = bits.join(' · ');
    } else if (!state.devices.length) {
      meta = 'No hub has uploaded to this dashboard yet.';
    } else if (state.devices.length > 1 && !state.device) {
      var newest = state.devices.reduce(function (a, d) {
        return (d.last_seen || 0) > (a.last_seen || 0) ? d : a;
      }, state.devices[0]);
      meta = state.devices.length + ' hubs have uploaded · last upload ' + ago(newest.last_seen);
    } else {
      meta = hubName(hub) + (hub && hub.fw ? ' · firmware ' + hub.fw : '') +
        ' · last upload ' + ago(hub && hub.last_seen);
    }
    text(ids['hubmeta-text'], meta);
    ids.hubmeta.setAttribute('data-state', dotState);
    if (hub && hub.device) ids.hubmeta.setAttribute('title', hub.device);
    else ids.hubmeta.removeAttribute('title');

    /* footer */
    if (s && s.range) {
      text(ids['footer-range'], 'Last ' + days + ' days · ' +
        dateAt(s.range.from) + ' to ' + dateAt(s.range.to) + ' · ' +
        (state.cot ? cotName(state.cot) : 'all cots'));
    } else {
      text(ids['footer-range'], 'Last ' + days + ' days');
    }
    text(ids['footer-updated'], state.lastGood
      ? 'Updated ' + clockAt(Math.round(state.lastGood / 1000)) + ' · checks every 30 s'
      : 'Waiting for the first read · checks every 30 s');

    /* scope note */
    var parts = [];
```

```
    parts.push('The last ' + days + ' days, ' +
      (state.cot ? cotName(state.cot) + ' only' : 'every cot') +
      '. A night is 20:00–06:00, named by the evening it starts.');

    if (state.devices.length > 1) {
      var line = 'The totals and charts cover every hub.';
      if (state.counts) {
        var bits = state.devices.map(function (d) {
          return hubName(d) + ' ' + num(state.counts.counts[d.device] || 0);
        });
        line += ' Timed records in this range: ' + bits.join(', ') +
          (state.counts.capped ? ' (most recent 2000)' : '') + '.';
      }
      if (state.device) line += ' "' + hubName(hub) + '" is selected above.';
      parts.push(line);
    }

    var deaf = deafNow();
    if (deaf.length) {
      parts.push(deaf.map(function (d) {
        return cotName(d.cot) + ' has not been heard since ' + clockAt(d.since) +
          ' (' + ago(d.since) + ')';
      }).join('; ') + ' — that cot was not being watched.');
    }

    text(ids['scope-note'], parts.join(' '));
    show(ids['scope-note'], true);
  }

  function setEmptyDetail(node, msg) {
    if (!node) return;
    if (!pristine[node.id || node.className]) pristine[node.id || node.className] = node.textContent;
    node.textContent = msg || pristine[node.id || node.className];
  }

  function renderOverview() {
    var s = state.summary;
    var empty = ids['empty-overview'];
    var body = ids['overview-body'];
    var emptyHead = empty.querySelector('h2');
    if (!pristine.head) pristine.head = emptyHead ? emptyHead.textContent : '';

    if (!s) {
      if (emptyHead) emptyHead.textContent = 'Cannot reach the dashboard service';
      setEmptyDetail(ids['empty-overview-detail'],
        'The page could not read the record from the service. It keeps trying every 30 seconds ' +
        'and will fill in by itself once the service answers.');
      show(empty, true);
      show(body, false);
      return;
    }

    var hasCries = (s.kpis && s.kpis.total_cries > 0) || (s.recent || []).length > 0;
    var hasSystem = state.system.length > 0;

    if (!hasCries && !hasSystem) {
      if (emptyHead) emptyHead.textContent = pristine.head;
      var hub = currentHub();
      if (hub) {
        setEmptyDetail(ids['empty-overview-detail'],
          hubName(hub) + ' last uploaded ' + ago(hub.last_seen) + ', but no cry episodes fall ' +
          'in the last ' + ((s.range && s.range.days) || state.days) + ' days' +
          (state.cot ? ' for ' + cotName(state.cot) : '') +
          '. Try a longer range, or another cot.');
      } else {
        setEmptyDetail(ids['empty-overview-detail'], null);
      }
      show(empty, true);
      show(body, false);
      return;
    }

    if (emptyHead) emptyHead.textContent = pristine.head;
    show(empty, false);
    show(body, true);
    renderKpis(s);
    renderUntimed(s);
    renderSystemStrip();
    renderOverviewGrid(s);
  }

  function renderPatterns() {
    var a = state.analytics;
    var empty = ids['empty-patterns'];
    var body = ids['patterns-body'];
```

```
  var emptyHead = empty.querySelector('h2');
  var emptyP = empty.querySelector('p');
  if (!pristine.phead) pristine.phead = emptyHead ? emptyHead.textContent : '';
  if (!pristine.pp) pristine.pp = emptyP ? emptyP.textContent : '';

  var total = a ? (a.duration_hist || []).reduce(function (x, b) { return x + (b.count || 0); }, 0) : 0;

  if (!a || !total) {
    if (emptyHead) emptyHead.textContent = a ? pristine.phead : 'Patterns could not be read';
    if (emptyP) {
      emptyP.textContent = a ? pristine.pp
        : 'The dashboard service did not answer the patterns request. The page keeps trying ' +
          'every 30 seconds. The Overview is unaffected.';
    }
    show(empty, true);
    show(body, false);
    return;
  }

  if (emptyHead) emptyHead.textContent = pristine.phead;
  if (emptyP) emptyP.textContent = pristine.pp;
  show(empty, false);
  show(body, true);

  var days = (a.range && a.range.days) || state.days;
  var scopeParts = ['Patterns cover the last ' + days + ' days across every cot and every hub.'];
  if (state.cot) {
    scopeParts.push('The cot filter above applies to the Overview only — ' +
      'the patterns API has no cot filter, so ' + cotName(state.cot) +
      ' is not singled out here.');
  }
  var s = state.summary;
  if (s && s.kpis && total !== s.kpis.total_cries) {
    scopeParts.push('This tab counts ' + num(total) + ' episodes against the Overview’s ' +
      num(s.kpis.total_cries) + ': patterns include the hub’s test records and ignore ' +
      'the cot filter, while the Overview counts cries only.');
  }
  scopeParts.push('Episodes the hub could not put a time on are left out of the patterns entirely.');
  text(ids['patterns-scope'], scopeParts.join(' '));
  show(ids['patterns-scope'], true);

  renderPatternsGrid(a);
}

function render() {
  tipHide();
  charts = [];
  renderControls();
  renderChrome();
  /* Who is here is drawn from the presence alone, so it is drawn
   * whatever the record has to show -- including nothing at all,
   * which is when the caregiver most wants to know. */
  renderPresence();
  renderOverview();
  renderPatterns();
  drawCharts();
}

/* ------------------------------------------------------------------ tabs */

function setTab(name) {
  state.tab = name;
  var over = name === 'overview';
  ids['tab-overview'].setAttribute('aria-selected', over ? 'true' : 'false');
  ids['tab-patterns'].setAttribute('aria-selected', over ? 'false' : 'true');
  show(ids['view-overview'], over);
  show(ids['view-patterns'], !over);
  tipHide();
  drawCharts();
}

