THIAGO LUIGI GONÇALVES LIMA

UTILIZANDO UM OBD PARA A CRIAÇÃO DE UM DASHBOARD INTERATIVO SOBRE AS INFORMAÇÕES DO VEÍCULO

LAVRAS – MG

2023

THIAGO LUIGI GONÇALVES LIMA

UTILIZANDO UM OBD PARA A CRIAÇÃO DE UM DASHBOARD INTERATIVO SOBRE AS INFORMAÇÕES DO VEÍCULO

Trabalho de conclusão de curso apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

Prof. PhD Luiz Henrique Andrade Correia
Orientador

LAVRAS – MG

2023

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Lima, Thiago Luigi Gonçalves
UTILIZANDO UM OBD PARA A CRIAÇÃO DE UM DASHBOARD INTERATIVO SOBRE AS INFORMAÇÕES DO VEÍCULO / Thiago Luigi Gonçalves Lima. – Lavras : UFLA, 2023.
73 p. : il.

TCC(Graduação)–Universidade Federal de Lavras, 2023.
Orientador: Prof. PhD Luiz Henrique Andrade Correia.
Bibliografia.

1. OBD-II. 2. Aplicação. 3. Android. 4. Flutter. 5. Bluetooth.

CDD-808.066

THIAGO LUIGI GONÇALVES LIMA

UTILIZANDO UM OBD PARA A CRIAÇÃO DE UM DASHBOARD INTERATIVO SOBRE AS INFORMAÇÕES DO VEÍCULO
USING AN OBD TO CREATE AN INTERACTIVE DASHBOARD WITH VEHICLE INFORMATION

Trabalho de conclusão de curso apresentado à Universidade Federal de Lavras, como parte das exigências do Curso de Ciência da Computação, para a obtenção do título de Bacharel.

APROVADA em 14 de Julho de 2023.

Prof. PhD Luiz Henrique Andrade Correia	UFLA
Prof. Dr Hermes Pimenta de Moraes Júnior	UFLA
Prof. Dr Neumar Costa Malheiros	UFLA

gov.br
Documento assinado digitalmente
LUIZ HENRIQUE ANDRADE CORREIA
Data: 24/07/2023 19:38:06-0300
Verifique em https://validar.iti.gov.br

Prof. PhD Luiz Henrique Andrade Correia
Orientador

LAVRAS – MG
2023

Dedico este trabalho a minha família.

AGRADECIMENTOS

Agradeço aos meus pais por terem acreditado em mim e me incentivado a buscar a formação acadêmica. A minha irmã por ser minha inspiração criativa. A minha esposa por me ajudar todo dia com as inúmeras coisas que lhe peço. Aos meus amigos Davi e Gian por terem me ajudado e emprestado seus carros.

O homem não teria alcançado o possível se, repetidas vezes, não tivesse tentado o impossível.
(Max Weber)

RESUMO

Desde a década de 1990, os carros vêm equipados com diversos sensores que propiciam várias informações sobre o estado do veículo e como ele está sendo utilizado. Alguns desses dados acabam na interface a que o motorista tem acesso, no painel, tais como: temperatura, velocidade ou rotação do motor. Entretanto, veículos possuem muitas outras informações que não são repassadas para o motorista, mas que poderiam ser, tais como a tensão da bateria, por exemplo. Mesmo não sendo exibidos, estes dados são coletados pelo veículo e podem ser recuperados através de uma entrada CAN (Controller Area Network). O objetivo deste projeto é criar uma aplicação multiplataforma que se conectará a um adaptador OBD Bluetooth e receberá essas informações. Ao final do projeto, espera-se criar um dashboard para que o motorista possa ver as várias informações do veículo de forma simples e eficiente.

Palavras-chave: OBD-II. Aplicação. Android. Flutter. Bluetooth.

ABSTRACT

Since the 1990s, cars have been equipped with various sensors that provide various information about the state of the vehicle and how it is being used. Some of this data ends up in the interface to which the driver has access, on the dashboard, such as: temperature, speed or engine speed. However, vehicles have a lot of other information that is not passed on to the driver, but that could be, such as the battery voltage, for example. Even if not displayed, this data is collected by the vehicle and can be retrieved through a CAN (Controller Area Network) entry. The goal of this project is to create a cross-platform application that will connect to a Bluetooth OBD adapter and receive this information. At the end of the project, we hope to create a dashboard so that the driver can see the various vehicle information in a simple and efficient way.

Keywords: OBD-II. Application. Android. Flutter. Bluetooth.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

Em 2022, a Organização Mundial de Saúde (WHO) reportou que aproximadamente 1,3 milhão de pessoas morrem a cada ano como resultado de acidentes de trânsito. Cerca de 3.700 pessoas morrem nas ruas a cada dia em todo o mundo. Alguma das causas são: baixos padrões de segurança dos veículos, falta de fiscalização e aplicação da lei, pessoas cansadas ou distraídas ao volante, pessoas dirigindo sob a influência de drogas ou álcool, velocidade alta ou a falta do uso do cinto de segurança ou capacete quando ocorre o acidente (WHO TEAM, 2022).

Na última década, a indústria automobilística tem buscado aprimorar a segurança do motorista e mitigar o número de acidentes instalando nos veículos dispositivos de segurança como freios ABS, *airbags*, sistemas de controle de estabilidade e tração, sensores de ultrapassagem, etc. (PöGEL; WOLF, 2012). Protocolos como o OBD2 (*On-Board Diagnostics*) também têm sido adotados, visando facilitar a detecção de problemas em veículos e fornecer informações fundamentais ao motorista, como velocidade e temperatura do motor, velocidade do veículo, tensão da bateria, entre outros (FURMANCZYK et al., 2014). Tais informações não são sempre disponibilizadas nos painéis de todos os veículos por padrão, o que reforça a importância deste protocolo, que pode coletá-las e disponibilizá-las a outros aplicativos ou dispositivos.

1.1 Descrição do problema

Alguns aplicativos que mostram informações do OBD, como o Torque Pro (HAWKINS, 2010), um dos mais antigos nesse gênero, e o Car Scanner ELM OBD2 (OVZ, 2018), possuem interfaces Android que exibem as informações fornecidas pela ECU (*Electronic Control Unit*, ou unidade de controle eletrônico, em uma tradução direta) do veículo, obtidas através do OBD2. No entanto, tais aplicações, além de estarem relativamente obsoletas, são limitadas ao considerarmos que não são *open source* e estão disponíveis apenas para o sistema operacional Android.

1.2 Motivação

O trabalho apresentado por Aris et al., "*Development of OBD-II Driver Information System*" (2007), aliado ao problema apresentado é o que funcionou como motivação para este trabalho.

A questão de ser *open source* é fundamental, pois além da privacidade, há também a capacidade de se expandir o que o aplicativo faz. É possível acrescentar funcionalidades específicas no aplicativo para cada situação; criar uma versão voltada para pesquisa, que registra os dados brutos para uma análise externa, ou que envia os dados diretamente do *smartphone* para um servidor em tempo real; e até mesmo criar uma versão mais simples do aplicativo para uso geral que só mostra as informações em tempo real, sem registrar nada. Já em relação à privacidade, pode-se utilizar uma versão com um algoritmo de encriptação mais avançado.

Além disso, o fato de que os aplicativos citados, bem como a maioria encontrada no mercado, são feitos tendo apenas em uma plataforma em mente, limita os usuários àquela plataforma. Ao oferecer o aplicativo em uma base multiplataforma, torna-se possível levar todas as suas vantagens para diversos meios. Assim, o usuário pode utilizar o aplicativo da forma que preferir em seus dispositivos.

1.3 Objetivo e solução proposta

Dado o exposto, o presente trabalho visa o desenvolvimento de uma nova aplicação *open source* que apresente os dados disponibilizados pelo OBD2 utilizando o *framework* Flutter, que possibilitará que venha a ser multiplataforma, além de ser extensibilidade em termos de quantidade e variedade de informações exibidas. No âmbito desta pesquisa, apenas um número seleto de dados será utilizado devido a limitações de escopo e dimensão.

Na seção a seguir, com o objetivo de embasar a proposta atual e definir os conceitos essenciais ao trabalho, o referencial teórico será desenvolvido, tratando das concepções de OBD, Protocolos de Comunicação, Comandos, entre outros. Em seguida, os trabalhos relacionados serão brevemente discutidos, identificando os aspectos positivos e negativos que validam a justificativa da presente pesquisa. Posteriormente, a metodologia e o desenvolvimento da aplicação serão trabalhados, seguidos pelos resultados, considerações finais do projeto e, por fim, as referências utilizadas.

2 REFERENCIAL TEÓRICO

Com a quantidade de carros circulando pelas ruas do mundo e o número de acidentes aumentando e matando cada dia mais, várias pesquisas têm sido realizadas para tentar mitigar este problema e aumentar a segurança dos motoristas. A indústria automobilística também tem buscado aprimorar a segurança do motorista e mitigar o número de acidentes, instalando nos veículos dispositivos de segurança como freios ABS, *airbags*, sistemas de controle de estabilidade e tração, sensores de ultrapassagem, etc. (PöGEL; WOLF, 2012).

Uma das tecnologias presente nos veículos atuais para auxiliar na segurança e no conforto ao dirigir é chamada de *Advanced Driver Assistance Systems* - ADAS (Sistema de Assistência Avançada de Direção, em tradução livre). É um sistema de controle veicular que usa sensores (tais como radar, LiDAR, lasers, visão computacional e processamento de imagem) para auxiliar o motorista a reconhecer e reagir a situações potencialmente perigosas no trânsito (GIETELINK et al., 2006).

Os veículos também podem utilizar comunicação V2V (*vehicle to vehicle*), V2I (*vehicle to infrastructure*), V2N (*vehicle to network*), V2P (*vehicle to pedestrian*) ou V2X (*vehicle to everything*). A comunicação V2V é uma tecnologia que permite que os automóveis troquem informações entre si com o intuito de prevenir acidentes. A tecnologia utiliza rádio-frequência para estabelecer a conectividade entre os carros, permitindo que eles enviem informações a até 300 metros de distância, inclusive para áreas distantes da visão do motorista.

O padrão utilizado pelo V2V é por redes sem fio usando a especificação IEEE 802.11p, uma variante do padrão usado para Wi-Fi (CANALTECH, 2014). Segundo a NHTSA (*National Highway Traffic Safety Administration*), a padronização é importante para a comunicação entre veículos de diferentes fabricantes. O V2V não identifica veículos individualmente, nem recolhe dados sobre o veículo ou proprietário (CANALTECH, 2014).

A comunicação V2I é um sistema que permite que os veículos se comuniquem com a infraestrutura de beira de estrada, como semáforos, sinalização rodoviária e estações meteorológicas (TECHTARGET, 2017).

Já a comunicação V2N conecta um veículo à infraestrutura da via, *data centers* e outros carros. Com isso, os motoristas são transformados em uma espécie de “batedores de estrada”. Portanto, se o sistema de navegação apresenta dificuldades com alguma tarefa que envolva precisão ou mudança de rua, um carro conectado a V2N pode se comunicar com outro para se atualizar e se ajustar (TEAGUE, 2021).

A comunicação V2P, por sua vez, é uma das mais complicadas dentre todas as citadas. A ideia por trás deste tipo de comunicação é que os veículos serão capazes de se comunicar com dispositivos móveis de pedestres para evitar acidentes. Isso pode incluir pessoas que estão caminhando, ciclistas ou até mesmo pessoas que estão saindo de transportes públicos. Se um veículo está tendo dificuldades para parar numa esquina, por exemplo, seus sistemas emitiriam uma notificação para os pedestres próximos para que eles esperem e não atravessem a rua (TEAGUE, 2021).

Por fim, a comunicação V2X se trata de uma comunicação entre um veículo e qualquer entidade que pode afetá-lo ou ser afetada por ele. É um sistema de comunicação veicular que incorpora outros tipos de comunicação mais específicos, como V2V, V2I, V2N, e V2P. A ideia geral é que um veículo seria capaz de usar suas ferramentas internas de comunicação para entregar em tempo real informações do tráfego, reagir preventivamente à mudanças nas condições da vida, reconhecer sinais rodoviários, avisos e mais (TEAGUE, 2021).

O fato de usar dados que descrevem o ambiente externo ao veículo ao invés de apenas dados internos é o que diferencia o ADAS do *Driver-Assistance Systems* - DAS (Sistema de assistência ao motorista, em tradução livre) (GALVANI, 2019). Também é possível ter fontes adicionais que não partem do próprio veículo, como por exemplo: outro veículo (utilizando comunicação V2V) e infraestrutura (utilizando comunicação V2I) (ARENA; PAU, 2019). ADAS são considerados sistemas em tempo real, uma vez que reagem rapidamente a múltiplas entradas e priorizam a informação recebida que irá evitar acidentes. Os sistemas usam um escalonamento de prioridade preventiva para organizar qual tarefa precisa ser realizada primeiro (SHAOUT; COLELLA; AWAD, 2011).

ADAS são categorizados em diferentes níveis dependendo da autonomia adotada pelo sistema e da escala providenciada pela SAE (*Society of Automotive Engineers*) e existem 6:

- No nível 0, o sistema não pode controlar o carro e só pode providenciar informação para que o motorista a interprete. Alguns sistemas considerados de nível 0 são: sensores de estacionamento, reconhecimento de sinalização de trânsito, sistema de informação de ponto cego e aviso de colisão frontal.

- Os níveis 1 e 2 são bem similares, uma vez que ambos dependem da decisão do motorista. A diferença é que o nível 1 pode assumir o controle apenas sobre uma funcionalidade, enquanto que o nível 2 pode assumir o controle sobre múltiplas funcionalidades. Exemplos de sistemas de nível 1: controle de cruzeiro adaptativo, assistente de frenagem de

emergência e sistema de manter o veículo na faixa. Exemplos de sistemas de nível 2: assistente de rodovia, sistema autônomo de evasão de obstáculo e baliza autônoma.

- Do nível 3 ao 5, a quantidade de controle do veículo aumenta, sendo o nível 5 onde o veículo se torna completamente autônomo. O sistema de motorista de rodovia é um sistema de nível 3, e o sistema de manobrista automatizado é de nível 4. Ambos não estavam plenamente em uso em 2019 (GALVANI, 2019).

De forma resumida, os níveis podem ser entendidos como: Nível 0 - Sem automação; Nível 1 - Controle compartilhado; Nível 2 - As mãos do motorista podem ficar livres; Nível 3 - Não é preciso prestar atenção; Nível 4 - Não é preciso pensar a respeito; Nível 5 - O volante é opcional.

Nas últimas décadas, inúmeras pesquisas relacionadas à estratégia ao se modelar e aprimorar sistemas de assistência e de monitoramento de direção foram desenvolvidas. Trabalhos relacionados ao ADAS no geral incluem: desenvolvimento de uma assistência fundamental para o motorista (como modelos de comportamento do motorista e segurança em rodovias), testes virtuais e o desenvolvimento desses ambientes (simuladores), métodos de teste, sensores (radar, LIDAR, visão, ultrassônico, etc.), fusão de dados e representação ambiental (mapas digitais, locais nos mapas, sistemas de *backend*), atuadores (sistemas de frenagem e volantes), motores e interface humana (dispositivos de entrada, visualização de dados, sistema de detecção da condição do motorista e sistemas de aviso), melhorias na estabilização de sistemas existentes (estabilização da frenagem e da direção), melhoria na capacidade de navegação pelas ruas (assistência de estacionamento e mudança de faixa), e o desenvolvimento de outros sistemas de assistência no futuro (WINNER et al., 2015).

