ChatGPT

Custom Truck Data Logging & Display Project

Project Overview

This project aims to build a **secondary digital dashboard** for a heavy-duty truck that supports both **SAE J1939 (CAN bus)** and **SAE J1708 (serial)** data links. The goal is to use a microcontroller system to **read and decode a wide range of engine, transmission, and ABS data** from the truck's electronic control modules (ECMs/TCM/ABS) and display them on a custom screen. This will provide more insight than the stock dash (e.g. engine load, boost, temperatures, fault codes, etc.), and also enable data logging and future integration (such as remote start via Home Assistant).

Key requirements include:

- **Capturing J1939 CAN-Bus data** (engine, transmission, etc. – e.g. Cummins ECM, Allison 1000 TCM).
- **Capturing J1708 serial data** (older subsystem data – e.g. ABS module on J1708).
- **Interfacing additional analog sensors** (aux fuel tank levels, dimmer, EGT, ambient temps, etc.).
- A suitable **microcontroller** (with Wi-Fi for future network integration) and interface hardware (CAN transceiver, RS485 transceiver for J1708).
- Decoding libraries or logic for **SAE J1939 PGNs/SPNs** and **SAE J1708/J1587 PIDs** to convert raw data into human-readable parameters.
- Phase-wise development: from bench-testing with mock data, to on-vehicle testing, data logging, basic text display, and later a graphical UI.

By the end, the microcontroller will output parsed data (initially via USB serial, later to a display) for parameters like: coolant temp, intake temp, exhaust temp, ambient temp; RPM, boost, load%, torque, horsepower; vehicle speed, gear, etc.; and diagnostic trouble codes (active engine/transmission/ABS faults). The system should be extensible for future remote monitoring and control. Below is a comprehensive specification covering **hardware**, **physical connections**, **software architecture**, and a **phase-by-phase implementation plan**.

Hardware Selection and Requirements

Microcontroller Choice

Given familiarity with the **ESP32** platform, an ESP32 is an excellent choice for this project. The ESP32 offers a dual-core microcontroller with built-in Wi-Fi (useful for future Home Assistant integration) and sufficient performance to handle multiple I/O tasks in real-time. Importantly, the ESP32 has an **integrated CAN controller** (often referred to as the TWAI interface) which supports CAN 2.0B at the hardware level [1] [2]. This means we only need to add a transceiver to interface with the truck's CAN bus. The ESP32 also has multiple UARTs (for J1708) and ADC channels (for analog sensors), making it suitable for this multi-protocol data acquisition.

Alternatives: Other microcontrollers with CAN support were considered (e.g. STM32 or Arduino Due/Teensy 4.1). A **Teensy 4.1** is very powerful and has dual CAN, but it lacks built-in Wi-Fi. An **Arduino Due** has CAN but is older and not as fast. Since the ESP32 meets requirements and adds Wi-Fi/Bluetooth, it's the preferred choice. Make sure to use an ESP32 variant that exposes the CAN TX/RX pins (ESP32 WROOM or WROVER modules do). For instance, the **ESP32 WROOM-32** module (commonly on dev boards) has a CAN controller (pins GPIO 21 = CAN_TX, GPIO 22 = CAN_RX in the ESP-IDF by default) – this can be leveraged with the right library.

The ESP32 will run the main firmware (likely developed using **C/C++** with Arduino framework or ESP-IDF, see Software section), handling data input from CAN, J1708, and sensors, and outputting decoded data.

CAN Bus Interface (SAE J1939)

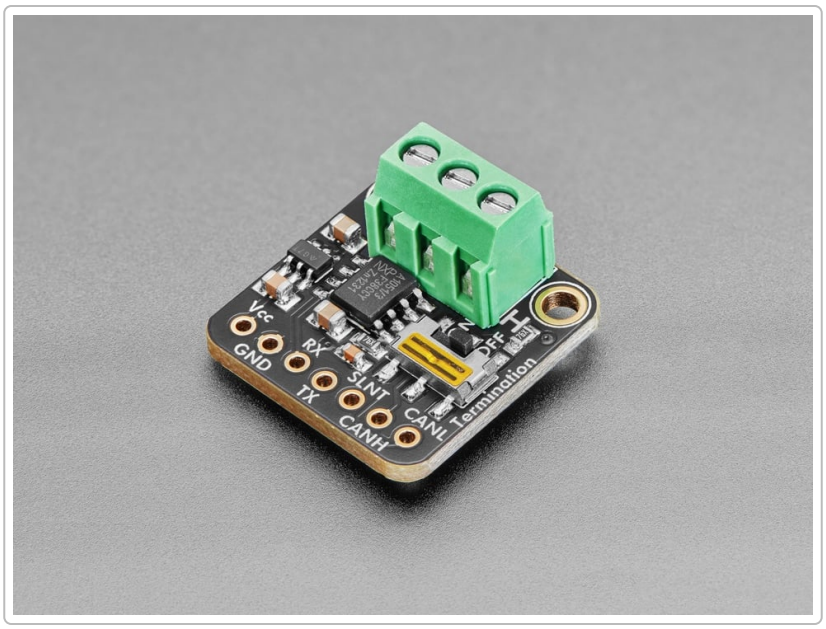
The truck's J1939 network is a **250 kbps CAN bus** (29-bit extended IDs) per the J1939 standard [3]. The ESP32's internal CAN controller operates at TTL logic levels and **must be paired with a CAN transceiver** to meet the ISO-11898 differential signaling and voltage levels [4].

Transceiver Hardware: Use a dedicated **HS-CAN transceiver** such as the **NXP TJA1051** or **Texas Instruments SN65HVD230/231/232** (3.3V compatible) or similar. These transceivers convert the ESP32's TX/RX CAN signals to the differential CAN High/CAN Low on the vehicle bus [2]. Many inexpensive modules or shields are available (e.g. an SN65HVD230 module). Additionally, **Adafruit's CAN Pal board** (based on TJA1051) is a convenient breakout that supports 3.3V logic, includes a built-in DC-DC to generate the required 5V for CAN bus drive, and even has switchable termination resistors [2]. Ensure whichever transceiver you use is powered appropriately (usually 3.3V logic, with on-board regulator if needed, or 5V if using a 5V-tolerant device) and is rated for automotive 12V/24V systems (most transceivers like TJA1051 are).

Termination: The J1939 bus should already have 120 Ω terminators at each end of the backbone (often inside the ECM and at the End-of-line connector). **Do not add an extra permanent terminator** with your device or it will disturb bus impedance. If your transceiver module has a termination resistor, disable or remove it unless your device will sometimes be the only node. Generally, when tapping into the truck's CAN (mid-bus via the diagnostic connector or existing wiring), no new termination is needed. (The Adafruit CAN Pal, for instance, allows enabling/disabling termination via a switch [5].)

Isolation (Optional): For additional robustness, some designs use isolated CAN transceivers to protect the microcontroller from transients/ground loops. This is not strictly required for a monitoring device, but good to keep in mind for harsh environments. At minimum, ensure proper grounding and perhaps transient suppression (TVS diode on CAN lines) for reliability.

CAN Bus Microcontroller Pins: The ESP32 CAN controller will use one TX and one RX pin. By default in ESP32 Arduino, the **CAN TX is GPIO 5 and CAN RX is GPIO 4** when using certain libraries (or GPIO 21/22 in ESP-IDF TWAI examples – this can be configured). We will map the ESP32's CAN TX to the transceiver's TXD, and CAN RX to transceiver's RXD. Also connect the transceiver's CANH and CANL to the truck (see Physical Connections below). Provide the transceiver with a stable 3.3V (or 5V as required) and ground.



Example of a CAN bus transceiver breakout (Adafruit CAN Pal board with NXP TJA1051). It converts 3.3V logic from the MCU into differential CAN signals (CANH/CANL) and includes an optional termination resistor (switchable) [2] *. Use a similar transceiver to interface the ESP32 to the truck's J1939 CAN bus.*

J1708 Interface (SAE J1708/J1587)

J1708 is an older serial communication bus used in heavy vehicles, often for diagnostics and subsystems (commonly running the **J1587 protocol** on top). It's a **two-wire differential bus similar to RS-485**, operating at **9600 bps** [3] . The physical layer is essentially RS-485 with some collision detection mechanism [6] .