/* ------------------------------------------------------------------ boot */

function boot() {
  ID_LIST.forEach(function (id) { ids[id] = document.getElementById(id); });
  if (!ids.kpis || !ids['overview-grid']) return;

  ids['tab-overview'].addEventListener('click', function () { setTab('overview'); });
  ids['tab-patterns'].addEventListener('click', function () { setTab('patterns'); });

  var tabs = [ids['tab-overview'], ids['tab-patterns']];
  tabs.forEach(function (btn, i) {
    btn.addEventListener('keydown', function (ev) {
      var k = ev.key, next = -1;
      if (k === 'ArrowRight') next = (i + 1) % tabs.length;
```

```
      else if (k === 'ArrowLeft') next = (i - 1 + tabs.length) % tabs.length;
      else if (k === 'Home') next = 0;
      else if (k === 'End') next = tabs.length - 1;
      if (next < 0) return;
      ev.preventDefault();
      tabs[next].focus();
      setTab(next === 0 ? 'overview' : 'patterns');
    });
  });

  /* The theme.  The inline script in <head> has already stamped the
   * saved choice (or the OS setting) before first paint; this keeps the
   * button's label honest, remembers a deliberate choice, and follows
   * the OS live while no choice has been made.  Charts are redrawn so
   * the heatmap's cell numbers pick the right ink for the new ramp. */
  function themeName() {
    return document.documentElement.getAttribute('data-theme') === 'dark' ? 'dark' : 'light';
  }
  function themeLabel() {
    if (ids.theme) {
      ids.theme.setAttribute('aria-label',
        themeName() === 'dark' ? 'Switch to light theme' : 'Switch to dark theme');
    }
  }
  themeLabel();
  if (ids.theme) {
    ids.theme.addEventListener('click', function () {
      var next = themeName() === 'dark' ? 'light' : 'dark';
      document.documentElement.setAttribute('data-theme', next);
      try { localStorage.setItem('crynet-theme', next); } catch (e) { /* private mode */ }
      themeLabel();
      drawCharts();
    });
  }
  if (window.matchMedia) {
    var mq = window.matchMedia('(prefers-color-scheme: dark)');
    var followOS = function () {
      var stored = null;
      try { stored = localStorage.getItem('crynet-theme'); } catch (e) { /* keep null */ }
      if (stored === 'light' || stored === 'dark') return;
      document.documentElement.setAttribute('data-theme', mq.matches ? 'dark' : 'light');
      themeLabel();
      drawCharts();
    };
    if (mq.addEventListener) mq.addEventListener('change', followOS);
  }

  ids.range.addEventListener('click', function (ev) {
    var btn = ev.target.closest ? ev.target.closest('button[data-days]') : null;
    if (!btn) return;
    var d = Number(btn.getAttribute('data-days'));
    if (!isFinite(d) || d === state.days) return;
    state.days = d;
    state.recentAll = false;
    state.nightsAll = false;
    renderControls();
    refresh();
  });

  ids.cot.addEventListener('change', function () {
    state.cot = Number(ids.cot.value) || 0;
    state.recentAll = false;
    refresh();
  });

  ids.device.addEventListener('change', function () {
    state.device = ids.device.value || '';
    render();
  });

  var resizeTimer = null;
  window.addEventListener('resize', function () {
    if (resizeTimer) clearTimeout(resizeTimer);
    resizeTimer = setTimeout(function () { tipHide(); drawCharts(); }, 160);
  });
  window.addEventListener('scroll', tipHide, true);

  document.addEventListener('visibilitychange', function () {
    if (document.hidden) return;
    /* Coming back to the tab is the moment the caregiver most wants
     * an answer, so the presence is asked for at once -- a browser
     * may have throttled the timers while the tab was in the
     * background. */
    refreshPresence();
    if (Date.now() - state.lastGood > POLL_MS) refresh();
```

```
      else drawCharts();
    });

    /* Nothing may escape as an unhandled rejection. */
    window.addEventListener('unhandledrejection', function (ev) {
      ev.preventDefault();
      setConn('stale', 'Offline · retrying every 30 s');
    });

    setTab('overview');
    setConn('loading', 'Connecting');
    refresh();
    refreshPresence();
  }

  if (document.readyState === 'loading') {
    document.addEventListener('DOMContentLoaded', boot);
  } else {
    boot();
  }
})();
```

dashboard/worker/index.js

743 lines · 28,507 bytes

```
/* CryNet dashboard — the hosted copy, as a Cloudflare Worker.
 *
 * This is a port of dashboard/server.py, which stays the offline copy a
 * caregiver can run with `python dashboard/server.py` and nothing else.
 * The two must agree field for field: the hub uploads the body that
 * firmware/tools/test_dashboard_contract.py builds, and the page in web/
 * renders whatever /api/v1 returns.  Where this copy differs from the
 * Python one it says why in a comment.
 *
 * The differences that matter:
 *   - Authentication covers every GET, not just the uploads.  On the
 *     public internet an open /api/v1/summary would publish the baby's
 *     whole cry record to anyone who guesses the URL.
 *   - Days, hours and nights are bucketed in the timezone the browser
 *     sends (?tz=Africa/Lagos).  This Worker runs in UTC; the caregiver
 *     does not.
 *   - The untimed-records window is split into two index-friendly halves
 *     instead of one CASE expression, so D1's 5M rows-read/day budget is
 *     spent on the window, not on the whole table.
 */

const NIGHT_START_HOUR = 20;
const NIGHT_END_HOUR = 6;
const DURATION_BUCKETS = [5, 15, 30, 60, 120, 300, 600];

/* ------------------------------------------------------------------ helpers */

/* Python's round() sends halves to the even side; Math.round sends them
   up.  The JSON the page eats should not depend on which copy served it. */
function pyRound(x) {
  const f = Math.floor(x);
  const d = x - f;
  if (d > 0.5) return f + 1;
  if (d < 0.5) return f;
  return f % 2 === 0 ? f : f + 1;
}
const r1 = (x) => pyRound(x * 10) / 10;
const r2 = (x) => pyRound(x * 100) / 100;

/* statistics.median(): the mean of the two middle values when even. */
function median(xs) {
  if (!xs.length) return null;
  const s = [...xs].sort((a, b) => a - b);
  const m = s.length >> 1;
  return s.length % 2 ? s[m] : (s[m - 1] + s[m]) / 2;
}

/* The hub's JSON values go through the same leniency as server.py's
   _int_or_none / _float_or_none: null and "" become NULL, anything that
   is not a number becomes NULL, a float that is a whole number stays a
   number (SQLite stores it fine in an INTEGER column). */
function toInt(v) {
  if (v === null || v === undefined || v === '') return null;
  const n = Number(v);
  return Number.isFinite(n) ? Math.trunc(n) : null;
}
```

```
function toFloat(v) {
  if (v === null || v === undefined || v === '') return null;
  const n = Number(v);
  return Number.isFinite(n) ? n : null;
}

function intParam(v, d) {
  const n = parseInt(v, 10);
  return Number.isFinite(n) ? n : d;
}

function json(obj, status = 200, extra = {}) {
  return new Response(JSON.stringify(obj), {
    status,
    headers: {
      'Content-Type': 'application/json',
      'Cache-Control': 'no-store',
      ...extra,
    },
  });
}

/* ------------------------------------------------------------ timezone */

const WEEKDAY = { Mon: 0, Tue: 1, Wed: 2, Thu: 3, Fri: 4, Sat: 5, Sun: 6 };

