UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ

CEZAR HENRIQUE CONRADO

MATEUS KAEL IZAR

PONTE DE LIGAÇÃO ENTRE REDES AUTOMOTIVAS VEICULARES

CURITIBA

2024

CEZAR HENRIQUE CONRADO
MATEUS KAEL IZAR

PONTE DE LIGAÇÃO ENTRE REDES AUTOMOTIVAS VEICULARES

CONNECTION BRIDGE BETWEEN VEHICLE AUTOMOTIVE NETWORKS

Trabalho de Conclusão de Curso de Graduação apresentado como requisito parcial para obtenção do título de Bacharel em Engenharia Eletrônica do Curso de Bacharelado em Engenharia Eletrônica da Universidade Tecnológica Federal do Paraná.

Orientador: Prof. Dr. Rubens Alexandre de Faria

CURITIBA
2024

CEZAR HENRIQUE CONRADO
MATEUS KAEL IZAR

PONTE DE LIGAÇÃO ENTRE REDES AUTOMOTIVAS VEICULARES

Trabalho de Conclusão de Curso de Graduação apresentado como requisito parcial para obtenção do título de Bacharel em Engenharia Eletrônica do Curso de Bacharelado em Engenharia Eletrônica da Universidade Tecnológica Federal do Paraná.

Data de aprovação: 28/junho/2024

Rubens Alexandre de Faria
Doutor
Universidade Tecnológica Federal do Paraná

Luiz Fernando Copetti
Mestre
Universidade Tecnológica Federal do Paraná

Luciane Agnoletti dos Santos Pedotti
Doutora
Universidade Tecnológica Federal do Paraná

CURITIBA
2024

AGRADECIMENTOS

Aos nossos familiares que nos apoiaram nos momentos difíceis e nos ofereceram suporte para conseguir trilhar este caminho.

Aos professores, por todos os conselhos, pela ajuda e pela paciência com a qual guiaram o nosso aprendizado.

Aos nossos colegas de curso, com os quais foi possível enfrentar o dia a dia de forma mais leve e construir uma rotina de estudos de apoio mútuo e motivação contínua.

Not all those who wander are lost. (TOLKIEN, 1954)

RESUMO

A finalidade deste trabalho acadêmico é projetar e construir um protótipo que estabeleça uma ponte de ligação entre duas redes de comunicação de veículos pesados. O padrão J1939/CAN foi estabelecido em 2000 pela *Society of Automotive Engineers* (SAE) como protocolo automotivo para substituir a rede J1708/J1587. Entretanto, é uma prática comum na indústria automotiva que sistemas obsoletos coexistam com sistemas atualizados. Pensando nisso, será apresentado um protótipo com a capacidade de requisitar informações da rede J1587 através da rede J1939 de maneira adaptável, onde o operador poderá escolher via mensagem CAN quais dados deseja resgatar da outra rede. Objetiva-se, desta forma, superar alternativas já presentes no mercado, onde o operador está restrito a uma lista pré-estabelecida de mensagens.

Palavras-chave: redes automotivas; sae j1587; sae j1939; comunicação veicular.

ABSTRACT

The purpose of this thesis is to design and build a prototype that establishes a connecting bridge between two heavy vehicle communication networks. The standard J1939/CAN was established in 2000 by the Society of Automotive Engineers (SAE) as automotive protocol to replace the J1708/J1587 network. However, it is a common practice in automotive industry that obsolete systems coexist with updated systems. Therefore, the prototype will be presented with the ability to request information from the J1587 network through the J1939 network in an adaptive way, where the operator can choose via CAN message which data you want to retrieve from the other network. In this way, the aim is to overcome alternatives already present on the market, where the operator is restricted to a pre-established list of messages.

Keywords: automotive networks; sae j1587; sae j1939; vehicular communication.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

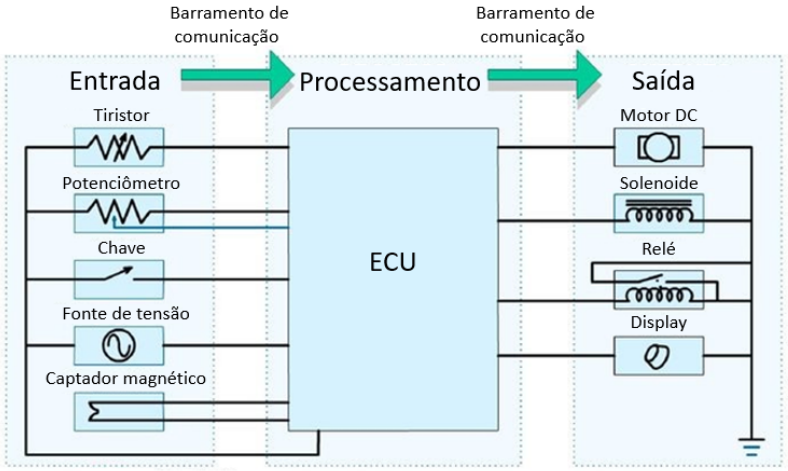
É difícil imaginar a vida sem um *software* embarcado. Isso porque dispositivos com *software* embarcado incluem *smartphones*, satélites, eletrodomésticos, marca-passos, sistemas de geração e distribuição de energia elétrica e unidades de controle eletrônico (EBERT; JONES, 2009).

Uma unidade de controle eletrônico, do inglês *Electronic Controlled Unit* (ECU), é um dispositivo de *software* embarcado responsável por controlar uma ou mais funções específicas de um veículo. Na atualidade, os veículos podem conter 100 ECUs ou mais, que controlam funções essenciais como motor e freio, funções secundárias como ar condicionado e funções de segurança e telemática (INSIDER, 2020).

As ECUs inicialmente começaram a ser utilizadas nos anos 1970, controlando os solenóides do carburador. Na década seguinte, com a introdução da injeção de combustível, as ECUs se tornaram responsáveis pelo gerenciamento de combustível e controle de ignição em veículos de motores a diesel. Nos anos 1990 a segurança veicular passou a ser uma das funções controladas eletronicamente. O controle do acelerador, turboalimentador e de vários sistemas de emissão passou a ser eletrônico a partir dos anos 2000. Nos anos 2010 adiante, as ECUs passam a ter controle completo sobre esses sistemas e diversas outras funções nos veículos, chegando a ter mais de cem entradas e saídas (AUTOPI, 2023).

Para coordenar esses sistemas, as ECUs recebem os sinais de entrada através de sensores e processam a informação através de atuadores para controlar a função a ser executada. Uma ECU pode receber sinais de entrada diretamente de outra ECU. As informações são transportadas dentro do veículo através de barramentos de comunicação como a Rede de Área Controlada, do inglês *Controller Area Network* (CAN), a Rede Interconectada Local, do inglês *Local Interconnect Network* (LIN) e a Rede Ethernet (ALAM *et al.*, 2019). O sistema está ilustrado na figura 1.

Figura 1 – Sistema de Controle Eletrônico



Fonte: Adaptado de Tharad (2019).

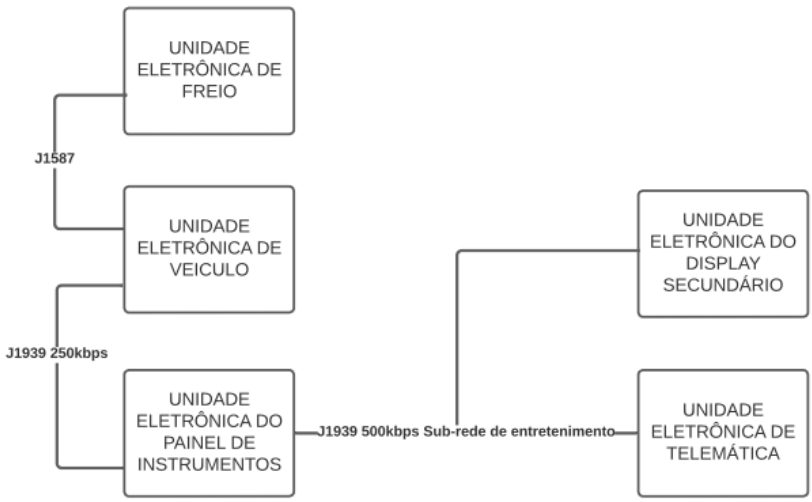
O CAN é um barramento de comunicação de dados em série predominante nos veículos de hoje. É um dos mais bem sucedidos protocolos de comunicação em termos de volumes de aplicação e de aceitação entre ramos da indústria, considerando os últimos vinte anos (EISELE, 2012).

Publicado em 1986, o protocolo J1587 da Sociedade de Engenheiros Automotivos, do inglês *Society of Automotive Engineers* (SAE), define as mensagens transmitidas na rede SAE J1708. O J1708 que especifica o *link* de dados e as camadas físicas, enquanto o J1587 especifica as camadas de transporte, rede e aplicação. No entanto, o J1587 se tornou obsoleto e está sendo substituído pelo SAE J1939 (SIMMASOFTWARE, 2023). O SAE J1939 é um protocolo de comunicação que opera baseado na camada física do protocolo CAN e especifica como a informação é repassada entre as ECUs de um veículo. Este vem sido utilizado em aplicações de diagnóstico e controle nas indústrias de transporte, máquinas agrícolas e equipamentos submarinos (VECTOR, 2023).

Realizar a leitura, interpretação, armazenamento e controle dos dados é uma atividade corriqueira de um engenheiro da área automotiva. Tarefas como realizar diagnóstico veicular, comprovar uma modificação requisitada ou funcionalidade oferecida incluem coletar, armazenar e analisar a comunicação realizada entre as ECUs (DALMOLIN, 2022).

Sendo a J1587/J1708 uma rede de comunicação obsoleta, as ferramentas de análise estão muito aquém das ferramentas de análise de dados CAN. *Softwares* comuns na indústria, como o CANalyzer, possuem uma gama de ferramentas muito maior para o J1939 do que para os módulos que utilizam J1587. Uma vez que existem atualmente sistemas que utilizam ambos os protocolos, como visto na figura 2, nota-se não só a necessidade de analisar dados de ambos os barramentos em uma mesma ferramenta, como também utilizar o ferramental mais avançado.

Figura 2 – Exemplo de arquitetura eletrônica utilizando J1587 e J1939.



Fonte: Autoria própria (2024).

Dessa forma, há a motivação para a implementação da ponte de ligação entre as duas redes automotivas veiculares. A transformação do protocolo obsoleto para o protocolo atualmente considerado como padrão irá facilitar a análise de problemas automotivos. Será possível utilizar as ferramentas atuais do protocolo J1939 para análise do J1587, devido à conversão proposta no trabalho. Isso representa um ganho para a indústria automotiva, dado que não existe uma forma prática de conversão entre protocolos e ainda é necessário trabalhar com o protocolo obsoleto quando necessário.

1.1 Objetivos

1.1.1 Objetivo geral

Desenvolver um dispositivo que permita requisitar informações do protocolo J1587 através do protocolo J1939, com base no referencial teórico adquirido. Validar o protótipo através de testes em veículo.

1.1.2 Objetivos específicos

O trabalho proposto tem como objetivos específicos:

- Projetar um circuito elétrico com transdutor J1939, transdutor J1587 e microcontrolador;
- Desenvolver um sistema embarcado para o microcontrolador para realizar a tradução bidirecional entre os sinais J1939 e J1587;
- Desenvolver o algoritmo para a requisição de dados privados da rede J1587;
- Testar o sistema em bancada;
- Validar o dispositivo no veículo.

1.2 Estrutura do trabalho

O texto está dividido em 5 capítulos. No capítulo 2 tem-se a apresentação dos referenciais teóricos (arquiteturas eletrônicas, unidade de controle eletrônico, redes de comunicação, protocolos CAN, J1587/J1708 e J1939).

No capítulo 3 são apresentados os materiais utilizados na construção do protótipo e os métodos utilizados no seu desenvolvimento e verificação. No capítulo 4 são discutidos os resultados obtidos e suas interpretações. As conclusões são parte do capítulo 5, bem como considerações sobre futuros trabalhos que podem ser realizados dentro do mesmo tema.

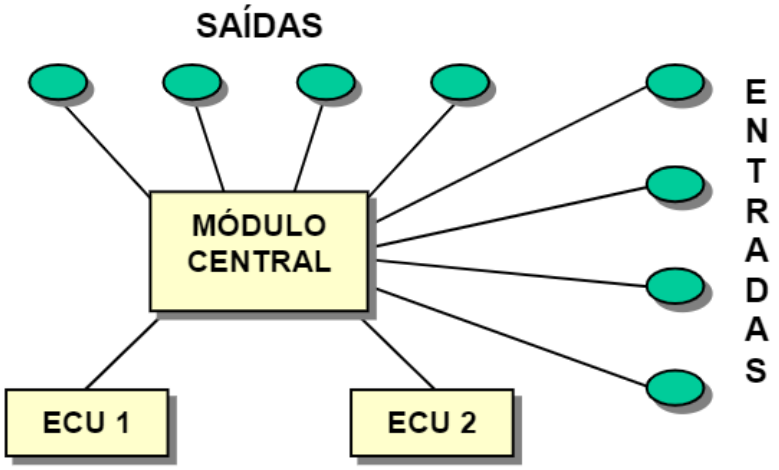
2 REFERENCIAL TEÓRICO

2.1 Arquiteturas eletrônicas

Conforme aumenta o número de módulos eletrônicos em um veículo, surge a necessidade de pensar nas implementações dos sensores e dos sistemas de controle. Esta forma de conexão em uma dada aplicação embarcada é chamada de arquitetura eletrônica (POWERS; KIRSON; ACTON, 1998).

Segundo Guimarães e Saraiva (2002), uma das formas de conexão dos módulos é a arquitetura centralizada, que pode ser vista na figura 3.

Figura 3 – Esquemático de arquitetura centralizada.



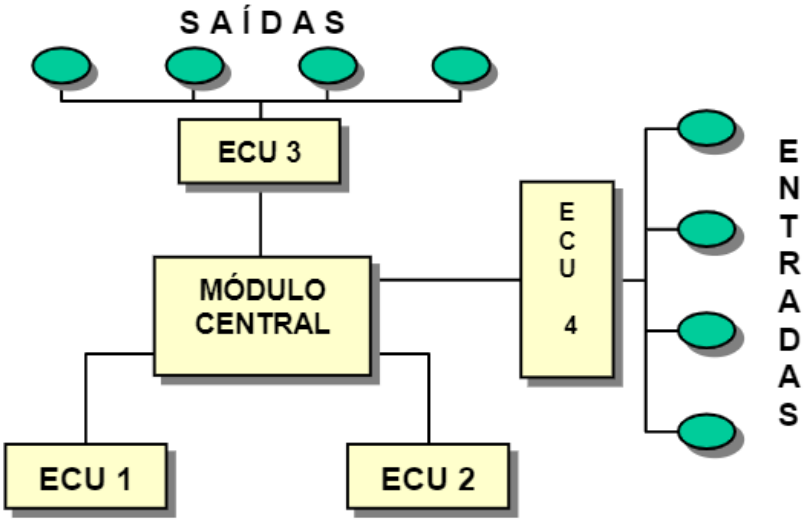
Fonte: Guimarães e Saraiva (2002).

Este tipo de arquitetura tem como vantagem o fato de que os dados de todas as unidades de controle eletrônico estarão disponíveis no módulo central ao mesmo tempo e a possibilidade de utilizar *hardware* simples, já que as conexões são somente os sensores, o cabeamento e o módulo eletrônico que gerencia o sistema. Como desvantagens, tem-se a limitação de expansão de sistema e a grande quantidade de cabeamento utilizado para fazer todas as conexões, que no caso de um veículo pesado aumentaria bastante o peso do sistema (STRAUSS; CUGNASCA; SARAIVA, 1998).

A segunda forma é conectar as ECUs de forma que fiquem espalhadas pelo veículo, mais próximo ao ponto de instalação dos sensores e dos dispositivos a serem controlados. Essa é a arquitetura distribuída, que está exemplificada na figura 4.

Sendo assim, a quantidade de cabeamento é reduzida significativamente, e há a possibilidade de expansão. Com a modularização do sistema, aumenta também a confiabilidade através da facilidade de validação dos módulos de maneira isolada. No entanto, como muitos dos dados são enviados no mesmo barramento, precisa-se determinar o Protocolo de Comunicação, que é o meio de comunicação entre as ECUs.

Figura 4 – Esquemático de arquitetura distribuída.



Fonte: Guimarães e Saraiva (2002).

Isso implica na necessidade de definição de prioridades e taxas de transmissão de dados na rede ideais para cada aplicação e na existência de um *software* de controle. A taxa de transmissão ideal afeta os tempos internos do *software* de controle e a escolha dos componentes eletrônicos a serem utilizados no projeto (FREDRIKSSON, 1997).

