UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

CURSO DE GRADUAÇÃO EM ENGENHARIA DE COMPUTAÇÃO

SAMUEL HENRIQUE GUIMARÃES ALENCAR

DESENVOLVIMENTO DE UM SISTEMA ANDROID DE INFORMAÇÃO E ENTRETENIMENTO PARA VEÍCULOS: VISUALIZAÇÃO DE DEFEITOS VEICULARES

QUIXADÁ

2024

SAMUEL HENRIQUE GUIMARÃES ALENCAR

DESENVOLVIMENTO DE UM SISTEMA ANDROID DE INFORMAÇÃO E ENTRETENIMENTO PARA VEÍCULOS: VISUALIZAÇÃO DE DEFEITOS VEICULARES

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Orientador: Prof. Dr. Francisco Helder Candido dos Santos Filho.

QUIXADÁ

2024

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Sistema de Bibliotecas
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

A355d Alencar, Samuel Henrique Guimarães.
Desenvolvimento de um sistema Android de informação e entretenimento para veículos: visualização de defeitos veiculares / Samuel Henrique Guimarães Alencar. – 2024.
113 f. : il. color.

Trabalho de Conclusão de Curso (graduação) – Universidade Federal do Ceará, Campus de Quixadá, Curso de Engenharia de Computação, Quixadá, 2024.
Orientação: Prof. Dr. Francisco Helder Candido dos Santos Filho.

1. Defeitos veiculares. 2. Controller Area Network (CAN). 3. Android Automotive OS (AAOS). 4. Raspberry Pi 4B. 5. Vehicle Hardware Abstract Layer (VHAL). I. Título.

CDD 621.39

SAMUEL HENRIQUE GUIMARÃES ALENCAR

DESENVOLVIMENTO DE UM SISTEMA ANDROID DE INFORMAÇÃO E ENTRETENIMENTO PARA VEÍCULOS: VISUALIZAÇÃO DE DEFEITOS VEICULARES

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia de Computação do Campus Quixadá da Universidade Federal do Ceará, como requisito parcial à obtenção do grau de bacharel em Engenharia de Computação.

Aprovada em: 19/09/2024.

BANCA EXAMINADORA

Prof. Dr. Francisco Helder Candido dos Santos Filho (Orientador)
Universidade Federal do Ceará (UFC)

Prof. Dra. Andréia Libório Sampaio
Universidade Federal do Ceará (UFC)

Prof. Dr. Elvis Miguel Galeas Stancanelli
Universidade Federal do Ceará (UFC)

Prof. Dr. Wagner Guimarães Al-Alam
Universidade Federal do Ceará (UFC)

Dedico este trabalho aos meus pais e avós, que nunca mediram esforços para garantir a minha educação e formação. Todo esforço é por vocês.

AGRADECIMENTOS

Primeiramente, gostaria de agradecer a Deus por toda a força e resiliência para seguir este caminho. Aos meus pais, Josélia Lima e Carlos Henrique, por apoiarem minhas escolhas, por serem minha base de amor, dedicação, força, e por seus incontáveis esforços, que me permitiram chegar até aqui. Aos meus avós, por todo o apoio para continuar nessa jornada. À minha tia, Eloisa Nogueira, por me tratar como um filho. À minha namorada, Larissa Matos, por ter me acompanhado nos meus dois últimos anos de graduação, me proporcionando infinitos momentos de felicidade, apoio, carinho e amor.

Aos meus amigos Ana Clara, Lucas Emanuel, Juliana Brígido, Sara Sena, Ana Júlia, Anne Louine, Iasmin Maia e Ana Rosa, por todo o companheirismo, amor, carinho, e pelos incontáveis momentos de diversão e choro que passamos juntos. A vida foi bondosa ao cruzar nossos caminhos, e estarei sempre aqui por vocês.

Aos meus amigos, praticamente irmãos, Henrique Veneranda, José Maia, Raul Alex e Wesley Bezerra, por todos os momentos de felicidade que passamos juntos, pelas noites de conversas nas calçadas, pelas partidas de futebol assistidas e disputadas. À minha amiga Débora Letícia, por ser uma fonte de apoio e conselhos primorosos, pelas conversas leves e sérias que tivemos, e pelo auxílio em assuntos acadêmicos.

Aos meus amigos, Arthur Kaleb, Césary Matheus, Daniel Martins, Gustavo Bezerra, Levi Saraiva e Renan Evangelista, por todo o companheirismo, churrascos, *gameplays* e momentos de risadas que me acompanham desde o ensino médio.

Aos meus amigos e primeiros colegas de apartamento, Daniel Araújo (*in memoriam*) e Aluizio Lucas, pelos momentos de riso e auxílio em programação.

Ao meu orientador, Francisco Helder, pela paciência e pelos conselhos no âmbito acadêmico e profissional. Aos meus amigos de faculdade, em especial Anderson Silva, Pedro Botelho e José Batista, pelo companheirismo e colaboração em nossos trabalhos em equipe, proporcionando infinitos momentos de felicidade e carinho.

Aos meus amigos de laboratório, Antonio Cesar, Erick Silva, Gabriel Moreira, Abdul-Hamid, Elias Filho, Imario Almeida, Pedro Odillon e Pedro Ítalo. E aos amigos de curso: Ítalo Moraes, David Machado, Paula Feitosa, Petrucio Filho, Rafael Gonçalves e David Melo.

Ao Instituto de Pesquisas Eldorado, pela oportunidade de estágio e pela confiança no meu trabalho. Aos amigos de trabalho, Robert Cabral, Levy Galvão e Álison Venâncio, pela paciência e pelos ensinamentos sobre *Android Automotive*.

“Sim, minha força está na solidão. Não tenho medo nem de chuvas tempestivas nem das grandes ventanias soltas, pois eu também sou o escuro da noite.” (Clarice Lispector)

RESUMO

Diante dos avanços tecnológicos, os veículos estão se tornando cada vez mais modernos e inteligentes, integrando sistemas que melhoram a segurança de condutores e passageiros. Entretanto, a comunicação de defeitos por meio de códigos e luzes ainda permanece ultrapassada e confusa, podendo gerar consequências graves em situações críticas. Nesse trabalho é apresentado o estudo de uma nova abordagem para a comunicação de defeitos veiculares utilizando o *Android Automotive* OS (AAOS) e o barramento *Controller Area Network* (CAN), ao contrário de soluções baseadas em *Android Mobile* e *On-Board Diagnostics II* (OBD-II). Essa abordagem consiste na construção de uma rede veicular com o barramento CAN para transmissão de dados, enquanto o AAOS, executado em uma *Raspberry Pi* 4B, simula um veículo conectado ao barramento, processando as informações e exibindo os defeitos de forma clara e eficaz. Um serviço no AAOS captura o tráfego da rede CAN e preenche a Camada de Abstração de *Hardware* do Veículo (VHAL), permitindo que a aplicação de infoentretenimento exiba os defeitos ao usuário de forma clara e eficaz. Os experimentos mostraram a comunicação entre dois nós na rede CAN, com a exibição em tempo real das informações e defeitos no aplicativo. Um teste de usabilidade realizado com potenciais usuários, indicou que a nova abordagem é eficaz e menos confusa na comunicação de defeitos veiculares do que os métodos tradicionais.

Palavras-chave: Defeitos veiculares; *Controller Area Network* (CAN); *Android Automotive OS* (AAOS); *Raspberry Pi* 4B; *Vehicle Hardware Abstract Layer* (VHAL).

ABSTRACT

Given the technological advances, vehicles are becoming increasingly modern and intelligent, integrating systems that enhance the safety of drivers and passengers. However, the communication of defects through codes and lights remains outdated and confusing, which can lead to serious consequences in critical situations. This work presents a study of a new approach to vehicle defect communication using Android Automotive OS (AAOS) and the Controller Area Network (CAN) bus, as opposed to solutions based on Android Mobile and On-Board Diagnostics II (OBD-II). This approach involves building a vehicular network with the CAN bus for data transmission, while the AAOS, running on a Raspberry Pi 4B, simulates a vehicle connected to the bus, processing information and displaying defects clearly and effectively. A service in the AAOS captures the CAN network traffic and populates the Vehicle Hardware Abstraction Layer (VHAL), allowing the infotainment application to display defects to the user in a clear and effective way. Experiments demonstrated communication between two nodes on the CAN network, with real-time display of information and defects on the application. A usability test conducted with potential users indicated that the new approach is more effective and less confusing in communicating vehicle defects than traditional methods.

Keywords: Vehicle defects; Controller Area Network (CAN); Android Automotive OS (AAOS); Raspberry Pi 4B; Vehicle Hardware Abstraction Layer (VHAL).

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE QUADROS

LISTA DE CÓDIGOS-FONTE

LISTA DE ABREVIATURAS E SIGLAS

AAOS	*Android Automotive OS*
ABS	*Anti-Lock Brake System*
ACK	*Acknowledgement*
ADAS	*Advanced Driver Assistance Systems*
adb	*Android Debug Bridge*
AIDL	*Android Interface Definition Language*
API	*Application Programming Interface*
ARM	*Advanced RISC Machine*
CAN	*Controller Area Network*
CARB	*California Air Resources Board*
CNH	Carteira Nacional de Habilitação
CPU	*Central Processing Unit*
CRC	*Cyclic Redundancy Check*
DLC	*Data Length Code*
DTC	*Diagnostic Trouble Code*
ECU	*Electronic Control Unit*
EOF	*End of Frame*
EV	*Electric Vehicle*
GNU	*GNU's not Unix*
GPIO	*General Purpose Input/Output*
GPS	*Global Positioning System*
HAL	*Hardware Abstraction Layer*
HDMI	*High-Definition Multimedia Interface*
I2C	*Inter-Integrated Circuit*
ID	Identificador
IDE	*Identifier Extension*
IFS	*Interframe Space*
IHC	Interação Humano-Computador
IMU	*Inertial Measurement Unit*
IoT	*Internet of Things*

ISO	*International Standardization Organization*
IVI	*In-Vehicle Infotainment*
MAC	Método de Avaliação de Comunicabilidade
MCU	*Microcontroller Unit*
OBD	*On-Board Diagnostic*
OSI	*Open Systems Interconnection*
PCM	*Powertrain Control Module*
RTR	*Remote Transmission Request*
SBC	*Single Board Computer*
SDRAM	*Synchronous Dynamic Random Access Memory*
SOF	*Start of Frame*
SPI	*Serial Peripheral Interface*
SRR	*Substitute Remote Request*
UART	*Universal Asynchronous Receiver/Transmitter*
UI	*User Interface*
UI/UX	*User Interface / User Experience*
USB	*Universal Serial Bus*
VHAL	*Vehicle Hardware Abstraction Layer*

LISTA DE SÍMBOLOS

GB	*Gigabyte*
GHz	Gigahertz
Z_o	Impedância (120Ω)
$kbps$	Kilobits por segundo
KB	Kilobyte
$Mbps$	Megabits por segundo
MHz	Megahertz
μs	Microssegundos
Ω	Ohm
V	Tensão

SUMÁRIO

1 INTRODUÇÃO

Diante dos avanços tecnológicos na área automotiva, os veículos estão se tornando cada vez mais modernos e inteligentes à medida que se transformam em plataformas de computação móvel. Com isso, a integração de sistemas torna-se essencial para melhorar a experiência e a interação dos motoristas e passageiros, além de facilitar a funcionalidade interna dos veículos (Jeong *et al.*, 2023; Kim *et al.*, 2014).

Nos veículos modernos, sistemas avançados desempenham um papel crucial na segurança dos condutores e passageiros. O sistema *Advanced Driver Assistance Systems* (ADAS), por exemplo, utiliza sensores e câmeras para melhorar a segurança nas estradas (Omerovic *et al.*, 2016). Além disso, os sistemas de *In-Vehicle Infotainment* (IVI) proporcionam uma interação rica entre o usuário e o veículo (Jeong *et al.*, 2023; Kim *et al.*, 2014). Esses sistemas têm a capacidade de monitorar, gerenciar e diagnosticar veículos por meio de redes veiculares (Jeong *et al.*, 2023).

Nesse contexto, surge o *Android Automotive OS* (AAOS)[1], desenvolvido pela Google e de código aberto, que oferece uma plataforma flexível e poderosa para aprimorar a interação entre as pessoas e os sistemas presentes no veículo. Com ele, é possível vincular o *hardware* do veículo ao *software Android* através de *Hardware Abstraction Layer*s (HALs), em português Camadas de Abstração de *Hardware*, específicas para veículos, tirando o máximo proveito da eletrônica automotiva (Moiz; Alalfi, 2022). Além disso, veículos como o Polestar – um carro elétrico da Volvo – já incorporam o sistema AAOS em sua infraestrutura (Polestar, 2022; Pajic; Bjelica, 2018).

À medida que a quantidade de *Electronic Control Unit* (ECU), em português Unidade de Controle Eletrônico, e componentes eletrônicos no setor automotivo moderno aumenta, a confiabilidade do automóvel é impactada pela complexidade da tecnologia implantada (Kumar; S.Sivaji, 2015). Alguns veículos possuem cerca de 70 ECUs e 2.500 sinais internos, transmitidos por redes veiculares, formando uma espécie de sistema nervoso do veículo, permitindo a comunicação entre ECUs (Zeng *et al.*, 2016).

Os automóveis utilizam o barramento CAN, rede veicular mais utilizada atualmente (Iseke, 2020; Zeng *et al.*, 2016), para conectar todos os ECUs em um sistema unificado ponto a ponto (Kumar; S.Sivaji, 2015). Desenvolvido por Robert Bosch em 1986, o CAN é um protocolo de comunicação serial que permite a conexão de controladores, sensores e atuadores por meio de

[1] Documentação do *Android* Automotivo. Acesso em: 10 set. 2023.

dois fios, CAN *High* e CAN *Low* (Khorsravinia *et al.*, 2017; Semiconductor, 2023; Corrigan, 2016).

Apesar dos avanços tecnológicos na área automotiva, um aspecto crucial da experiência do motorista permanece ultrapassado e muitas vezes frustrante: a comunicação de falhas e defeitos veiculares (Kim *et al.*, 2014). Enquanto os veículos modernos estão repletos de sensores sofisticados, ECUs e sistemas de infoentretenimento de última geração, os defeitos ainda são comunicados de maneira rudimentar e pouco esclarecedora aos motoristas e até para alguns mecânicos (Zeng *et al.*, 2016; Pajic; Bjelica, 2018).

Afinal, a comunicação feita através de códigos ou luzes de aviso enigmáticas contribui para a frustração e desorientação do condutor sobre as ações apropriadas a serem tomadas (Motors, 2018). Esse abismo entre a tecnologia embarcada nos veículos e a interpretação das mensagens de erro cria um sério problema de segurança. Em situações críticas, como problemas no sistema de freios ou no motor, a falta de informações claras e imediatas pode ter consequências potencialmente perigosas.

Em virtude disso, novos sistemas de informação e entretenimento estão surgindo para complementar os já existentes. As ideias vão desde o uso do barramento e protocolo CAN para capturar dados e fornecer informações do veículo ao usuário (Park; Lee, 2012), até um sistema de infoentretenimento automotivo que possui funções, como monitoramento e autodiagnóstico, que apoiam a segurança dos motoristas (Kim *et al.*, 2014).

Mesmo com toda a tecnologia envolvida, frequentemente os veículos modernos comunicam esses defeitos de maneira confusa, deixando os condutores desorientados e frustrados.

Este trabalho propõe um estudo sobre a implementação de um sistema de informação e entretenimento baseado em *Android Automotive OS* (AAOS) com o objetivo de apresentar defeitos veiculares a partir da rede CAN e comunicá-los de maneira clara e eficaz, tanto para os motoristas quanto para os mecânicos. Essa implementação deve ajudar a evitar confusões sobre o significado dos defeitos. O estudo proposto é dividido em duas partes. Na primeira, um microcontrolador conectado ao barramento CAN transmite mensagens com dados coletados a partir de sensores. A segunda parte do estudo envolve um sistema embarcado, conectado ao barramento CAN, que executa o *Android Automotive* (AAOS), acompanhado de um sistema de infoentretenimento capaz de apresentar os defeitos do veículo ao usuário. Além disso, o sistema embarcado desempenha a função de simular tanto um veículo quanto uma ECU.

1.1 Objetivos

1.1.1 Objetivo geral

Desenvolver e avaliar um sistema de informação e entretenimento baseado no *Android Automotive OS* (AAOS), capaz de apresentar os defeitos veiculares de forma clara e eficaz para motoristas e mecânicos.

1.1.2 Objetivos específicos

1. Implementar um sistema de *hardware* embarcado que simulará um veículo e uma ECU, capaz de receber dados através da rede CAN e executar *kernel Linux* e o AAOS.
2. Desenvolver um sistema de informação e entretenimento capaz de exibir os defeitos e dados do veículo, utilizando o AAOS.
3. Realizar testes e simulações para avaliar o sistema desenvolvido na comunicação de defeitos veiculares.

1.1.3 Organização do trabalho

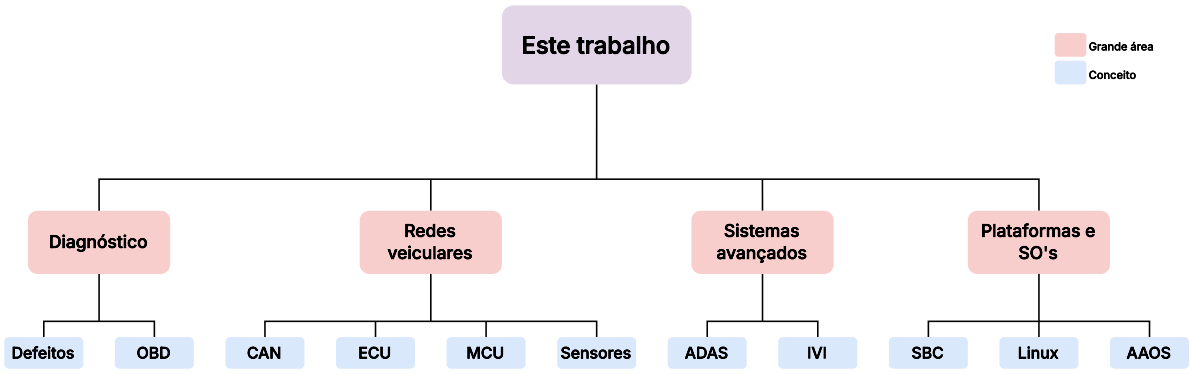
O restante deste trabalho está organizado da seguinte forma:

- Capítulo 2: descreve os conceitos fundamentais para a compreensão do trabalho proposto.
- Capítulo 3: explora os trabalhos relacionados e compara os aspectos em comum e divergentes em relação ao presente estudo.
- Capítulo 4: descreve a metodologia utilizada que deve ser abordada para o desenvolvimento deste trabalho.
- Capítulo 5: apresenta os resultados obtidos.
- Capítulo 6: descreve a conclusão deste trabalho, dificuldades encontradas e ideias para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo fundamenta os principais conceitos relacionados ao trabalho, trazendo a colaboração de cada conceito para o trabalho proposto.

Figura 1 – Relação entre os conceitos abordados neste trabalho



Fonte: Elaborado pelo autor (2024).

A Figura 1 mostra a organização dos conceitos que formam este trabalho, destacando quatro grandes áreas principais. As áreas são representadas na cor salmão, enquanto os conceitos discutidos e explicados nas seções abaixo estão em azul. Este trabalho explora essas áreas inter-relacionadas, oferecendo uma visão geral e tendências futuras no setor automotivo.

2.1 Defeitos veiculares

De acordo com a International Organization for Standardization (2017), defeito é: "imperfeição ou deficiência em um produto final, onde esse produto não atende aos seus requisitos ou especificações e precisa ser reparado ou substituído". Os defeitos podem se manifestar dentro de um sistema embarcado automotivo, sendo geralmente associados ao mau funcionamento de algum *hardware*, como sensores, placas microcontroladoras ou microprocessadas. Esses problemas podem ser atribuídos a várias causas, como problemas na alimentação ou conexão, ou até mesmo peças danificadas (Goelles *et al.*, 2020).

Atualmente, com o auxílio do sistema *On-Board Diagnostic* (OBD), tornou-se viável identificar e classificar os defeitos que ocorrem nos veículos, simplificando assim o processo de reparo e permitindo a rápida identificação. Este trabalho abordará defeitos que ocorrem tanto na aquisição de dados a partir do sensor, como na comunicação entre o sensor e o controlador dentro da rede CAN.

2.2 *On-Board Diagnostic* (OBD)

OBD, em português diagnóstico de bordo, é um sistema baseado em microprocessador que é responsável por realizar autodiagnóstico e relatar o desempenho dos sistemas e componentes do veículo que estão em mau funcionamento (Denton, 2020).

A origem do sistema OBD se dá diante do esforço de alguns governos do mundo, por meio de legislações, na redução e controle da emissão de gases poluentes por veículos. A Lei do Ar Limpo da Califórnia foi assinada e o *California Air Resources Board* (CARB) adotou regulamentos que exigiam que todos os carros do ano 1994 em diante fossem equipados com sistemas OBD. A tarefa desses sistemas se resumia em monitorar o desempenho dos sistemas de controle de emissões dos veículos e alertar os motoristas quando houver um mau funcionamento de um sistema/subsistema ou componente de controle de emissões (Denton, 2020).

Quando um problema é detectado, o sistema OBD é responsável por acender uma luz de advertência no *cluster* (painel) do veículo. No Brasil, essa luz é popularmente conhecida como "símbolo (ou luz) da injeção eletrônica", embora também possa ser exibida por meio de uma frase em inglês pedindo para verificar o problema (Denton, 2020).

Quando um defeito ou problema ocorre, o sistema armazena um *Diagnostic Trouble Code* (DTC), ou código de diagnóstico de problemas, que pode ser usado para rastrear e identificar o defeito e é formado por cinco caracteres alfanuméricos. Dessa forma, um mecânico é capaz de conectar uma ferramenta de verificação de diagnóstico que se comunicará com o microprocessador e recuperará essas informações. Isto permite o diagnóstico e a correção do problema, realizando uma substituição ou reparo do componente defeituoso, e após isso o reinício do sistema OBD e a limpeza dos DTCs são realizados (Denton, 2020).

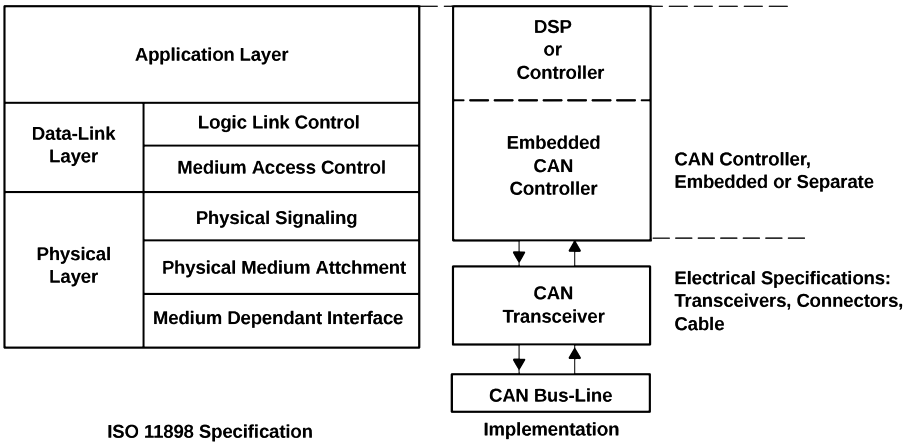
No início do sistema OBD, as normas não eram bem definidas. Os fabricantes desenvolviam e aplicavam seus próprios sistemas de forma livre, resultando em diferentes construções para cada veículo. Isso gerava dificuldades para as oficinas não autorizadas, que precisavam lidar com uma variedade de manuais, cabos e conectores. Esse cenário tornava o diagnóstico complexo e custoso. Para resolver esse problema, foram estabelecidas normas que incluíam a padronização do conector, uma ferramenta de varredura comum e um protocolo de comunicação compatível com veículos de todos os fabricantes, e assim surgiu o OBD-II. O conector OBD-II fornece interface com a rede veicular do veículo, como a rede CAN, permitindo que qualquer pessoa tenha acesso aos dados do seu automóvel (Denton, 2020).

2.3 *Controller Area Network* (CAN)

CAN é um barramento de comunicação serial definido pela *International Standardization Organization* (ISO), originalmente desenvolvido para a indústria automotiva, substituindo o complexo chicote por um barramento de dois fios. A especificação exige alta imunidade a interferências elétricas, capacidade de autodiagnóstico e correção de erros de dados. Esses recursos levaram à popularidade do CAN (Corrigan, 2016; INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2015).

O protocolo de comunicação CAN, de acordo com a ISO 11898[1], descreve como as informações são passadas entre dispositivos em uma rede e está em conformidade com o modelo *Open Systems Interconnection* (OSI), que é definido em termos de camadas. A comunicação real entre dispositivos conectados pelo meio físico é definida pela camada física do modelo, mostrada na Figura 2 (Corrigan, 2016).

Figura 2 – A arquitetura padrão ISO 11898 em camadas



Fonte: (Corrigan, 2016).

A arquitetura ISO 11898 define as duas camadas mais baixas do modelo OSI/ISO de sete camadas como a camada de enlace e a camada física na Figura 2 (Corrigan, 2016). Além disso, o CAN é assíncrono e multi-mestre comunicando-se com ECUs, sensores e atuadores em aplicações automotivas (Semiconductor, 2023). Por sua vez, o barramento CAN consiste em dois fios (CAN *High* e CAN *Low*) (Corrigan, 2016).

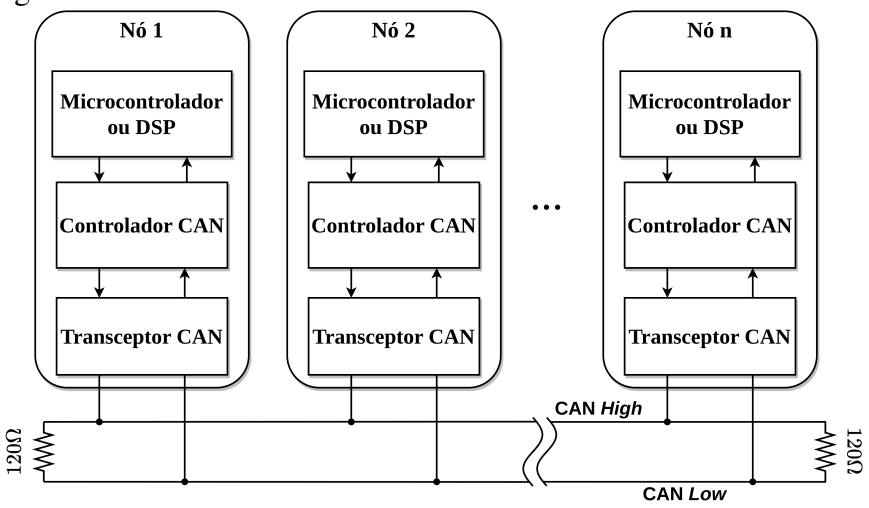
A Figura 3 ilustra o barramento CAN e exemplifica o conceito de um nó na rede CAN, que de acordo com a International Organization for Standardization (2015, p. 12) é "um conjunto, ligado a uma rede de comunicação, capaz de se comunicar através da rede de acordo

[1] ISO 11898. Acesso em: 13 set. 2023

com uma especificação de protocolo de comunicação".

De acordo com o padrão, o cabo é especificado como um par trançado, podendo ser blindado ou não, com uma impedância característica de 120 Ω (Z_o), buscando prevenir reflexões de sinal (Corrigan, 2016).

Figura 3 – Linha do barramento CAN



Fonte: Adaptado de (Corrigan, 2016).

Um sistema CAN envia mensagens usando o barramento serial como uma rede, visto que qualquer nó pode enviar uma mensagem para qualquer outro nó, devido a sua característica multi-mestre. Mesmo se algum falhar, os outros continuarão a funcionar corretamente e se comunicarão entre si. Qualquer nó na rede que deseja transmitir uma mensagem aguardará até que o barramento esteja livre. Cada mensagem possui um identificador, e todas as mensagens estão disponíveis para todos os outros nós na rede. O nó seleciona as mensagens relevantes e ignora as demais (Semiconductor, 2023).