/* Mirrors make_local() in server.py: the zone comes from the request,
   anything unknown falls back to UTC — which is where this runs.  The
   returned object carries exactly what the analytics need: an ISO date,
   the hour of day, and Python's weekday number (Monday = 0). */
function makeLocal(tz) {
  let dtf;
  try {
    dtf = new Intl.DateTimeFormat('en-CA', {
      timeZone: tz || 'UTC',
      year: 'numeric', month: '2-digit', day: '2-digit',
      hour: '2-digit', hourCycle: 'h23', weekday: 'short',
    });
  } catch (e) {
    return makeLocal('');
  }
  return (ts) => {
    const p = {};
    for (const part of dtf.formatToParts(new Date(ts * 1000))) p[part.type] = part.value;
    return {
      date: `${p.year}-${p.month}-${p.day}`,
      hour: Number(p.hour) % 24,
      weekday: WEEKDAY[p.weekday] ?? 0,
    };
  };
}

function dayBefore(iso) {
  const [y, m, d] = iso.split('-').map(Number);
  return new Date(Date.UTC(y, m - 1, d) - 86400000).toISOString().slice(0, 10);
}

function nightKey(ts, lt) {
  const l = lt(ts);
  if (l.hour >= NIGHT_START_HOUR) return l.date;
  if (l.hour < NIGHT_END_HOUR) return dayBefore(l.date);
  return l.date;
}

/* ------------------------------------------------------------ storage */

async function getDevices(db) {
  const r = await db.prepare('SELECT * FROM devices ORDER BY last_seen DESC').all();
  return r.results || [];
}

/* Mirrors Store.events().  A record with start_at 0 (or NULL) is one the
   hub made before it knew the time; it counts as evidence and is windowed
   by received_at.  server.py folds that into one CASE expression; here it
   is two ranges so each can use its own index. */
async function fetchEvents(db, { since = 0, until = 0, cot = 0, kind = '', limit = 500 }) {
  let sql = 'SELECT * FROM events WHERE ((start_at > 0 AND start_at >= ?';
  const a = [since];
  if (until) { sql += ' AND start_at <= ?'; a.push(until); }
  sql += ') OR ((start_at IS NULL OR start_at = 0) AND received_at >= ?';
  a.push(since);
  if (until) { sql += ' AND received_at <= ?'; a.push(until); }
  sql += '))';
  if (cot) { sql += ' AND cot = ?'; a.push(cot); }
  if (kind) { sql += ' AND kind = ?'; a.push(kind); }
```

```
  sql += ' ORDER BY (CASE WHEN start_at > 0 THEN start_at ELSE received_at END) DESC, seq DESC LIMIT ?';
  a.push(limit);
  const r = await db.prepare(sql).bind(...a).all();
  return r.results || [];
}

/* Mirrors Store.all_cry_events(): every cry, oldest first, timed only. */
async function allCryEvents(db, since, until) {
  let sql = "SELECT * FROM events WHERE kind IN ('cry','test')";
  const a = [];
  if (since) { sql += ' AND start_at >= ?'; a.push(since); }
  if (until) { sql += ' AND start_at <= ?'; a.push(until); }
  sql += ' ORDER BY start_at ASC';
  const r = await db.prepare(sql).bind(...a).all();
  return r.results || [];
}

/* The untimed count server.py gets by scanning the window in Python; a
   COUNT here reads the same rows without shipping them. */
async function countUntimed(db, since, until) {
  let sql = "SELECT COUNT(*) AS n FROM events WHERE kind='cry' AND coalesce(start_at,0)=0 AND received_at >= ?";
  const a = [since];
  if (until) { sql += ' AND received_at <= ?'; a.push(until); }
  const r = await db.prepare(sql).bind(...a).first();
  return r ? r.n : 0;
}

/* ------------------------------------------------------------- summary */

async function buildSummary(db, days, cot, tz) {
  const lt = makeLocal(tz);
  const now = Math.floor(Date.now() / 1000);
  const since = now - days * 86400;
  let events = await fetchEvents(db, { since, until: now + 86400, cot, kind: 'cry', limit: 100000 });
  events = events.filter((e) => e.kind === 'cry' || e.kind === 'test');
  events.sort((a, b) => (a.start_at || 0) - (b.start_at || 0));

  const durations = events.map((e) => Math.trunc(e.duration_s || 0));
  const totalS = durations.reduce((s, d) => s + d, 0);

  const perCot = new Map();
  const perDay = new Map();
  const perHour = new Array(24).fill(0);
  const perHourS = new Array(24).fill(0);
  const perWeekday = new Array(7).fill(0);
  const nights = new Map();
  const heatCounts = new Map();
  const heatSeconds = new Map();

  let untimed = 0;
  for (const e of events) {
    const ts = Math.trunc(e.start_at || 0);
    const c = Math.trunc(e.cot || 0);
    const dur = Math.trunc(e.duration_s || 0);

    // Which cot cried, how often and how loudly are facts whether or not
    // the hub knew the date, so they are counted for every record.
    // Only last_at needs a time.
    let b = perCot.get(c);
    if (!b) {
      b = { cot: c, cries: 0, seconds: 0, max_seconds: 0, last_at: 0, confidences: [] };
      perCot.set(c, b);
    }
    b.cries += 1;
    b.seconds += dur;
    b.max_seconds = Math.max(b.max_seconds, dur);
    if (e.peak_conf !== null && e.peak_conf !== undefined) b.confidences.push(Math.trunc(e.peak_conf));
    if (!ts) { untimed += 1; continue; }
    b.last_at = Math.max(b.last_at, ts);

    const local = lt(ts);
    const dkey = local.date;
    let day = perDay.get(dkey);
    if (!day) { day = { day: dkey, cries: 0, seconds: 0, cots: new Set() }; perDay.set(dkey, day); }
    day.cries += 1;
    day.seconds += dur;
    if (c) day.cots.add(c);

    perHour[local.hour] += 1;
    perHourS[local.hour] += dur;
    perWeekday[local.weekday] += 1;

    const nk = nightKey(ts, lt);
    let night = nights.get(nk);
    if (!night) { night = { night: nk, cries: 0, seconds: 0, cots: new Set() }; nights.set(nk, night); }
```

```
    night.cries += 1;
    night.seconds += dur;
    if (c) night.cots.add(c);

    if (c) {
      if (!heatCounts.has(c)) { heatCounts.set(c, new Array(24).fill(0)); heatSeconds.set(c, new Array(24).fill(0)); }
      heatCounts.get(c)[local.hour] += 1;
      heatSeconds.get(c)[local.hour] += dur;
    }
  }

  for (const b of perDay.values()) b.cots = [...b.cots].sort((x, y) => x - y);
  for (const b of nights.values()) b.cots = [...b.cots].sort((x, y) => x - y);

  const nightList = [...nights.values()].sort((a, b) => (a.night < b.night ? -1 : a.night > b.night ? 1 : 0));

  const gaps = [];
  for (let i = 1; i < events.length; i++) {
    const a = events[i - 1];
    const b = events[i];
    if (a.start_at && b.start_at) gaps.push(b.start_at - a.start_at);
  }

  const cotsSorted = [...perCot.values()].sort((a, b) => a.cot - b.cot);
  for (const b of cotsSorted) {
    const confs = b.confidences;
    delete b.confidences;
    b.avg_conf = confs.length ? r1(confs.reduce((s, x) => s + x, 0) / confs.length) : null;
  }

  const heatCots = [...heatCounts.keys()].sort((a, b) => a - b);
  const devices = await getDevices(db);
  const lastCryAt = events.reduce((m, e) => Math.max(m, Math.trunc(e.start_at || 0)), 0);