Uma das maiores dificuldades da engenharia de desenvolvimento de produto de uma montadora de veículos é determinar a arquitetura elétrica de um novo modelo. Normalmente, os produtos começam a ser desenvolvidos alguns anos antes do lançamento, o que dificulta ainda mais essa tomada de decisão. Para decidir a arquitetura eletrônica apropriada, deve-se considerar a complexidade do sistema que será controlado (o número de variáveis de entrada e saída e o tamanho do sistema), a disponibilidade dos dispositivos eletrônicos requeridos, o tempo necessário de implementação (considerando projeto, desenvolvimento dos protótipos, validação e instalação final) e o orçamento (GUIMARÃES, 2003).

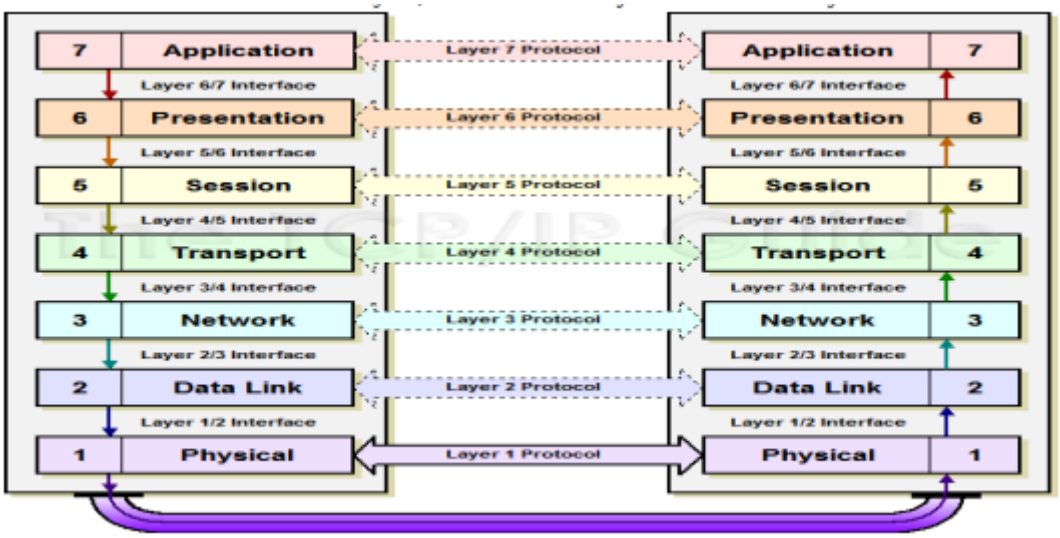
Considerando a aplicação automotiva, a arquitetura distribuída é a melhor solução para conectar as ECUs ao módulo central. A partir disso, o protocolo de comunicação utilizado passa a ser o desafio do desenvolvimento da arquitetura.

2.2 Camadas do protocolo OSI

Para que seja possível que dois sistemas consigam se comunicar com sucesso, é necessário definir um protocolo padrão de comunicação. A *International Organization for Standardization* (ISO) desenvolveu o modelo *Open Systems Interconnection* (OSI) com o objetivo de definir os padrões e tarefas necessárias para que dois sistemas sejam capazes de se comunicar com sucesso.

Segundo Kumar, Dalal e Dixit (2014), o modelo OSI consiste em subdividir um sistema de comunicação em partes menores denominadas de camadas. Uma camada no protocolo de comunicação representa um conjunto funções conceitualmente similares. Com isso, o modelo OSI prevê que o sistema de comunicações deve ser subdividido em 7 camadas, como representado na figura 5.

Figura 5 – Camadas do protocolo de comunicação do modelo OSI



Fonte: Kumar, Dalal e Dixit (2014).

2.2.1 Camada física

A camada física define todas as especificações elétricas e físicas para a transmissão de dados. Define a relação do dispositivo com o meio físico que ele transmite dados (e.g. fibra óptica, cobre, ar). Inclui também o *layout* de pinos, tensões, impedância, especificação de cabos (se houver), tempo de transmissão, entre outros. Essa camada realiza apenas a transmissão de dados, sem garantir a confiabilidade e consistência dos dados transmitidos.

2.2.2 Camada de enlace de dados

A camada de enlace de dados é responsável por garantir uma transmissão de dados confiável e consistente entre dois nós adjacentes do sistema. Para atingir esse objetivo, a camada de enlace de dados realiza a detecção e controle de erros, geralmente implementada por um algoritmo conhecido como *Cyclic Redundancy Check* (CRC). Lembrando que a camada de enlace de dados garante a consistência e confiabilidade na transmissão de dados entre dois nós adjacentes do sistema; garantir consistência e confiabilidade na transmissão de dados de ponta a ponta é responsabilidade das camadas superiores.

2.2.3 Camada de rede

A camada de rede possui como principais atividades o roteamento e o endereçamento lógico. Esta camada atribui endereços lógicos aos pacotes que são transmitidos para que as mensagens cheguem ao destino correto. Adicionalmente, a camada de rede pode implementar a transmissão de dados quebrando uma mensagem em vários fragmentos, enviando cada fragmento por uma rota diferente e reagrupando os fragmentos quando a mensagem chegar no destino desejado. Usualmente, essa funcionalidade de segmentação é realizada pela camada de transporte.

2.2.4 Camada de transporte

A camada de transporte é responsável por garantir a transmissão de dados de ponta a ponta. Nessa camada, é feito o controle de erros, controle de conexão, controle de fluxo, segmentação e reconstrução de mensagens. A camada de transporte possui dois protocolos definidos para transmissão de dados: o *Transmission Protocol Control* (TCP) e o *User Datagram Protocol* (UDP). O protocolo TCP garante a consistência e confiabilidade da transmissão, realizando a verificação de que a mensagem chegou corretamente ao destino através de técnicas de *three-way handshake*, *acknowledgement* e controle de fluxo. O protocolo UDP não garante a consistência confiabilidade da transmissão.

2.2.5 Camada de sessão

A camada de sessão é responsável por criar e manter a conexão entre dois sistemas durante o tráfego de dados. A conexão entre dois sistemas é denominada de sessão. Quando todos os dados são enviados pelo sistema, a sessão é encerrada. Ao iniciar uma nova sessão, a camada testa primeiramente se o sistema que deseja se conectar está disponível para conexão. Caso esteja disponível, a sessão é criada e o tráfego de dados é iniciado; caso não esteja disponível, a camada de sessão envia uma mensagem para a camada de apresentação informando sobre o erro e mostrando ao usuário através da camada de aplicação.

2.2.6 Camada de apresentação

A camada de apresentação pode ser interpretada como um tradutor. No sistema que envia dados, essa camada converte os dados da camada de aplicação para o formato que deve ser utilizado para transmissão desses dados; já no sistema que recebe os dados, a camada de apresentação converte os dados recebidos em um formato que a camada de aplicação consiga interpretar. A camada de apresentação é onde estão contidos os conceitos de codificação de

caracteres (e.g. ASCII, UTF-8), conversão de dados (e.g. *bit order*, CR-CR/LF), compressão de dados e criptografia de dados.

2.2.7 Camada de aplicação

A camada de aplicação é onde geralmente um usuário tem interação com o sistema. É nessa camada que os dados recebidos são exibidos através de páginas de navegador, mensagens de aplicativos de chat e assim por diante. Vários protocolos são executados nessa camada, como, por exemplo, DNS, FTP, HTTP, HTTPS, NFS, POP3, SMTP e SSH.

2.3 Comunicação serial RS232

A comunicação e transmissão de dados sempre foi um dos campos mais desafiadores e tem evoluído constantemente. Dados, significando essencialmente uma informação representada por um bit 0 ou por um bit 1, precisam ser enviados de um ponto a outro de um sistema. Dada essa premissa, surgiu a necessidade de padronizar a forma como os dispositivos de um sistema se comunicam entre si. Surgiram então regras denominadas de protocolos de comunicação para que um transmissor, ou *Data Terminal Equipment* (DTE), consiga se comunicar com sucesso com um receptor, ou *Data Communication Equipment* (DCE). (DAWOUD; DAWOUD, 2020)

Existem basicamente duas classificações principais no que diz respeito aos protocolos de comunicação, que é a comunicação serial e a comunicação paralela. Na comunicação serial, é transmitido um bit por vez de forma sequencial, enquanto na comunicação paralela são transmitidos vários bits de forma simultânea. Atualmente, o protocolo de comunicação serial mais popular é a especificação EIA/TIA-232-E. Esse protocolo, desenvolvido pela *Electronic Industry Associations* (EIA) e pela *Telecommunications Industry Association* (TIA) em 1962, é popularmente referenciado como RS232, onde RS significa *Recommended Standard*. (DAWOUD; DAWOUD, 2020)

O protocolo RS232 funciona de forma assíncrona, sem a necessidade de *clock* para sincronização entre o transmissor e o receptor. Porém, tanto o transmissor quanto o receptor precisam se comunicar com um *baud rate* definido, geralmente 19200 kbps ou 115202 kbps. Os dados são transmitidos com um *byte* por vez. No início de cada *byte*, o receptor deve se ressincronizar para fazer a leitura correta da mensagem transmitida. Para que isso seja possível, o início de uma mensagem é sempre composto por um bit denominado *Start bit* e o fim da mensagem é composto por um bit denominado *Stop bit*. Entre o *Start bit* e o *Stop bit*, tem-se o conteúdo da mensagem e mais um bit de paridade para controle de erros. O conjunto de bits que agrupa o *Start bit*, os dados da mensagem, o bit de paridade e o *Stop bit* é denominado de

frame. *Frame* é a menor unidade de informação que pode ser transmitida em uma comunicação serial. A figura 6 exemplifica como é a composição de um *frame* no protocolo RS232.

Figura 6 – *Frame do protocolo RS232*

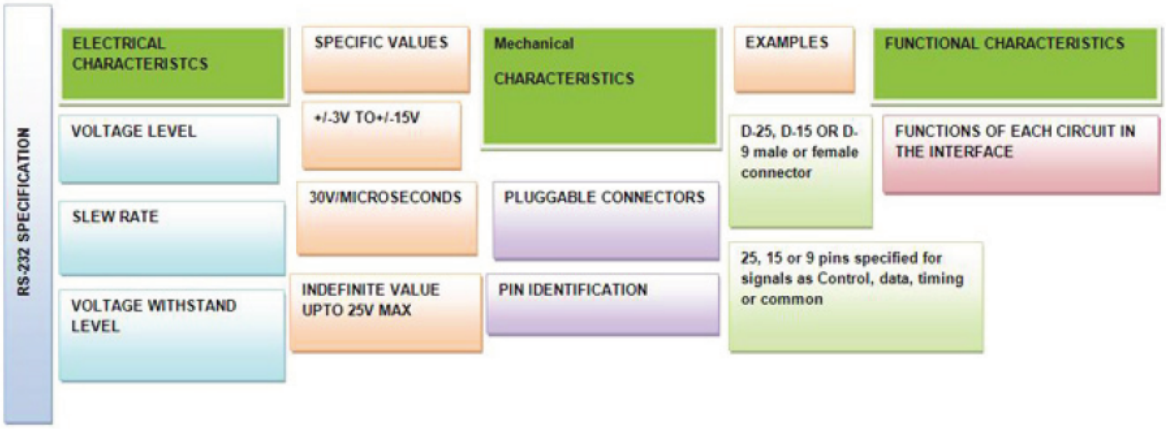
Fonte: Dawoud e Dawoud (2020).

O bit *Start bit* é sempre enviado com o nível lógico 0, representando o início da trasmissão da mensagem, de forma que o receptor pode se sincronizar para realizar a leitura da mensagem. Em seguida, tem-se os dados da mensagem sendo enviados, podendo conter sete ou oito bits, onde o bit menos significativo é enviado por primeiro e o bit mais significativo é enviado por último. Após os dados da mensagem, é enviado o bit de paridade, cuja função é a detecção de erros na transmissão, contando a quantidade de bits 1 enviados no frame. O bit de paridade é uma forma rudimentar de detecção de erros, onde é recomendado que outros mecanismos de detecção e correção de erros sejam implementados. Finalmente, tem-se o *Stop bit* sendo enviado após o bit de paridade, representando o fim do *frame*. O *Stop bit* é sempre enviado com o nível lógico 1.

O protocolo RS232 define, além do *frame*, as características físicas do meio de transmissão que devem ser seguidas. A especificação do protocolo RS232 define as características elétricas, como nível lógico de tensão e velocidade de resposta (*slew rate*), características funcionais e características mecânicas, como conectores e identificação de pinos. A figura 7 sumariza as principais características do protocolo RS232.

O nível lógico de tensão é um dos principais itens definido pela especificação do protocolo RS232. Com a definição do nivel lógico de tensão, os receptores conseguem identificar qual é o nível lógico sendo transmitido através da diferença de potencial no barramento. A especificação do protocolo RS232 prevê que o bit de nível 0, também conhecido como *Space*, é obtido a partir de uma diferença de potencial entre +3V e + 15V e o bit de nível 1, também conhecido como *Mark*, é obtido a partir de uma diferença de potencial entre -3V e -15V. A região com diferença de potencial entre -3V e +3V é denominada de região de

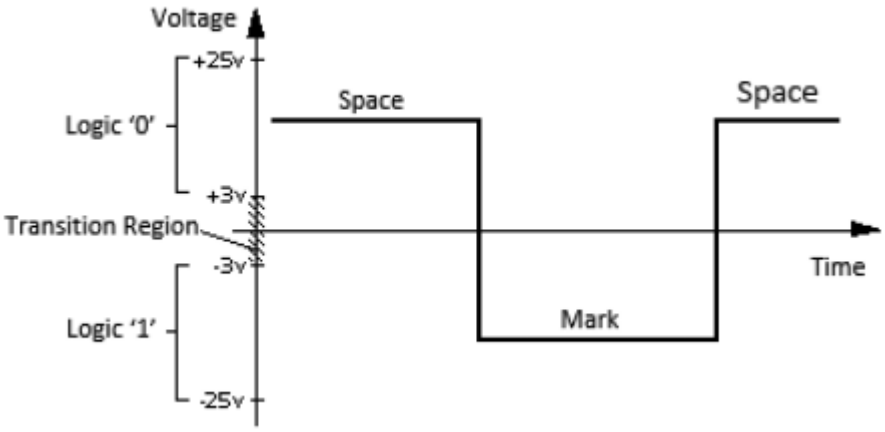
Figura 7 – *Especificações técnicas do protocolo RS232*



Fonte: Dawoud e Dawoud (2020).

transição. O modo de operação do protocolo RS232 prevê que a conexão entre transmissor e receptor é realizada com dois fios. Um fio é conectado no terra e o outro fio é responsável por gerar a diferença de potencial para transmissão dos níveis lógicos.

Figura 8 – *Nível lógico de tensão do protocolo RS232*

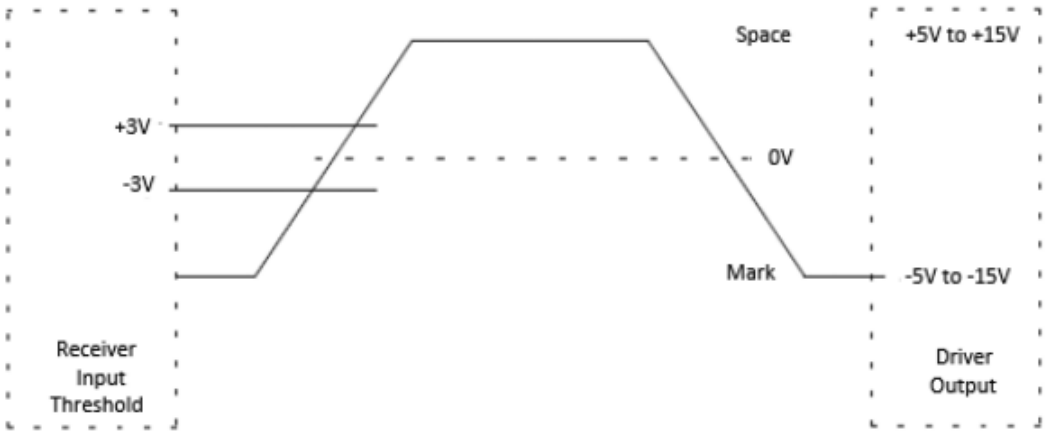


Fonte: Dawoud e Dawoud (2020).

O protocolo RS232 também apresenta uma especificação limitando a velocidade de resposta a uma variação de tensão. Essa restrição foi incluída com o intuito de diminuir a probabilidade de interferência indesejada entre sinais adjacentes. Dada essa premissa, a especificação do protocolo RS232 define a velocidade máxima de resposta em 30V/μs e a taxa máxima de transmissão em 20kbps. A figura 9 demonstra os níveis lógicos e os tempos de subida e descida.

