Technical Specification: Standard Telemetry Extension (Milestone E)

1. Protocol Architecture & Compliance

1.1. SAE J1979 Standard Protocol Implementation

The architecture for **Milestone E** expands the diagnostics layer to establish compliance with the **SAE J1979 / ISO 15031-5** specifications. These standards govern the exchange of digital information between a road vehicle's Electronic Control Units (ECUs) and external diagnostic tools over client-server networks. By basing the telemetry expansion on uniform parameter formats, the application guarantees backward compatibility with emissions-regulated legacy vehicles. It also supports high-speed message transactions on modern powertrain sub-buses running the **ISO 15765-4 (CAN)** physical link protocol.

1.2. Service 01 Query Optimization Matrix

To maintain an agile processing window and prevent serialization bottlenecks, the telemetry pipeline treats **Service 01 (Show Current Powertrain Diagnostic Data)** requests as high-priority real-time loops. The core dispatching layers execute a strictly synchronized command-response scheduling flow. This mechanism restricts concurrent data bus requests, preventing ECU traffic overflow. It optimizes throughput by managing inter-frame communication windows, ensuring a stable **250ms polling loop** across multiple parameters without exhausting the vehicle bus bandwidth.

1.3. Parameter Identification (PID) Addressing Topology

The diagnostic layer maps expanded standard registers into the centralized multiplexing array using standardized 1-byte functional addresses. Each parameter operates within a clean abstraction envelope that defines its query bytes, functional headers, and positive verification signatures:

- **Vehicle Speed:** Mode `01`, Parameter Identifier `0D`. Expects a positive acknowledgement sequence from the target node starting with the reflection signature `41 0D`.
- **Engine Coolant Temperature:** Mode `01`, Parameter Identifier `05`. Expects a positive acknowledgement sequence starting with the reflection signature `41 05`.
- **Engine Oil Temperature:** Mode `01`, Parameter Identifier `5C`. Expects a positive acknowledgement sequence starting with the reflection signature `41 5C`.

---

2. Low-Level Bitstream Decoupling & Parsing Algorithms

2.1. PID 0D: Vehicle Speed Transceiving Logic

2.1.1. Byte Formula Structure: A (Scale: 0 - 255 km/h)

The decoding algorithm for standard **Service 01 PID 0D** processes a single data byte (Byte A) returned by the engine management system. The mathematical calculation follows the **SAE J1979** specification directly, where the raw decimal representation of the hexadecimal value equals the actual speed metrics:
$$\text{Vehicle Speed (km/h)}=A$$
This provides a rigid operational scale ranging from 0 km/h (`0x00`) to 255 km/h (`0xFF`).

2.1.2. Alphanumeric Frame Validation

Before committing the raw string to the integer parsing engine, the validation matrix verifies the frame integrity. The inbound string must clear whitespace parsing anomalies and structurally match a minimum length of 6 characters while starting with the positive validation prefix `410D`. Any variation in structural layout or length drop redirects the value to a safe fallback state (`0`).

2.2. PID 05: Engine Coolant Temperature Verification

2.2.1. Byte Formula Structure: A - 40 (Scale: -40°C to +215°C)

The extraction of the engine thermal status via **Service 01 PID 05** converts a single incoming telemetry byte using a standardized offset formula:
$$\text{Coolant Temperature (°C)}=A-40$$
This mathematical transformation translates the unsigned integer byte values (0 to 255) into a signed physical scale tracking engine temperatures from -40°C to +215°C.

2.2.2. Signed/Unsigned Offset Processing

Because Java treats standard primitives as signed entities, the low-level parsing engine passes data blocks through an unsigned conversion boundary (`Integer.parseInt(..., 16)`). This prevents data corruption or sign inversion during the subtraction phase, allowing the platform to safely represent sub-zero engine temperatures during initial cold-start sessions.

2.3. PID 5C: Engine Oil Temperature Telemetry Extraction

2.3.1. Advanced Powertrain Register Calibration Matrix

The processing layer isolates **Service 01 PID 5C** to extract advanced engine oil metrics. The mathematical algorithm mimics the coolant structure (Value = A - 40), but operates on an independent register space dedicated to advanced powertrain thermal tracking. This parameter provides a key health safety metric, differentiating thermal loads between the engine cooling jackets and the internal lubrication system.