  return {
    device: devices.length ? devices[0] : null,
    devices,
    range: { from: since, to: now, days },
    kpis: {
      total_cries: events.length,
      timed_cries: events.length - untimed,
      untimed_cries: untimed,
      cries_per_day: days ? r2(events.length / days) : 0,
      total_cry_seconds: totalS,
      avg_cry_seconds: events.length ? r1(totalS / events.length) : 0,
      longest_cry_seconds: durations.length ? Math.max(...durations) : 0,
      median_cry_seconds: durations.length ? r1(median(durations)) : 0,
      cots_crying: perCot.size,
      nights_affected: nights.size,
      nights_per_week: days >= 7 ? r2(nights.size / (days / 7.0)) : 0,
      // A per-night average must divide like by like: a record with no
      // time cannot be attributed to a night.
      avg_cries_per_night: nights.size ? r2((events.length - untimed) / nights.size) : 0,
      last_cry_at: lastCryAt,
      median_gap_min: gaps.length ? r1(median(gaps) / 60.0) : null,
    },
    per_cot: cotsSorted,
    per_day: [...perDay.keys()].sort().map((k) => perDay.get(k)),
    per_night: nightList,
    per_hour: perHour.map((n, h) => ({ hour: h, cries: n, seconds: perHourS[h] })),
    per_weekday: perWeekday.map((n, w) => ({ weekday: w, cries: n })),
    heatmap: {
      cots: heatCots,
      counts: heatCots.map((c) => heatCounts.get(c)),
      seconds: heatCots.map((c) => heatSeconds.get(c)),
    },
    recent: events.slice(-400).map((e) => ({
      seq: e.seq, cot: e.cot, kind: e.kind,
      start: e.start_at, end: e.end_at,
      duration_s: e.duration_s, peak_conf: e.peak_conf,
      windows: e.windows, peak_rms: e.peak_rms,
      mean_rms: e.mean_rms,
    })),
  };
}

/* ------------------------------------------------------------ analytics */

function bucketPairs(edges) {
  const pairs = [];
  let lo = 0;
  for (const hi of edges) { pairs.push([lo, hi]); lo = hi; }
  pairs.push([lo, 1e9]);
  return pairs;
}
```

```

function bucketLabel(lo, hi) {
  if (hi >= 1e9) return `${Math.round(lo / 60)}+ min`;
  return hi - lo <= 60 ? `${lo}-${hi}s` : `${Math.floor(lo / 60)}-${Math.floor(hi / 60)} min`;
}

async function buildAnalytics(db, days, tz) {
  const lt = makeLocal(tz);
  const now = Math.floor(Date.now() / 1000);
  const since = now - days * 86400;
  const events = (await allCryEvents(db, since, now + 86400)).filter((e) => e.start_at);
  const untimed = await countUntimed(db, since, now + 86400);

  const durations = events.map((e) => Math.trunc(e.duration_s || 0));
  const confs = events
    .filter((e) => e.peak_conf !== null && e.peak_conf !== undefined)
    .map((e) => Math.trunc(e.peak_conf));

  const pairs = bucketPairs(DURATION_BUCKETS);
  const hist = pairs.map(([lo, hi]) => ({ label: bucketLabel(lo, hi), count: 0 }));
  for (const d of durations) {
    for (let i = 0; i < pairs.length; i++) {
      if (d >= pairs[i][0] && d < pairs[i][1]) { hist[i].count += 1; break; }
    }
  }

  const gaps = [];
  for (let i = 1; i < events.length; i++) gaps.push(events[i].start_at - events[i - 1].start_at);

  const nightsMap = new Map();
  for (const e of events) {
    const k = nightKey(Math.trunc(e.start_at), lt);
    if (!nightsMap.has(k)) nightsMap.set(k, []);
    nightsMap.get(k).push(e);
  }

  // The hour each cot is most often heard in, and how sharp that peak is
  // -- a caregiver wants to know "cot 2 always wakes at 03:00".
  const perCot = [];
  const cots = [...new Set(events.map((e) => Math.trunc(e.cot || 0)))].sort((a, b) => a - b);
  for (const eCot of cots) {
    const sub = events.filter((e) => Math.trunc(e.cot || 0) === eCot);
    const byHour = new Array(24).fill(0);
    for (const e of sub) byHour[lt(Math.trunc(e.start_at)).hour] += 1;
    let peakHour = 0;
    for (let h = 1; h < 24; h++) if (byHour[h] > byHour[peakHour]) peakHour = h;
    const total = byHour.reduce((s, x) => s + x, 0);
    const share = total ? r1((100.0 * byHour[peakHour]) / total) : 0;
    const secs = sub.reduce((s, e) => s + Math.trunc(e.duration_s || 0), 0);
    const subDurs = sub.map((e) => Math.trunc(e.duration_s || 0));
    perCot.push({
      cot: eCot,
      cries: sub.length,
      seconds: secs,
      avg_seconds: sub.length ? r1(secs / sub.length) : 0,
      max_seconds: sub.length ? Math.max(...subDurs) : 0,
      median_seconds: sub.length ? r1(median(subDurs)) : 0,
      by_hour: byHour,
      peak_hour: peakHour,
      peak_share: share,
      night_cries: sub.filter((e) => {
        const h = lt(Math.trunc(e.start_at)).hour;
        return h >= NIGHT_START_HOUR || h < NIGHT_END_HOUR;
      }).length,
    });
  }

  const nights = [...nightsMap.keys()].sort().map((k) => {
    const v = nightsMap.get(k);
    return {
      night: k,
      cries: v.length,
      seconds: v.reduce((s, e) => s + Math.trunc(e.duration_s || 0), 0),
      cots: [...new Set(v.map((e) => Math.trunc(e.cot || 0)))].sort((a, b) => a - b),
      first: Math.min(...v.map((e) => Math.trunc(e.start_at))),
      last: Math.max(...v.map((e) => Math.trunc(e.start_at))),
    };
  });

  return {
    range: { from: since, to: now, days },
    duration_hist: hist,
    confidence: {
      min: confs.length ? Math.min(...confs) : null,
      median: confs.length ? r1(median(confs)) : null,
```

```
      max: confs.length ? Math.max(...confs) : null,
    },
    gaps: {
      count: gaps.length,
      median_min: gaps.length ? r1(median(gaps) / 60.0) : null,
      max_min: gaps.length ? r1(Math.max(...gaps) / 60.0) : null,
      min_min: gaps.length ? r1(Math.min(...gaps) / 60.0) : null,
    },
    nights,
    per_cot: perCot,
    // Every pattern on this page is a statement about *when*.  A record
    // with no time cannot take part in one, so it is counted and
    // reported rather than quietly folded in.
    untimed_cries: untimed,
    insights: buildInsights(events, perCot, nightsMap, gaps),
  };
}

function buildInsights(events, perCot, nightsMap, gaps) {
  const out = [];
  if (!events.length) return ['No cries have been recorded yet.'];

  const total = events.length;
  out.push(`${total} cry episode${total !== 1 ? 's' : ''} recorded.`);

  for (const c of perCot) {
    if (c.cries < 3) continue;
    if (c.peak_share >= 40) {
      out.push(
        `Cot ${c.cot} is usually heard around ${String(c.peak_hour).padStart(2, '0')}:00 ` +
        `(${c.peak_share}% of its cries fall in that hour).`
      );
    }
    if (c.night_cries / c.cries > 0.6) {
      out.push(`Cot ${c.cot} cries are mostly at night (${c.night_cries} of ${c.cries}).`);
    }
  }

