Software Architecture

1. Purpose of This Document

This document defines the **software architecture** for the CAN-based embedded monitoring system. Building on the system and hardware architecture, it describes how software responsibilities are partitioned across the real-time microcontroller and the Linux-based processing environment.

The objective is to explain *what software components exist*, *what they are responsible for*, and *how they interact*, without yet introducing source code or implementation details.

---

2. Software Architecture Principles

The software design follows these guiding principles:

- **Determinism First**: Time-critical CAN reception is protected from non-deterministic workloads.
- **Separation of Concerns**: Each software component has a single, clear responsibility.
- **Loose Coupling**: Components communicate through defined interfaces.
- **Incremental Complexity**: The architecture supports gradual feature growth.
- **Testability**: Software components can be validated independently.

These principles directly support the non-functional requirements defined earlier.

---

3. Software Partitioning Overview

The software is partitioned into three major domains:

1. Microcontroller Firmware (Real-Time Domain)
2. Linux-Based Application Software (Non-Real-Time Domain)
3. Configuration, Logging, and Notification Services

Each domain is described in the following sections.

---

4. Microcontroller Firmware Architecture

4.1 Responsibilities

The microcontroller firmware is responsible for:

- Initializing and managing the CAN controller
- Receiving raw CAN frames

- Timestamping frames as they are received
- Buffering CAN data for transfer
- Transmitting data to the Linux processor

The firmware does **not**: - Decode CAN signals - Perform analytics or monitoring - Manage user interfaces

This strict limitation ensures predictable real-time behavior.

4.2 Internal Firmware Components (Conceptual)

At a conceptual level, the firmware consists of:

- CAN Driver Interface
- CAN Receive Handler
- Timestamp Service
- Data Buffer Manager
- Inter-Processor Communication Handler

These components may be implemented as tasks, modules, or state machines depending on the execution model.

---

5. Linux-Based Application Architecture

5.1 Responsibilities

The Linux-based software is responsible for:

- Receiving CAN data from the microcontroller
- Decoding CAN frames into signals using a DBC file
- Running monitoring and analysis algorithms
- Managing data storage
- Generating user notifications

This environment is chosen for its flexibility, library support, and ease of extension.

5.2 Major Software Components

The Linux application is conceptually divided into the following components:

- Data Ingestion Service
- CAN Frame Decoder
- Signal Monitoring Engine
- Data Logging Service
- Notification Service

Each component can evolve independently as the project grows.

---

6. Data Flow Description

1. CAN frames are received by the microcontroller.
2. Frames are timestamped and buffered.
3. Buffered data is transferred to the Linux processor.
4. Frames are decoded into physical signals.
5. Signals are evaluated by monitoring algorithms.
6. Data is stored for offline analysis.
7. Notifications are generated when required.

This linear flow keeps responsibilities clear and simplifies debugging.

---

7. Error Handling and Robustness (High-Level)

At this stage, error handling is defined conceptually:

- The microcontroller prioritizes data integrity over throughput.
- Data loss or overflow conditions are detectable.
- Linux applications handle malformed or missing data gracefully.

Detailed error-handling mechanisms will be defined later.

---

8. Relationship to Previous Documents

This software architecture:

- Implements the layered system architecture from PDF 0.4
- Aligns with the hardware responsibilities defined in PDF 0.5
- Supports all functional and non-functional system requirements

No contradictions or scope changes are introduced.

---

9. What This Document Does Not Define

This document intentionally does **not** define:

- Programming languages
- RTOS usage or task scheduling details
- Inter-processor communication protocols
- Algorithm implementations

- File formats or database schemas

These will be introduced incrementally during implementation phases.

---

10. Next Steps

With the software architecture defined, the project is now ready to move into **detailed design and implementation**, starting with:

- Inter-processor communication design
- CAN driver and firmware setup
- Linux-side data ingestion prototypes