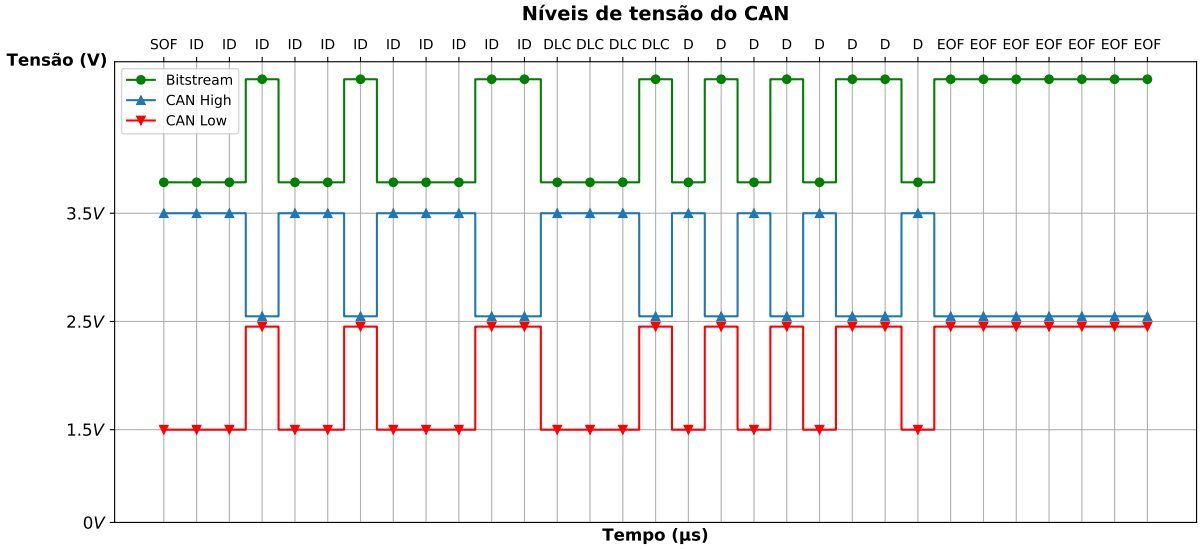
Além disso, o CAN foi projetado para oferecer alta velocidade e latência extremamente baixa, com taxas de transmissão que variam de 125 *kbps* a 1 *Mbps*, e proporciona uma comunicação em sinal diferencial, notado pelo formato da sua onda. Esses aspectos tornam o CAN uma escolha essencial em aplicações críticas (Corrigan, 2016).

2.3.1 Formas de onda

O CAN é conhecido por sua robustez e é amplamente utilizado devido às suas diversas características de confiabilidade. Além disso, utiliza uma abordagem de sinal diferencial, e o sinal em uma linha deve ser espelhado na outra de forma alinhada, o que torna o barramento mais protegido contra ruídos e interferências eletromagnéticas (Corrigan, 2016; Denton, 2020).

A Figura 4 mostra as formas de ondas de tensão elétrica nas linhas de transmissão CAN *High* e CAN *Low* em comparação com transmissão de *bits* ou *bitstream* (representado em verde) dos dados via barramento.

Figura 4 – Formas de onda do CAN



Fonte: Elaborada pelo autor baseado em (Denton, 2020; Corrigan, 2016).

A Figura 4 ilustra o sinal e os *bits* transmitidos envolvidos no protocolo, que são descritos na subseção 2.3.2.1. A comunicação ocorre dentro da faixa de tensão de $1,5\ V$ a $3,5\ V$ e opera em dois estados distintos: dominante e recessivo. Além disso, é evidente o espelhamento do sinal CAN *High* (representado em azul) no sinal CAN *Low* (indicado em vermelho).

Tabela 1 – Estados do barramento CAN

Estado	**CAN *High***	**CAN *Low***	**Diferença(CAN_H - CAN_L)**	***Bit***
Dominante	$3,5\ V$	$1,5\ V$	$2\ V$	0
Recessivo	$2,5\ V$	$2,5\ V$	$0\ V$	1

Fonte: Elaborada pelo autor baseado em (Corrigan, 2016).

A Tabela 1 demonstra com mais detalhes a distinção entre os estados dominante e recessivo. O estado dominante acontece quando a diferença de tensão entre o sinal CAN *High* e CAN *Low* é de $2\ V$ e quando o *bit* do dado transmitido é igual a 0. O estado recessivo acontece quando as tensões do sinais CAN *High* e CAN *Low* são iguais, ou seja, sua diferença é igual a $0\ V$ e quando o *bit* do dado transmitido é igual a 1. Quando nenhuma transmissão está em andamento, o barramento fica ocioso (*idle*) e em estado recessivo (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2015). Assim, a forma de onda gerada na

comunicação é diretamente influenciada pelo formato da mensagem transmitida e pelo seu conteúdo.

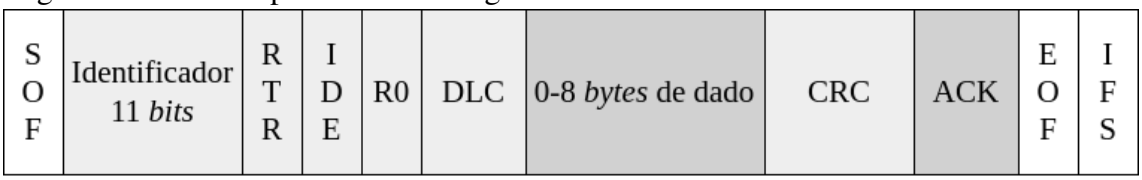
2.3.2 Formato da mensagem

O protocolo CAN possui dois padrões de formato de mensagem: o formato padrão e o estendido. O formato padrão possui um identificador de 11 *bits* e até 8 *bytes* de dados. Entretanto, o formato estendido possui um identificador de 29 *bits* e até 8 *bytes* de dados (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION, 2015).

Contudo, existe um método para arbitragem de mensagens em que os dispositivos competem pelo acesso ao barramento. Isso garante que as mensagens mais críticas tenham maior prioridade, mantendo a integridade do sistema. A arbitração é feita de acordo com o identificador da mensagem, quanto menor, maior a prioridade (Corrigan, 2016).

2.3.2.1 Formato padrão

Figura 5 – Formato padrão da mensagem CAN



Fonte: Elaborada pelo autor baseado em (Corrigan, 2016).

Os campos de *bits* na Figura 5 representam:

- *Start of Frame* (SOF): Um *bit* dominante que marca o início da mensagem.
- Identificador (ID): Identificador padrão de 11 *bits*. Estabelece a prioridade da mensagem e distingue uma das outras.
- *Remote Transmission Request* (RTR): Um *bit* para distinguir entre mensagens de solicitação de informações de outro nó (dominante) e mensagens de dados (recessivo).
- *Identifier Extension* (IDE): Um *bit* dominante indica um ID padrão. Um *bit* recessivo indica um ID estendido.
- R0: *Bit* reservado.
- *Data Length Code* (DLC): Campo de 4 *bits* que contém o número de *bytes* de dados que estão sendo transmitidos.
- Dados (D): Campo de até 64 *bits* (8 *bytes*) de dados da aplicação que podem ser transmiti-

dos.

- *Cyclic Redundancy Check* (CRC): Campo de 16 *bits* que contém o *checksum* do dados anteriores para a detecção de erros.
- *Acknowledgement* (ACK): Dois *bits*, um para confirmação e o segundo como um delimitador. Cada nó que recebe uma mensagem precisa substituir esse *bit* recessivo na mensagem original por um *bit* dominante, indicando que uma mensagem sem erros foi enviada. Caso o nó receptor detecte um erro e deixe este *bit* recessivo, ele descarta a mensagem e o nó emissor repete a mensagem após a rearbitragem.
- *End of Frame* (EOF): Campo de 7 *bits* recessivos que marca o final de uma mensagem e desativa o preenchimento de *bits*, indicando um erro de preenchimento quando dominante.
- *Interframe Space* (IFS): Campo de 7 *bits* recessivos reservado entre mensagens consecutivas, criando um intervalo de tempo que permite a sincronização dos nós na rede e a preparação para a chegada da próxima mensagem, assim evitando possíveis colisões.

Esses campos desempenham papéis cruciais no protocolo CAN, garantindo a transmissão eficiente e confiável de mensagens (Corrigan, 2016).

2.3.2.2 Formato estendido

Figura 6 – Formato estendido da mensagem CAN

SOF	Identificador 11 *bits*	SRR	IDE	Identificador 18 *bits*	RTR	R1	R0	DLC	0-8 *bytes* de dado	CRC	ACK	EOF	IFS

Fonte: Elaborada pelo autor baseado em (Corrigan, 2016).

Conforme mostrado na Figura 6, a mensagem no formato estendido é igual a mensagem no formato padrão com a adição e modificação de:

- *Substitute Remote Request* (SRR): Um *bit* para indicar a solicitação de dados adicionais de outro nó.
- IDE: Um *bit* recessivo que indica o formato estendido.
- R1: Um *bit* reserva adicional.

Esses campos adicionais e a modificação do *bit* IDE possibilitam o uso da mensagem CAN no formato estendido (Corrigan, 2016).

O protocolo CAN foi projetado para ser resistente a erros. Ele utiliza uma forma de detecção de erros baseado no campo de *bits* CRC, presente nos dois formatos, para verificar

a integridade dos dados transmitidos. Esse tratamento de falhas é integrado, se qualquer nó defeituoso aparecer, automaticamente ele será bloqueado (Wagh *et al.*, 2017). Entretanto, para tornar possível a utilização da rede CAN, é imprescindível a presença de um controlador CAN que funcione como a interface entre os dispositivos e o barramento.

2.3.3 Controlador CAN

Um controlador CAN pode ser entendido como uma interface entre a aplicação e o barramento CAN. Sua principal responsabilidade é converter os dados fornecidos pela aplicação em uma mensagem CAN adequada para a transmissão através do barramento. Da mesma forma, desempenha um papel fundamental na recepção da mensagem através do barramento e sua conversão em dados de volta para a aplicação (Semiconductor, 2023). Para viabilizar a comunicação CAN, é essencial a integração de um controlador CAN com um microcontrolador.

2.4 Microcontrolador

Microcontroller Unit (MCU), em português Microcontrolador, é um pequeno processador programável que consiste basicamente em uma *Central Processing Unit* (CPU), em português Unidade Central de Processamento, relógio do sistema, memória e periféricos. Os MCUs são notáveis pelo seu baixo consumo de energia, custo acessível e tamanho reduzido, mas com recursos limitados de computação e memória (Gao *et al.*, 2019).

O MCU desempenha o papel de "um cérebro inteligente", na medida em que lida com todas as tarefas do sistema, processa dados, toma decisões e controla periféricos. Devido aos seus recursos restritos e arquitetura relativamente simples, os MCUs são frequentemente dedicados a uma ou mais tarefas simples, em vez de processar múltiplas tarefas complexas simultaneamente (Tahat *et al.*, 2012).

2.5 Sensores

Sensores são os dispositivos básicos necessários para detectar e converter os parâmetros físicos em sinal elétrico a partir de estímulos. A sua função é emitir um sinal que seja capaz de ser convertido e interpretado por outros dispositivos. Esses estímulos variam desde calor, movimento, corrente, campo magnético e assim por diante (Vargas *et al.*, 2021; Bhuyan, 2010).

Além disso, eles são usados para perceber o mundo, ou seja, capturar parâmetros

da natureza como temperatura, aceleração, corrente elétrica, entre outras coisas. Com isso, a estrutura de um sensor fica mais ligada fisicamente ao ambiente em operação (Bhuyan, 2010).

Atuadores são dispositivos que executam ações físicas em resposta aos sinais dos sensores. Quando conectados a um microcontrolador, eles formam um nó da rede CAN, também conhecido como como Unidade de Controle Eletrônico (ECU) (Alam, 2018).

2.6 *Electronic Control Unit* (ECU)

Um nó da rede CAN é um dispositivo conectado ao barramento CAN que envia ou recebe mensagens. Por outro lado, uma Unidade de Controle Eletrônico (ECU) é um tipo específico de nó que atua como um computador embarcado em veículos, responsável pelo controle de sistemas individuais do veículo (Alam, 2018).

Entre as ECUs mais significativas estão aquelas responsáveis pelo controle do *Anti-Lock Brake System* (ABS), em português Sistema de Freios Antitravamento, pelo módulo de assistência ao estacionamento, pelo *Powertrain Control Module* (PCM), em português Módulo de Controle do Trem de Força, pela ECU de *airbag* e pela ECU de controle do cinto de segurança (Valasek; Miller, 2014). Em sua maioria, essas ECUs compartilham componentes essenciais, incluindo um microcontrolador, memória, uma porta de alimentação e uma interface de comunicação para interagir com outras ECU ou dispositivos de controle (Alam, 2018). Além disso, ECUs são essenciais para a integração de sistemas veiculares avançados de assistência ao motorista e de entretenimento em veículos.

2.7 Sistemas veiculares avançados

O *software* na indústria automotiva é composto por centenas de componentes que executam algoritmos complexos de monitoramento e sistemas de assistência ao motorista, com requisitos rígidos de segurança e processamento em tempo real. Os componentes do sistema automotivo hoje em dia são integrados não só com o propósito de capacitar o veículo para atender às expectativas de confiabilidade, mas também de entregar aos passageiros as diversas informações e entretenimento (Kenjić *et al.*, 2023).

Devido a essa diversidade de demandas e à sua abrangência, os sistemas veiculares estão passando por modificações e evoluções arquitetônicas, e atualmente estão desacoplados em vários domínios (Kenjić; Antić, 2023). Diante do ponto de vista do desenvolvimento de *software*,

dois domínios estão em rápida evolução: o Sistema Avançado de Assistência ao Motorista (ADAS), responsável pela assistência aos motoristas e recursos de condução autônoma, e o Infoentretenimento no Veículo (IVI), focado na experiência do passageiro (Kenjić *et al.*, 2023).

2.7.1 Advanced Driver Assistance Systems (ADAS)

O ADAS fornece suporte seguro aos motoristas utilizando sensores e câmeras para evitar colisões, detectar objetos em pontos cegos, manter o veículo na faixa correta e alertar sobre problemas (Omerovic *et al.*, 2016; Denton, 2020). Ele é oferecido por fornecedores que simplificam a integração e fornecem interfaces de comunicação entre componentes de *software* e outros serviços do sistema (Kenjić *et al.*, 2023).

O trabalho proposto está vinculado ao domínio do ADAS, especificamente na aquisição em tempo real de dados veiculares, relato de defeitos e sua integração com o IVI. No entanto, é importante ressaltar que este estudo não se concentra diretamente na implementação das funcionalidades complexas do ADAS, como as mencionadas anteriormente.

2.7.2 In-vehicle Infortainment (IVI) System

Traduzido do inglês, Sistema de Infoentretenimento no Veículo (IVI) ou Sistema de Informação e Entretenimento no Veículo, é um conjunto de *hardware* e *software* automotivo para entretenimento de áudio e vídeo. Os recursos comuns do IVI incluem multimídia, controle de temperatura, dados do veículo, navegação e diferentes visualizações provenientes do ADAS (Omerovic *et al.*, 2016).

Atualmente, a solução mais avançada para o domínio IVI é a plataforma *Android Automotive*, que possui uma estrutura adequada para atender aos requisitos automotivos (Google, 2023a). Esses requisitos incluem um melhor gerenciamento de energia e consumo de bateria, um processo de inicialização mais rápido, suporte a barramentos de rede como o CAN, e a integração de módulos de sensores para percepção ambiental (Omerovic *et al.*, 2016).

O IVI é impulsionado principalmente pelas demandas dos consumidores. O foco é voltado para *User Interface / User Experience* (UI/UX), traduzido do inglês Interface de Usuário / Experência de Usuário, criação de interfaces para interação do usuário e funcionalidades semelhantes às oferecidas pelos dispositivos móveis (Kenjić *et al.*, 2023).

2.7.2.1 Interface

Sobre interface, de acordo com Barbosa *et al.* (2021), "a interface de um sistema interativo compreende toda a porção do sistema com a qual o usuário mantém contato físico (motor ou perceptivo) ou conceitual durante a interação" (Moran, 1981). O contato com a interface envolve a interpretação do usuário daquilo que ele percebe durante o uso do sistema. Essa interpretação permite ao usuário compreender as respostas do sistema e planejar os próximos caminhos de interação (Barbosa *et al.*, 2021).

Diante disso, surge o conceito de comunicabilidade, proposto pela engenharia de semiótica contida na teoria de Interação Humano-Computador (IHC) (Souza, 2005 apud Barbosa *et al.*, 2021). A comunicabilidade de um sistema se refere à capacidade do usuário entender, através da interface, para que o sistema serve, a quem ele se destina, quais as vantagens de utilizá-lo e como ele funciona. Dessa maneira, as informações passadas pela interface podem ser transmitidas de forma compreensível, clara e eficaz (Barbosa *et al.*, 2021).

Com base no que foi exposto, compreende-se que, para que o trabalho proposto alcance seu objetivo, é necessário uma interface com boa comunicabilidade. Essa interface desempenha um papel crucial ao informar de forma clara e compreensível ao usuário os defeitos do veículo identificados pelo sistema. Para isso ser possível, é necessário a presença de um sistema IVI junto ao *Android Automotive*, tornando imprescindível o uso de um Computador de Placa Única (*Single Board Computer* (SBC)).

2.8 *Single Board Computer* (SBC)

Um SBC é um computador que possui microprocessador, memória, portas de entrada e saída e outras funcionalidades em um único dispositivo de tamanho bastante compacto. São altamente portáteis e eficientes em termos de energia, são muito populares nas comunidades de entusiastas e foram adaptados para uma variedade de projetos (Matthews *et al.*, 2016).

Possuem conectividade *Ethernet* e *Universal Serial Bus* (USB), suporte a protocolos de comunicação como *Serial Peripheral Interface* (SPI), *Inter-Integrated Circuit* (I2C) e *Universal Asynchronous Receiver/Transmitter* (UART). De forma geral, esses computadores são adequados para projetos com requisitos que não podem ser cumpridos por microcontroladores (McManus; Cook, 2021; Matthews *et al.*, 2016). Embora existam muitos tipos de SBCs, o *Raspberry Pi 4 Model B*, mostrado na Figura 17, se destaca como o mais popular e conhecido, e

compatível com o *kernel Linux* (Matthews *et al.*, 2016).

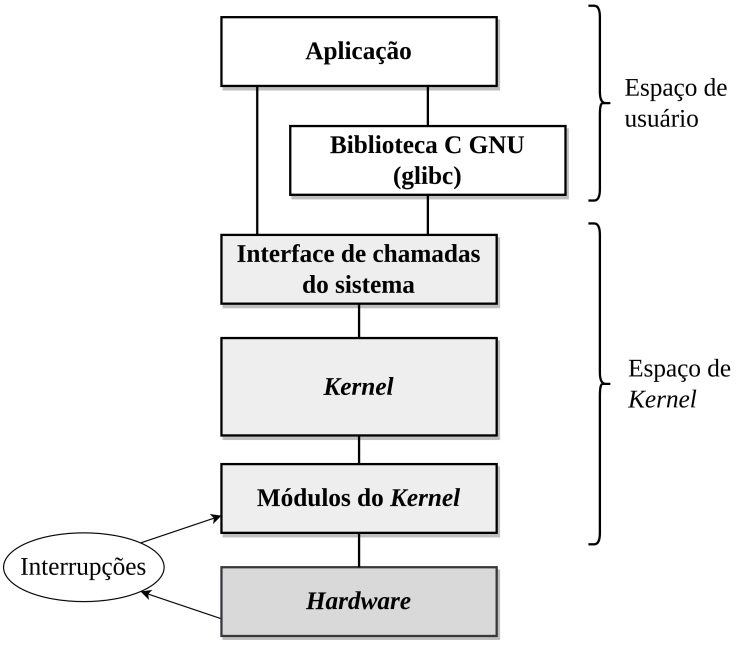
2.9 *Kernel Linux*

O *Linux* foi escrito em 1991 por Linus Torvalds. Ele foi inspirado no sistema operacional *Minix* escrito por Andrew S. Tanenbaum, 4 anos antes. Para ser mais preciso, Linus escreveu um *kernel*, que é apenas um componente de um sistema operacional. Para criar um sistema operacional completo, ele usou componentes do projeto *GNU's not Unix* (GNU), especialmente o conjunto de ferramentas, a biblioteca C (glibc) e ferramentas básicas de linha de comando (Vasquez; Simmonds, 2021).

O *kernel Linux* pode ser combinado com um espaço de usuário GNU para criar uma distribuição Linux completa que executa em *desktops* e servidores, que por vezes é chamada de GNU/*Linux* (Vasquez; Simmonds, 2021).

O *kernel* tem três funções principais: gerenciar recursos, fazer interface com *hardware* e fornecer uma *Application Programming Interface* (API) que ofereça um nível útil de abstração para programas de espaço do usuário (Vasquez; Simmonds, 2021), conforme resumido na Figura 7.

Figura 7 – Arquitetura do *Linux*



Fonte: Elaborada pelo autor baseado em (Vasquez; Simmonds, 2021).

Ademais, a estrutura robusta do *kernel Linux* serve como alicerce para inúmeros softwares e projetos. O projeto *Android*, juntamente com o *Android Automotive*, incorpora o

kernel Linux como elemento fundamental de sua arquitetura.

2.10 *Android Automotive* OS (AAOS)

Android Automotive OS é uma extensão do sistema operacional *Android* para funcionar diretamente com sistemas IVI. Por ser uma extensão do *Android*, o código-fonte do *Android Automotive* pode ser encontrado no repositório do *Android*, oferece os mesmos recursos de escalabilidade, robustez, modelos de segurança, ferramentas de desenvolvedor, a mesma infraestrutura do *Android*, juntamente com APIs adicionais para vincular, suportar e controlar características do veículo (Moiz; Alalfi, 2022).

O *Android* Automotivo é uma plataforma completa, de código aberto e altamente personalizável e é executada diretamente no *hardware* do veículo. O AAOS é frequentemente confundido com o *Android Auto*, mas eles diferem em alguns aspectos (Google, 2023a):

- ***Android Auto*** é uma plataforma executada no telefone do usuário, projetando aplicativos para um sistema de infoentretenimento presente no veículo por meio de uma conexão USB.
- ***Android Automotivo*** é um sistema operacional e uma plataforma executados diretamente no *hardware* do veículo. É uma plataforma completa, de código aberto e altamente personalizável que potencializa a experiência de infoentretenimento.

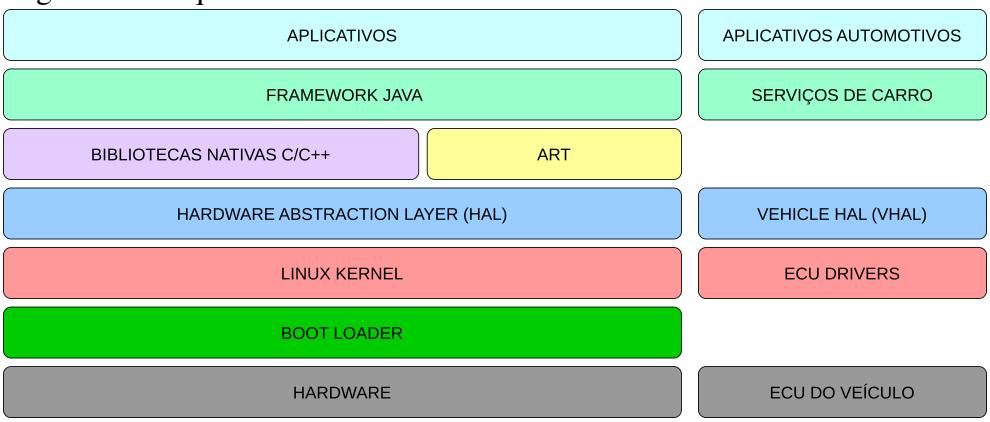
2.10.1 Arquitetura do Android Automotivo

O *Android* é baseado no *Linux*, um *kernel* de código aberto criado para diversos dispositivos. O diagrama a seguir mostra uma visão geral das camadas e dos componentes do AAOS (Google, 2020).

A Figura 8 representa a arquitetura da plataforma *Android* Automotivo, composta por várias camadas, seguindo a mesma estrutura da plataforma *Android*, mas com adição de camadas específicas para automóveis. O uso do *kernel Linux* permite que os fabricantes dos dispositivos desenvolvam *device drivers* para um *kernel* conhecido (Google, 2020). A HAL desempenha o papel mais importante aqui. Ela atua como uma camada de separação entre os dispositivos físicos de *hardware* e o *Android*, simplesmente abstraindo sua funcionalidade em interfaces de *software* (Iseke, 2020). Vários componentes e serviços principais do *Android*, como a HAL, são implementados por meio de código nativo que exige bibliotecas nativas programadas

em C e C++. A plataforma *Android* fornece as Java *Framework* APIs para expor a funcionalidade de algumas dessas bibliotecas nativas para as aplicações (Google, 2020).

Figura 8 – Arquitetura do *Android* Automotivo



Fonte: Elaborada pelo autor baseado em (Google, 2023b).

2.10.2 Vehicle Hardware Abstract Layer (VHAL)

A interface VHAL, ou Camada de Abstração de *Hardware* do Veículo, é uma das muitas HALs disponíveis no *Android* Automotivo e descreve a comunicação com as redes veiculares, como o CAN, através das funções presentes nela. A partir da versão 13 do *Android*, a VHAL passou a ser definida em AIDL, uma linguagem que permite criar uma interface entre serviços. Além disso, é baseada no acesso (leitura, escrita e assinatura) de uma propriedade, que é uma abstração para uma função específica. Essas propriedades contém metadados que podem ser desde inteiros até uma cadeia de caracteres, e com elas é possível vincular o *hardware* do veículo ao *software Android* (Iseke, 2020; Google, 2023c).

Adicionalmente, a VHAL é disponibilizada para a camada de aplicação pelo *CarPropertyManager*, que disponibiliza a API para manipulação das propriedades e seus valores. Com isso, é possível fazer com que um aplicativo nativo interaja com o veículo através dessa interface, e vice-versa (Iseke, 2020). Este trabalho visa desenvolver um serviço que registra os dados obtidos da rede CAN em propriedades da VHAL, permitindo que sejam acessados por um aplicativo, o qual exibirá os defeitos veiculares no sistema IVI do veículo.

3 TRABALHOS RELACIONADOS

Nesta seção serão retratados alguns trabalhos da literatura relacionados ao monitoramento do estado do veículo, a captura e interpretação de defeitos veiculares por meio do barramento CAN, além da criação de sistemas para visualização dessas informações.

3.1 *Integrated OBD-II and Mobile Application for Electric Vehicle (EV) Monitoring System*

O trabalho de Khorsravinia *et al.* (2017) aborda a crescente popularidade dos *Electric Vehicle*s (EVs), Veículos Elétricos, e sua conectividade à *internet*, que tem incentivado os usuários a adotar tecnologias para rastrear, monitorar e controlar seus veículos. No entanto, atualmente não existe um sistema unificado que integre todas essas funcionalidades em um único aplicativo. O objetivo deste estudo é desenvolver um sistema que se comunique com o sistema de diagnóstico a bordo de veículos elétricos, OBD, usando o protocolo de comunicação CAN e comunicação *Bluetooth*.

O trabalho descreve como coletar dados da rede CAN por meio de uma abordagem econômica e como implementar um sistema OBD integrado para monitorar mensagens CAN, como estado de carga e consumo de energia, em dispositivos *Android*. O sistema proposto também permite o controle de funções específicas do veículo, como travar e destravar portas e controlar o ar-condicionado.

Os autores realizaram um experimento de pré-validação utilizando dois métodos diferentes para a coleta de dados. No primeiro método foi utilizado um carro híbrido, fazendo a conexão entre os pinos CAN da interface OBD do carro e a placa microcontroladora. No segundo método, a coleta de dados é feita a partir de uma bancada. Isso envolve a presença de uma ECU particular conectada a uma fonte de energia e a conexão dos pinos CAN *High* e CAN *Low* com a placa microcontroladora.

Quadro 1 – Identificação de um código de falha

103108	Permanent	Accelerator pedal module, pedal sensor 2, operating range: Voltage too low

Fonte: (Khorsravinia *et al.*, 2017).

O Quadro 1 exibe um resultado alcançado durante o experimento, a identificação de um código de falha relacionado ao PCM, acompanhado de uma descrição que indica um problema na operação do módulo do pedal do acelerador devido à baixa tensão.

Em pesquisas futuras, os autores planejam expandir a aplicação, visando a utilização de todos os dados relevantes, com o objetivo de monitorar informações adicionais, como a localização de acidentes para solicitação de auxílio.

