ECE 699 – TERM REPORT

PROJECT: POWERFUZZER - AGL

By Prerona Ghosh

Dept: Electrical and Computer Engineering

Student ID: 21049973

<u>Automotive Grade Linux (AGL)</u>

AGL provides us with a set of APIs that enable developers to interact with various vehicle systems and functionalities. These APIs offer standardized access to vehicle data and services, making it easier for developers to develop applications without the need of working on adding support for underlying vehicle hardware.

Below is an overview of some of the key APIs and services offered by AGL:

1. Application Framework (AFM) APIs

The Application Framework (AFM) in AGL manages the lifecycle of applications, along with installation, updating, and removal utilities. It provides APIs for:

- *Application Lifecycle Management*: Start, stop, and manage applications.
- *Window Management*: Control window positioning and layering for applications.
- *Security and Permissions*: Manage permissions for accessing specific vehicle features.

2. Vehicle Signal Specification (VSS) APIs

AGL uses VSS for standardized vehicle signals and data access. The following application actions are offered as part of the API:

- *Read and Subscribe to Vehicle Data*: Access real-time data from various vehicle sensors (e.g., speed, fuel level, engine parameters).
- *Write Vehicle Data*: Send commands or settings to the vehicle systems (where permissible), like HVAC settings or seat adjustments.

3. Binding APIs

AGL provides bindings that act as high-level APIs to access core functionalities:

- *Bluetooth***:** Manage Bluetooth connections and functionalities such as pairing devices, streaming audio, and handling phone calls.
- *Audio Management*: Control audio routing, volume levels, and sound characteristics.
- *Network Management*: Manage connectivity settings and status, including WiFi and mobile data.
- *Location and Navigation*: Access GPS data and manage navigation sessions.

4. User Interface (UI) APIs

For developers focusing on custom user interfaces, AGL offers:

- *Widgets and Controls*: APIs for using standard widgets and controls in your applications.
- *Touch Input*: Handle touch inputs and gestures.
- *Display APIs*: Manage display settings and properties specific to different screens in the vehicle.

5. Multimedia APIs

These are specialized APIs for handling media playback and recording:

- *Media Playback and Control*: APIs for playing audio and video files, streaming media, and controlling playback like play, pause, skip, and rewind.
- *Camera and Video Management*: Access and manage video feeds from vehicle cameras, potentially useful for features like dashcams or parking assistance.

6. Telephony and Messaging APIs

For applications that involve communication, these APIs provide:

- *Call Handling***:** Manage incoming and outgoing calls.
- *SMS and Messaging*: Send and receive messages through the vehicle's telematics system.

7. Sensor APIs

These APIs allow access to various vehicle sensors for applications that monitor environmental or vehicle conditions:

- *Environmental Sensors***:** Access data from temperature, humidity, or air quality sensors.

- *Motion Sensors***:** Utilize accelerometer or gyroscope data for apps that may need to respond to vehicle movements.

These APIs allow for a wide range of applications, from simple media players to complex vehicle monitoring and control systems, leveraging the full capabilities of the AGL platform.

AGL VSS Model for Dashboard Development:

VSS defines a structured and hierarchical data model representing vehicle signals and attributes. For instance, vehicle speed signal is under a 'Chassis' branch, while battery information is under 'Powertrain'. By standardizing vehicle signals, VSS allows developers to create applications that are agnostic to the vehicle's make or model, ensuring compatibility across different vehicle platforms that also use VSS. It is designed to be extensible so that new signals and data attributes can be added as vehicles become more advanced and offer new features and sensors.

VSS APIs within AGL:

AGL uses VSS APIs to allow applications to interface with the vehicle's data and control systems. Here's how they enable interaction:

Read and Subscribe to Vehicle Data:

- *Real-Time Access:* Applications can read data from the vehicle's sensors in real-time. For example, an application can display the current speed, fuel level, or engine temperature.
- *Subscription Mechanism:* Beyond one-time reads, applications can subscribe to certain signals, meaning they can listen for changes in data over time. If an application subscribes to the vehicle speed signal, it will receive updates whenever the speed changes.
- *Data Abstraction:* The VSS APIs provide an abstraction layer over the actual hardware. This means the application doesn't need to know where and how the data is stored or acquired; it just needs to know what signal it requires.

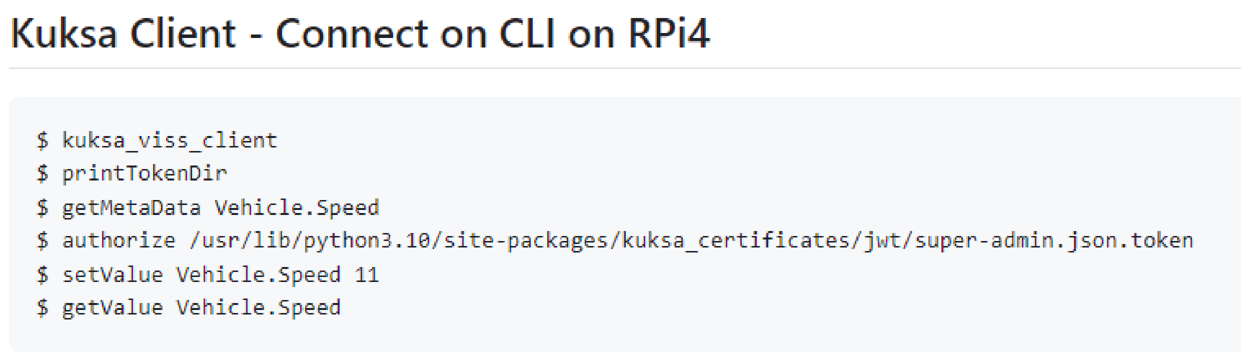
Write Vehicle Data:

- *Control Commands:* Some vehicle signals are not just for reading; they can also be written to. For example, changing the temperature setting of the HVAC (Heating, Ventilation, and Air Conditioning) system.
- *Permission-Based:* Writes are typically permission-controlled for safety and security. Not all signals can be written to, and not all applications will have the necessary permissions to write to those that can be changed.

- *Actuator Interaction:* The ability to write to vehicle data allows applications to interact with actuators—components like electric motors that move or control mechanisms or systems in the vehicle.

Configuring Kuksa Client to manipulate parameters on dashboard:

Apart from an automated script, we could manually connect to the Kuksa server to manipulate vehicle signals displayed on the dashboard using the following commands.



