📄 Phase 6.2 — Writing MF4 Files

---

1. Purpose of This Step

Phase 6.2 defines how the Linux side will **store real-time CAN frames into MF4 files** for later replay, analysis, machine learning, or predictive maintenance.

This step is necessary because:

- Predictive rules will need historical windows (5–300 seconds)
- Fault trending requires continuity across power cycles
- Cloud/off-board analytics may consume MF4 natively
- ISO timestamp formats must be preserved for traceability
- MF4 allows multi-signal expansion later (temperature, GPS, SoC, etc.)

---

2. What We Already Have

From Phase 5, we now have:

✔ Realtime CAN decode using DBC
✔ Stable MCU → Bridge → Python pipeline
✔ Continuous signal extraction (ECU_A, ECU_B, DCDC voltages)
✔ Validation with simulated degradation scenario

We are now adding:

➡ **Persistent storage**
➡ **Timestamp correctness**
➡ **Replay capability**
➡ **Minimal schema for predictive analytics**

---

3. MF4 Requirements (For Our Use Case)

We are not implementing every ASAM feature — just what this product needs.

Functional requirements

Requirement	Status
Store raw CAN frames	Yes
Store decoded signals	Yes
Store timestamps	Yes (Linux time)
Store bus metadata	Optional
Multi-channel	Optional (future)
Incremental write	Yes
Replay compatible	Yes
Compression	Optional
Cloud uploadable	Yes (future Phase)

Non-functional requirements

Property	Target
Write latency	< 10ms
CPU load	Low
File size	Efficient
Logs rollover	Daily / Size-based
Durability	Power-loss resilient
Open tooling	Cantools / asammdf / Vector

---

4. MF4 Contents (What Goes In)

For Phase 6, each MF4 file contains:

Raw channel:

```
timestamp, can_id, dlc, data[8], flags
```

Decoded channel (optional for now):

```
timestamp, ECUA_Supply_Voltage, ECUB_Supply_Voltage,
DCDC_Output_Voltage
```

Metadata:

```
DBC version
App build version
ECU → harness mapping
Start/end timestamps
```

---

5. Writing Strategy

Three writing strategies were considered:

A) Frame-by-frame write

- Lowest RAM usage
  − High overhead
  − Slow
  Rejected

B) Buffered block write (preferred)

- Good latency
- Efficient file structure
- Compatible with replay
  Selected

C) Full capture → write at end

− Requires large RAM
− Risky on power loss
Rejected

Selected block size: 100–500 ms window

---

6. File Lifecycle Model

File rotation options:

Mode	Use case
Daily	Fleet logging
Hourly	Telematics
Size (e.g., 10MB)	Storage constrained
Event-based	DTC / predictive events

For demo: **Size-based (5MB) + Manual flush**

---

7. Interaction With Predictive Maintenance Logic

The pipeline becomes:

```
CAN frames → DBC decode → Rules → MF4 write → (Cloud optional)
```

It allows:

- retrospective debugging
- model training
- event validation
- regulatory compliance (in real products)

The predictive logic may:
✓ run online (low latency)
✓ validate offline (historical signals)

---

8. How ASAM MDF4 Fits Telematics Systems

MDF4 solves 3 problems CSV/logs do not:

1. **Time alignment**
2. **Binary efficiency**
3. **Multi-channel merging**

This matches commercial telematics (OEMs, fleets, F1 motorsport, etc.)

---

9. Implementation Plan (Phase 6.3 Preview)

Linux implementation will include:

- `MdfWriter` class
- circular buffer for CAN frames
- rollover logic
- replay CLI/tooling
- test harness using synthetic degradation signals

---

10. Validation for Phase 6.2

Success criteria:

✔ MF4 file is created
✔ Timestamps are correct
✔ Data replay matches original inputs
✔ File can be opened with `asammdf` or Vector tools

Stretch goals:

▪ decoded channel export
▪ event-based file tagging

---

11. Risks & Notes

Risk	Mitigation
File corruption on power loss	journaled writes
CPU overhead	buffered writes
DBC version mismatch	metadata versioning
Flash wear	rotation strategy

---

12. Next Step

Proceed to **6.3 — Data replay & verification**

Where we will:

- replay MF4 → Bridge → decoder (same pipeline)
- confirm round-trip correctness
- prep the system for Phase 7 predictive rules