GLOW 2025: Serial Router Firmware Technical Advice

Technical Advisory — Echoes of Tomorrow Distributed Computer System

Artur Kraskov
Semester 7
ICT & OL, Delta
Fontys 2025-2026

Contents

Introduction

This document is complementary to the main GLOW 2025 Serial Router Firmware Analysis and Advice. This advisory describes the distributed computer system envisioned for Echoes of Tomorrow, focusing on the system topology, hardware/software building blocks, and integration guidance. It consolidates references to internal artifacts and authoritative vendor documentation so that engineering, production, and operations can converge on a consistent, supportable design.

1. System Topology (High-Level)

- **Orchestrator:** Mac Mini as the upstream controller, connected via USB CDC serial to a Teensy 4.1–based Serial Router.
- **Router:** Teensy 4.1 routes framed commands to downstream devices over multiple UARTs.
- **Endpoints:** ESP32-based nodes (arms) and a Centerpiece controller.
- **Links:** USB (Mac↔Teensy), UART (Teensy↔Endpoints); optional GPIO lines for future handshakes.

```
Mac Mini ⇄(USB CDC)⇄ Teensy 4.1 Router ⇄(UARTx)⇄ ESP32 Arms (xN)⇄(UART1)⇄ Centerpiece Controller
```

2. Hardware Components

Mac Mini (Controller)

- Role: Show control, sequencing, and operator interface.
- Interfaces: USB-C/USB-A to Teensy via CDC serial.
- References: https://support.apple.com/en-by/mac/mac-mini

Teensy 4.1 (Serial Router)

- Role: Protocol parsing, routing, observability, optional auto-acknowledge.
- Interfaces: USB CDC upstream; multiple UARTs downstream.
- References: PJRC Teensy 4.1 Product Page — https://www.pjrc.com/store/teensy41.html

ESP32 Modules (Arms)

- Role: Actuator control, local sensing, and endpoint protocol handling.
- Interfaces: UART to Teensy; optional Wi-Fi (out-of-scope for router comms).
- References: Espressif ESP32 Overview — https://www.espressif.com/en/products/socs/esp32

Centerpiece Controller

- Role: Coordinated control for central installation element.
- Interfaces: UART to Teensy; GPIOs TBD.

Cables & Level Compatibility

- USB cable for Mac↔Teensy.
- UART harnesses at 3.3V TTL levels (Teensy 4.1 native). Avoid 5V TTL unless level-shifted.

3. Software Components

Mac Side

- Show Control Application (TBD by Creative/Tools team)
- Serial Host Scripts (for testing): `host_sim/` and `states_diagram/` generators.

Router Firmware
- Location: `serial_router/` (branch: feature/teensy41_esp32_tx_rx)
- Key files: `serial_router/actual_code/final_serial_router_protocol_bridge/teensy_router.ino`.
- Build: Arduino IDE; see repo README.

Endpoint Firmware (ESP32)
- Debugging sketches under `actual_code/final_serial_router_protocol_bridge/` and `final_serial_router_protocol_bridge/`.

Documentation & Processes
- Acceptance & Integration Plan: `docs/GLOW 2025_ Serial Router Firmware acceptance test & integration.pdf`
- Protocol Agreement: states_diagram/GLOW Hardware Communication Agreements.pdf (latest PDF)
- States Diagrams: `states_diagram/final_router_protocol_bridge/output`

4. Teensy 4.1 Architecture and Firmware Implications

This section provides practical guidance for using Teensy 4.1 (ARM Cortex-M7, 3.3V I/O, multiple UARTs, USB device) effectively for the router firmware. It ties architectural points to the current implementation at `actual_code / final_serial_router_protocol_bridge / teensy_router.ino`.

4.1 CPU, Memory, and What It Means for Our Code

- **CPU headroom:** Teensy 4.1's high clock rate gives ample compute, but printf/logging can dominate latency. In our code, enabling `TRACE_SERIAL1_RX_BYTES` prints per-byte and will drastically reduce throughput; keep it off for integration and performance runs.
- **Dynamic allocation:** The sketch uses Arduino `String` objects for buffers and parsing. While convenient, repeated concatenation can fragment the heap on long runs. If extended uptime is a requirement, consider fixed-size ring buffers and C-string parsing for `macBuffer`/`espBuffer` and `parseMessage`.
- **Quick mitigation:** pre-reserve buffers early (e.g., `macBuffer.reserve(512); espBuffer.reserve(512);`) to avoid frequent reallocations.

4.2 Serial I/O Subsystem: USB CDC vs. Hardware UART

- **Upstream USB (CDC):** High effective bandwidth but host-dependent. Logging to `Serial` is fast, yet can block if the host isn't consuming. Use connection checks (e.g., only emit verbose logs when `Serial.dtr()` is true) and consider a leveled logger.

- **Downstream UARTs:** Code currently uses `Serial1` at `LINK_BAUD = 115200`. Teensy supports higher bauds (e.g., 230400, 460800, 921600) reliably with quality cabling and short runs. For production, profile 230400+ to reduce frame time without stressing endpoints.
- **Avoid unnecessary flushes:** `forwardToEsp` calls `Serial1.flush()`, which waits for transmission completion and adds avoidable latency. Generally remove `flush()` and rely on the UART's TX buffer; use `availableForWrite()` if you need backpressure.