To interface the ESP32 with J1708:

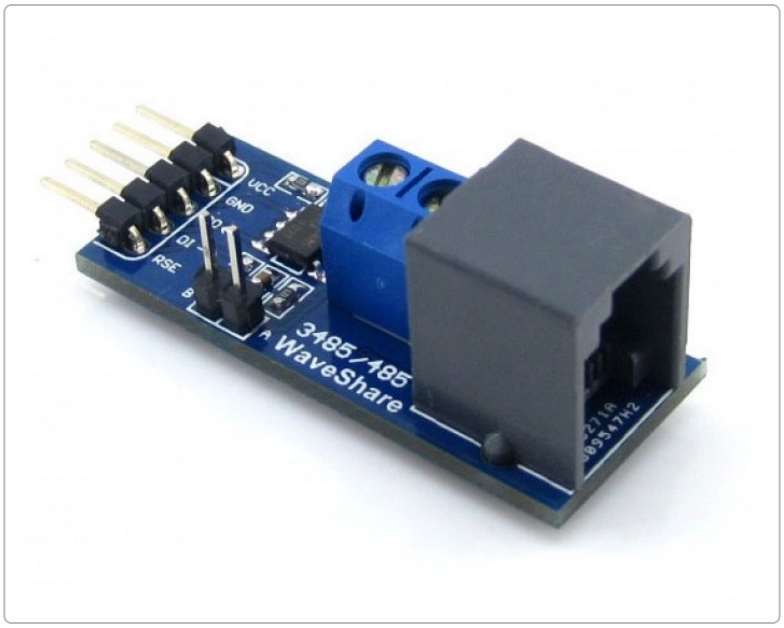
- Use an **RS-485 transceiver** (e.g. MAX485, TI SN75176, SP485, or any 5V/3.3V RS485 interface chip). This will convert between the differential J1708 bus lines and the ESP32's UART TX/RX lines.
- Many RS485 breakout boards are available; for example, the MAX485 board can run at 5V (some variants at 3.3V) and provides DI (driver in), RO (receiver out), and DE/RE (driver enable, receiver enable) pins [7] . Connect the transceiver's A/B pins to the truck's J1708+ and J1708- lines (through the diagnostic port or wiring, see below).

UART Connection: The ESP32 has multiple UARTs (UART0 is often for programming/serial monitor, so use UART1 or UART2 for J1708 to avoid conflict). For example, you could use **UART1 (TX on GPIO 4, RX on GPIO 36)** or any available pins for a hardware serial port. Connect the RS485's RO (receive out) to the ESP32's RX pin, and DI (driver in) to the ESP32's TX pin.

Driver Enable: RS485 is half-duplex, so the transceiver has a Driver Enable pin (and typically Receiver Enable). To **listen only (sniff)**, you can keep the transceiver in receive mode at all times (RE low, DE low on MAX485 to enable receiver and disable transmitter). If you want the ability to transmit on J1708 (e.g. to request data or acknowledge), you'll need to control DE/RE via an ESP32 GPIO. A common approach is tying RE and DE together and controlling them with one output: drive it HIGH to transmit, LOW to receive. The

Copperhill example for Arduino used a GPIO to toggle the RS485 transceiver's DE/RE line [8]. For initial monitoring, it's safe to set the transceiver to read-only to avoid any bus interference (J1708 can have collisions if two devices send simultaneously [6]). When we do implement sending (perhaps to request specific J1587 data or for remote commands), ensure to implement collision avoidance per J1708 spec (listen before talk).

Voltage Levels: Most RS485 transceivers are 5V devices, but you can find 3.3V versions or simply power at 5V and use level shifters if needed. Many MAX485-type chips will output TTL at 5V, which a 5V-tolerant ESP32 UART RX could handle, but to be safe use a 3.3V compatible transceiver (e.g. MAX3485) or a simple resistor divider on RO. Alternatively, some 5V transceivers output ~3.3V logic high which might be marginal – best to use proper logic levels.



Example of an RS485 interface module (SP485/MAX485 based). Such a transceiver board connects to the microcontroller via TX, RX, and a driver-enable pin, and to the J1708 bus via the A/B differential lines [7]. This allows the ESP32 to read and write J1708 serial data (at 9600 bps) safely.

Additional Sensors and I/O

Beyond the bus data, the system will monitor additional parameters via direct sensor inputs. We should plan for several analog inputs and possibly digital I/O for future expansion (like controlling a relay for remote start). Key extra sensors mentioned:

- **Auxiliary Fuel Tank Level Sensors:** Typically these are resistive float sensors. For example, many vehicle fuel senders are in ranges like 240–33 Ω or 0–90 Ω. The microcontroller cannot measure resistance directly, so we will use a **voltage divider** approach: connect the sensor in series with a known resistor and measure the voltage drop. For instance, tie one end of the sensor to a stable reference voltage (e.g. 3.3V from ESP32 or an external 5V), the other end to ground through a resistor, and measure the voltage at the sensor node. The reading can be converted to resistance, then to a fuel level percentage by calibration. Alternatively, if the truck's existing gauge or ECM

already provides a voltage on the sensor line, we could just **read that voltage** (buffered to avoid loading it). Ensure the ADC input has high impedance or use a unity-gain buffer op amp if needed, so as not to disturb the gauge circuit. **Multiple fuel tanks** mean multiple ADC inputs (ESP32 ADC1 has several channels). We should reserve an ADC channel for each tank sensor (two or three).

- **Instrument Dimmer (Dash Dimmer):** Likely a variable voltage (0 to ~12V) controlled by a potentiometer (rheostat) to dim dashboard lights. This can be read to allow the custom display to dim in sync with the dash. We'd use a voltage divider (to scale 12V down to <3.3V) and read it via an ADC. For example, a simple 3:1 divider (using ~100kΩ total for minimal current) can scale 0–12V into 0–4V, which is within a safe range if using the ESP32's ADC (which maxes ~3.3V; slight over-range can be clamped with a diode or just choose resistors for 0–3V span). This ADC reading gives the dimmer knob position, which we can later use to adjust display brightness via PWM or screen settings.

- **Exhaust Gas Temperature (EGT) Sensor:** For EGT, a **Type-K thermocouple** is the standard. Many aftermarket EGT probe kits come with a **K-type thermocouple** in a 1/8" NPT threaded fitting for installation in the exhaust manifold or pipe. These can measure up to ~1200°C (e.g. ~1300°C/2300°F max) [9] and provide a small voltage proportional to temperature. To interface with the microcontroller, use a **thermocouple amplifier/ADC** such as the **Maxim MAX6675 or MAX31855**. These ICs connect to the thermocouple leads and output a digital temperature reading over SPI (with cold-junction compensation). For example, the MAX31855 supports K-type and can read from -270°C to +1370°C, perfect for EGT. They are available as ready-made modules that connect via SPI. Alternatively, analog amplifiers like the AD8495 (which outputs 5 mV/°C analog) could be used and read via ADC, but the digital SPI solution is more straightforward and noise-immune. Plan for **one thermocouple amplifier channel per EGT sensor** (initially one EGT probe, but if you ever wanted multiple, ensure the microcontroller has enough SPI chip-selects or use an analog switch multiplexer). The EGT probe itself (1/8" NPT threaded K-type) can be sourced readily [10] – ensure it comes with the mating compression fitting and that the probe wire length is enough to reach the microcontroller's location (or extend with K-type thermocouple wire).

- **Ambient Air Temperature (Outside Air) and Cabin Temperature:** For these lower temperature readings, using thermocouples would work but is overkill. Instead, using digital temperature sensors or simple thermistors is easier. **Digital sensors** like the **DS18B20** (1-Wire bus, ±0.5°C accuracy) allow easy placement (e.g. one sensor mounted outside the cab, one inside). These only require one digital input pin for the one-wire bus (multiple sensors can share the bus) and are easy to read with existing libraries. They can operate from 3.3V and measure -55°C to +125°C, which covers ambient ranges well. Alternatively, an analog **NTC thermistor** (like a 10k @25°C type) with a resistor divider to ADC can be used for ambient temperature. However, DS18B20 or even I2C sensors (like BME280 for temp/humidity/pressure if desired) provide calibrated readings easily. We'll specify using **two DS18B20 sensors** – one routed to a shaded location outside (for outside air temp) and one inside the cab for interior temp (useful if we later implement climate monitoring or remote start conditions).