3.2 *Implementation of the Android-Based Automotive Infortainment System for Supporting Drivers' Safe Driving*

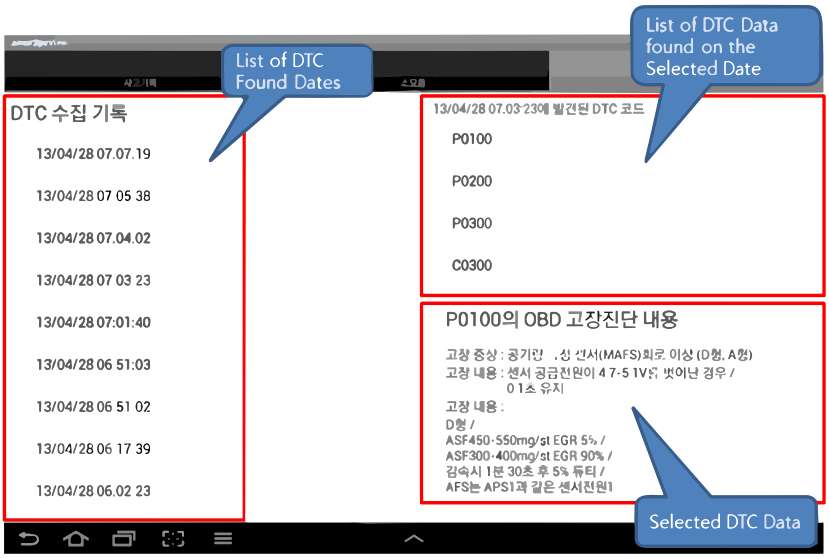
No estudo de Kim *et al.* (2014), foi abordada a implementação de um sistema de infoentretenimento automotivo baseado em *hardware* embarcado na plataforma *Android*. Esse sistema oferece suporte à segurança do motorista, incluindo funcionalidades como a caixa preta e autodiagnóstico, além de englobar as funções típicas de infoentretenimento automotivo, como navegação e multimídia. Além disso, proporciona um ambiente flexível que permite a adição de novas funções controladas pelo condutor, algo que geralmente não é viável nos sistemas tradicionais.

Os dados do veículo são coletados por meio da ECU e do protocolo OBD. Esses dados obtidos servem para a execução das funções de caixa preta e autodiagnóstico do veículo, que são posteriormente apresentadas aos ocupantes por meio de um sistema de informação e entretenimento.

Os autores projetaram o sistema de informação e entretenimento automotivo composto por dois componentes: um *hardware* embarcado e uma tela sensível ao toque. O *hardware* oferece serviços de infoentretenimento diretamente aos motoristas, enquanto o coletor de dados do veículo transmite informações para o *hardware* embarcado via *Bluetooth*. O seu sistema fornece três serviços básicos do *Android*: navegação e multimídia; um gravador de caixa preta que registra acidentes e coleta dados do veículo; e um revisor de gravações de caixa preta que lida com códigos de falhas, análise de acidentes e verificação da vida útil dos componentes consumíveis.

A Figura 9 mostra os resultados do experimento realizado em um passeio de carro pela cidade. Foram identificadas algumas falhas veiculares, na tela são mostradas as datas na qual as falhas foram encontradas, a lista de códigos encontrada ao ser selecionada a data específica e o dado presente nelas, mostrando que a detecção de falhas acontece e é mostrada ao usuário.

Figura 9 – Função de revisão da caixa preta e sua composição de tela



Fonte: (Kim *et al.*, 2014).

Os experimentos realizados pelos autores demostraram que foi possível capturar e gravar os dados internos do automóvel, detectar falhas e exibi-las através de um sistema de infoentrenimento no ambiente do *Android*.

Visando pesquisas futuras, os autores reforçam a necessidade de melhorar a interface gráfica para funções adicionais. Também será estudada uma forma de corrigir os erros que aconteceram durante a operação.

3.3 *An Integrated Embedded Solution for Driving Support*

Tufo *et al.* (2014) propõem uma solução embarcada que captura as informações do veículo (velocidade do veículo, rotação do motor, temperatura do líquido de refrigeração, consumo instantâneo de combustível etc.), provenientes de vários ECUs e as envia para um servidor para novos serviços de apoio à condução.

O trabalho consiste na captura de informações do barramento CAN através da interface OBD. As informações capturadas consiste em *timestamp*, identificador e o dado da mensagem CAN. Após a coleta de dados realizada pelo *hardware* embarcado, a informação coletada é enviada a um dispositivo móvel por meio de comunicação TCP/IP.

Por sua vez, a aplicação móvel foi desenvolvida em *Java* e foi projetada para executar as seguintes tarefas:

1. Conectar-se ao sistema embarcado;

2. Receber dados do barramento CAN através do circuito embarcado (OBD);
3. Exibir a informação para o usuário por meio de uma interface amigável;
4. Empacotar e enviar informações ao servidor via *Wifi* ou rede celular;

Além disso, a informação no aplicativo é constantemente atualizada para fornecer ao usuário informações sobre o veículo em tempo real.

A Figura 10 ilustra o estado da aplicação no momento da pesquisa. O lado esquerdo da imagem retrata o painel de um carro, promovendo a imersão do usuário ao sistema. Já no lado direito da imagem, é possível perceber a porta na cor vermelha, informando que ela tem algo que precisa ser consertado. Com isso, é possível perceber o fluxo de dados CAN chegando ao dispositivo móvel e o seu processamento na detecção de falhas e suporte ao motorista.

Figura 10 – Aplicação móvel que mostra ao usuário informações sobre o veículo

Fonte: (Tufo *et al.*, 2014).

3.4 *An Android-based IoT System for Vehicle Monitoring and Diagnostic*

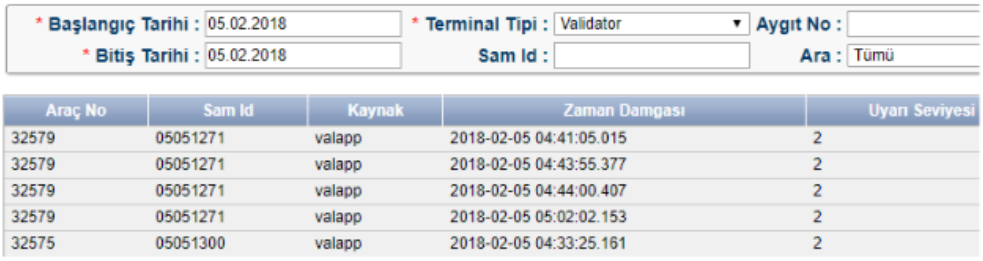
Türk e Challenger (2018) introduzem um sistema baseado em *Internet of Things* (IoT) com o propósito de monitorar o estado do veículo e identificar possíveis falhas, coletando informações do barramento CAN, bem como dados de *Global Positioning System* (GPS) e *Inertial Measurement Unit* (IMU). A primeira parte trata do projeto de *hardware* para obter dados do GPS e sensores IMU via barramento CAN. A segunda parte aborda a implementação do servidor, que envia dados do sistema embarcado do veículo para um servidor remoto e os torna úteis para o usuário.

O trabalho coleta informações a partir de um módulo controlador CAN, o MCP2515. Todos os dados são transmitidos pelo módulo CAN conectado ao processador *Advanced RISC Machine* (ARM) *Cortex*® A8 que executa o *Android*. Os dados da IMU são obtidos a partir de um

sensor acelerômetro, conectado através do I2C, e os dados de posição são obtidos através de um módulo GPS. Após a aquisição dos dados, eles são enviados para o servidor e disponibilizados aos usuários.

A Figura 11 mostra a tela do usuário final contendo as informações que são passadas via CAN. Segundo os autores, nesta tela o usuário pode visualizar todas as informações do CAN e do sensor IMU, associadas à hora e localização.

Figura 11 – Tela do usuário baseado na nuvem contendo as informações passadas via CAN



Fonte: (Türk; Challenger, 2018).

3.5 Análise comparativa

O trabalho de Khorsravinia *et al.* (2017) monitora e controla carros elétricos em um único aplicativo. Os dados da rede CAN são coletados por meio do OBD, utilizando um transceptor. No trabalho proposto, é utilizado o módulo MCP2515 para coletar dados da CAN e integrá-los ao *Android Automotive*, com uma interface comunicável para exibir defeitos em um sistema de informação e entretenimento automotivo.

Em Kim *et al.* (2014), é proposto o desenvolvimento de um sistema IVI baseado em *Android*, com foco na segurança, semelhante ao trabalho proposto. Porém, os dados são coletados por meio de um interpretador OBD, não faz uso do AAOS para o desenvolvimento do sistema, e as informações e defeitos do veículo são exibidas de forma confusa e aglomerada.

O trabalho de Tufo *et al.* (2014) descreve um sistema embarcado que coleta dados da rede CAN, usando OBD como interface. Embora não utilize AAOS, o sistema de Tufo *et al.* (2014) apresenta uma interface de usuário comunicável, apresentando os defeitos do veículo, semelhante à proposta deste trabalho.

Já Türk e Challenger (2018) propõem um sistema baseado em IoT que monitora e identifica possíveis defeitos. Similar ao trabalho proposto, também sugere a integração da rede CAN e o uso do módulo MCP2515 com o sistema *Android*. Entretanto, não desenvolve um

sistema de informação e entretenimento automotivo.

O Quadro 2 sintetiza diferenças e semelhanças do trabalho proposto com os relacionados.

Quadro 2 – Análise comparativa entre os trabalhos relacionados e este projeto

Trabalho	**Acesso direto ao CAN**	***Android Automotive***	**Visualização de defeitos**	**Sistema IVI**	**Interface comunicável**
Khorsravinia *et al.* (2017)	Sim	Não	Sim	Não	Não
Kim *et al.* (2014)	Não	Não	Sim	Sim	Não
Tufo *et al.* (2014)	Sim	Não	Sim	Sim	Sim
Türk e Challenger (2018)	Sim	Não	Sim	Não	Não
Esta proposta	Sim	Sim	Sim	Sim	Sim

Fonte: Elaborado pelo autor (2023).

4 PROCEDIMENTOS METODOLÓGICOS

Para atingir com sucesso os objetivos definidos neste trabalho, é fundamental seguir um conjunto de etapas metodológicas bem definidas, conforme ilustrado pelo fluxograma da Figura 12.

Figura 12 – Fluxograma da metodologia

Fonte: Elaborado pelo autor (2023).

O processo de validação e avaliação envolve a realização de experimentos em etapas específicas, garantindo a robustez e a eficácia das ações executadas. A seguir, cada uma das etapas e sub-etapas que compõem a metodologia são descritas detalhadamente, proporcionando uma visão abrangente do processo de pesquisa e desenvolvimento.

4.1 Construção da rede CAN

O primeiro passo para a construção da rede CAN foi definir os requisitos para comunicação na rede, que consistem em um conjunto de tarefas a serem executadas pelo *software*. Em seguida, foi realizada a seleção, estudo e compra de componentes e placas para construir uma rede CAN que suporte até dois nós. Foram feitas consultas em fóruns, *Google Scholar* e *blogs* de tecnologia embarcada para identificar os componentes mais usados em aplicações com a rede CAN. Além disso, foram pesquisados SBCs compatíveis com o *Android Automotive*.

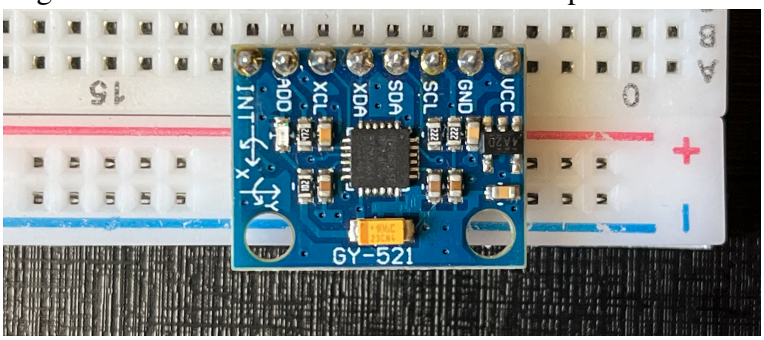
4.1.1 Componentes selecionados

Cada placa e componente identificado foi estudado individualmente através de documentos, *datasheets* e tutoriais. A seleção foi baseada em critérios como: conhecimento prévio, facilidade de uso, documentação adequada, comunidade ativa, compatibilidade com o *kernel Linux* e *Android* (para SBCs), e custo acessível.

4.1.1.1 Sensor MPU-6050

O MPU-6050, mostrado na Figura 13, se trata de um sensor acelerômetro e giroscópio capaz de medir aceleração e rotação nos três eixos (x, y e z), podendo detectar a variação brusca de velocidade do carro, útil no acionamento de *airbags*. A sua comunicação com o microcontrolador é através do I2C, protocolo bastante conhecido pela sua quantidade de fios, um para o dado (SDA) e outro para o *clock* (SCL) (IvenSense, 2013). O sensor atendeu aos critérios de: conhecimento prévio, facilidade de uso, boa documentação e custo acessível.

Figura 13 – Sensor Acelerômetro e Giroscópio - MPU6050

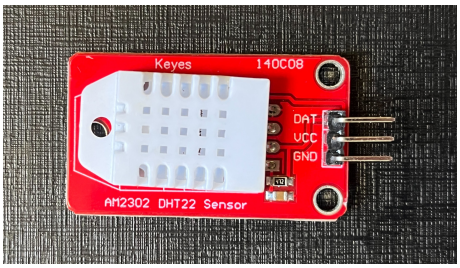


Fonte: Acervo pessoal do autor (2023).

4.1.1.2 Sensor DHT22

O DHT22, mostrado na Figura 14, é um sensor de temperatura e umidade com comunicação digital por um fio. Neste trabalho, o sensor fornecerá medidas de temperaturas ao sistema, simulando a temperatura do motor. O sensor não necessita de calibração, lê temperaturas entre $-40\ ^{\circ}C$ e $80\ ^{\circ}C$ e necessita de um tensão de entrada entre $3,3\ V$ e $6\ V$ (MaxDetect, 2013). O sensor atendeu os critérios de: facilidade de uso, custo acessível e documentação adequada.

Figura 14 – Sensor de Temperatura e Umidade - DHT22

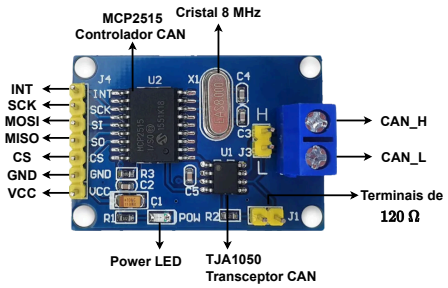


Fonte: Acervo pessoal do autor (2023).

4.1.1.3 Módulo MCP2515

O módulo MCP2515 escolhido é um controlador CAN da empresa *Microchip Technology* que implementa a versão 2.0B do CAN. É capaz de transmitir e receber dados no formato padrão e estendido. A interface entre o módulo e o MCU é feita através do protocolo SPI, que permite a comunicação com diversos dispositivos, formando uma rede (Microchip, 2019).

Figura 15 – Pinagem do módulo MCP2515



Fonte: Elaborada pelo autor baseado em (Lab, 2023).

A Figura 15 oferece uma visão detalhada do módulo MCP2515. A imagem destaca vários componentes, incluindo o circuito integrado MCP2515 em si, um cristal oscilador de 8 MHz, as duas linhas de transmissão CAN_H (CAN High) e CAN_L (CAN Low), dois terminais para habilitar a resistência de 120 Ω, o circuito integrado TJA1050 responsável pela interface entre o controlador do protocolo CAN e o barramento físico (Philips, 2003), um *LED* que indica o *status* de alimentação do módulo e, por fim, os pinos do protocolo SPI:

- *Interrupt* (INT): Pino para gerar uma interrupção.
- *Serial Clock* (SCK): Pino para fornecer o sinal de *clock* para o barramento SPI.
- *Master Output Slave Input* (MOSI): Pino para entrada serial de dados do barramento SPI.
- *Master Input Slave Output* (MISO): Pino para saída serial de dados para o barramento SPI.
- *Chip Select* (CS): Pino para seleção do dispositivo ou *chip*.
- *Ground* (GND): Pino responsável pelo aterramento.

- *Voltage Common Collector* (VCC): Pino dedicado para alimentação do módulo.

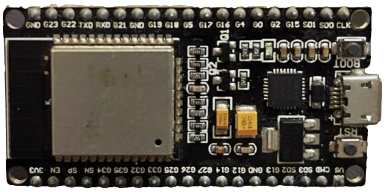
Esses pinos são essenciais para estabelecer a comunicação entre o módulo, um microcontrolador e o barramento CAN (Microchip, 2019). O MCP2515 foi escolhido por conta da compatibilidade com o *kernel Linux*, sua ótima documentação e comunidade ativa.

4.1.1.4 ESP32

O microcontrolador escolhido foi o ESP32, placa bastante versátil que se destaca pelo seu baixo custo, conectividade sem fio via *Wi-fi* e *Bluetooth*, grande quantidade de *General Purpose Input/Output*s (GPIOs), que são portas programáveis de entrada e saída de dados, como mostrado na Figura 16. Além disso, o ESP32 está bastante presente no cenário de IoT, mas é projetado para atender uma variedade de aplicações (Simões; Mafort, 2021).

É produzido pela Espressif Systems™ e baseado na arquitetura *Xtensa* LX6 da Tensilica™. A placa conta com o microprocessador de dois núcleos de 32 *bits* e 4 MB de memória *flash*. Além disso, conta com suporte a periféricos e aos principais protocolos de comunicação como: UART, I2C e SPI. Pode ser programada em C ou C++ por meio do ESP-IDF, *framework* oficial para o desenvolvimento de aplicações na ESP32 (Espressif, 2023).

Figura 16 – Microcontrolador *ESP32-WROOM*



Fonte: Acervo pessoal do autor (2023).

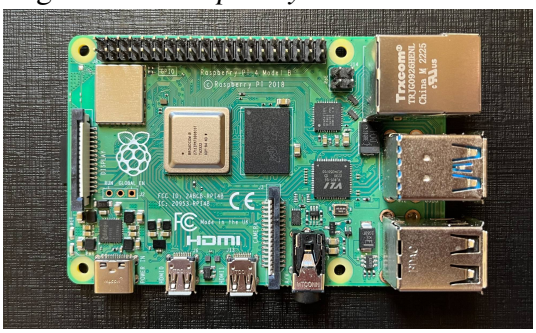
A ESP32 foi escolhida devido ao seu custo acessível, sua vasta documentação e sua comunidade extremamente ativa.

4.1.1.5 Raspberry Pi 4 Model B

A *Raspberry Pi 4B*, mostrada na Figura 17, é um SBC que possui um conjunto de *hardware* e interfaces robustas. Conta com um microprocessador *ARM-Cortex*™ A72 64 *bits* de até $1,5GHz$, pode oferecer até $8GB$ de memória *Synchronous Dynamic Random Access Memory* (SDRAM), em português Memória Dinâmica de Acesso Aleatório Sincronizado. Além disso, possui conectividade sem fio via *Wi-fi* e *Bluetooth* 5.0. Oferece GPIOs configuráveis

para SPI, I2C e UART (Raspberry, 2019). O contexto deste trabalho demanda um SBC de alto desempenho, razão pela escolha do *Raspberry Pi 4 Model B*, além de ser compatível com o *kernel Linux* e o AAOS e possuir comunidade bastante ativa.

Figura 17 – *Raspberry Pi 4B*



Fonte: Acervo pessoal do autor (2023).

4.1.2 Desenvolvimento do software de comunicação

Para possibilitar a comunicação entre os nós, *drivers* para o módulo e para os sensores foram desenvolvidos em C++, utilizando o *framework* ESP-IDF e programação orientada a objetos. No caso do *driver* para o módulo controlador CAN foi implementada a classe MCP2515 que herda a classe SPI, uma biblioteca auxiliar desenvolvida para manipular o barramento SPI do microcontrolador.

A biblioteca do acelerômetro permite a configuração e a leitura da aceleração em cada eixo. Também foi desenvolvida uma funcionalidade para a verificação de defeitos do sensor. Caso a biblioteca não consiga ler os dados do sensor, seja por desconexão ou falha na leitura no *driver* I2C, o valor de aceleração nos três eixos é definido com a macro INT16_MIN (-32768), um valor fora da escala de operação do sensor e extremamente improvável de ser alcançado em condições normais. Dessa forma, ao receber mensagens do sensor de aceleração, o sistema *Android* pode verificar a existência de um defeito no sensor.

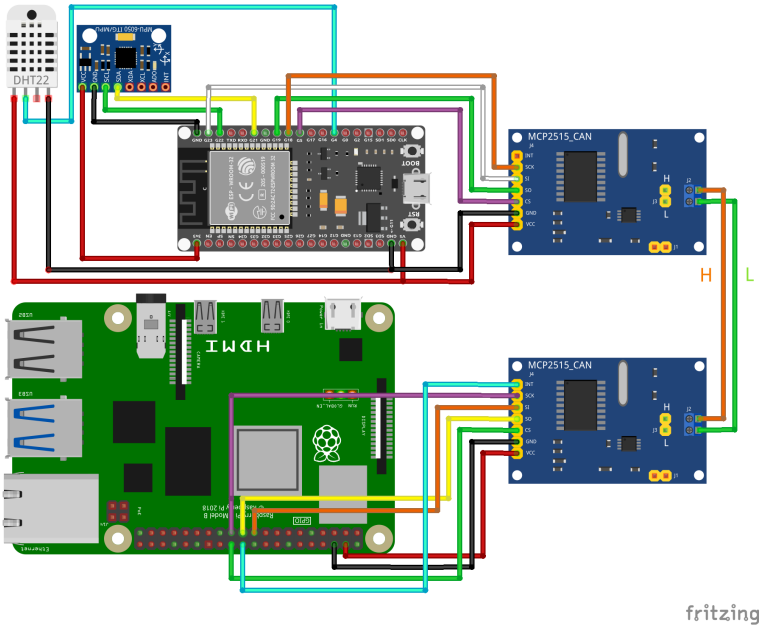
Já a biblioteca do sensor de temperatura, mais simples, permite a coleta da temperatura. A biblioteca também foi preparada para caso ocorra algum defeito com o sensor. Se não for possível ler o dado de temperatura, seja por desconexão, longo tempo para resposta ou até mesmo erro no *checksum* dos dados, o valor de temperatura é definido como $-273\ ^{\circ}C$ e enviado via CAN. Por se tratar do zero absoluto e ser um valor fora da escala de operação do sensor DHT22, é extramente improvável de ser alcançado em condições normais. Dessa forma, o sistema do AAOS conseguirá verificar a existência de um defeito no sensor.

O software que permite a comunicação da ESP32 com o barramento CAN foi desenvolvido no contexto do FreeRTOS, um sistema operacional de tempo real, para realizar tarefas em paralelo e gerenciar melhor os recursos do microcontrolador. Essa aplicação manipula os sensores através dos *drivers* desenvolvidos, captura suas leituras e as envia pela rede CAN.

4.1.3 Protótipo da rede CAN

A prototipação da rede CAN foi realizada no *Fritzing*, *software* voltado para *design* de *hardware*. Ela serviu como documentação e orientação para garantir a conexão correta de cada componente.

Figura 18 – Diagrama do circuito do protótipo



Fonte: Elaborada pelo autor (2023).

A Figura 18 ilustra o protótipo e conexão formada pelo dois nós. O primeiro nó, formado pela *Raspberry Pi* 4B e pelo módulo MCP2515, tem suas conexões detalhadas pelo Quadro 3.

Quadro 3 – Pinos de conexão entre MCP2515 e Raspberry Pi 4B

Pinos do MCP2515	VCC	GND	CS	MISO	MOSI	SCK	INT
Pinos da Raspberry	5V	GND	GPIO8	GPIO9	GPIO10	GPIO11	GPIO25

Fonte: Elaborado pelo autor (2023).

O segundo nó, responsável pelo envio de informações ao barramento, é formado pelo ESP32 e o sensores de aceleração (MPU6050) e temperatura (DHT22). O Quadro 4 e

Quadro 5 detalham a conexão dos sensores de aceleração e temperatura com o microcontrolador, respectivamente.

Quadro 4 – Pinos de conexão entre MPU-6050 e ESP32

Pinos do MPU-6050	VCC	GND	SCL	SDA
Pinos da ESP32	3V3	GND	GPIO22	GPIO21

Fonte: Elaborado pelo autor (2023).

Quadro 5 – Pinos de conexão entre DHT22 e ESP32

Pinos do DHT22	VCC	GND	DATA
Pinos da ESP32	5V	GND	GPIO4

Fonte: Elaborado pelo autor (2023).

Para tornar a comunicação com o primeiro nó possível, o ESP32 precisa estar conectado ao módulo MCP2515. A conexão entre os dois foi realizada como mostrado no Quadro 6. Vale ressaltar que a designação "N.C" é utilizada para indicar que determinados pinos não estão conectados nesta configuração.

Quadro 6 – Pinos de conexão entre MCP2515 e ESP32

Pinos do MCP2515	VCC	GND	CS	MISO	MOSI	SCK	INT
Pinos da ESP32	5V	GND	GPIO5	GPIO19	GPIO23	GPIO18	N.C

Fonte: Elaborado pelo autor (2023).

4.2 Integração com o *kernel Linux*

A segunda etapa do projeto envolveu a integração da rede CAN e do MCP2515 no *kernel Linux*. Como o *kernel* do *Android* é baseado no *kernel Linux*, a integração bem-sucedida nesse estágio assegura o funcionamento adequado no *Android Automotive*.

4.2.1 Preparação de imagem do kernel Linux

Para viabilizar a integração, o primeiro passo envolveu a preparação de um cartão micro SD com capacidade mínima de 8*GB*, contendo uma imagem do *kernel Linux* específica para o *Raspberry Pi 4B*. Para essa ação, foi utilizado o utilitário *Raspberry Pi Imager*[1]. Por meio do programa, foi selecionada a imagem do Ubuntu Server 22.04.3 LTS (64-bit) e o cartão micro

[1] Repositório do Raspberry Pi Imager. Acesso em: 17 out. 2023.

SD conectado ao computador. Em seguida, o programa preparou cada partição e o conteúdo necessário para o *Linux* ser executado na placa.

4.2.2 Compilação do módulo do kernel para o MCP2515

O segundo passo envolveu a busca em *blogs* e no repositório do *kernel* do *Raspberry Pi* por *drivers* do *Linux* para o módulo MCP2515 controlador da CAN. Um *driver* foi encontrado no repositório do *kernel*, o arquivo *mcp251x.c*[2].

O terceiro passo envolveu a compilação do *driver*, transformando-o em um *Kernel Object* (um arquivo .ko) compatível com a versão do *kernel* em uso. Para isso, foi criado um arquivo *Makefile*, que contém um conjunto de diretivas para a compilação do módulo. Em seguida, o arquivo *Makefile* junto com o *driver mcp251x.c* foram transferidos para o sistema de arquivos da *Raspberry* e em seguida compilado.

4.2.3 Inserção do módulo no kernel e implementação de um nó CAN

No quarto passo, o arquivo de configuração de *boot* do *kernel*, localizado em */boot/-firmware/config.txt*, foi adicionado as configurações (Karakurt, 2019):

Código-fonte 1 – Configuração do arquivo de inicialização do *kernel Linux*

```
1 dtparam=spi=on
2 dtoverlay=mcp2515-can0,oscillator=16000000,interrupt=25
3 dtoverlay=spi-bcm2835-overlay
```

Após as modificações, no quinto passo, o MCP2515 é conectado ao SPI 0 da *Raspberry Pi 4B* e o módulo é inserido no *kernel* utilizando o comando: ***insmod mcp251x.ko***.

Após o módulo ser inserido dentro do *kernel*, é feita a instalação do conjunto de ferramenta *can-utils*[3], o qual oferece uma variedade de utilitários para enviar e visualizar dados do tráfego CAN por meio de um *socket* CAN. Com a placa conectada à internet, deve ser utilizado o comando: ***sudo apt-get install can-utils*** para a sua instalação (Karakurt, 2019).

4.2.4 Experimento A

Neste experimento, a ESP32 atuou como o nó remetente, enviando informações da leitura do sensor acelerômetro e do sensor de temperatura, via rede CAN para o nó da

[2] *Driver mcp251x.c*. Acesso em: 18 out. 2023.
[3] Repositório do *can-utils*. Acesso em: 19 out. 2023.

Raspberry Pi. O objetivo foi avaliar o funcionamento adequado do módulo inserido no *kernel* e a transmissão correta de informações pela ESP32, bem como a leitura correta das informações por parte da *Raspberry*. Para monitorar o tráfego da rede CAN entre os nós, as ferramentas *can-utils* foram utilizadas.

4.3 Integração com o *Android Automotive OS*

A terceira etapa do projeto engloba a integração de todos os componentes e funcionalidades desenvolvidos nas etapas anteriores no ambiente do *Android Automotive* (AAOS). Isso é fundamental, uma vez que o principal objetivo deste trabalho é criar uma aplicação que opere no contexto do AAOS, apresentando os defeitos veiculares aos usuários de forma clara e eficaz.

4.3.1 Preparação do ambiente de desenvolvimento

Para gerar uma imagem do AAOS, é fundamental preparar o ambiente de compilação para o *kernel* e para o AAOS. O *kernel Android* para a Raspberry Pi 4B é conhecido como *Android-RPi Kernel*, e foi utilizada a versão 5.10. A configuração foi conduzida com base no tutorial[4] disponibilizado pelo Portal Embarcados[5], um *blog* brasileiro sobre sistemas embarcados.

