PDF 5.2 – DBC Decoding & Signal Handling

1. Purpose

This document defines how the Linux/Python side decodes CAN frames into physical voltage signals for the **harness resistance predictive maintenance demo**.

Scope:

- CAN message and signal definitions (standard 11-bit IDs)
- DBC layout and encoding (scale, offset, ranges)
- Mapping from raw frames → decoded `SignalSample`
- Handling timing, missing data, and demo fault injection
- How this feeds the Phase 5 Rule Engine

Implementation details (Python code) live under:

- `python/app/decoder.py`
- `python/dbc/harness_demo.dbc`
- `python/app/signal_state.py`

---

2. CAN Message Definitions (Standard IDs)

All messages use **11-bit standard IDs** and DLC = 8, with a single voltage signal in each.

Message	CAN ID (hex)	Sender	Description	DLC
`ATUS`			ply voltage	

`ATUS`	ply voltage
`TUS`	utput voltage

These IDs are:

- Simple to recognise on a trace
- Standard (not extended) to reduce complexity for learners
- Unique and contiguous, making the DBC easy to read

---

3. Signal Definitions

All three voltages share the same encoding.

3.1 Common Encoding

- **Bit length:** 16 bits (unsigned)
- **Byte order:** Intel / little-endian
- **Scale:** 0.1 V/bit
- **Offset:** 0.0 V
- **Physical range:** 0.0 – 40.0 V
- **Unit:** `"V"`

Conversion:

```
raw  = uint16(data[0..1])
phys = raw * 0.1
```

This provides:

- 0.1 V resolution (sufficient for harness degradation detection)

- Headroom beyond our operating range (up to 40 V)

3.2 ECU A Supply Voltage

- **Message:** `ECU_A_STATUS` (0x111)
- **Signal name:** `ECUA_Supply_Voltage`
- **Start bit:** 0
- **Length:** 16
- **Encoding:** as per common encoding

3.3 ECU B Supply Voltage

- **Message:** `ECU_B_STATUS` (0x112)
- **Signal name:** `ECUB_Supply_Voltage`
- **Start bit:** 0
- **Length:** 16
- **Encoding:** as per common encoding

3.4 DCDC Output Voltage

- **Message:** `DCDC_STATUS` (0x113)
- **Signal name:** `DCDC_Output_Voltage`
- **Start bit:** 0
- **Length:** 16
- **Encoding:** as per common encoding

---

4. Operating Ranges & PM Bands

The decoded physical voltages feed directly into the predictive maintenance logic.

4.1 Normal & DTC Ranges

- **Nominal operating point:** 14.0 V
- **OEM DTC band (conceptual):**
    - DTC if **V < 9.0 V** or **V > 16.0 V**

4.2 Predictive Maintenance Band

We are interested in **early degradation** while the system is still "legal":

PM band: 9.1 – 13.9 V

Within this band:

- No DTC is expected yet
- Voltage drop can indicate increasing harness resistance
- We can estimate **time to failure** before the DTC boundary is reached

---

5. DBC File Layout (`harness_demo.dbc`)

A simplified DBC snippet (using standard notation) for this demo:

```
VERSION "Harness Resistance Demo"

NS_ :
    NS_DESC_
    CM_
    BA_DEF_
    BA_
    VAL_
    CAT_DEF_
```

```
    CAT_
    FILTER
    BA_DEF_DEF_
    EV_DATA_
    ENVVAR_DATA_
    SGTYPE_
    SGTYPE_VAL_
    BA_DEF_SGTYPE_
    BA_SGTYPE_
    SIG_TYPE_REF_
    VAL_TABLE_
    SIG_GROUP_
    SIG_VALTYPE_
    SIGTYPE_VALTYPE_
    BO_TX_BU_
    BA_DEF_REL_
    BA_REL_
    BA_DEF_DEF_REL_
    BU_SG_REL_
    BU_EV_REL_
    BU_BO_REL_
    SG_MUL_VAL_

BS_:

BU_ ECU_A ECU_B DCDC

BO_ 273 ECU_A_STATUS: 8 ECU_A
 SG_ ECUA_Supply_Voltage : 0|16@1+ (0.1,0) [0|40] "V"  ECU_A

BO_ 274 ECU_B_STATUS: 8 ECU_B
 SG_ ECUB_Supply_Voltage : 0|16@1+ (0.1,0) [0|40] "V"  ECU_B

BO_ 275 DCDC_STATUS: 8 DCDC
 SG_ DCDC_Output_Voltage : 0|16@1+ (0.1,0) [0|40] "V"  DCDC
```

Notes:

- Message IDs here are decimal (`273`, `274`, `275`) corresponding to `0x111`, `0x112`, `0x113`.
- Each message has a single signal, making the demo very easy to follow.