Um exemplo de aplicação que pode ser integrado ao ADAS é a aplicação ACORE (*Algorithm for COllision waRning Error correction*). Ela analisa e envia mensagens sobre uma possível colisão com um veículo à frente (NETO et al., 2016). Os veículos transmitem mensagens periodicamente com informações sobre sua posição geográfica e velocidade (ALMEIDA et al., 2019). Quando um veículo recebe uma nova mensagem, ele analisa a sua distância em relação ao remetente e compara o resultado com a distância segura de frenagem. Essa distância é calculada considerando a eficiência do sistema de freio, peso do veículo, ângulo de inclinação da estrada, etc. (CHEN; SHEN; WANG, 2013). Se a distância calculada entre os veículos for menor que a distância segura, um alerta de colisão é emitido para o motorista. A ACORE limita-

se a emitir um alerta de possível colisão entre dois veículos, sem mecanismos para retransmitir alertas para outros veículos ou infraestruturas próximas.

Uma aplicação semelhante ao ACORE que funciona com um sistema de retransmissão de alertas pode utilizar redes veiculares, ou VANETs (*Vehicular Ad hoc Network*), para que todos os carros de uma região estejam cientes de seus arredores e não colidam entre si. Para realizar esta retransmissão de alertas, é possível fazer uso das VANETs, que propõem uma solução para a comunicação veicular, visando evitar situações de risco, auxiliar em caso de acidentes, prevenir colisões, controlar congestionamentos, etc. (AL-SULTAN et al., 2014).

Já as redes 5G têm potencial para fornecer comunicações entre veículos do tipo D2D (*Device to Device*). Com uma área de cobertura mais ampla, baixa latência e altas taxas de dados, essas redes devem oferecer melhor suporte a dispositivos com alta mobilidade, como veículos (QI et al., 2018), quando comparadas às redes de gerações anteriores, que apresentavam alta latência na transmissão de mensagens de alerta.

Espera-se que o 5G explore os benefícios da comunicação D2D, permitindo que os dispositivos se comuniquem diretamente com seus vizinhos na banda licenciada. Neste contexto, as Estações-Base podem ter diferentes níveis de envolvimento, desde o modo convencional (no qual o controle e os dados são passados pela Estação-Base), até deixar os dispositivos completamente responsáveis pelo estabelecimento da comunicação e troca de dados (TEHRANI; UYSAL; YANIKOMEROGLU, 2014).

Muitas destas pesquisas que buscam melhorar o conforto e a segurança do motorista, como por exemplo (HERMAWAN; HUSNI, 2020), (ARIS et al., 2007), (CHEN; PAN; LU, 2015), (MONIAGA et al., 2018) e (YEN et al., 2021), acabam sendo utilizadas no desenvolvimento de sistemas ADAS. Isso se deve ao fato de que elas utilizam os dados obtidos do veículo, como condição do veículo, velocidade, frenagem e direção, para descobrir padrões de direção. Estes padrões são utilizados para treinar modelos de inteligência artificial. Para capturar estes dados, tais pesquisas utilizam, dentre outras coisas, o protocolo OBD (*On-Board Diagnostic*), mais especificamente a sua segunda geração, o OBD2.

2.1 OBD

Os sistemas OBD dão ao motorista ou mecânico acesso ao status de vários subsistemas do veículo. A quantidade de informação de diagnóstico disponível via OBD variou muito desde o momento de sua introdução, no começo dos anos 1980. As primeiras versões do OBD sim-

plesmente acendiam uma luz que indicava que algo não estava funcionando corretamente; elas não ofereciam nenhuma informação acerca da natureza do problema. As implementações modernas do OBD usam uma porta de comunicação digital padronizada para providenciar dados em tempo real, além de uma série padronizada de códigos de diagnóstico de problemas (*Diagnostic Trouble Codes* - DTC, em inglês), que permite que uma pessoa consiga identificar e consertar rapidamente os problemas no veículo.

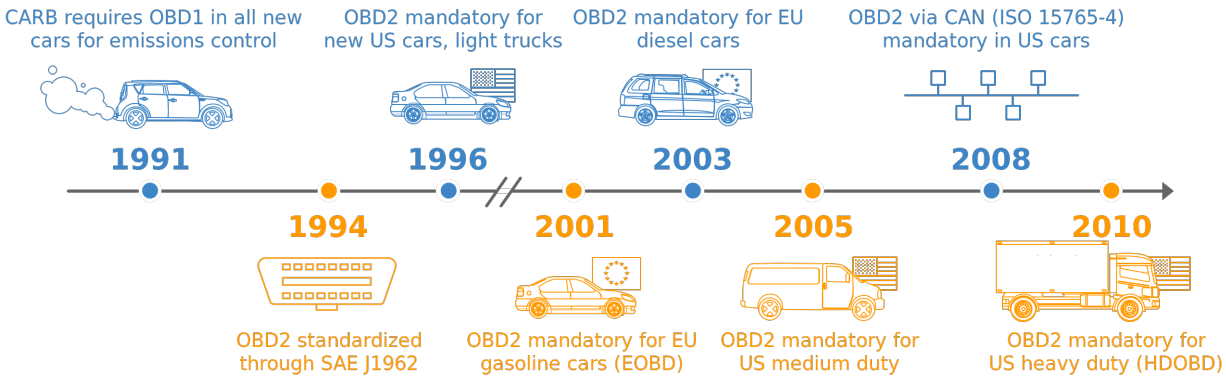
O motivo inicial de sua criação era encorajar os fabricantes de carros a desenvolverem um sistema de controle de emissão que permanecesse efetivo e funcional por toda a vida útil do veículo. Porém, a sua primeira versão não deu certo, uma vez que as informações de diagnóstico não eram padronizadas. Cada fabricante tinha seu próprio conector de diagnóstico (*Diagnostic Link Connector* - DLC), a posição onde tal conector se encontrava também mudava, os códigos de erro e diagnóstico variavam e cada um tinha um processo diferente para a leitura dos mesmos. Isso tornava muito difícil trabalhar com este sistema.

Surgiu então, no início dos anos 1990, uma segunda versão: o OBD2. Se tratava de uma melhoria tanto em capacidade quanto padronização. O novo padrão OBD2 especifica o tipo de conector de diagnóstico e sua pinagem (também chamado de DLC, mas agora sendo *Data Link Connector*), os protocolos de sinalização eletrônica disponíveis e o formato das mensagens. Ele também proporciona uma lista de parâmetros dos veículos para serem monitorados, além de instruções sobre como codificar os dados de cada parâmetro. No conector, há um pino responsável por fornecer energia para a ferramenta de leitura a partir da bateria. Isso dispensa a necessidade de se conectar a uma fonte externa de energia. Por fim, o padrão OBD2 também providencia uma extensa lista de DTCs. Como resultado desta padronização, um dispositivo único consegue consultar os computadores de bordo de qualquer veículo. Uma linha do tempo do uso do OBD pode ser vista na Figura 2.1.

Como sua criação se deu por causa da preocupação com a emissão de poluentes do veículo, apenas dados e códigos relacionados com a emissão do veículo devem ser transmitidos através dele. Porém, muitos fabricantes adotaram o DLC do OBD2 como ponto único de entrada do veículo no qual todos os sistemas são diagnosticados e programados. Os DTCs do OBD2 possuem quatro dígitos, precedidos por uma letra, sendo: P (de *powertrain*) para o motor e a transmissão, B (de *body*) para o corpo, C para o chassis e U para conexão.

Praticamente todos os carros modernos possuem suporte ao OBD2, e a maioria utiliza o protocolo de comunicação CAN (*Controller Area Network*). É importante dizer que, mesmo

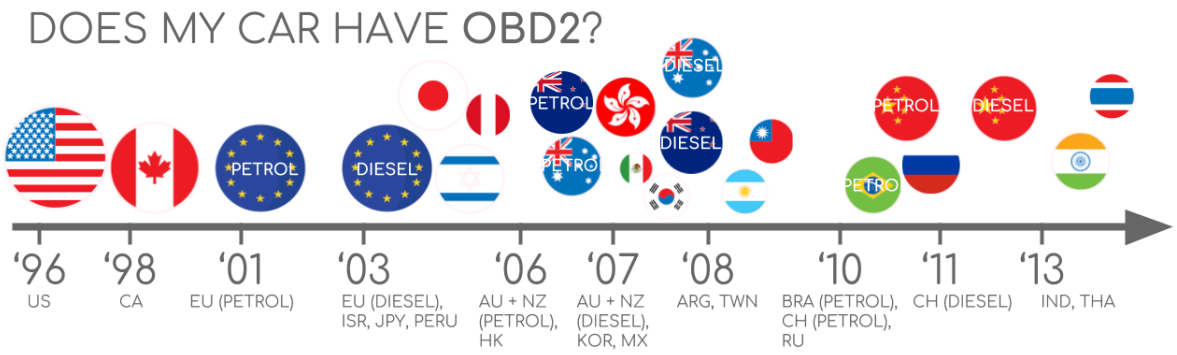
Figura 2.1 – Linha do tempo do OBD



Fonte: CSS Electronics (2023)

que o veículo possua um conector OBD2 de 16 pinos, ele pode não oferecer suporte ao OBD2. Uma forma de determinar se um veículo é compatível ou não é identificar quando e onde ele foi comprado (CSS Electronics, 2023), como mostra a Figura 2.2.

Figura 2.2 – Linha do tempo de adoção do OBD



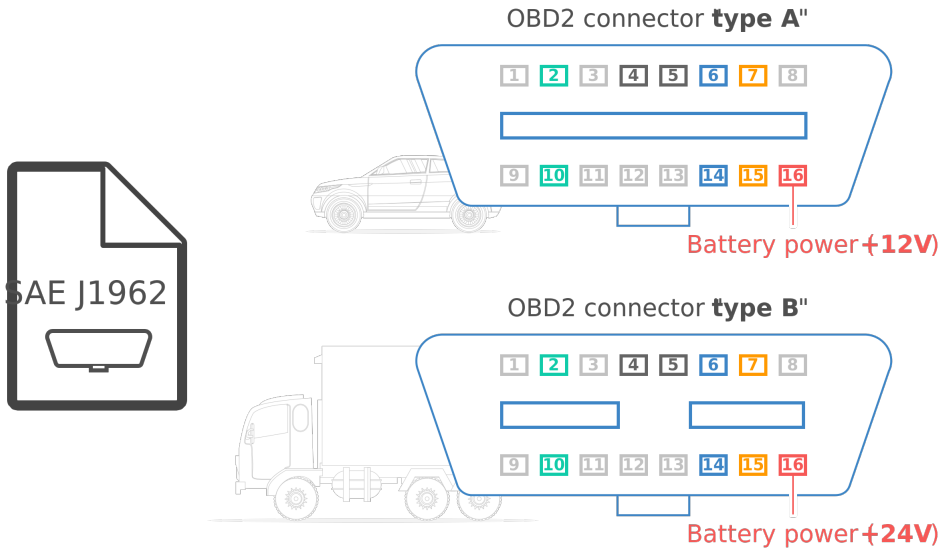
Fonte: CSS Electronics (2023)

2.2 Conector SAE J1962

A especificação do OBD2 providencia uma interface de hardware padronizada, o conector J1962 de 16 pinos. Este conector veio em dois modelos, o OBD2 A, que é mais utilizado em carros de pequeno porte e o OBD2 B, que é mais utilizado em carros de grande porte. Ambos possuem os mesmos pinos, porém eles fornecem duas saídas de fonte de alimentação diferentes (12V para o tipo A e 24V para o tipo B). Para ajudar a distinguir entre os dois tipos, o tipo B possui uma ranhura interrompida no meio, enquanto que, no tipo A, a mesma é contínua, como

mostra a Figura 2.3. Assim, um adaptador do tipo B será compatível tanto com o tipo A quanto com o tipo B, mas um adaptador do tipo A não será compatível com a entrada do tipo B (CSS Electronics, 2023).

Figura 2.3 – Tipos de conectores OBD

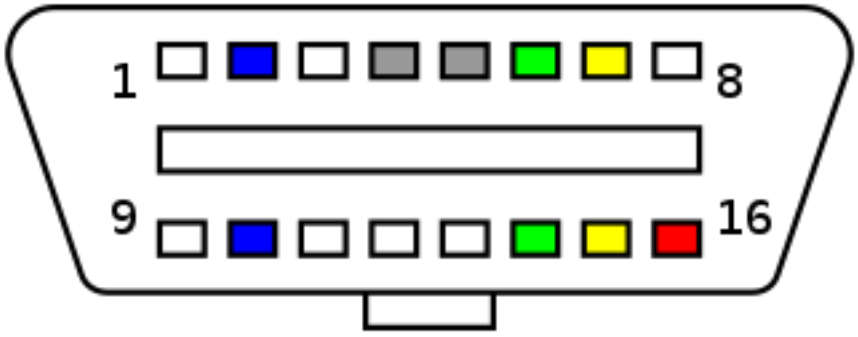


Fonte: CSS Electronics (2023)

Ao contrário da primeira versão do OBD, que poderia ser encontrado embaixo do capô, a versão 2 do padrão exigia que o conector se encontrasse a, no máximo, 0.61m do volante (a menos que houvesse alguma exceção por parte da fabricante, mas ainda era necessário que estivesse ao alcance do motorista).

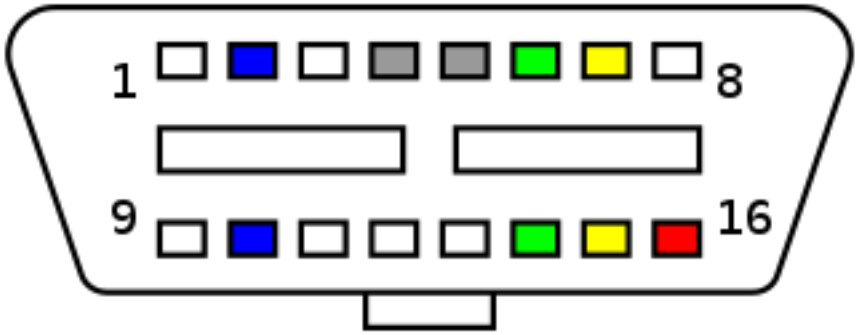
A SAE J1962 (PREEZ, 2022) define a pinagem do conector utilizando as Figuras 2.4 e 2.5 a seguir, bem como as informações contidas na Tabela 2.1.

Figura 2.4 – Conector OBD2 tipo A



Fonte: Wikipédia (2014)

Figura 2.5 – Conector OBD2 tipo B



Fonte: Wikipédia (2014)

Tabela 2.1 – Pinos do OBD2

Pino	Função	Pino	Função
1	A critério da fabricante	9	A critério da fabricante
2	*Bus positive line* SAE J1850 PWM e VPW	10	*Bus negative line* SAE J1850 PWM somente (não está presente na SAE J1850 VPW)
3	A critério da fabricante	11	A critério da fabricante
4	Aterramento do chassis	12	A critério da fabricante
5	Aterramento do sinal	13	A critério da fabricante
6	CAN *high* ISO 15765-4 e SAE J2284	14	CAN *low* ISO 15765-4 e SAE J2284
7	K-*Line* ISO 9141-2 e ISO 14230-4	15	L-*Line* ISO 9141-2 e ISO 14230-4
8	A critério da fabricante	16	Tensão da bateria

Fonte: Wikipédia (2014)

Como visto na Tabela 2.1, a pinagem do conector depende do protocolo de comunicação. O protocolo de comunicação mais utilizado é o CAN (via ISO 15765), ou seja, o pino 6 (CAN-H) e o 14 (CAN-L) estarão conectados.

2.3 CAN

O *On-board diagnostics*, OBD2, é um "protocolo de alto nível" (similar à linguagem). Ele também pode ser comparado a outros protocolos como o J1939 e o *CANopen*. Já o CAN é um método para comunicação (como um telefone) (CSS Electronics, 2023).