Para isso, foi necessária a instalação de vários pacotes e bibliotecas que serão utilizados no momento da compilação. Para a instalação, foi utilizado o seguinte comando: **sudo apt-get install git-core gnupg flex bison build-essential zip curl zlib1g-dev libc6-dev-i386 libncurses5 x11proto-core-dev libx11-dev lib32z1-dev libgl1-mesa-dev libxml2-utils xsltproc unzip fontconfig**

Além disso, para realizar o *download* dos códigos e futuramente conseguir acessar o Android de forma remota, foi necessário baixar e instalar as ferramentas *repo* e *adb*. A ferramenta *repo* é responsável por fazer o controle dos repositórios e dos códigos da Google. Por sua vez, o *adb* é responsável por fornecer acesso remoto ao sistema operacional. Foi utilizado o seguinte comando para *download*: **sudo apt install adb repo**

Após instalar todos os pacotes necessários, foi criado um diretório de trabalho para receber o código-fonte do *kernel*. Foi feita a configuração da pasta com o arquivo de *manifest* que gerencia e aponta todos os repositórios necessários da Google com o seguinte comando: **repo init -u https://github.com/android-rpi/kernel_manifest -b arpi-5.10**. Para baixar o código-fonte

[4] Integrando o barramento CAN ao Android Automotivo 13. Acesso em: 20 dez. 2023.
[5] Portal Embarcados. Acesso em: 20 dez. 2023.

do *kernel*, foi utilizado o seguinte comando: **repo sync**. Esse comando faz o *download* de todos os repositórios descritos no arquivo de *manifest*, previamente configurado (Alencar, 2023).

Em seguida, foi criado um novo diretório, no mesmo nível do diretório do *kernel*, para os arquivos que contém o projeto do AAOS. Semelhante ao que foi feito anteriormente, foi feita a configuração do diretório com o arquivo de *manifest* do AAOS 13 com o seguinte comando: **repo init -u https://android.googlesource.com/platform/manifest -b android-13.0.0_r52**. Além disso, foi necessário fazer o *download* de arquivos necessários para o funcionamento do *Android* na *Raspberry Pi* 4B, com o seguinte comando: **git clone https://github.com/snappautomotive/firmware_rpi-local_manifests .repo/local_manifests -b sa-arpi-13**. Por fim, foi feito o *download* com o comando **repo sync**. Esse processo foi bem demorado, pois são muitos arquivos e repositórios, totalizando cerca de 200 *GB* (Alencar, 2023).

4.3.2 Compilação do kernel, inserção do módulo MCP2515 e geração de imagem do AAOS

O segundo passo consistiu na integração do módulo MCP2515 e do CAN no *kernel* e no *Android*, baseando-se no tutorial[4] disponibilizado em uma comunidade de sistemas embarcados. Esse processo é semelhante ao descrito na subseção 4.2.2 e subseção 4.2.3, com pequenas variações no ambiente de compilação, mas mantendo a abordagem geral de compilação e inserção do módulo no *kernel* (Alencar, 2023).

4.3.2.1 Modificações no kernel Android

Para habilitar o suporte ao MCP2515 no *kernel* foi necessário customizar o arquivo de configuração da compilação **<kernel-dir>/common/build.config.arpi** de acordo com o *patch* presente no Apêndice A (Alencar, 2023).

O Código-fonte 19 é um arquivo *patch* que mostra as alterações sofridas pelo arquivo original. Então, foi adicionado “modules” e “dtbs” ao alvo `MAKE_GOALS` para realizar a compilação dos módulos e arquivos *device tree*. No alvo `FILE` foi adicionado os arquivos de *overlay* e módulos do *kernel* que devem ser salvos após a compilação (Alencar, 2023).

Após a realização das modificações, a compilação do *kernel* foi executada pelo *script* a partir da pasta raíz do *kernel*: **./build/build.sh** (Alencar, 2023).

4.3.2.2 Modificações no AAOS

Após a compilação do *kernel*, os arquivos gerados ficaram disponíveis na pasta **<kernel_dir>/out/arpi-5.10/dist**. Os módulos gerados foram copiados para dentro do arquivos do AAOS, para o diretório **<android_dir>/device/snappautomotive/rpi4_car/can/modules** criado especificamente para os recursos do CAN. Em seguida, na pasta acima, foi criado um arquivo chamado **can.mk**, responsável por inserir os módulos do *kernel* no sistema de compilação e na imagem gerada, mostrado no Código-fonte 16.

No Código-fonte 16, o parâmetro `PRODUCT_COPY_FILES` é usado durante o processo de compilação para especificar quais arquivos devem ser copiados para o sistema de arquivos do AAOS. Cada linha indica o caminho de origem do arquivo seguido pelo destino no sistema, separados por dois pontos. Então, todos os arquivos presentes em `$(LOCAL_PATH)/modules` serão copiados para `$(TARGET_COPY_OUT_VENDOR)/lib/modules/`, local onde normalmente ficam os módulos compilados dos fornecedores (Alencar, 2023).

Em seguida, o arquivo **init.can.rc** foi criado, mostrado no Código-fonte 17. Ele é responsável por inserir os módulos no *kernel* durante a inicialização do sistema, pois o AAOS geralmente não possui alguns *drivers* instalados e ativados. Portanto, foi necessário configurar a inicialização automática desses *drivers* (Alencar, 2023). Esses módulos trabalham em conjunto para garantir que as comunicações SPI e CAN funcionem corretamente no ambiente *Android*.

Para que os módulos sejam identificados e chamados pelo sistema de compilação, foi necessário modificar o arquivo de compilação **device/snappautomotive/rpi4_car/rpi4_car.mk**, adicionando ao cabeçalho a seguinte linha (Alencar, 2023):

Código-fonte 2 – Chamada ao arquivo can.mk pelo sistema de compilação

```
1 $(call inherit-product, device/snappautomotive/rpi4_car/can/can.mk)
```

O **rpi4_car.mk** é o principal arquivo de compilação do sistema *Android* para o dispositivo *Raspberry Pi* 4B. Ele define suas características, especifica quais aplicativos e serviços serão instalados e chama outros arquivos de compilação. Nesse contexto, o arquivo **can.mk** foi adicionado para ser chamado durante a compilação. Ao final do arquivo **rpi4_car.mk**, foi incluído o **init.can.rc**, responsável pela inicialização dos módulos, como mostra o Código-fonte 3 (Alencar, 2023).

Código-fonte 3 – Cópia do arquivo de inicialização dos módulos

```
1 PRODUCT_COPY_FILES += \
2   $(LOCAL_PATH)/can/init.can.rc:$(TARGET_COPY_OUT_VENDOR)/etc/init/hw
      /init.can.rc
```

Em seguida, foi adicionado o arquivo de inicialização dos módulos no arquivo principal de inicialização da placa e do sistema. Ainda na mesma pasta, o arquivo **init.rpi4.rc** foi modificado com o seguinte conteúdo (Alencar, 2023):

Código-fonte 4 – Importando arquivo de inicialização dos módulos

```
1 import /vendor/etc/init/hw/init.can.rc
```

Com a modificação mostrada pelo Código-fonte 4, o arquivo de inicialização dos módulos é chamado durante a inicialização do sistema, carregando os *drivers* necessários para as comunicações CAN e SPI. Por padrão, o sistema Android não permite que módulos do *kernel*, previamente compilados, sejam copiados diretamente para o sistema de arquivos no momento da compilação. Para contornar esse problema encontrado, o arquivo **device/snappautomotive/rpi4_car/BoardConfig.mk** foi modificado, adicionando a seguinte configuração (Alencar, 2023):

Código-fonte 5 – Configuração para permitir a cópia de módulos do *kernel*

```
1 BUILD_BROKEN_ELF_PREBUILT_PRODUCT_COPY_FILES := true
```

Além disso, o arquivo **device/snappautomotive/rpi4_car/boot/config.txt** foi modificado com a mesma configuração mostrado no Código-fonte 1. A modificação foi necessária para que o *kernel Android* conseguisse reconhecer o módulo MCP2515 e o barramento SPI fosse habilitado. Com a configuração mostrada pelo Código-fonte 5, foi possível realizar a compilação do sistema sem erros. Para a geração da imagem AAOS, a partir da raíz do Android, três comandos foram utilizados (Alencar, 2023):

1. Ativar do ambiente de compilação:
   a) **source build/envsetup.sh**
2. Definir o dispositivo alvo:
   a) **lunch rpi4_car-eng**
3. Compilar o sistema:
   a) **make –j$(nproc) ramdisk systemimage vendorimage**

A opção "`-j$(nproc)`" foi utilizada para paralelizar o processo de compilação em todos os núcleos (16) da máquina *host*. Após a execução dos comandos acima, a compilação prosseguiu por cerca de três horas e os arquivos foram gerados na pasta **out/target/product/rpi4_car/**.

Por fim, a gravação das imagens no cartão de 16 GB seguiu o passo a passo mostrado por Alencar (2023) na seção "Configuração do cartão micro SD".

4.3.3 Experimento B

Nesse experimento, foi testada a imagem do *Android*, gerada a partir da compilação descrita na subseção 4.3.2. Para isso, foi conectado o cartão de memória previamente configurado à placa, que foi conectada a um monitor de vídeo por meio da porta micro *High-Definition Multimedia Interface* (HDMI) e, por fim, ligada à uma fonte DC $5\,V@3\,A$.

No experimento, foi avaliado se o sistema é inicializado corretamente, o que deve ser indicado pela interface gráfica no monitor. Além disso, foi avaliado também a interação remota com o *Android* por meio da ferramenta *Android Debug Bridge* (adb) e inicialização dos módulos.

4.3.4 Integração da VHAL com dados da rede CAN

Após o experimento envolvendo a primeira imagem do *Android*, foi realizada a criação de propriedades na VHAL. Essas propriedades guardam os valores de leitura dos sensores. Em seguida, foi desenvolvido um serviço, que captura os dados da rede CAN e escrever esses dados nas propriedades da VHAL.

4.3.4.1 Habilitação e criação de propriedades VHAL

A VHAL foi atualizada no Android 13 para utilização da interface *Android Interface Definition Language* (AIDL), a qual não vem habilitada por padrão com o AAOS. Para habilitá-la, foi necessária a modificação em dois arquivos dentro da pasta **device/snappautomotive/rpi4_car/**. No arquivo **rpi4_car.mk** foi feita a substituição do serviço antigo, `android.hardware.automotive.vehicle@2.0-default-service`, pelo novo serviço da VHAL, o `android.hardware.automotive.vehicle@V1-default-service`.

A VHAL é um tipo de HAL presente no *Android* e seu uso precisa ser indicado por meio de uma *tag* no arquivo **manifest.xml**. Essa *tag* deve especificar o formato AIDL, o nome, a versão e a instância da VHAL, como mostrado na Código-fonte 6.

Código-fonte 6 – Adição do serviço da VHAL no arquivo manifest.xml

```
<hal format="aidl">
    <name>android.hardware.automotive.vehicle</name>
    <version>1</version>
    <fqname>IVehicle/default</fqname>
</hal>
```

Com a habilitação da nova VHAL, foi possível seguir em frente com a criação de novas propriedades para o propósito deste trabalho. A adição e configuração de novas propriedades foi uma tarefa complexa, exigindo a modificação de diversos arquivos. Entretanto, ela foi baseada em um guia[6] disponível no blog Medium por Suresh (2024).

As definição de uma propriedade customizada deve seguir a seguinte fórmula: PROP_A = UNIQUE_ID + VEHICLE_PROPERTY_TYPE_ID + VEHICLE_AREA_ID + VEHICLE_PROPERTY_GROUP_ID, onde (Google, 2024b; Suresh, 2024):

- **UNIQUE_ID**: Um ID único de 16 bits, que deve respeitar o intervalo 0x0100 - 0xFFFF.
- **VEHICLE_PROPERTY_GROUP_ID**: Um ID de 4 bits que determina o grupo da propriedade, SYSTEM ou VENDOR.
- **VEHICLE_AREA_ID**: Um ID de 4 bits que determina o tipo de área do veículo.
- **VEHICLE_PROPERTY_TYPE_ID**: Um ID de 8 bits que representa o tipo de dado.

Baseando-se nisso, foram criadas quatro propriedades para este trabalho, duas para cada sensor utilizado. Uma propriedade guardará o valor de leitura do sensor e a outra propriedade guardará o código de defeito ou *status* do mesmo. Para o sensor de temperatura, foram criadas as seguintes propriedades:

- INFO_TEMPERATURE_DHT22 = 0x1000 + 0x20000000 + 0x01000000 + 0x00600000
- FAULT_CODE_TEMPERATURE_DHT22 = 0x1001 + 0x20000000 + 0x01000000 + 0x00100000

A propriedade INFO_TEMPERATURE_DHT22 é responsável por armazenar o valor da temperatura capturada pelo sensor DHT22. Ela é composta pelo ID único (0x1000), pelo grupo VENDOR (0x20000000), pela área GLOBAL do veículo (0x01000000) e pelo tipo de dado *float* (0x00600000). A propriedade FAULT_CODE_TEMPERATURE_DHT22 armazena o código de *status* ou defeito do sensor. Ela é formada pelo ID único (0x1001), grupo VENDOR, área GLOBAL e tipo de dado *string* (0x00100000).

[6] Adding Custom Vehicle Properties in AOSP 13 AIDL VHAL: Unlocking Customizations. Acesso em: 15 abr. 2024.

Seguindo a mesma lógica, foram criadas as seguintes propriedades para o sensor acelerômetro:

- INFO_ACCELEROMETER_MPU6050 = 0x1002 + 0x20000000 + 0x01000000 + 0x00410000
- FAULT_CODE_ACCELEROMETER_MPU6050 = 0x1003 + 0x20000000 + 0x01000000 + 0x00100000

A propriedade INFO_ACCELEROMETER_MPU6050 é formada pelo ID (0x1002), grupo VENDOR, área GLOBAL e o tipo de dados é um vetor de inteiros (0x00410000). Ela é responsável por guardar o valor bruto capturado pelo acelerômetro nos eixos X, Y e Z. A propriedade FAULT_CODE_ACCELEROMETER_MPU6050 segue a mesma lógica da FAULT_CODE_TEMPERATURE_DHT22, mudando apenas o ID.

Com as propriedades e seus valores organizados na fórmula, foi iniciado o processo de modificação dos arquivos e compilação para o sistema reconhecê-las. Os arquivos responsáveis pelas interfaces automotivas foram modificados, de acordo com a ordem a seguir (Suresh, 2024):

1. **hardware/interfaces/automotive/vehicle/aidl/android/hardware/automotive/vehicle/VehicleProperty.aidl**
2. **hardware/interfaces/automotive/vehicle/aidl/aidl_api/android.hardware.automotive.vehicle/current/android/hardware/automotive/vehicle/VehicleProperty.aidl**
3. **hardware/interfaces/automotive/vehicle/aidl/aidl_api/android.hardware.automotive.vehicle/1/android/hardware/automotive/vehicle/VehicleProperty.aidl**

Os três arquivos acimas são escritos em AIDL, uma linguagem de definição de interface do *Android*, muito útil quando diferentes aplicativos precisam se comunicar entre si (Google, 2024a). O primeiro arquivo foi modificado conforme mostrado no Código-fonte 18. Nele contém as fórmulas e a documentação necessária para cada propriedade, incluindo o modo de acesso, o tipo de atualização ao longo do tempo e a unidade de medida, quando aplicável. Essas informações devem ficar dentro da estrutura *enum VehicleProperty* do arquivo.

O segundo e terceiro arquivo foram, preenchidos com o mesmo conteúdo, como mostrado no arquivo Código-fonte 15. Nele contém o nome da propriedade e o resultado final da fórmula mostrada anteriormente, sendo assim, o número pelo qual a propriedade é identificada e reconhecida pelo AAOS.

Em seguida, as propriedades foram configuradas, definindo o valor inicial, modo de acesso, o tipo de atualização e taxa de amostragem. A modificações foram feitas no arquivo

hardware/interfaces/automotive/vehicle/aidl/impl/default_config/include/DefaultConfig.h. O Código-fonte 20 mostra as configurações para cada propriedade criada. As propriedades INFO_TEMPERATURE_DHT22 e INFO_ACCELEROMETER_MPU6050 foram inicializadas com zero. A propriedade FAULT_CODE_TEMPERATURE_DHT22 foi inicializado com "TMP-0", onde "TMP" se refere ao sensor de temperatura e "0" indica que o sensor está operando corretamente, sem defeitos. A mesma lógica foi empregada para a propriedade do sensor MPU6050.

Após criação e definição das propriedades, a compilação foi realizada, mas sem sucesso. O sistema de compilação gera um valor *hash* para os arquivos AIDL e realiza a comparação com o valor gerado na primeira compilação. Como os arquivos do sistema foram alterados, o valor *hash* foi modificado e a compilação foi interrompida. Para resolver esse problema, foi necessário gerar um novo *hash*. O comando, mostrado no Código-fonte 14, foi disponibilizado por Suresh (2024) na seção "Troubleshooting" de seu artigo. Ao executar o comando, um novo código *hash* foi gerado e copiado para o arquivo **hardware/interfaces/automotive/vehicle/aidl/aidl_api/android.hardware.automotive.vehicle/1/.hash**, dessa forma a comparação não será divergente (Suresh, 2024).

Para incluir as alterações realizadas, o código-fonte do AAOS foi compilado novamente com o comando "**m android.hardware.automotive.vehicle**" (Suresh, 2024). Com o "**m**" é possível compilar todo o *Android* e também módulos isolados.

O comando acima faz a compilação apenas do módulo responsável pelas interfaces e funções veiculares. Após a compilação do módulo, uma nova imagem foi gerada, utilizando apenas o "**m**".

4.3.4.2 Desenvolvimento do serviço CAN2VHAL

Para capturar os dados que chegam até o Android Automotivo via CAN e preencher as propriedades VHAL criadas, foi desenvolvido um serviço em C++ chamado CAN2VHAL. Ele é composto por uma classe SocketCan responsável por realizar toda abstração de configuração e inicialização do *socket*, bem como a leitura e envio de mensagens.

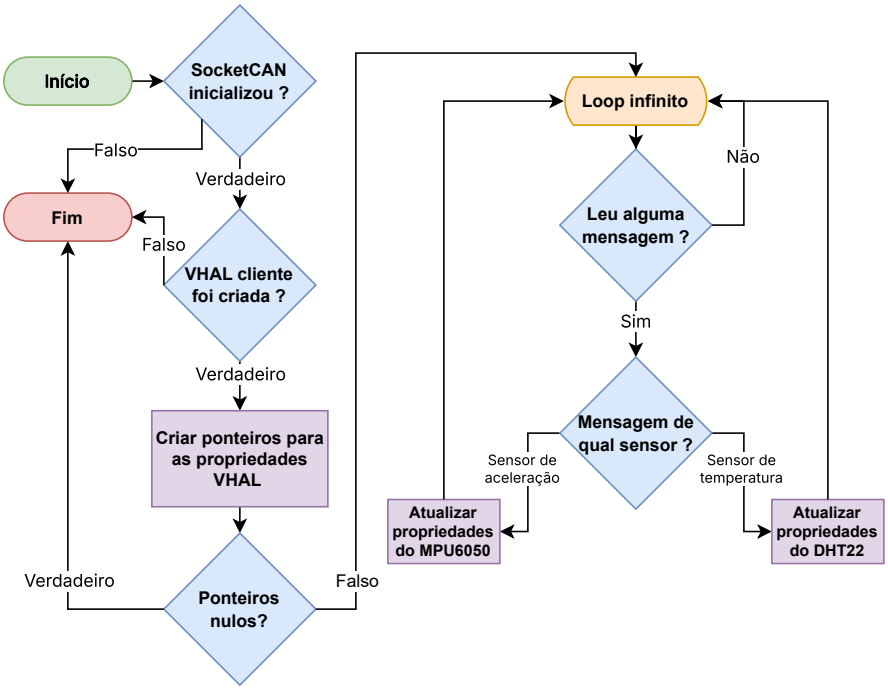
A Figura 19 ilustra os passos realizados pelo serviço desenvolvido. Primeiramente, a classe SocketCan é inicializada, tentando abrir uma conexão *socket* no *kernel*. Caso não seja possível, o programa é encerrado, o que pode indicar que a rede CAN e o MCP2515 não foram corretamente inicializados ou a interface de rede não tenha sido levantada, tornando impossível

capturar as mensagens enviadas pelo nó do microcontrolador.

Se a conexão *socket* for bem-sucedida, uma tentativa de instanciar a classe AidlVhalClient é realizada, possibilitando a interação com a VHAL. Se a instanciação falhar, o programa é encerrado, pois não será possível interagir com a propriedades criadas. Em caso de sucesso, os ponteiros para as propriedades são criados a partir de métodos da AidlVhalClient. Se os ponteiros não forem nulos, o *loop* é inicializado; caso contrário, o programa é encerrado.

Dentro do *loop*, é verificada a chegada de mensagens CAN no *socket* através de *pooling*. Caso tenha chegado uma mensagem, é feita uma checagem para determinar a qual sensor pertence essa mensagem. Em seguida, as propriedades correspondentes são atualizadas e o programa retorna ao início do *loop*.

Figura 19 – Fluxograma do serviço CAN2VHAL



Fonte: Elaborado pelo autor (2024).

4.3.5 Desenvolvimento do sistema de informação e entretenimento veicular

O quarto passo foi composto pelo desenvolvimento do aplicativo final, que proporciona interação com os usuários. É um sistema de informação e entretenimento veicular que tem a capacidade de identificar defeitos veiculares, apresentando-os aos ocupantes do veículo de maneira clara, objetiva e de fácil compreensão.

4.3.5.1 Modificações na camada de framework

Após a definição e configuração da propriedades criadas, é preciso torná-las acessíveis ao aplicativo desenvolvido. Para isso, foi necessário fazer algumas modificações na Car API (Suresh, 2024).

Na primeira modificação, foi alterado o nível de proteção da permissão do grupo VENDOR. No arquivo **packages/services/Car/service/AndroidManifest.xml** a permissão CAR_VENDOR_EXTENSION é definida entre as linhas 206 e 209. O nível de proteção, definido na linha 207, foi alterado para "normal". Essa mudança foi feita para evitar problemas de permissão na escrita ou leitura das propriedades pelo aplicativo, pois cada propriedade da VHAL está vinculada a uma permissão. (Suresh, 2024).

Na segunda modificação, foi atribuída a permissão CAR_VENDOR_EXTENSION para cada propriedade criada. O conteúdo do Código-fonte 21 foi adicionado na classe do arquivo **packages/services/Car/car-lib/src/android/car/VehiclePropertyIds.java** (Suresh, 2024).

Após a adição das novas propriedades na camada de *framework*, o *Android* foi recompilado, para que as mudanças sejam refletidas na imagem. Primeiro, a compilação do Car API foi feita no diretório **packages/services/Car/car-lib** usando o comando "**mm**", que compila e instala todos os módulos presentes no diretório. Em seguida, a compilação de todo o código-fonte foi realizada a partir do diretório raiz com o comando "**m**". (Suresh, 2024).

Após a compilação, a imagem antiga foi atualizada remotamente. Com a placa conectada à rede Wi-Fi, o acesso remoto foi estabelecido via adb usando o comando **adb connect <ip>:5555**. O usuário *root* foi ativado com **adb root**. Com as variáveis de ambiente carregadas e o dispositivo alvo selecionado, os comandos **adb remount** e **adb sync** foram executados no diretório **out/**. O comando **adb remount** configurou a partição */system* para o modo de escrita, enquanto **adb sync** sincronizou os arquivos compilados com os do AAOS no cartão SD.

4.3.5.2 Definição dos requisitos e prototipação das telas do aplicativo

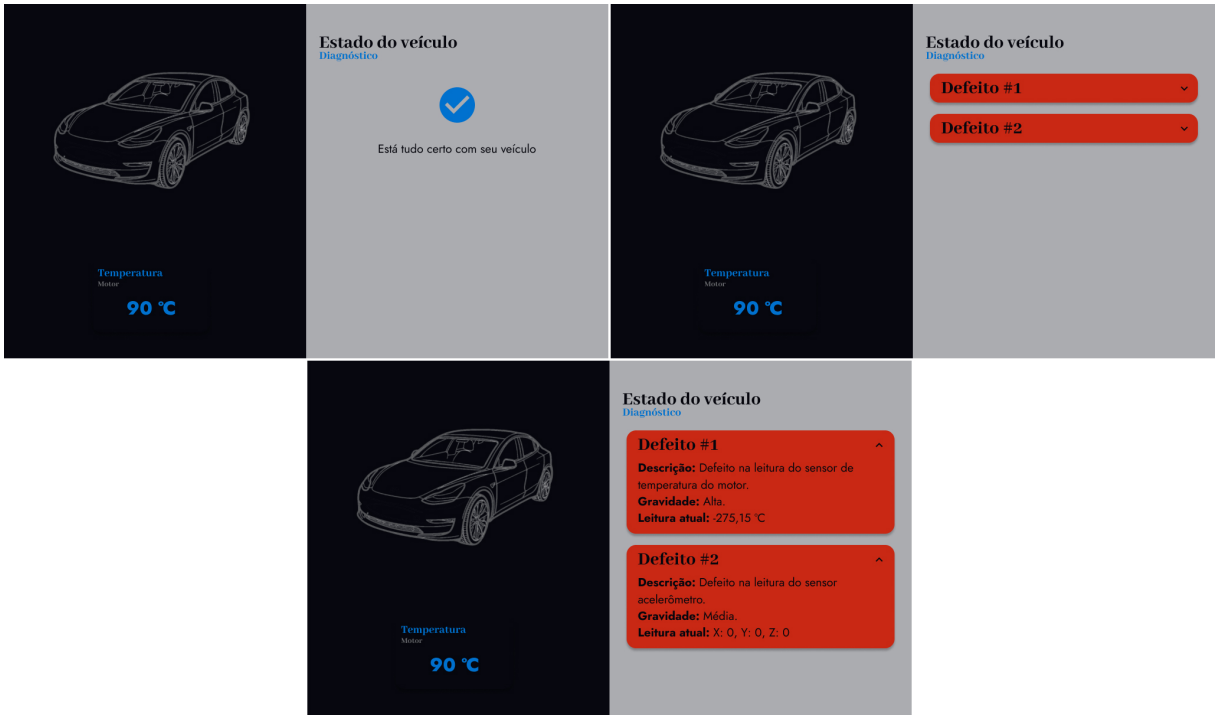
Para o desenvolvimento do aplicativo, foi definido um conjunto de requisitos funcionais e não funcionais que são fundamentais para atingir o objetivo geral deste estudo. Com os requisitos bem definidos, a prototipação das telas do aplicativo foi realizada. Para essa tarefa, foi utilizada a ferramenta Figma[7], bastante conhecida e amplamente usada por desenvolvedores e

[7] Figma: a ferramenta de design de interface colaborativa. Acesso em: 02 fev. 2024

designers UI/UX para construir o *design* da interface de sites ou aplicativos móveis.

A Figura 20 ilustra o *design* inicial das telas do aplicativo. Para construir o protótipo dessas interfaces, foram consultados e utilizados como inspiração protótipos de aplicativos para carros elétricos da comunidade[8] do Figma. Elementos como cores, estilos de fontes, cards, botões e a estrutura de organização foram adaptados ao contexto da aplicação, devido à falta de conhecimento e experiência em *design* de telas e no uso da ferramenta.

Figura 20 – Design inicial do sistema de informação e entretenimento veicular



Fonte: Elaborado pelo autor (2024).

A ideia central do *design* mostrado pela Figura 20 é uma divisão vertical da tela em dois. O lado esquerdo é responsável por mostrar informações gerais do veículo e do sensor de temperatura, através de uma imagem do veículo e de um *card* que mostra a temperatura do motor. O lado direito foi designado para o diagnóstico do veículo, mostrando os defeitos detectados por meio de *cards*, que com o toque se expandem e mostram mais informações acerca do defeito, como descrição, gravidade e leitura atual.

4.3.5.3 Desenvolvimento da aplicação

O aplicativo foi desenvolvido em Kotlin com o *framework* Jetpack Compose, amplamente utilizados no desenvolvimento nativo de aplicativos *Android*. Para o desenvolvimento,

[8] Figma Community. Acesso em: 02 fev. 2024

foi utilizado o *Android Studio*[9] (versão Hedgehog | 2023.1.1), um ambiente de desenvolvimento integrado específico para desenvolver aplicativos para a plataforma *Android*. Após a conclusão do desenvolvimento, o aplicativo foi implantado no *hardware* de destino, seguindo para testes e a avaliação para garantir o funcionamento esperado.

4.3.6 Experimento C

O experimento C teve como objetivo validar o serviço CAN2VHAL e aplicativo implementado no AAOS. No cenário de teste, foi avaliado se o serviço recebe corretamente as mensagens CAN, preenche as propriedades da VHAL de maneira adequada e se o aplicativo se comporta como o esperado.