- **Other Analog Sensors:** The spec also mentions leaving headroom for more sensors – e.g. perhaps an **additional pressure sensor** (if needed for something like oil pressure if not on CAN, or boost if better resolution needed) or other temperatures (transmission oil temp if no data, etc.). The microcontroller should have a few ADC channels free for expansion. We should also plan some spare **GPIOs for digital inputs/outputs**: e.g. a digital input could monitor an ignition key-on signal (though we have ACC powering the unit, so that implies key-on already), and a digital output could

drive a **relay for remote start**. For example, one could connect a relay across the starter signal or use the controller to send a J1939 start request if the engine ECM supported it (though typically remote start is easier done via a physical relay). In any case, including a **transistor or relay driver circuit on a GPIO** (with proper flyback diodes and using a FET or ULN2003 driver IC) is wise if future start/stop automation is planned.

- **Power Supply:** (For completeness) The microcontroller and sensors will be powered from the vehicle's ACC/Run 12V supply (as noted). A robust DC-DC regulator (12V to 5V/3.3V) will be used to power the system. Ensure it can handle automotive voltage spikes and noise. A buck converter module or automotive-rated LDO (for lower current sections) can be used. Also include filtering (capacitors, perhaps a transient suppressor) to protect the electronics. Since the user will handle power design separately, we won't detail it further, but just ensure the microcontroller and transceivers get stable regulated supply in the appropriate voltage.

Summary of Hardware Components:

- *ESP32 dev board* (e.g. ESP32 DevKit or a custom ESP32 board) – main controller.
- *CAN Bus transceiver* (e.g. SN65HVD230 or NXP TJA1051-based module) – for J1939.
- *RS485 transceiver* (e.g. MAX485 or SP3485 module) – for J1708.
- *Thermocouple amplifier* (e.g. MAX31855K) and *K-type EGT probe* (1/8" NPT) – for EGT.
- *Temperature sensors:* 2× DS18B20 (for outside & inside ambient temps).
- *Resistor divider networks* and wiring for fuel tank sensors & dimmer to ADC.
- *Miscellaneous:* connectors (for diagnostic port tapping), wiring harnesses, a small prototyping PCB or shield to mount all components in the vehicle, an SD card module if logging to local storage is desired, and a basic display (for Phase 4+, e.g. a simple character LCD or small TFT for text output).

Physical Connections and Installation

Physically interfacing with the truck is critical. The truck likely has a standard **9-pin Deutsch diagnostic port** (round connector) under the dash. This port provides access to both J1939 CAN and J1708 wiring. According to the SAE J1939-13 standard, the pinout of the 9-pin connector is as follows:

[11] **9-pin Deutsch (J1939/J1708) Diagnostic Connector Pinout:**

- **Pin A:** Ground (return)
- **Pin B:** Battery + (unswitched +12V or +24V supply)
- **Pin C:** J1939/CAN 1 High (CAN_H)
- **Pin D:** J1939/CAN 1 Low (CAN_L)
- **Pin E:** CAN Shield (grounded shield for CAN, if used)
- **Pin F:** J1587/J1708 + (positive line)
- **Pin G:** J1587/J1708 – (negative line)
- **Pin H:** J1939/CAN 2 High (if a secondary CAN present, on newer Type II connector)
- **Pin J:** J1939/CAN 2 Low

Most likely, this truck being older uses **pins C and D for the main J1939 bus**, and **pins F and G for the J1708 bus** [11]. Pin B can supply power (though typically that's battery power – since we prefer ACC power,

you might splice into an ACC circuit instead for the microcontroller, to avoid draining battery when ignition is off). Pin A is ground and should be common to our device ground.

Connecting the Microcontroller to the Vehicle: We have two main options:

1. **Via the Diagnostic Port:** Easiest non-intrusive method – use a **Y-cable** or adapter from the 9-pin port. For example, you can obtain a 9-pin Deutsch male to female Y-cable: one branch goes to your Nexiq (if you still want to use it), and the other branch to our device. This allows concurrent connection. Otherwise, you can unplug the Nexiq and plug in our device as needed. Connect:
2. CAN transceiver's CAN_H to Pin C, CAN_L to Pin D.
3. RS485 transceiver's "A" (non-inverting) line to Pin F, "B" (inverting) line to Pin G.
4. Device ground to Pin A.

5. (Optionally device power to Pin B if you want to use battery power – but a better approach is to use an **ignition-switched source** so the device only runs with key on. If Pin B is constant 12V, you might not want to draw from it unless you implement a low-power sleep or ensure to turn off with ignition. Alternatively, find an ACC fuse tap and feed your regulator from that.)

6. **Direct Wiring:** Tap into the CAN and J1708 wires in the dash harness or at the ECM/TCM connectors. Heavy vehicles often have twisted pair wiring for J1939 (usually green and yellow wires) and a pair for J1708 (often orange or gray wires). If you identify these, you can splice into them. However, using the diag port is simpler and reversible.

Ensure a **common ground** between the microcontroller and vehicle. The ESP32 ground (and RS485/CAN ground references) should tie to vehicle ground (Pin A or any chassis ground) to ensure proper signal reference.

Mounting: The microcontroller board and associated circuitry should be enclosed in a small project box. This box can be mounted under the dash. The thermocouple amplifier and other sensor interfaces can be inside the same box. The thermocouple probe will be routed through the firewall to the exhaust manifold (drill & tap 1/8" NPT if not already present, and insert the probe). Outside air temp sensor can be mounted on the front of the vehicle (out of direct sunlight/wind), wired back inside. Inside temp sensor can be simply placed inside the cab (hidden in dash or headliner). Fuel tank sensor wires would be tapped near the existing gauge wiring or at the tank senders.

EMI/Noise considerations: Route the CAN and J1708 connection wires twisted if possible, especially if extending from the diag port. The length is short under dash, so it should be fine. The thermocouple leads (usually provided as shielded or twisted pair) should remain away from ignition coils or high-current lines to avoid noise – using the amplifier's digital output mitigates some interference issues. Likewise, analog sensor wires (fuel, dimmer) might benefit from some filtering (simple RC filter or software averaging) due to vehicle electrical noise or fuel slosh in case of fuel level.

Pinout Summary for Microcontroller: Here's a proposed mapping of microcontroller pins to functions (this can be adjusted based on the board and library used, but for concreteness):

- **ESP32 CAN TX/RX → CAN Transceiver:** e.g. GPIO 21 (TX), GPIO 22 (RX) to transceiver. (These correspond to ESP32's built-in CAN TX/RX in ESP-IDF; in Arduino library it might use different default pins – adjust accordingly.)
- **ESP32 UART1 (TX/RX) → RS485 Transceiver (J1708):** e.g. use UART1 with TX on GPIO 17, RX on GPIO 16 (for instance). Connect TX->DI, RX<-RO. Also GPIO 5 as DE/RE control for RS485 (set LOW for listen, HIGH when transmitting).
- **ESP32 ADC channels for analog sensors:**
- ADC1_CH0 (GPIO 36) for Fuel Tank 1 sensor.
- ADC1_CH3 (GPIO 39) for Fuel Tank 2.
- ADC1_CH6 (GPIO 34) for Fuel Tank 3 or spare.
- ADC1_CH4 (GPIO 32) for Dimmer input (with voltage divider).
- (ESP32 has ADC2 as well, but ADC2 channels can't be used when Wi-Fi is on. So prefer ADC1 channels for these sensors to avoid Wi-Fi interference issues.)
- **One-Wire bus for DS18B20:** Use a GPIO (e.g. GPIO 25) with a 4.7k pull-up resistor to 3.3V to connect both outside and inside DS18B20 sensors.
- **SPI for Thermocouple (MAX31855):** Use VSPI or HSPI on ESP32. For example, SCLK on GPIO 18, MISO on GPIO 19, CS on GPIO 5 (just an example; adjust if DE for RS485 used GPIO5, pick another CS like GPIO 15). If multiple thermocouples later, use multiple CS lines.
- **Digital outputs/inputs:** e.g.
- GPIO 33 for a cooling fan or warning LED if needed,
- GPIO 26 for a relay control (starter relay for remote start, etc.),
- GPIO 4 for an input from an IGN sense (if we want to detect key position beyond just power supply).

- Keep some spares.

