

PDF 5.3 — Validation of DBC Decode & Bridge Pipeline

Phase 5 — Linux Software Architecture

1. Purpose of This Validation Phase

Phase 5.3 was dedicated to validating that the full MCU → Linux pipeline could reliably:

1. Receive real CAN frames
2. Transfer frames across the router bridge
3. Decode frames using a DBC on Linux
4. Display decoded engineering values
5. Sustain continuous operation without freezing
6. Support multiple signal streams simultaneously
7. Maintain performance under realistic in-vehicle timing (100 ms)
8. Avoid buffer stalls, deadlocks, or Py/MCU starvation

This phase transitions the project from **raw data transport** (Phase 4) to **interpreted engineering signals**, enabling the next phases (MF4 storage + predictive maintenance).

2. Test Setup & Conditions

2.1 Hardware

- Arduino UNO Q
- MCP2515 CAN shield (16MHz)
- USB/Serial for debug
- Host laptop running App Lab & Python
- External CAN TX software (manual simulation)

2.2 Software Components

er	Component	Status
	5 driver	
	ridge	
	N reader	1 Phase 4
	Bridge	
	+ Bitstruct	ded
	.demo.dbc	

3. Demonstrated Pipeline

The final validated pipeline is:

```
CAN Bus →
MCP2515 →
MCU RX (CAN) →
Bridge.notify() →
Linux Bridge →
Python handler →
Cantools.decode() →
Engineering signal output
```

This completes the intended data path for real in-vehicle use.

4. Observed Issues & Resolution

Phase 5.3 uncovered several non-obvious system behaviors that had to be understood and mitigated:

4.1 App Lab Freezing / UI Lockup

Symptoms:

- Mouse unresponsive
- UI threads blocked
- Serial output not updating
- Python logs repeating old output

Root causes identified:

- Excessive rate of Bridge notifications
- Lack of Python backpressure handling
- Continuous stdout print causing terminal “scroll storm”
- Attempt to decode every frame immediately
- Cantoools decode is blocking, not async
- No batching or FIFO drainage

Resolutions:

- Reduced CAN frequency (→ 100 ms)
- Decoded only after first appearance (ONCE mode)
- Added stats printing every 1s
- Moved MCU prints to slow-rate stats
- Removed unnecessary flushing
- Introduced bounded processing loop

These changes transformed the system from unstable to robust.

4.2 DBC Loading Issues

Early failures:

Invalid syntax at line 34

Cause:

- DBC header formatting differences
- Cantools strict parsing
- Residual cached file in old App folder

Resolutions:

- Cleaned DBC to compliant structure
- Test harness printed offending lines
- Migrated to new app folder
- Verified via DBC debug utility

Outcome:

DBC now loads cleanly:

ECU_A_STATUS (0x111)

ECU_B_STATUS (0x112)

DCDC_STATUS (0x113)

4.3 Byte Order & Scaling

DBC defines:

0|16@1+ (0.1,0)

0x8D = decimal 141

141 × 0.1 = **14.1 V**

Decoded successfully:

ECUA_Supply_Voltage = 14.1 V

This confirmed:

- byte alignment
- scaling logic
- endian handling (Motorola)
- correct DLC reading

5. Demonstration: Multi-Signal Decode

Final validation test used 3 signals:

Message	Signal	Sample Value
STATUS	upply_Voltage	
STATUS	upply_Voltage	
STATUS	output_Voltage	

Output stream (sample):

ECUA_Supply_Voltage = 14.1 V

ECUB_Supply_Voltage = 14.1 V

DCDC_Output_Voltage = 14.0 V

6. Stability & Performance Results

Metric	Result
frequency	table
ss	
ock	
rvation	
eadlock	
growth	
test	ites
rsiveness	
ial decode	

7. Why This Step Matters

This validation proves several system-critical assumptions:

- ✓ raw CAN can be interpreted into engineering units
- ✓ DBC-driven decoding works on-device
- ✓ performance is sufficient for predictive maintenance
- ✓ no cloud dependency is required
- ✓ decode happens before storage (important for Phase 6-8)
- ✓ pipeline now supports domain logic

This step marked the transition from:

Frames → Meaning

8. Link to Predictive Maintenance Scenario

This enables Demo Scenario:

Harness degradation detected by voltage divergence between ECU_A, ECU_B, and DCDC

Required prerequisites:

- ✓ multi-signal decode
- ✓ real-time values
- ✓ comparable engineering units
- ✓ stable timing

Phase 7 will apply rules to these values.

9. Key Learnings

Technical Learnings

- DBC decode is computationally expensive
- Python prints can stall the system
- App Lab needs batching for high-rate output
- MCU must not be spammed with Monitor prints
- Bridge is reliable under rate-limited conditions

Architectural Learnings

- decode should happen Linux side
 - Phase 6 should store decoded + raw
 - MF4 is suitable for replay & debugging
 - predictive rules can run at 100ms easily
-

10. State at End of Phase 5.3

Deliverables:

- ✓ working 3-signal decode demo
- ✓ real CAN transport validated
- ✓ DBC validated
- ✓ MCU stable under load
- ✓ Linux stable under decode
- ✓ Console readable
- ✓ No deadlocks
- ✓ Performance acceptable

Risk removed:

- ⊘ “Can this decode on edge?”
 - ⊘ “Will App Lab freeze?”
 - ⊘ “Can decode handle multi-signal?”
-

11. Next Phase: Phase 6 (Data Storage & Replay)

Phase 6 introduces:

- MF4 recording
- raw + decoded storage
re-play & validation loops
- comparison of real vs reconstructed frames

This enables:

- Phase 7 predictive logic
- Phase 8 alerting

12. Conclusion

Phase 5.3 validated that the system has achieved:

Complete end-to-end semantic observability.

The system now understands the content of the CAN traffic it transports — a foundational requirement for predictive maintenance and edge analytics.