O Controlador de Rede de Área (*Controller Area Network*, CAN *bus*) é um protocolo robusto de barramento veicular. Ele foi feito para permitir que microcontroladores e dispositivos

pudessem se comunicar entre si sem um computador dedicado. É um protocolo baseado em mensagens; para cada dispositivo, o dado é transmitido de forma serial de forma que mais de um dispositivo possa transmitir ao mesmo tempo. O dispositivo com a maior prioridade continua transmitindo, e aqueles com uma prioridade menor esperam.

Seu desenvolvimento começou em 1983 na Robert Bosch GmbH. O protocolo foi liberado oficialmente em 1986 na Conferência SAE em Detroit, Michigan. O primeiro chip controlador CAN foi introduzido pela Intel em 1987, e logo em seguida pela Philips (CAN in Automation, 2023). Em 1993, a ISO liberou o padrão ISO 11898 que foi, posteriormente, reestruturado em duas partes: a ISO 11898-1, que cobre a camada de enlace, e a ISO 11898-2 que cobre a camada física para CANs de alta velocidade. A ISO 11898-3 foi lançada depois e cobre a camada física de uma CAN de baixa velocidade e tolerante a falha. Tanto a ISO 11898-2 quanto a ISO 11898-3 não são parte da especificação Bosch CAN 2.0.

O protocolo OBD2 especifica um conjunto de 5 protocolos no qual ele pode ser executado. Desde 2008, o protocolo CAN *bus* (ISO 15765) tem sido o protocolo obrigatório para o OBD2 em todos os carros vendidos nos EUA (CSS Electronics, 2023). A ISO 15765 se refere a um conjunto de restrições aplicadas ao padrão CAN (que é, por sua vez, definido pela ISO 11898). Pode-se dizer que a ISO 15765 “define a CAN para carros”. A ISO 15765-4 descreve a camada física, de enlace e de conexão, buscando padronizar a interface CAN para equipamento de teste externo. A ISO 15765-2, por sua vez, descreve a camada de transporte (ISO TP) para mandar fragmentos (*frames*) da CAN com cargas que excedem 8 bytes. Este subpadrão também é chamado de comunicação de diagnóstico sobre CAN (*Diagnostic Communication over* CAN - DoCAN), como pode ser visto na Figura 2.6.

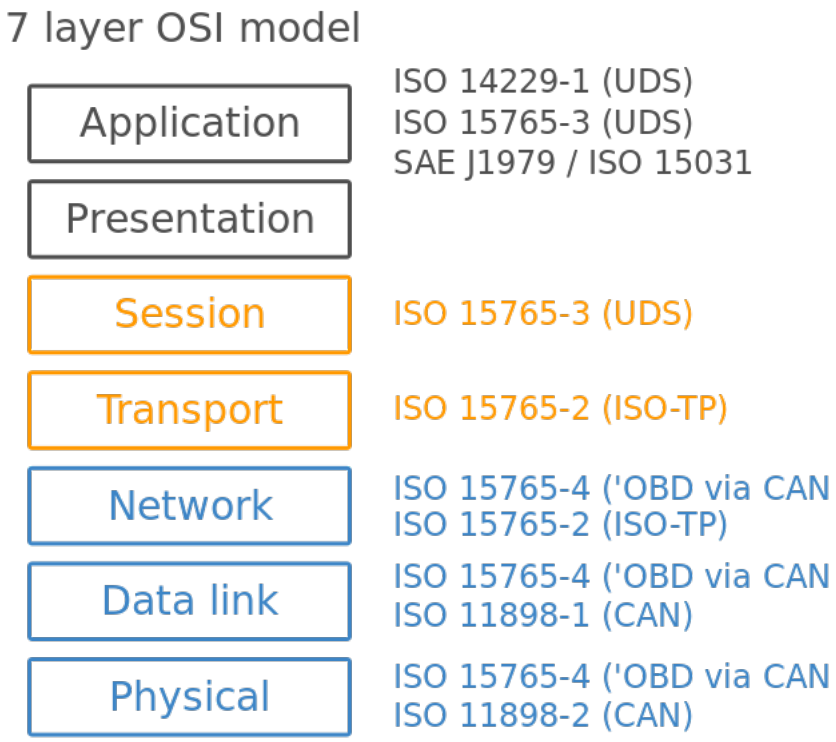
2.4 ELM 327

O ELM 327 é um microcontrolador programado para traduzir a interface OBD. O ELM 327 original foi implementado no microcontrolador PIC18F2480 da Microchip Technology. Ele abstrai o protocolo de baixo nível e apresenta uma interface simples que pode ser chamada através de uma UART (*Universal Asynchronous Receiver Transmitter*, Transmissor, ou Receptor Universal Assíncrono em tradução própria). Geralmente, essa UART é chamada utilizando um software conectado por USB, Bluetooth ou Wi-Fi (MILLER, 2023).

As funções podem variar, mas tipicamente se tratam das seguintes:

- leitura de códigos de diagnóstico, tanto códigos genéricos quanto de fabricantes;

Figura 2.6 – Camadas do modelo OSI



Fonte: CSS Electronics (2023)

- limpeza de alguns códigos de erro para desligar a luz de “Confira o motor”;
- exibição de dados dos sensores;
- rotação do motor;
- temperatura do líquido de refrigeração;
- status do sistema de combustível;
- posição do acelerador.

Os protocolos que o ELM327 suporta (ELM Electronics, 2023) são os seguintes:

- SAE J1850 PWM (41.6 kbit/s);
- SAE J1850 VPW (10.4 kbit/s);
- ISO 9141-2 (5 baud init, 10.4 kbit/s);
- ISO 14230-4 KWP (5 baud init, 10.4 kbit/s);

- ISO 14230-4 KWP (fast init, 10.4 kbit/s);
- ISO 15765-4 CAN (11 bit ID, 500 kbit/s);
- ISO 15765-4 CAN (29 bit ID, 500 kbit/s);
- ISO 15765-4 CAN (11 bit ID, 250 kbit/s);
- ISO 15765-4 CAN (29 bit ID, 250 kbit/s);
- SAE J1939 (250 kbit/s);
- SAE J1939 (500 kbit/s).

O aparelho utilizado para conectar a porta SAE J1962 neste trabalho e extrair os dados do veículo utiliza o ELM327.

2.4.1 Protocolos de comunicação

Um veículo compatível com OBD2 pode usar qualquer um dos 5 protocolos de comunicação descritos a seguir (OBDTESTER.COM, 2020).

2.4.1.1 SAE J1850 PWM

Este protocolo era mais usado pela Ford. Ele usa o pino 2 e 10, e sua taxa de transmissão é de 41.6Kb/s. A sigla PWM vem do termo em inglês *Pulse-Width Modulation*, que é traduzido como Modulação por Largura de Pulso (MLP). O pino 2 é o *Bus* + e o pino 10 é o *Bus* -. A alta voltagem é de +5 V. O comprimento da mensagem é restrito a 12 bytes, incluindo a verificação cíclica de redundância.

2.4.1.2 SAE J1850 VPW

Este protocolo, por sua vez, era mais usado em veículos da GM. Ele usa o pino 2 e sua taxa de transmissão é de 10.4Kb/s. O pino 2 é o *Bus* +. A alta voltagem é de + 7 V, e o ponto de decisão é de +3.5 V. O comprimento da mensagem é restrito a 12 bytes, incluindo a verificação cíclica de redundância.

2.4.1.3 ISO 9141-2

Este protocolo é mais antigo e foi bastante utilizado na Europa pela Chrysler, bem como em veículos asiáticos entre os anos 2000 e 2004. Ele usa o pino 7 (K-*Line*) e pode conter, também, o pino 15 (L-*Line*). Possui uma taxa de transmissão assíncrona em série de 10.4 Kb/s (MAHAJAN; S.K.PARCHANDEKAR; TAHIR, 2017). O comprimento da mensagem é de no máximo 260 bytes.

2.4.1.4 ISO 14230-4 (KWP2000)

Este protocolo é muito comum em veículos desde 2003, usando a ISO 9141 K-*Line*. Ele usa o pino 7 (K-*line*) e pode também conter o pino 15 (L-*line*). Sua camada física é idêntica à ISO 9141-2. A mensagem pode conter até 255 bytes, e a sigla KWP vem de *Keyword Protocol*, que significa protocolo de palavras-chave, em uma tradução livre.

Existem duas variações deste protocolo. Elas se diferem apenas no método de inicializar a comunicação, e ambas usam 10400 bits por segundo.

- ISO 14230-4 KWP (5 baud init, 10.4 Kb/s);
- ISO 14230-4 KWP (fast init, 10.4 Kb/s).

2.4.1.5 ISO 15765-4/SAE J2480 (CAN-BUS)

Este é o protocolo mais moderno, obrigatório em todos os veículos vendidos após 2008 nos Estados Unidos. Atualmente, é utilizado na maioria dos carros. Ele usa os pinos 6 (CAN *High*) e 14 (CAN *Low*).

Existem quatro variantes deste protocolo. Elas se diferenciam apenas pelo comprimento do identificador e pela velocidade do barramento.

- ISO 15765-4 CAN (11 bit ID, 500 Kb/s);
- ISO 15765-4 CAN (29 bit ID, 500 Kb/s);
- ISO 15765-4 CAN (11 bit ID, 250 Kb/s);
- ISO 15765-4 CAN (29 bit ID, 250 Kb/s).

Um resumo das informações apresentadas acima pode ser visto na Tabela 2.2.

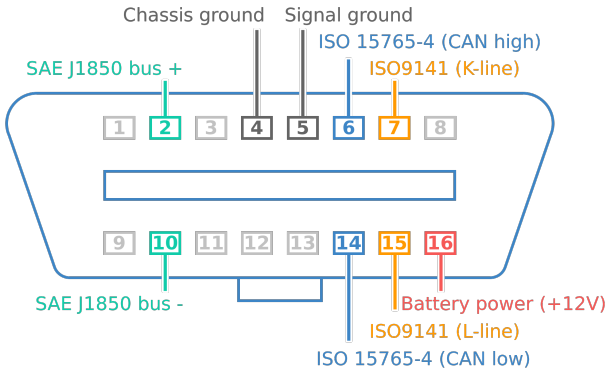
Tabela 2.2 – Protocolos OBD

Protocolo	Taxa de Transmissão
SAE J1850 PWM	41.6 Kb/s
SAE J1850 VPW	10.4 Kb/s
ISO 9141-2	10.4 Kb/s
ISO 14230-4 (KWP2000)	10.4 Kb/s
ISO 15765-4/SAE J2480 (CAN-BUS)	500 Kb/s

Fonte: Wikipédia (2014)

O protocolo pode ser determinado a partir da pinagem demonstrada na Figura 2.7

Figura 2.7 – Pinagem do conector CAN



Fonte: CSS Electronics (2023)

É possível determinar qual protocolo seu veículo está usando ao verificar a pinagem do conector OBD2, como demonstrado na Tabela 2.3.

Tabela 2.3 – Pinos utilizados por cada protocolo

Protocolo	Pino 2	Pino 6	Pino 7	Pino 10	Pino 14	Pino 15
J1850 PWM	Obrigatório	-	-	Obrigatório	-	-
J1850 VPW	Obrigatório	-	-	-	-	-
ISO9141/14230	-	-	Obrigatório	-	-	Opcional
ISO15765 (CAN)	-	Obrigatório	-	-	Obrigatório	-

Fonte: Wikipédia (2014)

A Fiat, a Alfa e a Lancia também utilizaram um CAN-BUS tolerante a falhas com taxa de transmissão de 50kb/s. Este, porém, não é compatível com o padrão OBD2.

O restante dos pinos também pode ser utilizado; entretanto, eles geralmente se conectam a outras centrais do veículo. Essas centrais não estão relacionadas com o motor. Ferramentas genéricas utilizando o OBD2 não conseguem se comunicar com outras centrais que não sejam relacionadas ao motor. Para se comunicar com essas centrais, como por exemplo as do *airbag*, som, etc., é necessário um *software* específico do fabricante como o FiCOM (Fiat/Alfa/Lancia), o FoCOM (Ford/Mazda) ou o HiCOM (Hyundai/Kia).

2.4.2 Comandos

O OBD2 permite acessar dados da unidade de controle do motor (*Engine Control Unit*, ECU) e oferece uma fonte valiosa de informação quando se está diagnosticando problemas dentro do veículo. *On-board diagnostics Parameter* IDs (OBD2 PIDs) são códigos usados em ferramentas de diagnóstico para requisitar dados do veículo.

2.4.2.1 SAE J1979

O padrão SAE J1979 define um método para requisitar vários dados de diagnóstico e uma lista de parâmetros padronizados que podem ser disponibilizados pela ECU. Esses parâmetros disponíveis são endereçados por *Parameter Identification Numbers* (PIDs, ou Números Identificadores de Parâmetros, em tradução livre) definidos pela SAE J1979. Os fabricantes não são obrigados a implementar todos os PIDs listados na J1979, e eles podem incluir PIDs proprietários que não estão listados. O sistema de requisição e recuperação de dados do PID fornece acesso a dados de performance em tempo real, assim como os códigos de erro já levantados pelo sistema.

O padrão SAE J1979 também define 10 serviços de diagnóstico. Antes de 2002, esses serviços eram referenciados como modos, e é possível ver os dois termos sendo utilizados dependendo do material. Esses serviços são apresentados na Tabela 2.4.

Alguns dos PIDs mais comuns do serviço 01 podem ser vistos na Tabela 2.5.

As letras A e B na coluna de fórmulas representam o primeiro e o segundo *byte* dos dados. Por exemplo, quando um comando retorna dois *bytes* de dados 0F 19, A = 0F e B = 19.

Os outros serviços fogem do escopo do projeto atual, portanto não serão demonstrados aqui. Para a lista completa de PIDs padrão, incluindo dos outros serviços, basta se referir a SAE J1979.

Tabela 2.4 – Serviços oferecidos pelo protocolo OBD

Serviço / Modo (hex)	**Descrição**
01	Mostrar dados atuais
02	Mostrar *frame data* congelado
03	Mostrar DTCs salvos
04	Limpar DTC e valores salvos
05	Resultados dos testes envolvendo o sensor de monitoramento de oxigênio (apenas para quando não se usa CAN)
06	Resultados dos testes (incluindo o sensor de monitoramento de oxigênio para quando se usa CAN)
07	Mostrar DTCs pendentes
08	Controle de operação de componente interno
09	Requisitar informações do veículo
0A	Mostrar DTCs que foram armazenados permanentemente

Fonte: Wikipédia (2014)

Tabela 2.5 – PIDs comuns do serviço 01

PIDs (hex)	PID (Dec)	*Bytes* de dados retornados	Descrição	Valor mínimo	Valor máximo	Unidade	Fórmula
05	5	1	Temperatura do líquido de arrefecimento do motor	-40	215	°C	A - 40
0C	12	2	Velocidade do motor	0	16383,75	rpm	$\frac{256A+B}{4}$
0D	13	1	Velocidade do veículo	0	255	km/h	A
2F	47	1	Quantidade de combustível no tanque	0	100	%	$\frac{100}{255}A$

Fonte: Wikipédia (2014)

2.4.2.2 CAN (11-bit) Bus Format

A *query* (busca) e a *response* (resposta) a um PID ocorrem no barramento da CAN do veículo. Requisições padrão do OBD e suas respostas usam endereços funcionais. A leitura de diagnóstico se inicia com uma *query* usando o CAN ID 7DFh, que age como um endereço de

broadcast e aceita *responses* de qualquer ID do alcance 7E8h até 7EFh. As ECUs que podem responder a *queries* consideram o endereço funcional de broadcast 7DFh e um ID designado no alcance entre 7E0h e 7E7h. Sua resposta possui o ID designado mais 8. Por exemplo: 7E8h até 7EFh. O barramento CAN também pode ser utilizado para comunicações além daquelas previstas no OBD.

2.4.2.3 Query

A *query* é enviada ao veículo pelo barramento CAN no ID 7DFh usando 8 *bytes* de dados. Esses *bytes* são demonstrados na Tabela 2.6.