4.4 Avaliação do sistema de informação e entretenimento

A quarta etapa do projeto consistiu na execução da avaliação de usabilidade do aplicativo. Nesse estágio, foi avaliada a capacidade do sistema de apresentar os defeitos de forma compreensível para o público-alvo. O sistema desenvolvido integra apenas dois sensores, totalizando quatro cenários distintos de avaliação, cada um provocando uma modificação específica na interface do aplicativo. Os cenários são descritos da seguinte forma:

- **Primeiro cenário**: Não há defeitos, e ambos os sensores operam normalmente.
- **Segundo cenário**: Ocorre um defeito no sensor de temperatura.
- **Terceiro cenário**: O sensor de aceleração apresenta defeito.
- **Quarto cenário**: Ambos os sensores estão com defeitos.

4.4.1 Teste de usabilidade

Finalizado o desenvolvimento do aplicativo, foi iniciada a avaliação do mesmo, com o objetivo de verificar a comunicação de defeitos, eficácia e a opinião dos usuários sobre a aplicação, sua utilidade e nível de satisfação.

A avaliação de IHC pode ser feita através de métodos de investigação, inspeção ou observação. Para a avaliação do aplicativo desenvolvido neste estudo, foi escolhida a avaliação por observação. Esse método permite ao avaliador coletar dados sobre as situações em que os participantes realizam atividades, com ou sem o apoio de tecnologia computacional. Com a aná-

[9] Android Studio. Acesso em: 22 jan. 2024

lise dos dados registrados, é possível identificar problemas reais enfrentados pelos participantes, e não apenas problemas potenciais previstos pelo avaliador, como ocorre em uma avaliação por inspeção. (Barbosa *et al.*, 2021).

A avaliação através da observação possui três métodos: o teste de usabilidade, o Método de Avaliação de Comunicabilidade (MAC) e prototipação em papel (Barbosa *et al.*, 2021). O primeiro método foi escolhido, pois é possível avaliar a usabilidade de um sistema a partir de atividades e experiências dos seus usuários-alvo (Rubin *et al.*, 2008). Os objetivos da avaliação determinam quais critérios de usabilidade devem ser medidos, geralmente através de perguntas específicas e dados mensuráveis (Barbosa *et al.*, 2021). Para avaliar se a comunicação dos defeitos foi clara e objetiva, pode-se questionar: "Quantos usuários conseguiram responder com sucesso as tarefas?" e "Quanto tempo os usuários levaram para cumprir determinada tarefa?" (Barbosa *et al.*, 2021).

De acordo com Nielsen (2000), com uma amostra de cinco participantes é possível encontrar 80% dos problemas de uma interface. Quanto mais usuários são adicionados, menos é aprendido, pois os problemas começam a se repetir. Nesse contexto, cinco moradores do distrito de Flores, localizado no município de Russas, no estado do Ceará, que repeitassem os critérios: não possuir formação na área de tecnologia, possuir Carteira Nacional de Habilitação (CNH) e possuir experiência em direção de veículos, foram aleatoriamente convidados a participar.

A Tabela 2 apresenta as atividades realizadas durante todo o período a avaliação, desde a preparação até o relato dos resultados (Barbosa *et al.*, 2021).

Tabela 2 – Atividades para avaliação da interface do sistema

Atividade	**Tarefa**
Preparação	• Definir tarefas para os participantes executarem • Definir termo de consentimento e roteiros das entrevistas • Preparar material para observar e registrar o uso • Executar um teste-piloto
Coleta de dados	• Apresentar o termo de consentimento e entrevista pré-teste • Observar e registrar a performance do participante • Gravar o vídeo da interação de cada participante • Entrevista pós-teste para colhimento de opiniões e níveis de satisfação
Interpretação	• Reunir, contabilizar e sumarizar os dados coletados dos participantes individualmente
Relato dos resultados	• Relatar a performance e a opinião dos participantes

Fonte: Elaborada pelo autor baseado em (Barbosa *et al.*, 2021).

A atividade de *preparação* contou com a definição das tarefas que os participantes executariam durante a avaliação, presentes no Apêndice D. Com elas, foi possível avaliar

se o usuário compreendeu a natureza dos defeitos, quanto tempo levou e quais ações tomou para alcançar essa compreensão. Além disso, a meta de usabilidade estabelecida é que cada participante leve no máximo 40 segundos para concluir cada tarefa. Também foram formulados o termo de consentimento (Apêndice B) e os roteiros para as entrevistas pré-teste (Apêndice C) e pós-teste (Apêndice E).

Finalizando a preparação, foi realizado um teste-piloto para identificar erros, difícil compreensão das perguntas, dúvidas ou qualquer ambiguidade existente nas entrevistas. Após a realização do teste, foram feitas pequenas modificações nas perguntas e atividades.

Na atividade de *coleta de dados*, a avaliação foi iniciada com a apresentação do termo de consentimento, cujo aceite do usuário foi registrado em áudio. Em seguida, foi realizada uma entrevista pré-teste para identificar o perfil dos participantes através de perguntas rápidas.

Para simular os defeitos em cada cenário, os sensores foram desconectados, impedindo assim a sua leitura pelo microcontrolador. Após a configuração de cada cenário, o participante pôde interagir por cerca de 10 segundos com o aplicativo. Em seguida, o avaliador conduziu o participante através das atividades correspondentes, fazendo perguntas relacionadas com a interface e como ela variava de acordo com o estado dos sensores, conforme detalhado no Apêndice D.

O áudio e o uso do aplicativo por cada participante foram gravados em cada atividade realizada. Ao fim das atividades, foi realizada a entrevista pós-teste para coleta de opinião dos usuários sobre a aplicação, sua utilidade e nível de satisfação. Cada entrevista durou em média 15 minutos.

Na *interpretação* e *consolidação dos resultados*, o avaliador analisou os dados coletados com o objetivo de medir as métricas definidas e contabilizar os erros cometidos pelos participantes em cada atividade.

5 RESULTADOS

Neste capítulo são apresentados os resultados deste trabalho. Os dados apresentados foram coletados a partir de compilações do *kernel* e do AAOS, além de experimentos realizados, prototipagem, desenvolvimento de *software* e um teste de usabilidade.

5.1 Construção da rede CAN

Nesta seção são mostrados os resultados obtidos na construção da rede CAN, como a definição de requisitos, o *software* de comunicação e o desenvolvimento do protótipo da rede.

5.1.1 Requisitos e software de comunicação com a rede CAN

Foram definidos alguns requisitos com o objetivo de orientar o desenvolvimento das funcionalidades e para garantir a qualidade do sistema. O Quadro 7 mostra os requisitos definidos.

Quadro 7 – Requisitos funcionais e não funcionais para o *software* de comunicação com a rede CAN

Requisitos funcionais	**RF1:** Deve enviar mensagens CAN para outro nó. **Prioridade:** Essencial.
	RF2: Deve ler as mensagens CAN recebidas. **Prioridade:** Essencial.
	RF3: Deve adquirir dados dos sensores. **Prioridade:** Importante.
Requisitos não funcionais	**RNF1:** A comunicação deve ser robusta e confiável.
	RNF2: Deve garantir compatibilidade entre as mensagens enviadas.
	RNF3: Deve garantir uma comunicação eficaz.
	RNF4: Deve ter baixa latência na transmissão de dados.
	RNF5: A rede deve ser escalável.

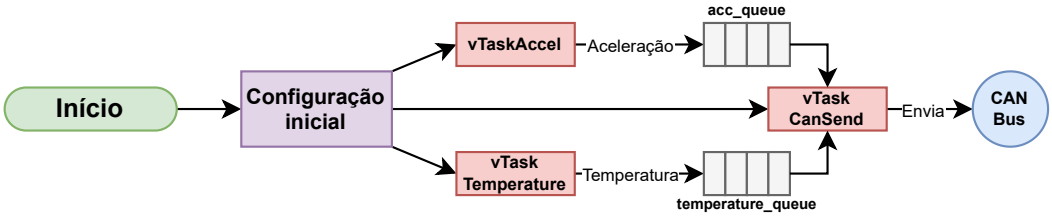
Fonte: Elaborado pelo autor (2023).

Os requisitos funcionais descrevem as funcionalidades que o *software* deve oferecer para satisfazer as necessidades em uma comunicação via rede CAN. Por sua vez, os requisitos não funcionais descrevem características globais para que o *software* se torne realidade.

A Figura 21 ilustra o fluxograma da aplicação de comunicação com a rede CAN para o ESP32, desenvolvida no contexto do FreeRTOS. Na configuração inicial, as *tasks* e filas são criadas, utilizando três *tasks* e duas filas. A `vTaskAccel` lê os dados do sensor de aceleração e preenche a fila `acc_queue`. A `vTaskTemperature` captura os dados de temperatura e preenche a fila `temperature_queue`. A `vTaskCanSend` consome ambas as filas, prepara as mensagens

e as envia ao barramento. Dessa forma, cada *task* possui uma única responsabilidade e são executadas a cada 50 ms.

Figura 21 – Fluxograma do *software* de comunicação no ESP32

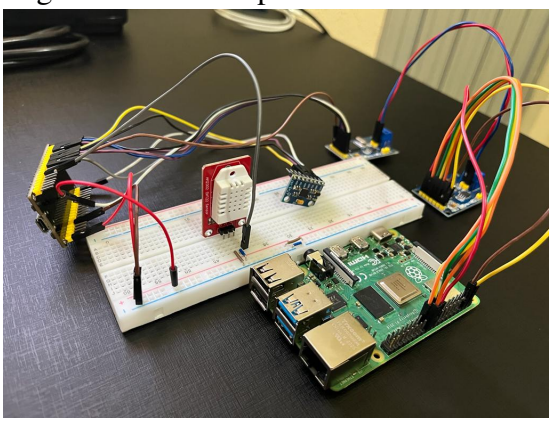


Fonte: Elaborada pelo autor (2024).

5.1.2 Protótipo da rede CAN

O protótipo da rede CAN, representado pela Figura 22, foi realizado com base no desenho mostrado pela Figura 18 e nas conexões detalhadas pelos Quadros 3, 4, 5 e 6. Com o protótipo montado, foi possível estabelecer a comunicação entre os nós. Sendo o nó formado pela ESP32, sensor de temperatura e aceleração, responsável por enviar informações até o nó formado pela *Raspberry Pi* 4B.

Figura 22 – Protótipo final da rede CAN



Fonte: Acervo pessoal do autor (2023).

5.2 Integração com o *kernel Linux*

Nesta seção, são apresentados os resultados obtidos na integração com o *Linux*, compilação e inserção de módulo, assim como a execução do experimento A.

5.2.1 *Compilação do módulo do kernel para o MCP2515*

A compilação do módulo foi realizada dentro do sistema de arquivos da *Raspberry*. Foi criado um diretório contendo o arquivo *mcp251x.c* e o arquivo *Makefile* com o seguinte conteúdo:

Código-fonte 7 – *Makefile* para compilar o módulo

```
1 obj-m+=mcp251x.o
2 all:
3   make -C /lib/modules/$(shell uname -r)/build M=$(shell pwd) modules
```

Arquivos do tipo *Makefile* são compostos por *targets*, dependências e o que será executado. A linha "obj-m+=mcp251x.o" especifica o nome do módulo do *kernel* que está sendo compilado e adiciona na variável de sistema do *kernel*. O *target "all:"* consiste na compilação do módulo presente no diretório a partir dos arquivos *kernel* presente no sistema da placa. Após a criação do arquivo, foi executado o comando ***make*** no terminal.

Figura 23 – Resultado da compilação

```
ubuntu@ubuntu:~$ tree mcp251x/
mcp251x/
├── Makefile
├── Module.symvers
├── mcp251x.c
├── mcp251x.ko
├── mcp251x.mod
├── mcp251x.mod.c
├── mcp251x.mod.o
├── mcp251x.o
└── modules.order
```

Fonte: Elaborada pelo autor (2023).

A Figura 23 mostra os arquivos gerados pela compilação realizada a partir do *Makefile*. A presença do arquivo *mcp251x.ko* mostra que o módulo foi compilado com sucesso.

5.2.2 *Inserção do módulo no kernel e implementação de um nó CAN*

Com o módulo conectado à placa de acordo com Quadro 3 e o arquivo de inicialização do *kernel* com as modificações mostradas na subseção 4.2.3, o módulo foi inserido dentro do *kernel* utilizando o seguinte comando: ***insmod mcp251x.ko***. O resultado dessa inserção pode ser observado na Figura 24, que mostra algumas mensagens do *kernel*. Na última mensagem é mostrado que o módulo MCP2515 foi inicializado com sucesso.

Figura 24 – Mensagens do *kernel* após a inserção do módulo

```
[    8.709446] CAN device driver interface
[    8.738787] videodev: Linux video capture interface: v2.00
[    8.747824] snd_bcm2835: module is from the staging directory, the q
[    8.770263] bcm2835_audio bcm2835_audio: card created with 8 channel
[    8.887315] mcp251x spi0.0 can0: MCP2515 successfully initialized.
```

Fonte: Elaborada pelo autor (2023).

Após a inserção do módulo, o CAN foi reconhecido como uma interface de rede (*can0*). Sua configuração foi realizada através do comando: "**sudo ip link set can0 up type can bitrate 500000**". Com este comando, a interface *can0* é levantada e configurada com a taxa de transmissão em 500 *kpbs*. Dessa forma, a *Raspberry Pi 4B* pode ler e enviar mensagens CAN, tornando-se mais um nó na rede.

5.2.3 Resultados do Experimento A

O experimento A consistiu em avaliar e validar a transmissão de mensagens entre os dois nós, o funcionamento adequado do módulo inserido no *kernel* e leitura correta das mensagens pela *Raspberry*.

A Figura 25 mostra as mensagens que estão sendo enviadas pelo microcontrolador via CAN. A mensagem com ID $0x123$ corresponde aos dados do acelerômetro e a mensagem com ID $0x124$ corresponde ao dado de temperatura lido pelo sensor.

Figura 25 – Envio de mensagens CAN pela ESP32

```
--------SEND ACCELERATION DATA--------
ID_SEND = 0x123 | DLC_SEND = 6 | X_SEND = -15632 | Y_SEND = -2408 | Z_SEND = -664
--------SEND TEMPERATURE DATA--------
ID_SEND = 0x124 | DLC_SEND = 4 | TEMP_SEND = 32 ºC
--------SEND ACCELERATION DATA--------
ID_SEND = 0x123 | DLC_SEND = 6 | X_SEND = -15512 | Y_SEND = -2488 | Z_SEND = -840
--------SEND TEMPERATURE DATA--------
ID_SEND = 0x124 | DLC_SEND = 4 | TEMP_SEND = 32.1 ºC
--------SEND ACCELERATION DATA--------
ID_SEND = 0x123 | DLC_SEND = 6 | X_SEND = -15656 | Y_SEND = -2540 | Z_SEND = -724
--------SEND TEMPERATURE DATA--------
ID_SEND = 0x124 | DLC_SEND = 4 | TEMP_SEND = 32 ºC
```

Fonte: Elaborada pelo autor (2023).

Com o MCP2515 conectado, o módulo inserido no *kernel* e a interface de rede *can0* configurada, foi possível monitorar o tráfego da rede no *Raspberry Pi* 4B. Para isso, foi utilizado o binário *candump*, presente no conjunto de ferramentas *can-utils*, da seguinte forma: ***candump -c can0***. O argumento "-c" habilita o modo cor e o argumento "can0" indica a interface a ser monitorada.

A Figura 26 exibe o tráfego do barramento CAN capturado. Cada linha exibe uma mensagem CAN na seguinte ordem: o nome da interface monitorada, o identificador da mensagem, o tamanho dos dados (DLC) em *bytes* e o dado transmitido em formato hexadecimal.

Figura 26 – Dados monitorados na interface *can0*

```
ubuntu@ubuntu:~$ candump -c can0
  can0  123   [6]  C4 48 F4 18 FF 88
  can0  124   [4]  9A 99 03 42
  can0  123   [6]  C4 08 F4 24 FF 98
  can0  124   [4]  33 33 03 42
  can0  123   [6]  C4 B0 F4 24 FF 68
  can0  124   [4]  9A 99 03 42
  can0  123   [6]  C3 DC F4 0C FF 10
  can0  124   [4]  33 33 03 42
  can0  123   [6]  C3 CC F4 24 FE AC
  can0  124   [4]  33 33 03 42
  can0  123   [6]  C4 6C F3 E8 FF A8
  can0  124   [4]  33 33 03 42
  can0  123   [6]  C4 34 F4 10 FF 70
  can0  124   [4]  9A 99 03 42
  can0  123   [6]  C4 38 F4 2C FF D4
  can0  124   [4]  33 33 03 42
  can0  123   [6]  C4 80 F4 34 FF 14
```

Fonte: Elaborada pelo autor (2023).

No tráfego da Figura 26 é possível notar a mensagem de ID 0x123 e tamanho de seis *bytes*, que corresponde a mensagem do acelerômetro. A aceleração de cada eixo é definida por 16 bits (2 bytes), mas a mensagem CAN exige armazenamento em um *array* de 8 *bytes*. Portanto, a aceleração é dividida em duas partes de 1 *byte* cada, com o primeiro par de *bytes* representando o eixo X, o segundo o eixo Y e o último o eixo Z. A mensagem de ID 0x124 e tamanho de 4 *bytes* corresponde ao valor de temperatura obtido através do sensor. Por se tratar de um ponto flutuante, foi necessário converter o valor em *bytes* para fazer o envio.

Apesar dos dados estarem em formato hexadecimal, é possível perceber que o ID e o tamanho dos dados recebidos são iguais aos enviados pelo ESP32, conforme mostrado na Figura 25. Com isso, o experimento atingiu o seu objetivo ao demonstrar que a comunicação dos dois nós foi realizada com sucesso.

5.3 Integração com o *Android Automotive OS*

Nesta seção, são detalhados os resultados obtidos na compilação e modificação do código-fonte do *kernel* e do *Android Automotive*.

5.3.1 Preparação do ambiente de desenvolvimento

Para gerar imagens do AAOS vários passos foram executados. O primeiro deles foi preparar o ambiente de compilação, realizando o *download* dos códigos-fonte do *kernel* e do projeto *Android*, além de pacotes necessários para realizar a compilação para outras arquiteturas e plataformas.

A Figura 27 ilustra o ambiente de compilação do *kernel*, gerado pelo comando **repo sync**, contendo todos os arquivos necessários para a compilação. O diretório *commom* contém todo o código do *kernel* e arquivos de configuração do processo de compilação, as demais pastas contém outros arquivos de configuração, *scripts* e os compiladores.

Figura 27 – Ambiente de compilação do *kernel Android*

```
build          common          hikey-modules  out        prebuilts-master
build.config   common-modules  kernel         prebuilts  tools
```

Fonte: Elaborada pelo autor (2024).

A Figura 28 ilustra o ambiente de compilação do *Android*, resultado gerado pelo comando **repo sync**. Por se tratar de um sistema operacional completo, contém mais arquivos e pastas que o ambiente do *kernel*.

Figura 28 – Ambiente de compilação do projeto *Android* Automotivo

```
Android.bp      BUILD        external         out              system
art             cts          frameworks       packages         test
bionic          dalvik       hardware         pdk              toolchain
bootable        developers   kernel           platform_testing tools
bootstrap.bash  development  libcore          prebuilts        vendor
build           device       libnativehelper  sdk              WORKSPACE
```

Fonte: Elaborada pelo autor (2024).

Os resultados mostrados pelas Figura 27 e Figura 28 mostram que a preparação do ambiente de compilação e *download* dos códigos-fontes tanto do *kernel* quanto do AAOS foram realizados com sucesso.

5.3.2 Compilação do kernel, inserção do módulo MCP2515 e geração de imagem do AAOS

Nesta subseção, é discutido os resultados obtidos na compilação do *kernel*, módulos e geração da primeira imagem do *Android* Automotivo.

5.3.2.1 Modificações no kernel Android

A compilação do *kernel* consistiu em uma compilação cruzada para arquitetura ARM x64 da *Raspberry Pi 4B*. Além disso, contou com algumas modificações no arquivos de configuração de compilação.

A Figura 29 mostra a pasta **out/arpi-5.10/dist**, gerada pela compilação, contendo os arquivos necessários para a inicialização e configuração do *kernel Android*. O arquivo `Image.gz`, sinalizado em vermelho, é uma imagem binária do *kernel* que realiza a inicialização do *kernel* e deve ser gravada no cartão de memória.

Figura 29 – Arquivos gerados pela compilação do *kernel*

```
→  ls out/arpi-5.10/dist
abi.prop              can-raw.ko                mcp251x.ko
bcm2711-rpi-400.dtb   Image.gz                  spi-bcm2835.ko
bcm2711-rpi-4-b.dtb   kernel-headers.tar.gz     test_mappings.zip
can-dev.ko            kernel-uapi-headers.tar.gz vc4-kms-v3d-pi4.dtbo
can.ko                mcp2515-can0.dtbo
```

Fonte: Elaborada pelo autor (2024).

Além disso, os módulos do *kernel* – arquivos com extensão ".ko" – também foram compilados e gerados devido as modificações feitas no arquivo de compilação na subseção 4.3.2.1. Esses módulos são *drivers* que podem ser carregados dinamicamente, sem a necessidade de reiniciar o sistema. Os módulos *can-dev.ko, can-raw.ko* e *can.ko* adicionam suporte para rede CAN. Os módulos *mcp251x.ko* e *spi-bcm2835* permitem o reconhecimento do MCP2515 e da comunicação SPI, respectivamente.

Por fim, foram gerados arquivos *device tree*, *bcm2711-rpi-400.dtb* e *bcm2711-rpi-4-b.dtb*, que descrevem o *hardware* da *Raspberry Pi* 4B para o *kernel*. Os *overlays mcp2515-can0.dtbo* e *vc4-kms-v3d-pi4.dtbo* também foram gerados, responsáveis por habilitar o controlador MCP2515 na interface CAN0 e permitir o suporte gráfico da placa, respectivamente.

5.3.2.2 Modificações no AAOS

A compilação do AAOS foi um sucesso, gerando todos os arquivos necessários para sua inicialização no diretório **out/target/product/rpi4_car**, como ilustrado na Figura 30.

Figura 30 – Arquivos gerados pela compilação do Android Automtoivo

```
➜  ls out/target/product/rpi4_car/
android-info.txt       installed-files.json          obj
apex                   installed-files-ramdisk.json  previous_build_config.mk
appcompat              installed-files-ramdisk.txt   ramdisk
build_fingerprint.txt  installed-files-root.json     ramdisk.img
build_thumbprint.txt   installed-files-root.txt      root
clean_steps.mk         installed-files.txt           symbols
data                   installed-files-vendor.json   system
debug_ramdisk          installed-files-vendor.txt    system.img
dexpreopt_config       misc_info.txt                 userdata.img
fake_packages          module-info.json              vendor
gen                    module-info.json.rsp          vendor.img
```

Fonte: Elaborada pelo autor (2024).

Os arquivos *ramdisk.img, system.img* e *vendor.img* são partes essenciais do *Android*. A imagem *ramdisk.img* contém *scripts* de inicialização, *drivers* e bibliotecas essenciais. Ela é chamada pelo *kernel* durante o *boot* para montar as partições principais. A imagem *system.img* contém a maior parte do AAOS, é onde são armazenados aplicativos, *frameworks* e binários essenciais do sistema. Durante o *boot* todo o seu conteúdo é montado na partição **/system**. Na imagem *vendor.img* contém binários, *drivers* e outros componentes do fornecedor necessários para *hardwares* específicos. Durante a inicialização seu conteúdo é montado em **/vendor**.

Após a compilação, o diretório **vendor/** contém o resultado de algumas modificações realizadas no sistema para suportar a rede CAN. Na subseção 4.3.2.2 os módulos *kernel* foram adicionados no processo de geração de imagem, bem como o arquivo de inicialização deles.

A Figura 31 exibe o diretório **vendor/lib/modules/**, contendo os módulos do *kernel* compilados anteriormente. Isso confirma que eles foram devidamente copiados para o sistemas de arquivo do AAOS e no caminho especificado pelo arquivo `can.mk`, durante sua compilação.

Figura 31 – Módulos do *kernel* copiados para o Android Automotivo

```
samuel in target/product/rpi4_car
➜  ls vendor/lib/modules/
can-dev.ko  can.ko  can-raw.ko  mcp251x.ko  spi-bcm2835.ko
```

Fonte: Elaborada pelo autor (2024).

A Figura 32 exibe o diretório **vendor/etc/init/hw/**, contendo o arquivo `init.can.rc` responsável pela inicialização dos módulos. Isso confirma que o arquivo também entrou no processo de compilação e foi copiado para o *Android*. Dessa forma, os módulos do *kernel* e o arquivo que os inicializa, estão disponíveis e visíveis pelo sistema operacional.

Figura 32 – Arquivo de inicialização dos módulos

```
samuel in target/product/rpi4_car
→  ls vendor/etc/init/hw/
init.can.rc  init.rpi4.rc  init.rpi4.usb.rc
```

Fonte: Elaborada pelo autor (2024).

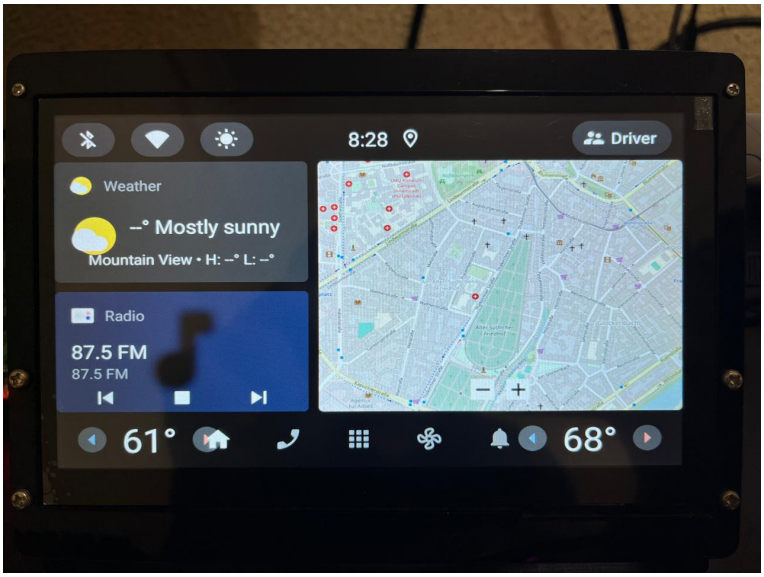
5.3.3 Resultados do experimento B

O experimento B consistiu em avaliar a imagem do AAOS gerada a partir das customizações feitas. O experimento foi realizado na *Raspberry Pi* 4B, com auxílio de um *display* LCD sensível ao toque de sete polegadas, com resolução de 1024x600. Entretanto, é possível conectar teclado e mouse nas portas USBs da placa para interagir com o sistema.

A Figura 33 mostra a tela inicial do AAOS sendo exibida pelo *display* LCD. A interface do *Android* Automotivo é bastante semelhante a dos dispositivos móveis *Android*, ela mantém a navegação e configurações familiares, facilitando a adaptação dos usuários.

Com o sistema inicializado, foi realizada a conexão com a internet via Wi-Fi para possibilitar o acesso remoto via adb. Com o endereço IP da placa disponível e visível nas configurações de rede, um computador na mesma rede pode se conectar com ela através do comando: **adb connect <ip-da-placa>:5555**.

Figura 33 – Interface gráfica do Android Automotivo



Fonte: Acervo pessoal do autor (2024).

A Figura 34 mostra a conexão realizada entre o computador e a placa através do adb. Após a conexão, é possível acessar o sistema de arquivos do AAOS ou executar comandos usando o comando **adb shell**.

Figura 34 – Conexão com adb e execução de comandos

```
samuel in ~
➜  adb connect 192.168.18.77:5555
* daemon not running; starting now at tcp:5037
* daemon started successfully
connected to 192.168.18.77:5555
samuel in ~ took 3,1s
➜  adb shell dmesg | grep mcp
[    8.838659] mcp251x spi0.0 can0: MCP2515 successfully initialized.
[   77.991832] Modules linked in: mcp251x can_dev can_raw can spi_bcm2835
```

Fonte: Elaborada pelo autor (2024).

A Figura 34 também mostra a executação do comando **adb shell dmesg | grep mcp**, utilizado para capturar as mensagens do *kernel* e filtrar o resultado com a palavra "mcp". O comando retornou duas linhas: a primeira indica que o módulo MCP2515 foi inicializado com sucesso e a segunda confirma que os módulos compilados anteriormente também foram inicializados. Isso demonstra que o arquivo de inicialização foi corretamente executado e os módulos foram reconhecidos pelo sistema.

Com isso, o experimento B obteve o resultado esperado, demonstrando que o sistema inicializou corretamente e que sua interface gráfica foi exibida no *display*, mostrou que foi possível acessar a placa remotamente com o adb e que os módulos foram inicializados corretamente.

5.3.4 Integração da VHAL com dados da rede CAN

Nesta subseção, são mostrados os resultado alcançados na habilição e criação de propriedades VHAL, bem como o desenvolvimento do serviço CAN2VHAL.

