ESP32 Wi-Fi CSI + Audio Streaming Pipeline

A real-time embedded data engineering project:
RF channel sensing and audio acquisition from microcontroller to browser

Mirosław Bober

June 2026

Abstract

This project implements an end-to-end real-time telemetry pipeline that streams two heterogeneous sensor modalities — Wi-Fi Channel State Information (CSI) and 16 kHz PCM audio — from an ESP32-S3 microcontroller, through a Raspberry Pi ingestion server, to a browser-based visualization and recording client. The emphasis throughout is on data engineering concerns: binary wire protocols, lossy-by-design transport, backpressure and drop policies at every stage, rate control via a feedback loop, fan-out pub/sub distribution, and durable storage. The CSI stream is additionally processed into a live presence/motion detector using per-subcarrier background subtraction and an asymmetric-EMA adaptive noise floor. The document also covers a non-trivial embedded debugging case: a silently corrupted CSI source that produced syntactically valid but physically meaningless data, and the data-validation tooling built to detect it.

1 System Overview

1.1 Hardware

A practical hardware note: the INMP441 is *soldered directly onto the ESP32 board* rather than connected over jumper wires. The I^2S bus clocks at $16\,\text{kHz} \times 64 = 1.024\,\text{MHz}$; on loose wiring this picked up audible noise and bit errors, which disappeared entirely once the signal path was shortened to board-to-board solder joints. Garbage suppressed at the analog/digital boundary is garbage that no downstream stage has to filter out.

1.2 Topology

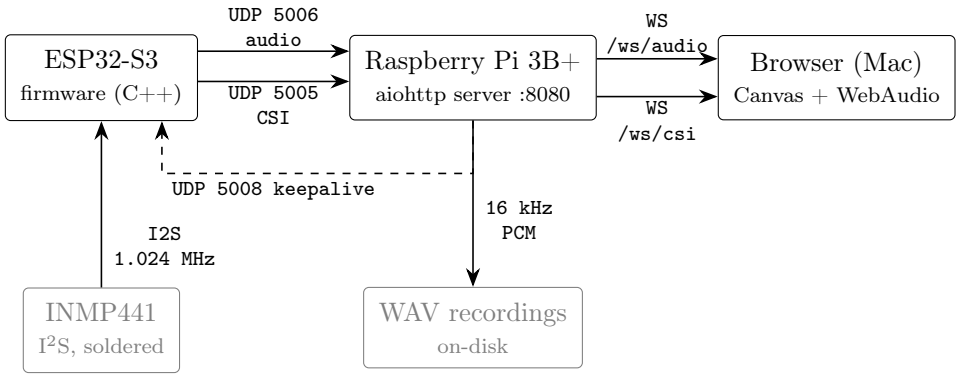

Figure 1: Data flow. Solid arrows carry payload; the dashed arrow is the control-plane feedback loop that *generates* the CSI stream (Section 3.3).

The pipeline is deliberately split into three tiers with distinct responsibilities:

Stream	Rate	Datagram	Bandwidth	Loss policy
Audio (PCM)	31.25 pkt/s	1024 B payload	32 kB/s	send-and-forget, TX backoff
CSI	∼50 frames/s	148–276 B	∼10–14 kB/s	queue drop (16) + hub drop (32)
Keepalive	50 pkt/s (demand)	1 B	negligible	none needed
WAV storage	on demand	—	1.92 MB/min	durable

Table 1: Per-stream characteristics. Total sustained sensor uplink ≈45 kB/s.

Edge (ESP32-S3)
Acquisition only. Samples are framed into compact binary datagrams and pushed out as fast as they are produced. No buffering beyond what is needed to amortize packet overhead; no retransmission.

Ingest (Raspberry Pi)
Stateless fan-out hub. Receives UDP, validates frames, republishes to any number of WebSocket subscribers, and optionally persists audio to WAV. Also runs the keepalive control loop.

Client (browser)
All signal processing and rendering. Parsing the binary CSI header, amplitude extraction, background subtraction, motion scoring, waterfall rendering and audio playback all happen client-side, keeping the Pi's CPU budget trivial.

2 Edge Tier: Firmware Data Acquisition

The firmware (`src/main.cpp`, Arduino-ESP32 3.x on ESP-IDF 5.5 via the `pioarduino` platform) runs two independent producers on the two LX7 cores.