  if (nightsMap.size) {
    const counts = [...nightsMap.values()].map((v) => v.length);
    const avg = counts.reduce((s, x) => s + x, 0) / counts.length;
    out.push(
      `On average ${avg.toFixed(1)} cry episodes per night across ` +
      `${nightsMap.size} night${nightsMap.size !== 1 ? 's' : ''}.`
    );
    let worstK = null;
    let worstV = null;
    for (const [k, v] of nightsMap) {
      if (worstV === null || v.length > worstV.length) { worstK = k; worstV = v; }
    }
    if (counts.length > 1 && worstV.length > avg * 1.5) {
      out.push(`The night of ${worstK} was the heaviest, with ${worstV.length} episodes.`);
    }
  }

  if (gaps.length) {
    const med = median(gaps) / 60.0;
    if (med < 30) {
      out.push(`Cries come roughly every ${med.toFixed(0)} minutes on average, which is a dense pattern.`);
    } else if (med > 240) {
      out.push(`Cries are well spread out -- about every ${(med / 60).toFixed(1)} hours.`);
    }
  }

  const durs = events.map((e) => Math.trunc(e.duration_s || 0));
  if (durs.length) {
    const medD = median(durs);
    if (medD >= 60) {
      out.push(`The median cry lasts ${(medD / 60).toFixed(1)} minutes, which is long for settling complaints.`);
    } else if (medD <= 15) {
      out.push(`The median cry lasts ${Math.round(medD)} seconds -- short, settling sounds.`);
    }
  }
  return out;
}

/* ------------------------------------------------------------------ auth */

async function sha256Bytes(s) {
  const d = await crypto.subtle.digest('SHA-256', new TextEncoder().encode(s));
  return new Uint8Array(d);
}

/* timingSafeEqual needs equal lengths, so both sides are hashed first. */
async function safeEqual(a, b) {
```

```
  const [da, db] = await Promise.all([sha256Bytes(a), sha256Bytes(b)]);
  return crypto.subtle.timingSafeEqual(da, db);
}

/* The hub sends X-CryNet-Token (net token <value> on its console); the
   caregiver's browser sends the Basic login it was asked for when the
   page was opened.  With neither secret set — local development — every
   request is allowed, exactly like server.py with no --token. */
async function authorised(request, env) {
  const token = env.UPLOAD_TOKEN || '';
  const pass = env.DASHBOARD_PASSWORD || '';
  const user = env.DASHBOARD_USER || 'caregiver';
  if (!token && !pass) return true;

  const got = request.headers.get('X-CryNet-Token') || '';
  if (token && got && (await safeEqual(got, token))) return true;

  const h = request.headers.get('Authorization') || '';
  if (pass && h.startsWith('Basic ')) {
    let dec = '';
    try { dec = atob(h.slice(6)); } catch (e) { return false; }
    const i = dec.indexOf(':');
    const u = i < 0 ? dec : dec.slice(0, i);
    const p = i < 0 ? '' : dec.slice(i + 1);
    if ((await safeEqual(u, user)) && (await safeEqual(p, pass))) return true;
  }
  return false;
}

function unauthorised() {
  return json({ error: 'unauthorised' }, 401, {
    'WWW-Authenticate': 'Basic realm="Amazing\'s CryNet", charset="UTF-8"',
  });
}

/* ------------------------------------------------------------- upload */

async function handleUpload(db, request) {
  let body;
  try { body = await request.json(); } catch (e) {
    return json({ error: 'bad json' }, 400);
  }
  const device = String((body && body.device) || 'unknown');
  const hub = (body && body.hub) || {};
  const fw = String(hub.fw ?? '');
  const name = String(hub.name ?? '');
  const now = Math.floor(Date.now() / 1000);
  const evs = (body && body.events) || [];

  // One batch, so a half-applied upload cannot happen: the pair
  // (device, seq) makes re-uploading the same records harmless.
  const stmts = [];
  for (const ev of evs) {
    stmts.push(
      db.prepare(
        'INSERT OR IGNORE INTO events (device, seq, kind, cot, start_at,' +
        ' end_at, duration_s, peak_conf, windows, peak_rms, mean_rms,' +
        ' boot, uptime_ms, received_at) VALUES (?,?,?,?,?,?,?,?,?,?,?,?,?,?)'
      ).bind(
        device,
        toInt(ev.seq) ?? 0,
        String(ev.kind ?? 'cry'),
        toInt(ev.cot),
        toInt(ev.start),
        toInt(ev.end),
        toInt(ev.duration_s),
        toInt(ev.peak_conf),
        toInt(ev.windows),
        toFloat(ev.peak_rms),
        toFloat(ev.mean_rms),
        toInt(ev.boot),
        toInt(ev.uptime_ms),
        now
      )
    );
  }
  stmts.push(
    db.prepare(
      'INSERT INTO devices (device, name, fw, first_seen, last_seen)' +
      ' VALUES (?,?,?,?,?)' +
      ' ON CONFLICT(device) DO UPDATE SET' +
      '   name=COALESCE(excluded.name, devices.name),' +
      '   fw=COALESCE(excluded.fw, devices.fw),' +
      '   last_seen=excluded.last_seen'
    ).bind(device, name || null, fw || null, now, now)
```

```
  );

  const results = await db.batch(stmts);
  let accepted = 0;
  for (let i = 0; i < evs.length; i++) accepted += (results[i] && results[i].meta && results[i].meta.changes) || 0;
  return json({ accepted, received: evs.length, server_time: now });
}

/* ------------------------------------------------------------ presence */

/* A hub is taken to be there if it has said something in the last
   ninety seconds.  The hub says every thirty seconds, so three is three
   chances missed before the dashboard calls it gone -- the same rule of
   three the offline server keeps.  The owner set the constraint that
   decides the rate: nothing may cost the cots anything, and they share
   the hub's radio, so the hub posts every 30 s rather than every 15.
   The chain -- the hub speaks every 30 s, this waits 90 s, the page
   asks every 10 s -- means a hub that is switched off reads as gone in
   about two minutes. */
const PRESENCE_STALE_S = 90;

/* Mirrors server.py's build_presence(): who is here, and which of them
   is still answering.  A record says what happened; this says whether the
   thing that watches is working, which is the other question. */
async function buildPresence(db) {
  const now = Math.floor(Date.now() / 1000);
  const devices = await getDevices(db);
  const demo = new Set();
  const hubs = devices.map((d) => {
    const seen = Number(d.last_seen) || 0;
    const isDemo = (d.fw || '') === 'demo';
    if (isDemo) demo.add(d.device);
    return {
      device: d.device,
      name: d.name,
      fw: d.fw,
      at: seen,
      age_s: seen ? now - seen : null,
      // The demo hub is not a thing that can go offline -- it is a made-up
      // hub with nothing posting it.
      online: isDemo || (!!seen && now - seen <= PRESENCE_STALE_S),
      demo: isDemo,
    };
  });
  const r = await db.prepare('SELECT * FROM presence ORDER BY device, cot').all();
  const nodes = (r.results || []).map((p) => {
    // A cot is online only while the hub that sees it still is.  The row
    // is the hub's last word, and `at` says when it was said; on the
    // same clock the hub itself is judged by, a row the hub has stopped
    // refreshing goes stale.  Without this, the post a hub made just
    // before a power cut would hold its cots "online" for ever -- the
    // hub reads as gone while the cots read as watched, which is the
    // one lie this panel must not tell (D-108).
    const at = Number(p.at) || 0;
    const fresh = at > 0 && now - at <= PRESENCE_STALE_S;
    return {
      device: p.device,
      cot: p.cot,
      online: demo.has(p.device) || (!!p.online && fresh),
      deaf: !!p.deaf,
      rssi: p.rssi,
      at: p.at,
    };
  });
  return { now, hubs, nodes };
}

