CAN Bus Stream Parser & Real-Time Diagnostics

---

1. Introduction

This document explains the **design and working** of a Python-based CAN Bus Stream Parser with Real-Time Diagnostics. The system processes a rolling stream of CAN frames from a CSV file, decodes vehicle signals using a DBC-like mapping, maintains real-time state, and generates diagnostic events.

The design is aligned with **automotive embedded diagnostics**, similar to ECU validation and CAN monitoring tools used in industry.

---

2. Objective

The goal of this system is to:

- Read raw CAN frames from a CSV file using a **Command Line Interface (CLI)**
- Decode CAN data into meaningful vehicle signals
- Maintain a real-time state of the vehicle
- Detect abnormal or unsafe conditions
- Generate structured diagnostic events
- Handle errors gracefully during continuous streaming

---

3. What We Have to Prove

This implementation proves that:

- Bit-level CAN signal decoding is correct (including endian handling)
- The system can process a rolling CAN stream efficiently
- Diagnostic rules can detect unsafe or implausible behavior
- CLI-based tools can be used for real automotive diagnostics
- The program is robust against missing, unknown, or out-of-order frames

---

4. System Flow Description

4.1 Start & CLI Argument Reading

The program starts execution and reads mandatory CLI arguments:

- `--in frames.csv` → Input CAN frames file
- `--out events.jsonl` → Output diagnostic events file

This enforces disciplined usage similar to professional diagnostic tools.

4.2 Open Input CSV File

The CSV file contains a rolling CAN stream with the following fields:

- `timestamp_ms`
- `can_id_hex`
- `dlc`
- `data_hex`

The file is opened in read mode, simulating a live CAN log capture.

4.3 Load DBC-like Mapping

A Python dictionary acts as a simplified **DBC file**, mapping:

- CAN ID → Signal definitions
- Signal attributes:
    - start bit
    - length
    - scaling factor
    - endianness

This enables decoding of signals such as:

- Vehicle speed
- Engine RPM
- Steering angle

4.4 Initialize Real-Time State

Before processing frames, the system initializes state variables:

- `speed = None`
- `rpm = None`
- `steering = None`
- `last_timestamp = None`

This state represents the current vehicle condition.

4.5 Rolling CAN Stream Processing

The program iterates over each CSV row, treating it as a **live CAN stream**.

For every frame:

- Timestamp, CAN ID, DLC, and data are parsed

4.6 CAN ID Validation

The system checks:

- **Is the CAN ID present in the DBC mapping?**

If **NO**:

- Frame is ignored or logged as a warning

If **YES**:

- Signal decoding is performed

4.7 Signal Decoding

For valid CAN IDs:

- Bit-level extraction is performed
- Endian handling (little / big endian)
- Scaling and conversion to physical values

Decoded signals are then used to update real-time state.

4.8 Update Real-Time State

The decoded values update:

- Vehicle speed
- Engine RPM
- Steering angle

This state is continuously refreshed with each incoming frame.

4.9 Diagnostic Rule Evaluation

Diagnostic rules are applied on the updated state:

- Sudden vehicle speed jumps
- Excessive steering rate change
- RPM exceeding valid operating range

4.10 Event Creation

If any diagnostic rule fails:

- A structured JSON event is created
- Event fields include:
    - Timestamp
    - Event type
    - Detailed description

4.11 Alert Aggregation (Stretch Goal)

To avoid alert flooding:

- Burst events are grouped
- Alerts are rate-limited

This simulates real ECU diagnostic aggregation.

4.12 Write Events to Output File

Generated diagnostic events are written to `events.jsonl` in JSON Lines format.

4.13 Error Handling

The system gracefully handles:

- Missing CAN frames
- Out-of-order timestamps
- Decode failures

Errors do not crash the system; processing continues.

4.14 Close Files & End

After processing all frames:

