PDF 3.3 – CAN Receive Loop & Buffering

1. Purpose

This document defines how CAN frames are **received, handled, and buffered** on the MCU once the driver layer has been initialized.

It focuses on: - receive-loop structure - buffering strategy - decoupling of real-time acquisition from slower processing

This document builds on: - **PDF 3.1 – MCU Responsibilities & Task Design** - **PDF 3.2 – MCP2515 Driver Overview**

---

2. Design Objectives

The receive loop and buffering mechanism are designed to:

- Preserve CAN frame ordering
- Minimise frame loss under nominal load
- Make overload and loss *observable*
- Prevent slow operations from blocking reception

The design prioritises **determinism and transparency** over maximum throughput.

---

3. Receive Loop Structure

3.1 Polling-Based Reception

In Phase 3, CAN reception is implemented using a **polling loop**:

- The MCU repeatedly queries the driver for available frames
- If no frame is present, the call returns immediately
- If a frame is available, it is processed synchronously

This approach was chosen because it: - simplifies initial integration on a new platform - avoids interrupt-related complexity - allows precise observation of timing and load effects

Polling is an *implementation choice* for Phase 3 and does not preclude interrupt-driven reception in later phases.

---

4. Frame Handling Sequence

When a CAN frame is detected, the MCU performs the following steps **in order**:

1. Read frame metadata (ID, DLC)
2. Copy payload bytes
3. Assign a reception timestamp
4. Push the frame into a ring buffer
5. Update reception counters

Each step is bounded in execution time to preserve determinism.

---

5. Ring Buffer Rationale

5.1 Why Buffering Is Required

CAN frame arrival timing is dictated by the bus, not the MCU.

Operations such as: - diagnostic printing - inter-processor communication - data formatting

may take longer than the inter-frame gap.

A buffer is therefore required to **decouple reception from consumption**.

---

6. Ring Buffer Design

6.1 Structure

The buffering mechanism uses a **fixed-size ring buffer** with:

- statically allocated memory
- head and tail indices
- wrap-around behavior

Dynamic memory allocation is explicitly avoided.

6.2 Push Semantics

- Each received frame is pushed into the buffer immediately
- If the buffer is full, the frame is dropped
- A buffer overflow counter is incremented

This makes data loss visible rather than silent.

---

7. Decoupling Producer and Consumer

7.1 Producer

- The CAN receive loop acts as the producer
- It runs as frequently as possible
- It must never wait on the consumer

7.2 Consumer

- Diagnostic output and later IPC act as consumers
- Consumption rate may be slower or bursty

The ring buffer provides temporal isolation between the two.

---

8. Instrumentation and Counters

The receive and buffering logic maintains explicit counters:

- `rx_count` – total frames successfully read from the driver
- `push_ok` – total frames enqueued into the buffer
- `buf_overflow` – frames dropped due to full buffer
- `rx_error` – driver-level receive errors

These counters are essential for: - validating correct operation - quantifying loss - supporting later performance analysis

---

9. Diagnostic Output Strategy

9.1 Rate Limiting

Diagnostic output is intentionally rate-limited:

- Only a small number of frames are printed per second
- Summary statistics are printed periodically

This prevents output mechanisms from perturbing reception behavior.

9.2 Output Formatting Constraints

On the Arduino UNO Q platform, diagnostic output is transported via the RouterBridge infrastructure.

To maintain readability: - Each diagnostic message is assembled as a single line - A single output call is used per message

---

10. Failure Modes and Observability

The receive loop is designed to make failure modes explicit:

- Buffer saturation → `buf_overflow` increments
- Driver issues → `rx_error` increments
- Consumer lag → divergence between `rx_count` and printed frames

No automatic recovery is attempted in Phase 3.

---

11. Implementation Reference

The receive loop and buffering logic are implemented in:

`MCU.ino`

Source code is intentionally excluded from this document.

---

12. Next Document

Proceed to **PDF 3.4 – Timing & Data Integrity**.