2.1 Audio path (core 1)

The INMP441 delivers 24-bit samples in 32-bit I^2S frames at 16 kHz. The main loop performs a small fixed-point DSP chain per sample:

1. `raw >> 8` — extract the 24-bit sample from the 32-bit frame;
2. `(raw24 × GAIN) >> 8` — digital gain (default 20×) and rescale into the 16-bit range;
3. saturating clip to $[-32768, 32767]$.

Four 128-sample DMA reads are coalesced into one 1024-byte UDP datagram (`READS_PER_PACKET = 4`). This packetization constant was tuned empirically: sending one datagram per DMA read exhausted the lwIP `pbuf` pool (`ENOMEM`, errno 12) under load, while 4× coalescing cuts the packet rate to 31.25 pkt/s for the same 32 kB/s payload — a classic throughput-vs-buffer-pressure tradeoff resolved at the producer. When `endPacket()` still reports failure, the loop backs off 2 ms instead of flooding a saturated stack.

2.2 CSI path (core 0) and the ISR-to-task handoff

Wi-Fi CSI is the physical-layer channel estimate the radio computes for every received frame: one complex coefficient (I, Q) per OFDM subcarrier (64–128 subcarriers depending on the frame type, int8 pairs). It is exposed via `esp_wifi_set_csi_rx_cb()`, which fires *in the Wi-Fi driver task on core 0* — a context where blocking or doing real work is prohibited.

The firmware therefore uses a standard producer/consumer split:

- the callback only copies the record (RSSI, noise floor, ≤256 bytes of IQ) into a 16-deep FreeRTOS queue with a zero-tick timeout — **if the queue is full, the record is dropped**, never blocked on;
- a dedicated sender task pinned to core 0 drains the queue, prepends a 20-byte header and transmits over a *separate* UDP socket, so CSI traffic cannot interleave half-written audio packets.

This "drop at the producer, never block the hot path" policy recurs at every stage of the pipeline (Section 3.2).

2.3 Wire format

CSI frames use a compact little-endian header (RuView ADR-018 layout) with a magic number for cheap demultiplexing and validation:

Offset	Size	Field	Notes
0	4	magic	`0xC5110001`; gate for parsers and the ingest server
4	1	node_id	sensor identity (multi-node ready)
5	1	n_antennas	1
6	2	n_subcarriers	64 or 128 depending on frame type
8	4	freq_mhz	derived from the AP channel at association time
12	4	sequence	monotonic; enables loss measurement downstream
16	1	rssi	dBm, int8
17	1	noise_floor	dBm, int8
18	2	reserved	—
20	$2n$	IQ payload	int8 (I, Q) pairs, one per subcarrier

Table 2: CSI datagram layout (20-byte header + payload, ≤276 bytes total). Audio datagrams are headerless raw PCM — the port number is the schema.

A sequence number in a UDP protocol costs four bytes and buys observability: any consumer can compute loss rate without coordination. The Python `struct` format string `"<IBBHIIbb2x"` on the receiving side is the single source of truth mirrored from the firmware's manual `memcpy` packing.

2.4 Transport choice

Both streams are UDP by design. For a real-time sensor feed, a stale sample is worthless: TCP's retransmission and head-of-line blocking would convert packet loss into *latency* on all subsequent data, which is strictly worse than a gap. Freshness is prioritized over completeness at every stage — loss is handled by *measuring* it (sequence numbers, rate counters) rather than repairing it.

Supporting firmware-level reliability decisions: modem sleep is disabled (`WiFi.setSleep(false)`, essential for continuous streaming), TX power is pinned to 19.5 dBm, reconnection is non-blocking with a 5 s retry interval so the audio loop never stalls on a dead link, and transmission is gated on link state to avoid `EHOSTUNREACH` storms.

3 Ingest Tier: the Raspberry Pi Server

The server (`webapp/server/`, ∼300 lines of Python) is a single-process `asyncio`/`aiohttp` application deployed as a systemd unit (`csi-audio.service`) on the Pi, reachable over Tailscale. It replaced an earlier standalone relay process; collapsing relay, web server and recorder into one event loop removed an entire network hop and a failure domain.

3.1 UDP ingestion

Two `asyncio` datagram endpoints (ports 5005/5006) deliver each packet to a synchronous callback. The CSI callback validates the magic number and minimum length before publishing — malformed or foreign datagrams are dropped at the door. Both callbacks also record the ESP's source address, which makes the system **zero-configuration with respect to the sensor's IP**: the device announces itself by streaming, and the keepalive loop learns its address from observed traffic rather than from config.