Tabela 2.6 – Estrutura de uma *query*

	Byte							
Tipo de PID	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
SAE *Standard*	Número de *bytes* adicionais	Serviço	Código PID					

Fonte: Wikipédia (2014)

Um exemplo de requisição da temperatura do líquido de arrefecimento do motor pode ser visto na Tabela 2.7.

Tabela 2.7 – Exemplo de *query*

	Byte							
Tipo de PID	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
SAE *Standard*	2	01	05					

Fonte: Wikipédia (2014)

2.4.2.4 Response

O veículo responde às requisições usando uma mensagem de 8 *bytes* de dados. O número de *bytes* necessários para resposta é variável dependendo da requisição, porém a mensagem sempre possui 8 *bytes*. As Tabelas 2.8 e 2.9 demonstram um exemplo de resposta.

No exemplo da Tabela 2.9, no terceiro *byte* estaria o valor que seria usado na fórmula para calcular a temperatura do líquido de arrefecimento do motor. Supondo que na resposta veio o valor 60, e considerando que a fórmula para o cálculo é: A - 40, A é o valor vindo no

Tabela 2.8 – Estrutura de uma *response*

	Byte							
CAN *Address*	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
SAE *Standard*	Número de *bytes* adicionais	Serviço (como na *query*, porém adiciona-se 40h ao valor do serviço)	Código PID	Parâmetro, *byte* 0	Parâmetro, *byte* 1 (opcional)	Parâmetro, *byte* 2 (opcional)	Parâmetro, *byte* 3 (opcional)	Parâmetro, *byte* 4 (opcional)

Fonte: Wikipédia (2014)

Tabela 2.9 – Exemplo de *response*

	Byte							
CAN *Address*	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
SAE *Standard*	1	41h	05	Parâmetro, *byte* 0				

Fonte: Wikipédia (2014)

terceiro *byte*, ou seja, 60. Portanto, no momento da requisição, a temperatura do líquido é: 60 - 40 = 20°C. Também é possível usar *queries* e obter *responses* específicas para cada veículo, que não fazem parte do padrão SAE, mas isto foge do objetivo deste trabalho.

2.5 Ambiente e linguagens utilizadas

A seguir, encontra-se uma breve descrição da linguagem utilizada no trabalho, bem como do ambiente e das ferramentas empregadas.

2.5.1 Dart

Dart [1] é uma linguagem de programação desenvolvida pela Google (DART, 2023). Ela foi lançada na GOTO Conference 2011, e seu objetivo inicial era substituir a linguagem JavaScript como linguagem principal presente nos navegadores. Os programas desenvolvidos nesta linguagem podem ser executados em uma máquina virtual ou compilados para o JavaScript. Em 2013, sua primeira versão estável foi lançada, Dart 1.0. Em agosto de 2018, o Dart 2.0 foi lançado, um *reboot* da linguagem com um novo foco: o desenvolvimento *client-side* para Web e dispositivos móveis.

[1] https://dart.dev/

2.5.2 Flutter

O Flutter é um kit de desenvolvimento de interface de usuário de código aberto, criado pela Google em 2015, baseado na linguagem de programação Dart. O Flutter possibilita a criação de aplicativos compilados nativamente para os sistemas operacionais Android, iOS, Windows, Mac, Linux, Fuchsia e Web. Sua primeira versão estável foi lançada em dezembro de 2018 [2] (FLUTTER, 2023).

O Flutter é composto de 5 componentes fundamentais:

- a plataforma Dart;
- o Flutter *Engine*;
- a *Foundation Library*;
- *Design-Specific Widgets*;
- ferramentas de desenvolvimento Flutter.

2.5.2.1 A plataforma Dart

Os aplicativos Flutter são escritos na linguagem Dart, e portanto fazem uso de várias das funcionalidades da linguagem. Enquanto a aplicação é desenvolvida e depurada, ela também é executada na máquina virtual Dart, que possui uma *engine* de execução *just-in-time*. Isso possibilita uma compilação rápida, assim como o "*hot reload*", que trata de modificações no código fonte da aplicação que podem ser injetadas numa aplicação em execução. Isso permite que a aplicação possa ser modificada e testada mais rapidamente, e sua reinstalação não é necessária sempre que há uma mudança no código. O Flutter expande ainda mais o conceito de "*hot reload*" oferecendo suporte ao *stateful hot reload*, que trata das mudanças no código fonte serem refletidas imediatamente na aplicação em execução, sua reinicialização não sendo necessária e nenhum estado sendo perdido.

Para uma melhor performance, a versão de produção (ou *release*) dos aplicativos Flutter em todas as plataformas utilizam uma compilação AOT (*ahead-of-time*), exceto para a versão Web, na qual o código é convertido para JavaScript. O Flutter herda o gerenciador de pacotes Pub do Dart, onde é possível publicar e utilizar pacotes Dart, além de *plugins* específicos para Flutter.

[2] https://flutter.dev/

2.5.2.2 Flutter Engine

A *engine* do Flutter é escrita principalmente em C++ e provê um suporte de renderização de baixo nível, podendo utilizar tanto a biblioteca gráfica Skia, feita pela Google, quanto uma camada gráfica customizada chamada "Impeller". Além disso, ela se comunica com o SDK (*Software Development Kit*) específico de cada plataforma, como os oferecidos pelo Android ou iOS para implementar funções de acessibilidade, arquivos, rede, suporte para *plugins* nativos e mais.

2.5.2.3 Foundation Library

A *Foundation Library* (biblioteca de fundação, em uma tradução literal) é escrita em Dart e oferece classes básicas, além de funções que são utilizadas para construir as aplicações Flutter, como as APIs para comunicação com a *engine*.

2.5.2.4 Design-Specific Widgets

O *framework* Flutter possui dois kits de *widgets* que estão de acordo com linguagens de design específicas, sendo eles: Material Design e Cupertino. O Material Design possui *widgets* que implementam a linguagem de design da Google de mesmo nome. Já os *widgets* Cupertino implementam o guia de interface humana do iOS de acordo com a Apple. O Flutter permite que desenvolvedores usem cada kit onde quiserem, portanto é possível usar o kit Cupertino no Android e o Material Design no iOS, por exemplo. Pacotes de terceiros também podem ser utilizados para ajustar automaticamente o design do aplicativo para o sistema operacional no qual ele está em execução.

O Flutter mantém suporte oficial para as seguintes IDEs (*Integrated Development Environment*) e editores através de *plugins*:

- IntelliJ IDEA;
- Android Studio;
- Visual Studio Code;
- Emacs.

Outras IDEs podem ser usadas com *plugins* feitos pela comunidade ou usando as ferramentas do Flutter pela linha de comando.

2.5.2.5 Widgets

O componente básico em um aplicativo Flutter é o *widget*, que por sua vez pode ser constituído de outros *widgets*. Um *widget* descreve a lógica, a interação e o design de um elemento da interface gráfica com uma implementação similar aos componentes do React. Ao contrário de outros *toolkits cross-platform*, que desenham os *widgets* usando componentes nativos da plataforma, o Flutter renderiza os *widgets* numa base pixel-a-pixel.

Existem dois tipos fundamentais de *widgets*: *stateless* e *stateful*. Os *widgets stateless* só se atualizam quando a entrada muda. Isso faz com que eles sejam muito eficientes. Os *widgets stateful* podem chamar o método setState() para atualizar seu estado interno e ser redesenhado na tela.

Apesar dos *widgets* serem o método primário de se construir uma aplicação em Flutter, eles podem ser substituídos pelo desenho direto na tela. Essa função é usada ocasionalmente para se implementar jogos no Flutter.

3 TRABALHOS RELACIONADOS

Atualmente, muitos *softwares* seguem sendo observados e analisados após serem publicados para uso. Isso faz com que seus desenvolvedores possam ver o padrão de uso de sua criação e seus erros e possam corrigi-los ou aperfeiçoá-los prontamente. O fato que automóveis são máquinas complexas e relativamente recentes, bem como as tecnologias aplicadas em seus motores, faz com que haja uma enorme necessidade de estudá-los até mesmo depois de disponibilizados para o público geral.

Graças a popularização do padrão OBD, várias aplicações foram desenvolvidas para utilizá-lo para este fim. No mercado, há alguns aplicativos que oferecem o mesmo que a proposta deste trabalho. Alguns exemplos são o Torque Pro (HAWKINS, 2010), um dos mais antigos nesse gênero, e o Car Scanner ELM OBD2 (OVZ, 2018), um dos mais populares e bem avaliados. Ambos estão disponíveis apenas para o Android, mas é possível que existam outros para o iOS. A interface do Torque Pro, no entanto, está desatualizada e bastante confusa. Já o Car Scanner ELM OBD2 é mais recente e possui uma interface mais elaborada, porém ambos não são *open source* e são feitos apenas para uma plataforma.

Este padrão tem sido utilizado extensivamente não apenas no mercado, mas também no meio acadêmico. Yen et al., em seu trabalho "*Combining a Universal OBD-II Module with Deep Learning to Develop an Eco-Driving Analysis System*" (2021), mostra como é possível utilizá-lo com o auxílio de redes neurais profundas para analisar o consumo de combustível de veículos em diversas condições de rodovia.

Já Furmanczyk et al., em seu trabalho "*Integrating ODB-II, Android, and Google App Engine to Decrease Emissions and Improve Driving Habits*" (2014), propôs o uso do OBD para melhorar os hábitos de direção, bem como para diminuir as emissões de carbono.

Zaldivar et al. mostra em sua pesquisa "*Providing accident detection in vehicular networks through OBD-II devices and Android-based smartphones*" (2011) que é possível usar os sensores do veículo para detectar acidentes e *smartphones* para informar a rede do acidente e ligar para os serviços de emergência.

Ramai et al., por sua vez, propõe em "*Framework for Building Low-Cost OBD-II Data-Logging Systems for Battery Electric Vehicles*" (2022) a criação de um *framework* que coleta diversos dados referentes a bateria de um veículo elétrico. Tais informações são enviadas para um servidor web que, além de mostrá-las em um *dashboard*, também as armazena para o estudo de padrões e integração com carregadores inteligentes futuramente.

Como dito anteriormente, o trabalho apresentado por Aris et al., "*Development of OBD-II Driver Information System*" (2007), é o que funcionou como motivação para este trabalho. Nele, discute-se que, apesar do protocolo existir e fornecer bastante informação sobre o veículo, no mercado há poucas ferramentas capazes de analisar seus dados. Tais ferramentas possuem limitações que impedem o suporte pleno às suas funções, como o monitoramento dos dados em tempo real, o sistema de escaneamento do código de erro e o *logging* dessa informação. Com isso em mente, os autores propõem a criação de uma solução.

O trabalho deles, no entanto, está desatualizado e utiliza um hardware especializado para se comunicar com o OBD através da CAN. O objetivo é utilizar essa base, focada na apresentação de uma solução robusta, sem limitações, e trazê-la para a era dos *smartphones*, fazendo uso de sua versatilidade e conectividade.

Através da utilização do OBD, é possível coletar diversas informações; sendo um protocolo, ele funciona em todos os veículos, independentemente da fabricante ou do modelo. Esse fato explica sua utilização nos trabalhos mencionados. O Flutter, portanto, seria a contraparte no lado do usuário, pois se trata de um *framework* multiplataforma que pode criar aplicações para diferentes dispositivos a partir do mesmo código. Juntos, ambos oferecem uma ferramenta genérica o suficiente para agradar diversos usuários, deixando também espaço para crescimento em trabalhos futuros e justificando sua escolha para este trabalho.

Na Tabela 3.1 é possível ver uma comparação entre os trabalhos apresentados e o que foi proposto aqui.

Tabela 3.1 – Comparativo entre os trabalhos relacionados

Trabalho	**Natureza do trabalho**	**Plataforma Alvo**	**Objetivo**
Trabalho proposto	Aplicação de código aberto	Multiplataforma	Mostrar dados ao motorista
Torque Pro	Aplicação de código fechado	Android	Mostrar dados ao motorista
Car Scanner ELM OBD2	Aplicação de código fechado	Android	Mostrar dados ao motorista
Combining a Universal OBD-II Module with Deep Learning to Develop an Eco-Driving Analysis System	Pesquisa acadêmica	Plataforma própria	Analisar o consumo de combustível de veículos em diversas condições de rodovia
Integrating OBD-II, Android and Google App Engine to Decrease Emissions and Improve Driving Habits	Pesquisa acadêmica	Android	Melhorar os hábitos de direção e diminuir as emissões de carbono
Providing accident detection in vehicular networks through OBD-II devices and Android-based smartphones	Pesquisa acadêmica	Android	Usar os sensores do veículo para detectar acidentes e o *smartphone* para informar a rede do acidente, além de ligar para os serviços de emergência
Framework for Building Low-Cost OBD-II Data-Logging Systems for Battery Electric Vehicles	Pesquisa acadêmica	Plataforma própria	Criar um *framework* que coleta informações sobre a bateria do veículo e envia para um servidor onde elas podem ser exibidas ou estudadas
Development of OBD-II Driver Information System	Pesquisa acadêmica	Plataforma própria	Mostrar dados ao motorista

Fonte: Autoria própria (2023)

4 METODOLOGIA

O objetivo proposto neste trabalho é desenvolver uma aplicação que irá requisitar os dados do veículo, através de um dispositivo que, conectado a sua porta CAN, recebe pedidos e retorna os dados solicitados. A metodologia utilizada para tal será desenvolvida nas subseções a seguir.

4.1 Ferramentas

Para a realização deste trabalho foram utilizadas algumas ferramentas, tais como o Visual Studio Code, Android Studio, Flutter, Git, o GitHub e o ELM5.

O Visual Studio Code [1] é um editor de texto de código aberto. Ele é desenvolvido pela Microsoft e é feito utilizando o framework Electron. Ele está disponível para MacOS, Linux e Windows (MICROSOFT, 2023).

O Android Studio [2] é a IDE (*Integrated Development Environment*, ambiente de desenvolvimento integrado em tradução livre) oficial para o desenvolvimento de aplicações para o sistema operacional Android. Ela está disponível para Windows, MacOS e Linux (GOOGLE, 2023).

O Flutter [3], como apresentado no Capítulo 2, é um kit de desenvolvimento de interface de usuário de código aberto que utiliza a linguagem Dart. Ele possibilita a criação de aplicativos compilados nativamente para os sistemas operacionais Android, iOS, Windows, Mac, Linux, Fuchsia e para a Web (FLUTTER, 2023).

Para o controle de versão foram utilizados o git [4] e o Github [5]. O git é um programa open source de versionamento de código. Com ele é possível rastrear todas as mudanças no código e ter as datas dessas mudanças e os autores de tais mudanças (SOFTWARE FREEDOM CONSERVANCY, 2023). O Github é um site que permite a criação de repositórios git. Assim, o código desenvolvido localmente era então enviado para o servidor remoto do Github. Para que o código possa ser acessado por outros (GITHUB INC., 2023).

[1] https://code.visualstudio.com/
[2] https://developer.android.com/studio/
[3] https://flutter.dev/
[4] https://git-scm.com/
[5] https://github.com/

O ELM5 [6] é um dispositivo que engloba um microchip ELM 327, um módulo Bluetooth HC-05 e um conector SAE J1962 (CARINFO24, 2023). E é este dispositivo que é conectado a porta CAN do veículo (Figura 4.1).

