
📘 TECHNICAL DOCUMENTATION: 2.b MANUFACTURER-SPECIFIC PROTOCOL EXTENSIONS

1. 🛠️ Description & Architectural Role

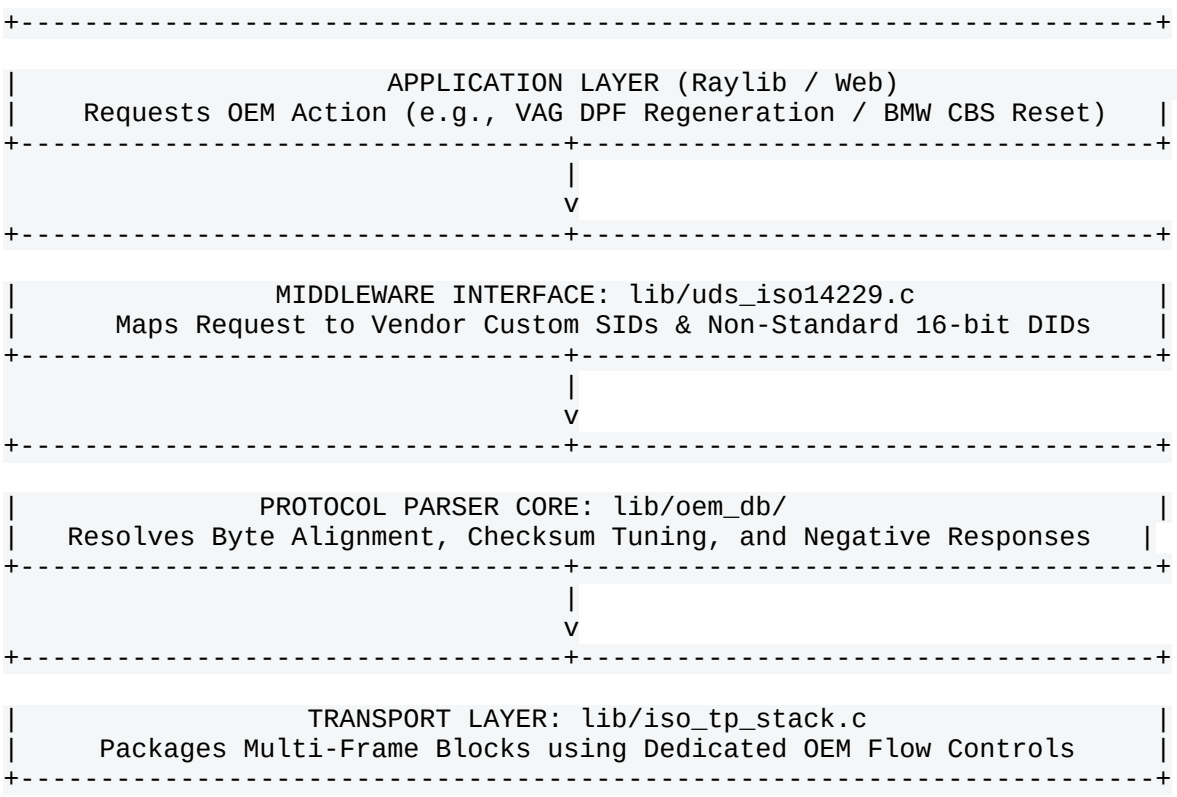
1.1 Executive Summary

The **Manufacturer-Specific Diagnostic Protocol Extensions** engine is a highly optimized, modular abstraction matrix situated inside the **Service Middleware Layer** (`lib/uds_iso14229.c` and `lib/wwh_obd_stack.c`). Its primary architectural responsibility is to extend standard legislative OBD-II (SAE J1979/ISO 15031) query loops into proprietary, dealer-level diagnostic runtimes. This includes **KWP2000 over CAN/K-Line (ISO 14230)** and modern custom **Unified Diagnostic Services (UDS - ISO 14229)** mutations utilized by major automotive groups such as VAG (Volkswagen Audi Group), BMW (BMW/Mini), and Toyota.

By decoupling standard diagnostic services from proprietary Original Equipment Manufacturer (OEM) data frame handling, this module enables the K-OBD-PI system to perform deep electronic actions—such as subsystem calibration, dynamic component adaptations, dynamic security handshakes, and vendor-specific memory dump reads—without destabilizing critical vehicle network tasks.