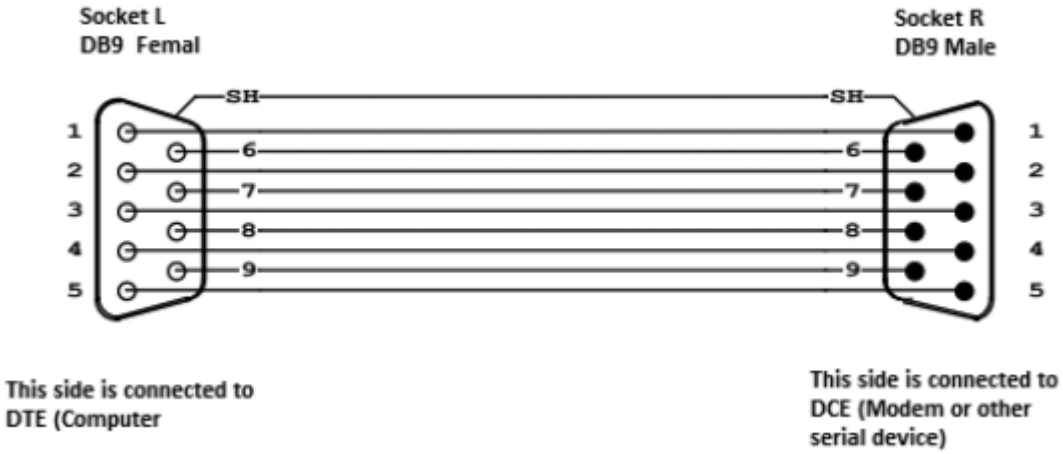
O protocolo de comunicação RS232 define os conectores que devem ser utilizados para a conexão entre transmissor e receptor. Os conectores definidos pela especificação do protocolo RS232 são os conectores DB9 e DB25. A figura 10 representa a conexão realizada entre um conector DB9 macho e um conector DB9 fêmea.

Figura 9 – *Nível lógico e velocidade de resposta no protocolo RS232*



Fonte: Dawoud e Dawoud (2020).

Figura 10 – *Conexão entre conectores DB9*



Fonte: Dawoud e Dawoud (2020).

2.4 Comunicação serial RS485

O protocolo RS232 é uma interface que conecta um transmissor (DTE) a um receptor (DCE) com uma taxa máxima de transmissão de 20kbps e um cabo com comprimento máximo de 15 metros. Devido a essas limitações, a *Electronic Industry Associations* (EIA) e a *Telecommunications Industry Association* (TIA) desenvolveram o protocolo RS485 com o objetivo de conseguir conectar vários dispositivos no mesmo barramento, comunicando em maiores distâncias e com uma taxa maior de transmissão de dados. (DAWOUD; DAWOUD, 2020)

Um dos principais problemas com a comunicação serial RS232 é a grande interferência sofrida no barramento. Como os níveis lógicos de tensão são medidos a partir de um terra como sinal de referência, o protocolo RS232 fica suscetível a ruídos e interferências, diminuindo a distância com que o sinal pode chegar de forma íntegra. O protocolo de comunicação serial

RS485, em contrapartida, prevê que o nível lógico de tensão é gerado a partir da diferença de potencial entre dois fios Sig+ e Sig-, sem depender do sinal de terra. Esse modo de operação garante que a transmissão de dados seja menos suscetível a ruídos e interferências, onde a informação pode ser transmitida a maiores distâncias e com uma maior taxa de transmissão. (DAWOUD; DAWOUD, 2020)

Os níveis lógicos transmitidos no barramento são definidos pela diferença de potencial entre os dois sinais gerados. Uma diferença de potencial maior que 0,2V representa um nível lógico 0, enquanto que uma diferença de potencial menor que -0,2V representa o nível lógico 1. O protocolo RS485 não padroniza nenhum modelo de conector a ser utilizado, podendo ter um cabo de comprimento de até 1200 metros. O protocolo RS485 prevê também que possam ser conectados ao barramento 32 dispositivos.

2.5 Comunicação SPI

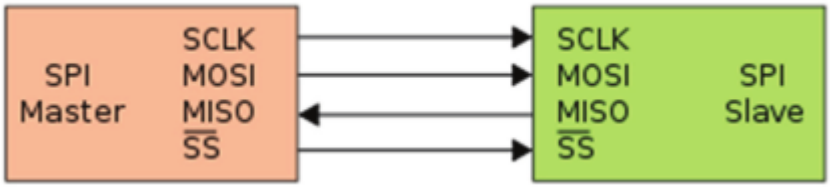
Serial Peripheral Interface (SPI) é uma das interfaces de comunicação mais populares utilizadas em sistemas embarcados. Desde a introdução no fim da década de 1980 pela Motorola, o protocolo de comunicação SPI tem sido usado amplamente em sistemas embarcados para curtas distâncias. O protocolo SPI está disponível para uso em praticamente todas as arquiteturas, incluindo 8051, x86, ARM, PIC, AVR, MSP, entre outras. (DAWOUD; DAWOUD, 2020)

SPI é um protocolo de transferência de dados síncrono e serial onde dois ou mais dispositivos seriais são conectados entre si em modo *full-duplex* . A arquitetura utilizada no protocolo é a arquitetura *master-slave*. A interface SPI suporta que um barramento possua mais de um *slave*, porém pode existir apenas um *master*. O dispositivo *master* é o responsável pela criação e controle da conexão, originando os *frames* para escrita e leitura. Assim que a conexão é iniciada, o *master* e os *slaves* podem ambos transmitir e receber dados simultaneamente; por se tratar de uma conexão *full-duplex*, o *master* pode transmitir dados para um *slave* e o *slave* pode transmitir dados para o *master* simultaneamente. Apenas o *master* pode controlar o sinal de *clock*, que é utilizado para sincronização dos dispositivos (DAWOUD; DAWOUD, 2020).

A interface SPI é composta por um barramento com quatro fios. O barramento é representado pela figura 11, que é composto pelos seguintes sinais:

- *Master out Slave in* (MOSI);
- *Master in Slave out* (MISO);
- *Serial clock* (SCLK);
- *Slave select* (SS) ou Chip Select (CS).

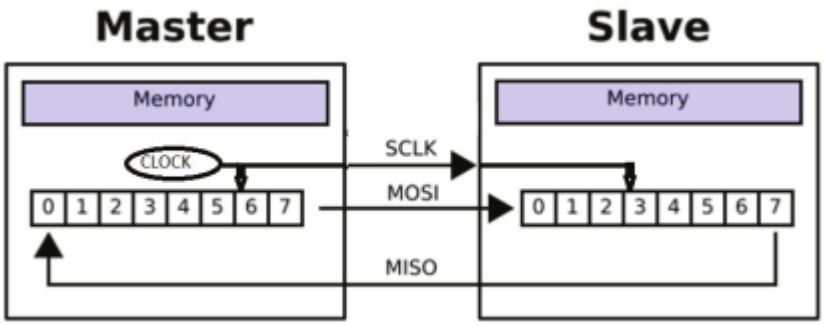
Figura 11 – *SPI single master to slave*



Fonte: Dawoud e Dawoud (2020).

Toda a operação e transferência de dados do protocolo SPI é realizada através dos registradores de deslocamento (*shift registers)*. Cada dispositivo, seja um dispositivo *master* ou *slave*, possui um registrador de deslocamento de 8 bits. Um requisito para implementar o protocolo SPI é que o dispositivo *master* e os dispositivos *slave* devem ser conectados de mode que formem um *buffer* circular. Desse modo, a saída do registrador de deslocamento do dispositivo *master* (MOSI) deve ser conectado na entrada do registrador de deslocamento do dispositivo *slave*. A saída do registrador de deslocamento do dispositivo *slave* (MISO) deve ser conectada na entrada do registrador de deslocamento do dispositivo *master*. Isso garante que o protocolo SPI realize a transmissão de dados através de um buffer circular (DAWOUD; DAWOUD, 2020). Essa conexão que deve ser implementada no protocolo SPI é exemplificada pela figura 12.

Figura 12 – *SPI Hardware*

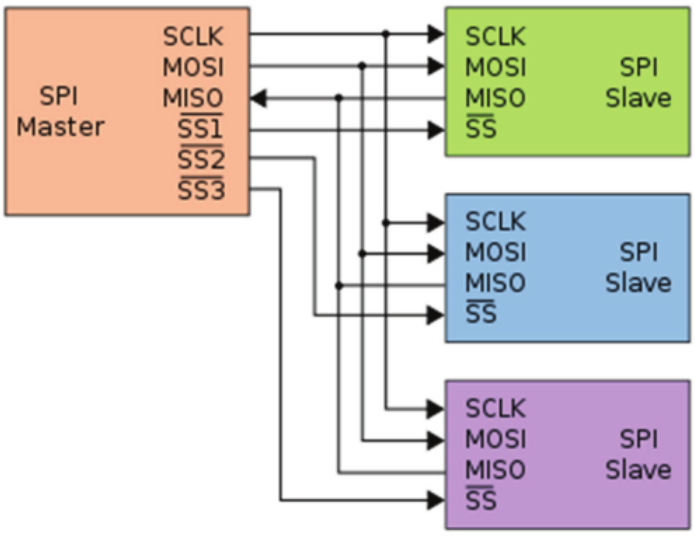


Fonte: Dawoud e Dawoud (2020).

O sinal *Slave select* (SS) é utilizado pelo dispositivo *master* para definir com qual dispositivo *slave* será feita a transmissão e recepção de dados. Duas configurações são possíveis no protocolo SPI quando há mais de um dispositivo *slave* conectado ao barramento, que são a **Configuração em paralelo** e a **Configuração em cascata**. Na configuração em paralelo, todos os dispositivos *slave* são conectados diretamente ao dispositivo *master*, porém apenas um dispositivo *slave* ficará ativo por vez durante a transmissão de dados. Na configuração em cascata, os dispositivos *slave* são conectador entre si e com o dispositivo *master*, onde todos sempre ficam ativos durante a transmissão de dados (DAWOUD;

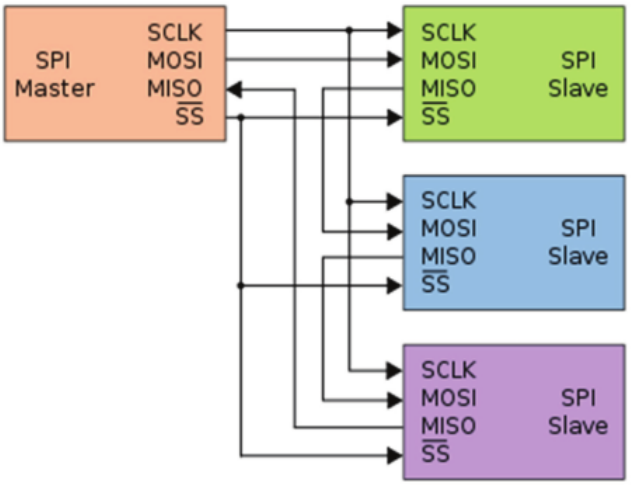
DAWOUD, 2020). A figura 13 exemplifica como é feita a configuração em paralelo e a figura 14 exemplifica como é feita a configuração em cascata.

Figura 13 – *SPI Configuração em paralelo*



Fonte: Dawoud e Dawoud (2020).

Figura 14 – *SPI Configuração em cascata*



Fonte: Dawoud e Dawoud (2020).

2.6 SAE J1587

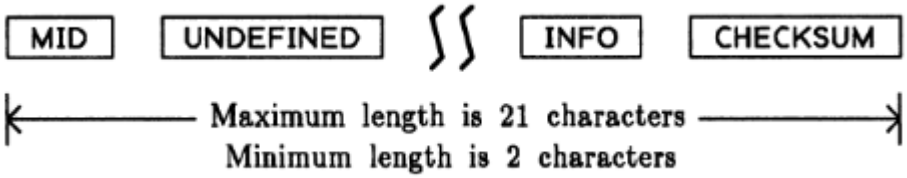
O protocolo SAE J1587 é uma extensão do protocolo SAE J1708. O protocolo SAE J1708 foi desenvolvido pela SAE com o intuito de padronizar a comunicação entre unidades eletrônicas de controle de veículos pesados, como caminhões e ônibus. O protocolo SAE J1939 foi desenvolvido para substituir os protocolos J1708/J1587, porém ainda existem inúmeros sistemas legados que ainda utilizam o protocolo J1587.

Para melhor entender o protocolo SAE J1587, é necessário primeiro revisar os conceitos associados com o protocolo SAE J1708. O protocolo SAE J1708 segue o modelo de comunicação OSI, implementando a camada física e parte da camada de enlace de dados (STEPPER, 1990).

No que diz respeito à camada física, o protocolo SAE J1587 define que a comunicação entre unidades eletrônicas de controle devem seguir o padrão de comunicação serial RS485 (BASIC, 2004) e a taxa de transmissão de dados deve ser de 9600bps (STEPPER, 1990). Os níveis lógicos, assim como no protocolo SAE J1939, são definidos a partir da diferença de potencial entre dois fios trançados que representam o barramento serial. Uma diferença maior que +0.2V no barramento representa o nível lógico **0** e uma diferença menor que -0.2V representa o nível lógico **1** (SAASTAMOINEN, 2008).

No que diz respeito à camada de enlace de dados, o protocolo SAE J1708 define que uma mensagem transmitida deve possuir entre 2 e 21 caracteres. Um caractere é definido como um campo de 10 bits, sendo 1 bit indicando o ínicio do caractere, oito bits contendo dados e 1 bit indicando o fim do caractere. O primeiro caractere da mensagem deve ser obrigatoriamente um campo chamado *Message Identifier* (MID), que define o endereço do transmissor da mensagem. O último caracter da mensagem é chamado de *CheckSum*, que é gerado aplicando o complemento de 2 na mensagem. Entre o *Message Identifier* e o *CheckSum*, tem-se os dados da mensagem. A figura 15 representa a definição de mensagem padronizada pelo protocolo SAE J1708.

Figura 15 – Formato de mensagem do protocolo J1708

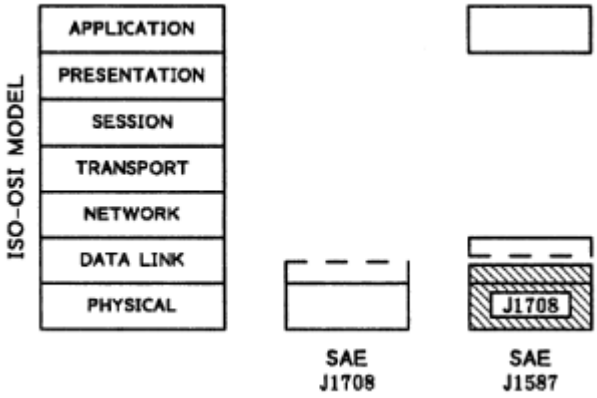


Fonte: Stepper (1990).

O protocolo SAE J1708 não define o padrão que deve ser seguido para a transmissão dos dados da mensagem. Cada um dos protocolos que extendem o SAE J1708 devem definir o seu padrão de envio de dados da mensagem. Com isso, o protocolo SAE J1587 tem a responsabilidade de definir um padrão para o envio dos dados da mensagem, implementando a camada de enlace de dados e a camada de aplicação. A figura 16 representa como cada um dos protocolos é implementado de acordo com o modelo OSI.

Com base no modelo de mensagem definido pelo protocolo SAE J1708, o protocolo SAE J1587 surgiu com o objetivo padronizar e definir como será a composição dos dados de uma mensagem. Desse modo, o protocolo SAE J1708 definiu um campo chamado de *Parameter Identification* (PID), que define qual é a informação que está sendo representada na mensagem. O PID é representado por um caractere e a quantidade de caracteres do dado vinculado com o

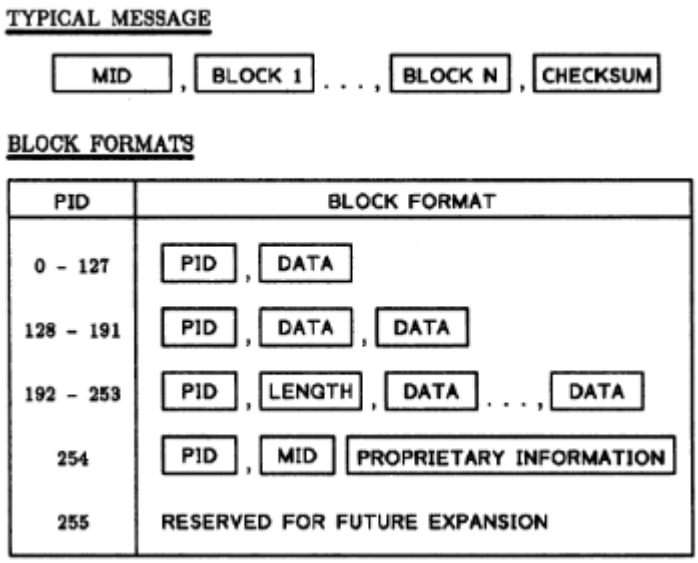
Figura 16 – Modelo OSI dos protocolos J1708 e J1587



Fonte: Stepper (1990).

PID é variável. Um PID em conjunto com os dados é denominado de bloco dentro da mensagem do SAE J1708, onde uma mensagem pode conter vários blocos, desde que caibam dentro de uma mensagem de 21 caracteres. A figura 17 demonstra a subdivisão de uma mensagem SAE J1708 em blocos.