Figura 4.1 – ELM5

Fonte: Autoria própria (2023)

4.2 Software

Para a requisição dos dados, a aplicação se comunicará usando Bluetooth. O módulo Bluetooth do ELM5, entretanto, utiliza uma conexão serial. Então a comunicação da aplicação com o ELM5 precisa ser serial. Sendo assim, é preciso tomar cuidado ao se criar o socket, para que ele seja serial. O pseudo-código mostrando a execução da aplicação pode ser visto a seguir.

```
Listar todos os dispositivos emparelhados com aparelho
rodando a aplicacao
```

[6] https://www.elmelectronics.com/

```
Se o ELM5 foi selecionado:
    Abrir um socket serial Bluetooth com o ELM5
    Enviar comandos de configuracao para configurar o microchip ELM 327
    A cada 1 segundo:
        Requisitar os dados do veiculo
        Se recebeu novos dados
            Atualiza o valor dos dados na tela
```

O escopo deste trabalho será limitado a coletar dados, tais como: velocidade do motor, velocidade do veículo e a quantidade de combustível no tanque. No entanto, a aplicação será desenvolvida de tal forma que, em pesquisas futuras, seja possível implementar a coleta de outros tipos de dados e de funcionalidades.

A velocidade do motor representa a magnitude da velocidade rotacional do virabrequim. Em geral, a rotação é expressa em RPM (rotação por minuto). O virabrequim é uma peça fundamental do motor de combustão interna, responsável por transformar o movimento linear dos pistões em movimento rotativo, que é propagado para a transmissão do veículo.

Expressa em quilômetro por hora (km/h), a velocidade do veículo mostra o quão rápido o veículo está e quantos quilômetros serão percorridos em uma hora caso o veículo permaneça nessa mesma velocidade.

A quantidade de combustível no tanque será apresentada utilizando uma porcentagem. Este dado mostrará quanto do tanque de combustível do veículo estará preenchido por combustível.

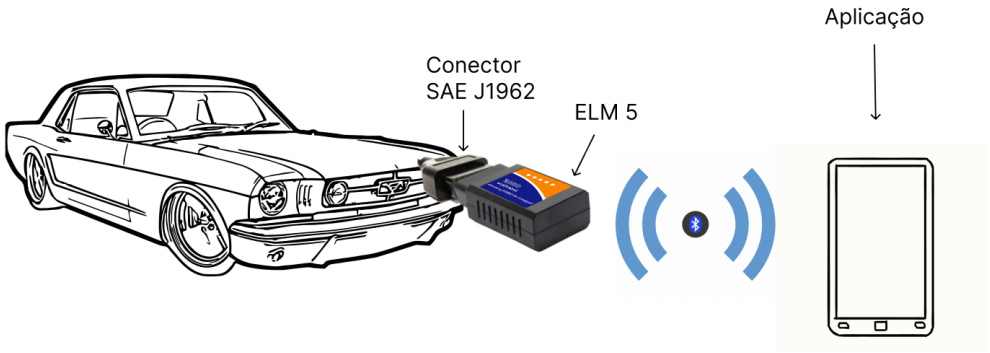
Serão utilizados três veículos para o teste deste trabalho. Uma Saveiro 2011, um Polo 2019 e um Focus 2019. O mesmo ELM5 será conectado em cada um deles e a aplicação será testada. Primeiro será verificado se a conexão é estabelecida, na sequência será conferido se os valores presentes na aplicação condizem com aqueles apresentados no painel do carro.

A Figura 4.2 mostra o fluxo dos dados. A aplicação pede o dado, esse pedido é enviado por Bluetooth, o sinal Bluetooth emitido pelo smartphone (exemplo da figura) é captado pelo módulo Bluetooth do ELM5 que, então, decodifica a mensagem e repassa para o ELM 327 que pede a informação do veículo através da porta SAE J1962. Após isso, a resposta do veículo realiza o caminho inverso.

O código da aplicação pode ser encontrado no Github [7].

[7] https://github.com/thiagoluigi7/obd_visualizer_auto

Figura 4.2 – Fluxo dos dados



Fonte: Autoria própria (2023)

5 DESENVOLVIMENTO

Neste capítulo serão definidos os requisitos funcionais e os requisitos não funcionais do trabalho. A aplicação também será apresentada, mostrando alguns protótipos de interface da mesma, terminando numa discussão sobre usabilidade.

5.1 Requisitos funcionais

Os requisitos funcionais são funções que a aplicação deve apresentar ao ser concluída. Ela pode vir a ter mais funcionalidades no futuro, mas estas são absolutamente necessárias para que se tenha uma primeira versão da aplicação. Estes requisitos estão apresentados na Tabela 5.1.

Tabela 5.1 – Requisitos funcionais

ID	Descrição
RF 01	Conectar a aplicação com o dispositivo na CAN do veículo utilizando Bluetooth
RF 02	Exibir na tela da aplicação a velocidade do motor
RF 03	Exibir na tela da aplicação a velocidade do veículo
RF 04	Exibir na tela da aplicação o combustível no tanque

Fonte: Autoria própria (2023)

5.1.1 RF 01

Para atingir este objetivo será preciso conectar a aplicação ao dispositivo na CAN do veículo. O meio de conexão escolhido foi o *Bluetooth*. Para se conectar a porta CAN do carro e começar a requisitar as informações utilizando o protocolo OBD2, a maioria dos dispositivos disponíveis utiliza o microcontrolador ELM 327. Esse microcontrolador apresenta uma interface mais simples para que seja possível se comunicar com o OBD2. Dispositivos que possuem capacidade *Bluetooth* utilizam um módulo *bluetooth* HC-05.

Esse pacote (ELM 327 + HC-05) utiliza uma comunicação serial, então para que a aplicação possa se comunicar com o ELM é preciso iniciar uma comunicação serial *bluetooth* do *smartphone* com o módulo *Bluetooth* HC-05. A aplicação abre este *socket Bluetooth* e manda alguns comandos AT que são exclusivos do ELM para configurá-lo.

Os seguintes comandos de configuração são enviados nesta ordem:

1. AT Z. Para resetar o ELM para o estado de fábrica caso ele tenha sido configurado anteriormente;

2. AT E0. Para desativar o *echo*. O *echo* é uma função na qual o ELM manda de volta o comando recebido. Pode ser utilizado no desenvolvimento para testar, porém na aplicação aqui só iria poluir o *socket* com dados desnecessários;

3. AT SP 0. O ELM suporta os vários protocolos de comunicação do OBD2. Este comando serve para informar o ELM para procurar qual é o melhor para o veículo em que ele está conectado no momento;

4. AT S0. Por padrão, o ELM possui espaço em suas respostas para facilitar a leitura, como no exemplo: '00 11 22 33' ao invés de '00112233'. Com este comando a inclusão desses espaços é desativada, pois ela adiciona uma latência para retornar a resposta. O usuário final não terá contato com esta resposta, portanto isso é irrelevante;

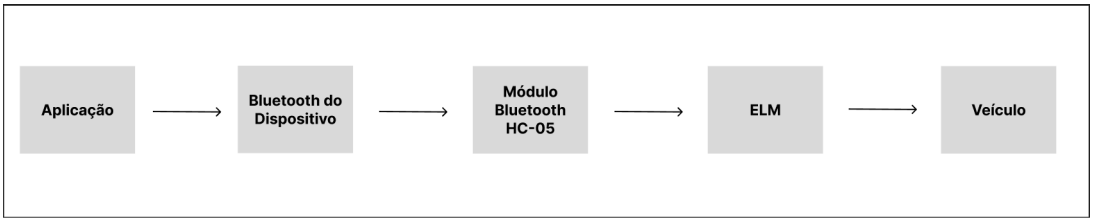
5. AT M0. O ELM possui uma memória não volátil onde ele pode armazenar o protocolo que foi selecionado. Porém, para manter a aplicação genérica para o uso em múltiplos carros, essa memória é desativada utilizando este comando.

6. AT AT 1. Este comando pede para o ELM definir um tempo de *timeout* automaticamente.

7. 01 00. Este é o primeiro comando dentre os de configuração que não são para o ELM em si, mas para o veículo. É um comando simples que apenas retorna todos os PIDs que são suportados pelo veículo. Essa informação não é levada em consideração e só é utilizada para testar a conexão do ELM com o veículo.

O fluxo de conexão é apresentado na Figura 5.1.

Figura 5.1 – Fluxo de conexão



Fonte: Autoria própria (2023)

5.1.2 RF 02

Para este requisito, a *query* utilizada está presente na Tabela 5.2.

Tabela 5.2 – Exemplo de *query* para o RF 02

Byte		
2	01	12

Fonte: Autoria própria (2023)

E a resposta é apresentada na Tabela 5.3

Tabela 5.3 – Exemplo de resposta para o RF 02

Byte				
4	41h	12	A	B

Fonte: Autoria própria (2023)

Para converter este retorno em um valor real, é preciso utilizar o valor de A e de B na seguinte fórmula $\frac{256A+B}{4}$.

O resultado será um valor entre 0 e 16.383,75 que representará a quantidade de rotações que o motor está tendo em um minuto.

5.1.3 RF 03

Para este requisito, a *query* utilizada está presente na Tabela 5.4.

Tabela 5.4 – Exemplo de *query* para o RF 03

Byte		
1	01	13

Fonte: Autoria própria (2023)

E a resposta está na Tabela 5.5

Tabela 5.5 – Exemplo de resposta para o RF 03

Byte			
3	41h	13	A

Fonte: Autoria própria (2023)

O valor de A já representa a velocidade do carro em Km/h.

5.1.4 RF 04

Para este requisito, a *query* utilizada está presente na Tabela 5.6.

Tabela 5.6 – Exemplo de *query* para o RF 04

Byte		
1	01	47

Fonte: Autoria própria (2023)

E a resposta está na Tabela 5.7

Tabela 5.7 – Exemplo de resposta para o RF 04

Byte			
3	41h	47	A

Fonte: Autoria própria (2023)

Para converter o valor de A para a porcentagem que representa a quantidade de combustível no tanque, basta utilizar a seguinte fórmula: $\frac{100}{255}A$

5.2 Requisitos não funcionais

Os requisitos não funcionais são aqueles requisitos que não necessariamente são funcionalidades de código, mas que precisam estar presentes. Eles são apresentados na Tabela 5.8

Tabela 5.8 – Requisitos não funcionais

ID	**Descrição**
RNF 01	A interface precisa ser flexível para funcionar em diversos tamanhos de tela diferentes
RNF 02	O aplicativo precisa ser desenvolvido utilizando o framework Flutter
RNF 03	O aplicativo deve aceitar comandos de toque

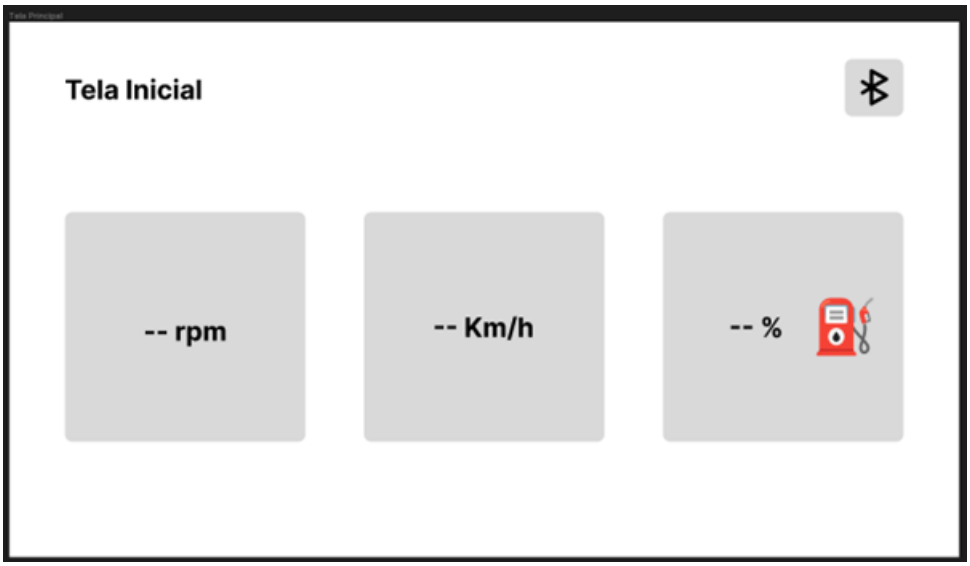
Fonte: Autoria própria (2023)

5.3 Apresentação

A aplicação será utilizada como um painel de carro onde as informações serão apresentadas. Sendo assim, é necessário que ela funcione primariamente na orientação horizontal (paisagem).

A Figura 5.2 apresenta o design inicial da aplicação. Se trata da tela inicial, composta por *widgets* de dados, também chamados de “mostradores”. Cada *widget* representa uma informação. Como para este trabalho somente a velocidade do motor, a velocidade do veículo e o combustível no tanque serão utilizados, apenas três *widgets* serão exibidos na tela. No canto superior direito, há um botão para navegar para a tela de configurar a conexão da aplicação com o adaptador na porta CAN do veículo.

Figura 5.2 – Tela inicial



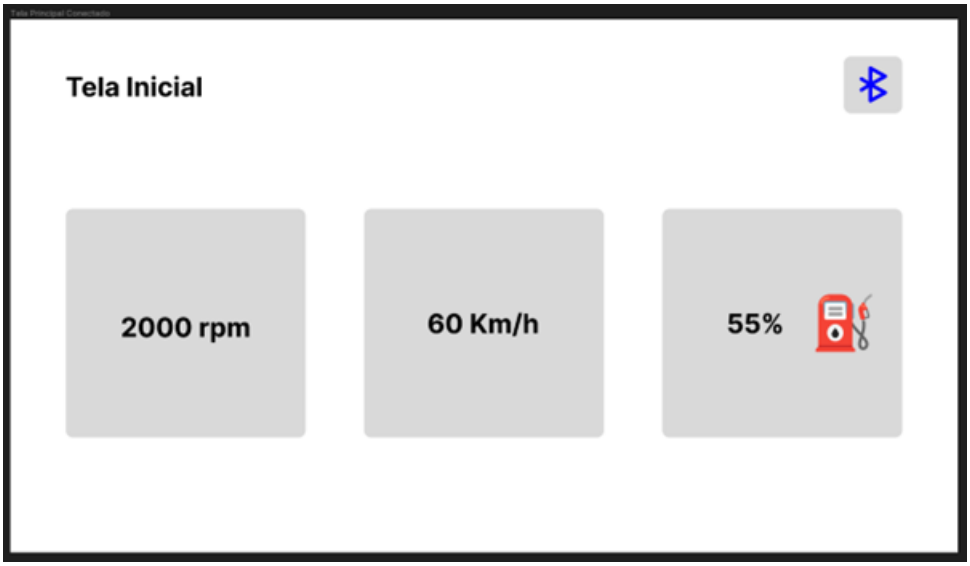
Fonte: Autoria própria (2023)

Já a Figura 5.3 mostra a tela inicial da aplicação quando a conexão já foi estabelecida. O botão para navegar para a tela de configurar a conexão apresenta um *feedback* visual que mostra que a aplicação já se encontra conectada ao adaptador na CAN. Os dados também passam a aparecer nos *widgets*.

Por sua vez, a Figura 5.4 mostra a tela de configurar a conexão. A tela apresenta quatro botões. Dentre esses botões, os três principais usam o *widget* base, mas serão clicáveis. O botão “Conectar” irá abrir a tela de conexão *Bluetooth* do aparelho, e o botão "Desconectar"irá cortar a conexão *Bluetooth* da aplicação.