Kuksa Client - Connect on CLI on RPi4

```
$ kuksa_viss_client
$ printTokenDir
$ getMetaData Vehicle.Speed
$ authorize /usr/lib/python3.10/site-packages/kuksa_certificates/jwt/super-admin.json.token
$ setValue Vehicle.Speed 11
$ getValue Vehicle.Speed
```

'setValue' and 'getValue' commands return us JSON data which either confirm that value has been updated successfully, or that there has been an error.

<u>AGL Lab Setup:</u>

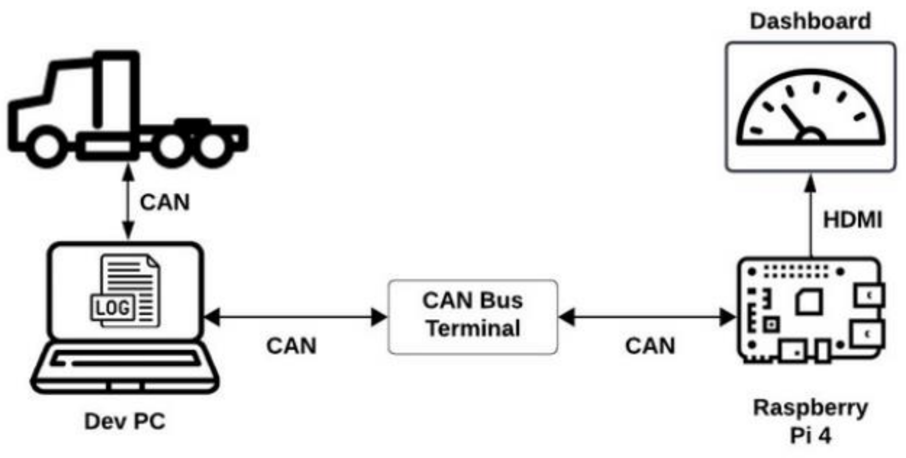


Figure 1: Lab test setup

The above is a representation of a vehicle data monitoring with a display system using a CAN bus network useful for the purposes of testing in the absence of real-time CAN data and hardware. It can be used to replay CAN data log collected from the truck onto the Raspeberry Pi 4 that is running AGL and have that update the Dashboard GUI connected on the other end for testing. This can be done using can (virtual CAN) module on a Linux distribution.

Hardware Setup:

First step is to plug a Kvaser adapter to the computer and the other end to the CAN bus terminal and check if the computer detected the adapter correctly using the command '`dmesg | tail`'. Then we can set up a virtual CAN network on the computer by loading up the `vcan` module, which is the CAN network driver. Configure the CAN interface as follows and activate it. The commands are as follows:

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

Plug another Kvaser adapter to the Raspberry Pi 4 and the CAN bus terminal. Check if `vcan0` is available once plugged in using `ifconfig`. If not found automatically, we need to run the above commands on the RPi4 to setup the network manually.

Replaying CAN data using a Virtual network:

We then must `ssh` into the Raspberry Pi 4, start two terminals, one to send CAN data and the other to monitor the reception of data on the CAN interface. To replay CAN data on the virtual CAN interface, we need to use the following commands:
On terminal 1, use the command `candump can0` to monitor received data.
On terminal 2, use command `canplayer -I <name_of_candump>.log` to send data.

The candump log will contain a different CAN interface configuration, either can0 or can1 since the truck would have a different configuration. Hence, we need to modify the command on terminal 2 with additional configuration: `canplayer -I <name_of_candump>.log vcan=can0`

Sending Physical CAN data:

In order to send physical CAN data, we can load `can` module supported on most Linux distributions instead of `vcan` and run the following commands with the same hardware setup:

```
sudo modprobe can
sudo ip link set can0 type can bitrate 250000
sudo ip link set up can0
```

This loads up can network driver, configures a CAN interface with a bitrate of 250,000 bits per second and activates it using the last command. We can then use the development PC to run a `cansend` command on `can0` instead of `canplayer` in the following format:

```
cansend <device> <can_frame>
```

For example, the command `cansend can0 12345678#1234567812345678` sends a CAN frame on the `can0` interface with an extended CAN ID of 0x12345678 and a data payload of 8 bytes with the data 0x1234567812345678.

AGL in the Truck:

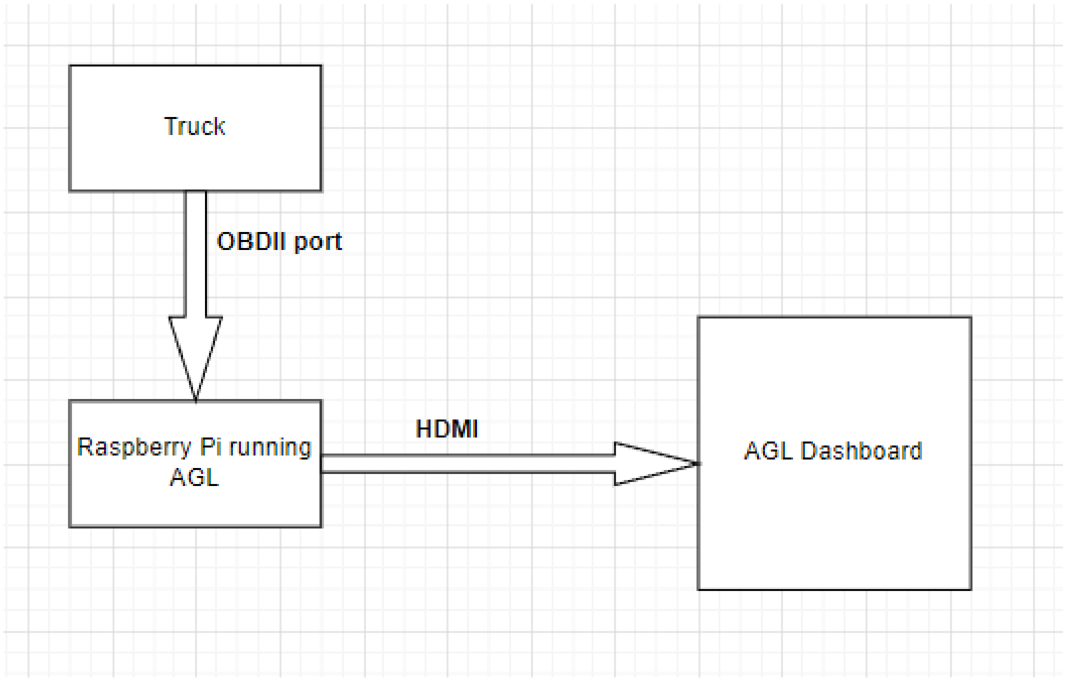


Figure 2: Representation of AGL setup in the truck

The setup in the truck is similar with some differences to capture real time data from the truck.

