# GLOW 2025 Firmware & Automated Acceptance Test Realization

Teensy 4.1 Serial Router

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

---

## Contents



# Introduction

Within the scope of GLOW 2025 Serial Router firmware was analysed, designed and realized. This document covers automated acceptance test.

## Background

Previous work includes:

- GLOW 2025 Project Analysis & Advice [1]
- GLOW 2025 Firmware Design [2]
- Github with realized firmware [3]

# Objectives

- Define a virtual (host-only) acceptance procedure that exercises the Teensy router firmware's protocol handling and measures timing, resource use, and performance without target hardware.
- Keep protocol details out of scope; defer to the Serial Communication Agreement document.

# Scope

- **Firmware under test (UUT):** `actual_code/final_serial_router_protocol_bridge/teensy_router.ino`.
- **Test harness:** use `host_sim/` C++ harness extended to reuse the shared CommandLibary (`CmdLib.h`) for protocol framing/parsing, ensuring parity with firmware expectations.
- **Metrics:** end-to-end latency, throughput, buffer occupancy/overruns, error handling behavior, and long-run stability.

# Environment

- C++17 toolchain (clang++) available.
- CommandLibrary available at https://github.com/GLOW-Delta-2025/Utils/tree/main/CommandLibary
- No microcontroller required; virtual serial endpoints via in-process mocks.
- **Realized test:** https://github.com/GLOW-Delta-2025/serial-router/tree/feature/teensy41_esp32_tx_rx/host_sim

# Acceptance Objectives (virtual)

1) Framing and parsing robustness
   - Given a byte stream with valid frames, the router extracts exactly those frames and no extras.
   - Given noise, partial frames, and escape sequences, router drops/recovers appropriately.
2) Routing correctness
   - ESP→Mac path: after forwarding to Mac, router issues CONFIRM (ACK) to ESP as per Agreement; Mac→ESP path has no ACK.

3) Error handling
   - On malformed frames or bad device/command, router emits COMM_ERROR toward ESP when source is ESP; logs diagnostic for Mac-originated errors.
4) Timing/Performance
   - Latency: simulated T1–T5 timestamps show median ≤ 1 ms and 99th percentile ≤ 5 ms under configured workload (host-sim timing model).
   - Throughput: sustain ≥ 200 messages/s of small frames without parser backlog.
5) Resource behavior
   - No buffer overflows; backlog depth bounded; no memory growth over 10-minute soak.

# Test Strategy

- Use `host_sim/` to emulate two half-duplex streams (ESP-side and Mac-side) feeding/consuming framed messages.
- Build and parse messages via CommandLibrary (`CmdLib.h`) so acceptance scenarios use the exact same message semantics as firmware.
- Add counters and timestamp hooks in the harness to collect:
  - per-frame enqueue/dequeue times (T1–T5)
  - buffer sizes at poll points
  - counts of errors, drops, acks
- Deterministic workloads: fixed-size frames, burst patterns, and randomized intervals.

# Procedures

1. Build harness
   - From `serial-router/host_sim/`, build with the shared command library include path:
   g++ -std=c++17 -Wall -Wextra -pedantic -I../../CommandLibary/CommandLibary main.cpp base_connector.cpp -o host_sim_demo

Expose functions to inject frames on ESP-side and Mac-side, and to read outputs from the opposite side.

2. **Smoke:** Valid frame loopback
   - Inject N=100 valid ESP→Mac frames; expect N seen on Mac side, N ACKs on ESP side, 0 errors.
3. **Noise and partials**
   - Interleave noise bytes and split frames across injections; expect recovery with all valid frames delivered, 0 ACKs missing.
4. **Error cases**
   - Malformed headers, bad checksum/length, unknown device; expect COMM_ERROR to ESP when source is ESP; no ACKs.
5. **Timing load**
   - Run 60 seconds at 200 msgs/s. Capture latency histogram and backlog max. Acceptance: median ≤1 ms; p99 ≤5 ms; backlog max ≤ 5 frames; 0 drops.

6. **Soak**
   - Run 10 minutes at a mixed bursty pattern; ensure no memory growth and no cumulative drops.

# Pass/Fail Criteria

All objectives satisfied across procedures with thresholds met or exceeded.

# CommandLibrary

Using `CmdLib.h` ensures a single source of truth for the message format in both firmware and simulator, minimizing drift in framing and parsing rules and making virtual tests representative of target behavior.

---

# Distributed System Realization

## System Elements

- **Nodes:** Mac host (control/orchestration), Teensy 4.1 router (protocol bridge), ESP32 endpoint (device executor).
- **Communication Channels:** USB CDC (Mac↔Teensy) and UART (Teensy↔ESP) virtually represented by `MockSerial` links.
- **Logical Devices / Addresses:** e.g. `MASTER`, `ARM1`, `CENTER` reflecting distinct actuator groupings.