Figura 17 – Formato de um bloco de uma mensagem do protocolo J1587



Fonte: Stepper (1990).

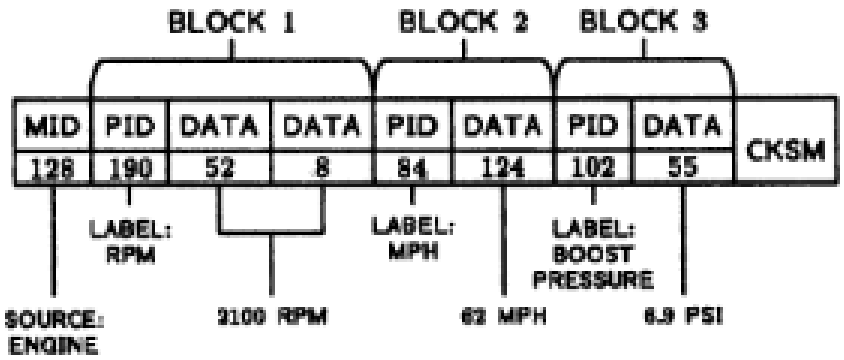
De acordo com o número identificador do PID, tem-se a seguinte definição quanto à quantidade de caracteres dos dados enviados:

- 0 - 127: um PID dentro dessa faixa de identificação possui apenas um caractere de dados;
- 128 - 191: um PID dentro dessa faixa de identificação possui dois caracteres de dados;

- 192 - 253: um PID dentro dessa faixa não possui um número de caracteres definidos. O segundo caractere do bloco representa o tamanho do bloco;
- 254: é conhecido como *Data Link Escape PID* e é reservado para os fabricantes;
- 255: PID que representa que a segunda página de identificação do protocolo SAE 1587 está sendo utilizada (faixa 256 - 511).

Toda a padronização de PID está contemplada nos documentos da SAE. Em relação ao MID, a faixa a partir do ID 128 é reservada especificamente para o protocolo SAE J1587, enquanto que os valores menores que 128 representam as mesmas ECU's padronizadas a partir do protocolo SAE J1708. Um exemplo de mensagem que é enviada ao barramento serial está representada a partir da figura 18. Nessa mensagem, tem-se o caractere do MID com o valor 128, que representa, de acordo com a padronização da SAE, o identificador do motor. Na mensagem em questão, tem-se 3 blocos de mensagens sendo enviadas ao barramento, com os respectivos valores de PID sendo 190, 84 e 102; a padronização SAE J1587 prevê que os PID's são, respectivamente, associados com a rotação do motor (RPM), a velocidade do veículo (MPH) e a pressão do dínamo (PSI). O PID com a informação da rotação do motor está na faixa em que são necessários dois caracteres para compor os dados transmitidos, enquanto os PID's de velocidade do veículo e pressão do dínamo estão na faixa em que é previsto o envio de apenas um caractere de dado. Ao fim da mensagem, tem-se o *checksum*, calculado a partir do complemento de dois.

Figura 18 – Exemplo de uma mensagem do protocolo SAE J1587



Fonte: Stepper (1990).

É interessante destacar que existe um PID específico para requisitar informações de uma unidade eletrônica de controle específica, que é o PID 128. O PID 128 está contemplado na segunda faixa que vai de 128-191, o que implica que devem ser mandados dois caracteres logo em seguida representando os dados da mensagem. No exemplo da figura 19, é exemplificado o comportamento descrito, onde o primeiro caracter dos dados representa o PID que deseja

Figura 19 – Exemplo de uma requisição para uma ECU específica do barramento

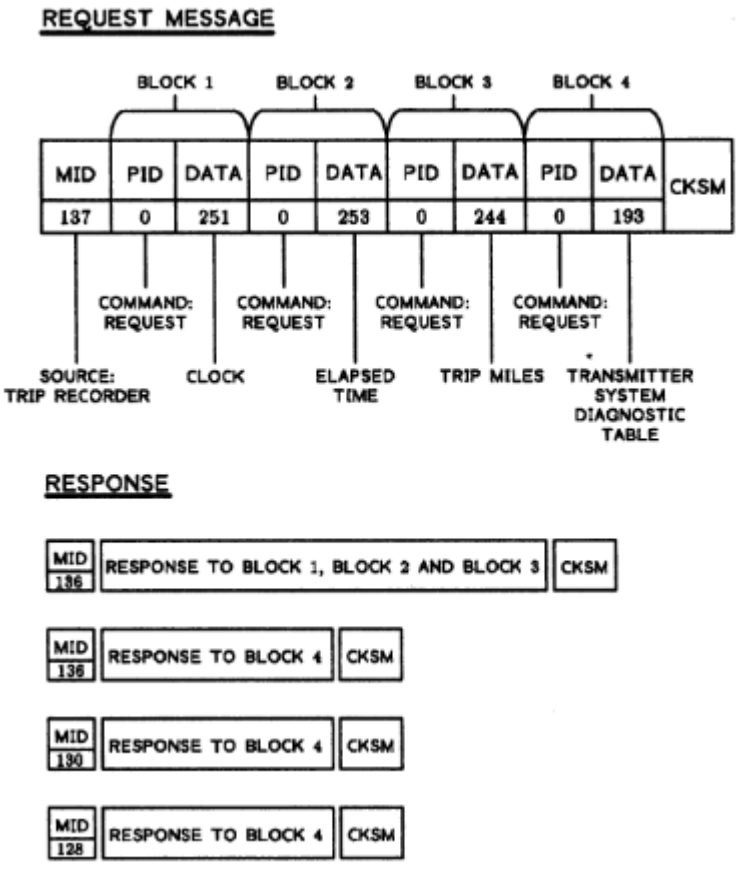
MID	PID	DATA	DATA	CKSM
137	128	194	128	

Fonte: Stepper (1990).

ser lido e o segundo caractere dos dados representa o MID da ECU que deve retornar uma resposta com os dados do PID solicitado.

Existe também um PID específico para requisições que são direcionadas para todas as unidades eletrônicas de controle conectadas ao barramento serial, que é o PID 0. A figura 20 representa uma mensagem enviada para todas as ECU's do barramento e as respectivas respostas recebidas. A mensagem enviada com a solicitação de dados possui 4 blocos, onde todas as ECU's do barramento serial realizam a leitura da mensagem e realizam uma ação se acharem necessário. Uma ECU que receber a requisição pode escolher para quais blocos enviará uma resposta. No exemplo da figura, a ECU de MID 136 responde aos blocos 1,2 e 3, enquanto as ECU's restantes respondem apenas ao bloco 4.

Figura 20 – Exemplo de uma requisição para todas as ECU's do barramento



Fonte: Stepper (1990).

O protocolo SAE J1587 também prevê o conceito de *Subsystem Identification* (SID). SID's são números que representam subdivisões dentro de um mesmo controlador (MID) sem a necessidade de existir um PID relacionado, sendo possível definir até 255 SID's para cada MID. A faixa de SID's de 0 a 150 contém valores pré-determinados para cada MID de forma separada. A faixa de SID's de 151 a 155 são definidos como códigos de diagnóstico do sistema e a faixa de 156 a 255 contém SID's pré-definidos que são de uso comum para todos os MID's. A figura 21 mostra os valores pré-definidos da faixa de 0 a 150 para o MID 163.

Figura 21 – Representação do SID no protocolo SAE J1587

Vehicle Security SIDs (MID = 163)

0	Reserved
1	Transceiver Antenna
2	Security Transponder
3–150	Reserved for future assignment by SAE

Fonte: Truck, Committee *et al.* (2008).

Para o controle e tratamento de erros, o protocolo SAE J1587 possui o *Failure Mode Identifier* (FMI). Os códigos de erros são padronizados nos documentos do protocolo SAE J1587. A figura 22 mostra quais são os valores pré-definidos para cada um dos erros previstos pelo protocolo.

Figura 22 – Representação do FMI no protocolo SAE J1587

0	Data valid but above normal operational range (that is, engine overheating)
1	Data valid but below normal operational range (that is, engine oil pressure too low)
2	Data erratic, intermittent, or incorrect
3	Voltage above normal or shorted high
4	Voltage below normal or shorted low
5	Current below normal or open circuit
6	Current above normal or grounded circuit
7	Mechanical system not responding properly
8	Abnormal frequency, pulse width, or period
9	Abnormal update rate
10	Abnormal rate of change
11	Failure mode not identifiable
12	Bad intelligent device or component
13	Out of Calibration
14	Special Instructions
15	Reserved for future assignment by the SAE Subcommittee

Fonte: Truck, Committee *et al.* (2008).

2.7 Controller Area Network - CAN

Controller Area Network (CAN) é um protocolo de comunicação serial desenvolvido pela Bosch na década de 1980 com o objetivo de simplificar a camada física de comunicação, que

antes era complexa e com grande quantidade de fios, para um barramento simples com apenas dois fios. Originalmente, então, foi desenvolvido para a indústria automotiva, mas logo se tornou popular em outras aplicações, como automação industrial. O foco do trabalho será direcionado à aplicação automotiva.

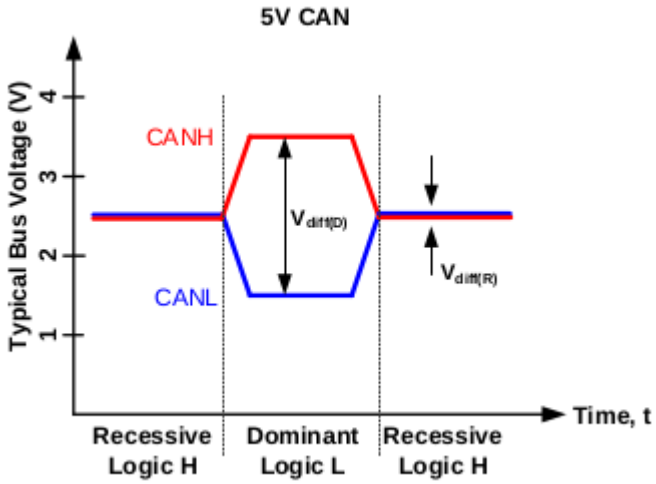
O protocolo CAN foi implementado com base no modelo OSI. Entretanto, o protocolo CAN possui somente a implementação das duas camadas inferiores, que são a camada física e a camada de enlace de dados. As outras camadas devem ser implementadas dependendo do uso que será feito a partir do protocolo CAN.

2.7.1 Camada física

O barramento CAN consiste em um par de fios trançados, que são chamados de *Can High* (CANH) e *Can Low* (CANL). Os níveis lógicos no barramento são gerados a partir da diferença de potencial entre CANH e CANL. Segundo Blackman e Monroe (2013), os transmissores que se conectam ao barramento CAN podem operar com uma alimentação de 5V ou 3,3V. Entretanto, independentemente do nível de tensão de alimentação, a diferença de potencial gerada entre CANH e CANL seguem o mesmo padrão, permitindo que unidades eletrônicas de controle com níveis de alimentação diferentes consigam transmitir dados no mesmo barramento CAN.

Os receptores no barramento CAN determinam o nível lógico a partir da diferença de potencial no barramento entre os fios CANH e CANL. No barramento CAN, o nível lógico de bit 0 é denominado de bit dominante, enquanto que o nível lógico de bit 1 é denominado de bit recessivo. A figura 23 mostra como são gerados os níveis de tensão de um transmissor com nível de tensão de alimentação de 5V, enquanto a figura 24 demonstra o mesmo conceito, porém para transmissores com nível de tensão de alimentação de 3,3V.

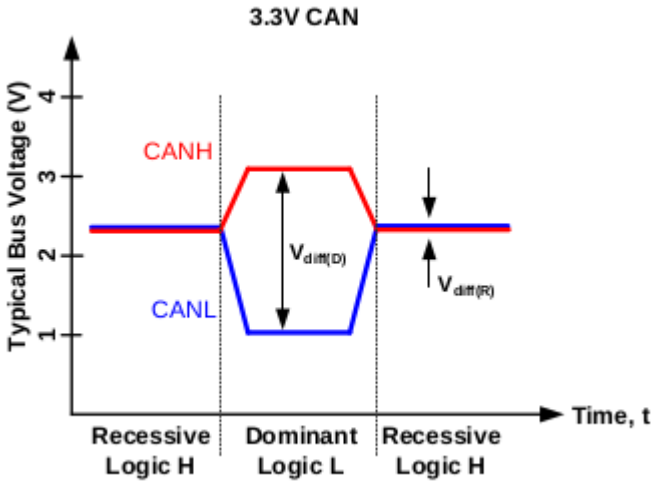
Figura 23 – Níveis lógicos no barramento CAN com transmissor de 5V



Fonte: Blackman e Monroe (2013).

Para transmissores com nível de tensão de alimentação de 5V, quando o transmissor precisa enviar um bit recessivo ao barramento CAN, tanto o CANH quanto o CANL ficam com um nível de tensão de aproximadamente 2,5V (Vcc/2), gerando uma diferença de potencial de 0V no barramento CAN. Quando um bit dominante precisa ser enviado, o transmissor aplica uma tensão de 3,5V no CANH e uma tensão de 1,5V no CANL, gerando uma diferença de potencial de 2V no barramento CAN.

Figura 24 – Níveis lógicos no barramento CAN com transmissor de 3,3V



Fonte: Blackman e Monroe (2013).

Para transmissores com nível de tensão de alimentação de 3,3V, é gerada a mesma diferença de potencial para a transmissão de dados, comparando com transmissores com nível de tensão de alimentação de 5V. Para o envio de um bit recessivo, tanto o CANH quanto o CANL ficam com um nível de tensão de aproximadamente 2,3V, gerando uma diferença de potencial de 0V no barramento CAN. Quando um bit dominante precisa ser enviado, o transmissor aplica uma tensão de 3V no CANH e uma tensão de 1V no CANL, gerando uma diferença de potencial de 2V no barramento CAN.

Outra característica importante no barramento CAN diz respeito à taxa de transmissão. A taxa de transmissão de dados no barramento CAN diminui à medida que o comprimento do barramento CAN aumenta. A figura 25 demonstra o relacionamento entre taxa de transmissão de dados por comprimento do barramento CAN.

2.7.2 Camada de enlace de dados

As principais características do protocolo CAN, de acordo com Voss (2008a), são as seguintes:

- arbitragem de mensagens não-destrutivas;
- acesso ao barramento com prioridade *multi-master*;

Figura 25 – Taxa de transmissão de dados x comprimento

BUS LENGTH (m)	SIGNALING RATE (kbps)
30	1000
100	500
250	250
500	125
1000	62.5

Fonte: HPL (2002).

- mensagem *multicast*;
- requisição de dados de forma remota;
- flexibilidade na configuração;
- consistência de dados;
- detecção e sinalização de erros;
- retransmissão automática de mensagens que perderam a arbitragem;
- retransmissão automática de mensagens destruídas por erros;
- distinção entre erros temporários e falhas permanentes nos nós;
- desativação autônoma de nós com defeito;

Em uma rede CAN, uma unidade eletrônica de controle, quando conectada ao barramento, não precisa saber de nenhuma informação sobre as configurações do sistema. De acordo com BOSCH (1991), isso implica que um barramento CAN possui as seguintes características:

- **flexibilidade do sistema**: nós podem ser adicionados ao barramento CAN sem a necessidade de nenhuma alteração de *software* ou *hardware* em qualquer um dos outros nós e na camada de aplicação;
- **roteamento de mensagens**: o conteúdo da mensagem é nomeado através de um identificador. O identificador não indica qual é o destino da mensagem, mas contém o significado dessa mensagem de forma que todos os nós possam ler esse identificador e decidir se deve tomar alguma ação em cima dessa mensagem;
- **multicast**: como consequência do conceito anterior, qualquer número de nós pode receber e processar uma mesma mensagem simultaneamente;

- **consistência de dados**: dentro de um barramento CAN, é garantido que uma mensagem seja aceita simultaneamente por todos os nós ou por nenhum nó. Desse modo, a consistência de dados é feita por multicast e por tratamento de erros;
- **taxa de bits**: a velocidade de transmissão em um barramento CAN pode variar em diferentes sistemas; porém, dentro de um único sistema, a taxa de transmissão é uniforme e fixa;
- **prioridade entre mensagens**: o identificador define qual é a prioridade da mensagem através do conceito de arbitragem;
- **multimaster**: quando o barramento está livre, qualquer unidade eletrônica de controle pode transmitir uma mensagem. A unidade eletrônica de controle com a mensagem de maior prioridade ganhará acesso ao barramento;
- **arbitragem**: extendendo um pouco do conceito de multimaster, caso duas unidades eletrônicas de controle comecem a transmitir suas mensagens ao mesmo tempo, temos um cenário de conflito no barramento, que é resolvido pelo conceito de arbitragem. Durante o processo de arbitragem, cada transmissor compara o nível do bit transmitido com o nível que é considerado dominante pelo barramento. Os bits validados estão inclusos no identificador da mensagem, e o transmissor que enviar uma mensagem com um bit dominante ganhará acesso ao barramento.
- **sinalização de erro**: mensagem corrompidas são marcadas como mensagem com erro por qual nó que detectou o erro. Essas mensagens são abortadas e retransmitidas automaticamente;
- **classificação de falhas**: os nós de um barramento CAN são capazes de distinguir perturbações rápidas de erros permanentes. Desse modo, nós com defeito são desligados do barramento.
- **conexões**: o barramento CAN, teoricamente, não possui limite quanto ao número de nós que podem ser conectados a ele. Na prática, o número total de unidades eletrônicas de controle será limitada pelo tempo de *delay* e cargas elétricas no barramento.
- **single channel**: o barramento CAN consiste de um único canal com a transmissão de bits.
- **valores no barramento**: o barramento pode ter apenas um dos dois valores lógicos complementares: dominante ou recessivo. Durante a transmissão simultânea de um bit recessivo e de um bit dominante, o valor resultante no barramento será o bit dominante.