The truck is equipped with an OBD-II port, which is a standardized gateway in vehicles that allows external electronics to interface with the vehicle's computer system to access vehicle status, diagnostic codes, and real-time data. This port is located under the dashboard of the truck and gives access to a range of data like engine parameters, vehicle speed, fuel consumption, and others. We need to connect the Raspberry Pi 4 flashed with AGL to the OBDII port instead of the lab development computer in the truck to capture real time CAN data. The OBDII port should be connected to a Kvaser USB-CAN adapter which would then be plugged into the RPi4. The RPi4 should be connected to a monitor on the other end using an HDMI cable, same as the lab setup.

To get real time data out of the OBDII port, the truck needs to be either on Battery mode or the engine needs to be running. On battery mode, only some data is accessible, such as fault codes or certain vehicle status information. However, it does not provide all the parameters that are only available when the engine is running.

The CAN interface needs to be setup just as above and a bitrate of 250,000 bits per seconds needs to be set. Basic monitoring of CAN data received by the AGL can be checked using `candump` command.

Connecting to the Dashboard GUI:

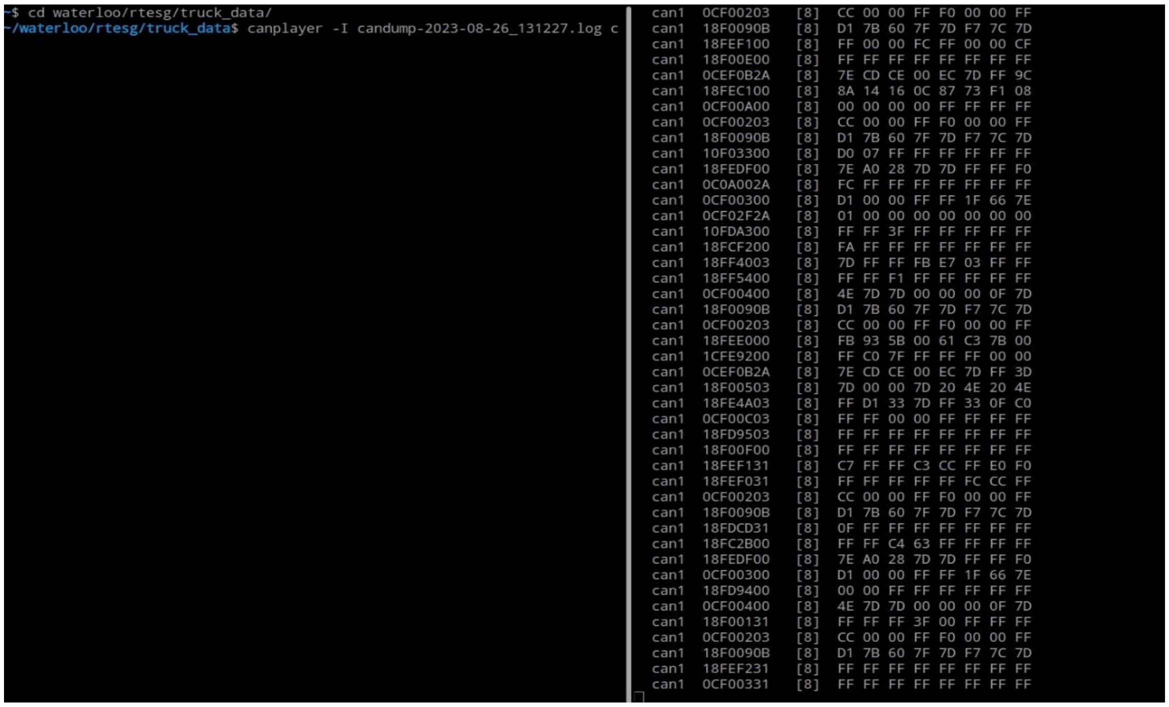
We use KUKSA Val (Vehicle Abstraction Layer) to communicate the received data on the CAN interface (virtual or physical) to the AGL cloud service.

KUKSA.val is a component of the Eclipse KUKSA project, which aims to provide a unified framework for vehicle-to-cloud (V2C) communication. The primary goal of KUKSA.val is to abstract vehicle data handling from the underlying complexity of vehicle interfaces, offering a standardized API to interact with different types of automotive data. This makes it easier to create applications that can interact with a variety of vehicle systems without needing detailed knowledge of the underlying hardware or communication protocols.

For this purpose, I have created a script that encapsulates the following steps:

- Initialize a raw CAN socket and bind it to the appropriate interface.
- Initialize the Kuksa VAL via the 'kuksa_viss_client' Python library, authorize it and then prepare to receive the CAN messages and signals.
- Listen for incoming CAN frames, decode them using an appropriate package, and create can message objects.
- Process CAN messages, check the arbitration_id and then use the processed id to check against specific PGNs (Parameter Group Numbers) to identify the type of message.
- If the message matches known PGNs, decode the CAN frame data, extracts values such as vehicle speed or traveled distance, and display them.
- If the PGN matches certain values, update the corresponding vehicle parameters via the Kuksa client using **setValue** calls.