## LED strips in Virtual Environment

Instead of physical actuators LED arrays for STAR operations are abstracted. Each command (`MAKE_STAR`, `SEND_STAR`, `CANCEL_STAR`, `ADD_STAR`, `STAR_ARRIVED`) represents an actuation intent or status update. Acceptance tests validate that intents traverse the distributed path correctly, ensuring future physical actuator drivers will receive coherent, timely commands.

## Status Feedback

Status frames and confirmations serve instead of sensors with feedback (on device signal processing):

- `CONFIRM:<CMD>` frames model acknowledgment sensors (successful actuation start/completion).
- `REQUEST:STAR_ARRIVED` models an event sensor signaling completion/arrival state.

Error conditions (`COMM_ERROR`) function as fault sensors. Virtual tests inject malformed messages to ensure fault detection propagation is reliable.

## Timing

Defined latency points (T1–T5 conceptual) covered by: per-frame enqueue (ingress), parser extraction, routing decision, forward emission, and reception on the opposite endpoint. Procedures 5 (Timing load) and 6 (Soak) capture distribution (median, p99) and backlog behavior.

## Resource Use (concise)

What we track

- **Ingress backlog:** bytes waiting in the router's receive buffers (`mac_buffer`, `device_buffer`).
- **Egress backlog:** bytes queued on the next hop after the router forwards (`mac_port.available()`, `esp_port.available()`).
- **Mapping size:** pending correlation entries (`origin_by_command.size()`).

How it works

- The router polls, accumulates bytes, extracts frames, and forwards them. After each short `pump(...)` iteration we can sample the three signals above and write them to CSV (not included).

Pass criteria (resource use)

- During the 10-minute soak all three signals remain bounded (no monotonic growth). Mapping size drops when responses are routed; ingress/egress backlogs fluctuate within a small band, indicating no leaks or queue saturation.

## Performance

Performance rationale (why these numbers)

- **Link budget:** at 115200 baud UART (~11.52 kB/s), a typical frame of ~50–60 bytes yields ~190–230 msgs/s theoretical one-way capacity. Targeting ≥ 200 msgs/s exercises the full UART without running it to the ragged edge, leaving firmware/host overhead headroom.
- **Latency:** median ≤ 1 ms, p99 ≤ 5 ms keeps routing overhead well below the 10 ms end-to-end goal cited in the design doc. On Teensy 4.1 (600 MHz), parser/routing work is CPU-light; the UART is the bottleneck, so single-ms targets are practical and keep choreography snappy.
- **Backlog ≤ 5 frames:** at 200 msgs/s this equates to ~25 ms buffered at worst, confirming we drain faster than we receive. A small, bounded queue minimizes memory use and jitter.
- **Expected traffic:** with 4–5 endpoints at 10–20 Hz command rates, worst case is ~50–100 requests/s from Mac, with roughly matching confirmations/status from ESP. The 200 msgs/s target covers this with margin.

- **Headroom path:** if future loads demand more, raising UART to 230400/460800 baud increases capacity proportionally while keeping the same acceptance thresholds meaningful.

# Conclusion

## Distributed Correctness

Routing correctness and selective ACK behavior confirm distributed coordination rules (one-way confirmations only after ESP-originated status or command execution). Noise/error recovery scenarios validate robustness under degraded inputs typical of multi-node links.

## Future Physical Extension

Mapping in this virtual phase provides a template for attaching real actuators (e.g., LED arrays, servos) and sensors (position, brightness feedback) by replacing ESP mock handlers with hardware drivers while retaining CommandLib framing, preserving timing/resource instrumentation hooks.

# References

1. Kraskov A., (2025), GLOW 2025 Analysis and Advice, [Online] Available:
   Canvas:
   https://fhict.instructure.com/courses/13084/assignments/271126?module_item_id=1372588
   PDF:
   https://github.com/GLOW-Delta-2025/serial-router/blob/feature/teensy41_esp32_tx_rx/docs/GLOW%202025%20Project%20Analysis%20%26%20Advice.pdf
2. Kraskov A., (2025), GLOW 2025 Firmware Design, [Online] Available:
   Canvas:
   https://fhict.instructure.com/courses/13084/assignments/271125?module_item_id=1372587
   PDF:
   https://github.com/GLOW-Delta-2025/serial-router/blob/feature/teensy41_esp32_tx_rx/docs/GLOW%202025_%20Serial%20Router%20Firmware%20Design.pdf
3. Serial Router Firmware GitHub:
   https://github.com/GLOW-Delta-2025/serial-router/tree/feature/teensy41_esp32_tx_rx