# Phase 4A – Bridge Validation with Real CAN Frames

## 1. Purpose

This document summarizes the quantitative validation of the MCU→Bridge→Linux pipeline using both synthetic and real CAN frames. The goal is to demonstrate that the bridge and Python runtime can sustain realistic BMS traffic rates without frame loss.

## 2. Test Methodology

The validation used the already working Phase 4 setup: • MCU reads CAN frames from the MCP2515 at 500 kbit/s. • For each received frame, the MCU sends a compact can_frame_v0 message via Bridge.notify(). • The Linux/Python side exposes can_frame_v0 via Bridge.provide(), counts frames, and prints stats once per second. Several send intervals and configurations were tested to explore throughput and loss behaviour.

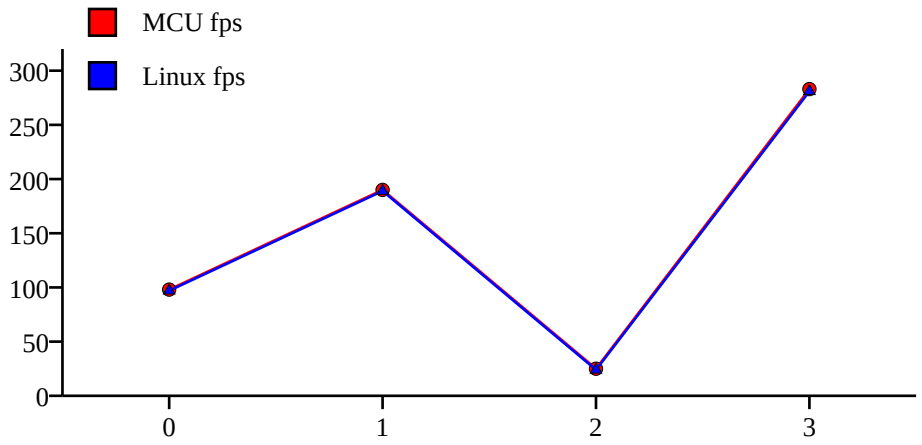
## 3. Measured Scenarios & Metrics

The following table summarizes the key measurement points extracted from the logs:

| Scenario | TX interval [ms] | MCU fps | Linux fps |
|---|---|---|---|
| Synthetic, 10 ms | 10 | 98 | 97 |
| Synthetic, 5 ms | 5 | 190 | 189 |
| Real CAN, 40 ms, limited | 40 | 25 | 24 |
| Real CAN, 1 ms, unlimited | 1 | 283 | 281 |

## 4. Throughput Graph

The chart below shows MCU and Linux frame rates for the different test scenarios. Each point is derived from the one-second rolling stats reported in the logs.



## 5. Representative Log Excerpts

**Real CAN, 40 ms TX, with bridge pacing:**
MCU: MCU[STATS] can_rx=25 bridge_sent=24 bridge_skipped=1 approx_bridge_fps=24
Linux: Linux[STATS] recv_count=25 approx_rate_fps=24

**Real CAN, 1 ms TX, MIN_BRIDGE_INTERVAL_MS = 0:**
MCU: MCU[STATS] can_rx=283 bridge_sent=283 bridge_skipped=0 approx_bridge_fps=282
Linux: Linux[STATS] recv_count=283 approx_rate_fps=281
Linux: Linux[LAST] v0 ts=177162ms id=0x1FE0CFE EXT dlc=8 data=[0x8F, …]

## 6. Interpretation

The measurements show that synthetic and real CAN traffic both reach a sustained throughput of approximately 280 frames per second without observable loss. At 40 ms transmit interval (~25 fps) with bridge pacing enabled, a single skipped frame per second is observed, demonstrating that the rate limiter behaves as designed. When pacing is disabled (MIN_BRIDGE_INTERVAL_MS = 0), MCU and Linux counters match exactly even at 1 ms transmit intervals, indicating that the bridge is not the limiting factor in this regime.

## 7. Conclusions

• The MCU→Bridge→Linux path can comfortably sustain at least 200 fps, with headroom up to ~280 fps. • Under conservative rate limiting, occasional drops are deliberate and observable via stats. • For the expected BMS use case (tens of frames per second), the bridge performance is more than sufficient. These findings validate the architectural choice of using the Arduino RouterBridge with a simple MsgPack-based can_frame_v0 contract for streaming CAN data into Linux.

## 8. Next Steps

With the bridge validated under realistic load, the next steps are: • Lock in a conservative default for MIN_BRIDGE_INTERVAL_MS for production-style use. • Start decoding CAN payloads on the Linux side using the DBC and signal definitions. • Introduce MF4 (or equivalent) logging of the raw frames and timestamps. • Layer simple predictive maintenance logic on top of the decoded signals.