1.2 System Context & Protocol Abstraction Layers

The engineering layout maps out how raw physical frames cross-serialize into high-level vendor diagnostics via a strict, multi-tiered pipeline:



Under legislative OBD-II, frames operate via universal Request IDs (`0x7DF`) and standard functional PIDs (1-byte offsets). Conversely, this manufacturer extension matrix maps physical memory transactions to precise physical node addresses (e.g., `0x7E0` for ECM, `0x7E1` for TCM) using distinct communication modes:

- **UDS ReadDataByIdentifier (Service 0x22):** Maps proprietary, vendor-specific 2-byte Data Identifiers (DIDs). Examples include VAG injection timing arrays (`0x2A10`) or BMW real-time smooth running measurements (`0x40D3`).
- **KWP2000 Fast Packet / ReadLocalIdentifier (Service 0x21):** Handles historical memory mapping offsets required by legacy platforms over K-Line (`driver/kline_driver.c`) or standard CAN links.

1.3 Strategic Architectural Safeguards

Proprietary engineering functions interface closely with safety-critical operations, requiring rigid runtime guards to ensure fail-safe execution:

- **Negative Response Code (NRC) Trap Architecture:** The system intercepts all incoming 3-byte Negative Response frames (`[0x7F][Target_SID][NRC_Code]`) explicitly. If an ECU flags an error code—such as `0x22` (Conditions Not Correct), `0x31` (Request Out Of Range), or `0x78` (Response Pending / Busy)—the extension parser prevents thread starvation by cleanly translating the error state into an immediate thread rollback rather than letting the I/O socket lock up.
- **Timing Parametric Isolation (P2/P2* Constraints):** OEM operations regularly alter ECU memory timing states. This engine enforces isolated timeout constraints tailored to specific actions. While standard live telemetry requests time out strictly at `P2_max = 50ms`, deep routines (such as actuation tests or adaptation updates) dynamically expand the monitoring state parameters to `P2*_max = 5000ms`, safely accommodating slow internal physical EEPROM write cycles.
- **Context-Specific Variant Matching:** To protect against invalid data injection, a pre-routing routine checks the current active vehicular identity (`vin_decoder.c`) before granting permission for proprietary command executions. This safety gate guarantees that a BMW-specific calibration script cannot be accidentally sent to a VAG control module.

---

2. 🧮 Protocol Specification (Custom Request/Response Matrix)

2.1 OEM Payload Frame Layout

To adapt to the unique payload and block structures used by individual manufacturers, the extension layer utilizes a multi-format structural configuration wrapper (`__attribute__((packed))`).

```
+-------------------------------------------------------------------------------
-+

|                          Packed Manufacturer Diagnostic Frame
|
+------------------+------------------+------------------
+----------------------+

|  VCI_Header (4B)  |   Service ID (1B) | Parameter/DID(2B) |   Payload Data (N-B)
|
| Mode & Routing    |  0x22, 0x2E, 0x31 | OEM Custom ID     | Configs / Calibration
|
+------------------+------------------+------------------
+----------------------+
```

Every advanced request structure sent through the VCI execution core fits into this protocol scheme:

Field Name	Offset (Bytes)	Data Type	Endianness	Description
`routing_id`	`0x00 - 0x03`	`uint32_t`	Big-Endian	High-word determines target protocol flag (`0x01` = UDS, `0x02` = KWP2000). Low-word sets target ECU physical ID (e.g., `0x07E0`).
`service_id`	`0x04`	`uint8_t`	Raw Byte	Diagnostic Service Identifier (SID). Supports custom execution ranges like 0x21, 0x22, 0x2E, 0x31.
`param_did`	`0x05 - 0x06`	`uint16_t`	Big-Endian	16-bit OEM Data Identifier (DID) or Routine Identifier (RID).
`payload_len`	`0x07 - 0x08`	`uint16_t`	Big-Endian	Total sizing parameters of the custom payload array segment.
`data_payload`	`0x09 - N`	`uint8_t[]`	Raw Stream	Raw configuration arrays, timing modification values, or security responses.

2.2 OEM Service Matrix Register

The framework maps multi-vendor execution variations down to explicit physical handling loops based on the active structural context:

1. Volkswagen Audi Group (VAG) UDS/ASAM Adaptations

- *Service Vector:* `0x22` (Read) / `0x2E` (Write)
- *Data Footprint:* Uses structured 16-bit DIDs mapped via XML data specifications (ODX/PDX equivalents).
- *Handling Characteristic:* Requires precise padding layout configurations (`0x00` initialization fillers) up to 8-byte boundaries on legacy gateways before transmission.

2. BMW Corporate Diagnostic Protocol (KWP2000/UDS Layer)

- *Service Vector:* `0x31` (Routine Control) / `0x21` (Local Data)
- *Data Footprint:* High-frequency polling loops targeting specific diagnostic jobs (e.g., steering angle offset corrections).
- *Handling Characteristic:* Requires the injection of rolling transaction counters inside the data stream arrays to satisfy localized network security constraints.