/* The hub says who is here.  One row per cot, replaced each time, so
   what is stored is always the last thing the hub knew. */
async function handlePresence(db, request) {
  let body;
  try { body = await request.json(); } catch (e) {
    return json({ error: 'bad json' }, 400);
  }
  const device = String((body && body.device) || 'unknown');
  const hub = (body && body.hub) || {};
  const fw = String(hub.fw ?? '');
  const name = String(hub.name ?? '');
  const now = Math.floor(Date.now() / 1000);
  const nodes = Array.isArray(body && body.nodes) ? body.nodes : [];

  const stmts = [
    db.prepare(
      'INSERT INTO devices (device, name, fw, first_seen, last_seen)' +
      ' VALUES (?,?,?,?,?)' +
```

```
      ' ON CONFLICT(device) DO UPDATE SET' +
      '   name=COALESCE(excluded.name, devices.name),' +
      '   fw=COALESCE(excluded.fw, devices.fw),' +
      '   last_seen=excluded.last_seen'
    ).bind(device, name || null, fw || null, now, now),
  ];
  let written = 0;
  for (const n of nodes) {
    const cot = toInt(n && n.cot);
    if (cot === null || cot === undefined) continue;
    stmts.push(
      db.prepare(
        'INSERT INTO presence (device, cot, online, deaf, rssi, at)' +
        ' VALUES (?,?,?,?,?,?)' +
        ' ON CONFLICT(device, cot) DO UPDATE SET' +
        '   online=excluded.online,' +
        '   deaf=excluded.deaf,' +
        '   rssi=excluded.rssi,' +
        '   at=excluded.at'
      ).bind(
        device,
        cot,
        n && n.online ? 1 : 0,
        n && n.deaf ? 1 : 0,
        toInt(n && n.rssi),
        now
      )
    );
    written++;
  }
  await db.batch(stmts);
  return json({ ok: true, cots: written, server_time: now });
}

/* ------------------------------------------------------------- router */

export default {
  async fetch(request, env) {
    const url = new URL(request.url);
    const path = url.pathname;
    const q = url.searchParams;
    try {
      if (path === '/api/v1/health') {
        const r = await env.DB.prepare('SELECT COUNT(*) AS n FROM devices').first();
        return json({ ok: true, time: Math.floor(Date.now() / 1000), devices: r ? r.n : 0 });
      }

      if (path === '/api/v1/events' && request.method === 'POST') {
        if (!(await authorised(request, env))) return json({ error: 'unauthorised' }, 401);
        return await handleUpload(env.DB, request);
      }

      if (path === '/api/v1/presence' && request.method === 'POST') {
        if (!(await authorised(request, env))) return json({ error: 'unauthorised' }, 401);
        return await handlePresence(env.DB, request);
      }

      if (path.startsWith('/api/')) {
        if (!(await authorised(request, env))) return unauthorised();
        if (path === '/api/v1/devices') return json({ devices: await getDevices(env.DB) });
        if (path === '/api/v1/presence') return json(await buildPresence(env.DB));
        if (path === '/api/v1/summary') {
          return json(
            await buildSummary(env.DB, intParam(q.get('days'), 14), intParam(q.get('cot'), 0), q.get('tz') || '')
          );
        }
        if (path === '/api/v1/analytics') {
          return json(await buildAnalytics(env.DB, intParam(q.get('days'), 30), q.get('tz') || ''));
        }
        if (path === '/api/v1/events') {
          const rows = await fetchEvents(env.DB, {
            since: intParam(q.get('from'), 0),
            until: intParam(q.get('to'), 0),
            cot: intParam(q.get('cot'), 0),
            kind: q.get('kind') || '',
            limit: Math.min(intParam(q.get('limit'), 200), 2000),
          });
          return json({ events: rows });
        }
        return json({ error: 'not found' }, 404);
      }

      // The page and its assets — behind the same login, so the browser
      // asks once on opening the dashboard and then attaches it to every
      // fetch the page makes.
```

```
      if (!(await authorised(request, env))) return unauthorised();
      return env.ASSETS.fetch(request);
    } catch (err) {
      return json({ error: 'internal', detail: String(err) }, 500);
    }
  },
};
```

dashboard/worker/schema.sql

48 lines · 1,666 bytes

```
-- The same record dashboard/server.py keeps, plus the index that lets the
-- Worker's window queries reach untimed rows without scanning the table.
-- Apply with:
--   npx wrangler d1 execute crynet --remote --file worker/schema.sql

CREATE TABLE IF NOT EXISTS devices (
    device     TEXT PRIMARY KEY,
    name       TEXT,
    fw         TEXT,
    first_seen INTEGER,
    last_seen  INTEGER
);

CREATE TABLE IF NOT EXISTS events (
    device      TEXT NOT NULL,
    seq         INTEGER NOT NULL,
    kind        TEXT NOT NULL,
    cot         INTEGER,
    start_at    INTEGER,   -- epoch seconds, 0 = the hub did not know the time
    end_at      INTEGER,
    duration_s  INTEGER,
    peak_conf   INTEGER,
    windows     INTEGER,
    peak_rms    REAL,
    mean_rms    REAL,
    boot        INTEGER,
    uptime_ms   INTEGER,
    received_at INTEGER,
    PRIMARY KEY (device, seq)
);

CREATE INDEX IF NOT EXISTS idx_events_start    ON events (start_at);
CREATE INDEX IF NOT EXISTS idx_events_kind     ON events (kind);
CREATE INDEX IF NOT EXISTS idx_events_received ON events (received_at);

-- Who is here.  The cots have no network of their own -- they talk to
-- the hub over its radio -- so the hub is the only thing that can answer
-- for them, and it does so once a minute whether or not there is a cry
-- to record.  One row per cot, replaced each time the hub speaks.
CREATE TABLE IF NOT EXISTS presence (
    device  TEXT NOT NULL,
    cot     INTEGER NOT NULL,
    online  INTEGER NOT NULL,
    deaf    INTEGER NOT NULL DEFAULT 0,
    rssi    INTEGER,
    at      INTEGER,
    PRIMARY KEY (device, cot)
);
```

The shipped offline images, and the record that guards them

firmware/prebuilt/SHA256SUMS

14 lines · 1,060 bytes

```
# SHA-256 of the prebuilt flash images.
# Verify them with: python firmware/tools/flash_unit.py --check

aefe21ad9f40735b7ebc07c2b6ad3eeb440477baed575bfc205f78a043c6282e  hub/bootloader.bin
3c10f0757e31a83f16a623658d7bc0bc07a913ee87372a61a2097abb3ca9a8f8  hub/firmware.bin
463793a642ab4a905bd6b4c0b07c749d4ba199ef459b0a91450ca21c0cd581d4  hub/partitions.bin
32e19c4aa6c614d30d3f1cd90c34040c690d30bc4e7683ac157828258c1c2b24  node/app.bin
f94c5d786a7a8fab06ac5d10e33bf37711a6697636dc037559ea19cc410a17f0  node/boot_app0.bin
```

