# PDF 4.4 – First CAN Frame Across the Bridge

## 1. Purpose

This document defines the **first transfer of real CAN data across the MCU ↔ Linux bridge**.

It represents the next deliberate baby step after Phase 4.2 and 4.3: - the bridge is proven - message contracts are defined - now a *single CAN frame* is transported

The objective is correctness and observability, **not throughput**.

---

## 2. Scope

### In scope:

- One CAN frame at a time
- MCU → Linux only
- Fixed message contract
- Low transmission rate
- Full visibility on both sides

### Out of scope:

- Batching
- High-rate streaming
- Persistence (MF4)
- DBC decoding
- Error recovery

---

## 3. Preconditions

The following must already be validated:

- CAN frames are reliably received and buffered on the MCU (Phase 3)
- Bridge communication is stable (Phase 4.2)
- Message contract is agreed and versioned (Phase 4.3)

No changes to Phase 3 architecture are required.

---

## 4. Message Contract Used

This step uses the **planned CAN frame contract (v0)** defined in PDF 4.3.

### Message name

- `can_frame_v0`

### Payload fields

| Field | Type | Description |
|---|---|---|
| version | uint8 | Message format version (0) |
| timestamp | uint32 | MCU reception time (ms) |
| can_id | uint32 | Arbitration ID |
| dlc | uint8 | Data length code |
| data[8] | uint8[8] | Raw CAN payload |
| flags | uint8 | Standard / Extended indicator |

Only **one frame** is sent per message.

---

## 5. Frame Selection Strategy

To minimize complexity, the MCU selects:

- the most recently received CAN frame
- from the existing ring buffer
- without batching or aggregation

If no frame is available, no message is sent.

---

## 6. Transmission Rate

The CAN frame transfer rate is deliberately limited:

- At most **one frame per second**
- Independent of CAN bus rate

This ensures: - bridge stability - clear debugging output - no interference with CAN reception

---

## 7. MCU Responsibilities

The MCU is responsible for:

- reading CAN frames as defined in Phase 3
- selecting a single frame for transmission
- packaging the frame according to the v0 contract
- sending the message via the bridge

The MCU does **not**: - retry failed transfers - block waiting for Linux - modify buffering behavior

---

## 8. Linux Responsibilities

The Linux-side application is responsible for:

- receiving `can_frame_v0` messages
- validating payload structure and version
- printing/logging the received frame

No decoding or persistence is performed at this stage.

---

## 9. Validation Criteria

This step is considered successful if:

- Linux receives CAN frame messages periodically
- Received fields match MCU-side values
- Extended vs standard ID flags are correct
- MCU remains stable and continues CAN reception

---

## 10. Failure Modes to Observe

Potential issues include:

- malformed payloads
- missing or duplicated frames
- incorrect timestamps or IDs
- bridge instability under CAN load

Each failure is diagnosable due to low message rate.

---

## 11. Why This Step Is Important

This is the **first time real vehicle bus data crosses the processor boundary**.

It validates: - end-to-end data path (CAN → MCU → Bridge → Linux) - correctness of message contracts - architectural separation between acquisition and processing

---

## 12. Implementation Reference

Reference implementations will be provided in:

- `MCU-Phase4.4.ino`
- Linux Phase 4 CAN receiver module

Source code is intentionally excluded from this document.

---

## 13. What This Step Does Not Prove

This step does not yet address:

- sustained throughput
- frame batching
- data storage (MF4)
- signal decoding
- predictive maintenance logic

Those are introduced in later phases.

---

## 14. Next Document

Proceed to **PDF 4.5 – Buffered CAN Frames Across the Bridge**.