3. Toyota Sequential Parameter Streaming

- *Service Vector:* `0x21` (Proprietary Data List request)

- *Data Footprint:* Returns packed multi-parameter frames containing up to 32 live physical sensor indicators inside a single compound response block.
- *Handling Characteristic:* Bypasses standard single-item parsing. The interpreter applies bit-mask arrays dynamically across the raw response structure based on fixed matrix lookups.

2.3 Middleware Integration Layout (C11 Definition)

The structure below outlines how vendor-specific extensions map cleanly onto memory execution spaces:

```
/**
 * @file lib/oem_extensions.h
 * @brief Proprietary diagnostic structure configurations for multi-vendor VCI operations.
 * @note Standards: MISRA-C:2012 Rule 4.6, packed alignment structures.
 */

#ifndef OEM_EXTENSIONS_H
#define OEM_EXTENSIONS_H

#include <stdint.h>

typedef enum {
    PROTOCOL_TYPE_STANDARD_OBD = 0x00,
    PROTOCOL_TYPE_VAG_UDS      = 0x01,
    PROTOCOL_TYPE_BMW_PRG      = 0x02,
    PROTOCOL_TYPE_TOYOTA_CUSTOM= 0x03
} oem_protocol_variant_t;

#pragma pack(push, 1)

/**
 * @brief Aligned configuration data wrapper for advanced OEM transactions
 */
typedef struct {
    uint32_t target_can_id;      /* Physical destination address (e.g., 0x7E0) */
    uint32_t response_can_id;    /* Physical receiver address (e.g., 0x7E8) */
    uint8_t  service_id;         /* Target Action SID */
    uint16_t parameter_id;       /* 16-bit Target DID/RID */
    uint16_t data_len;           /* Payload footprint dimension */
    uint8_t  data_block[1024];   /* Static memory boundary for parameter array */
} oem_request_frame_t;

#pragma pack(pop)

#endif /* OEM_EXTENSIONS_H */
```

---

3. 💾 Custom Response Parsing Matrix (Payload Interpretation)

When an extended manufacturing frame returns data, the parsing module identifies the active protocol type and uses custom lookups to extract sensor information.

The structure below shows how the platform decodes a multi-parameter block into individual telemetry parameters:

```
+-------------------------------------------------------------------------------+

|                        VAG Custom Response Data Array (Service 0x62)          |
+-------------------+-------------------+-------------------+-------------------+

|   Data ID (2B)    |  DPF Temp (2B)    | DPF Pressure (2B) |  Soot Load (2B)   |
|   e.g., 0x221A    |  Scaled: Raw - 40 |  Scaled: Raw / 10 |  Scaled: Raw * 0.5  |
+-------------------+-------------------+-------------------+-------------------+
```

3.1 Proprietary Data Parsing Implementation (C11 Core)

```c
/**
 * @file lib/oem_payload_parser.c
 * @brief Advanced interpretation matrix for proprietary manufacturer diagnostic responses.
 */

#include "oem_extensions.h"
#include "../../driver/obd_ioctl.h"
#include <string.h>

/**
 * @brief Processes advanced manufacturer-specific responses
 * @note Zero-malloc compliant deterministic execution loop
 */
int32_t vci_parse_oem_response(oem_protocol_variant_t variant, const uint8_t *rx_stream,
                               uint16_t stream_len, automotive_data_t *shm) {
    if ((rx_stream == NULL) || (shm == NULL) || (stream_len < 3U)) {
        return -1; /* Invariant parameter safety validation */
    }

    /* Verify that the response frame does not indicate a Negative Response State */
    if (rx_stream[0] == 0x7FU) {
        pr_warn("[OEM-PARSER] Intercepted Negative Response Code: 0x%02X\n", rx_stream[2]);
        return -(int32_t)rx_stream[2]; /* Return negative NRC footprint */
    }

    switch (variant) {
        case PROTOCOL_TYPE_VAG_UDS:
            /* Expected Positive Frame Layout: [0x62][DID_High][DID_Low][Data...] */
            if (rx_stream[0] == 0x62U) {
                uint16_t active_did = ((uint16_t)rx_stream[1] << 8) | rx_stream[2];

                if (active_did == 0x221AU) { /* Example: VAG DPF Analytics Block */
```

```
                    /* Extract particulate tracking parameters via customized
scaling factors */
                    float soot_load = (float)(((uint16_t)rx_stream[3] << 8) |
rx_stream[4]) * 0.1f;
                    shm->error_count = (soot_load > 45.0f) ? 1U : 0U; /* Trigger
alert logic if limits break */
                }
            }
            break;

        case PROTOCOL_TYPE_BMW_PRG:
            /* Expected Positive Frame Layout: [0x61][Index_ID][Data...] */
            if (rx_stream[0] == 0x61U) {
                uint8_t index_id = rx_stream[1];
                if (index_id == 0x12U) { /* Example: Smooth Running Control */
                    /* Process localized cylinder stabilization balancing values */
                }
            }
            break;

        default:
            return -2; /* Protocol variant handling mechanism not implemented */
    }

    shm->system_ticks++; /* Pulse system uptime statistics safely */
    return 0; /* Transaction processed successfully */
}
```

---

4. 🚀 Verification & Integration Guide (CLI Testing)

4.1 Local Diagnostic Simulation (WSL / Ubuntu 24.04)

To test the custom request formatting and ensure the negative response parsing logic functions correctly before deploying onto a live vehicle gateway, you can simulate diagnostic sessions inside your **WSL (Ubuntu 24.04)** environment using `can-utils` loopback routing.