- **acknowledgment**: todos os receptores checam a consistência da mensagem e marcam a mensagem como consistente ou inconsistente;
- **modo de economia de energia**: para reduzir o consumo de energia do sistema, um barramento CAN pode ser configurado para entrar em modo de espera, sem nenhuma atividade interna do sistema de configuração e desconectado de todos os nós. O modo de espera é encerrado caso haja qualquer atividade no barramento ou atividade interna do sistema de configuração.

O barramento CAN garante máxima segurança na transferência de dados. Para isso, medidas de detecção de erro, sinalização e tratamento devem ser implementadas em todos os nós do barramento.

2.7.3 *Frames*

Seguindo o padrão CAN, todas as mensagens são denominadas de *frames*. Todo *frame* enviado ao barramento deve estar de acordo com o padrão definido para cada um dos seus tipos possíveis, que são:

- *data frame*;
- *remote frame*;
- *error frame*;
- *overload frame*

Antes de entrar no detalhe de cada um dos tipos de *frame*, é necessário realizar uma contextualização sobre a arbitragem de mensagens. Como mencionado anteriormente, um barramento CAN é orientado a eventos. Desse modo, qualquer um dos nós pode enviar uma mensagem a qualquer momento. Essa característica gera um conflito no barramento, já que dois nós podem tentar enviar uma mensagem exatamente ao mesmo tempo.

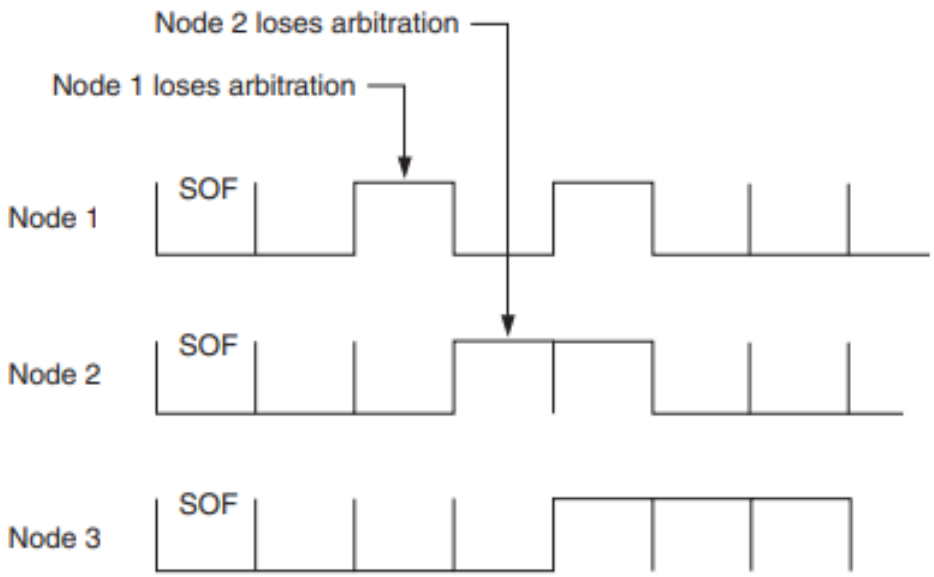
A estratégia utilizada pelo barramento CAN para lidar com esses conflitos é a arbitragem de mensagens. A arbitragem de mensagens é realizado a nível de bit, onde é definido um bit chamado de dominante, que é o bit 0, e um bit chamado de recessivo, que é o bit 1. Desse modo, um bit 0 tem prioridade em relação a um bit 1.

Dentro da mensagem, existe um campo específico que define qual é a prioridade da mensagem. O barramento CAN realiza a leitura de um bit por vez do campo de prioridade; a mensagem cujo bit atual é dominante tem prioridade em relação as outras mensagens, sendo transmitida para os outros nós, enquanto as outras mensagens serão retransmitidas assim que o barramento ficar livre novamente. Caso duas mensagens possuam o primeiro bit de

identificação sendo dominante, o barramento inicia a leitura do segundo bit, e assim por diante, até que seja definida qual é a mensagem de maior prioridade.

A Figura 26 exemplifica como é o processo de arbitragem executado no barramento CAN. Existem três mensagens a serem transmitidas para o barramento. O SOF representa o ínicio de um *frame* e é sempre dominante (bit 0); desse modo, a mensagem que será transmitida pelo primeiro nó possui a identificação **0101000**, a mensagem que será transmitida pelo segundo nó possui a identificação **0011000** e a mensagem que será transmitida pelo terceiro nó possui a identificação **000111**.

Figura 26 – Demonstração do processo de arbitragem de mensagens



Fonte: Cook e Freudenberg (2007).

O primeiro bit de identificação da mensagem transmitida pelo primeiro nó é **0**, o primeiro bit de identificação da mensagem transmitida pelo segundo nó é **0** e o primeiro bit de identificação da mensagem transmitida pelo terceiro nó também é **0**. Assim, o primeiro bit de todos os três nós é dominante e ainda não é possível definir qual é a mensagem mais prioritária. O barramento CAN começa, então, a realizar a leitura do segundo bit de identificação das mensagens. O segundo bit das mensagens que os nós estão tentando transmitir são, respectivamente, **1**, **0** e **0**. Portanto, o segundo e terceiro nós possuem bits dominantes, enquanto o primeiro nó possui um bit recessivo. O primeiro nó perde a arbitragem nesse momento e o barramento CAN agora precisa definir se a mensagem do segundo ou do terceiro nó possui prioridade. Ao analisar o terceiro bit de identificação das mensagens, tem-se que o segundo nó possui um bit recessivo (bit 1), enquanto que o terceiro nó possui um bit dominante (bit 0). Nesse momento, já é possível definir que a mensagem do terceiro nó é a mais prioritária entre as três mensagens enviadas pelos três nós.

Quando a transmissão da mensagem do terceiro nó for finalizada, o barramento ficará livre para receber uma nova mensagem. O primeiro e o segundo nós farão a retransmissão das mensagens e novamente será decidida qual é a mensagem mais prioritária através do conceito de arbitragem de mensagens. A mensagem do segundo nó é mais prioritária que a mensagem do primeiro nó e será transmitida. Quando a transmissão da mensagem do segundo nó for finalizada, o barramento novamente ficará livre para uma nova transmissão e a mensagem do primeiro nó será finalmente transmitida. Lembrando que se uma nova mensagem mais prioritária chegar ao barramento antes da primeira mensagem ser transmitida, essa nova mensagem será transmitida antes da mensagem do primeiro nó. A mensagem do primeiro nó deverá aguardar novamente o barramento CAN ficar livre para que o primeiro nó tente novamente fazer a transmissão da sua mensagem.

Com o conhecimento sobre bit recessivo, bit dominante e arbitragem de mensagens, já é possível entrar no detalhe do que compõe cada um dos diferentes tipos de frame suportados por um barramento CAN.

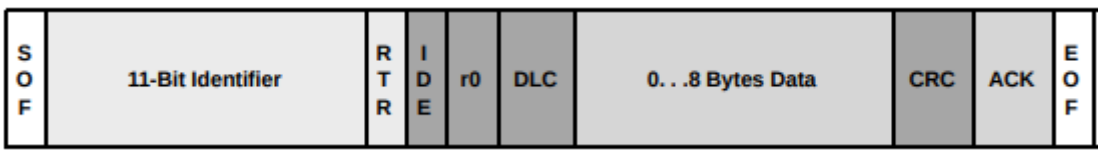
2.7.3.1 *Data frame*

Data frame é a mensagem responsável pela transmissão de dados. É usado quando uma unidade eletrônica de controle deseja enviar para todos os outros nós pelo barramento uma mensagem contendo dados, onde cada um dos nós que receber a mensagem decidirá se tomará uma ação de acordo com os dados recebidos ou apenas fará a validação de que a mensagem está consistente.

Existem dois modelos de *data frame* previstos pelo protocolo CAN: o *data frame* padrão, que é conhecido como CAN 2.0A e cujo identificador possui 11 bits, e o *data frame* extendido, que é conhecido como CAN 2.0B e cujo identificador possui 29 bits.

A Figura 27 representa quais são os campos que compõem um *data frame* do modelo Standard CAN.

Figura 27 – Standard CAN (CAN 2.0A)



Fonte: HPL (2002).

- *Start of Frame* (SOF): é o primeiro campo que compõe o frame e representa o ínicio do frame. O campo possui apenas 1 bit e deve sempre ter o valor do bit dominante;
- Identificador de 11 bits (11-Bit *Identifier*): é um conjunto de bits que representa a identificação de um nó do barramento CAN e define a prioridade da mensagem

que está sendo transmitida. Como exemplificado anteriormente, é com base no campo de identificador da mensagem que é aplicado o conceito de arbitragem de mensagem para definir qual é a mensagem com maior prioridade, no caso em que existe mais de uma mensagem a ser transmitida pelo barramento ao mesmo tempo. No formato *Standard* CAN, o campo de identificador de mensagem possui 11 bits;

- *Remote Transmission Request* (RTR): o RTR é um campo que possui apenas 1 bit com o propósito de identificar se a mensagem enviada é um *data frame* ou um *remote frame*. O valor de bit dominante indica que a mensagem é um *data frame*, enquanto que o valor de bit recessivo indica que a mensagem é um *remote frame*;

- *Identifier Extension* (IDE): o *Identifier Extension* é um campo de 1 bit que existe com o propósito de indicar se o data frame foi criado a partir do padrão *Standard* CAN ou *Extended* CAN. O valor de bit dominante indica que a mensagem enviada está no padrão *Standard* CAN, enquanto que o valor de bit recessivo indica que a mensagem enviada está no padrão *Extended* CAN;

- r0: o campo r0 é um campo de 1 bit, reservado para uso em aplicações futuras cujo valor atualmente é mantido com o valor de um bit dominante;

- *Data Length Code* (DLC): O *Data Length Code* é um campo de 4 bits que determina qual é o tamanho dos dados que estão sendo enviados no *data frame*. O valor do Data Length Code representa a quantidade de bytes que estão sendo transmitidos pelo campo DataField. A Figura 28 exemplifica como são representados os bits para cada quantidade de bytes possíveis de serem transmitidos pelo DataField, levando em consideração o conceito de bit dominante (abreviado por *d* na figura) e bit recessivo (abreviado por r na figura).

- *Data Field*: contém a informação que está sendo transmitida pela mensagem. Um *data frame* pode transmitir uma mensagem contendo de 0 a 8 bytes (0 a 64 bits).

- *Cyclic Redundancy Check* (CRC): o *Cyclic Redundancy Check* é um campo composto por 16 bits, sendo 15 bits utilizados para detecção de erros e um bit delimitador no fim do campo que é sempre recessivo. O transmissor calcula o CRC com base no valor do *Start of Frame*, Identificador, RTR, IDE, DLC e *Data Field*; a mensagem é enviada para o barramento com o CRC calculado e todos os nós que receberem a mensagem refazem o cálculo do CRC para garantir que a informação chegou corretamente. Como o CRC é um campo de 15 bits, de acordo com BOSCH (1991), é utilizado o seguinte polinômio para o seu cálculo: $x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1$. Para realizar o cálculo do CRC, são incluídos ao fim da mensagem 15 bits recessivos; após a inclusão desses bits, é

Figura 28 – Representação dos valores do Data Length Code

Number of Data Bytes	Data Length Code			
	DLC3	DLC2	DLC1	DLC0
0	d	d	d	d
1	d	d	d	r
2	d	d	r	d
3	d	d	r	r
4	d	r	d	d
5	d	r	d	r
6	d	r	r	d
7	d	r	r	r
8	r	d	d	d

Fonte: BOSCH (1991).

feita a operação XOR entre a mensagem com os 15 bits recessivos e a representação binária do polinômio aplicado, que é **1100010110011001**. O resultado dessa operação é um CRC de 15 bits que é incluído ao fim da mensagem. A mensagem com o CRC é transmitida pelo barramento com a inclusão do bit delimitador e todos os nós que recebem a mensagem realizam a operação XOR da mensagem recebida com o CRC com o mesmo polinômio. Se o resultado for zero, significa que a mensagem chegou corretamente aos receptores, enquanto que um resultado diferente de zero representa que a mensagem não chegou corretamente e o receptor envia uma mensagem de erro ao barramento CAN;

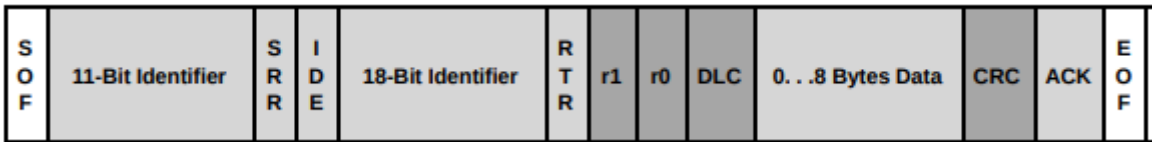
- *Acknowledge* (ACK): ACK é um campo composto de 2 bits, sendo o primeiro bit utilizado para representar se mensagem foi lida corretamente pelo receptor e o segundo bit é o delimitador do ACK que é sempre recessivo. Toda mensagem é enviada ao barramento CAN com o primeiro bit do campo ACK com o valor de bit recessivo; quando a mensagem é recebida com sucesso, o nó receptor altera o valor do primeiro bit do campo ACK para um valor de bit dominante para indicar que a mensagem foi recebida com sucesso. Caso a mensagem não seja recebida com sucesso, o nó receptor não faz nenhuma alteração no primeiro bit do campo ACK. Após todos os receptores terem feita a leitura da mensagem, o nó transmissor valida o valor do primeiro campo do bit ACK para confirmar que pelo menos um nó conseguiu receber a mensagem com sucesso. Se nenhum nó

receber a mensagem com sucesso e marcar um bit dominante no primeiro bit do campo ACK, o transmissor entende que a mensagem não foi recebida com sucesso. Desse modo, é feita posteriormente uma tentativa de retransmissão pelo nó transmissor quando o barramento estiver novamente livre, lembrando que essa mensagem precisa passar novamente pelo processo de arbitragem para que seja enviada novamente ao barramento.

- *End of Frame* (EOF): representa o fim da mensagem. O *End of Frame* é um campo composto de 7 bits, onde devem ser transmitidos 7 bits recessivos.

O padrão CAN 2.0B, conhecido como *Extended* CAN, é representado pela Figura 29.

Figura 29 – Extended CAN (CAN 2.0B)



Fonte: HPL (2002).

A maioria dos campos do padrão *Extended* CAN (2.0B) possuem a mesma finalidade definida para o padrão CAN (2.0A). A definição de cada campo do padrão *Extended* CAN, em comparação ao formato *Standard* CAN, é mostrada a seguir.

- *Start of Frame* (SOF): possui a mesma definição apresentada no Standard CAN;
- Identificador de 11 bits (11-Bit *Identifier*): é composto de 11 bits e representa a primeira parte do identificador da mensagem;
- *Substitute Remote Request*(SRR): o SRR é um campo de 1 bit apenas que é enviado sempre com o valor recessivo. A posição do SRR no *data frame* do *Extended* CAN é a mesma posição do RTT no *data frame* do Standard CAN; considerando que a mensagem é um *data frame*, o valor do bit SRR no *Extended* CAN sempre será recessivo, enquanto que o valor do bit RTT no *Standard* CAN será sempre dominante. Se um *data frame* no padrão *Standard* CAN e um *data frame* no padrão *Extended* CAN tiverem os 11 bits de identificação iguais, o SRR sempre recessivo garante que o data frame do padrão Standard CAN sempre terá prioridade em relação ao *data frame* do padrão *Extended* CAN.
- *Identifier Extension* (IDE): possui a mesma definição apresentada no *Standard* CAN. Para o padrão *Extended* CAN, o IDE possui um bit de valor recessivo, indicando que os próximos 18 bits são o complemento dos primeiros 11 bits do identificador.