- **Display connection:** (Phase 4-5) If using a simple serial or I2C character LCD, allocate I2C pins (ESP32 default I2C on GPIO 21 SDA, 22 SCL – but those might conflict with CAN if using same, we can remap CAN to other pins since ESP32 CAN TX/RX can actually be on GPIO 5 and GPIO 4 for instance with the CAN driver library as referenced by many examples [1]. We'll plan carefully to avoid conflicts). If using an SPI TFT, use the SPI bus with another CS for the display. Since UI is a later phase, just note we have options.

Finally, ensure **all grounds are common** (sensor grounds, transceiver grounds, ESP32 ground tied to vehicle ground). Also, decouple power to each module (place 0.1 µF and say 10 µF capacitors near transceivers, etc.) to smooth out any noise.

Software Architecture and Data Decoding

The software will be structured to handle **multiple data streams** concurrently: the J1939 CAN bus frames, the J1708 serial bytes, analog sensor readings, and output tasks (sending data to serial or display). Using the ESP32, we can leverage its dual-core and FreeRTOS capabilities to separate tasks (for example, one task can handle CAN receive and parsing, another handles J1708 parsing, another updates the display or logs data). However, even a simpler loop with non-blocking reads and interrupts could suffice.

Language and Platform

We will use **C/C++** for implementation, likely with the **Arduino framework** on the ESP32 (for simplicity of libraries and familiarity). Arduino-ESP32 integrates FreeRTOS under the hood, so we can create tasks or use loop scheduling as needed. Alternatively, the native **ESP-IDF** could be used for more control, but Arduino libraries for J1939/J1708 exist which simplifies development. Python is not ideal for on-device in this case (MicroPython could theoretically be used for simpler CAN tasks, but robust J1939 decoding in MicroPython would be challenging). Thus, we'll proceed with C/C++ and make use of existing **libraries** for protocol decoding where possible.

J1939 Data Handling

Overview of J1939: SAE J1939 is a higher-layer protocol over CAN bus, used extensively in heavy-duty vehicles for realtime data exchange [12]. It uses the extended 29-bit CAN ID and organizes messages by **Parameter Group Numbers (PGNs)**, with each PGN containing one or more **Suspect Parameter Numbers (SPNs)** which are the actual measured parameters (engine speed, coolant temp, etc.) [13]. Most J1939 messages are broadcast continuously (e.g. engine data broadcast at, say, 50ms or 100ms intervals), and some are on-request. The bus speed is typically 250 kbps in our truck [14]. We will be primarily *listening* to J1939 broadcasts.

Library/Stack: Implementing full J1939 decoding from scratch is complex (there are hundreds of PGNs and SPNs defined). Fortunately, we can leverage existing J1939 stacks: - **ARD1939** – a SAE J1939 protocol stack originally for Arduino (Uno/Mega) and extended to ESP32 [15] [16]. This is referenced by Copperhill Technologies. Copperhill provides examples of a J1939 network scanner and J1939-to-USB gateway, and a precompiled ARD1939 stack [17]. The ARD1939 is robust, but the full source might not be open (pre-compiled to respect licensed protocols) [18]. Still, they have provided an Arduino library or at least a compiled library for ESP32 that can be used. - **SimpleJ1939** – an open-source library (GPL-3.0) on GitHub by vChavezB [19]. It was built for MCP2515 CAN but can likely be adapted to ESP32's internal CAN. It's a simplified stack focusing on sending/receiving J1939 formatted frames, based on Copperhill's example code [20]. It might not cover decoding all SPNs, but provides a framework to deal with J1939 message assembly/disassembly. - **Alternatively**, we could use lower-level CAN libraries (ESP32 has an **ESP32CAN** library or using the built-in Twai driver in ESP-IDF) and then do custom parsing. If a full database of J1939 parameters (like a **DBC file** or the J1939 Digital Annex) is available, we could map PGNs to parameter definitions. However, given time, a library approach is easier.

Receiving J1939: We will configure the ESP32 CAN controller at 250 kbps, extended frame format. Use an existing **CAN driver** (e.g. the **ESP32CAN** library from Collin80 or similar, which was used in Copperhill's code [21]). This driver will handle receiving CAN frames (ID + 8 data bytes). We then feed these frames into the J1939 protocol handler. The handler needs to manage: - **Transport Protocol (TP):** J1939 messages longer than 8 bytes use a transport protocol (BAM or RTS/CTS). A good stack will handle assembling multi-packet messages (like diagnostic messages DM1, etc., which can be longer). ARD1939 likely covers this, as J1939 can go up to 1785 bytes in multi-packet [22]. - **PGN filtering:** We may not need to receive absolutely every PGN. But initially, since the goal is to get "as much parameter as possible," we might accept all and then filter in software which SPNs to use. However, for performance, it might be wise to filter out unused messages if the bus is very busy. We can start open and then refine. - **Decoding SPNs:** The library or code should translate raw data bytes to meaningful values. For example: - **Engine Speed** is typically PGN 61444 (Engine Electronic Controller 1, EEC1). Within that PGN, **SPN 190** = Engine Speed, which is two bytes at a

scale of 0.125 rpm/bit and offset 0 [23]. So if the data bytes for SPN 190 come in as (for example) 0x13 0x88 (hex) which is 5000 in decimal, that would be 5000 * 0.125 = 625 rpm. - **Engine Coolant Temperature** is PGN 65262 (Engine Temperature 1). SPN 110 is coolant temp, 1 byte, with scale 1°C/bit, offset -40°C [23]. So a value of e.g. 100 (0x64) means 60°C. - There are many such parameters: fuel temperature, oil pressure, boost pressure (often called intake manifold pressure – PGN 65270, SPN 102 for boost in kPa), etc. We will rely on either the library's built-in SPN conversion or manually implement conversions using the J1939 spec values (we can extract formulas from the J1939 standard or public summaries as needed).

Diagnostic Trouble Codes (DTCs): Active fault codes on J1939 are reported via **DM1 messages** (PGN 65226). DM1 contains a list of active DTCs, each identified by an SPN and an FMI (failure mode indicator), plus occurrence count [24]. Our code should listen for PGN 65226 and parse any DTCs. We can display them as "SPN/FMI" numbers or even translate to text if a lookup table is available. (Translating SPN to meaning might require the J1939 standard list or manufacturer info, which is beyond scope initially, but listing the numeric code is a good start).

J1939 Request Support: Some data might not be broadcast until requested (e.g. some vehicles require a request for certain info like odometer or VIN). The project's initial goal is to read what's readily available. Later, we can implement sending a **Request PGN (PGN 59904)** for specific PGNs, if needed, to retrieve things like VIN (PGN 65260) or other rarely broadcast data. If so, the microcontroller would need to occasionally transmit on the bus. This is fine, but it should be done in a way that doesn't conflict (J1939 has message arbitration – our device would just act as another node with its own source address, maybe using an arbitrary address 0x80 if free, or using address 0xFE for a tool). This is advanced and can be added in Phase 3+ when we have basic reading working.

Libraries Integration: The **Free J1939 stack by Copperhill (ARD1939)** might be ideal if accessible. According to Copperhill, they have a **J1939 Network Scanner** and a **J1939->USB gateway** example [17]. We can use the network scanner sketch as a starting point – it likely listens to all PGNs and prints them with decoded values. That would already fulfill the "USB serial output of all parsed data" requirement for initial testing. ARD1939 on ESP32 may require the use of their modified CAN driver (they mentioned some modifications to the Due CAN library for ESP32 [25], but those might be abstracted away for us).

If ARD1939 is not directly accessible as source, we could use the **Arduino-SAE-J1939 library** published by **SAE jCOM (Simma?)** or others. There is mention of a **jCOM1939 library** for Arduino/ESP32. Given that simplicity is key, we will attempt to find an open library first. If not, writing a limited parser for key PGNs is an option.

Output of Data: Initially, we will output all relevant parameters to the serial console (over USB). This output will list parameter names and values. For example, the microcontroller could print a line like:

```
EngineSpeed=1500 rpm, EngineLoad=75%, BoostPressure=20 psi, CoolantTemp=85°C,
TransTemp=90°C, ...
```

updated in real-time. We'll include as many as feasible. If using a library, it may have a callback for each received message – we can then format the known SPNs to text. Unknown or unused messages can be ignored or logged for analysis.