Results:



```
~$ cd waterloo/rtesg/truck_data/
~/waterloo/rtesg/truck_data$ canplayer -I candump-2023-08-26_131227.log c
```

```
  can1  0CF00203   [8]  CC 00 00 FF F0 00 00 FF
  can1  18F0090B   [8]  D1 7B 60 7F 7D F7 7C 7D
  can1  18FEF100   [8]  FF 00 00 FC FF 00 00 CF
  can1  18F00E00   [8]  FF FF FF FF FF FF FF FF
  can1  0CEF0B2A   [8]  7E CD CE 00 EC 7D FF 9C
  can1  18FEC100   [8]  8A 14 16 0C 87 73 F1 08
  can1  0CF00A00   [8]  00 00 00 00 FF FF FF FF
  can1  0CF00203   [8]  CC 00 00 FF F0 00 00 FF
  can1  18F0090B   [8]  D1 7B 60 7F 7D F7 7C 7D
  can1  10F03300   [8]  D0 07 FF FF FF FF FF FF
  can1  18FEDF00   [8]  7E A0 28 7D 7D FF FF F0
  can1  0C0A002A   [8]  FC FF FF FF FF FF FF FF
  can1  0CF00300   [8]  D1 00 00 FF FF 1F 66 7E
  can1  0CF02F2A   [8]  01 00 00 00 00 00 00 00
  can1  10FDA300   [8]  FF FF 3F FF FF FF FF FF
  can1  18FCF200   [8]  FA FF FF FF FF FF FF FF
  can1  18FF4003   [8]  7D FF FF FB E7 03 FF FF
  can1  18FF5400   [8]  FF FF F1 FF FF FF FF FF
  can1  0CF00400   [8]  4E 7D 7D 00 00 00 0F 7D
  can1  18F0090B   [8]  D1 7B 60 7F 7D F7 7C 7D
  can1  0CF00203   [8]  CC 00 00 FF F0 00 00 FF
  can1  18FEE000   [8]  FB 93 5B 00 61 C3 7B 00
  can1  1CFE9200   [8]  FF C0 7F FF FF FF 00 00
  can1  0CEF0B2A   [8]  7E CD CE 00 EC 7D FF 3D
  can1  18F00503   [8]  7D 00 00 7D 20 4E 20 4E
  can1  18FE4A03   [8]  FF D1 33 7D FF 33 0F C0
  can1  0CF00C03   [8]  FF FF 00 00 FF FF FF FF
  can1  18FD9503   [8]  FF FF FF FF FF FF FF FF
  can1  18F00F00   [8]  FF FF FF FF FF FF FF FF
  can1  18FEF131   [8]  C7 FF FF C3 CC FF E0 F0
  can1  18FEF031   [8]  FF FF FF FF FF FC CC FF
  can1  0CF00203   [8]  CC 00 00 FF F0 00 00 FF
  can1  18F0090B   [8]  D1 7B 60 7F 7D F7 7C 7D
  can1  18FDCD31   [8]  0F FF FF FF FF FF FF FF
  can1  18FC2B00   [8]  FF FF C4 63 FF FF FF FF
  can1  18FEDF00   [8]  7E A0 28 7D 7D FF FF F0
  can1  0CF00300   [8]  D1 00 00 FF FF 1F 66 7E
  can1  18FD9400   [8]  00 00 FF FF FF FF FF FF
  can1  0CF00400   [8]  4E 7D 7D 00 00 00 0F 7D
  can1  18F00131   [8]  FF FF FF 3F 00 FF FF FF
  can1  0CF00203   [8]  CC 00 00 FF F0 00 00 FF
  can1  18F0090B   [8]  D1 7B 60 7F 7D F7 7C 7D
  can1  18FEF231   [8]  FF FF FF FF FF FF FF FF
  can1  0CF00331   [8]  FF FF FF FF FF FF FF FF
```