- Identificador de 18 bits (18-Bit *Identifier*): é composto de 18 bits e representa a segunda parte do identificador da mensagem;
- *Remote Transmission Request* (RTR): possui a mesma definição apresentada no *Standard* CAN. É representado pelo bit logo após ao identificador de 18 bits no padrão Extended CAN, enquanto no *Standard* CAN é representado pelo bit logo após ao identificador de 11 bits.
- r1: no padrão *Extended* CAN, foi adicionado um segundo bit reservado para operações futuras, cujo valor atualmente é mantido com o valor de um bit dominante;
- r0: possui a mesma definição apresentada no *Standard* CAN;
- *Data Length Code* (DLC): possui a mesma definição apresentada no *Standard* CAN;
- *Data Field*: possui a mesma definição apresentada no *Standard* CAN;
- *Cyclic Redundancy Check* (CRC): possui a mesma definição apresentada no *Standard* CAN. A diferença é que no *Extended* CAN a quantidade de bits que são considerados no cálculo do CRC é maior em relação ao *Standard* CAN, já que o *Extended* CAN possui o SRR, o r1 e o 18-Bit *Identifier* a mais em sua composição.
- *Acknowledge* (ACK): possui a mesma definição apresentada no *Standard* CAN;
- *End of Frame* (EOF): possui a mesma definição apresentada no *Standard* CAN;

Um exemplo prático de um *data frame* que é enviado em um barramento de um sistema automobilístico são as mensagens enviadas pelos sensores de segurança, como por exemplo, o sensor de *airbag*. O sensor de *airbag* transmite através do barramento CAN o seu estado atual, indicando se tudo está operando de forma correta ou se há um problema a ser verificado. As unidades de controle eletrônicas que receberem o *data frame* realizarão a validação da mensagem e então decidirão se devem tomar alguma ação de acordo com o conteúdo da mensagem recebida, por exemplo, indicar no painel do automóvel que há um problema com o *airbag*.

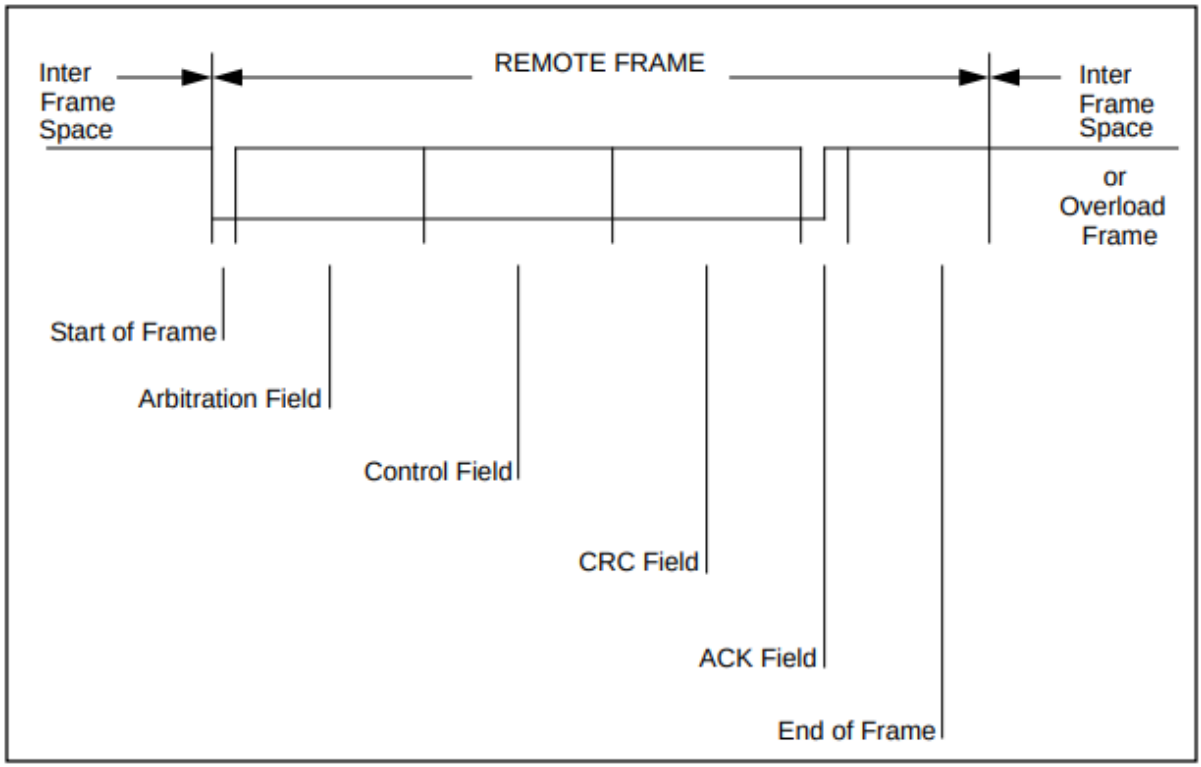
2.7.3.2 *Remote frame*

O *remote frame* é uma mensagem que tem como propósito solicitar dados de uma unidade eletrônica de controle específica. Essa é uma funcionalidade muito útil no barramento CAN, onde um determinado nó não precisa ficar apenas esperando as mensagens serem enviadas pelo barramento e pode enviar um *remote frame* solicitando informações de um nó específico. Essa funcionalidade é chamada *Remote Transmission Request* (RTR). Por

exemplo, o sistema de segurança de um carro recebe atualizações de status frequentes de sensores críticos como o sensor de *airbag*, porém, pode não receber com tanta frequência atualizações de status de outros sensores, como o sensor de pressão de óleo e o sensor de baixa bateria. Periodicamente, o sistema de segurança pode solicitar informações desses sensores enviando um *remote frame* com o objetivo de fazer uma validação completa de todos os sensores conectados no barramento CAN. Essa funcionalidade pode ser utilizada para minimizar o tráfego de dados no barramento CAN sem perder a integridade de dados no barramento.

O *remote frame* possui a mesma composição de bits apresentada pelo *data frame*, com a diferença de que não são enviados os *bytes* do *DataField*. A Figura 30 mostra como é a composição de um *remote frame*.

Figura 30 – *Remote frame*



Fonte: BOSCH (1991).

O *remote frame*, assim como o *data frame*, é representado tanto pelo padrão CAN 2.0A (Standard CAN), quanto pelo padrão CAN 2.0B (*Extended CAN*), podendo ter um identificador de 11 bits ou um identificador de 29 bits. O barramento CAN reconhece um *remote frame* através do campo RTR; quando o bit é recessivo, indica que a mensagem enviada é um *remote frame*, enquanto um bit dominante indica que a mensagem é um *data frame*. Para solicitar uma informação de um nó específico, o *remote frame* envia no campo de identificador o conjunto de bits que identifica aquele nó em específico. Todos os nós do barramento CAN receberão a mensagem, realizarão a validação da sua integridade e apenas o nó com aquele

identificador realizará uma ação em cima do *remote frame*, que será enviar ao barramento CAN um *data frame* com a informação solicitada.

No exemplo do sensor de pressão de óleo citado anteriormente, quando o sistema de segurança precisar requisitar alguma informação sobre o sensor de pressão de óleo, ele enviará ao barramento CAN um *remote frame* contendo o identificador da unidade eletrônica de controle responsável pelo sensor de pressão do óleo. Quando a unidade eletrônica de controle do sensor de pressão do óleo identificar que há uma requisição da sua atualização de status, este enviará ao barramento CAN um *data frame* com a informação requisitada, que chegará ao sistema de segurança.

2.7.3.3 *Error frame*

Como o protocolo CAN foi inicialmente desenvolvido para uso na indústria automobilística, um ponto crítico para que o seu uso fosse aceito no mercado diz respeito à eficiência no tratamento de erros. O protocolo CAN possui a habilidade de determinar quais são as condições de erros a serem validadas e operar de determinado modo baseado na severidade dos problemas encontrados. Os nós de um barramento CAN podem funcionar como um nó normal, sendo capaz de enviar e receber mensagens normalmente, até ser desligado completamente baseado na severidade dos erros encontrados. Nenhum nó com problema monopolizará a banda da rede, já que esses nós problemáticos serão desligados antes de derrubar o sistema todo. Esse comportamento é conhecido como Confinamento de Erros (*Fault Confinement*).

Segundo Pazul (1999), existem 5 tipos de erros definidos no protocolo CAN:

- *CRC Error*: o CRC Error é gerado quando a validação realizada no campo CRC do *Data Frame* e do *Remote Frame* apresenta um resultado incorreto. Como apresentado anteriormente, o transmissor calcula o CRC com base em um polinômio de grau 15 e transmite o valor calculado. Todos os nós que recebem a mensagem realizam a operação XOR da mensagem recebida utilizando o mesmo polinômio e quando o resultado é diferente de zero, significa que há uma divergência no valor do CRC. Quando essa divergência é detectada, o nó que recebeu a mensagem e detectou a inconsistência envia ao barramento CAN uma mensagem de erro; com isso, o transmissor poderá reenviar uma nova tentativa de transmissão da mensagem.

- *Acknowledge Error*: todos os nós que recebem um *data frame* ou um *remote frame* precisam informar ao barramento CAN que a mensagem foi recebida com sucesso. Como apresentado anteriormente, o campo ACK do *data frame* ou do *remote frame* é enviado pelo transmissor ao barramento CAN com o valor de bit recessivo e deve ser atualizado com o valor de um bit dominante pelos nós para determinar

que a mensagem foi recebida com sucesso. Após a leitura da mensagem pelos receptores, o transmissor faz a leitura do valor do campo ACK e valida se existe um bit dominante. Caso o valor do campo ACK ainda seja o valor de um bit recessivo, significa que nenhum nó conseguiu receber a mensagem com sucesso e um *Acknowledge Error* é gerado. Quando isso acontece, o transmissor envia uma mensagem de erro ao barramento CAN e após poderá reenviar uma nova tentativa de transmissão da mensagem. Lembrando que se apenas um nó conseguir receber a mensagem com sucesso e marcar o campo ACK como dominante, nenhum erro será gerado.

- *Form Error*: como definido anteriormente no *data frame* e *remote frame*, o campo EOF (*End of Frame*), o delimitador do ACK e o delimitador do CRC devem sempre possuir os seus bits contendo o valor de um bit recessivo. Caso seja enviado um bit dominante em um desses campos, o nó receptor que detectou a inconsistência gerará um *Form Error* e mandará uma mensagem de erro ao barramento CAN.

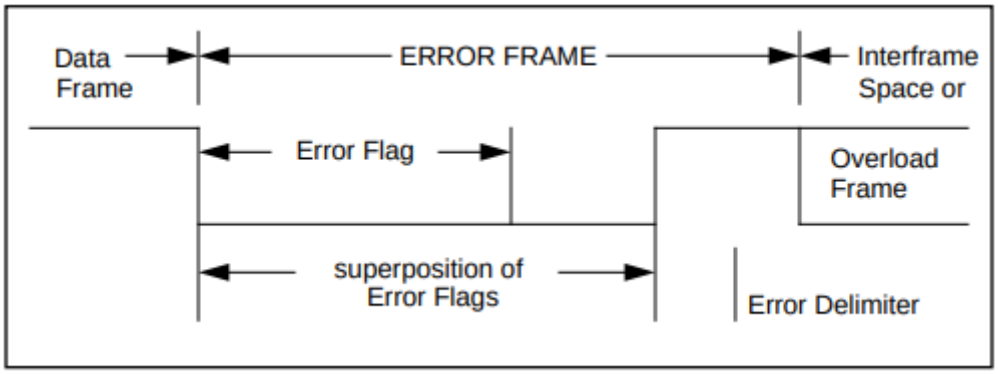
- *Bit Error*: o protocolo CAN prevê que um transmissor faça o monitoramento das mensagens que são enviadas ao barramento, para garantir a integridade da informação. Desse modo, quando o transmissor envia uma mensagem, ele realiza a leitura nos níveis de tensão no barramento CAN para validar que os bits foram enviados corretamente. Por exemplo, se o transmissor enviar um bit recessivo e detectar, ao realizar a leitura dos níveis de tensão do barramento CAN, que o mesmo bit está com valor de um bit dominante no barramento CAN, um *Bit Error* será gerado pelo transmissor e uma mensagem de erro será enviada. As únicas exceções que não geram um *Bit Error* estão relacionadas com o campo de identificador, onde o processo de arbitragem está ocorrendo e a mensagem pode não possuir a maior prioridade para ocupar o barramento CAN, e o campo de ACK, onde um nó que recebe a mensagem pode alterar o valor do campo para um bit dominante ao confirmar que a mensagem foi recebida com sucesso.

- *Stuff Error*: o protocolo CAN utiliza o método de transmissão NRZ (*Non-Return-To-Zero*). Isso significa que um mesmo valor de bit, seja dominante ou recessivo, não pode ser enviado consecutivamente ao barramento CAN mais do que 5 vezes. Quando um mesmo valor de bit é enviado consecutivamente mais do que 5 vezes, um bit de valor oposto será incluído a cada 5 bits. Por exemplo, se uma mensagem contem a informação **111111011000000**, ela receberá um bit de valor invertido sempre que for detectada uma sequência de mais de 5 bits consecutivos com o mesmo valor, resultando em **11111**0**10110000**0**0**. Os nós que receberem a mensagem usarão o *stuffing* bit para sincronização, mas ignorarão o bit ao fazer a leitura do conteúdo da mensagem. Caso seis bits com o mesmo valor

sejam enviados consecutivamente, o método NRZ foi violado e um *Stuff Error* será gerado, sendo mandada uma mensagem de erro ao barramento CAN.

Quando qualquer um dos erros apresentados acima é gerado, a mensagem de erro enviada ao barramento CAN é conhecida como *Error Frame*. Um *Error Frame* é composto por dois campos conhecidos como *Error Flag* e *Error Delimiter*. O *Error Flag* é um campo composto de 6 bits, onde são enviados 6 bits dominantes ou 6 bits recessivos, dependendo do estado atual do nó no barramento CAN; já o *Error Delimiter* é um campo de 8 bits onde todos os bits são enviados recessivos. Como o *Error Frame* viola a regra de transmissão NRZ com *Bit Stuffing*, todos os outros nós responderão enviando também um *Error Frame*. A Figura 31 representa a composição de um *Error Frame*.

Figura 31 – Error frame



Fonte: BOSCH (1991).

O protocolo CAN consegue definir se um erro é temporário, devido a algum ruído na rede, por exemplo, ou se o erro representa um defeito no nó. De acordo com Gaujal e Navet (2002), é possível identificar a severidade desse erro através do Confinamento de Erros. O Confinamento de Erros prevê que o protocolo CAN possua uma contagem da quantidade de *Error Frames* enviados por um nó. Com esse propósito, existem duas variáveis de contagem de erros previstas:

- *Transmit Error Counter (TEC)*: variável que conta a quantidade de erros gerados por um nó enquanto estiver com o propósito de transmissão. A variável é incrementada quando um *Acknowledge Error* ou um *Bit Error* é gerado pelo transmissor;
- *Receive Error Counter (REC)*: variável que conta a quantidade de erros gerador por um nó quando estuver com o propósito de recepção. A variável é incrementada quando um *CRC Error*, *Form Error* ou *Stuff Error* é gerado pelo receptor

Toda vez que um nó transmite uma mensagem com sucesso, a contagem do TEC do nó transmissor é diminuída, não podendo ter o seu valor menor do que zero. Toda vez que um nó recebe uma mensagem com sucesso, a contagem do REC do nó transmissor é diminuída,

não podendo ter o seu valor menor do que zero. Ainda de acordo com Gaujal e Navet (2002), existem três estados que o protocolo CAN prevê para representar a condição atual de um nó no barramento:

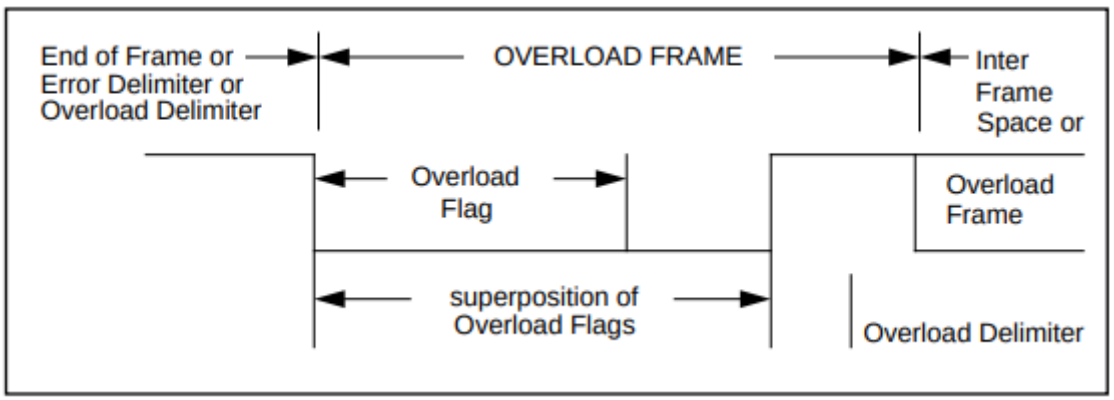
- *Error Active (REC <= 127 e TEC <= 127)*: representa o modo de operação normal de um nó no barramento, onde o mesmo pode enviar um *data frame* ou *remote frame* sem restrições. Ao enviar um *Error Frame*, um nó nesse estado preenche o campo *Error Flag* com 6 bits dominantes;
- *Error Passive ((REC > 127 ou TEC > 127) e TEC <= 255)*: um nó nesse estado pode transmitir um *data frame* ou *remote frame* normalmente, porém, sempre deve esperar um intervalo de 8 bits em relação ao último *frame* para tentar transmitir um novo *frame*. Ao enviar um *Error Frame*, um nó nesse estado preenche o campo *Error Flag* com 6 bits recessivos;
- *Bus-Off (TEC > 255)*: quando a contagem de erros de transmissão ultrapassar 255, o nó é automaticamente removido do barramento CAN, ficando impossibilitado de transmitir e de receber mensagens.