J1587 (J1708) Data Handling

Overview of J1708/J1587: SAE J1708 defines the physical layer (RS-485 at 9600 bps) and SAE J1587 is the message protocol on it. J1587 messages are transmitted in a simple byte-oriented format (usually a *MID* = Message ID as the first byte indicating which ECU or system, followed by data bytes and a checksum). The maximum message length is 21 bytes [26], and they often use ASCII or scaled data in a simpler way than J1939. J1587 has its own set of PID (Parameter IDs) for data. For instance, engine speed, coolant temp, etc. had J1587 PIDs (though in a J1939-equipped engine, those might not be heavily used, depending on the year). More importantly, **diagnostic codes** (DTCs) were often reported via J1587 on older systems. For example, the ABS likely sends J1587 fault codes. Allison transmissions (if older) could also have J1587 data. However, since our transmission and engine use J1939, we mainly want J1708 for the ABS. The ABS module's active faults can be read via J1587 PID 65 (ABS fault code, as an example), or it might just broadcast them periodically.

Library: We will use an **Arduino J1708 library** if possible. There is one by Jeremy Daily on GitHub [27] (MIT License) that handles reading bytes from a serial port and assembling J1708 messages. This library likely provides callbacks or structures for complete messages. If not available, implementing J1708 parsing manually is feasible: - Continuously read bytes from the UART at 9600 baud. - Identify message boundaries (J1708 messages end when a checksum byte is verified or a silence on the bus for a certain period – typically 2 byte-times of silence indicates end of message). - Validate checksum (J1587 uses a checksum: sum of all bytes incl. MID and data and checksum = 255 (0xFF) mod 256). - Once a full message is captured and verified, interpret it: The first byte is MID (e.g. MID 128 might be engine, 130 trans, 140 ABS, etc.). Following bytes are in the form PID/value. We would need a J1587 PID list to decode meaning. For instance: - PID 84 = Wheel Sensor ABS Left Front Speed (just hypothetical example), - PIDs for fault codes might be in the 128+ range containing SID or FMI, depending on how ABS reports. - A *simple approach* initially: We can print out the raw MID and PIDs data in hex or decimal to see what's coming from ABS. With the actual data captured, we can refer to an SAE J1587 reference to decode. Since J1587 is older, a lot of that info is publicly documented or at least accessible in older manuals.

Likely J1708 usage in our case: The Bendix or Meritor ABS ECU in a pre-2005 truck often only had J1587 communications. It would broadcast an ABS warning or fault. For example, **MID 137** might be ABS (just as an example, actual MID for ABS could be 128 or 130 – need to check J1587 MID list). It may periodically send out a message if there's an active wheel speed sensor fault, etc. We will capture these. If possible, we'll decode to human-readable (e.g. "ABS Active Fault: Wheel Speed Sensor RR"). This requires knowing the SID (Subsystem Identifier) and FMI (like J1939's SPN/FMI, J1587 has a concept of fault codes as well, often a PID for the error code and one for the occurrence count).

Implementing Reading: We configure the chosen UART at 9600 baud in 8N1 mode. The J1708 library (if used) will give us messages. If not, we use an interrupt-driven or loop-driven read: - Use a small buffer, keep adding incoming bytes. - Reset buffer when a gap in reception indicates message end (alternatively, the library might do this). - Compute checksum to verify message integrity. - Then parse by looking at known PIDs.

Decoding PIDs: To maximize information, we should be prepared to decode at least common ones: - Engine (if any, but since engine is on J1939, it might not broadcast on J1587 concurrently in this vehicle year – though some transitional years did dual broadcasting). - ABS: likely has PIDs for wheel speeds, or just uses J1939? If the ABS is truly only J1708, it might not broadcast wheel speeds to dash, just faults. Many ABS

modules only report faults over J1587 and rely on J1939 for wheel speed sharing if needed. However, given user's note, the ABS codes are what we want. - There could be other modules on J1708 like older body controller, but if not needed we focus on ABS.

We might not get a ton of *sensor* data from J1708 aside from diagnostic info. But to be sure, after hooking up, we will **log raw J1708 data** (Phase 3) and decode further as needed.

We'll include the Jeremy Daily J1708 library in the project and use it to parse frames, then handle them in a callback where we can print or save the PIDs.

Data Logging and Storage

During development (Phase 3), we plan to **record raw data** from both buses to analyze coverage. We can implement a logging function: - For J1939: log every received PGN (with timestamp and raw bytes, plus decoded if possible). - For J1708: log every message (MID and bytes). This can either go to a connected PC via serial or be stored on an SD card if portability is needed. Since we have an ESP32 with possible filesystem or an SD card slot, we might integrate an SD card to store logs in CSV or binary format.

However, initially simply streaming to serial (which can be captured by a terminal program) is sufficient. The user mentioned possibly using the Nexiq to log data – but once our device is working, it can itself serve as a logging tool.

Display Output (Phase 4 and 5)

Once data capture and decoding are verified, we will attach a **display** to show selected parameters in real-time. For Phase 4 (basic text UI), a simple solution is a standard **LCD (16x2 or 20x4 character display)** or a small **OLED (e.g. 128x64 monochrome)** for textual output. For example, a 20x4 character LCD over I2C (which only uses two pins) could display a set of readings, updating every second or so. We might arrange pages (e.g. Engine page, Trans page, etc.) or a rotating display due to limited characters. But since Phase 5 intends a more advanced UI, Phase 4 can be minimal (even just one page listing some values continuously).

We won't delve deeply into UI design here, but keep in mind the categories of data to display (as requested):

- **Engine Metrics:** RPM, Engine Load %, Boost Pressure, Instantaneous fuel economy (MPG) if available (J1939 does broadcast fuel rate, so MPG can be computed with speed), Calculated Torque and Power (some J1939 ECUs broadcast current torque % and we can estimate HP from torque*RPM), etc.
- **Temperatures:** Engine coolant temp, Oil temp (if available), Intake air temp (post-intercooler), maybe Exhaust gas temp (from our sensor), Ambient temp (from our sensor), Transmission fluid temp.
- **Transmission:** Current Gear, Vehicle Speed, possibly torque converter lockup status or "overdrive on/off" (Allison might have a signal or we infer from gear vs commanded gear), etc. J1939 has a PGN for Transmission Actual Gear and Torque Converter status (PGN 65272 Transmission State, for instance).
- **Fuel Level:** If the truck's fuel level is available on J1939 (PGN 65276 has fuel level as SPN 96) we could read it, but since we also have direct sensors for aux tanks, we display those too.

- **Diagnostics:** Active DTCs from engine, trans, ABS. We can show an icon or message if any active fault. Pressing a button could show details (SPN/FMI or J1587 code). In Phase 4 text mode, we might cycle through any active code and print them.

For Phase 5, a graphical UI can be planned using something like **TFT LCD + LVGL (LittleVGL)** or even an embedded web dashboard via Wi-Fi. But since that's for future expansion, we'll focus on ensuring our architecture can supply the needed data easily to whatever UI.

Software Modules & Stack Summary

To summarize the software components:

- **CAN Driver**: Low-level CAN interface to receive/transmit frames. (e.g. ESP32CAN library or built-in TWAI driver).
- **J1939 Stack**: Handles message assembly (especially multi-packet), PGN identification, and provides either callbacks for specific PGNs or a way to query last value of SPNs. (Potentially using ARD1939 or similar).
- **J1939 Application Decoder**: Our code that uses the stack to extract needed SPNs and scale them. We will create a dictionary or a set of structures for each parameter we care about, with its SPN, scaling, etc., if the library doesn't already do scaling.
- **J1708 Driver**: UART interface at 9600bps.
- **J1587 Parser**: As part of library or custom, to assemble messages and extract PIDs.
- **J1587 Application Decoder**: Functions to handle known PIDs (especially for ABS codes). For example, if MID corresponds to ABS, and a certain PID indicates an active fault code, we interpret it.
- **Sensor Readers**:
- Analog sensor reading (use ESP32 ADC API to read values periodically, apply filtering).
- Thermocouple reading (SPI interface to MAX31855, get temperature in °C).
- 1-Wire reading for DS18B20 (using DallasTemperature Arduino library or similar).
- **Data Manager**: Possibly a task or loop that combines data from all sources into a coherent set of "latest values" for each parameter. This could simply be global variables or a struct updated by each respective input handler.
- **Output Handlers**:
- Serial output: for debugging/logging. Could be simply printing in the loop or via a separate debug task.
- Display output: in Phase 4, update the LCD/OLED with selected values every X milliseconds. In Phase 5, a GUI library task will handle drawing, and we feed it the values.