```
9b2fa5e4af7cf282b2db5ed726b26bcca0aec7946eb3fdacc14444195ba754b3  node/bootloader.bin
148b959cbff1c38aa8e1d5c0ba9d612c54997b945e56a63f41223eef650653a1  node/partitions.bin
bfcf93382084823087ae3875ef098790a1f1db89e46c33ee946703632567e247  wearable/app.bin
f94c5d786a7a8fab06ac5d10e33bf37711a6697636dc037559ea19cc410a17f0  wearable/boot_app0.bin
9b2fa5e4af7cf282b2db5ed726b26bcca0aec7946eb3fdacc14444195ba754b3  wearable/bootloader.bin
148b959cbff1c38aa8e1d5c0ba9d612c54997b945e56a63f41223eef650653a1  wearable/partitions.bin
```

firmware/prebuilt/BUILD_INFO.json

138 lines · 10,716 bytes


```
{
  "schema": 2,
  "rules": {
    "exclude_dirs": [
      ".pio",
      "build",
      "managed_components",
      ".git",
      "__pycache__",
      "esp-tflite-micro",
      "esp-nn"
    ],
    "exclude_names": [
      "c_cpp_properties.json",
      "launch.json",
      "ipch"
    ],
    "exclude_globs": [
      "sdkconfig*",
      ".browse.c_cpp.db*"
    ],
    "keep_globs": [
      "sdkconfig.defaults"
    ]
  },
  "roles": {
    "hub": {
      "roots": [
        "firmware/hub_infer"
      ],
      "extras": [
        "firmware/tools/fetch_deps.sh",
        "firmware/tools/patch_deps.py"
      ],
      "files": {
        "firmware/hub_infer/.gitignore": "67cca8c6c85c4698d0b5592a07a40ef2c16e9270cbe2773c6798944272525125",
        "firmware/hub_infer/.vscode/extensions.json": "913d379558f6a8b0f08778a4f80b4ef0dd2e30b26b451adcd39b307763a9ff53",
        "firmware/hub_infer/.vscode/settings.json": "f380089f7878a01b5f12e57ec11986b3d5dea2faa6496293f0f08a0bc426c551",
        "firmware/hub_infer/CMakeLists.txt": "22e3d0a8b187c09b76e08653a5c588df8b9bfa276bdc1251e19cf380f0a1f5eb",
        "firmware/hub_infer/components/ble_alert/CMakeLists.txt":
"2bdf627133e4db5982f19c510cd290a9c788e3e8ae20e120744e5dcdf2110969",
        "firmware/hub_infer/components/ble_alert/ble_alert.c":
"bb2d698f49efa7639236ee55c20df711532797854524d016e794cc265b9aa3c3",
        "firmware/hub_infer/components/ble_alert/include/ble_alert.h":
"33dc3f925e4596f8c908ed8c1d91b574001bc2ea6e9a69cf4c9bbb9b52e6b2b6",
        "firmware/hub_infer/components/crynet_clock/CMakeLists.txt":
"183d8efa1ea2495660b7370505137e51371b7b7345449895ab6fbdb525bc4775",
        "firmware/hub_infer/components/crynet_clock/crynet_clock.c":
"d5eaedf600d072623538c0741acc0fc20d612d0ca9ca8ae8fd2a25810e38fc3b",
        "firmware/hub_infer/components/crynet_clock/include/crynet_clock.h":
"cd3b6272c1d15bc06623d2375e16432e61d9b84e2f050c46589283e77ebc46fa",
        "firmware/hub_infer/components/crynet_dsp/CMakeLists.txt":
"e65742c58a41df1c013daf9c6fe5c5cd36a4158df2b2220d5874cff5cef03ff2",
        "firmware/hub_infer/components/crynet_dsp/crynet_dsp.c":
"a87869b6dcb3bfa0db215b16def72e6b3789febf2b00606c7cbe4547c82033d5",
        "firmware/hub_infer/components/crynet_dsp/include/crynet_dsp.h":
"3a487005f77f3113342b8aa0347e989b964e91abcf950d78594f1c6c742ac444",
        "firmware/hub_infer/components/crynet_dsp/include/crynet_probe_refs.h":
"740268292c36b47f64994efd496be43479f7d503f96e342b9f3a64849b4279a6",
        "firmware/hub_infer/components/crynet_dsp/reference/dsp_crynet.c":
"f55264d708c8b113945919cae46e3f7b8e51db7c81e3056521d32349950f54e2",
        "firmware/hub_infer/components/crynet_dsp/reference/dsp_crynet.h":
"22e6baf6651f56e5a18adedc9d041961d06860db504013b711c216196307a4a4",
        "firmware/hub_infer/components/crynet_infer/CMakeLists.txt":
"20d7d60a7dfd2c8bb1a92fd6c3ae33f76122cbdace8898beae49fa19c5a77182",
        "firmware/hub_infer/components/crynet_infer/crynet_infer.cpp":
"8c43793017a8b43223d2cf33c5c7a2deeb52f9018c51a34333d97f866cf8a0bc",
        "firmware/hub_infer/components/crynet_infer/include/crynet_infer.h":
"327569c3e86f984792f9a84c20020a5b2d75b97218f2fb9b2b2fb75cd71c8085",
        "firmware/hub_infer/components/crynet_log/CMakeLists.txt":
"cabe69915f12209b7d1517661aebb745dc0903b6a5509aaa1c0f00398a9fa995",
```