2.7.3.4 *Overload frame*

Quando um nó recebe mensagens de forma mais rápida do que pode processar, ele envia ao barramento CAN uma mensagem conhecida como *Overload Frame*, indicando que é necessário um tempo de espera até o envio de um próximo *Data Frame* ou *Remote Frame*.

O *Overload Frame* consiste de dois campos, um *Overload Flag* e um *Overload Delimiter*. O *Overload Flag* é um campo com 6 bits, onde todos os bits são enviados com o valor de um bit dominante; já o *Overload Delimiter* é um campo de 8 bits, onde todos os bits são enviados com o valor de um bit recessivo. A Figura 32 representa a composição de um *Overload Frame*.

Figura 32 – *Overload frame*



Fonte: BOSCH (1991).

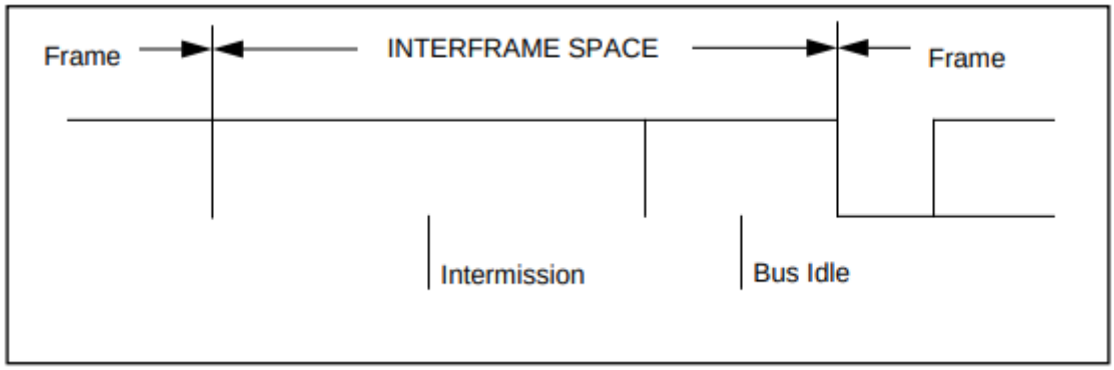
2.7.4 *Interframe spacing*

Data Frames e *Remote Frames* são separados de *frames* anteriores por um campo chamado ***INTERFRAME SPACE***. Essa separação é feita independentemente do tipo de *frame* que preceder o *Data Frame* ou *Remote Frame* a ser enviado, seja o *frame* anterior um *Data Frame*, *Remote Frame*, *Error Frame* ou *Overload Frame*. Em contrapartida, um *Error Frame* ou um *Overload Frame* não é precedido por um *Interframe Space* e múltiplos *Overload Frame* não são separados por um *Interframe Space*.

Um *Interframe Space* possui duas representações de acordo com as seguintes condições:

- nós que não estão no estado *Error Passive* ou nós que foram receptores da mensagem anterior possuem um campo chamado ***Intermission*** e um campo chamado ***Bus Idle*** (ver Figura 33);

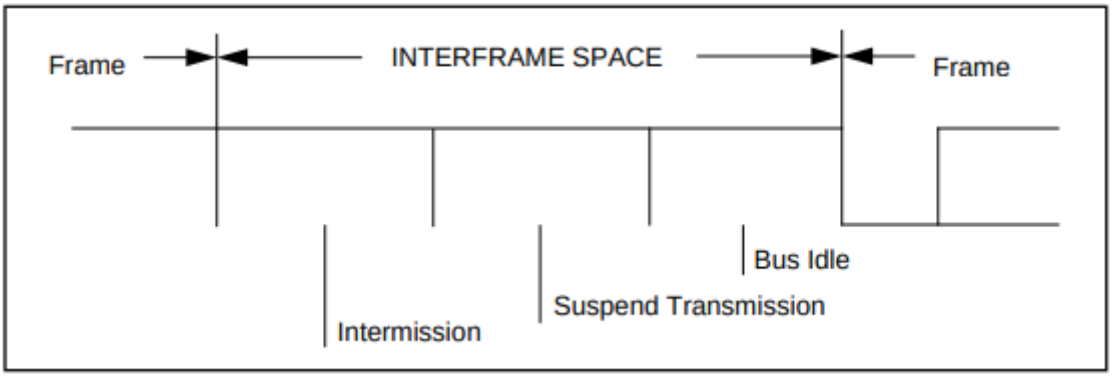
Figura 33 – *Interframe Space*



Fonte: BOSCH (1991).

- nós no estado *Error Passive* que foram transmissores da mensagem anterior possui, além dos campos ***Intermission*** e ***Bus Idle***, um campo chamado ***Suspend Transmission*** (ver Figura 34)

Figura 34 – *Interframe Space* em transmissores no estado *Error Passive*



Fonte: BOSCH (1991).

O campo *Intermission* é composto por três bits que devem ser sempre mandados com o valor de um bit recessivo. Durante a transmissão do campo de *Intermission*, nenhum nó tem permissão de transmitir um *Data Frame* ou um *Remote Frame*. A única operação permitida durante a transmissão do *Intermission* é o envio de um *Overload Frame*. Um *Overload Frame* possui a mesma composição de um *Error Frame* enviado por um nó no estado *Error Active*; o que diferencia um *Overload Frame* de um *Error Frame* é o momento de envio de cada um. Enquanto o *Error Frame* é enviado durante a transmissão de uma mensagem, o *Overload Frame* só pode ser enviado durante a transmissão do campo *Intermission*.

O *Bus Idle* representa que o barramento está disponível para receber mensagens. O campo de *Bus Idle* se mantém com um valor de bit recessivo até que algum nó comece a transmitir uma mensagem. Se alguma mensagem já teve uma tentativa de envio anterior, porém não foi enviada por ter menor prioridade no processo de arbitragem, será feita uma nova tentativa de envio logo após o último bit do campo *Intermission*. Quando um bit dominante é detectado durante o *Bus Idle*, esse bit é interpretado como um *Start of Frame*.

O campo *Suspend Transmission* é enviado por um nó no estado *Error Passive* que havia acabado de transmitir uma mensagem. Logo após o fim da transmissão do *Intermission*, o nó envia oito bits com valor recessivo. Esse nó só poderá enviar uma nova mensagem ou reconhecer o barramento como disponível após o fim da transmissão do *Suspend Transmission*. Isso não impede que qualquer outro nó identifique o barramento como disponível e comece a transmitir uma nova mensagem.

2.8 SAE J1939

Apesar de ser extremamente eficiente em aplicações menores, o protocolo CAN sozinho não é suficiente para comunicação em sistemas maiores, já que a comunicação entre dispositivos é limitada a apenas 8 *bytes* por mensagem. Com isso, vários protocolos foram desenvolvidos a partir do protocolo CAN para permitir o envio de mensagens com um maior tamanho do que o especificado no protocolo CAN.

No que diz respeito aos sistemas automobilísticos, a **Sociedade de Engenheiros de Automóveis** (do inglês, *Society of Automotive Engineers*, ou SAE) padronizou regras de comunicação a serem seguidas em todos os sistemas automotivos, criando o protocolo SAE J1939. O protoloco SAE J1939 é implementada com base no modelo OSI e no protocolo CAN. Segundo Voss (2008b), as principais características do protocolo SAE J1939 são:

- deve utilizar um par de fios trançados;
- o comprimento do barramento não pode ser maior do que 40 metros;
- taxa de transmissão padrão de 250kbps;
- não devem ser conectadas mais de 30 ECU's na rede;

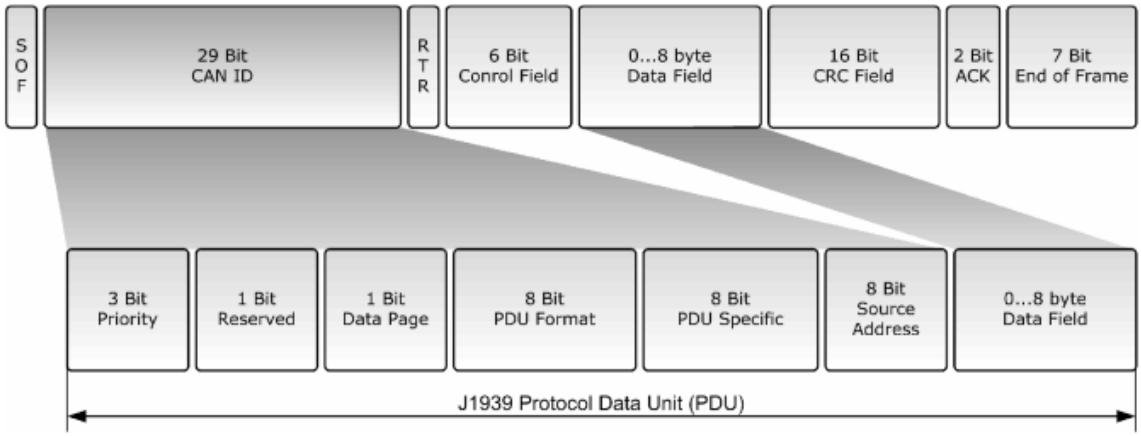
- não devem ser conectados mais de 253 controladores no barramento, onde uma ECU pode possuir vários controladores;
- um transmissor pode se comunicar diretamente com uma ECU em específico ou pode mandar uma mensagem para todas as ECU's da rede;
- suporte para mensagem com tamanho máximo de até 1785 *bytes*;
- padronização através da definição de *Parameter Groups* (PG);
- gerenciamento de rede

Existem vários documentos publicados pela SAE com o objetivo de especificar e padronizar o uso do protocolo SAE J1939. Em Voss (2008b), são citados os seguintes documentos:

- J1939: *Recommended Practice for a Serial Control and Communications Vehicle Network 1*;
- J1939/01: *Recommended Practice for Control And Communications Network for On-Highway Equipment*;
- J1939/02: *Agricultural and Forestry Off-Road Machinery Control and Communication Network*;
- J1939/11: *Physical Layer - 250k bits/s, Twisted Shielded Pair*;
- J1939/13: *Off-Board Diagnostics Connector*;
- J1939/15: *Reduced Physical Layer, 250k bits/sec, Un-Shielded Twisted Pair (UTP)*;
- J1939/21: *Data Link Layer*
- J1939/31: *Network Layer*
- J1939/71: *Vehicle Application Layer*
- J1939/73: *Application Layer - Diagnostics*
- J1939/74: *Application - Configurable Messaging*
- J1939/75: *Application Layer - Generator Sets and Industrial*
- J1939/81: *Network Management*

O principal documento que descreve o formato de mensagem utilizado pelo protocolo J1939 é o SAE J1939/21, onde é padronizado que toda mensagem enviada deve seguir o formato *Extended CAN*. É utilizado o campo de identificador de 29 bits previsto pelo formato *Extended CAN*; o campo de identificador em conjunto com o campo de *Data Field* é conhecido no protocolo SAE J1939 como *Protocol Data Unit* (PDU). A figura 35 mostra a representação do PDU no protocolo SAE J1939.

Figura 35 – Representação de um *Protocol Data Unit*



Fonte: Voss (2008b).

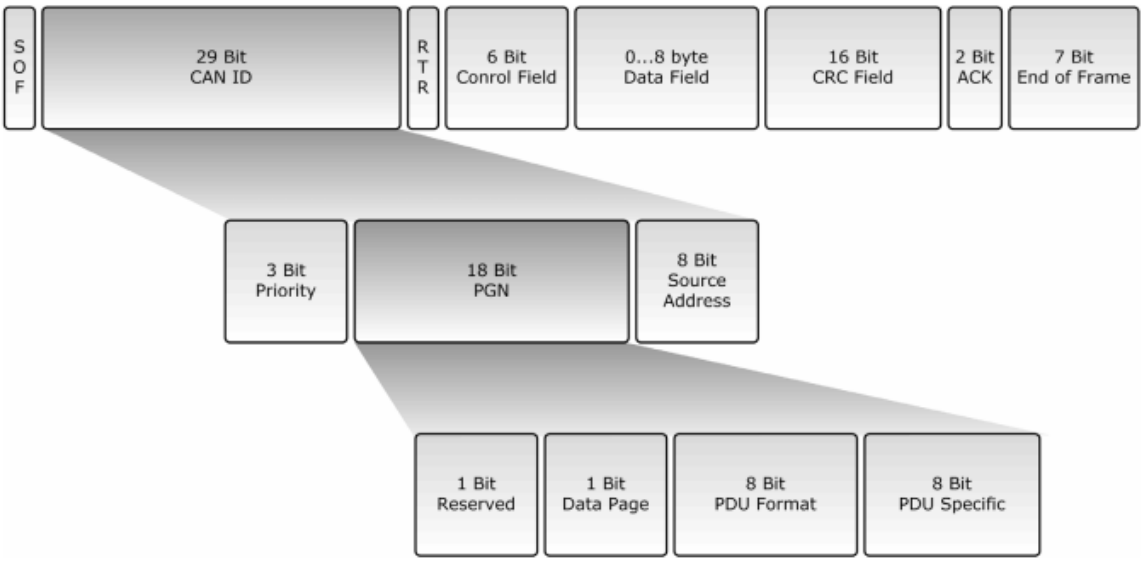
No protoloco SAE J1939, o campo de identificador de 29 bits é subdividido em três campos principais, que são o campo de prioridade, contendo 3 bits, o campo PGN (*Parameter Group Number*), composto de 18 bits, e o campo de endereço de origem, composto por 8 bits. O campo PGN ainda possui mais subdivisões, possuindo um campo de 1 bit reservado, um campo de 1 bit denominado de *Data Page*, um campo de 8 bits denominado de *PDU Format* e um último campo de 8 bits denominado de *PDU Specific*. O formato de mensagem padronizado para o protocolo SAE J1939 é mostrado na figura 36.

O principal objetivo do *Parameter Group Number*, do inglês número de grupo de parâmetro, é padronizar e identificar as ECU's que são conectadas no barramento; um exemplo de ECU que é representada por um PGN é a ECU que monitora a temperatura do motor. Além do PGN, tem-se o campo de prioridade, que é composto de 3 bits, onde o valor 000 possui a maior prioridade e o valor 111 possui a menor prioridade, e o campo de endereço de origem, que representa o endereço de origem da transmissão e deve ser único para cada ECU conectada no barramento.

O *Parameter Group Number* é descrito no documento SAE J1939/21 e toda a padronização e listagem é realizada no documento SAE J1939/71. Um exemplo de PGN descrito no documento SAE J1939/71 é representado na figura 37.

No exemplo em questão, o documento SAE J1939/71 padronizou que o PGN atribuído à unidade que monitora a temperatura do motor possui o número **65262**, ou **0xFEEE** em hexadecimal, além de outras informações, como o nível de prioridade de uma mensagem

Figura 36 – Formato de mensagem do protocolo SAE J1939



Fonte: Voss (2008b).