The actual file in the repo can include comments and additional metadata, but this structure is sufficient for decoding.

---

6. Python Decoding Flow

6.1 From Bridge to `CanFrame`

The MCU sends a `can_frame_v0` payload across the Bridge.
Python reconstructs:

```
class CanFrame:
    def __init__(self, ts_mcu_ms, can_id, dlc, data, is_extended):
        self.ts_mcu_ms = ts_mcu_ms
        self.can_id = can_id        # 0x111, 0x112, 0x113
        self.dlc = dlc              # 8
        self.data = bytes(data[:dlc])
        self.is_extended = is_extended  # always False in this demo
```

6.2 DBC-Based Decode

Using a DBC library (e.g. `cantools`):

```
db = cantools.database.load_file("python/dbc/harness_demo.dbc")

def decode_frame(frame: CanFrame) -> list["SignalSample"]:
    msg = db.get_message_by_frame_id(frame.can_id)
    if msg is None:
        return []

    decoded = msg.decode(frame.data, decode_choices=False)

    samples = []
```

```python
    for sig_name, phys_value in decoded.items():
        sig = SignalSample(
            name=sig_name,
            ts_mcu_ms=frame.ts_mcu_ms,
            value=float(phys_value),
            unit="V",
            source_id=frame.can_id,
        )
        samples.append(sig)
    return samples
```

Decoded `SignalSample`s are then passed to `SignalState`.

---

7. SignalState & Time Handling

7.1 Latest Values

For rules we only need the **latest** value of each key signal:

- `ECUA_Supply_Voltage`
- `ECUB_Supply_Voltage`
- `DCDC_Output_Voltage`

`SignalState` maintains:

```python
latest: dict[str, SignalSample]
history: dict[str, deque[SignalSample]]
```

7.2 History Window

- A fixed rolling window (e.g. last **60 seconds**) per signal is kept
- Used for:

- simple smoothing
- slope calculation `dV/dt`
- persistence checks

7.3 Message Period & Staleness

The TX tool will send frames at a known period (e.g. 100 ms).
We define **stale** if:

```
now_ts - last_sample.ts_mcu_ms > 1.5 × message_period
```

If a signal is stale:

- It is not used in harness classification
- The rule engine can optionally raise a **“missing data”** info event

---

8. Demo Fault Injection: How the TX Generates Values

The CAN TX software will emulate failures by modifying just the raw voltage value (first 2 bytes).

Example encoding:

```
phys_voltage = 13.2 V
raw = int(phys_voltage / 0.1) = 132
data[0] = 132 & 0xFF       # LSB
data[1] = (132 >> 8) & 0xFF # MSB
```

8.1 Harness A Fault Scenario

- `DCDC_Output_Voltage` fixed ~ 14.0 V
- `ECUB_Supply_Voltage` fixed ~ 14.0 V

- `ECUA_Supply_Voltage` ramps: 14.0 → 10.0 V over time

8.2 Harness B Fault

- `ECUA_Supply_Voltage` fixed ~ 14.0 V
- `DCDC_Output_Voltage` fixed ~ 14.0 V
- `ECUB_Supply_Voltage` ramps: 14.0 → 10.0 V

8.3 Harness C Fault

- `DCDC_Output_Voltage` fixed ~ 14.0 V
- `ECUA_Supply_Voltage` and `ECUB_Supply_Voltage` both ramp together: 14.0 → 10.0 V

By plotting decoded values or observing logs, viewers can clearly see the divergence patterns that the rule engine will classify.

---

9. Interface to Rule Engine

The Rule Engine (Phase 7) consumes only the **decoded** values:

```
V_A = signal_state.get_latest_value("ECUA_Supply_Voltage")  # may be None
V_B = signal_state.get_latest_value("ECUB_Supply_Voltage")
V_D = signal_state.get_latest_value("DCDC_Output_Voltage")
```

Rules then operate in physical units (volts), independent of:

- CAN IDs
- bit positions
- scaling factors

This clean separation makes the PM logic readable for system engineers.

---

10. Summary

- All demo messages use **standard 11-bit CAN IDs** (`0x111`, `0x112`, `0x113`).
- Each message contains a single **16-bit voltage signal**, scaled by `0.1 V/bit`.
- A small DBC (`harness_demo.dbc`) defines encoding and is used by Python via a DBC library.
- The decoder converts raw frames into `SignalSample`s in volts, stored in `SignalState`.
- Demo fault injection is achieved by ramping encoded voltages in the TX tool to simulate harness resistance.
- The Rule Engine later classifies which harness is degrading based purely on decoded voltages.