Figure 3: CAN replay results on AGL

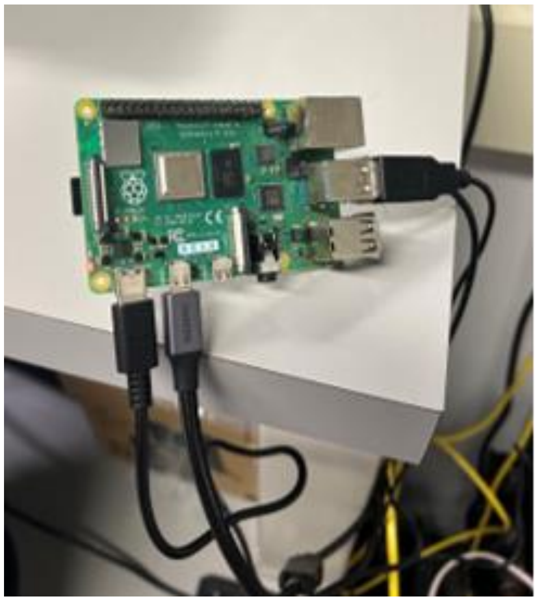


Figure 4: Raspberry Pi 4 running AGL

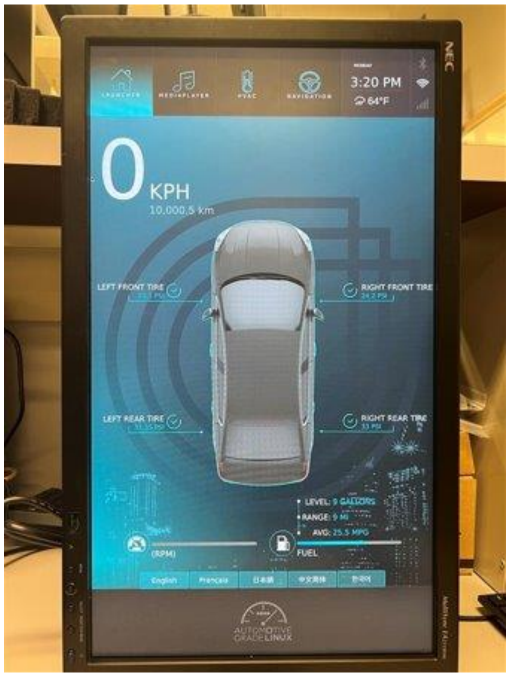


Figure 5: AGL Dashboard view

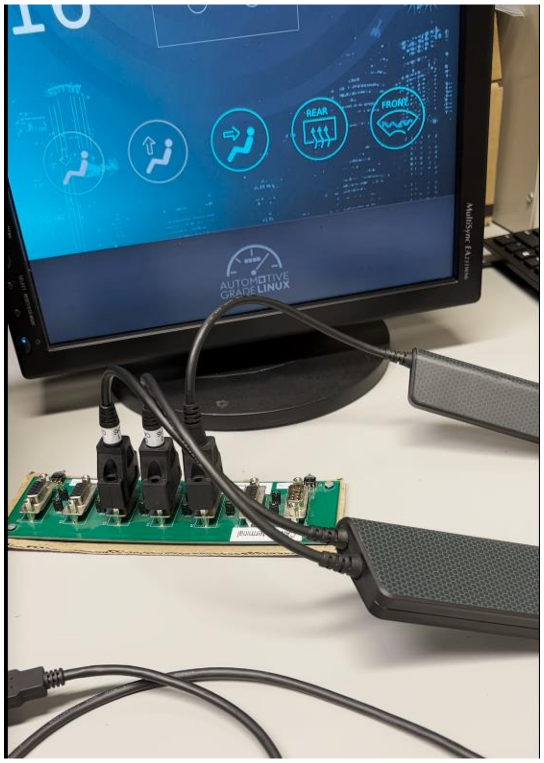
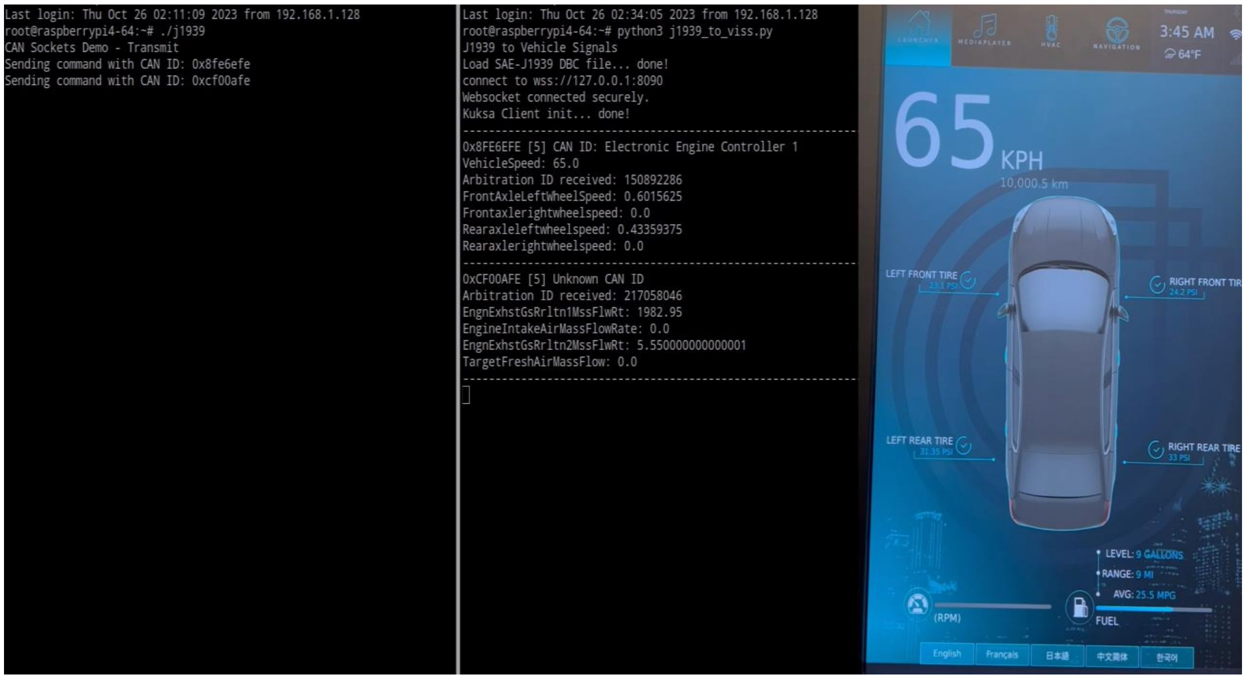


Figure 6: CAN Bus terminal and Kvaser adapters


```
Last login: Thu Oct 26 02:11:09 2023 from 192.168.1.128
root@raspberrypi4-64:~# ./j1939
CAN Sockets Demo - Transmit
Sending command with CAN ID: 0x8fe6efe
Sending command with CAN ID: 0xcf00afe
```

```
Last login: Thu Oct 26 02:34:05 2023 from 192.168.1.128
root@raspberrypi4-64:~# python3 j1939_to_viss.py
J1939 to Vehicle Signals
Load SAE-J1939 DBC file... done!
connect to wss://127.0.0.1:8090
Websocket connected securely.
Kuksa Client init... done!
-------------------------------------------------------------
0x8FE6EFE [5] CAN ID: Electronic Engine Controller 1
VehicleSpeed: 65.0
Arbitration ID received: 150892286
FrontAxleLeftWheelSpeed: 0.6015625
Frontaxlerightwheelspeed: 0.0
Rearaxleleftwheelspeed: 0.43359375
Rearaxlerightwheelspeed: 0.0
-------------------------------------------------------------
0xCF00AFE [5] Unknown CAN ID
Arbitration ID received: 217058046
EngnExhstGsRrltn1MssFlwRt: 1982.95
EngineIntakeAirMassFlowRate: 0.0
EngnExhstGsRrltn2MssFlwRt: 5.550000000000001
TargetFreshAirMassFlow: 0.0
-------------------------------------------------------------
```

Figure 7: CAN data transmission and reception, dashboard updated with vehicle data

J1939 Protocol and decoding Truck data:

The J1939 protocol is an industry-standard networking protocol used for communication within heavy-duty vehicles such as trucks, buses, and off-road vehicles. It is based on the CAN (Controller Area Network) bus, which provides a method for the vehicle's electronic units to communicate with each other without a host computer. J1939 is built on top of CAN by defining a structured way to use the protocol for vehicle communication.

- *CAN Bus Foundation*: J1939 utilizes the CAN 2.0B specification with the extended 29-bit identifier.
- *Speed*: Typically operates at 250 kbps for trucks and buses, which is sufficient for the real-time data exchange and control tasks required in vehicles.
- *Message Format*: J1939 messages are packaged in a standard CAN frame, which includes fields like the arbitration ID (the message identifier that also dictates priority), control, data, and CRC fields for error checking.

J1939 Addressing:

- *Parameter Group Numbers (PGNs):* These identify the group of parameters or the type of message being transmitted. PGNs determine the message's format and content.
- *Source Addresses (SAs):* Every control unit (ECU) on a J1939 network has a unique address from which it sends messages.

J1939 Message Structure:

- *Extended Identifiers:* J1939 uses the extended 29-bit identifiers in the CAN frame. This identifier includes the priority, PGN, and source address.
- *Data Field:* Up to 8 bytes long, as per the limits of the standard CAN protocol. J1939 also defines a transport protocol to allow longer messages to be sent in multiple frames.