5.3.4.1 Habilitação e criação de propriedades VHAL

A habilitação da VHAL consistiu em duas modificações para incluir a versão em AIDL na imagem do *Android* Automotivo. Primeiramente, foi realizada a substituição do serviço antigo para o novo serviço com interface em AIDL. Também foi descrito em um arquivo de *manifest* a HAL usada, sua instância e formato.

A Figura 35 mostra os *logs* do sistema filtrados pela palavra "VHAL". Nos *logs*, é indicado que os *scripts* de inicialização da VHAL, "VHAL.earlyInit" e "VHAL.init", foram executados em 30 ms e 290 ms, respectivamente. Isso confirma que as modificações para habilitar a nova versão da VHAL em AIDL foram bem-sucedidas e ela iniciou corretamente.

Figura 35 – Vehicle HAL inicializada

```
130|rpi4_car:/ # logcat -b all | grep VHAL
01-15 19:59:07.554   583   583 V CAR.InitTiming: VHAL.earlyInit took to complete: 30ms
01-15 19:59:09.261   583   583 I CAR.VmsHalService: Initializing VmsHalService VHAL property
01-15 19:59:09.438   583   583 V CAR.InitTiming: VHAL.init took to complete: 290ms
```

Fonte: Elaborada pelo autor (2024).

A criação de propriedades exigiu a definição no código-fonte e a configuração de suas respectivas interfaces. As propriedades foram integradas ao sistema, permitindo que informações fossem acessadas e gerenciadas através da VHAL.

A Figura 36 mostra o resultado da ferramenta *dumpsys*, que fornece informações sobre serviços do sistema. Ao passar o serviço da VHAL e os identificadores como argumento, a ferramenta retornou informações acerca das propriedades INFO_TEMPERATURE_DHT22 (ID 559943680) e FAULT_CODE_TEMPERATURE_DHT22 (ID 554700801).

Figura 36 – Visualização das propriedades do sensor de temperatura

```
→  adb shell dumpsys android.hardware.automotive.vehicle.IVehicle/default --get
 559943680 554700801
1: VehiclePropValue{timestamp: 6222169792, areaId: 0, prop: 559943680, status:
AVAILABLE, value: RawPropValues{int32Values: [], floatValues: [0.000000], int64
Values: [], byteValues: [], stringValue: }}
2: VehiclePropValue{timestamp: 6222174774, areaId: 0, prop: 554700801, status:
AVAILABLE, value: RawPropValues{int32Values: [], floatValues: [], int64Values:
[], byteValues: [], stringValue: TMP-0}}
```

Fonte: Elaborada pelo autor (2024).

O primeiro resultado da Figura 36, determinado por “1:”, mostra informações sobre a propriedade INFO_TEMPERATURE_DHT22, que armazena o valor da temperatura. No campo de *status*, é visto que a propriedade está “*AVAILABLE*” (disponível). No campo de valores, é mostrado que em “*floatValues*” o valor está setado em 0.000000, exatamente como configurado nas definições padrão da propriedade.

O segundo resultado, determinador por “2:”, mostra as informações da propriedade FAULT_CODE_TEMPERATURE_DHT22, que armazena o código de *status* do sensor de temperatura. No campo de *status*, é visto que a propriedade está disponível. No campo de valores, é mostrado que em “*stringValue*” está setado o código padrão “TMP-0”, conforme configurado anteriormente. Além disso, os outros tipos estão vazios, indicando que não estão sendo utilizados ou não foram definidos.

A Figura 37 mostra o uso da ferramenta *dumpsys*, mas dessa vez para o segundo par de propriedades criadas para o sensor de aceleração, INFO_ACCELEROMETER_MPU6050 (ID 557912066) e FAULT_CODE_ACCELEROMETER_MPU6050 (ID 554700803).

Figura 37 – Visualização das propriedades do sensor de aceleração

```
➜  adb shell dumpsys android.hardware.automotive.vehicle.IVehicle/default --get
 557912066 554700803
1: VehiclePropValue{timestamp: 9192915179, areaId: 0, prop: 557912066, status:
AVAILABLE, value: RawPropValues{int32Values: [0, 0, 0], floatValues: [], int64V
alues: [], byteValues: [], stringValue: }}
2: VehiclePropValue{timestamp: 9192919290, areaId: 0, prop: 554700803, status:
AVAILABLE, value: RawPropValues{int32Values: [], floatValues: [], int64Values:
[], byteValues: [], stringValue: ACC-0}}
```

Fonte: Elaborada pelo autor (2024).

O primeiro resultado mostra mais informações acerca da propriedade responsável por armazenar a aceleração bruta nos eixos X, Y e Z, a INFO_ACCELEROMETER_MPU6050. O campo *status* indica que a propriedade está disponível no sistema. O campo de valores, do tipo “int32Values”, é um *array* de três posições para armazenar a aceleração bruta em cada eixo, configurado como “[0, 0, 0]” conforme definido na configuração padrão da propriedade.

O segundo resultado mostra a FAULT_CODE_ACCELEROMETER_MPU6050, propriedade que armazena o código de *status* do sensor de aceleração. Ela também está disponível para o sistema e possui um valor do tipo “stringValue” inicializado como “ACC-0”, conforme definido na configuração padrão.

Com o auxílio da ferramenta *dumpsys*, foi possível constatar o pleno funcionamento das propriedades criadas, confirmando que elas estão disponíveis e que se suas configurações foram estabelecidas conforme o esperado. Diante dos resultados obtidos, observa-se que essa etapa foi bem-sucedida e que as propriedades estão prontas para auxiliar a comunicação entre o aplicativo e os dispositivos conectados na rede CAN.

5.3.4.2 Desenvolvimento do serviço CAN2VHAL

O desenvolvimento do serviço CAN2VHAL seguiu o fluxograma ilustrado pela Figura 19. O serviço foi desenvolvido dentro do código-fonte do AAOS, no diretório **device/s-nappautomotive/rpi4_car/can/can2vhal/**. Para desenvolver um serviço dentro do código-fonte do *Android* foi criado o arquivo `Android.bp`, responsável por “ensinar” o *Android* a compilar o código desenvolvido.

Ele é composto por um conjunto de propriedades no formato *chave: "valor"*, que especificam como os diferentes componentes do projeto devem ser compilados. O Código-fonte 8 mostra o arquivo `Android.bp` do serviço CAN2VHAL, um executável escrito em C++.

Código-fonte 8 – Configuração do arquivo de *Android.bp* para serviço VHAL

```
cc_binary {
    name: "can2vhal",
    srcs: ["can2vhal.cpp", "socket_can.cpp"],
    vendor: true,
    shared_libs: ["libbase", "libbinder", "libbinder_ndk",
        "libhidlbase", "liblog", "libutils"],
    static_libs: ["libvhalclient",
        "android.hardware.automotive.vehicle-V1-ndk"
    ],
    defaults: ["vhalclient_defaults"], //<AidlVhalClient.h>
    cflags: ["-Wall", "-Werror", "-Wextra"],
}
```

O módulo *cc_binary* especifica a compilação de um executável C++. A propriedade *name* define o nome do executável, enquanto a propriedade *srcs* lista os arquivos fonte necessários para sua compilação. No caso, o arquivo `socket_can.cpp` contém toda a lógica e métodos referentes à classe SocketCan, e o arquivo `can2vhal.cpp` contém a função principal. A propriedade *vendor*, quando definida como verdadeira, instala o executável na partição /vendor após a compilação. As propriedades *static_libs* e *shared_libs* especificam, respectivamente, as bibliotecas estáticas e dinâmicas utilizadas pelo executável. A propriedade *defaults* inclui o cabeçalho `AidlVhalClient.h` que contém as funções da VHAL. Por fim, a propriedade *cflags* define as flags de compilação.

O arquivo `can2vhal.cpp` contém a inicialização do *socket* CAN e do cliente da VHAL, além da lógica principal do serviço para atualização da propriedades da VHAL. O Código-fonte 9 contém o trecho de código responsável pela inicialização do *socket* CAN.

Na primeira linha, a classe `SocketCan` é instanciada, passando como argumento o nome da interface de rede `can0`, que será gerenciada pela classe. Entre a segunda e a quarta linha, é feita a chamada ao método booleano `Init()` da instância `socket_can`. Este método é responsável por abrir e realizar a ligação (*bind*) do socket CAN à interface especificada. Se a inicialização for bem-sucedida, o método retorna *true*; caso contrário, retorna *false*.

Código-fonte 9 – Inicialização do *socket* CAN

```
tcc::aaos::can::SocketCan socket_can("can0");
if(!socket_can.Init()) {
  ALOG(LOG_ERROR, TAG, "Failed to initialize socket CAN");
  return 1;
}
```

Se `Init()` retornar false, indicando falha na inicialização do socket CAN, o código dentro do bloco condicional será executado, registrando uma mensagem de erro no sistema com

a macro `ALOG`. Em seguida o programa é encerrado com o código “1”, sinalizando ao sistema operacional a ocorrência de erro na execução.

O Código-fonte 10 mostra a inicialização do cliente VHAL. Na primeira linha, é feita a chamada ao método estático `tryCreateAidlClient`, passando o descritor do serviço da VHAL em AIDL.

Código-fonte 10 – Inicialização do cliente VHAL

```
1  std::shared_ptr<IVhalClient> vhal_client = IVhalClient::
       tryCreateAidlClient(AIDL_VHAL_SERVICE);
2  if (vhal_client == nullptr) {
3    ALOG(LOG_ERROR, TAG, "Failed to create VHAL client");
4    return 1;
5  }
```

Esse método retorna um ponteiro inteligente para o cliente VHAL, se o ponteiro retornado for nulo, a criação do cliente não foi bem-sucedida. Então, o bloco condicional será executado, registrando uma mensagem no *log* do sistema e o programa é encerrado. Caso contrário, o cliente foi corretamente inicializado e o programa continuará sua execução.

O Código-fonte 11 cria e inicializa variáveis para representar propriedades para cada sensor, utilizando o cliente VHAL. Ao passar o ID da propriedade – acessível pelo `enum class` *VehicleProperty* – como argumento do método `createHalPropValue`, um ponteiro para o valor da propriedade é retornado. Caso algum ponteiro retornado seja nulo, a aplicação é encerrada; caso contrário, a execução continuará.

Código-fonte 11 – Inicialização de propriedades VHAL

```
1  auto acc_axes = vhal_client->createHalPropValue(toInt(VehicleProperty
       ::INFO_ACCELEROMETER_MPU6050));
2  auto acc_fault = vhal_client->createHalPropValue(toInt(
       VehicleProperty::FAULT_CODE_ACCELEROMETER_MPU6050));
3  auto temp = vhal_client->createHalPropValue(toInt(VehicleProperty::
       INFO_TEMPERATURE_DHT22));
4  auto temp_fault = vhal_client->createHalPropValue(toInt(
       VehicleProperty::FAULT_CODE_TEMPERATURE_DHT22));
5
6  if (acc_axes == nullptr || acc_fault == nullptr ||
7      temp == nullptr || temp_fault == nullptr) {
8    ALOG(LOG_ERROR, TAG, "Failed to create VHAL properties");
9    return 1;
10 }
```

Após todas essas etapas, o programa segue para o laço de repetição infinito, no qual será observado a chegada de mensagens no *socket* CAN. O método booleano `ReadCanMessage` da instância `socket_can` retorna *true* quando uma mensagem é lida da rede CAN e preenche a

estrutura `frame` passada como referência. Se não houver mensagens, o método retorna *false* e o laço continua sendo executado.

Se a leitura for bem-sucedida, o campo `can_id` será verificado, como mostrado no Código-fonte 12. Caso o identificador corresponda ao das mensagens do acelerômetro, os dados de aceleração em cada eixo são recuperados por deslocamento e soma de *bits*. Com esses valores, a propriedade correspondente é atualizada usando o método `setInt32Values` e sincronizada com o cliente VHAL através do método `setValueSync`.

Em seguida, é verificada a presença de defeitos no sensor: se os valores de aceleração forem todos iguais a INT16_MIN, a propriedade de *status* é definido como “ACC-E1”; caso contrário, o *status* “ACC-0” é mantido, indicando que não há defeitos. A macro INT16_MIN corresponde ao valor -32768. Em condições normais, é extremamente improvável que os três eixos de aceleração simultaneamente alcancem esse valor, o que significa que o sensor está com defeito.

Código-fonte 12 – Lógica do laço infinito para captura de mensagens do sensor de aceleração

```
while (1) {

  if(!socket_can.ReadCanMessage(frame)) {
    ALOG(LOG_ERROR, TAG, "Failed to read CAN message");
  } else {
    switch (frame.can_id) {
      case ACCELEROMETER_CAN_ID: {
        axis_x = (frame.data[0] << 8) | frame.data[1];
        axis_y = (frame.data[2] << 8) | frame.data[3];
        axis_z = (frame.data[4] << 8) | frame.data[5];

        acc_axes->setInt32Values({axis_x, axis_y, axis_z});
        vhal_client->setValueSync(*acc_axes);

        if(axis_x == INT16_MIN && axis_y == INT16_MIN && axis_z ==
            INT16_MIN) {
          acc_fault->setStringValue("ACC-E1");
          vhal_client->setValueSync(*acc_fault);
        } else {
          acc_fault->setStringValue("ACC-0");
          vhal_client->setValueSync(*acc_fault);
        }
        break;
      }
```

Caso o identificador da mensagem CAN corresponda ao das mensagens do sensor de temperatura, o dado é recuperado a partir de uma conversão de *bytes* para ponto flutuante com auxílio da função `BytesToFloat`. Com o valor de temperatura recuperado, as propriedades relacionadas ao sensor são preenchidas e atualizadas, como mostrado no Código-fonte 13.

Código-fonte 13 – Lógica para captura de mensagens do sensor de temperatura

```
case TEMPERATURE_CAN_ID: {
  temperature = BytesToFloat(frame.data);
  temp->setFloatValues({temperature});
  vhal_client->setValueSync(*temp);

  if (temperature <= -273) {
    temp_fault->setStringValue("TMP-E1");
    vhal_client->setValueSync(*temp_fault);
  } else {
    temp_fault->setStringValue("TMP-0");
    vhal_client->setValueSync(*temp_fault);
  }
  break;
}
```

A verificação de defeitos segue a mesma lógica aplicada ao sensor de aceleração. Foi verificado se o dado recebido é menor igual a $-273\ °C$, que representa o zero absoluto. Em condições normais, é extremamente improvável o sensor capturar esse valor, indicando que o sensor está com defeito. Caso essa condição seja alcançada, a propriedade de *status* é atualizada para “TMP-E1”; caso contrário, ela preenchida com o código “TMP-0”.

O código completo desse serviço pode ser encontrado no repositório[1] do trabalho, hospedado no GitHub, no diretório **android-stuff/can/can2vhal/**.

5.3.5 Desenvolvimento do sistema de informação e entretenimento veicular

Esta subseção apresenta os resultados alcançados no desenvolvimento do aplicativo proposto por este trabalho.

5.3.5.1 Definição dos requisitos e prototipação das telas do aplicativo

A primeira etapa para o desenvolvimento do aplicativo contemplou a definição de requisitos funcionais e não funcionais, com o objetivo de tornar a experiência do usuário mais fácil e orientar o desenvolvimento.

O Quadro 8 mostra os requisitos definidos. Os requisitos funcionais descrevem as funcionalidades que a aplicação oferece para satisfazer as necessidades do usuário final. Por outro lado, os não funcionais descrevem características globais do aplicativo.

[1] Repositório do trabalho disponível em: github.com/SamuelLost/tcc-can-aaos-esp

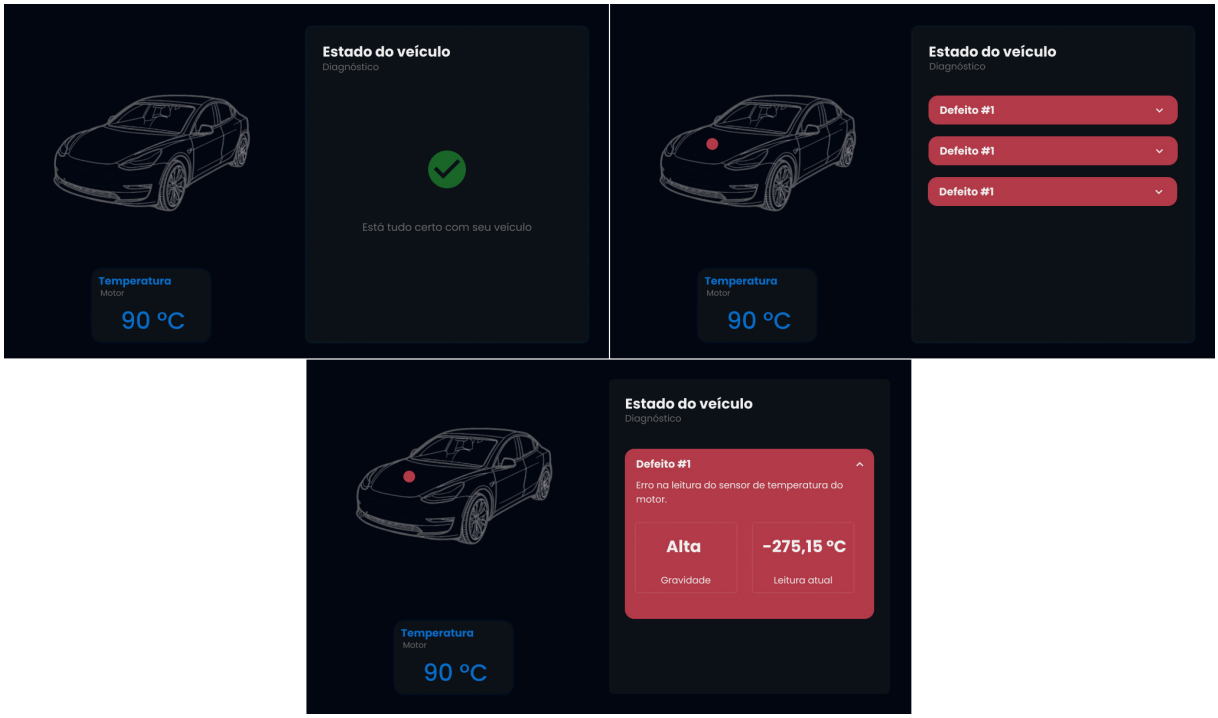
Quadro 8 – Requisitos funcionais e não funcionais para o aplicativo

Requisitos funcionais	**RF1:** Deve exibir e detalhar os defeitos do veículo de forma simultânea. **Prioridade:** Essencial.
	RF2: Deve monitorar os dados do veículo de forma simultânea. **Prioridade:** Essencial.
	RF3: Deve sugerir ações que o motorista pode tomar em resposta ao defeito. **Prioridade:** Essencial.
	RF4: Deve gerar alertas visuais quando um novo defeito é detectado. **Prioridade:** Essencial.
	RF5: Deve mostrar a gravidade dos defeitos e dado atual do sensor. **Prioridade:** Importante.
Requisitos não funcionais	**RNF1:** A interface do usuário deve ser intuitiva e fácil de navegar.
	RNF2: O aplicativo deve garantir a integridade dos dados de diagnóstico.
	RNF3: O aplicativo deve permitir a adição fácil de novos tipos de sensores.
	RNF4: A linguagem para descrever os defeitos deve ser clara e objetiva.

Fonte: Elaborado pelo autor (2024).

A Figura 38 ilustra a prototipação final das três telas do aplicativo. Nela é possível ver a imagem do carro, a temperatura do motor do carro, a descrição dos defeitos e um alerta visual de onde está o defeito no veículo.

Figura 38 – Prototipação final da interface do aplicativo



Fonte: Elaborada pelo autor (2024).

A primeira interface (esquerda superior) é mostrada quando não existe a presença de um defeito, indicado pela mensagem “Está tudo certo com seu veículo” e um ícone verde de confirmação na seção de estado e diagnóstico do veículo. Além disso, a figura do veículo não está sinalizada com nenhum defeito.

Na segunda (direita superior) e terceira (centro inferior) interfaces é mostrada a presença de defeitos, indicado pelos *cards* em vermelho na seção de diagnóstico e estado, indicado pelo detalhe em vermelho no capô do veículo. Na terceira interface, é possível ver o *card* do defeito expandido, mostrando os detalhes acerca do mesmo.

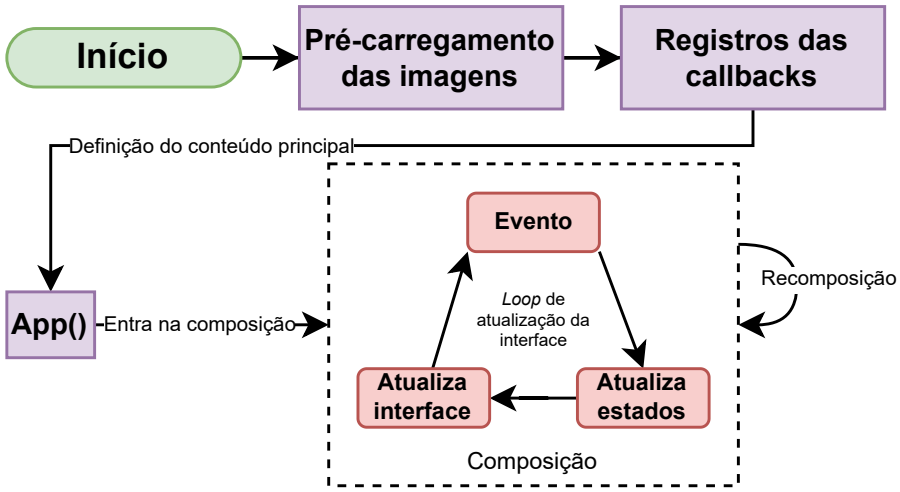
5.3.5.2 Desenvolvimento da aplicação

O sistema de informação e entretenimento veicular foi desenvolvido com Kotlin e o *framework* Jetpack Compose, conforme mencionado nos procedimentos metodológicos. Esse *framework* torna o desenvolvimento de uma *User Interface* (UI) no *Android* mais rápida e fácil.

A Figura 39, mostra o fluxo de execução do aplicativo desenvolvido. Primeiramente é realizado o pré-carregamento das imagens em SVG do veículo e a configuração da *cache* utilizando a biblioteca Coil[2]. Esse pré-carregamento torna a atualização da interface mais dinâmica e fluida, eliminando demoras ao atualizar a imagem.

Em seguida, é realizado o registro das *callbacks* para cada propriedade criada na VHAL. Elas são chamadas quando um evento de mudança no valor da propriedade acontece, sendo possível recuperar o novo valor da propriedade e realizar ações com base nessas mudanças. Isso possibilita atualização dinâmica e em tempo real na interface ou no comportamento da aplicação.

Figura 39 – Fluxograma de execução do aplicativo



Fonte: Elaborada pelo autor (2024).

A função `App()` é o *composable* principal do aplicativo, composto de outros *composables* menores que formam as interfaces. Esta função inicia a composição da UI, onde, a

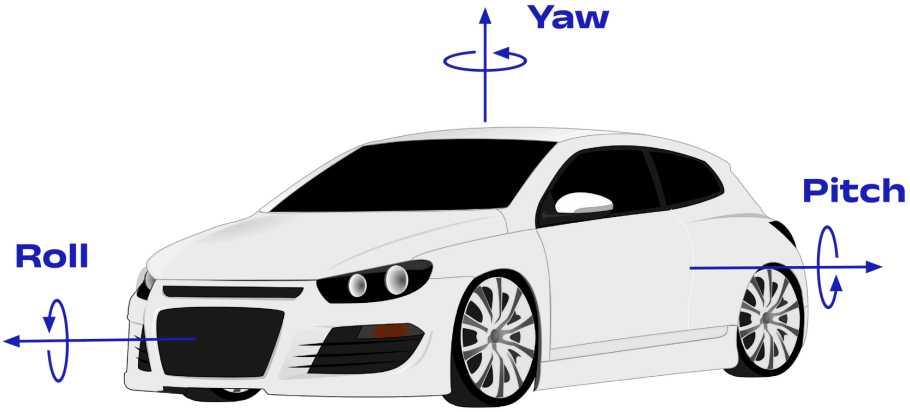
[2] Saiba mais sobre o Coil em: https://coil-kt.github.io/coil/. Acesso em: 10 abr. 2024

partir de eventos disparados pela mudança de valores das propriedades da VHAL, os estados são atualizados e a interface é recomposta para refletir essas mudanças. O ciclo de recomposição é contínuo e ocorre sempre que um estado muda, garantindo que a interface esteja sempre sincronizada com os dados.

Durante a implementação das interfaces, foram realizadas algumas mudanças em relação ao protótipo final feito no Figma. A estratégia de sinalização visual dos defeitos na imagem do veículo foi alterada, ao invés de apenas um ponto em vermelho no lugar onde estaria o defeito, a área toda foi pintada para facilitar a visualização pelo usuário.

A segunda alteração foi a adição de mais um *card* responsável por mostrar dados do veículo. Com os dados de aceleração foi realizado o cálculo do ângulo *roll*, ilustrado na Figura 40, para mostrar o comportamento do veículo em curvas e inclinações.

Figura 40 – Definições de ângulos de *roll*, *pitch* e *yaw* de um veículo



Fonte: Adaptado de (Wang *et al.*, 2020).

Esse *card* foi incluído apenas com o objetivo ser mais uma visualização para o usuário. Como descrito por Montalvo (2021, p.30-31), foi utilizada a seguinte fórmula para o cálculo do ângulo *roll* (ϕ):

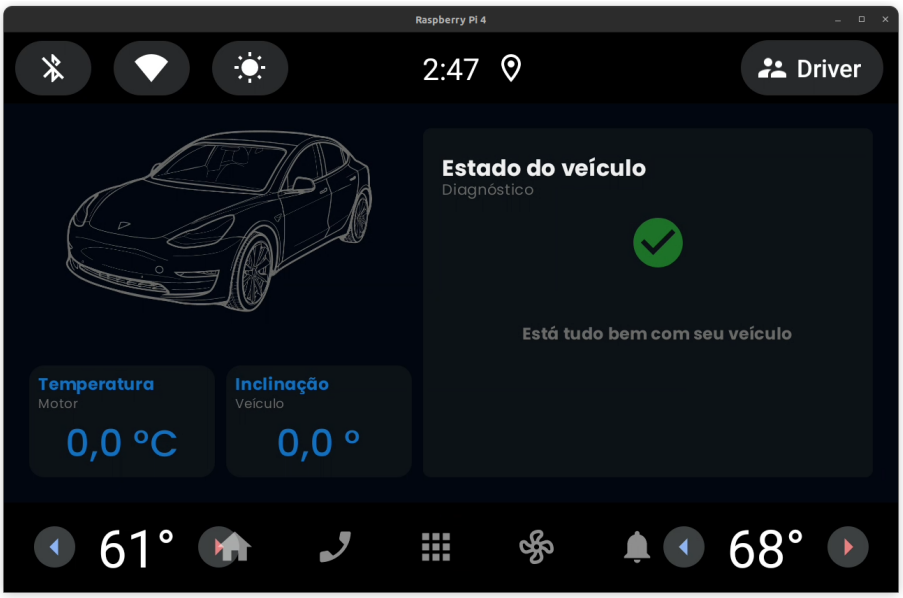
$$\phi = \arctan\left(\frac{\bar{a}_y}{\bar{a}_z}\right) \times \frac{180}{\pi} \tag{5.1}$$

Onde, o ângulo *roll* (ϕ) é determinado pelo arco tangente da razão entre os componentes normalizados da aceleração nos eixos Y ($\bar{a}_y$) e Z ($\bar{a}_z$). Posteriormente, o valor obtido é convertido de radianos para graus, multiplicando-se por $\frac{180}{\pi}$.

A Figura 41 mostra o aplicativo implantado na *Raspberry Pi* 4B com o sistema *Android* Automotivo. Esta é a tela padrão do aplicativo quando não há dados preenchidos nas propriedades da VHAL. No lado esquerdo temos a imagem do veículo e os *cards* que exibem os

dados de temperatura do motor e a inclinação do veículo em tempo real. No lado direito, temos a visualização do estado e diagnóstico do veículo, onde os defeitos são mostrados.

Figura 41 – Interface do aplicativo desenvolvido



Fonte: Elaborada pelo autor (2024).

A instalação do aplicativo foi realizada através do *Android Studio* e da conexão via adb. Ao abrir o projeto no *Android Studio* com o adb conectado, a *Raspberry Pi* era automaticamente identificada como um dispositivo. Assim, ao executar o projeto, o aplicativo era instalado e executado automaticamente no AAOS. O código-fonte do aplicativo está disponível no repositório[3] do trabalho.

5.3.6 Resultados do experimento C

O experimento C foi realizado com o objetivo de validar a integração de todas as partes do sistema desenvolvido. O cenário de teste deste experimento contempla a visualização dos dados de aceleração e temperatura pelo aplicativo.