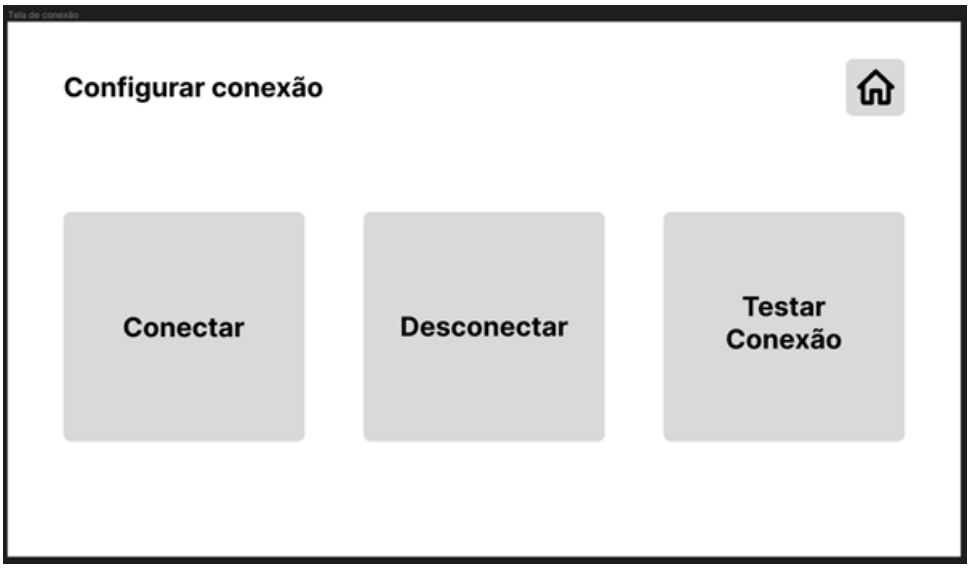
O botão “Testar Conexão” será utilizado para testar a conexão entre a aplicação e o adaptador na porta CAN do veículo. Internamente, uma *query* simples será realizada, e caso um resultado válido retorne, um *feedback* visual será mostrado no botão. A cor do botão ficará verde por um breve período de tempo caso o teste tenha passado, como visto na Figura 5.5. Do contrário, o botão ficará vermelho, como demonstrado pela Figura 5.6.

Figura 5.3 – Tela inicial - conectado



Fonte: Autoria própria (2023)

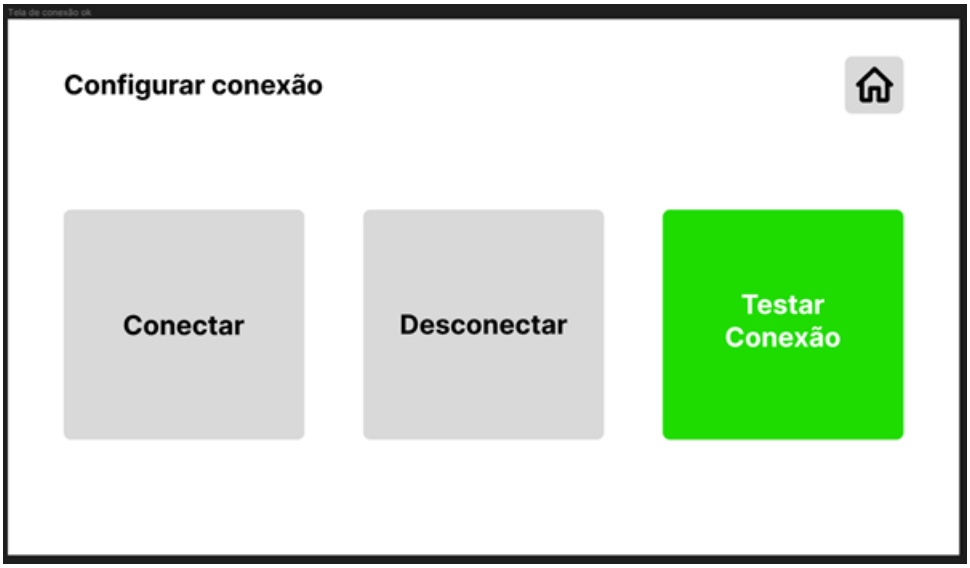
Figura 5.4 – Tela de conexão



Fonte: Autoria própria (2023)

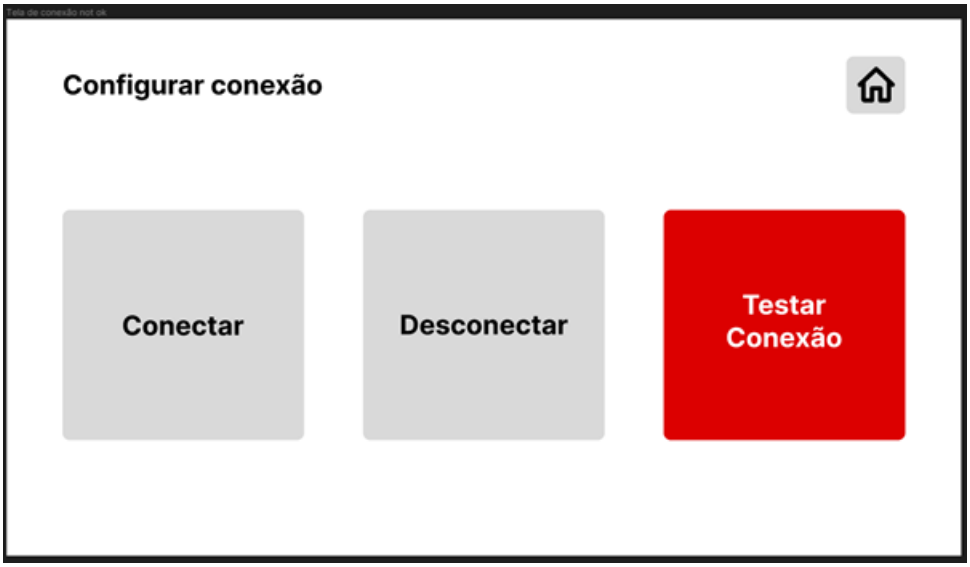
O último, por sua vez, é o botão *home*, que não utiliza o *widget* base. Ao clicar nele, o usuário será levado para a tela inicial da aplicação.

Figura 5.5 – Tela de conexão - botão de testar conexão



Fonte: Autoria própria (2023)

Figura 5.6 – Tela de conexão - botão de testar conexão falhou



Fonte: Autoria própria (2023)

5.4 Usabilidade

A usabilidade está ligada ao tempo necessário para realizar uma tarefa, à eficiência para sua realização e ao índice de erros neste processo. Sendo assim, essa aplicação, levando em conta que será utilizada em um veículo, precisa atender a alguns requisitos:

- não pode distrair o motorista: a segurança no trânsito é prioridade, portanto, a aplicação não pode demandar a atenção do motorista. Ela deve funcionar sem a intervenção do

motorista e não pode requisitar a atenção do mesmo por meio de notificações ou sons. Ela pode possuir alertas, desde que o motorista os tenha programado anteriormente. Estes alertas não podem ser súbitos e também não podem ser visuais, isto é, podem mostrar conteúdo na tela, mas não podem depender desse conteúdo para transmitir a informação. O conteúdo na tela também não pode ser chamativo, pois isso poderia distrair a atenção do motorista;

- precisa ser fácil de se enxergar: a informação precisa ter um tamanho significativo para que possa ser vista e compreendida rapidamente, sem demandar esforço visual do motorista. Para tal, os componentes da interface foram projetados para ser grandes. Além disso, a tela terá pouco conteúdo e quase nenhuma animação, para que o motorista possa olhar de relance e identificar a informação que deseja sem se confundir com efeitos de fundo;

- precisa ser simples de se navegar: a aplicação será composta basicamente de duas telas, a Tela Inicial e a tela Configurar Conexão. A navegação entre essas telas será feita através de um botão simples, localizado no canto superior direito da interface. Esse botão, apesar de bem visível, não fica na frente de outras informações, pois a ideia é que ele não precise ser utilizado com frequência;

- precisa ser amigável: o objetivo da aplicação é mostrar informações ao motorista, e o ideal é que ele tenha o que precisa sem a necessidade de tocar constantemente no painel. Entretanto, caso ele queira ver diversas informações diferentes e todos esses mostradores não caibam na tela, é importante que seja fácil alternar entre os mostradores disponíveis. Para isso, a interface usa um esquema de grade semelhante à tela de aplicativos dos *smartphones*, onde cada mostrador é como um aplicativo no celular. Isso é um aspecto positivo, considerando que os *smartphones* já estão bem difundidos na sociedade e que o conceito de grade de aplicações já é bem aceito e validado. Assim, ao olhar para tela, o usuário perceberá de imediato que cada mostrador se refere a uma informação, que se trata de um “app”.

Para mover entre os mostradores disponíveis, é possível deslizar a tela para os lados, assim como em um *smartphone*. Sendo assim, a tela inicial terá páginas de mostradores. Essas páginas terão um número fixo de mostradores, e a navegação entre as páginas também será fixa. Isso significa que quando uma pessoa deslizar para passar para a próxima página, imediatamente

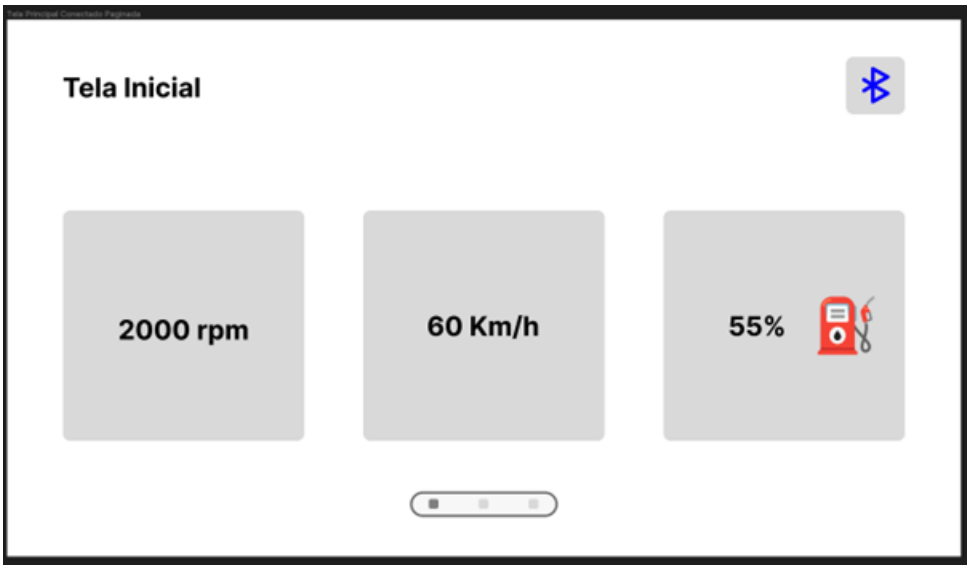
uma animação para trocar de página será exibida, não sendo possível passar só um pouco da mesma.

Esta estrutura de distribuição dos mostradores faz com que um motorista que usa diversos mostradores possa acessar o que ele deseja deslizando horizontalmente sem tirar os olhos da estrada. Por exemplo, se o mostrador que ele quer está na terceira página e ele sabe disso, basta que deslize o dedo na tela duas vezes e a informação será exibida. Não é necessário que fique olhando para a tela para ver se a informação necessária já passou ou não.

Com estes requisitos, a utilização da aplicação se torna segura, simples e prática. O motorista não precisará aprender algo totalmente novo, podendo trazer um conhecimento próprio de mundo (nesse caso, o uso de um *smartphone*) para a aplicação. A localização de seus mostradores é facilmente memorizável, tornando possível a navegação segura entre suas páginas, além de ter a segurança de conseguir a informação que deseja sem precisar tirar os olhos da estrada por longos períodos de tempo.

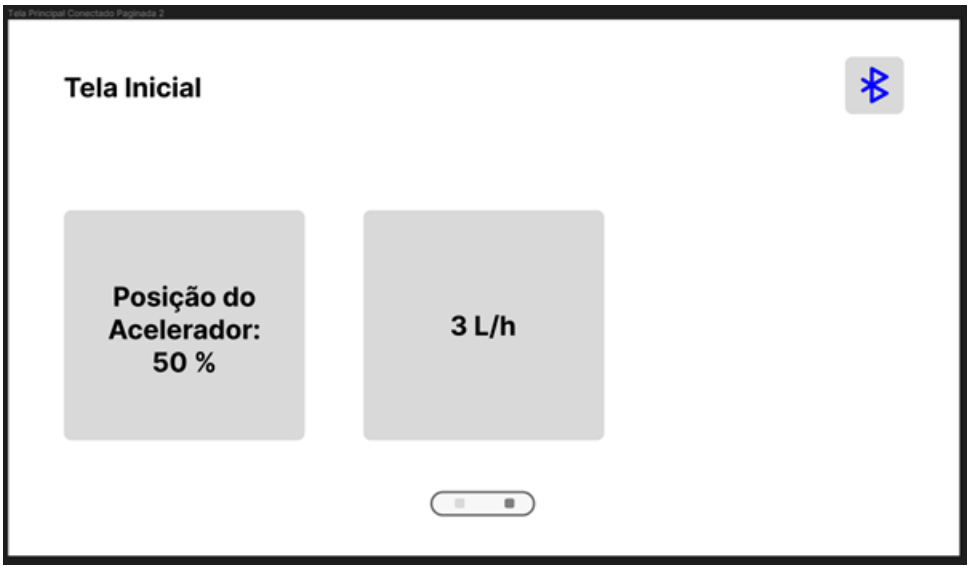
Uma parte do que foi tratado no quesito de usabilidade não será implementado neste trabalho, como por exemplo a estrutura de distribuição dos mostradores, uma vez que, para a presente aplicação, apenas três serão desenvolvidos. No entanto, tais fatores demonstram o quão extensível a interface é. Abaixo, as Figuras 5.7, 5.8 e 5.9 demonstram exemplos de possíveis casos de uso onde a interface mostra sua escalabilidade.

Figura 5.7 – Tela inicial com mais elementos



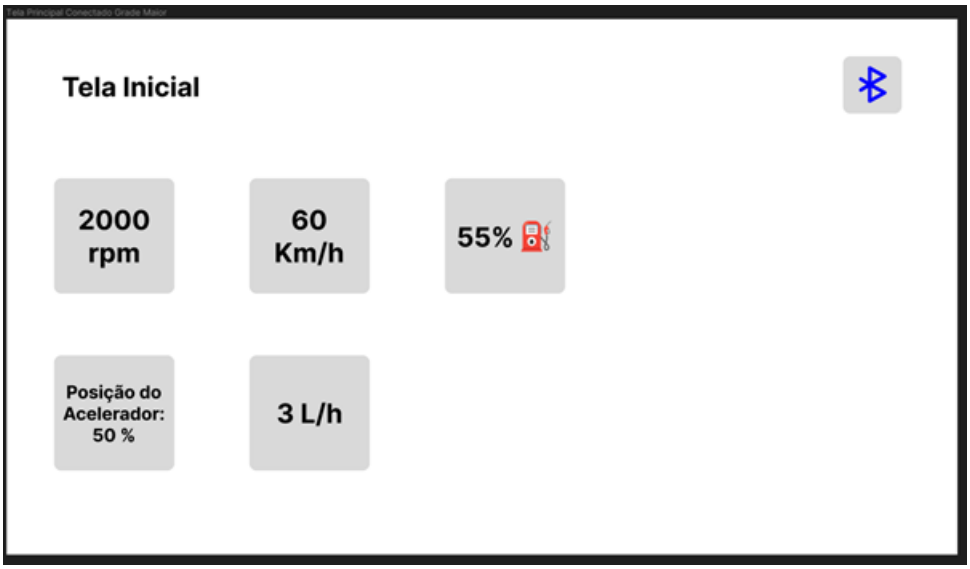
Fonte: Autoria própria (2023)

Figura 5.8 – Tela inicial na segunda página de elementos



Fonte: Autoria própria (2023)

Figura 5.9 – Tela inicial com mais elementos por página



Fonte: Autoria própria (2023)

6 RESULTADOS

A aplicação foi desenvolvida com sucesso, atendendo aos requisitos funcionais e não funcionais citados no Capítulo 5. Durante seu desenvolvimento, ela foi testada em três veículos: um Volkswagen Saveiro 2011, um Volkswagen Polo MPI 2019 e um Ford Focus 2019. Na Saveiro, a aplicação consegue se conectar ao ELM e configurá-lo, porém o ELM não consegue encontrar um protocolo de comunicação compatível com o veículo. Já no Polo e no Focus, a conexão e a comunicação foram realizadas sem problemas.

