PDF 4.2 – Bridge Sanity Check & First Message

1. Purpose

This document defines the **first, minimal interaction across the MCU ↔ Linux bridge**.

The goal is not to transfer CAN data, but to **prove that inter-processor communication works at all**, in a controlled and observable way.

This is the first deliberate “baby step” of Phase 4.

---

2. Why a Sanity Check Is Required

Before moving real data across the bridge, several assumptions must be validated:

- The bridge infrastructure is operational
- Messages sent from the MCU arrive at the Linux processor
- Message boundaries are preserved
- Communication does not block or destabilize the MCU

Attempting to send CAN frames immediately would make failures ambiguous and difficult to diagnose.

---

3. Scope of This Step

In scope:

- One-way communication: **MCU → Linux**
- A fixed, known message
- Periodic transmission at a low rate
- Observable confirmation on the Linux side

Out of scope:

- CAN data
- Buffering
- Serialization complexity
- Error recovery
- Performance optimization

---

4. Message Definition (Minimal Contract)

The first message across the bridge is intentionally trivial.

Characteristics:

- Fixed message type
- Fixed payload
- Human-readable

Example content (conceptual):

- A static string (e.g. “bridge_alive”)
- Or a simple counter value

The exact content is less important than **reliable delivery**.

---

5. Transmission Strategy

5.1 Rate

Messages are transmitted at a **slow, fixed rate** (e.g. once per second).

This ensures:

- minimal load on the MCU
- clear visibility on the Linux side

5.2 MCU Behavior

- Message transmission must be non-blocking
- Failure to transmit must not affect CAN reception

---

6. Linux-Side Expectations

On the Linux processor, a simple application is responsible for:

- attaching to the bridge endpoint
- receiving messages
- printing or logging received content

No parsing or interpretation is required at this stage.

---

7. Validation Criteria

This step is considered successful if:

- messages appear on the Linux side at the expected rate
- message content matches what was sent
- MCU remains stable and responsive
- no degradation in CAN reception (Phase 3 behavior preserved)

---

8. Failure Modes to Observe

Potential failures at this stage include:

- no messages received
- intermittent reception
- message duplication
- MCU stalls or resets

Each failure mode provides clear diagnostic information before proceeding further.

---

9. Documentation Rationale

This step is intentionally documented as a standalone PDF to:

- demonstrate disciplined, incremental integration
- make assumptions explicit
- show how risk is reduced step-by-step

---

10. Implementation Reference

The reference implementation for this step will be provided in:

- `MCU-Phase4.2.ino` (MCU-side)
- Linux-side Phase 4 application (to be introduced in Phase 5)

Source code is intentionally excluded from this document.

---

11. What This Step Does Not Prove

This step does **not** yet demonstrate:

- throughput capability
- data integrity under load
- buffering across the bridge
- CAN data transfer

Those concerns are addressed incrementally in later Phase 4 documents.

---

12. Phase 4.2 Outcome (Validated on Hardware)

This sanity check was successfully validated on hardware.

Observed behavior:

- The MCU periodically sent a `bridge_alive` message once per second
- The Linux-side application reliably received each message
- Message payloads (counter values) arrived intact and in order
- The MCU remained stable and responsive throughout the test

This confirms that the Arduino Bridge infrastructure is operational and suitable for incremental expansion in later Phase 4 steps.

---

13. PDF 4.2.1 – Implementation Walkthrough & Key Learnings

This section documents the **exact code used** for the Phase 4.2 sanity check and highlights the key discoveries required to make the bridge work reliably on the Arduino UNO Q platform.

13.1 MCU Implementation (MCU-Phase4.2.ino)

```
#include <Arduino_RouterBridge.h>

static uint32_t counter = 0;

void setup() {
  Monitor.begin();
  delay(300);

  Monitor.println("MCU: Phase 4.2 starting...");

  Bridge.begin();
```

```
  Monitor.println("MCU: Bridge.begin() done");

  // Allow Linux/Python runtime to fully start
  delay(5000);
}

void loop() {
  Bridge.notify("bridge_alive", counter);

  Monitor.print("MCU: notified bridge_alive ");
  Monitor.println(counter);

  counter++;
  delay(1000);
}
```

Key characteristics:

- Bridge is initialized once during setup
- Messages are sent from the main loop, not from interrupts
- A deliberate startup delay ensures the Linux side is ready
- Transmission is slow and non-blocking

13.2 Linux / Python Implementation (python/main.py)

```
import time
from arduino.app_utils import App, Bridge

print("Python: Phase 4.2 bridge listener starting", flush=True)

def bridge_alive(counter: int):
    print(f"Linux: bridge_alive counter={counter}", flush=True)

# Register callable endpoint for MCU
Bridge.provide("bridge_alive", bridge_alive)

def loop():
    time.sleep(0.1)  # keep application alive

App.run(user_loop=loop)
```

Key characteristics:

- `Bridge.provide()` is used to expose a callable endpoint
- The Python process is kept alive explicitly

- Output is flushed to ensure visibility in App Lab logs

13.3 Key Learnings & Root Causes

Several important findings were required to achieve a working bridge:

1. **API mismatch discovery**
   The Arduino UNO Q Bridge API does not support `Bridge.on()`.
   The correct pattern is `Bridge.provide()` on Linux and `Bridge.notify()` (or `Bridge.call()`) on the MCU.

2. **Process lifetime matters**
   If the Python process exits, App Lab shuts down the entire application.
   A persistent event loop is mandatory.

3. **Startup sequencing is critical**
   The MCU can send messages before the Python runtime is ready.
   Introducing a startup delay avoids silent message loss.

4. **Small payloads simplify debugging**
   Using a single counter value made correctness obvious and removed ambiguity.

13.4 Why This Is a Milestone

This step represents the first confirmed data path across the MCU ↔ Linux boundary.

It proves:

- the bridge infrastructure works
- message contracts can be enforced
- the system can now evolve safely toward real CAN data transfer

This milestone enables the next Phase 4 steps with confidence.

---

14. Next Document

Proceed to **PDF 4.3 – Message Formats & Contracts**.