Com o nó da ESP32 e o nó da *Raspberry Pi* 4B conectados na rede CAN, o serviço CAN2VHAL foi executado para capturar as mensagens enviadas através do barramento e preencher as propriedades VHAL. A Figura 42 ilustra o seu funcionamento, exibindo nos *logs* os dados de aceleração e temperatura (em *Celsius*) recuperados através do *socket* CAN.

[3] Código-fonte disponível em: github.com/SamuelLost/tcc-can-aaos-esp

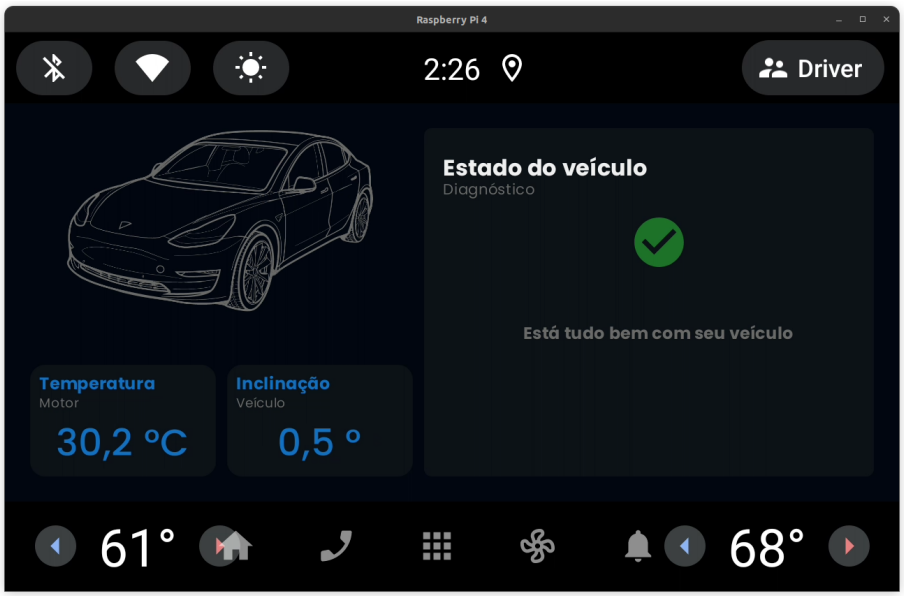
Figura 42 – Execução do serviço CAN2VHAL

```
→  adb shell can2vhal
Socket CAN initialized
VHAL client created
Axis X: -12808, Axis Y: -2212, Axis Z: -9876
Temperature: 30.2
Axis X: -12828, Axis Y: -2196, Axis Z: -9940
Temperature: 30.2
Axis X: -12908, Axis Y: -1992, Axis Z: -9840
Temperature: 30.2
```

Fonte: Elaborada pelo autor (2024).

A Figura 43 mostra o aplicativo enquanto o serviço CAN2VHAL captura as mensagens CAN e preenche as propriedades da VHAL. O *card* de temperatura do motor mostra o valor de $30,2$ °C presente na propriedade. Da mesma forma, o *card* de inclinação do veículo mostra o ângulo *roll* calculado com os valores de aceleração armazenados pela propriedade. Neste caso, não há nenhum defeito com o veículo, como indicado pela mensagem "Está tudo bem com o seu veículo" e o ícone de confirmação em verde na visualização do estado de veículo à direita.

Figura 43 – Interface do aplicativo sem existência de defeitos



Fonte: Elaborada pelo autor (2024).

A Figura 44 ilustra a interface do aplicativo após a desconexão do sensor de temperatura. Há uma sinalização visual na imagem do veículo, com a área do capô destacada em vermelho para indicar o local do defeito. Na visualização à direita, o diagnóstico do veículo é apresentado através de um *card* contendo informações como descrição do defeito, ação recomendada para o motorista, gravidade do problema e a leitura atual do sensor.

O *card* de diagnóstico indica que o defeito ocorreu na leitura do sensor e que sua gravidade é classificada como alta. A recomendação para o motorista é parar o carro e procurar a

oficina mais próxima. Além disso, a leitura atual do sensor é mostrada como $-273,1$ °C, valor fora da escala de operação, conforme configurado anteriormente.

Figura 44 – Interface do aplicativo com defeito no sensor de temperatura

Fonte: Elaborada pelo autor (2024).

A Figura 45 exibe a interface após o sensor de aceleração ser desconectado. Dessa vez, a área do para-choque é destacada, o *card* de diagnóstico indica que o defeito ocorreu na leitura do sensor, com gravidade classificada como média. A recomendação é que o motorista leve o carro em uma oficina para reparos imediatos. Além disso, a inclinação indica o valor de -135 °, o que, se o carro não estiver capotado, indica um defeito no sensor.

Figura 45 – Interface do aplicativo com defeito no sensor de aceleração

Fonte: Elaborada pelo autor (2024).

Por fim, a Figura 46 exibe a interface quando há defeitos em ambos os sensores simultaneamente. A imagem do veículo apresenta as áreas do capô e do para-choque destacadas, e os dados do veículo mostrados indicam valores fora da normalidade. À direita, os *cards* de diagnóstico são exibidos em sua forma minimizada padrão, indicando os defeitos presentes sem as informações adicionais, que podem ser acessadas tocando nos *cards* para expandi-los.

Figura 46 – Interface do aplicativo com defeitos nos sensores de temperatura e de aceleração

Fonte: Elaborada pelo autor (2024).

Os resultados obtidos com o experimento C mostraram que as mensagens CAN estão sendo recebidas e repassadas até as propriedades da VHAL pelo serviço CAN2VHAL, conforme o esperado. Também demonstraram que o sistema de infoentretenimento foi capaz de exibir os dados do veículo armazenados nas propriedades, e que acima de tudo também conseguiu comunicar os defeitos que ocorrem no veículo através de imagens e texto, além de sugerir ações ao motorista. Portanto, esses resultados concluem o primeiro e o segundo objetivo específico deste trabalho.

5.4 Avaliação do sistema de informação e entretenimento

Nesta seção, são apresentados os resultados obtidos no teste de usabilidade do sistema de infoentretenimento desenvolvido. O teste teve como objetivo principal avaliar a comunicação de defeitos veiculares, eficácia e nível de satisfação do usuário com o aplicativo.

Cinco moradores do distrito de Flores, foram convidados para participar da avaliação.

Para preservar o anonimato, eles serão chamados de **P1**, **P2**, **P3**, **P4** e **P5**. Primeiramente, foi realizada a entrevista pré-teste (Apêndice C) para identificar o perfil dos participantes. Em seguida, os participantes utilizaram o aplicativo e realizaram as atividades (Apêndice D). Por fim, foi realizada a entrevista pós-teste (Apêndice E) para coletar o *feedback* dos participantes.

5.4.1 Entrevista pré-teste

A Tabela 3 apresenta o perfil dos participantes. Dos cinco participantes, dois são do sexo feminino (**P3** e **P5**) e três são do sexo masculino (**P1**, **P2** e **P4**), com idades variando entre 22 e 46 anos. Todos possuem CNH, documento oficial que, no Brasil, comprova a capacidade para conduzir veículos automotores terrestres. Em relação ao nível de escolaridade, **P2**, **P3** e **P4** concluíram o ensino superior, **P1** possui ensino técnico e **P5** ainda não concluiu o ensino superior.

Tabela 3 – Perfil do participantes

Participante	**Sexo**	**Idade**	**Escolaridade**	**Possui CNH?**	**Anos de experiência em direção de veículos**
P1	M	24	Ensino técnico completo	Sim	1
P2	M	46	Ensino superior completo	Sim	28
P3	F	27	Ensino superior completo	Sim	5
P4	M	34	Ensino superior completo	Sim	14
P5	F	22	Ensino superior incompleto	Sim	1

Fonte: Elaborado pelo autor (2024).

Na Tabela 3, observa-se uma diversidade significativa em relação à experiência na direção de veículos. **P2** e **P4** destacam-se por possuírem vasta experiência, com 28 e 14 anos, respectivamente. **P3** tem uma experiência intermediária, com cinco anos, enquanto **P1** e **P5** possuem apenas um ano de experiência cada.

5.4.2 Atividades dos participantes

O Apêndice D mostra as atividades executadas pelos participantes nos quatro cenários da avaliação. As quatro atividades criadas, possui perguntas fechadas e abertas, buscando compreender melhor a percepção dos usuários. Abaixo são apresentados os resultados da aplicação das atividades e das perguntas feitas em cada um dos cenários de avaliação.

5.4.2.1 Atividade 1 - Sem defeitos

Questão 1. Você observa algum sinal ou indicativo de defeito no veículo? Poderia descrever quais sinais ou indicativos levaram você a essa conclusão?

- **P1** - "Não. Eu acho que pelas informações que aparecem para mim. No carro não aparece nada em vermelho e na tela só informa a temperatura e inclinação, a seta verde e a mensagem 'está tudo bem com seu veículo'."
- **P2** - "Não. A temperatura do motor está dentro do esperado e a inclinação está dentro do normal, porque o carro está em um estado parado, onde essa inclinação pode ser causada pelo próprio asfalto. E esse ícone está dando um 'OK', assim como a mensagem que está apresentando: 'está tudo bem com o seu veículo'."
- **P3** - "Não. O diagnóstico que diz que está tudo bem com o veículo."
- **P4** - "Não. Temperatura do motor está normal, a inclinação está normal – varição pequena –, o ícone verde e a mensagem 'está tudo bem com seu veículo'."
- **P5** - "Não. Tem um quadrado grande do estado do veículo que está dizendo: 'está tudo bem com o seu veículo' e um ícone verde."

Questão 2. Você percebeu alguma indicação de que algum sensor estaria defeituoso? Se sim, qual?

Todos os participantes responderam que não perceberam nenhuma indicação de defeito nos sensores, selecionando o item fechado "d) Não, não percebi indicações".

Questão 3. Com base nas informações apresentadas pelo aplicativo, você acredita que seu carro precisa de reparos imediatos? (Sim ou não)

Segundo os participantes, o carro não precisa de reparos e responderam à pergunta com o item "b) Não".

5.4.2.2 Atividade 2 - Sensor de temperatura

Questão 1. Você observa algum sinal ou indicativo de defeito no veículo? Poderia descrever quais sinais ou indicativos levaram você a essa conclusão?

- **P1** - "Sim. A imagem vermelha no capô do carro, a temperatura que aparece em -273 ºC e a descrição do estado do veiculo."

- **P2** - “Sim. Apresentando uma baixa temperatura no motor. No ícone está apresentando ‘defeito no sensor de temperatura’, a leitura atual e a figura mostra um aquecimento ou resfriamento súbito no capô do carro.”
- **P3** - “Sim. O diagnóstico: ‘defeito no sensor de temperatura’.”
- **P4** - “Sim. O diagnóstico.”
- **P5** - “Sim. O capô do carro está vermelho e no quadrado do estado do veículo está: ‘defeito no sensor de temperatura’.”

Questão 2. Quantos defeitos você identificou no veículo?

Todos os participantes identificaram apenas um defeito nesse cenário.

Questão 3. Você identificou quais sensores estão afetados pelos defeitos? Se sim, quais?

Os participantes responderam que identificaram o sensor de temperatura como defeituoso, escolhendo o item fechado “a) Sim, sensor de temperatura”.

Questão 4. Após interagir com o aplicativo, como você avaliaria a gravidade do defeito apresentado? (Nos níveis baixa, média e alta)

Todos os participantes avaliaram a gravidade como alta, selecionando o item “c) Alta” do questionário.

Questão 5. As informações apresentadas pelo aplicativo sugerem a necessidade de reparos imediatos no veículo? (Sim ou não)

Segundo os participantes, o aplicativo sugere a necessidade de reparos imediatos.

5.4.2.3 Atividade 3 - Sensor de aceleração

Questão 1. Você observa algum sinal ou indicativo de defeito no veículo? Poderia descrever quais sinais ou indicativos levaram você a essa conclusão?

- **P1** - “Sim. A imagem da frente do carro em vermelho e os detalhes ao lado: ‘defeito no sensor de aceleração’.”
- **P2** - “Sim. A cor do para-choque mudou e está indicando que a leitura de inclinação está negativa.”

- **P3** - “Sim. O diagnóstico: ‘defeito no sensor de aceleração’.”
- **P4** - “Sim. O defeito no sensor de aceleração e a figura do para-choque.”
- **P5** - “Sim. O para-choque do desenho do carro em vermelho e o fato de que está escrito ‘defeito no sensor de aceleração’ no quadro de estado do veículo.”

Questão 2. Quantos defeitos você identificou no veículo?

Todos os participantes responderam que identificaram apenas um defeito.

Questão 3. Você identificou quais sensores estão afetados pelos defeitos? Se sim, quais?

Os participantes identificaram o sensor de aceleração como defeituoso, escolhendo o item “b) Sim, sensor de aceleração” do questionário.

Questão 4. Após interagir com o aplicativo, como você avaliaria a gravidade do defeito apresentado? (Baixa, média ou alta)

Após interagirem com o aplicativo, os participantes avaliaram o defeito como gravidade média, escolhendo o item “b) Média”.

Questão 5. As informações apresentadas pelo aplicativo sugerem a necessidade de reparos imediatos no veículo?

Segundo os participantes, o aplicativo sugere a necessidade de reparos.

5.4.2.4 Atividade 4 - Ambos os sensores

Questão 1. Você observa algum sinal ou indicativo de defeito no veículo? Poderia descrever quais sinais ou indicativos levaram você a essa conclusão?

- **P1** - “Sim. A imagem do carro mostra a frente em vermelho e o capô também, mais as informações adicionais.”
- **P2** - “Sim. Mudou a cor do capô do carro e do para-choque dianteiro, informa o defeito de aceleração e temperatura.”
- **P3** - “Sim. O diagnóstico: ‘defeito no sensor de temperatura e no sensor de aceleração’.”
- **P4** - “Sim, defeito do sensor de temperatura e de aceleração, e os desenhos no carro.”
- **P5** - “Sim. O capô e o para-choque estão vermelhos e dois retângulos dizendo que tem

defeito nos sensores no estado do veículo.”

Questão 2. Quantos defeitos você identificou no veículo?

Neste cenário, os participantes identificaram dois defeitos presente no veículo.

Questão 3. Você identificou quais sensores estão afetados pelos defeitos? Se sim, quais?

Os participantes identificaram que tanto o sensor de temperatura quanto o sensor de aceleração foram afetados por algum defeito, selecionando o item “c) Sim, ambos os sensores”.

Questão 4. Após interagir com o aplicativo, como você avaliaria a gravidade do defeito no sensor de temperatura? (Baixa, média ou alta)

Os participantes avaliaram como gravidade alta.

Questão 5. Após interagir com o aplicativo, como você avaliaria a gravidade do defeito no sensor de aceleração? (Baixa, média ou alta)

Os participantes avaliaram como gravidade média.

Questão 6. As informações apresentadas pelo aplicativo sugerem a necessidade de reparos imediatos no veículo? (Sim ou não)

Todos os participantes afirmaram que o aplicativo sugere a necessidade de reparos imediatos.

5.4.3 Entrevista pós-teste

Abaixo são apresentadas as opiniões, sugestões e níveis de satisfação dos participantes em relação ao aplicativo coletados durante na entrevista pós-teste.

Questão 1. Qual seu nível de satisfação com o aplicativo? (Nos níveis 1 a 5, onde 1 é nada satisfeito e 5 muito satisfeito)

Os cinco participantes demonstraram estar muito satisfeitos com o aplicativo.

Questão 2. Qual o nível de utilidade do aplicativo na sua opinião? (Nos níveis 1 a 5, onde 1 é nada útil e 5 muito útil)

Os cinco participantes consideraram o aplicativo muito útil. **P1** destacou: "Por ter a imagem mostrando onde está o defeito, eu avalio como 5. Pois, para uma pessoa que não entende, só de olhar para imagem já fica ciente do que está acontecendo."

Questão 3. O que você entendeu sobre este aplicativo?

- **P1** - "Informa o estado do veiculo, pois muitas pessoas geralmente não sabem identificar quando tem algum defeito. Então ter um aplicativo assim é muito importante."
- **P2** - "Uma forma de mostrar de imediato qual reparo deve ser realizado no veículo, mostrando que o carro está com defeitos."
- **P3** - "Ele ajuda a identificar os problemas que o carro está apresentando, os defeitos."
- **P4** - "É uma ferramenta boa, pois mostra em tempo real algum defeito que possa ter no carro e ainda lhe dá um status da gravidade da situação. Por que tem pessoas que não sabem a gravidade que se tá passando, se continuar andando o carro pode até parar totalmente e gerar mais prejuízo em função de uma não manutenção imediata."
- **P5** - "Mostra defeitos nos sensores do carro ou mostra se tá tudo bem, se não tem defeito nenhum."

Questão 4. O aplicativo cumpre com o que ele se propõe?

Segundo os participantes, o aplicativo cumpriu com o que ele se propôs.

Questão 5. Qual foi sua maior dificuldade ao usar o aplicativo?

- **P1** - "Nenhuma, o aplicativo é bem direto. Talvez na parte dos *cards* e informações adicionais. Alguma pessoa leiga teria dificuldade em saber que precisa clicar."
- **P2** - "Nenhuma, está claro e objetivo."
- **P3** - "A falta de conhecimento sobre veículos em geral."
- **P4** - "Nenhuma."
- **P5** - "Nenhuma."

P4 e **P5** não fizeram destaques em relação ao aplicativo.

Questão 6. Quais sugestões para melhoria do aplicativo você daria?

- **P1** - “Mudar a cor de destaque do defeito de acordo com a gravidade.”
- **P2** - “Não, por mim, já está completo, bem dinâmico e assertivo.”
- **P3** - “A capacidade de processamento do sistema. Respostas mais rápidas e um *hardware* melhor.”
- **P4** - “Nenhuma.”
- **P5** - “Mudar a cor de acordo com a gravidade.”

5.4.4 Relato e interpretação dos resultados

Além das respostas e opiniões dos participantes, foram coletados o tempo de resposta e a pontuação total de cada um. O questionário possui 19 questões, valendo um ponto cada. A partir das gravações das avaliações, o intervalo entre o fim da pergunta e o fim da resposta foi medido com auxílio do editor de vídeo Kdenlive[4]. Essa medição permitiu determinar o tempo que cada participante levou para responder cada pergunta, gerando o tempo de resposta para cada atividade.

A Tabela 4 mostra a pontuação e o tempo de resposta em cada atividade e tempo total de todas atividades alcançado por cada um dos participantes. É possível notar que todos os participantes obtiveram pontuação máxima nos questionários, alcançando um resultado excelente. Além da pontuação, também é mostrado o tempo que cada participante levou para concluir as atividades. **P3** se destacou por sua agilidade em responder as perguntas e perceber as mudanças na interfaces, levando apenas $50,5$ segundos para responder todas as atividades.

Tabela 4 – Pontuação e tempo de resposta de cada participante

Participante	Pontos	Atividade 1 (s)	Atividade 2 (s)	Atividade 3 (s)	Atividade 4 (s)	Total (s)
P1	19/19	19	30	16,5	16	81,5
P2	19/19	21	25	20,5	17	83,5
P3	19/19	9,5	13	10,5	17,5	50,5
P4	19/19	20,5	15,5	24	14,5	74,5
P5	19/19	13	19,5	17	28,5	78

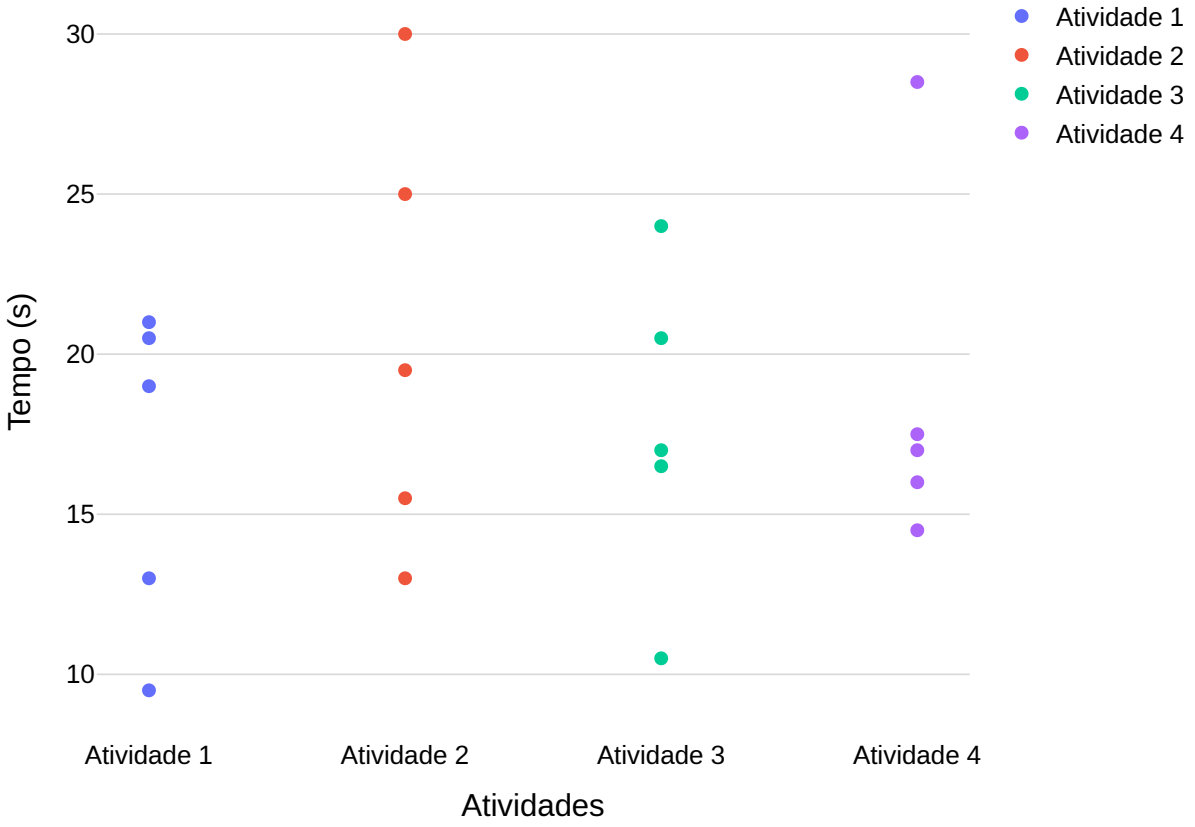
Fonte: Elaborado pelo autor (2024).

A Figura 47 ilustra a variação do tempo de resposta em cada atividade por meio de um gráfico de pontos. Os pontos representam o tempo de resposta dos participantes no teste de

[4] Saiba mais sobre o Kdenlive. Acesso em: 01 ago. 2024

usabilidade. Os pontos em roxo representam a Atividade 1, os pontos em vermelho a Atividade 2, em verde a Atividade 3 e em rosa a Atividade 4.

Figura 47 – Distribuição dos tempos de resposta por atividade no teste de usabilidade



Fonte: Elaborada pelo autor (2024).

A Atividade 1 mostrou tempos de resposta relativamente consistentes e com pouca variabilidade entre os participantes. Isso pode ser atribuído ao fato de ser o primeiro contato com o aplicativo, no qual ainda não apresentava defeitos e continha pouca informação na interface, facilitando a navegação inicial.

A Atividade 2, por outro lado, apresentou a maior variabilidade nos tempos de resposta, conforme evidenciado pelos pontos mais espalhados. Essa variação pode ter sido causada pela introdução de defeitos na interface, que aumentou a complexidade da tarefa. A presença de mais informações e elementos visuais pode ter desafiado os participantes de maneiras diferentes, dependendo do nível de compreensão de cada um à nova configuração da interface.

Nas Atividades 3 e 4, observou-se uma diminuição na variabilidade dos tempos de resposta. Essa redução pode ser um indicativo de que os participantes estavam mais familiarizados com o aplicativo e as atividades propostas. A adaptação à interface, mesmo com a presença de defeitos e informações adicionais, parece ter ocorrido progressivamente, tornando a

execução mais eficiente das tarefas. A diminuição na variabilidade sugere que os participantes desenvolveram uma melhor compreensão do aplicativo, resultando em tempos de resposta mais próximos entre si.

De modo geral, os participantes não encontraram grandes dificuldades na utilização do aplicativo. **P3** comentou que teve dificuldades no início, o mesmo ressaltou que o problema não era o aplicativo e sim a falta de conhecimento sobre veículos em geral, mas não demorou muito para compreender as atividades e as informações mostradas pelas interfaces. **P4** questionou a relação entre o defeito de aceleração e o para-choque. Após a explicação de que sensores acelerômetros podem ser colocados em para-choques para detectar colisões, sua dúvida foi esclarecida.

Todos os participantes demonstraram grande satisfação com o aplicativo e o consideraram muito útil. Eles também disseram ter gostado do destaque visual dos defeitos na figura do veículo, o que facilitou a visualização e localização dos problemas. **P1** e **P4** elogiaram a estrutura e as cores escolhidas para as interfaces. Como sugestão, **P1** e **P4** sugeriram utilizar uma cor distinta para cada nível de gravidade dos defeitos. **P3** sugeriu melhorar o *hardware* e o desempenho do sistema, que apresentou alguns engasgos e travamentos devido ao AAOS tomar muito processamento e a *Raspberry Pi* 4B possuir apenas os recursos mínimos para sua execução.

Os resultados da avaliação de usabilidade demonstraram que todos os participantes completaram com sucesso as tarefas propostas, associaram corretamente os elementos textuais e gráficos aos defeitos, o que confirma que a comunicação dos defeitos foi clara e objetiva. Além disso, todos os participantes concluíram as tarefas em menos de 40 segundos. Portanto, pode-se afirmar que os critérios e a meta de usabilidade foram plenamente alcançados, cumprindo assim o terceiro objetivo específico deste trabalho.

6 CONCLUSÕES E TRABALHOS FUTUROS

Este trabalho apresentou o desenvolvimento e avaliação de um sistema de informação e entretenimento veicular baseado em *Android Automotive OS* (AAOS), que é capaz de apresentar os defeitos veiculares. O trabalho foi composto de diversas etapas, evolvendo microcontroladores, *kernel Linux*, *Android* embarcado, desenvolvimento de aplicativos e até avaliação de Interação Humano-Computador (IHC).

Diante dos resultados obtidos na análise e interpretação dos dados coletados por meio das atividades e entrevistas realizadas antes e após o teste de usabilidade, observou-se que os usuários não enfrentaram dificuldades ao utilizar o aplicativo, apresentando excelente desempenho nas atividades. A forma de visualização de defeitos proposta pelo aplicativo foi bem recebida, sendo considerada clara e objetiva. Os usuários demonstraram facilidade em associar os elementos gráficos e textuais à presença ou à ausência de defeitos no veículo, além de interagirem com os *cards* clicáveis para obter mais informações sobre os defeitos, como a gravidade e as ações recomendadas.

Ao contrário dos métodos utilizados hoje em dia e os abordados pelos trabalhos relacionados – que utilizam códigos e luzes de aviso –, a abordagem de visualização de defeitos veiculares apresentada por este trabalho evitou confusões e desorientações. Os participantes souberam exatamente qual defeito estava ocorrendo, sua gravidade e as ações a serem tomadas, evidenciando a clareza e eficácia da solução proposta. Assim, os usuários agora podem identificar rapidamente defeitos críticos, compreender a gravidade e tomar decisões imediatas, como parar o veículo, evitando possíveis consequências perigosas. Dessa forma, conclui-se que este trabalho cumpriu tanto os objetivos específicos quanto o objetivo geral proposto.

As limitações deste trabalho incluem a dificuldade em configurar e customizar o *Android* Automotivo, a falta de conhecimento prévio sobre desenvolvimento de aplicativos nativo para sistemas *Android* e também sobre IHC. Apesar dessas limitações, o aplicativo demonstrou ser útil e eficaz na visualização de defeitos veiculares, com potencial para melhorias e expansão.

Como trabalhos futuros, têm-se as sugestões destacadas pelos participantes do teste de usabilidade, integrar o aplicativo ao sistema de *build* do AAOS, permitindo sua instalação automática ao gerar uma nova imagem, e incluir novos sensores, escalando a rede CAN. Também se recomenda otimizar o desempenho do AAOS na *Raspberry Pi* 4B, removendo componentes que não são usados ou considerando a aquisição de uma placa com mais recursos para executar o AAOS, a fim de garantir uma execução mais fluida do sistema.

REFERÊNCIAS

ALAM, M. S. U. **Securing vehicle Electronic Control Unit (ECU) communications and stored data**. Tese (Doutorado) – Queen's University (Canada), 09 2018.

ALENCAR, S. H. G. **Integrando o barramento CAN ao Android Automotivo 13**. Embarcados, 2023. Disponível em: https://embarcados.com.br/integrando-o-barramento-can-ao-android-automotivo-13/. Acesso em: 20 dez. 2023.