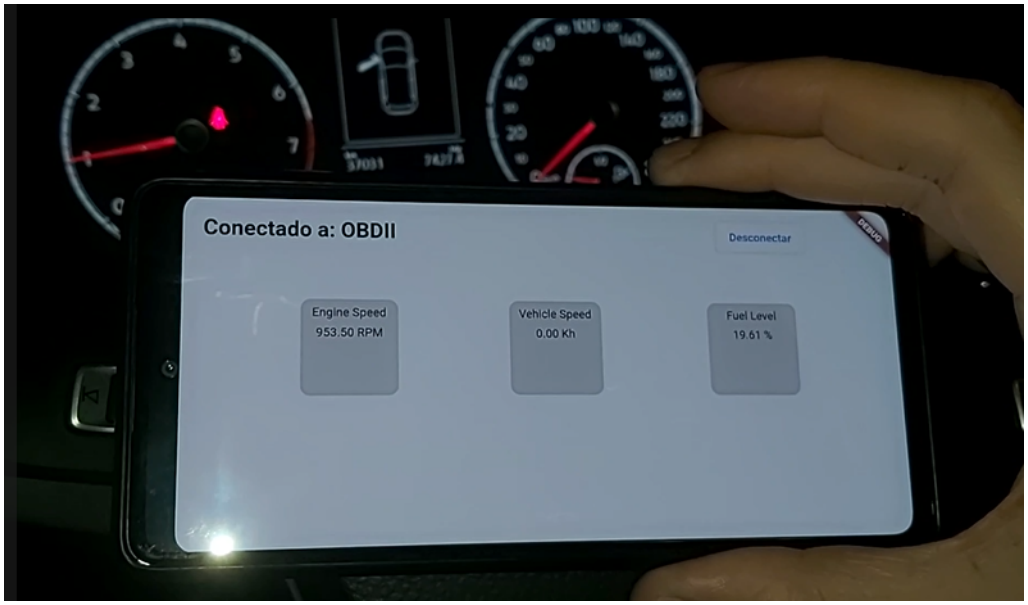
Na versão final da aplicação, a interface foi simplificada e não há a página de configuração de conexão. Ao invés de ter esta tela, ao abrir a aplicação, caso ela não esteja conectada a nenhum dispositivo *Bluetooth*, a aplicação irá mostrar a lista de dispositivos pareados. Uma vez que o usuário escolher o dispositivo na lista e tocá-lo, a lista de dispositivos pareados irá sumir. Em segundo plano, os comandos para configurar o ELM serão executados e os três mostradores (velocidade do veículo, do motor e quantidade de combustível) serão exibidos como no protótipo. No canto superior esquerdo da tela, ao invés do nome da página, considerando que a aplicação atual possui apenas uma, mostra-se a qual dispositivo a aplicação está conectada.

Na Figura 6.1 é possível ver o ELM5 conectado a porta SAE J1962 do Polo. E nas Figuras 6.2 e 6.3 é possível ver os dados exibidos na interface, já atualizada, e compará-los aos dados do painel. Porém, o que não é possível ver nas figuras, pois no momento da captura o carro estava parado, é a diferença dentre a velocidade apresentada pela aplicação e pelo painel do carro.

Figura 6.1 – ELM5 conectado ao Polo

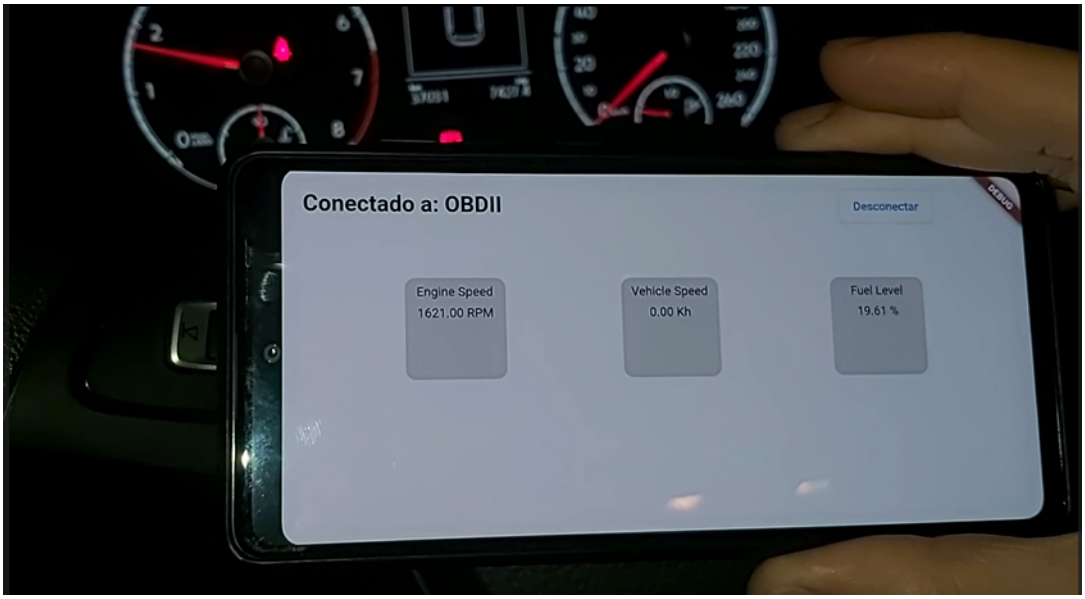
Fonte: Autoria própria (2023)

Figura 6.2 – Teste da aplicação final no Polo



Fonte: Autoria própria (2023)

Figura 6.3 – Teste da aplicação final no Polo



Fonte: Autoria própria (2023)

Nos testes realizados no Ford Focus, a velocidade mostrada pela aplicação era cerca de 5 KM/H menor do que aquela apresentada no painel do carro. Entretanto, quando o aplicativo Waze foi colocado no meio para comparação, foi possível ver que a velocidade apresentada pela aplicação e pelo Waze são bastante parecidas, sendo a velocidade do Waze também diferente daquela apresentada no painel.

Por fim, constatou-se que o ELM sempre retorna uma resposta quando uma requisição é feita, e no caso do carro estar com o *bus* ocupado, o ELM retorna '*NODATA*' como reposta. A aplicação foi programada para pedir ao veículo novas informações a cada 1 segundo para evitar tais retornos, pois os mesmos atrapalham o tratamento dos dados realizado pela aplicação.

O código da aplicação pode ser encontrado no Github [1].

[1] https://github.com/thiagoluigi7/obd_visualizer_auto

7 CONCLUSÃO

Tendo em vista os resultados descritos no Capítulo anterior, o desenvolvimento da aplicação foi bem sucedido. A interface permite que os motoristas utilizem a aplicação com facilidade como previsto, o que é um aspecto positivo no quesito usabilidade.

Já no que diz respeito aos testes, verificou-se que o protocolo de comunicação não é encontrado na Saveiro. Nos outros dois veículos utilizados, o Polo e o Focus, a conexão foi bem sucedida, e dados semelhantes aos exibidos no painel dos carros foram recebidos. Em relação aos padrões, viu-se que, mesmo sendo vastamente adotados em diversos países, ainda é possível encontrar veículos em que eles não estão implementados.

É pertinente considerar que trabalhos futuros podem modificar a aplicação de tal forma que seja possível atualizar mais rapidamente as informações, considerando o problema encontrado nos Resultados relacionado à taxa de atualização. Também é possível que a aplicação seja expandida para que possa funcionar utilizando WIFI ou USB como meio de conexão além do *Bluetooth*. Devido à natureza do *Bluetooth*, e como ele funciona de maneira diferente em cada plataforma, neste trabalho sua implementação só foi realizada no Android, mas trabalhos futuros podem expandir a aplicação adicionando a implementação do *bluetooth* para a conexão em outro sistema operacional.

Aqui é importante frisar que, apesar da aplicação ter sido desenvolvida apenas para o Android, o uso do Flutter ainda é um diferencial. Isso se deve ao fato de que, para que haja o desenvolvimento para outra plataforma, só seria necessário adicionar a parte da conexão *bluetooth*, dispensando o desenvolvimento da interface e outras funções que a aplicação venha a ter futuramente.

Também é possível que a interface da aplicação seja trabalhada de tal forma que haja uma tela de configurações para reordenar os mostradores, adicionar mostradores, além de conter opções de *logging*.

REFERÊNCIAS

AL-SULTAN, S. et al. A comprehensive survey on vehicular Ad Hoc network. **Journal of Network and Computer Applications**, v. 37, p. 380–392, jan. 2014. ISSN 1084-8045. Disponível em: <https://www.sciencedirect.com/science/article/pii/S108480451300074X>. Acesso em: 27 de fev. de 2023.

ALMEIDA, P. H. et al. RODOGE: Protocolo de disseminação de mensagens de alerta de acidentes com regras e controles de reenvio. In: **Anais do Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC)**. SBC, 2019. p. 266–279. ISSN 2177-9384. Disponível em: <https://sol.sbc.org.br/index.php/sbrc/article/view/7365>. Acesso em: 27 de fev. de 2023.

ARENA, F.; PAU, G. An Overview of Vehicular Communications. **Future Internet**, Multidisciplinary Digital Publishing Institute, v. 11, n. 2, p. 27, fev. 2019. ISSN 1999-5903. Disponível em: <https://www.mdpi.com/1999-5903/11/2/27>. Acesso em: 27 de fev. de 2023.

ARIS, I. et al. DEVELOPMENT OF OBD-II DRIVER INFORMATION SYSTEM. v. 4, n. 5, p. 253–259, 2007. Disponível em: <http://ijet.feiic.org/journals/J-2007-V2014.pdf>. Acesso em: 27 de fev. de 2023.

CAN in Automation. **History of CAN technology**. 2023. Disponível em: <https://www.can-cia.org/can-knowledge/can/can-history/>. Acesso em: 27 de fev. de 2023.

CANALTECH. **Governo americano quer padronizar comunicação entre veículos para salvar vidas**. 2014. Disponível em: <https://arquivo.canaltech.com.br/carros/Governo-americano-quer-padronizar-comunicacao-entre-veiculos-para-salvar-vidas/>. Acesso em: 27 de fev. de 2023.

CARINFO24. **What is ELM327 Scanner and How to Use It**. 2023. Disponível em: <https://carinfo24.com/blog/2020-01-08-what-is-elm327-scanner-and-how-to-use-it/>. Acesso em: 27 de fev. de 2023.

CHEN, S.-H.; PAN, J.-S.; LU, K. Driving Behavior Analysis Based on Vehicle OBD Information and AdaBoost Algorithms. **Proceedings of the International MultiConference of Engineers and Computer Scientists**, v. 1, p. 102–106, mar. 2015. Disponível em: <https://www.iaeng.org/publication/IMECS2015/IMECS2015_pp102-106.pdf>. Acesso em: 27 de fev. de 2023.

CHEN, Y.-L.; SHEN, K.-Y.; WANG, S.-C. Forward collision warning system considering both time-to-collision and safety braking distance. In: **2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA)**. [s.n.], 2013. p. 972–977. ISSN 2158-2297. ISSN: 2158-2297. Disponível em: <https://ieeexplore.ieee.org/document/6566508>. Acesso em: 27 de fev. de 2023.

CSS Electronics. **OBD2 Explained - A simple intro [2023]**. 2023. Disponível em: <https://www.csselectronics.com/pages/obd2-explained-simple-intro>. Acesso em: 27 de fev. de 2023.

DART. **Dart**. 2023. Disponível em: <https://dart.dev/>. Acesso em: 27 de fev. de 2023.

ELM Electronics. **OBD**. 2023. Disponível em: <https://www.elmelectronics.com/products/ics/obd/>. Acesso em: 27 de fev. de 2023.

FLUTTER. **Flutter**. 2023. Disponível em: <https://flutter.dev/>. Acesso em: 27 de fev. de 2023.

FURMANCZYK, C. et al. Integrating ODB-II, Android, and Google App Engine to Decrease Emissions and Improve Driving Habits. dez. 2014. Disponível em: <https://core.ac.uk/display/100228416>. Acesso em: 27 de fev. de 2023.

GALVANI, M. History and future of driver assistance. **IEEE Instrumentation & Measurement Magazine**, v. 22, n. 1, p. 11–16, fev. 2019. ISSN 1941-0123.

GIETELINK, O. et al. Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations. **Vehicle System Dynamics**, Taylor & Francis, v. 44, n. 7, p. 569–590, jul. 2006. ISSN 0042-3114. Disponível em: <https://doi.org/10.1080/00423110600563338>. Acesso em: 27 de fev. de 2023.

GITHUB INC. **GitHub**. 2023. Disponível em: <https://github.com/>. Acesso em: 27 de fev. de 2023.

GOOGLE. **Android Studio**. 2023. Disponível em: <https://developer.android.com/studio>. Acesso em: 27 de fev. de 2023.

HAWKINS, I. **Torque Pro (OBD2 / Carro)**. 2010. Versão 1.12.100. Milton Keynes. Software. Disponível em: <https://play.google.com/store/apps/details?id=org.prowl.torque&gl=US>. Acesso em: 27 de fev. de 2023.

HERMAWAN, G.; HUSNI, E. Acquisition, Modeling, and Evaluating Method of Driving Behavior Based on OBD-II: A Literature Survey. **IOP Conference Series: Materials Science and Engineering**, IOP Publishing, v. 879, n. 1, p. 012030, jul. 2020. ISSN 1757-899X. Disponível em: <https://dx.doi.org/10.1088/1757-899X/879/1/012030>. Acesso em: 27 de fev. de 2023.

MAHAJAN, G.; S.K.PARCHANDEKAR, M.; TAHIR, M. M. Implementation and validation of k line (iso 9141) protocol for diagnostic application. **International Research Journal of Engineering and Technology**, v. 04, n. 07, p. 708–713, jul. 2017. ISSN 2395-0056. Disponível em: <https://www.irjet.net/archives/V4/i7/IRJET-V4I7181.pdf>. Acesso em: 27 de fev. de 2023.

MICROSOFT. **Visual Studio Code**. 2023. Disponível em: <https://visualstudio.microsoft.com/downloads/>. Acesso em: 27 de fev. de 2023.

MILLER, T. **ELM327: What Is The Best Genuine ELM327 Adapters 2023 (Bluetooth/Wi-fi/USB)?** 2023. Disponível em: <https://www.obdadvisor.com/elm327/>. Acesso em: 27 de fev. de 2023.

MONIAGA, J. V. et al. Diagnostics vehicle's condition using obd-ii and raspberry pi technology: study literature. **Journal of Physics: Conference Series**, IOP Publishing, v. 978, n. 1, p. 012011, mar. 2018. ISSN 1742-6596. Disponível em: <https://dx.doi.org/10.1088/1742-6596/978/1/012011>. Acesso em: 27 de fev. de 2023.

NETO, J. B. P. et al. An error correction algorithm for forward collision warning applications. In: **2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)**. [S.l.: s.n.], 2016. p. 1926–1931. ISSN 2153-0017. ISSN: 2153-0017. Acesso em: 27 de fev. de 2023.

OBDTESTER.COM. **OBD-II Protocols**. 2020. Disponível em: <http://www.obdtester.com/obd2_protocols#:~:text=OBD-II>. Acesso em: 27 de fev. de 2023.

OVZ. **Car Scanner ELM OBD2**. 2018. Versão 1.98.6. 2018. Software. Disponível em: <https://play.google.com/store/apps/details?id=com.ovz.carscanner>. Acesso em: 27 de fev. de 2023.