3.2 Fan-out hub with bounded queues

Distribution to browser clients goes through a small pub/sub primitive (`hub.py`): each WebSocket subscriber owns a bounded `asyncio.Queue` (32 frames), and `publish()` never awaits:

```
def publish(self, data: bytes) -> None:
    for q in self._queues:
        if q.full():
            q.get_nowait() # drop OLDEST frame
        q.put_nowait(data)
```

The drop-*oldest* policy is the load-bearing decision. With drop-newest, a slow consumer (a throttled background tab, a flaky mobile link) would accumulate a permanently stale 32-frame backlog and render the past; with drop-oldest, it loses frames but always renders *now*, and one slow client can never apply backpressure to the ingest path or to other subscribers. This is the same policy the firmware applies at its FreeRTOS queue — the pipeline degrades by uniform thinning, end to end.

3.3 Keepalive: a control loop that generates the data

CSI has a property that makes it unusual as a data source: the ESP32 computes CSI only for frames it *receives*. An idle station overhears almost nothing, so the stream's sample rate is not a firmware constant — it equals the rate of inbound traffic.

The server exploits this with a demand-driven control loop (`keepalive.py`): while at least one client is subscribed to the CSI hub, it sends a 1-byte UDP datagram to the ESP at 50 Hz. Each datagram's reception triggers one CSI callback, so **the keepalive rate directly sets the CSI sample rate** ($\sim$50 frames/s). When the last viewer disconnects, the loop goes quiet and the sensor's airtime cost drops to essentially zero. Notably, the ESP does not even listen on the keepalive port — the payload is irrelevant; only the RF reception event matters.

3.4 Durable storage

The recorder (`recorder.py`) taps the audio callback and appends raw PCM to a timestamped WAV file (16 kHz, mono, 16-bit), controlled via a REST API (`/api/record/*`). Listing recordings derives duration arithmetically from file size (seconds = (bytes − 44)/32000) instead of opening each file, and deletion validates that the name is a bare `*.wav` filename — a path-traversal guard on the only user-controlled filesystem input.

4 Client Tier: In-Browser Signal Processing

The browser (`static/app.js`, vanilla JS, no framework) subscribes to both WebSocket feeds and does all DSP locally.

4.1 CSI parsing and amplitude extraction

Each binary frame is parsed with `DataView`/`Int8Array` views (zero-copy over the WebSocket `ArrayBuffer`), and per-subcarrier amplitude is computed as $A_k = \sqrt{I_k^2 + Q_k^2}$. A 200-frame ring buffer ($\approx$4 s at 50 Hz) forms the working window. Because the radio interleaves 64- and 128-subcarrier frames depending on the captured frame type, each render pass selects the *dominant* frame size in the window and discards the minority — a small schema-on-read step that keeps the downstream matrix rectangular.

4.2 Waterfall: per-subcarrier background subtraction

Raw CSI amplitude is dominated by the static multipath profile of the room. To make *change* visible, the waterfall renders deviation from a per-subcarrier temporal mean:

$$D_{k,t} = \left| A_{k,t} - \bar{A}_k \right|, \qquad \bar{A}_k = \tfrac{1}{T} \textstyle\sum_t A_{k,t},$$

color-mapped through a 256-entry viridis LUT with the color scale clipped at the 99th percentile of D (robust to single-frame outliers). A still room renders near-black; a person moving through the Fresnel zones renders as bright streaks across subcarriers. Rendering is throttled to one frame per 80 ms, decoupling the 50 Hz data rate from the paint rate.

4.3 Motion detection: asymmetric-EMA noise floor

The motion score per frame is the mean absolute frame-to-frame change, $m_t = \frac{1}{N} \sum_k |A_{k,t} - A_{k,t-1}|$. The detector's robustness comes from how the reference "quiet" level adapts. The noise floor ν follows an **asymmetric exponential moving average**:

$$\nu \leftarrow \begin{cases} 0.7\,\nu + 0.3\,m & m < \nu \quad \text{(falls fast)} \\ 0.997\,\nu + 0.003\,m & m \geq \nu \quad \text{(rises very slowly)} \end{cases}$$

Sustained motion therefore cannot inflate its own detection threshold (the floor rises with a $\sim$5-minute time constant), while a return to stillness re-anchors it within a second. The displayed score is a short EMA of m/ν in units of "$\times$ noise", with motion declared above a fixed 2.2$\times$ threshold. The same algorithm exists twice — prototyped in Python/matplotlib (`csi_view.py`) and ported verbatim to JS — with the desktop version retained as the reference implementation.