4.3 Parser and Framing Robustness

- **Framing model:** `processBuffer` seeks `!!` then the next `##`. This is correct given `#` is reserved in the Agreement, but note that on buffer overflow (`MAX_BUFFER_CHARS`), the code clears the buffer entirely. Prefer dropping up to the next `!!` and keeping the remainder to improve recovery.
- **Strict parameters:** `parseMessage` enforces that a closing `}` ends the frame (no trailing chars). That's good for sanity, but be sure the Agreement forbids whitespace after `}`; otherwise, trim trailing whitespace before the check.
- **Token bounds:** `String tokens[8]` caps tokens; current protocol fits, but guards against overflow by rejecting `tokenCount >= 8` (already implemented). Keep it.

4.4 Routing Table and Pending Correlation

- **Pending map:** `MAX_PENDING = 16` with FIFO eviction. That's adequate for low concurrency but can mis-route rare late confirmations. Add a lightweight TTL (store a timestamp with each `PendingEntry`) and clear stale entries periodically.
- **Address decor:** `forwardToEsp` prepends `MASTER` to the address chain, which is helpful for provenance. Ensure external consumers don't treat this as a routing error.
- **Multi-endpoint growth:** The current sketch uses only `Serial1` downstream. Teensy 4.1 supports multiple hardware UARTs; you can extend to a compile-time routing table, e.g., `{ "ARM1": &Serial6, "ARM2": &Serial2, ... }` and select the Stream at runtime by destination.

4.5 Latency, Throughput, and Timing

- **Frame time** at 115200 bps: Roughly 1 byte ≈ 87 µs; 64-byte frame ≈ 5.6 ms on the wire (one way), plus parsing/log overhead. The acceptance plan should budget overhead separately and avoid per-byte logging during timing tests.
- **Jitter sources:** `Serial.print` inside the hot path, `flush()`, and dynamic `String` growth. Mitigate by: removing `flush()`, reducing log verbosity, pre-reserving buffers, and batching prints (build a single line and print once).
- **Time stamps:** For traceability, append `micros()` stamps to key router events (RX complete, parsed, forwarded). Teensy's timing functions are cheap, and the added context helps triage.

4.6 Safety and Fault Handling

- **Buffer overflow policy:** Instead of clearing all state when `buffer.length() >= MAX_BUFFER_CHARS`, drop up to the next `!!` and count an error. This prevents losing a well-formed frame tail that's already in the buffer.
- **Malformed frames:** Current behavior discards frames without `##` until one appears. Add a watchdog timeout per stream (elapsedMillis) to purge stuck partial frames and log an error.
- **USB presence:** Printing when USB isn't open can block. Gate verbose logs behind a check or add a compile-time `LOG_LEVEL`.

4.7 Suggested Low-Risk Improvements (directly actionable)

- Remove `Serial1.flush()` in `forwardToEsp`.
- Add `macBuffer.reserve(512); espBuffer.reserve(512);` in `setup()` and consider `MAX_BUFFER_CHARS = 768` for burst tolerance.
- Introduce `availableForWrite()` checks before large writes to `Serial` and `Serial1`.
- Add optional timestamped logs for message lifecycle events with a macro like `LOG(level, msg)`.
- Implement a simple TTL in `PendingEntry` (e.g., `uint32_t tsMs;`) and purge stale entries.

5. Communication Agreements (Summary and Links)

- The Serial Communication Agreement governs framing, addressing, verb semantics, payload rules, and timing/baud requirements.
- Source: Internal PDF "GLOW hardware communication agreements.pdf" (latest). Acceptance criteria in `docs/acceptance_integration_plan.md` are derived from this agreement.

6. Deployment & Operations Considerations

- **Versioning:** Tag firmware and host tools; record commit hashes in runbooks.
- **Observability:** Enable USB logging at INFO for operations; DEBUG only for diagnostics.
- **Safety:** Validate isolation (no cross-port broadcast); confirm correct UART port mapping before power-up.
- **Spares:** Maintain a hot spare Teensy and extra cables; keep a pre-flashed recovery image.

7. Risks and Mitigations

- **Protocol Drift:** Freeze a versioned Agreement; require sign-off for changes.

- **Hardware Variance:** New ESP boards may alter UART behavior; revalidate baud and voltage.
- **Noise/EMI:** Use twisted pair/grounding; verify parser resynchronization (see acceptance AT-009).
- **Throughput Stress:** Validate sustained rates with soak tests (AT-007); adjust buffers if needed.

8. Recommended Next Steps

- Finalize pin maps and baud rates per endpoint; update wiring drawing.
- Establish a per-stage integration calendar referencing §11 of the acceptance plan.
- Prepare a minimal ops runbook with flashing steps, log capture, and rollback.

9. References (Internal)

- Repo Root: ./
- Acceptance & Integration Plan: ./docs/acceptance_integration_plan.md
- Host Simulation: ./host_sim/
- Routing Firmware: ./actual_code/
- States Diagrams: ./states_diagram/