```
          "firmware/hub_infer/components/crynet_log/crynet_log.c":
"0e304e1791f45b07a84d7067d882d5d0de36fccff0e36247b7fce100b3cef48f",
          "firmware/hub_infer/components/crynet_log/include/crynet_log.h":
"0bd994ddb935d5bc4715bc47c8a16e9b81be81f549dee349fa401dd357cee8ac",
          "firmware/hub_infer/components/crynet_model/CMakeLists.txt":
"c7a06a20aa984828d37f47db247a8b5790d89167a940a12ea1b3a27a183b7737",
          "firmware/hub_infer/components/crynet_model/crynet_model.c":
"20491354debf62d427e33289e1e6d5a2a80fe64ec8722c317a8f75ce6bff3d0f",
          "firmware/hub_infer/components/crynet_model/include/crynet_model.h":
"201c9bd944892608c5c164a85bc984b9d09aea1acbb36b2d7577e49a942a35f6",
          "firmware/hub_infer/components/crynet_model/include/crynet_tables.h":
"b2aa28b322ac385eea27db9c20da4b435bb71694d560327d5164be5a1c45d72d",
          "firmware/hub_infer/components/crynet_model/include/deploy_config.json":
"be0aa3f2e4d7d2591229ca77c49ce8a849339650af1371663b1e4de2d9ad41a4",
          "firmware/hub_infer/components/crynet_model/include/model_data.h":
"80e92e09eb142b6f1909c921a0e9443cd573c18d15767d35f8d8af73fe576f37",
          "firmware/hub_infer/components/crynet_model/reference/c_parity.json":
"470deddb3cd70185e16f17d0f6a5f75dabf5e6dcd37bf30aaa15fbbbdae6883b",
          "firmware/hub_infer/components/crynet_model/reference/crynet_float.tflite":
"402e310cc5a073795c27724d4eda23d576a121d9445b645a4bca8b185f835291",
          "firmware/hub_infer/components/crynet_model/reference/crynet_int8.tflite":
"4aef55d8cfc0fb75aa944e73945ef5b832f12d71479c48f82aff58563126bc11",
          "firmware/hub_infer/components/crynet_model/reference/metrics.json":
"03d12b50b8eab657d88461641dec64e3a4a52ecadb354cbc04664fb1e7851cdc",
          "firmware/hub_infer/components/crynet_net/CMakeLists.txt":
"cc483be7c142b5bd4a1391b1bea003716ceb4669a2d5632b5599dea7c2ded500",
          "firmware/hub_infer/components/crynet_net/crynet_net.c":
"89c4019fa20de9c5d3de3317446cf3e649a54898ee1236d981a83f2713461e69",
          "firmware/hub_infer/components/crynet_net/include/crynet_net.h":
"faf0f5d0225e54a2fdbbcea7e4e89e51219aa3ab1b358d6c774cea3aa611cf10",
          "firmware/hub_infer/components/crynet_proto/CMakeLists.txt":
"1b8d66e8014ea1af60893fb3c00317991a31216d7c586b72e08025567f464579",
          "firmware/hub_infer/components/crynet_proto/crynet_proto.c":
"cbc2cda02070788e2116ada0b2267504ec43ebb4c538ad8316674f53b89b38e1",
          "firmware/hub_infer/components/crynet_proto/include/crynet_proto.h":
"d6bcce7f9ca6cb66ec7a005f3dfddb4add60e00050c99fb2f85b4fbf611fd0d7",
          "firmware/hub_infer/components/hub_display/CMakeLists.txt":
"14d7eb92da644e55684ca3138383df8d9a327b3f9a734c378cbece353fea6219",
          "firmware/hub_infer/components/hub_display/font5x7.h":
"dad5e2396896eea05408e5ad89ad16b8a842766e68d8e203e5f11fa1974de335",
          "firmware/hub_infer/components/hub_display/hub_display.c":
"5c228eee859e09704007f1a0e1290a6f5b3f23218fa10617bf14e2e5f03bc3f5",
          "firmware/hub_infer/components/hub_display/ili9341.c":
"f868c0f2b5c61ee6a798909e225172b907d165171b17e55d40f15799f92b8090",
          "firmware/hub_infer/components/hub_display/ili9341.h":
"b098d83277af29e1ea30841f36729e00001532486be8ec186fad0d983c57afb9",
          "firmware/hub_infer/components/hub_display/include/hub_display.h":
"95aae75ecfa848b86489d7575bd11a6a4bb9ac3e10fc59caab353c81fb404565",
          "firmware/hub_infer/components/node_link/CMakeLists.txt":
"90a68e077672383a4c781825a1ec09903530b44a5a657736fa7e7f7eaeee0987",
          "firmware/hub_infer/components/node_link/include/node_link.h":
"b17e8933f1f77e6d7bd41926c3ec3a0ea0e1797eb61eaa986bf3da5ca8b8b4f4",
          "firmware/hub_infer/components/node_link/node_link.c":
"07dfbf843fda84d194e9895af9a0a4418f7dd109449684663f0c0bbd35f2f8fc",
          "firmware/hub_infer/dependencies.lock": "6923506ab28bbcd25b26f90db7403ff404ac128a095a48ce5cd1bac56c6c6074",
          "firmware/hub_infer/partitions_16mb.csv": "ba922656212139c51d98a7b6bfe78ff17ebacd48aeb168b7560261623337c10a7",
          "firmware/hub_infer/platformio.ini": "248092a6cc420f57be1108ad9b77655ebf3041fca0f02b1487691d99b209d72c",
          "firmware/hub_infer/sdkconfig.defaults": "ed566e4679766d1c6c2bc9f25d2b77584b4fc2d0372da79add90438a216ffee4",
          "firmware/hub_infer/src/CMakeLists.txt": "0ceac4fef551f4bbd68e183279d36aa8f18fde5d32db77569eb7b70eb779c620",
          "firmware/hub_infer/src/console.c": "23a3351bdcaab86a230e95412a9349216af2a18e963b588316bf9cce24ef9c51",
          "firmware/hub_infer/src/console.h": "e5899803e42dd1843f4a621e772c076a59e8fcc6dc65206c6f44b2e6e93dab9e",
          "firmware/hub_infer/src/link_stage.c": "929dae962238baf52fb5ae8416286b192f70d4da1d1de62d24251ff4fdc7923f",
          "firmware/hub_infer/src/link_stage.h": "ceab549ff0ee2534e48c08784049729016901c61cf74a504ace10793f3e1c4bc",
          "firmware/hub_infer/src/main.c": "da163e9447079fa8a7072f09ac8bda681a636cceb43ee1e322d222eeb0c2e246",
          "firmware/hub_infer/src/probes.c": "c2043e1804f5e8efec81b4a09c5db32946966de0a8899d42cffe311a92d952c1",
          "firmware/hub_infer/src/probes.h": "4030321230e876f8054ceb36c15bdf2eddae2e6b32bf931895f0ed019f97c14f",
          "firmware/tools/fetch_deps.sh": "41143522bc5ac4f50559dcd21444fdc56b5a15dd16e942b3600d69da705b3a42",
          "firmware/tools/patch_deps.py": "16560caf7c506e8d27115644a52e3812335ec448adee3c6afca513fe29f22f6c"
        },
        "source_fingerprint": "a2e67b37a9c474a65742f78591556e2d6e6aed760a8ba48140f7891ae6102d30",
        "images": {
          "bootloader.bin": "aefe21ad9f40735b7ebc07c2b6ad3eeb440477baed575bfc205f78a043c6282e",
          "firmware.bin": "3c10f0757e31a83f16a623658d7bc0bc07a913ee87372a61a2097abb3ca9a8f8",
          "partitions.bin": "463793a642ab4a905bd6b4c0b07c749d4ba199ef459b0a91450ca21c0cd581d4"
        }
      },
      "node": {
        "roots": [
          "firmware/node/node_crynet"
        ],
        "extras": [],
        "files": {
          "firmware/node/node_crynet/node_crynet.ino": "1d9dacf11a042d5ba1545dbfc6848c78d9cdf07b4980617bad13daa302bc80ae"
        },
        "source_fingerprint": "b3c0f42bc4cbdc7d594ab969598cd8e9cdf39195f8890faabe481fc40f5a6813",
```

```
      "images": {
        "app.bin": "32e19c4aa6c614d30d3f1cd90c34040c690d30bc4e7683ac157828258c1c2b24",
        "boot_app0.bin": "f94c5d786a7a8fab06ac5d10e33bf37711a6697636dc037559ea19cc410a17f0",
        "bootloader.bin": "9b2fa5e4af7cf282b2db5ed726b26bcca0aec7946eb3fdacc14444195ba754b3",
        "partitions.bin": "148b959cbff1c38aa8e1d5c0ba9d612c54997b945e56a63f41223eef650653a1"
      }
    },
    "wearable": {
      "roots": [
        "firmware/wearable/wearable_crynet"
      ],
      "extras": [],
      "files": {
        "firmware/wearable/wearable_crynet/wearable_crynet.ino":
"6b7e7c10def7e4fa6df84747e0fb4837659eeaf912f10187c4f97605dc98596d"
      },
      "source_fingerprint": "7d2d3e53ab8f8390a8f6861e26a4a5d8caa30ebacc7d610c1d010aa3355b9826",
      "images": {
        "app.bin": "bfcf93382084823087ae3875ef098790a1f1db89e46c33ee946703632567e247",
        "boot_app0.bin": "f94c5d786a7a8fab06ac5d10e33bf37711a6697636dc037559ea19cc410a17f0",
        "bootloader.bin": "9b2fa5e4af7cf282b2db5ed726b26bcca0aec7946eb3fdacc14444195ba754b3",
        "partitions.bin": "148b959cbff1c38aa8e1d5c0ba9d612c54997b945e56a63f41223eef650653a1"
      }
    }
  }
```