4.4 Audio playback under a non-secure context

The Pi is served over plain HTTP, where `AudioWorklet` is unavailable (secure context required). Playback instead schedules each PCM chunk as an `AudioBufferSourceNode` started at a running `nextTime` cursor, with the cursor's initial lead over `currentTime` acting as a tunable jitter buffer (default 200 ms). If the backlog ever exceeds target + 300 ms (e.g. after a background-tab stall), incoming chunks are dropped until the queue drains — the drop-oldest philosophy again, applied to playout.

5 Case Study: Validating the Data Source Itself

The hardest bug in the project was a **silent data integrity failure**. The CSI pipeline ran end to end — packets flowed, headers parsed, rates looked right — but the IQ payload of every frame was the byte sequence $0, 1, 2, 3, \ldots$: a placeholder buffer from a broken driver state, not a channel measurement. Every transport-level health check passed while the data was physically meaningless.

Two artifacts came out of the investigation:

- `csi_analyze.py`, an offline validator that parses raw pcap captures and tests the *statistics* of the payload rather than its syntax: per-subcarrier temporal standard deviation, the count of unique frames, and RSSI spread, ending in an explicit `LIVE` / `STILL FROZEN` verdict. Real RF channels vary frame-to-frame; a frozen buffer has zero temporal variance.
- The root cause: `WiFi.disconnect(true)` during reconnection fully powers down the radio on the ESP32-S3, and after re-association the CSI engine returns the stub buffer forever. The fix is a plain `WiFi.begin()`. Two secondary constraints were established along the way: CSI must be enabled *after* association, and from the main task rather than the Wi-Fi event-handler context.

The transferable lesson is a data engineering one: *schema validation is not data validation.* A pipeline that checks magic numbers and lengths but never looks at the distribution of the values will happily ship a frozen sensor's output for weeks. The statistical validator now provides a one-command answer to "is this source live?".

6 Testing and Deployment

The server has a pytest suite (`webapp/tests/`, six modules) covering the hub's drop-oldest semantics, CSI frame gating, recorder lifecycle and path-traversal guard, keepalive gating on subscriber count, and UDP ingestion. The key testability decision is in `make_app()`: UDP transports and the keepalive task are started by `main()`, *not* by the application factory, and the packet callbacks are exposed on the app object — so tests inject synthetic datagrams by calling `on_csi()`/`on_audio()` directly, with no sockets and no timing dependence.

Deployment is a two-command `rsync` + `systemd` flow: sync the tree to `/opt/csi-audio`, build the venv, enable `csi-audio.service`. Firmware is built and flashed with PlatformIO from the Mac; the pinned `pioarduino` platform (IDF 5.5) is itself a recorded decision, since the older Arduino core 2.x/IDF 4.4 exhibited the frozen-CSI driver bug on the S3.

7 Throughput Summary

Stream	**Rate**	**Datagram**	**Bandwidth**	**Loss policy**
Audio (PCM)	31.25 pkt/s	1024 B	32 kB/s	send-and-forget, TX backoff
CSI	∼50 frames/s	148–276 B	∼10–14 kB/s	queue drop (16) + hub drop (32)
Keepalive	50 pkt/s (demand)	1 B	negligible	none needed
WAV storage	on demand	—	1.92 MB/min	durable

Table 3: Per-stream characteristics. Total sustained sensor uplink ≈45 kB/s.

8 Conclusions

The project demonstrates a complete real-time data pipeline across three heterogeneous tiers — C++ firmware on FreeRTOS, an asyncio ingest server, and an in-browser DSP client — held together by a small hand-rolled binary protocol. The recurring design themes are: prefer freshness over completeness (UDP everywhere, drop-oldest at every queue); make loss observable instead of impossible (sequence numbers, rate counters); move computation to where CPU is free (all DSP in the browser, the Pi only forwards bytes); drive data production by demand (the keepalive loop); and validate the *content* of a data source, not just its envelope (the frozen-CSI statistical validator). The same node-ID'd frame format and fan-out hub extend naturally to the next step: multiple CSI nodes feeding a single ingest point for spatial sensing.