- Files are closed
- Output buffers are flushed
- Program terminates cleanly

---

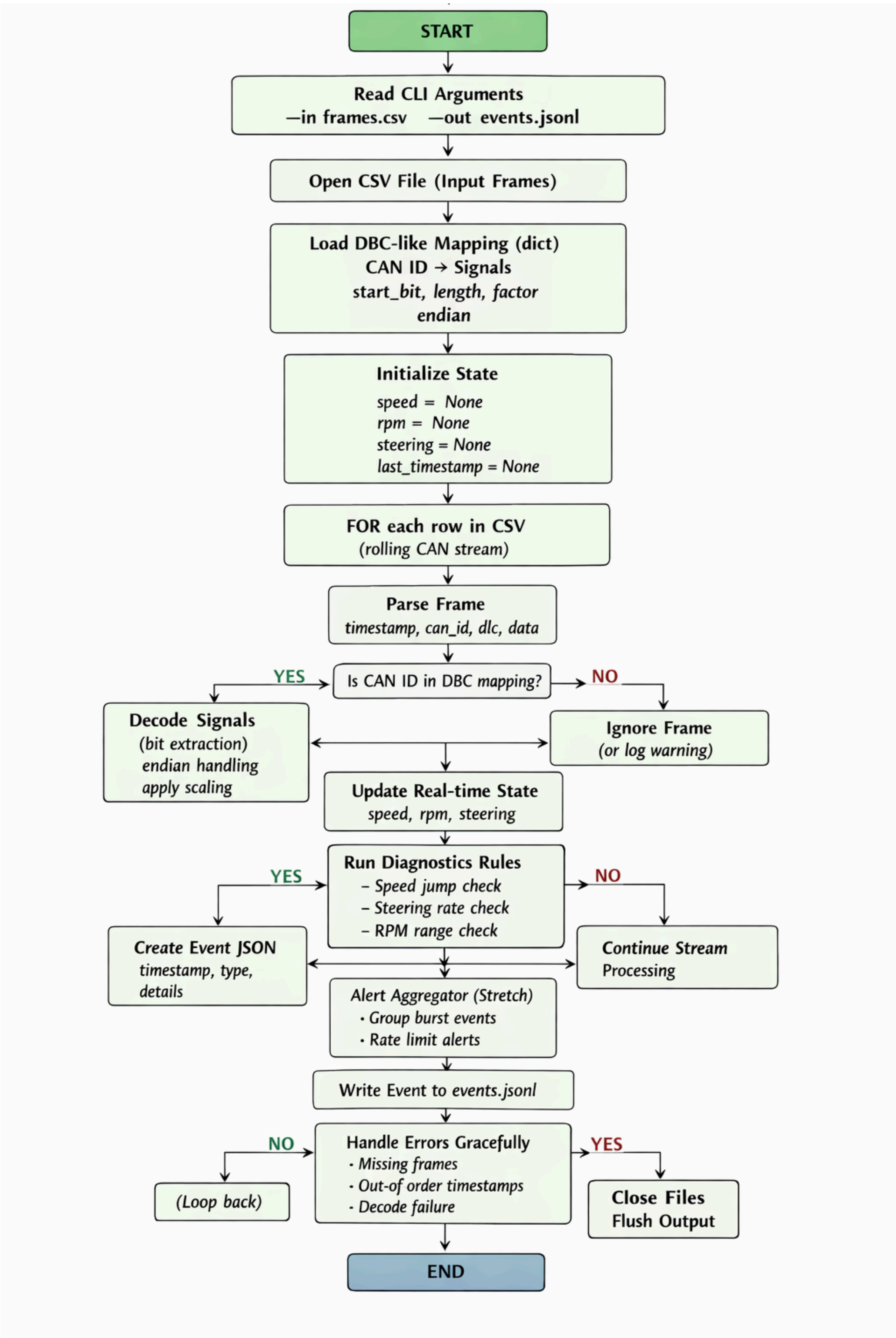
5. Conclusion

This design demonstrates a complete **CAN bus diagnostic pipeline**, covering:

- CLI-based execution
- Real-time stream processing
- Signal decoding
- Automotive diagnostic logic
- Robust error handling

The solution closely resembles tools used in **ECU validation, vehicle diagnostics, and automotive software testing**.

FLOWCHART :


START
Read CLI Arguments
—in frames.csv —out events.jsonl
Open CSV File (Input Frames)
Load DBC-like Mapping (dict)
CAN ID → Signals
start_bit, length, factor
endian
Initialize State
speed = None
rpm = None
steering = None
last_timestamp = None
FOR each row in CSV
(rolling CAN stream)
Parse Frame
timestamp, can_id, dlc, data
YES
Is CAN ID in DBC mapping?
NO
Decode Signals
(bit extraction)
endian handling
apply scaling
Ignore Frame
(or log warning)
Update Real-time State
speed, rpm, steering
YES
Run Diagnostics Rules
– Speed jump check
– Steering rate check
– RPM range check
NO
Create Event JSON
timestamp, type,
details
Continue Stream
Processing
Alert Aggregator (Stretch)
• Group burst events
• Rate limit alerts
Write Event to events.jsonl
NO
Handle Errors Gracefully
• Missing frames
• Out-of order timestamps
• Decode failure
YES
(Loop back)
Close Files
Flush Output
END