BARBOSA, S. D. J.; SILVA, B. d.; SILVEIRA, M. S.; GASPARINI, I.; DARIN, T.; BARBOSA, G. D. J. **Interação Humano-Computador e Experiência do Usuario**. Brasil: Auto publicação, 2021.

BHUYAN, M. **Intelligent Instrumentation**: Principles and applications. 1. ed. Boca Raton: Routledge, 2010. ISBN 9781420089530.

CORRIGAN, S. **Introduction to the Controller Area Network (CAN)**. Dallas, Texas, 2016. 17 p. Disponível em: https://www.ti.com/lit/an/sloa101b/sloa101b.pdf. Acesso em: 18 set. 2023.

DENTON, T. **Advanced automotive fault diagnosis**: Automotive technology: Vehicle maintenance and repair. Reino Unido: Routledge, 2020.

ESPRESSIF. **ESP32-WROOM-32E Datasheet**. China, 2023. Version 1.6. Disponível em: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf. Acesso em: 26 set. 2023.

GAO, C.; LUO, L.; ZHANG, Y.; PEARSON, B.; FU, X. Microcontroller Based IoT System Firmware Security: Case Studies. In: IEEE, 2., 2019, Orlando, EUA. **Proceedings of the International Conference on Industrial Internet (ICII)**. Orlando: IEEE, 2019. p. 200–209.

GOELLES, T.; SCHLAGER, B.; MUCKENHUBER, S. Fault Detection, Isolation, Identification and Recovery (FDIIR) Methods for Automotive Perception Sensors Including a Detailed Literature Survey for Lidar. **Sensors**, v. 20, n. 13, 2020. ISSN 1424-8220.

GOOGLE. **Platform architecture**. Google, 2020. Disponível em: https://developer.android.com/guide/platform. Acesso em: 04 out. 2023.

GOOGLE. **Android Automotive**. Google, 2023. Disponível em: https://source.android.com/docs/automotive. Acesso em: 04 out. 2023.

GOOGLE. **Architecture overview**. Google, 2023. Disponível em: https://source.android.com/docs/core/architecture. Acesso em: 24 out. 2023.

GOOGLE. **Vehicle Hardware Abstraction Layer (VHAL) Overview**. Google, 2023. Disponível em: https://source.android.com/docs/automotive/vhal. Acesso em: 12 out. 2023.

GOOGLE. **Android Interface Definition Language (AIDL)**. Google, 2024. Disponível em: https://developer.android.com/develop/background-work/services/aidl. Acesso em: 05 mai. 2024.

GOOGLE. **Property configurations**. Google, 2024. Disponível em: https://source.android.com/docs/automotive/vhal/property-configuration. Acesso em: 15 abr. 2024.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **Road vehicles — Controller area network (CAN) — Part 1**: Data link layer and physical signalling. Geneva, CH, 2015. Disponível em: https://www.iso.org/standard/63648.html. Acesso em: 26 set. 2023.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **Systems and software engineering – Vocabulary**. Geneva, CH, 2017. Disponível em: https://www.iso.org/standard/71952.html. Acesso em: 08 out. 2023.

ISEKE, M. **Android automotive and Physical Car Interaction: Android Automotive OS Book**. 2020. Disponível em: https://www.androidautomotivebook.com/android-automotive-and-physical-car-interaction/. Acesso em: 09 set. 2023.

IVENSENSE. **MPU-6000/MPU-6050 Product Specification**. California, USA, 2013. Rev. 3.4. Disponível em: https://d229kd5ey79jzj.cloudfront.net/974/MPU-6000-Datasheet1.pdf. Acesso em: 30 out. 2023.

JEONG, S.; RYU, M.; KANG, H.; KIM, H. K. Infotainment system matters: Understanding the impact and implications of in-vehicle infotainment system hacking with automotive grade Linux. In: ACM SPECIAL INTEREST GROUP ON SECURITY, AUDIT AND CONTROL (SIGSAC), 13., 2023, Charlotte, USA. **CODASPY '23: Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy**. New York: ACM, 2023. p. 201–212. ISBN 9798400700675.

KARAKURT, T. **MCP2515 CANBus module installation on RaspberryPi**. Nevada, Canada: GitHub, 2019. Disponível em: https://github.com/tolgakarakurt/CANBus-MCP2515-Raspi. Acesso em: 19 out. 2023.

KENJIć, D.; ANTIć, M. Connectivity Challenges in Automotive Solutions. **IEEE Consumer Electronics Magazine**, v. 12, n. 5, p. 53–59, 2023.

KENJIć, D.; ŽIVKOV, D.; ANTIć, M. Automated Data Transfer From ADAS to Android-Based IVI Domain Over SOME/IP. **IEEE Transactions on Intelligent Vehicles**, v. 8, n. 4, p. 3166–3177, 2023.

KHORSRAVINIA, K.; HASSAN, M. K.; RAHMAN, R. Z. A.; AL-HADDAD, S. A. R. Integrated OBD-II and mobile application for electric vehicle (EV) monitoring system. In: IEEE CONTROL SYSTEMS SOCIETY, 2., 2017, Kota Kinabalu, Malaysia. **Proceedings of the 2nd International Conference on Automatic Control and Intelligent Systems (I2CACIS)**. Kota Kinabalu: IEEE, 2017. p. 202–206.

KIM, M.; LEE, J.-e.; JANG, J.-w. Implementation of the android-based automotive infortainment system for supporting drivers' safe driving. In: KOREA INFORMATION PROCESSING SOCIETY (KIPS), 8., 2013, Berlin, Germany. **Proceedings of the 8th International Conference on Ubiquitous Information Technologies and Applications (CUTE 2013)**. Berlin: Springer, 2014. p. 501–508. ISBN 978-3-642-41671-2.

KUMAR, T.; S.SIVAJI. Android-Based Vehicle Monitoring and Tracking System Using ARM7 and CAN Technology. **International Journal of Science Engineering and Advance Technology,IJSEAT**, v. 3, n. 4, 2015.

LAB, D. P. **How to use MCP2515 SPI CAN bus module with Arduino**. DIY Projects Lab, 2023. Disponível em: https://diyprojectslab.com/mcp2515-spi-can-bus-module-with-arduino/. Acesso em: 20 set. 2023.

MATTHEWS, S. J.; BLAINE, R. W.; BRANTLY, A. F. Evaluating single board computer clusters for cyber operations. In: NATO COOPERATIVE CYBER DEFENCE CENTRE OF EXCELLENCE, 8., 2016, Washington, USA. **Proceedings of the 8th International Conference on Cyber Conflict (CyCon U.S.)**. Washington: IEEE, 2016. p. 1–8.

MAXDETECT. **Digital relative humidity and temperature sensor RHT03**. USA, 2013. Disponível em: https://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/Weather/RHT03.pdf. Acesso em: 30 out. 2023.

MCMANUS, S.; COOK, M. **Raspberry Pi for dummies**. New Jersey, USA: John Wiley & Sons, 2021. ISBN 978-1-119-79682-4.

MICROCHIP. **Stand-Alone CAN Controller with SPI Interface**. Chandler, Arizona, EUA, 2019. Rev. J.

MOIZ, A.; ALALFI, M. H. A survey of security vulnerabilities in android automotive apps. In: IEEE/ACM, CO-LOCATED WITH ICSE 2022, 3., 2022, Pittsburgh, USA. **EnCyCriS '22: Proceedings of the 3rd International Workshop on Engineering and Cybersecurity of Critical Systems**. New York, USA: IEEE/ACM, 2022. p. 17–24. ISBN 9781450392907.

MONTALVO, C. **Aerospace Mechanics and Controls**. University of South Alabama Institutional Repository, 2021. Disponível em: https://jagworks.southalabama.edu/southalabama-oer-engineer-textbook/2/. Acesso em: 12 mai. 2024.

MORAN, T. P. The command language grammar: a representation for the user interface of interactive computer systems. **International Journal of Man-Machine Studies**, v. 15, n. 1, p. 3–50, 1981. ISSN 0020-7373.

MOTORS, G. **Manual do Proprietário - Chevrolet Ônix 2019**. Brasil, 2018. 72–87 p. Disponível em: https://www.chevrolet.com.br/content/dam/chevrolet/mercosur/brazil/portuguese/index/services/owner-manuals/06-pdf/my185-onix-52152046-por-20171110-v0-3-low.pdf. Acesso em: 12 nov. 2023.

NIELSEN, J. **Why you only need to test with 5 users**. Nielsen Norman Group, 2000. Disponível em: https://www.nngroup.com/articles/why-you-only-need-to-test-with-5-users. Acesso em: 12 nov. 2023.

OMEROVIC, K.; JANJATOVIC, J.; MILOSEVIC, M.; MARUNA, T. Supporting sensor fusion in next generation android In-Vehicle Infotainment units. In: IEEE, 6., 2016, Berlin, Germany. **Proceedings of the 6th International Conference on Consumer Electronics - Berlin (ICCE-Berlin)**. Berlin: IEEE, 2016. p. 187–189.

PAJIC, N.; BJELICA, M. Integrating Android to Next Generation Vehicles. In: IEEE, 3., 2018, Novi Sad, Serbia. **Proceedings of the Zooming Innovation in Consumer Technologies International Conference (ZINC)**. Novi Sad: IEEE, 2018. p. 152–155.

PARK, S. H.; LEE, S. Y. Development of on-board diagnosis via can for a hvi (human vehicle interface) technology. In: IEEE, 10., 2012, Leganes, Spain. **Proceedings of the 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)**. Leganes, Spain: IEEE, 2012. p. 839–840.

PHILIPS. **High speed CAN transceiver TJA1050**. Netherlands, 2003. Rev. 4. Disponível em: https://www.nxp.com/docs/en/data-sheet/TJA1050.pdf. Acesso em: 24 set. 2023.

POLESTAR. **Polestar 2 - Infotainment features**. Polestar, 2022. Disponível em: https://www.polestar.com/global/polestar-2/infotainment/. Acesso em: 04 nov. 2023.

RASPBERRY. **Raspberry Pi 4 Model B**. United Kingdom (UK), 2019. Realese 1. Disponível em: https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf. Acesso em: 26 set. 2023.

RUBIN, J.; CHISNELL, D.; SPOOL, J. **Handbook of Usability Testing**: How to plan, design, and conduct effective tests. 2. ed. Indianapolis: Wiley, 2008. ISBN 9780470185483.

SEMICONDUCTOR, L. **CAN Controller**. Hillsboro, Oregon, EUA, 2023. Rev. 01.0.

SIMÕES, D. T.; MAFORT, L. S. S. Automotive Event Logger Based on the ESP32 Platform. **Simpósio Internacional de Engenharia Automotiva, SIMEA**, 2021.

SOUZA, C. S. de. **The Semiotic Engineering of Human-Computer Interaction**. Cambridge, Massachusetts: The MIT Press, 2005. ISBN 9780262271363.

SURESH, S. B. **Adding Custom Vehicle Properties in AOSP 13 AIDL VHAL**: Unlocking Customizations. Medium, 2024. Disponível em: https://medium.com/@subin.bsuresh27/adding-custom-vehicle-properties-in-android-aosp-13-aidl-unlocking-customizations-d2a89c29fc64. Acesso em: 15 abr. 2024.

TAHAT, A.; SAID, A.; JAOUNI, F.; QADAMANI, W. Android-based universal vehicle diagnostic and tracking system. In: IEEE, 16., 2012, Harrisburg, USA. **Proceedings of the 16th International Symposium on Consumer Electronics (ISCE)**. Harrisburg: IEEE, 2012. p. 137–143.

TUFO, M.; FIENGO, G.; MONTANARO, U. An Integrated Embedded Solution for Driving Support. In: INFORMATION TECHNOLOGY SOCIETY (ITG) - VDE, 1., 2014, Dortmund, Germany. **Proceedings of the European Conference on Smart Objects, Systems and Technologies**. Dortmund: IEEE, 2014. p. 1–5.

TüRK, E.; CHALLENGER, M. An android-based iot system for vehicle monitoring and diagnostic. In: IEEE, 26., 2018, Izmir, Turkey. **Proceedings of the 26th Signal Processing and Communications Applications (SIU)**. Izmir: IEEE, 2018. p. 1–4.

VALASEK, C.; MILLER, C. Adventures in automotive networks and control units. **Technical White Paper, IOActive**, 2014.

VARGAS, J.; ALSWEISS, S.; TOKER, O.; RAZDAN, R.; SANTOS, J. An Overview of Autonomous Vehicles Sensors and Their Vulnerability to Weather Conditions. **Sensors**, v. 21, n. 16, 2021. ISSN 1424-8220.

VASQUEZ, F.; SIMMONDS, C. **Mastering Embedded Linux Programming**: Create fast and reliable embedded solutions with linux 5.4 and the yocto project 3.1 (dunfell). Birmingham: Packt Publishing, 2021. ISBN 9781789535112.

WAGH, P. A.; PAWAR, R. R.; NALBALWAR, S. Vehicle speed control and safety prototype using controller area network. In: THE NORTHCAP UNIVERSITY, INDIA AND UNIVERSITY OF DAYTON, USA, 1., 2017, Chennai, India. **Proceedings of the International Conference on Computational Intelligence in Data Science (ICCIDS)**. Chennai: IEEE, 2017. p. 1–5.

WANG, Q.; FANG, H.; YIN, H. Reliability analysis of concrete barriers under vehicular crashes using augmented rbfs. **Structural and Multidisciplinary Optimization**, v. 61, 03 2020.

ZENG, W.; KHALID, M. A. S.; CHOWDHURY, S. In-Vehicle Networks Outlook: Achievements and Challenges. **IEEE Communications Surveys & Tutorials**, v. 18, n. 3, p. 1552–1571, 2016.

APÊNDICE A – CÓDIGOS DE PROGRAMAÇÃO

Código-fonte 14 – Comando para gerar novo *hash* de arquivos AIDL

```
cd 'hardware/interfaces/automotive/vehicle/aidl/aidl_api/android.
    hardware.automotive.vehicle/1' && { find ./ -name "*.aidl" -print0
     | LC_ALL=C sort -z | xargs -0 sha1sum && echo latest-version; } |
     sha1sum | cut -d " " -f 1
```

Código-fonte 15 – Declaração com o identificador de cada propriedade

```
INFO_TEMPERATURE_DHT22 = 559943680,
FAULT_CODE_TEMPERATURE_DHT22 = 554700801,
INFO_ACCELEROMETER_MPU6050 = 557912066,
FAULT_CODE_ACCELEROMETER_MPU6050 = 554700803,
```

Código-fonte 16 – Arquivo de compilação can.mk

```
PRODUCT_COPY_FILES += \
 $(LOCAL_PATH)/modules/mcp251x.ko:$(TARGET_COPY_OUT_VENDOR)/lib/
    modules/mcp251x.ko \
 $(LOCAL_PATH)/modules/can-dev.ko:$(TARGET_COPY_OUT_VENDOR)/lib/
    modules/can-dev.ko \
 $(LOCAL_PATH)/modules/spi-bcm2835.ko:$(TARGET_COPY_OUT_VENDOR)/lib/
    modules/spi-bcm2835.ko \
 $(LOCAL_PATH)/modules/can.ko:$(TARGET_COPY_OUT_VENDOR)/lib/modules/
    can.ko \
 $(LOCAL_PATH)/modules/can-raw.ko:$(TARGET_COPY_OUT_VENDOR)/lib/
    modules/can-raw.ko
```

Código-fonte 17 – Arquivo de inicialização dos módulos

```
on boot
    insmod /vendor/lib/modules/spi-bcm2835.ko
    insmod /vendor/lib/modules/can.ko
    insmod /vendor/lib/modules/can-raw.ko
    insmod /vendor/lib/modules/can-dev.ko
    insmod /vendor/lib/modules/mcp251x.ko
```

Código-fonte 18 – Definição das novas propriedades no arquivo AIDL

```
/**
 * Temperature property
 * @change_mode VehiclePropertyChangeMode:CONTINUOUS
 * @access VehiclePropertyAccess:READ_WRITE
 * @unit VehicleUnit:CELSIUS
 */
INFO_TEMPERATURE_DHT22 = 0x1000 + 0x20000000 + 0x01000000
    + 0x00600000, // VehiclePropertyGroup:VENDOR,VehicleArea:GLOBAL,
        VehiclePropertyType:FLOAT
/**
 * Fault code Temperature property
 * @change_mode VehiclePropertyChangeMode:CONTINUOUS
 * @access VehiclePropertyAccess:READ_WRITE
 */
FAULT_CODE_TEMPERATURE_DHT22 = 0x1001 + 0x20000000 + 0x01000000
    + 0x00100000, // VehiclePropertyGroup:VENDOR,VehicleArea:GLOBAL,
        VehiclePropertyType:STRING
/**
 * Accelerometer property
 * @change_mode VehiclePropertyChangeMode:CONTINUOUS
 * @access VehiclePropertyAccess:READ_WRITE
 */
INFO_ACCELEROMETER_MPU6050 = 0x1002 + 0x20000000 + 0x01000000
    + 0x00410000, // VehiclePropertyGroup:VENDOR,VehicleArea:GLOBAL,
        VehiclePropertyType:INT32_VEC
/**
 * Fault code Accelerometer property
 * @change_mode VehiclePropertyChangeMode:CONTINUOUS
 * @access VehiclePropertyAccess:READ_WRITE
 */
FAULT_CODE_ACCELEROMETER_MPU6050 = 0x1003 + 0x20000000 + 0x01000000
    + 0x00100000, // VehiclePropertyGroup:VENDOR,VehicleArea:GLOBAL,
        VehiclePropertyType:STRING
```

Código-fonte 19 – Modificações feitas no arquivo build.config.arpi

```
project common/
diff --git a/build.config.arpi b/build.config.arpi
index 33d9fdb9e..520ff61ef 100644
--- a/build.config.arpi
+++ b/build.config.arpi
@@ -13,6 +13,8 @@ Image.gz
 broadcom/bcm2711-rpi-4-b.dtb
 broadcom/bcm2711-rpi-400.dtb
 overlays/vc4-kms-v3d-pi4.dtbo
+modules
+dtbs
 "
 FILES="
@@ -20,4 +22,10 @@ arch/arm64/boot/Image.gz
 arch/arm64/boot/dts/broadcom/bcm2711-rpi-4-b.dtb
 arch/arm64/boot/dts/broadcom/bcm2711-rpi-400.dtb
 arch/arm64/boot/dts/overlays/vc4-kms-v3d-pi4.dtbo
+arch/arm64/boot/dts/overlays/mcp2515-can0.dtbo
+drivers/net/can/spi/mcp251x.ko
+drivers/spi/spi-bcm2835.ko
+drivers/net/can/dev/can-dev.ko
+net/can/can.ko
+net/can/can-raw.ko
 "
```

Código-fonte 20 – Declaração das configurações de cada propriedade

```
// Temperature sensor
{.config =
        {
            .prop = toInt(VehicleProperty::INFO_TEMPERATURE_DHT22),
            .access = VehiclePropertyAccess::READ_WRITE,
            .changeMode = VehiclePropertyChangeMode::CONTINUOUS,
            .minSampleRate = 1.0f,
            .maxSampleRate = 10.0f,
        },
 .initialValue = {.floatValues = {0.0f}}},
// Fault code for temperature sensor
{.config =
        {
            .prop = toInt(VehicleProperty::
               FAULT_CODE_TEMPERATURE_DHT22),
            .access = VehiclePropertyAccess::READ_WRITE,
            .changeMode = VehiclePropertyChangeMode::CONTINUOUS,
            .minSampleRate = 1.0f,
            .maxSampleRate = 10.0f,
        },
 .initialValue = {.stringValue = "TMP-0"}},
// Accelerometer sensor
{.config =
        {
            .prop = toInt(VehicleProperty::INFO_ACCELEROMETER_MPU6050
               ),
            .access = VehiclePropertyAccess::READ_WRITE,
            .changeMode = VehiclePropertyChangeMode::CONTINUOUS,
            .minSampleRate = 1.0f,
            .maxSampleRate = 10.0f,
        },
 .initialValue = {.int32Values = {0, 0, 0}}},
// Fault code for accelerometer sensor
{.config =
        {
            .prop = toInt(VehicleProperty::
               FAULT_CODE_ACCELEROMETER_MPU6050),
            .access = VehiclePropertyAccess::READ_WRITE,
            .changeMode = VehiclePropertyChangeMode::CONTINUOUS,
            .minSampleRate = 1.0f,
            .maxSampleRate = 10.0f,
        },
 .initialValue = {.stringValue = "ACC-0"}},
```

Código-fonte 21 – Atribuição da permissão CAR_VENDOR_EXTENSION às propriedades criadas

```
/**
 * Temperature info property
 * <p>
 * Requires permission: {@link Car#PERMISSION_VENDOR_EXTENSION}.
 * PERMISSION_VENDOR_EXTENSION is already defined in Car.java for
    vendor
 * </p>
 */
@RequiresPermission(Car.PERMISSION_VENDOR_EXTENSION)
@AddedInOrBefore(majorVersion = 33)
public static final int INFO_TEMPERATURE_DHT22 = 559943680;
/**
 * Temperature fault code property
 * <p>
 * Requires permission: {@link Car#PERMISSION_VENDOR_EXTENSION}.
 * PERMISSION_VENDOR_EXTENSION is already defined in Car.java for
    vendor
 * </p>
 */
@RequiresPermission(Car.PERMISSION_VENDOR_EXTENSION)
@AddedInOrBefore(majorVersion = 33)
public static final int FAULT_CODE_TEMPERATURE_DHT22 = 554700801;
/**
 * Accelerometer info property
 * <p>
 * Requires permission: {@link Car#PERMISSION_VENDOR_EXTENSION}.
 * PERMISSION_VENDOR_EXTENSION is already defined in Car.java for
    vendor
 * </p>
 */
@RequiresPermission(Car.PERMISSION_VENDOR_EXTENSION)
@AddedInOrBefore(majorVersion = 33)
public static final int INFO_ACCELEROMETER_MPU6050 = 557912066;
/**
 * Accelerometer fault code property
 * <p>
 * Requires permission: {@link Car#PERMISSION_VENDOR_EXTENSION}.
 * PERMISSION_VENDOR_EXTENSION is already defined in Car.java for
    vendor
 * </p>
 */
@RequiresPermission(Car.PERMISSION_VENDOR_EXTENSION)
@AddedInOrBefore(majorVersion = 33)
public static final int FAULT_CODE_ACCELEROMETER_MPU6050 = 554700803;
```

APÊNDICE B – TERMO DE CONSENTIMENTO

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

TERMO DE CONSENTIMENTO

Pesquisa: DESENVOLVIMENTO DE UM SISTEMA ANDROID DE INFORMAÇÃO E ENTRETENIMENTO PARA VEÍCULOS: VISUALIZAÇÃO DE DEFEITOS VEICULARES.
Pesquisador: Samuel Henrique Guimarães Alencar.
Orientador: Prof. Dr. Francisco Helder Candido dos Santos Filho.

O objetivo desta pesquisa é: avaliar se a comunicação de defeitos veiculares através do aplicativo é clara e objetiva, e coletar dados sobre a experiência de uso do mesmo. Vale ressaltar que o foco do teste é o aplicativo, e não o participante. Estamos interessados em como o aplicativo funciona e como podemos melhorá-lo, com base no *feedback* e na experiência do usuário.

Por isto, convidamos você a colaborar com a nossa pesquisa, composta de três etapas:

1. Etapa de pré-avaliação.
    a) Entrevista para identificação do perfil dos participantes.
    b) Explicação do aplicativo.
2. Uso do aplicativo.
    a) Provocação dos defeitos.
    b) Perguntas sobre o estado dos sensores e do veículo.
3. Entrevista sobre a opinião e nível de satisfação do participante com o aplicativo.

Para decidir sobre sua participação, é importante que você tenha algumas informações adicionais:

- Os dados coletados serão vistos apenas pelos pesquisadores responsáveis. As entrevistas e a interação serão gravadas, somente para podermos analisar com cuidado os dados coletados.
- A publicação dos resultados desta pesquisa, que é exclusivamente para fins acadêmicos, pauta-se no respeito à privacidade e o anonimato do participante será preservado.
- O consentimento para participação é uma escolha livre, e esta participação pode ser interrompida a qualquer momento, caso você precise ou deseje.

De posse das informações acima, você está de acordo com as informações e concorda em participar da pesquisa ?

- ☐ Estou de acordo com as informação e concordo participar da pesquisa
- ☐ Não estou de acordo com as informações e não quero participar da pesquisa

APÊNDICE C – ENTREVISTA PRÉ-TESTE

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

ENTREVISTA PRÉ-TESTE

Perguntas rápidas com o objetivo de traçar e identificar o perfil dos participantes.

1. Qual a sua escolaridade? _____
2. Você tem CNH (Carteira Nacional de Habilitação)?
   a) Sim
   b) Não
3. Qual a sua idade? _____
4. Qual foi o sexo atribuído no seu nascimento?
   a) Masculino
   b) Feminino
   c) Prefiro não dizer
   d) Outro: _____
5. Quantos anos de experiência em direção de carros? _____

APÊNDICE D – ATIVIDADES PARA OS PARTICIPANTES

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

ATIVIDADES PARA OS PARTICIPANTES

Atividade 1 - Sem defeitos

1. Você observa algum sinal ou indicativo de defeito no veículo? Poderia descrever quais sinais ou indicativos levaram você a essa conclusão?
2. Você percebeu alguma indicação de que algum sensor estaria defeituoso? Se sim, qual?

   a) Sim, sensor de temperatura
   b) Sim, sensor de aceleração
   c) Sim, ambos os sensores
   d) Não, não percebi indicações

3. Com base nas informações apresentadas pelo aplicativo, você acredita que seu carro precisa de reparos imediatos?

   a) Sim
   b) Não

Atividade 2 - Sensor de temperatura

1. Você observa algum sinal ou indicativo de defeito no veículo? Poderia descrever quais sinais ou indicativos levaram você a essa conclusão?
2. Quantos defeitos você identificou no veículo?
3. Você identificou quais sensores estão afetados pelos defeitos? Se sim, quais?

   a) Sim, sensor de temperatura
   b) Sim, sensor de aceleração
   c) Sim, ambos os sensores
   d) Não, não percebi indicações

4. Após interagir com o aplicativo, como você avaliaria a gravidade do defeito apresentado?

   a) Baixa
   b) Média
   c) Alta

5. As informações apresentadas pelo aplicativo sugerem a necessidade de reparos imediatos no veículo?

a) Sim b) Não

Atividade 3 - Sensor de aceleração

1. Você observa algum sinal ou indicativo de defeito no veículo? Poderia descrever quais sinais ou indicativos levaram você a essa conclusão?
2. Quantos defeitos você identificou no veículo?
3. Você identificou quais sensores estão afetados pelos defeitos? Se sim, quais?

   a) Sim, sensor de temperatura
   b) Sim, sensor de aceleração
   c) Sim, ambos os sensores
   d) Não, não percebi indicações

4. As informações apresentadas pelo aplicativo sugerem a necessidade de reparos imediatos no veículo?

   a) Baixa b) Média c) Alta

5. Com base nas informações apresentadas pelo aplicativo, você acredita que seu carro precisa de reparos imediatos?

   a) Sim b) Não

Atividade 4 - Ambos os sensores

1. Você observa algum sinal ou indicativo de defeito no veículo? Poderia descrever quais sinais ou indicativos levaram você a essa conclusão?
2. Quantos defeitos você identificou no veículo?
3. Você identificou quais sensores estão afetados pelos defeitos? Se sim, quais?

   a) Sim, sensor de temperatura
   b) Sim, sensor de aceleração
   c) Sim, ambos os sensores
   d) Não, não percebi indicações

4. Após interagir com o aplicativo, como você avaliaria a gravidade do defeito no sensor de temperatura?

   a) Baixa b) Média c) Alta

5. Após interagir com o aplicativo, como você avaliaria a gravidade do defeito no sensor de aceleração?

a) Baixa b) Média c) Alta

6. As informações apresentadas pelo aplicativo sugerem a necessidade de reparos imediatos no veículo?

a) Sim b) Não

APÊNDICE E – ENTREVISTA PÓS-TESTE

UNIVERSIDADE FEDERAL DO CEARÁ
CAMPUS QUIXADÁ

ENTREVISTA PÓS-TESTE

Série de perguntas para colher opiniões, observações e sugestões sobre o sistema.

1. Qual o seu nível de satisfação com o aplicativo? (Nos níveis 1 a 5, onde 1 é baixo e 5 é alto)

   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5

2. Qual o nível de utilidade do aplicativo na sua opinião?

   ○ 1 ○ 2 ○ 3 ○ 4 ○ 5

3. O que você entendeu sobre este aplicativo?
4. O aplicativo compre com o que ele se propõe?

   a) Sim b) Não

5. Qual foi sua maior dificuldade ao usar o aplicativo?
6. Quais sugestões para melhoria do aplicativo você daria?