- Future network output: the ESP32 could send data via Wi-Fi. For Home Assistant, the common way is via **MQTT** or **HTTP API**. We might prepare by structuring data such that it's easy to publish. For instance, a routine that formats a JSON or a series of MQTT topics like `truck/engine/rpm` etc. For now, we won't implement it, but keep the code modular.

- **Error Handling & Redundancy**: The code should handle cases like bus not present or message missing. For example, if J1708 is not connected, the task might just time out reads – that's fine. If J1939 doesn't provide a certain SPN (like perhaps the truck might not have an exhaust temp on J1939), our display should handle that gracefully (show N/A or blank). Also, ensure the microcontroller doesn't reset due to buffer overflow if traffic is heavy – use adequate buffer lengths

and possibly drop frames if the processing can't keep up (though at 250k and typical message rates, ESP32 should handle it, given it's done in many OBD projects).

Phase-wise Implementation Plan

We will implement the project in several **phases**, each building on the last. This ensures we validate each part step by step:

Phase 1: Bench Testing with Simulated Data

Goal: Develop the parsing logic for J1939 and J1587 using **mock data**, before touching the actual truck. This will verify that our software correctly decodes known values.

- **Obtain or Simulate J1939 data:** We can either use recorded data (if available from a similar engine) or manually craft some example CAN frames that correspond to known PGNs. For instance, simulate PGN 61444 (EEC1) with a certain RPM value, PGN 65262 with a coolant temp, etc. The ARD1939 library examples might include a "J1939 Network Simulator" that generates dummy data [17]. If available, run that on a second microcontroller or on the ESP32 itself in a test mode to feed the J1939 parser. Alternatively, we could use a PC tool (like a CAN simulator) to send frames to the ESP32 (if we have a CAN interface).

- **Simulate J1708 data:** Write a small Arduino sketch (maybe on an Arduino Uno connected to the ESP32's RX) that periodically sends a J1587 message. For example, craft a fake ABS message: MID 140 (ABS), then a couple of data bytes and checksum. Or simpler, use the ESP32's second UART in a loopback test where it sends and then reads its own messages (though collision detect might complicate that). Even just verifying that the J1708 library can assemble and parse a known example string is good. If documentation is available, we can use a known example from a J1587 spec (e.g., MID 128 Engine sending PID 84 = "Road Speed 65 mph") to test decoding.

- **Implement decoding logic:** In this phase, print the decoded values to the serial console in a human-readable format. Cross-verify against the expected values from the test data. For instance, if our mock frame had engine speed = 1200 rpm, ensure the output says 1200 rpm.

- **Parameter Catalog:** Create a structure listing all the parameters we intend to decode:

- Engine: RPM, Engine Load, Throttle Position, Engine Torque %, Engine Power, Fuel Rate, Coolant Temp, Oil Pressure, Oil Temp, Intake Manifold Temp, Boost Pressure, Battery Voltage (if available on J1939), etc.
- Transmission: Gear, Transmission Oil Temperature (Allison typically broadcasts this), Torque Converter Lockup (maybe as a status bit in some PGN), Output Shaft Speed.
- Vehicle: Speed (could come from engine ECU or ABS ECU on J1939 as wheel-based vehicle speed), Odometer (if available via PGN 65248, etc.).
- Ambient: maybe outside temp from engine ECU if it has sensor (or from our sensor).
- Fuel: Fuel level (if on J1939 SPN 96) plus our auxiliary tank sensor readings (we'll output those as well, though those are just ADC values scaled to percentage).
- ABS: Possibly wheel speeds (if ABS shares them on J1939, some do via PGN 65265 "ABS Wheel Speed Information" on newer vehicles, but if not, we skip), mostly ABS fault code.

- DTCs: Engine and Trans via J1939 DM1, ABS via J1587. We should be prepared to output at least the identifiers.

We don't necessarily implement *all* of these in Phase 1, but design the code structure to make adding them easy, and maybe simulate a subset.

- **Success criteria Phase 1:** When feeding known values, the system prints correct decoded results and handles the variety of data without crashing. The parameter catalog and conversion formulas are verified with sample data (e.g., if we feed coolant byte = 100 (which means 60°C after -40 offset), the output indeed shows "60°C"). We also verify the J1708 message parsing on a test string (for example, craft a message with a known PID and see that we can extract it).

Phase 2: Hardware Integration & In-Vehicle Testing (Initial)

Goal: Build the hardware setup and connect the system to the actual truck to start reading live data.

- **Assemble hardware:** Set up the ESP32 with the CAN transceiver and RS485 transceiver on a breadboard or protoboard. Double-check wiring against the truck's diagnostic port pinout before connecting. Include the power supply (if testing in the truck, perhaps initially power the ESP32 via USB from a laptop to avoid any power issues, using the truck's diag port only for signals and ground). Alternatively, use a benchtop 12V supply through a regulator to power the setup for testing outside the truck.

- **Safety check:** Before plugging into the truck's port, measure with a multimeter the voltage between Pin B (battery+) and Pin A (ground) to confirm correct pin orientation. Ensure no accidental shorts. Also measure resistance between Pin C and D with truck off – should read ~60 Ω (two 120 Ω terminators in parallel) confirming the CAN bus presence [28] [29]. J1708 pins F-G normally measure high impedance relative to anything (since it's not biased when no device transmitting, or some small bias resistors maybe).

- **Run the firmware in the truck:** With the microcontroller connected to a laptop (for power and serial debugging), turn the truck key to "run" (engine off initially). The microcontroller should initialize the CAN at 250k and begin listening, and open the UART at 9600 for J1708.

- Observe the serial output. Ideally, we should see real values updating for many parameters. It's likely we'll immediately catch engine RPM (if key on engine off, RPM=0 but as soon as engine runs it should show), coolant temp, etc.
- If nothing is coming on CAN, check connections and possibly the CAN driver initialization (maybe the bus uses 29-bit IDs – ensure our filter isn't excluding them; best to configure to accept extended frames).

- Once confirmed, start the engine to get live values. Note if the data seems plausible (e.g., engine speed rising, voltages around 13-14V for system voltage if available, etc.).

- **J1708 data:** With engine running (or at least key on), see if any J1708 messages are captured. Possibly no continuous data from ABS unless moving, or unless a fault is active. An idea: During key-on engine-off, many ABS units do a self-check and if there's a fault (like a disconnected sensor) they might send a code. If no faults, the ABS might be quiet. You could provoke a fault (e.g., unplug a

wheel speed sensor) to test, but that's not always easy. Alternatively, drive the truck a short distance with the laptop to see if any J1708 messages stream (like speed info).

- If capturing data is hard in real-time, consider using the **Nexiq** to log some J1708 traffic as the user suggested, then feed that log to our system offline. But let's assume we can at least detect something. At minimum, ensure our RS485 circuit is receiving: maybe temporarily enable a test mode where we transmit a J1708 message and see if we receive it (loopback via the bus). But be cautious not to transmit on the actual truck bus except in a controlled way.

- **Verify multi-ECU presence:** Check if we see separate data for engine and trans. The Allison 1000 TCM on J1939 should broadcast transmission gear and possibly torque converter slip, etc. If we don't see expected trans data, it might be on J1587 in older models. However, per Allison documentation, the 1000/2000 series use J1939 exclusively [30]. So likely, trans data is indeed on J1939. We'll confirm by looking for PGNs that come from the TCM's source address (each ECU has a source address, e.g., engine usually 0 or 1, transmission maybe 3). If using a network scanner example, it may list all active source addresses and PGNs. This is very useful to identify what data the truck broadcasts.

- **Tweak and Fix:** This phase might uncover issues. E.g., we might realize some PGNs are not parsed due to our library filter settings – we may need to adjust acceptance filters to truly get all extended frames. Or the J1708 parsing might get confused if our DE line handling is wrong (maybe need to tie DE low to listen properly). We will fix any such issues. Also, measure CPU utilization (printing too much too fast can slow loop – we might need to throttle debug prints).

- **Success criteria Phase 2:** The system **successfully reads data from the truck** for at least the basic parameters. We can see engine RPM, coolant temp, etc., changing as expected. We have confidence the hardware is reliable in the vehicle environment (no resets or errors during operation). At this stage, even if not all parameters are decoded, we have a live data stream.

Phase 3: Data Logging and Full Data Decoding

Goal: Using our hardware connected to the truck, **collect real data and refine the decoding for all relevant parameters**. Essentially, this is an R&D phase to ensure we're not missing anything and to expand our parameter coverage.