PREEZ, M. **OBD stands for on-board diagnostics and defines the modern fuel managed vehicles electronic interface system**. 2022. Pinout Guide. Disponível em: <https://pinoutguide.com/CarElectronics/car_obd2_pinout.shtml>. Acesso em: 27 de fev. de 2023.

PöGEL, T.; WOLF, L. Prediction of 3G network characteristics for adaptive vehicular Connectivity Maps (Poster). In: **2012 IEEE Vehicular Networking Conference (VNC)**. [S.l.: s.n.], 2012. p. 121–128. ISSN 2157-9865. ISSN: 2157-9865.

QI, W. et al. SDN-Enabled Social-Aware Clustering in 5G-VANET Systems. **IEEE Access**, v. 6, p. 28213–28224, 2018. ISSN 2169-3536.

RAMAI, C. et al. Framework for Building Low-Cost OBD-II Data-Logging Systems for Battery Electric Vehicles. **Vehicles**, Multidisciplinary Digital Publishing Institute, v. 4, n. 4, p. 1209–1222, dez. 2022. ISSN 2624-8921. Disponível em: <https://www.mdpi.com/2624-8921/4/4/64>. Acesso em: 27 de fev. de 2023.

SHAOUT, A.; COLELLA, D.; AWAD, S. Advanced Driver Assistance Systems - Past, present and future. In: **2011 Seventh International Computer Engineering Conference (ICENCO'2011)**. [S.l.: s.n.], 2011. p. 72–82.

SOFTWARE FREEDOM CONSERVANCY. **Git**. 2023. Disponível em: <https://git-scm.com/>. Acesso em: 27 de fev. de 2023.

TEAGUE, C. **Everything you need to know about V2X Technology**. 2021. Autoweek. Disponível em: <https://www.autoweek.com/news/technology/a36190311/v2x-technology/>. Acesso em: 27 de fev. de 2023.

TECHTARGET. **Vehicle to infraestructure (V2I or v2i)**. 2017. Boston. Disponível em: <https://www.techtarget.com/whatis/definition/vehicle-to-infrastructure-V2I-or-V2X#:~:text=Vehicle-to-infrastructure(V2I,streetlights,signageandparkingmeters>. Acesso em: 27 de fev. de 2023.

TEHRANI, M. N.; UYSAL, M.; YANIKOMEROGLU, H. Device-to-device communication in 5G cellular networks: challenges, solutions, and future directions. **IEEE Communications Magazine**, v. 52, n. 5, p. 86–92, maio 2014. ISSN 1558-1896.

WHO TEAM. **Road Traffic Injuries - World Health Organization (WHO)**. 2022. Disponível em: <https://www.who.int/news-room/fact-sheets/detail/road-traffic-injuries>. Acesso em: 27 de fev. de 2023.

WINNER, H. et al. **Handbook of Driver Assistance Systems: Basic Information, Components and Systems for Active Safety and Comfort**. [S.l.]: Springer International Publishing, 2015. Google-Books-ID: FeHVoQEACAAJ. ISBN 9783319123516.

YEN, M.-H. et al. Combining a Universal OBD-II Module with Deep Learning to Develop an Eco-Driving Analysis System. **Applied Sciences**, Multidisciplinary Digital Publishing Institute, v. 11, n. 10, p. 4481, jan. 2021. ISSN 2076-3417. Disponível em: <https://www.mdpi.com/2076-3417/11/10/4481>. Acesso em: 27 de fev. de 2023.

ZALDIVAR, J. et al. Providing accident detection in vehicular networks through OBD-II devices and Android-based smartphones. In: **2011 IEEE 36th Conference on Local Computer Networks**. [S.l.: s.n.], 2011. p. 813–819. ISSN 0742-1303. ISSN: 0742-1303.

A APÊNDICES

A.1 Main

```dart
import 'package:flutter/material.dart';
import 'package:obd_visualizer_auto/pages/home.dart';

void main() {
  runApp(const MyApp());
}
```

A.2 HomePage

```dart
import 'dart:async';
import 'dart:convert';

import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:obd2_plugin/obd2_plugin.dart';
import 'package:flutter_bluetooth_serial/flutter_bluetooth_serial.dart';
import 'package:obd_visualizer_auto/utils/utils.dart';
import 'package:obd_visualizer_auto/widgets/device_list.widget.dart';
import 'package:permission_handler/permission_handler.dart';

import '../utils/globals.dart';
import '../widgets/base.widget.dart';

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    SystemChrome.setPreferredOrientations([
      DeviceOrientation.landscapeLeft,
```

```
      DeviceOrientation.landscapeRight,
    ]);
    SystemChrome.setEnabledSystemUIMode(SystemUiMode.immersiveSticky);
    return MaterialApp(
      title: 'OBD Visualizer',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: MyHomePage(title: 'OBD Visualizer'),
    );
  }
}

class MyHomePage extends StatefulWidget {
  MyHomePage({super.key, required this.title});

  final String title;
  final Obd2Plugin obd2 = Obd2Plugin();

  @override
  State<MyHomePage> createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  List<BluetoothDevice> devices = [];
  late Timer? timer;

  Future<void> _getDevices() async {
    if (await Permission.bluetoothScan.request().isGranted &&
        await Permission.bluetoothConnect.request().isGranted) {
      List<BluetoothDevice> list = await widget.obd2.getPairedDevices;
      _setList(list);
```

```
  }
}

void _setList(List<BluetoothDevice> list) {
  setState(() => devices = list);
}

Future<void> _setAutoFetchData() async {
  timer = Timer.periodic(const Duration(seconds: 1), (_) async {
    if (Globals.configured.value) {
      await _getData();
    }
  });
}

void _setResponseLogic(command, response, requestCode) {
  debugPrint('$command => $response');
  if (command == 'PARAMETER') {
    var parsedResponses = jsonDecode(response);
    for (final returnedResponse in parsedResponses) {
      String response = returnedResponse['response'] == ''
          ? '-- ${returnedResponse['unit']}'
          : '${double.parse(returnedResponse['response']).toStringAsFixed(2)} $
      if (returnedResponse['PID'] == '01 0C') {
        Globals.engineSpeed.value = response;
      } else if (returnedResponse['PID'] == '01 0D') {
        Globals.vehicleSpeed.value = response;
      } else if (returnedResponse['PID'] == '01 2F') {
        Globals.fuelLevel.value = response;
      }
    }
  }
```

```
}

Future<void> _getConnection(BluetoothDevice device) async {
  try {
    Globals.device.value = device;

    await widget.obd2.getConnection(device, (connection) {
      debugPrint('Device ${device.name} selected');
    }, (message) {
      debugPrint('Error: $message');
    });

    await widget.obd2.setOnDataReceived((command, response, requestCode) {
      _setResponseLogic(command, response, requestCode);
    });

    if (await widget.obd2.isListenToDataInitialed) {
      await Future.delayed(Duration(
          milliseconds:
              await widget.obd2.configObdWithJSON(Globals.configJson)));

      Globals.configured.value = true;

      return;
    }

    debugPrint('Not initialized');
  } catch (e) {
    Globals.configured.value = false;
    debugPrint('Error: $e');
    showSnackBar(context, 'Erro na conexão');
  }
```

```
}

Future<void> _disconnect() async {
  Globals.device.value = null;
  Globals.configured.value = false;
  await widget.obd2.disconnect();
}

Future<void> _getData() async {
  try {
    debugPrint('Fetching data...');
    await Future.delayed(Duration(
        milliseconds:
            await widget.obd2.getParamsFromJSON(Globals.paramJson)));
  } catch (e) {
    debugPrint('Error fetching data: $e');
  }
}

@override
void initState() {
  super.initState();
  _getDevices();
  _setAutoFetchData();
}

@override
void dispose() {
  super.dispose();
  timer?.cancel();
}
```

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Center(
      child: Column(
        mainAxisSize: MainAxisSize.max,
        children: <Widget>[
          const SizedBox(height: 10),
          Row(
              mainAxisAlignment: MainAxisAlignment.start,
              crossAxisAlignment: CrossAxisAlignment.start,
              children: <Widget>[
                Row(children: <Widget>[
                  const SizedBox(width: 20),
                  ValueListenableBuilder<BluetoothDevice?>(
                      valueListenable: Globals.device,
                      builder: (context, value, Widget? child) {
                        return Text(
                            Globals.device.value != null
                                ? 'Conectado a: ${Globals.device.value!.name}'
                                : 'Não conectado',
                            style: const TextStyle(
                                fontSize: 25.0, fontWeight: FontWeight.bold));
                      }),
                  const SizedBox(width: 350),
                  ValueListenableBuilder<BluetoothDevice?>(
                      valueListenable: Globals.device,
                      builder: (context, value, Widget? child) {
                        return Visibility(
                          visible: Globals.device.value != null,
                          child: OutlinedButton(
                            onPressed: _disconnect,
```

```
                    child: const Text('Desconectar'),
                  ),
                );
              }),
        ]),
      ]),
const SizedBox(height: 50),
ValueListenableBuilder<BluetoothDevice?>(
    valueListenable: Globals.device,
    builder: (context, value, Widget? child) {
      return Visibility(
        visible: Globals.device.value != null,
        child: Row(
            crossAxisAlignment: CrossAxisAlignment.center,
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: [
              ValueListenableBuilder<String>(
                  valueListenable: Globals.engineSpeed,
                  builder: (context, value, Widget? child) {
                    return BaseWidget(
                      name: 'Engine Speed',
                      parsedValue: value,
                    );
                  }),
              ValueListenableBuilder<String>(
                  valueListenable: Globals.vehicleSpeed,
                  builder: (context, value, Widget? child) {
                    return BaseWidget(
                      name: 'Vehicle Speed',
                      parsedValue: value,
                    );
                  }),
```

```
                        ValueListenableBuilder<String>(
                            valueListenable: Globals.fuelLevel,
                            builder: (context, value, Widget? child) {
                              return BaseWidget(
                                name: 'Fuel Level',
                                parsedValue: value,
                              );
                            })
                        ]),
                  );
                }),
            ValueListenableBuilder<BluetoothDevice?>(
                valueListenable: Globals.device,
                builder: (context, value, Widget? child) {
                  return Visibility(
                      visible: Globals.device.value == null,
                      child: DeviceList(
                          devices: devices, getConnection: _getConnection));
                }),
          ],
        ),
      ),
    );
  }
}
```

A.3 Classes Utilitárias

A.3.1 Globals

```
import 'package:flutter/material.dart';
import 'package:flutter_bluetooth_serial/flutter_bluetooth_serial.dart';
```

```
class Globals {
  static ValueNotifier<BluetoothDevice?> device = ValueNotifier<BluetoothDevice?>(r

  static ValueNotifier<bool> configured = ValueNotifier<bool>(false);

  static ValueNotifier<String> engineSpeed = ValueNotifier<String>('');
  static ValueNotifier<String> vehicleSpeed = ValueNotifier<String>('');
  static ValueNotifier<String> fuelLevel = ValueNotifier<String>('');

  static const String configJson = '''
        [
          {
              "command": "AT Z",
              "description": "",
              "status": true
          },
          {
              "command": "AT E0",
              "description": "",
              "status": true
          },
          {
              "command": "AT SP 0",
              "description": "",
              "status": true
          },
          {
              "command": "AT S0",
              "description": "",
              "status": true
          },
```

```
        {
            "command": "AT M0",
            "description": "",
            "status": true
        },
        {
            "command": "AT AT 1",
            "description": "",
            "status": true
        },
        {
            "command": "01 00",
            "description": "",
            "status": true
        }
    ]''';

static const String paramJson = '''
    [
        {
            "PID": "01 0C",
            "length": 2,
            "title": "Engine Speed",
            "unit": "RPM",
            "description": "<double>, (( [0] * 256) + [1] ) / 4",
            "status": true
        },
        {
            "PID": "01 0D",
            "length": 1,
            "title": "Vehicle Speed",
            "unit": "Kh",
```

```
            "description": "<int>, [0]",
            "status": true
        },
        {
            "PID": "01 2F",
            "length": 1,
            "title": "Fuel Level",
            "unit": "%",
            "description": "<int>, ( 100 / 255 ) * [0]",
            "status": true
        }
    ]
    ''';
}
```

A.3.2 Utils

```
import 'package:flutter/material.dart';

void showSnackBar(BuildContext context, String message) {
  final scaffold = ScaffoldMessenger.of(context);
  scaffold.showSnackBar(
    SnackBar(
      content: Text(message),
      action: SnackBarAction(
          label: 'UNDO', onPressed: scaffold.hideCurrentSnackBar),
    ),
  );
}
```

A.4 Widgets

A.4.1 Base

```
import 'package:flutter/material.dart';

class BaseWidget extends StatefulWidget {
  const BaseWidget({
    super.key,
    required this.name,
    required this.parsedValue,
  });

  final String name;
  final String parsedValue;

  @override
  BaseWidgetState createState() => BaseWidgetState();
}

class BaseWidgetState extends State<BaseWidget> {
  @override
  Widget build(BuildContext context) {
    return SizedBox(
      width: 100,
      height: 100,
      child: Stack(alignment: Alignment.center, children: <Widget>[
        Positioned(
            top: 0,
            left: 0,
            child: Container(
                width: 100,
                height: 100,
```

```
                decoration: const BoxDecoration(
                    borderRadius: BorderRadius.only(
                      topLeft: Radius.circular(10),
                      topRight: Radius.circular(10),
                      bottomLeft: Radius.circular(10),
                      bottomRight: Radius.circular(10),
                    ),
                    color: Color.fromRGBO(217, 217, 217, 1))),
          Column(children: [
            const SizedBox(height: 6),
            Text(widget.name),
            const SizedBox(height: 6),
            Text(widget.parsedValue),
            // const SizedBox(height: 10),
          ])
        ]),
      );
  }
}
```

A.4.2 DeviceList

```
import 'package:flutter/material.dart';
import 'package:flutter_bluetooth_serial/flutter_bluetooth_serial.dart';
import 'package:obd_visualizer_auto/widgets/device_list_item.widget.dart';

class DeviceList extends StatefulWidget {
  const DeviceList({
    super.key,
    required this.devices,
    required this.getConnection,
  });
```

```
  final List<BluetoothDevice> devices;
  final Function getConnection;

  @override
  DeviceListState createState() => DeviceListState();
}

class DeviceListState extends State<DeviceList> {
  @override
  Widget build(BuildContext context) {
    return Container(
      margin: EdgeInsets.all(2),
      height: 150.0,
      width: 700.0,
      child: ListView.builder(
          padding: const EdgeInsets.all(2),
          itemCount: widget.devices.length,
          scrollDirection: Axis.horizontal,
          shrinkWrap: true,
          itemBuilder: (BuildContext context, int index) {
            return Container(
                margin: const EdgeInsets.only(top: 2, bottom: 2),
                padding: const EdgeInsets.all(15),
                child: DeviceListItem(
                    name: widget.devices[index].name,
                    address: widget.devices[index].address,
                    device: widget.devices[index],
                    getConnection: widget.getConnection));
          }),
    );
  }
}
```

A.4.3 DeviceListItem

```
import 'package:flutter/material.dart';
import 'package:flutter_bluetooth_serial/flutter_bluetooth_serial.dart';

class DeviceListItem extends StatefulWidget {
  DeviceListItem({
    super.key,
    required this.name,
    required this.address,
    required this.device,
    required this.getConnection,
  });

  String? name;
  String? address;
  BluetoothDevice device;
  Function getConnection;

  @override
  DeviceListItemState createState() => DeviceListItemState();
}

class DeviceListItemState extends State<DeviceListItem> {
  @override
  Widget build(BuildContext context) {
    return OutlinedButton(
      onPressed: () {
        widget.getConnection(widget.device);
      },
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
```

```
            Text('Nome: ${widget.name}'),
            const SizedBox(height: 10),
            Text('Endereço: ${widget.address}'),
          ],
        ),
      );
    }
}
```