2.3.2. Response Timing Limits and Inter-frame Delays

Due to the slower calculation priorities assigned to auxiliary thermal parameters inside older automotive ECUs, PID 5C transactions require protective time gaps. The communication engine maintains an inter-frame transmission spacing of at least **10ms to 20ms** following a PID 5C dispatch. This safeguard prevents bus timeout exceptions and ensures stable data synchronization with the ELM327 controller.

---

3. Multiplexing & Thread-Safe Telemetry Distribution

3.1. Cyclic Sweeping Engine Execution Configuration (250ms Polling Interleaves)

To achieve predictable telemetry stream updates without saturating the vehicle's internal communication networks, the application deploys a **Cyclic Sweeping Polling Engine**. This engine utilizes a single-threaded architecture managed by a `ScheduledExecutorService` set at a strict **250ms execution heartbeat**. Rather than flooding the CAN Bus with concurrent queries, the engine sequentially interleaves the requested Parameter IDs (RPM, Speed, Coolant, and Oil Temperature) into automated dispatch intervals, matching the physical bus bandwidth limits of automotive controllers.

3.2. Sequential Command-Response Transaction Pipelines

Automotive communication links over ELM327 RFCOMM serial channels are fundamentally half-duplex. The system enforces a rigid **Sequential Command-Response Transaction Pipeline** to block overlapping transactions. The thread worker must complete a full write-and-read exchange cycle:
$\text{Transmit Request} \longrightarrow \text{Hardware Propagation} \longrightarrow \text{Buffer Read} \longrightarrow \text{String Synthesis}$
before the next scheduled command can write to the `OutputStream`. If a specific register query encounters an execution delay, the pipeline stalls the scheduler thread until a timeout threshold is breached, ensuring no data packet collision occurs on the serial bus interface.

3.3. Main-Thread Graphical Buffer Interfacing via Reactive Architecture (LiveData Framework)

Once the background thread finishes parsing raw hexadecimal frames into clear numerical string metrics, the data must cross the execution boundary to the Android Main Thread for rendering. This interaction relies on a reactive, architecture-aware data distribution pattern driven by the **Android Jetpack LiveData Framework**:

- **Asynchronous Shifting:** The background worker passes values to the UI layer using `MutableLiveData.postValue()`, which safely routes data packages across thread pools.
- **Lifecycle Observability:** The presentation activities register lightweight, lifecycle-safe observers that listen to changes automatically.
- **UI Decoupling:** This structure completely decouples the data processing engine from graphic rendering lifecycles, ensuring the interface remains fully responsive at 60 FPS while background telemetry loops continue execution at different rates.

---

4. Defensive Boundary Conditions & Exception Containment

4.1. Handling Intermittent Network Noise and "NO DATA" Signal Frames

Automotive environments are inherently noisy, with physical networks prone to temporary signal drops due to electromagnetic interference, engine cranking, or heavy bus load. The application implements defensive parsing boundaries to safely intercept these structural anomalies:

- **Token Interception:** The `ResponseParser` actively scans every inbound string buffer for standardized SAE/ELM327 fault tokens, including `"NO DATA"`, `"CAN ERROR"`, `"BUS ERROR"`, and `"STOPPED"`.
- **Graceful Containment:** When a fault token is detected, the engine suppresses the throwing of a fatal runtime exception. It logs the network anomaly and immediately substitutes a safe fallback value (`"0"`). This mechanism isolates the transient bus failure, allowing the scheduling engine to preserve loop continuity and smoothly proceed to the next polling cycle.

4.2. Hexadecimal Range Validations and Bitstream Corruption Protection Gates

To protect the mathematical decoding engine from processing corrupted frames, the platform enforces strict structural verification gates prior to any data type transformation:

- **Structural Regex Filtering:** Inbound string buffers are stripped of operational artifacts (e.g., echo tokens, prompt characters `>`) and checked against predefined length thresholds (minimum of 6 characters for standard single-byte payloads).
- **Boundaries Checking:** The parsing logic validates that the hexadecimal data substring contains only valid characters (`[0-9A-Fa-f]`) before executing base-16 conversions.
- **Exception Safety Gates:** The entire conversion phase is wrapped inside a localized `try-catch` block targeting `NumberFormatException`. If a corrupted or malformed bitstream manages to bypass the regex filters, it is caught instantly at the parsing boundary. The system discards the corrupted frame, forces a safe default value, and shields the core thread pool from unexpected crashes.