- **Logging Raw Data:** Modify the firmware to act as a **data logger**. For example, record a few minutes of driving data. Because we have two interfaces (CAN and serial), synchronize or timestamping might be beneficial. An approach: maintain a buffer in memory (or stream to SD if available) where each record is like: `[timestamp][bus][message]`. For CAN, log the 29-bit ID and data bytes; for J1708, log the MID and data bytes. If using serial output capture instead, we can simply print messages as they arrive with a timestamp (ESP32 can use `millis()` or an RTC). Example log line:

```
[CAN] 18FEF100 8 bytes: XX XX XX XX XX XX XX XX
[J1708] MID 140: XX XX XX ... CS
```

This raw log can later be analyzed on PC or even by the micro itself if we write a small parser to decode offline.

- **Identify Unknown Data:** Review the logged CAN IDs (PGNs) and see if any important ones are not yet decoded. If using a J1939 database (like lists from the standard or public resources [23]), match the PGNs. For example, we might notice PGN 65265 (0xFEF9) and realize that's Wheel Speed info from ABS (if this truck's ABS outputs via J1939 – some do if it's a newer ABS with traction control). Or PGN 65269 (0xFEFD) might appear (that's Cruise Control/Vehicle Speed). By recognizing them, we can add those to our decoder.

- If the ABS is indeed only on J1708, perhaps the ABS ECU doesn't put wheel speeds on J1939, but sometimes the engine or transmission might put out vehicle speed (from their own sensors or via ABS gateway).
- Check for **PGN 65267** – often Ambient conditions (outside temperature, barometric pressure, etc., if the engine ECU has that input). If present, we may not need our outside sensor for display (though ours could double-check the cab vs engine intake ambient).

- Check for any fuel economy related PGNs (like Fuel Consumption PGN 65266 or 65257 for average economy). If the engine supports it, we decode it.

- **Expand decoding logic:** Incorporate new SPNs and PGNs discovered. Verify conversions: For each new parameter, find the scaling from the J1939 standard or other documentation. For instance:

- SPN 91 (Accelerator Pedal Position): 0.4%/bit offset 0.
- SPN 92 (Percent Load At Current Speed).
- SPN 102 (Boost Pressure) – actually Manifold Pressure, which combined with barometric can give boost. Might have to subtract barometric to get actual boost.

- SPN 86 (Transmission Oil Temp) – etc. Use available references like the J1939-71 standard summary or online lists. **Cite example:** According to an introduction, Engine Coolant Temp is SPN 110, 1°C/bit with -40 offset in PGN 65262 [23]. We will systematically apply the correct formula for each.

- **J1708 detailed decode:** Analyze any J1708 messages from the log. If we have active ABS faults, identify the MID (likely ABS ECU's MID) and the fault format. Many J1587 fault codes use a PID 194 or similar, which encodes the SID (subsystem) and FMI. For example, we might find something like MID 140, PID 98 with some byte values that correspond to a wheel sensor fault. Using an **SAE J1587 reference** (which lists PIDs), decode if possible. Since obtaining the full J1587 spec might be tricky, we can also find hints from forums or documents. Copperhill might have an example list of J1587 parameters as well. If needed, simply report the MID and PID numbers on the display and documentation, so at least the user can research those codes manually if not auto-translated.

- If the ABS is newer (with ATC), it might actually be on J1939 (the document snippet suggests ABS with ATC often moved to J1939 [31]). If that is the case, we'd see PGN 65269 with wheel speeds from ABS, and ABS might send DM1 on J1939 for ABS faults. We should check if any DM1 (PGN 65226) appears with a source address that is not engine (maybe one for trans, one for ABS). DM1 contains the source address of the ECU with the fault as part of the data, or if multiple, the DM1 can contain multiple DTCs from different ECUs (depending if centralized or per ECU DM). Actually, J1939 DM1 is broadcast by each ECU separately. So Engine will broadcast its own DM1, Trans its own, ABS its own, etc. If the ABS is J1939-capable, it will broadcast DM1 on CAN. If not, then only the engine and trans will broadcast DM1 on CAN, and ABS faults remain on J1708. We will handle both possibilities.

- **Testing remote start viability:** Though not implementing, note if engine has a **Remote Start** mode or PTO (Power Take-Off) messages. Some engines allow start/stop via J1939 if in a certain mode (but often require special commands and safety conditions). It's more pragmatic that we'll use a relay. But as part of data collection, check if any data can confirm engine running status besides RPM (RPM > 0 obviously means running, or engine oil pressure, etc., but RPM is simplest).

- **Memory usage and performance:** By now, with many parameters, ensure the ESP32 has enough memory. Avoid using very large arrays; use `float` for scaled values or even integer with scale factors to save space if needed. The ESP32 has plenty of RAM for this, and our data volume is not huge.

- **Success criteria Phase 3:** We have a **robust decoding setup that covers nearly all the parameters listed in project goals**, validated against real truck data. The system should run for extended periods logging data without crashes. We should also have resolved any discrepancies (for example, if a value doesn't match the truck's gauge, double-check conversion). At this point, the microcontroller essentially functions as a "scan tool" outputting live data via serial, which was the intermediate goal.

Phase 4: Basic On-Screen Display

Goal: Integrate a simple screen and display key data in real-time, with minimal graphics.

- **Choose a display:** A convenient choice is a small **OLED display (128x64)** using I2C (e.g. SSD1306 0.96" OLED) or a larger character LCD. Given readability, a 20x4 character LCD might show more data at once (4 lines of text). But an OLED can show small graphs or icons if needed. We will assume using a **20x4 I2C LCD** for now, as it's simple to update text. Another option is a **TFT like 2.8" SPI display** in text mode; however, controlling that at high refresh with Arduino might be slower, so for Phase 4, simpler is fine.

- **Implement a display update loop:** For example, update the display 2 times a second (500 ms interval). On each update, refresh the displayed values. Avoid updating more often than needed to keep it flicker-free. The output could be organized into multiple pages or a single page if everything fits. Since we have many parameters, likely multiple pages are needed. For instance:

- *Page 1: Engine* – RPM, Boost, Load%, Coolant Temp, Oil Press, Oil Temp.
- *Page 2: Transmission* – Gear, Torque Converter Lock (yes/no or slip%), Trans Temp, Vehicle Speed, maybe Axle oil temp if available.
- *Page 3: Other* – Ambient temp, Intake Temp, EGT, Fuel levels (Main tank %, Aux tank %), Battery Voltage.
- *Page 4: Diagnostics* – If any active code, show "Engine Code SPN XXX FMI YYY", or "ABS Code: (description or number)". If none, say "No Active Faults".

We can cycle pages automatically every few seconds or use a button to toggle pages (if a spare GPIO configured for a button input). For now, maybe automatic cycling or just fix one page with the most critical info, to keep it simple.

- **Formatting:** Ensure that data formatting is clear. Use units (°C, RPM, etc.) where possible. Since space is limited on a small display, use abbreviations (e.g., "Eng RPM: 1500", "Coolant: 85C", "Boost: 20psi"). We can mix metric/imperial as needed (perhaps coolant in °C but could also convert to °F if desired – maybe allow a config in future).

- **Testing on bench:** Before installing in truck, test the display with simulated data (from Phase 1's mock environment or by replaying a log) to ensure the update logic works and doesn't crash. The I2C update should be fine, just check that updating doesn't block other tasks excessively (I2C is relatively slow, but updating 20x4 characters is quick enough not to hamper 250k CAN reading, especially if done at 2Hz).

- **In-truck test:** Mount the display where visible (temporarily) and run the system while the engine is on. Verify values on the screen match the truck's gauges for sanity (RPM, coolant, etc.). Adjust any formatting issues (e.g., if RPM goes to 10000, make sure field width is enough, etc., though for a diesel truck RPM won't exceed ~2500 or so).

- **User feedback:** At this stage, the basic display is working, and you can drive with it to see if the refresh rate and content are useful. Perhaps we find that certain values fluctuate too rapidly to read (in which case we might average them or update them slower). We might also decide to highlight abnormal values (e.g., flash if coolant over temp). Those enhancements could be added if needed.