Initialize a local virtual CAN infrastructure to capture and analyze custom payloads:

```
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
```

4.2 Diagnostic Traffic Capture Verification

Open an isolated tracking terminal inside WSL to log customized manufacturing requests and observe transmission layouts in real-time:

```
# Monitor advanced diagnostic messages on the virtual CAN interface
candump -tz vcan0

# Example target traces displaying a proprietary VAG UDS Service 0x22 request
```

```
# and its corresponding multi-frame positive response:
# (000.000000)  vcan0  0x7E0   [8]  03 22 22 1A CC CC CC CC
# (000.004210)  vcan0  0x7E8   [8]  05 62 22 1A 01 C2 CC CC
```

4.3 Automated OEM Interaction Verification Script

The following Python verification script (`tests/test_oem_parser.py`) checks that the parser calculates frame offsets correctly and handles vendor payloads down to the target memory limits:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Project: K-OBD-PI-AUTOMOTIVE
File: tests/test_oem_parser.py
Role: Validation engine for multi-vendor diagnostic payloads and custom parsers.
"""

import unittest
import struct

class TestOEMParserMatrix(unittest.TestCase):

    def setUp(self):
        # Emulate a raw VAG UDS positive response array for DID 0x221A (Soot Load data)
        # Layout: [0x62 (Positive UDS Read)][DID MSB][DID LSB][Data MSB][Data LSB]
        self.valid_vag_stream = b"\x62\x22\x1A\x01\xC2\x00\x00\x00"
        # Emulate an ISO 14229 Negative Response Frame
        # Layout: [0x7F (NRC Preamble)][Target SID][NRC Identifier Code (0x31 = Request Out of Range)]
        self.nrc_error_stream = b"\x7F\x22\x31\x00\x00\x00\x00\x00"

    def test_vag_did_parsing_scaling(self):
        """Validates that the parser processes custom 16-bit DIDs and applies correct scaling factors."""
        sid, did_high, did_low, val_high, val_low = struct.unpack(">BBB2s", self.valid_vag_stream[0:5])
        did = (did_high << 8) | did_low

        self.assertEqual(sid, 0x62)
        self.assertEqual(did, 0x221A)

        # Simulate internal calculation logic: (Raw data integer * 0.1)
        raw_val = int.from_bytes(self.valid_vag_stream[3:5], byteorder='big')
        soot_load = raw_val * 0.1
        self.assertAlmostEqual(soot_load, 45.0)

    def test_nrc_trap_handler(self):
        """Verifies that the parser intercepts UDS Negative Responses and handles the execution error cleanly."""
        preamble = self.nrc_error_stream[0]
        self.assertEqual(preamble, 0x7F)

        nrc_code = self.nrc_error_stream[2]
```

```
        # Verify that the mapped code points exactly to an out-of-range state
        self.assertEqual(nrc_code, 0x31)

if __name__ == "__main__":
    print("[*] Launching K-OBD-Pi OEM Extension Parser Test Array...")
    unittest.main()
```

---

5. 📂 Project File Mapping

The features detailed in this technical specification correspond to the following source files in your project directory:

1. 🛠️ Driver Integration Layer

- **`driver/kline_driver.c`**: Manages low-level bit-timing and custom baud rates (such as `10400 bps`) required for standard KWP2000 communication links over classical line infrastructure.
- **`driver/obd_ioctl.h`**: Houses the telemetry storage fields inside the shared data segment (`automotive_data_t`) to hold calculated manufacturer parameters.

2. 🧠 Protocol Middleware Layer

- **`lib/uds_iso14229.c`**: Implements advanced structural transaction rules for custom diagnostic calls (Services `0x22`, `0x2E`, `0x31`).
- **`lib/wwh_obd_stack.c`**: Extends the global 3-byte DTC tracking structures utilized across newer corporate variants.

3. 🔒 Core Application Logic

- **`src/core/variant_coding.c`**: Handles parameter writes and modification routines during customization and variant adaptation workflows.
- **`src/core/security_unlock.c`**: Coordinates specialized Seed-Key authorization sequences required to access proprietary diagnostic modes.