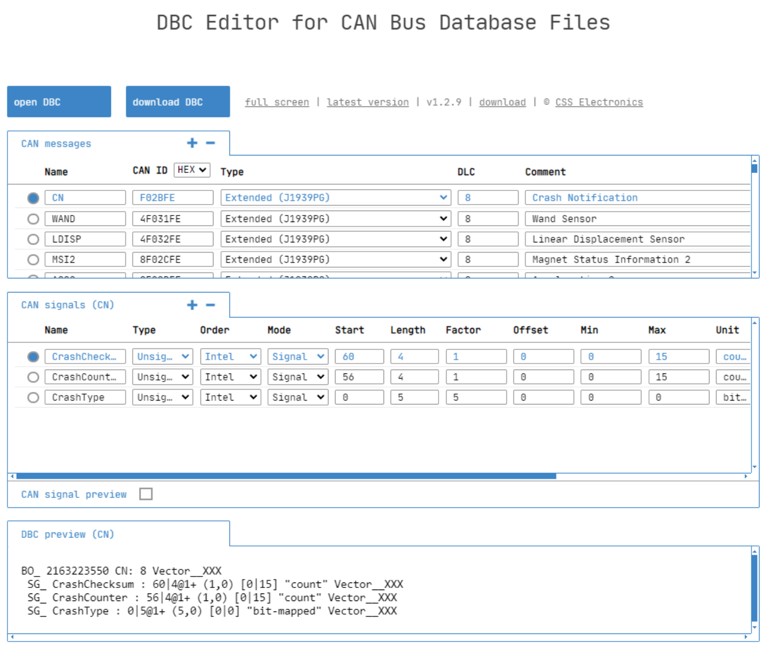
Working with J1939 messages for AGL:

- ECUs broadcast their data on the network in their messages, using specific PGNs that any other device on the network can receive and process.
- A device or tool designed to read data from the J1939 network would listen for these messages and, upon receiving them, would extract the data from the payload.
- Our script uses the DBC file that defines PGNs and format of data within each PGN (SPNs).

- Interpreting the messages involves decoding the data from the raw bytes into meaningful values according to the definition for each PGN and SPN. This process often uses a DBC or a CAN database file that maps the message structure.
- When an ECU sends data, it constructs a CAN frame with the appropriate PGN, sets the data bytes, and broadcasts it onto the bus.
- The bus arbitration process, which is inherent in CAN, manages the transmission priority based on the message identifier.
- J1939 also allows ECUs to send control commands, such as requests for data or commands to actuate systems.
- Control commands typically use specific PGNs for requests and follow a set structure that includes the requested PGN and, if necessary, the destination address.

Hardware Interface and Tools Required:

- We need a CAN interface that supports J1939. Many interfaces come with libraries that facilitate interaction with the CAN bus.
- We have utilized available Python libraries such as 'can-utils' for Linux which come with pre-built utilities to interact with CAN bus and can be used with J1939 if the PGNs and data structures are known.
- Extracting PGNs and signals were done using online DBC editor tool as shown below.


DBC Editor for CAN Bus Database Files
open DBC
download DBC
full screen | latest version | v1.2.9 | download | © CSS Electronics
CAN messages
CAN signals (CN)
CAN signal preview
DBC preview (CN)

Name	CAN ID (HEX)	Type	DLC	Comment
CN	F02BFE	Extended (J1939PG)	8	Crash Notification
WAND	4F031FE	Extended (J1939PG)	8	Wand Sensor
LDISP	4F032FE	Extended (J1939PG)	8	Linear Displacement Sensor
MSI2	8F02CFE	Extended (J1939PG)	8	Magnet Status Information 2

Name	Type	Order	Mode	Start	Length	Factor	Offset	Min	Max	Unit
CrashCheck…	Unsig…	Intel	Signal	60	4	1	0	0	15	cou…
CrashCount…	Unsig…	Intel	Signal	56	4	1	0	0	15	cou…
CrashType	Unsig…	Intel	Signal	0	5	5	0	0	0	bit…

```
BO_ 2163223550 CN: 8 Vector__XXX
 SG_ CrashChecksum : 60|4@1+ (1,0) [0|15] "count" Vector__XXX
 SG_ CrashCounter : 56|4@1+ (1,0) [0|15] "count" Vector__XXX
 SG_ CrashType : 0|5@1+ (5,0) [0|0] "bit-mapped" Vector__XXX
```

Figure 8: DBC Editor tool

- The signals within the data field had to be parsed according to their start bit, length, and scale as defined by the DBC file.

Example of J1939 message:

Using the cansend utility from can-utils, a J1939 message would be sent like this:

```
cansend can0 18FEF100#1122334455667788
```

Here, 18FEF100 is the 29-bit identifier where 18 is the priority, FEF1 is the PGN and 00 is the source address. The data 1122334455667788 is 8 bytes long and would represent the actual data being transmitted on the network.

Challenger Truck Specification:

The specification is a comprehensive technical overview of the electronic systems in the Challenger truck, specifically focusing on the various Electronic Control Units (ECUs), CAN bus systems, connectors, fuse boxes, and the vehicle's physical features and specifications. Below is a brief overview of ECUs, features, and communication networks and systems on the truck.

- Equipped with a Cummins ISX15 diesel engine
- Automatic transmission
- Advanced ABS for braking and stability control
- Includes ACC (Adaptive Cruise Control) with braking and collision mitigation
- Utilizes J1939 CAN bus system for communication across ECUs
- Comes with comfort and utility features such as air brakes, spring suspension, power inverters and sleeper cab

Vehicle Electronic Control Units (ECUs):

- *Engine Control Module (ECM)*: Manages engine performance and emissions, located on the engine's left side, powered by specific engine side fuses.
- *Transmission Control Module (TCM)*: Controls the transmission shifts, located underneath the cab, powered by specific engine side fuses.
- *ABS ECU*: Ensures proper functionality of the braking system and includes ESP/ESC systems. It also interfaces with the ACC and collision mitigation system, which uses a radar module at the front bumper.
- *Cab ECU (CECU):* Manages interior functions like climate control, wipers, turn signals, and headlamps.

- *Chassis Module*: Responsible for exterior lights and external sensors but does not directly communicate on the CAN bus via the OBD connector.

CAN Busses and Protocols:

- *V-CAN bus:* Uses the SAE J1939 protocol at a baud rate of 250 kb/s with around 58% bus utilization at idle.
- *J1587 bus:* An older protocol used for heavy-duty vehicle diagnostic systems, connected to the ABS ECU.
- *D-CAN bus:* Another implementation of the J1939 protocol, connected to the CECU.

Connectors and Ports:

- *9pin Deutsch male receptacle connector*: Commonly known as the OBD connector, allows for diagnostic tools to interface with the vehicle's CAN bus system.
- *9pin Deutsch to DB9 cable*: Converts the truck's proprietary connector to a standard DB9 connector.

Fuse Boxes:

- The specification details the power and ignition source for each ECU via specific fuses located on both engine side and dash side fuse boxes.

Additional Technical Data:

- A list of source addresses seen from the OBD II port during vehicle idle, which represent different ECUs and their functions.
- Specific proprietary PGNs (Parameter Group Numbers) sent from components like the radio and door controllers, which are unique identifiers for CAN bus messages.
- The document also contains miscellaneous information, such as vehicle keys possessors and photographic metadata from a OnePlus A5010 device used for documentation.