Phase 5: Advanced UI and Integration (Future Work)

Goal: Plan for a more sophisticated user interface and integration with external systems (like Home Assistant). This phase is more open-ended and might be done after initial deployment, but we outline it for completeness:

- **High-resolution Graphics UI:** Switch to a larger color TFT display (e.g. a 5" or 7" if you plan to really create a custom dash, or use something like Nextion display). Implement using a GUI library like **LVGL (LittlevGL)** which can create dashboards with gauges, graphs, etc. This would require more processing but the ESP32 is capable especially if not overloaded. Alternatively, use a second microcontroller or an SBC (like Raspberry Pi) for the UI that receives data from the ESP32 (via serial or network). But keeping it all on one device is doable for moderate graphics.

- We could create virtual analog gauges for RPM, speed, etc., or simple bar graphs for temps. Or just a nicer arrangement of text and icons.

- **User Interaction:** Possibly add buttons or touchscreen for navigating pages, resetting trip data, viewing stored fault codes history, etc.

- **Data Logging to Cloud:** Implement Wi-Fi communication to send data to a server or Home Assistant. The ESP32 could connect to a hotspot or local Wi-Fi when in range. For Home Assistant:

- Using **MQTT**: e.g., publish `truck/engine/rpm = 1500` etc. Home Assistant can subscribe to these topics and make a dashboard or automate actions (like if temp < -10°C and time 3am, auto-start engine via an MQTT command back to the device).
- Alternatively, use **REST API** or **Home Assistant API** if the device is on the same network.
- Even without Home Assistant, one could send data to a cloud database or Google Sheets for analysis of trips, etc., but that's beyond initial scope.

- We will design the code such that adding an MQTT client is straightforward (ensuring it doesn't disrupt the timing of critical tasks – perhaps run network comms on one core and CAN on the other core of ESP32 to avoid conflicts).

- **Remote Start/Stop Control:** Add the logic for automatic start. For example, if the device detects ambient temp below a threshold and the truck has been off for a certain time, it could activate a relay to turn the ignition and start the engine, then run until coolant reaches say 60°C or for X minutes, then shut it off. This logic can either be autonomous in the device or commanded by Home Assistant. If autonomous, safety checks are needed (ensure truck is in neutral, parking brake on, etc. – some of these conditions might be read via J1939 too, e.g., park brake status PGN from ABS or body controller). For now, we just keep the integration in mind and possibly output a "engine is running" status that Home Assistant can use to decide if it needs to issue a start command or not.

- Actually reading tachometer (RPM) is the simplest way to know if the engine successfully started. We already have RPM via J1939, so that's covered.

- **Testing and Hardening:** With advanced features, test for stability. When Wi-Fi is on, ESP32's ADC2 channels can't be used (we assigned analogs to ADC1 earlier to avoid this). Also, ensure the realtime performance (the CAN reading) remains solid when Wi-Fi tasks run. We may need to raise task priorities or use interrupts for CAN reading to avoid losing frames if the CPU is busy with networking. The ESP32 is fairly capable though, and typically CAN interrupts will buffer a number of frames.

At the end of Phase 5, we would have a polished product: a custom digital dash showing extensive data, logging to cloud, and possibly controlling aspects of the truck.

Conclusion

In summary, this project will utilize an **ESP32 microcontroller** with appropriate bus transceivers to tap into the truck's J1939 CAN network and J1708 serial network, **decode a comprehensive set of engine, transmission, and ABS parameters**, and display them in a customized interface. We have specified the hardware components (ESP32, CAN transceiver, RS485 transceiver, sensors like thermocouples and temperature sensors) and outlined how to wire them to the vehicle and microcontroller [11]. On the software side, we will leverage existing **SAE J1939 protocol libraries** [32] [20] and a **J1708/J1587 parsing library** [27] to handle the heavy lifting of decoding these protocols, allowing us to focus on formatting the output and integrating additional sensors. The project is structured in incremental phases to ensure reliability at each step – from bench simulation to live data logging, and ultimately to user-friendly display output and network connectivity.

By following this spec, we will create a **flexible platform** for truck data monitoring that not only addresses the immediate desire for more gauge information, but also lays the groundwork for future upgrades like remote diagnostics, data-driven maintenance alerts, and home automation integration (e.g. automatic cold start). The result will be a powerful tool giving the truck owner deep insight into the vehicle's performance and health, far beyond what the stock dashboard offers.

References Used:

- Simma Software. *J1587 vs J1939: Understanding the Key Differences.* (Overview of J1708/J1587 vs J1939 characteristics) 3 26

- Copperhill Tech Blog. *Monitoring SAE J1708/J1587 Data Traffic Using Arduino.* (Details on J1708 physical layer using RS485 and Arduino connections) 6 7

- Digi International. *9-pin Deutsch J1939 Connector Pinout.* (Pin definitions for J1939/J1708 diagnostic connector) 11

- Copperhill Tech Blog. *J1939 Protocol Stack for ESP32 (ARD1939).* (Use of ESP32 for J1939 with external transceiver, reference to ARD1939 stack) 32 4

- vChavezB on GitHub. *SimpleJ1939 Library README.* (Open source Arduino J1939 parsing library, based on Copperhill examples) 20

- Jeremy Daily on GitHub. *J1708 Library README.* (Arduino library for J1708 serial communication) 27

- CSS Electronics. *J1939 Explained - Simple Intro.* (J1939 basics: 29-bit ID, 250k baud, PGN/SPN concept, etc.) 12 13

- SAE J1939 Standard references via FCar and others. (Mapping of SPNs like coolant temp in PGN 65262, etc.) 23

- Scribd – Freightliner Bulletin. *J1939 Datalink Diagnostics.* (Confirmation that Allison 1000/2000 uses J1939, and diagnostic connector info) 30 33

- Adafruit Industries. *CAN Pal (TJA1051) Transceiver Product Info.* (Details of using a 3.3V CAN transceiver with ESP32 and termination considerations) 2

- SensorConnection.com. *EGT Probe Description.* (Capabilities of a typical EGT K-type probe, up to 1300°C and 1/8" NPT fitting) 9

---

1 2 5 Adafruit CAN Pal - CAN Bus Transceiver [TJA1051T/3] : Adafruit Industries, Unique & fun DIY electronics and kits
https://www.adafruit.com/product/5708?srsltid=AfmBOooen7fA4bFMe8uqGFG-otrQ9hfZQPiNIk_UIFQOw1ud63gQaa2R

3 26 J1587 vs J1939: Understanding the Key Differences
https://simmasoftware.com/j1587-vs-j1939/

4 15 16 17 18 21 25 32 J1939 Protocol Stack Sketch for ESP32 Using the Arduino IDE - Copperhill
https://copperhilltech.com/blog/sae-j1939-protocol-stack-sketch-for-esp32-using-the-arduino-ide/

6 7 8 Monitoring SAE J1708/J1587 Data Traffic Using The Arduino Mega2560 Or Arduino Due - Copperhill
https://copperhilltech.com/blog/monitoring-sae-j1708j1587-data-traffic-using-the-arduino-mega2560-or-arduino-due/

9 EGT Probe Exhaust Gas Temperature Sensor | 1/8" Diameter
https://thesensorconnection.com/product/egt-probe-exhaust-gas-temperature-sensor-18-diameter?srsltid=AfmBOorIxXGZMlY7zLgr_gpMLZCuQwWncxgDZk5m3D_sPPoV2wajb2r9

10 1/8" NPT EGT K Type Thermocouple Exhaust Probe 1200°C ...
https://www.amazon.com/Thermocouple-Exhaust-Temperature-Sensors-Stainless/dp/B0CZRS82D2

11 Connector pinout
https://docs.digi.com/resources/documentation/digidocs/90001930/reference/r_hardware_connector_pinout.htm

12 13 14 22 24 J1939 Explained - A Simple Intro [2025] – CSS Electronics
https://www.csselectronics.com/pages/j1939-explained-simple-intro-tutorial

19 20 GitHub - vChavezB/SimpleJ1939: A simple j1939 communication library for Arduino
https://github.com/vChavezB/SimpleJ1939

23 A Brief Introduction to the SAE J1939 Protocol - FCAR Tech USA
https://www.fcarusa.com/TechSupport/FAQ/brief-introduction-sae-j1939-protocol

27 GitHub - jeremydaily/J1708: An Arduino library for SAE J1708 serial communications for heavy vehicles.
https://github.com/jeremydaily/J1708

28 29 30 31 33 J1939 Datalink Diagnostics | PDF | Electrical Connector | Resistor
https://www.scribd.com/document/478961331/J1939-Datalink-Diagnostics