PDF 4.1 – Inter-Processor Communication Overview (Arduino Bridge)

1. Purpose

This document introduces **Inter-Processor Communication (IPC)** for the project and explains *why* a communication bridge between the MCU and the Linux processor is required.

It establishes: - the problem IPC is solving - the constraints unique to this platform - the rationale for taking **incremental (“baby step”) progressions** in Phase 4

This document is intentionally conceptual and contains no code.

---

2. System Context

The system consists of two distinct processing domains:

- **MCU domain**
- Real-time, deterministic
- Directly connected to CAN hardware
- Responsible for data acquisition and buffering
- **Linux processor domain**
- Non-real-time
- Capable of storage, analytics, and networking
- Responsible for decoding, persistence, and predictive maintenance

A controlled interface is required to move data safely between these domains.

---

3. Why Inter-Processor Communication Is Needed

The MCU is intentionally limited to: - bounded execution time - minimal memory usage - deterministic behavior

Tasks such as: - decoding DBC signals - writing MF4 files - running analytics - sending alerts

are better suited to Linux.

IPC allows these responsibilities to be separated while maintaining a coherent system.

---

4. Platform-Specific IPC Constraints

On the Arduino UNO Q platform:

- The MCU and Linux processor coexist on the same board
- Communication is mediated by the **Arduino RouterBridge infrastructure**
- Traditional UART-style assumptions do not fully apply

This means IPC must be: - explicitly structured - tolerant of buffering and chunking - non-blocking on the MCU side

---

5. Why We Are Taking Incremental Steps

Inter-processor communication introduces multiple risks simultaneously:

- blocking behavior on the MCU
- uncontrolled memory growth
- message loss or reordering
- timing interference with CAN reception

Attempting to move full CAN frames immediately would make failures difficult to diagnose.

Instead, Phase 4 adopts a **deliberate incremental strategy**.

---

6. Incremental IPC Strategy (Overview)

Phase 4 will progress through the following conceptual steps:

1. **Bridge availability check**
2. Prove that messages can traverse the bridge
3. No CAN data involved
4. **Periodic structured messages**
5. Send counters or timestamps
6. Validate direction, pacing, and loss
7. **Single CAN frame transfer**
8. Fixed ID and payload
9. Minimal buffering

10. **Buffered CAN frame transfer**
11. Multiple frames per message
12. Back-pressure considerations
13. **Error and edge-case behavior**
14. Linux slow or unavailable
15. MCU remains stable

Each step is validated and documented before proceeding.

---

7. Design Principles for Phase 4

The following principles guide IPC design:

- **MCU protection first** – IPC must never compromise CAN reception
- **Explicit contracts** – message formats are defined and versioned
- **Non-blocking behavior** – no waits in the MCU hot path
- **Observability** – success and failure are visible

---

8. What This Document Does Not Define

This document does not yet specify:

- concrete message formats
- serialization mechanisms
- buffer sizes
- Python-side implementation

These are introduced gradually in subsequent Phase 4 documents.

---

9. Relationship to Previous Phases

Phase 4 builds directly on:

- Phase 3's validated CAN acquisition
- Phase 3's buffering and timing guarantees

No changes to Phase 3 architecture are required.

---

10. Next Document

Proceed to **PDF 4.2 – Bridge Sanity Check & First Message**.

This next document will describe the **first minimal data transfer across the bridge**, intentionally independent of CAN.