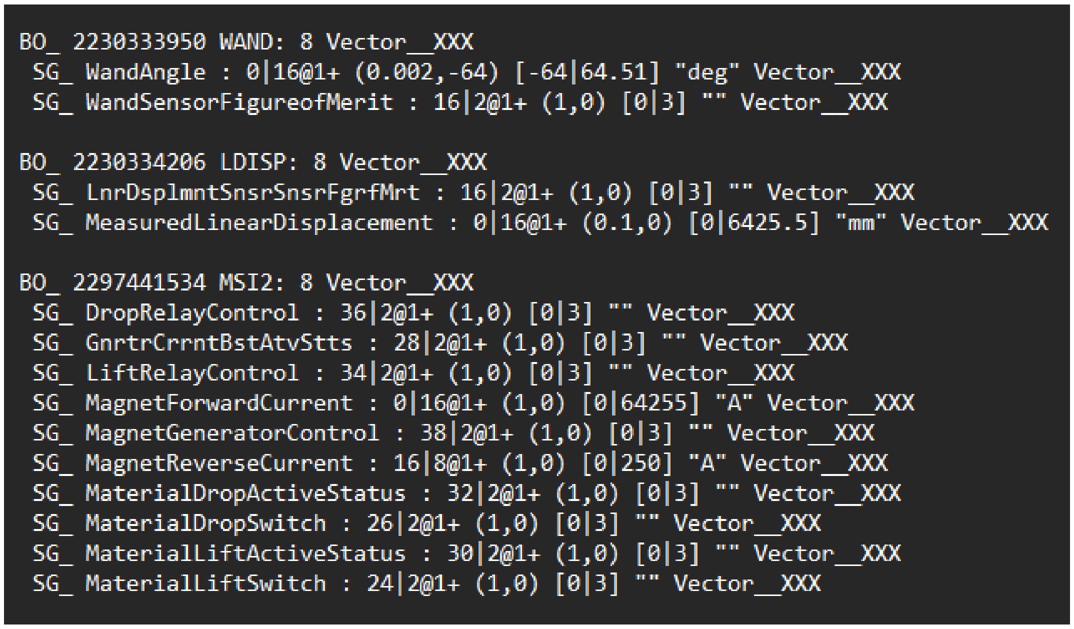
Analysis of DBC File used for the project:

The DBC file acts as a map to decode raw CAN messages on the truck network into meaningful data formats, for understanding vehicle dynamics, diagnosing issues, and developing features. It gets loaded into the software using a script and automatically matches the raw data with message and signal definitions from the DBC to ensure unsupported CAN messages or unknown commands are treated differently than known commands.

Overview:

- VERSION: No specific version info is provided.
- NS_ (New Symbols):

    - Includes various definitions like NS_DESC_, CM_ (Comment), BA_DEF_ (Attribute Definitions), and more. These are placeholders and definitions for attributes and metadata that is used throughout the DBC file.
- Definitions and Placeholders:
    - VAL_: Indicates the start of value tables, which are used to map signal values to human-readable strings.
    - CAT_DEF_, CAT_: Categorical definitions.
- FILTER: Related to filtering specific messages or signals.
- BA_DEF_DEF_: Definitions for attribute defaults.
- EV_DATA_, ENVVAR_DATA_: Related to environment variable data.
- SGTYPE_, SGTYPE_VAL_, BA_DEF_SGTYPE_, BA_SGTYPE_, SIG_TYPE_REF_: Definitions related to signal types.



```
BO_ 2230333950 WAND: 8 Vector__XXX
 SG_ WandAngle : 0|16@1+ (0.002,-64) [-64|64.51] "deg" Vector__XXX
 SG_ WandSensorFigureofMerit : 16|2@1+ (1,0) [0|3] "" Vector__XXX

BO_ 2230334206 LDISP: 8 Vector__XXX
 SG_ LnrDsplmntSnsrSnsrFgrfMrt : 16|2@1+ (1,0) [0|3] "" Vector__XXX
 SG_ MeasuredLinearDisplacement : 0|16@1+ (0.1,0) [0|6425.5] "mm" Vector__XXX

BO_ 2297441534 MSI2: 8 Vector__XXX
 SG_ DropRelayControl : 36|2@1+ (1,0) [0|3] "" Vector__XXX
 SG_ GnrtrCrrntBstAtvStts : 28|2@1+ (1,0) [0|3] "" Vector__XXX
 SG_ LiftRelayControl : 34|2@1+ (1,0) [0|3] "" Vector__XXX
 SG_ MagnetForwardCurrent : 0|16@1+ (1,0) [0|64255] "A" Vector__XXX
 SG_ MagnetGeneratorControl : 38|2@1+ (1,0) [0|3] "" Vector__XXX
 SG_ MagnetReverseCurrent : 16|8@1+ (1,0) [0|250] "A" Vector__XXX
 SG_ MaterialDropActiveStatus : 32|2@1+ (1,0) [0|3] "" Vector__XXX
 SG_ MaterialDropSwitch : 26|2@1+ (1,0) [0|3] "" Vector__XXX
 SG_ MaterialLiftActiveStatus : 30|2@1+ (1,0) [0|3] "" Vector__XXX
 SG_ MaterialLiftSwitch : 24|2@1+ (1,0) [0|3] "" Vector__XXX
```

Figure 9: DBC file internal signal and frame definitions

Challenger - CAN BUS Architecture:

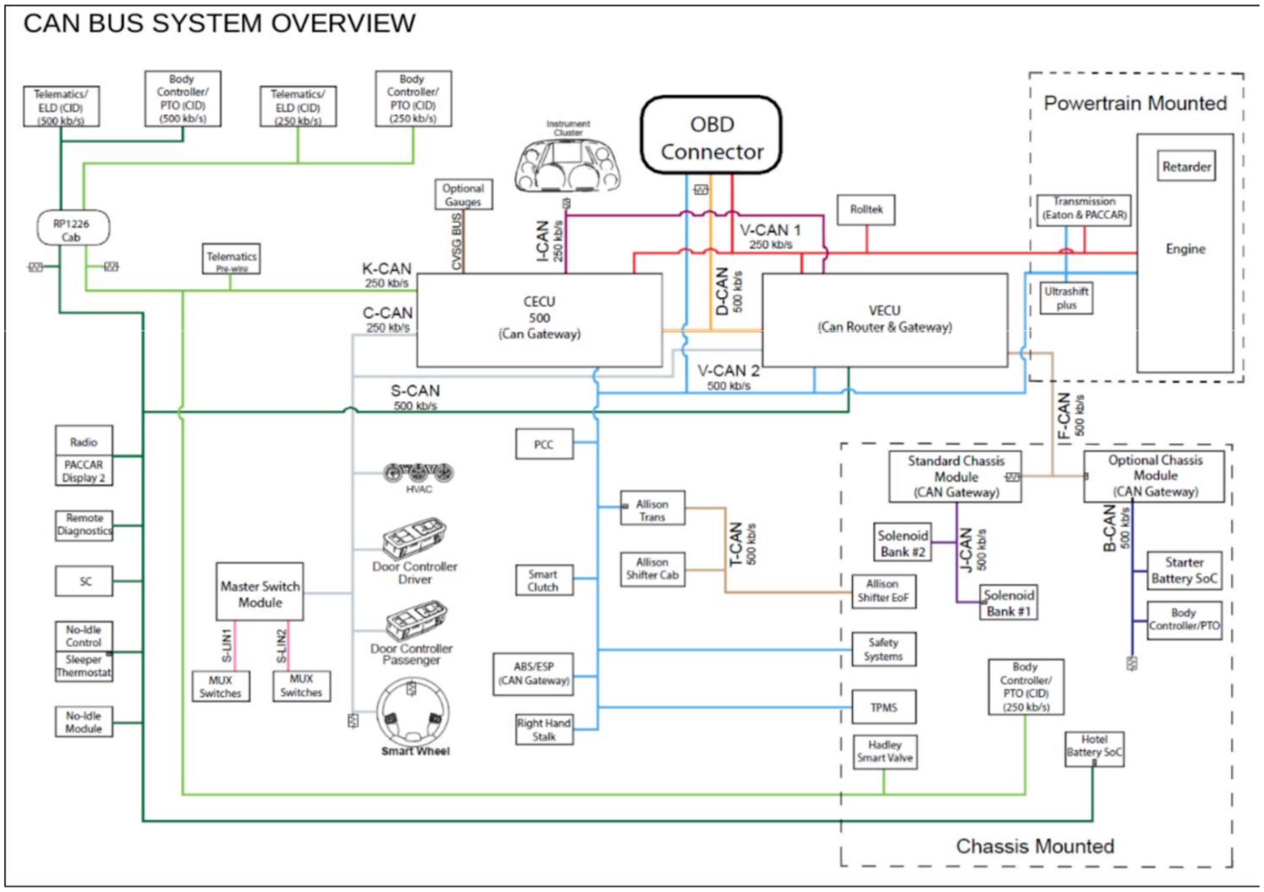


Figure 10: Architecture diagram of CAN system in truck

This diagram represents the Controller Area Network (CAN) Bus System Overview in the Challenger truck, displaying various control modules, sensors, and actuators interconnected through different CAN subnetworks. Below is a brief overview of each major section.

Overview:

The CAN bus (Controller Area Network) is a highly resilient vehicle bus standard that facilitates communication between microcontrollers and various devices within a network without the need for a central host computer. Originally developed to streamline the complex electrical wiring systems in automobiles by using a message-based protocol, CAN bus has significantly reduced the bulk and complexity of wiring harnesses, leading to improved reliability and efficiency. Its utility extends beyond automotive applications, being widely adopted in industrial automation, medical equipment, and other embedded systems where robust data exchange is crucial. This protocol enables devices to communicate with each other using a method of data transmission that is both secure and efficient, making it ideal for environments where error resilience and low latency are important.

CAN Subnetworks and Speeds:

- *Telematics/ELD (Electronic Logging Device) (500 kb/s):* High-speed networks for transmitting vital telemetry and logging data. This network typically handles high-priority or high-bandwidth data like vehicle tracking, driver behavior, and real-time feedback for fleet management systems.

- *K-CAN (250 kb/s):* A medium-speed network likely used for body control functions such as lighting, HVAC (Heating, Ventilation, and Air Conditioning), and other interior functions. The 250 kb/s speed suggests it carries less critical data than the faster 500 kb/s networks.

- *C-CAN (250 kb/s)*: Another medium-speed network possibly dedicated to chassis-related functions, including ABS (Anti-lock Braking System) and ESP (Electronic Stability Program).

- *S-CAN (500 kb/s):* This is a high-speed network, possibly dedicated to safety-critical functions such as airbags or active safety systems.

Interconnected Components:

- *ECU (Electronic Control Unit):* Acts as a central processor for the vehicle, interpreting data from various sensors and making decisions for actuators.

- *OBD (On-Board Diagnostics) Connector:* Provides diagnostic access to the vehicle. Technicians use this port to diagnose issues and monitor the health of the vehicle.

- *VECU (Vehicle Electronic Control Unit):* Manages electronic functions within the vehicle, such as engine control and transmission shifting.

- *Transmission Controls:* Include electronic modules for controlling the automatic transmission, ensuring optimal gear shifting and driveability.

- *Body Controllers:* Govern various functions related to the vehicle's body, such as door control, lighting, and auxiliary power.

- *ABS/ESP Systems:* Critical for vehicle safety, these systems control the anti-lock braking system and the electronic stability program.

- *TPMS (Tire Pressure Monitoring System):* Monitors the air pressure inside the pneumatic tires on various types of vehicles.

- *Diagnostics and Fault Tolerance:* The diagram shows a Rolltek system connected to the diagnostic port, which shows that there are advanced safety and rollover mitigation systems in place. The redundancy and fault tolerance mechanisms inherent in the CAN bus topology also ensure that the failure of a single node does not compromise the entire network.

- *Physical Layer and Signal Integrity:* The physical wiring, connectors, and terminators of the CAN bus are critical to the integrity of the communication. Differences in the bus speed (500 kb/s vs. 250 kb/s) affect the physical properties, like cable length and shielding.

- *CAN Protocols and Messaging:* Understanding the specific CAN protocols in use (e.g., CANopen, ISOBUS) and the message structure is crucial. Messages are constructed with identifiers that dictate their priority.

- *Integration of Sensors and Actuators:* The system integrates a variety of sensors and actuators, from the transmission solenoids to the HVAC controls, each playing a role in vehicle operation.

- *Performance Optimization and Real-time Considerations:* Optimizing the CAN bus involves ensuring real-time capabilities are met, which is crucial for safety-related components like airbags or brakes. Prioritization of messages and handling bus load are key factors.