Appendix

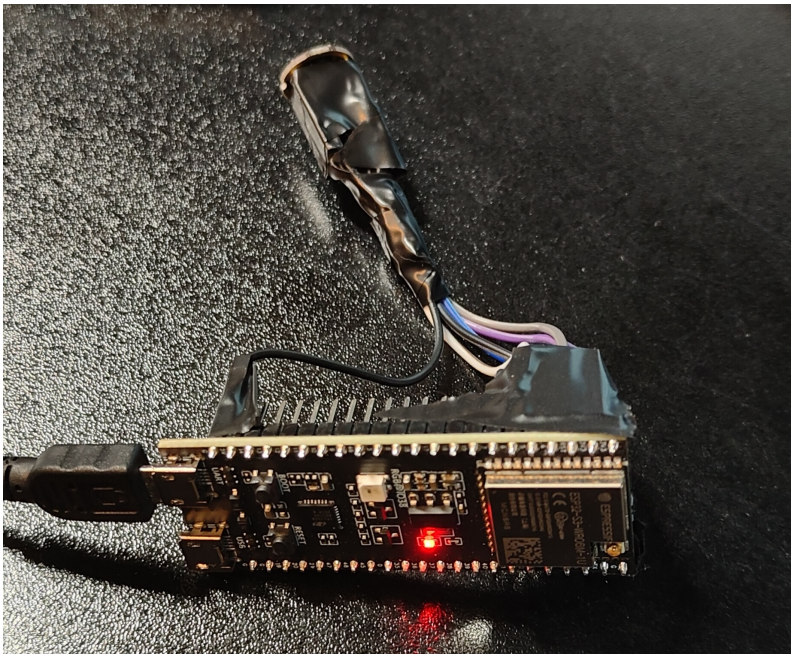

Figure 2: Bare non-encased ESP32 used during software development.

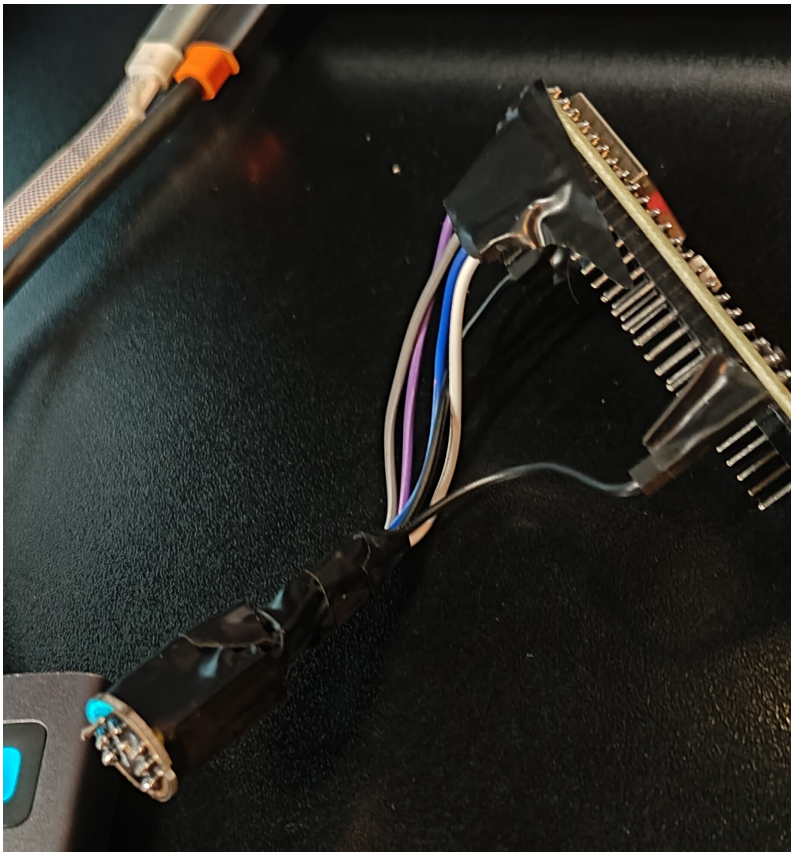

Figure 3: Side view. The microphone is soldered and wrapped in electrical tape.