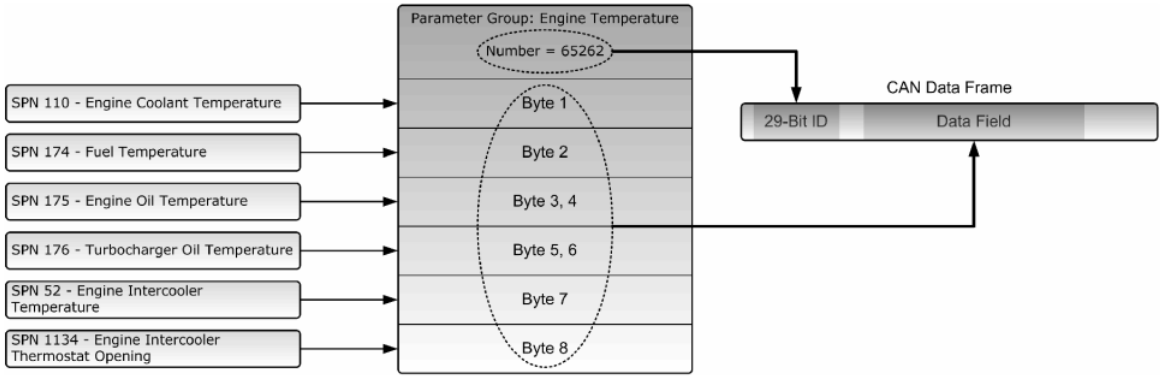
Figura 37 – Exemplo de um PGN descrito no documento SAE J1939/71

PGN 65262		**Engine Temperature**	
Transmission Rate		1 sec	
Data Length		8 bytes	
Data Page		0	
PDU Format (PF)		254	
PDU Specific (PS)		238	
Default Priority		6	
PG Number		65262 ($FEEE_{hex}$)	
Description of Data			**SPN**
Byte	1	Engine Coolant Temperature	110
	2	Fuel Temperature	174
	3, 4	Engine Oil Temperature	175
	5, 6	Turbocharger Oil Temperature	176
	7	Engine Intercooler Temperature	52
	8	Engine Intercooler Thermostat Opening	1134

Fonte: Voss (2008b).

transmitida por essa unidade. É possível ver no exemplo que há dentro de um PGN uma subdivisão conhecida como *Suspect Parameter Number* (SPN). O documento SAE J1939/71 padroniza qual informação será transmitida por cada byte do campo *Data Field* referente ao PGN associado. A informação sobre a temperatura do combustível, por exemplo, é transmitida por um *frame* que contem o PGN **65262** através do segundo *byte* do campo *Data Field*. A figura 38 demonstra o relacionamento entre um PGN, seus SPN's e o *Extended CAN*.

Figura 38 – Representação de um PGN e seus SPN's



Fonte: Voss (2008b).

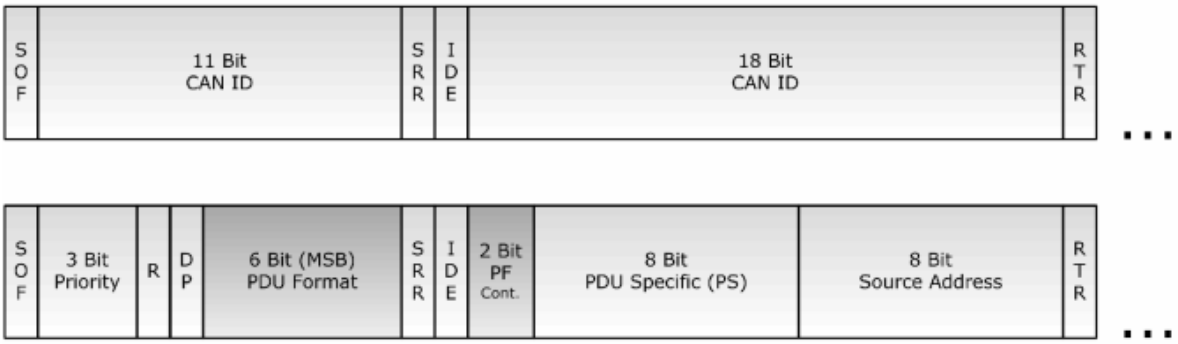
O *Parameter Group Number* é subdivido nos seguintes campos:

- R: campo de 1 bit que é reservado para propósitos futuros. Deve ser sempre transmitido com o valor de um bit dominante.
- *Data Page* (DP): campo de 1 bit que determina a página em que é definido a mensagem no padrão especificado pelos documentos de padronização do SAE J1939. Atualmente, esse bit é sempre transmitido com o valor 0 por todas as ECU's citadas na especificação, e o bit 1 é reservado para uma futura expansão.
- *PDU Format* (PF): campo de 8 bits que define a forma de comunicação que será adotada pela ECU ao transmitir a mensagem. Se o valor do campo tiver um valor entre 0 e 239, significa que a mensagem enviada ao barramento CAN possuirá um endereço de destino específico. Caso o valor do campo seja entre 240 e 255, significa que a mensagem será transmitida ao barramento CAN para todos os nós conectados no barramento, cabendo a cada nó realizar a leitura da mensagem e definir se realizará alguma ação de acordo com a mensagem.
- *PDU Specific* (PS): campo de 8 bits que possui uma determinada função dependendo do valor do campo *PDU Format*. Se o campo *PDU Format* possuir um valor entre 0 e 239, isso implica que o campo *PDU Specific* contém o valor do endereço de destino para o qual deve ser transmitida a mensagem; caso o campo

PDU Format possua um valor entre 240 e 255, isso implica que não é necessário preencher nenhum endereço de destino e o campo *PDU Specific* é utilizado como uma extensão do campo *PDU Format*.

Com a subdivisão dos campos do *Parameter Group Number*, é possível representar o modelo final de um frame do protocolo SAE J1939. A figura 39 demonstra como é definido o identificador de 29 bits no protocolo SAE J1939 com base no modelo *Extended CAN*.

Figura 39 – Representação detalhada do campo de identificação do protocolo SAE J1939



Fonte: Voss (2008b).

3 MATERIAIS E MÉTODOS

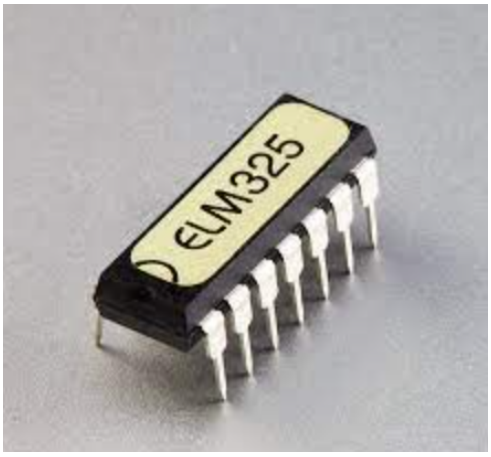
Inicialmente foi feita a construção da fundamentação teórica dos protocolos J1587 e J1939, conforme estudo do referencial bibliográfico. A partir dos materiais especificados a seguir, é construído o protótipo de testes e a versão finalizada em placa de circuito impresso, utilizando as metodologias descritas na seção 3.2.

3.1 Materiais

Nesta seção é feito o levantamento dos componentes e circuitos integrados utilizados para a correta leitura do protocolo J1587 e conversão para J1939. Isso foi feito a partir do estudo sobre os microcontroladores disponíveis no mercado.

Como interpretador do protocolo J1587, o objetivo é utilizar o circuito integrado ELM325, visto na figura 40.

Figura 40 – ELM325

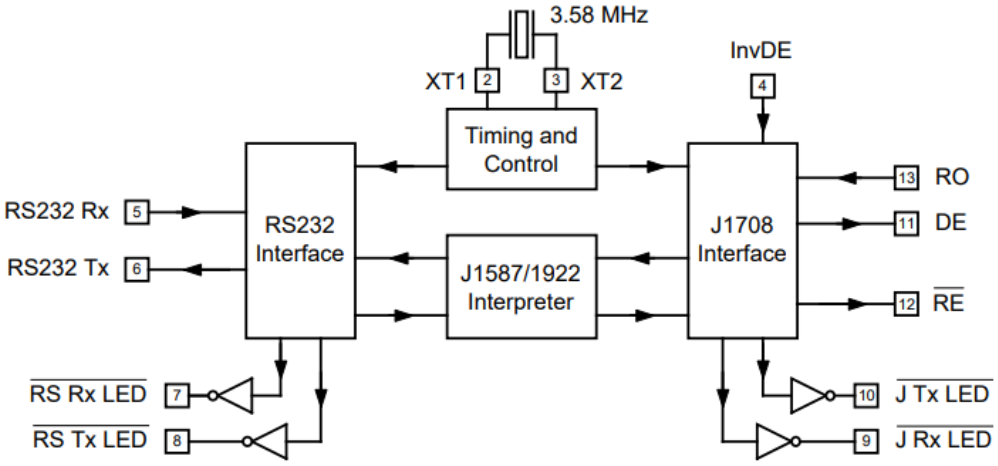


Fonte: ELMElectronics (2023b).

Este circuito integrado foi desenvolvido especificamente para esta função. Através de uma interface J1708 conectada internamente com um interpretador de mensagens J1587, esse dispositivo é capaz de coletar e transmitir as informações da rede.

O ELM325 opera em uma tensão de aproximadamente 5V, consumindo uma corrente de até 2mA em caso de máxima utilização. Ele funciona utilizando comandos do tipo AT enviados através de uma interface RS232. Dessa forma, diversas funções relacionadas à coleta de informações podem ser configuradas. O esquemático do funcionamento interno do componente pode ser visto na figura 41.

Figura 41 – Diagrama de blocos do ELM325



Fonte: ELMElectronics (2023a).

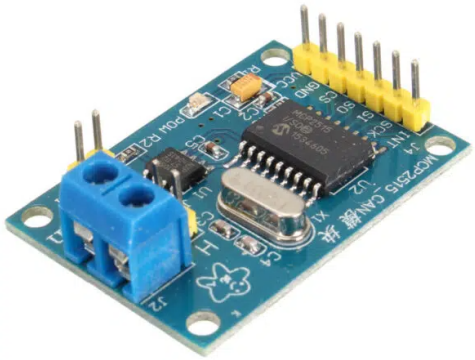
Neste projeto pretende-se utilizar o microcontrolador ESP32-WROOM-32, para realizar a lógica de conversão entre os protocolos. Ele foi escolhido devido ao baixo custo e ampla flexibilidade no design do projeto. Este microcontrolador pode ser visto na figura 42.

Figura 42 – ESP32

Fonte: Saravati (2023).

O módulo CAN BUS MCP2515, visto na figura 43, é um componente utilizado para comunicação com redes CAN, muito utilizadas no ambiente industrial e automotivo. Este opera em uma tensão de 5V com corrente de 5mA. Pode-se conectar este módulo ao microcontrolador, efetuar a leitura dos dados da rede CAN e posteriormente enviar esses dados ou acionar algum dispositivo conforme com as informações colhidas. Este módulo será utilizado como transceptor do protocolo J1939.

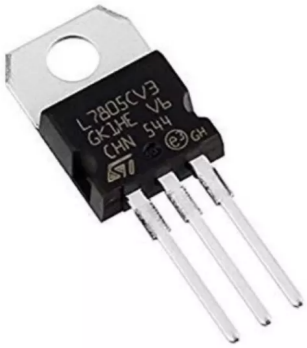
Figura 43 – MCP2515 e TJA1050



Fonte: MakerHero (2023).

Para garantir a estabilidade do sistema, uma vez que a tensão de operação de um veículo pesado pode variar entre 23 até 28V, um regulador de tensão linear do tipo positivo foi escolhido, o LM7805 visto na 44. Este também garante a alimentação de 5V conforme o projeto necessita.

Figura 44 – LM7805



Fonte: CentralChip (2023).

A comunicação entre as unidades da rede automotiva utiliza transceptor RS-485, conforme a norma SAE J1708. Este transceptor está disponível na forma do circuito integrado da marca MAXIM, o MAX485 que é visto na figura 45. Este componente é conectado ELM325, conforme a exemplificação da norma J1708.

Figura 45 – MAX485

Fonte: Analog (2023).

O VN1610 é um módulo fabricado pela Vector que é uma solução flexível e econômica para aplicações CAN/CAN FD, LIN, K-Line e J1708. Opera com excelente desempenho e tempos de latência mínimos, além da alta precisão. Este módulo será utilizado para realizar a leitura dos dados recebidos através do microcontrolador como se tivessem sido originalmente enviados no barramento J1939.

Figura 46 – VN1610

Fonte: Vector (2023).

Além dos dispositivos citados, também são utilizados no projeto capacitores, resistores, diodos e LEDs conforme a necessidade do circuito. Com o levantamento finalizado, será montado o circuito para a leitura do protocolo J1587. Primeiramente, o circuito será montado em *protoboard*. A versão final do protótipo será feita em placa de circuito impresso.

3.2 Protótipo

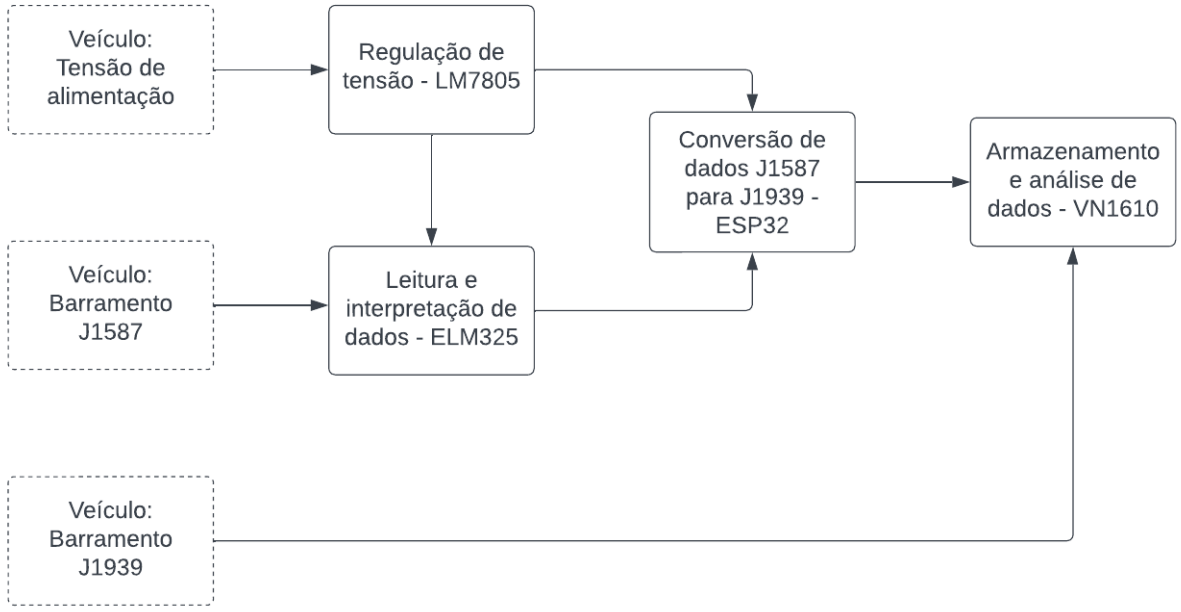
O protótipo será composto pelos componentes centrais descritos na seção 3.1, e a lógica de construção do circuito pode ser vista na figura 47.

O primeiro bloco ilustra o regulador de tensão LM7805. Este recebe a energia proveniente da fonte de 24V, utilizada para simular a tensão de um veículo real e a lineariza para distribuir para o resto do circuito, limitando a saída em 5V.

O bloco seguinte ilustra a leitura e interpretação dos sinais realizados pelo ELM325. Os dados são então enviados através da comunicação serial RS232 para o microcontrolador ESP32 ilustrado no terceiro bloco. O microcontrolador implementa o código desenvolvido para traduzir os sinais J1587 para J1939, e então envia-os na rede, para serem lidos pelo usuário através do módulo VN1610, ilustrado no último bloco.

O propósito do desenvolvimento do protótipo é que este funcione de forma que os usuários não precisem alterar a configuração dos componentes para que a leitura de dados seja realizada com sucesso.

Figura 47 – Diagrama de blocos do protótipo.



Fonte: Autoria própria (2024).

Com o protótipo montado, serão feitos testes de leitura para validação dos requisitos necessários para realizar a correta leitura de mensagens do protocolo J5187.

Nesse passo será feita a conversão entre os protocolos. O microcontrolador receberá a mensagem no protocolo J1587 e realizará a conversão para o protocolo J1939. Primeiramente será testado o envio da mensagem ao microcontrolador de forma adequada e então será desenvolvido o código de conversão entre os protocolos. Com o código pronto, será feita a

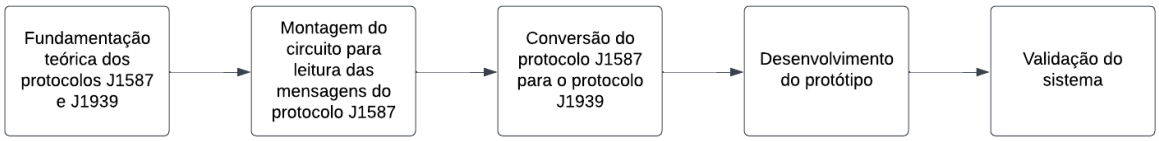
leitura do sinal de saída e validado que a mensagem foi corretamente gerada de acordo com os requisitos funcionais do protocolo J1939 através de testes de bancada.

Com o projeto montado em *protoboard* funcionando corretamente e todos os testes propostos realizados com sucesso, será feito o desenvolvimento e montagem da placa de circuito impresso da ponte de ligação entre redes automotivas veiculares.

O passo final será a validação do sistema. Com a placa montada, será escolhido um *software* de mercado para que seja feita a leitura e decodificação da mensagem gerada.

O passo a passo da metodologia utilizada está ilustrado na figura 48.

Figura 48 – Diagrama de blocos da metodologia utilizada.



Fonte: Autoria própria (2024).

4 RESULTADOS

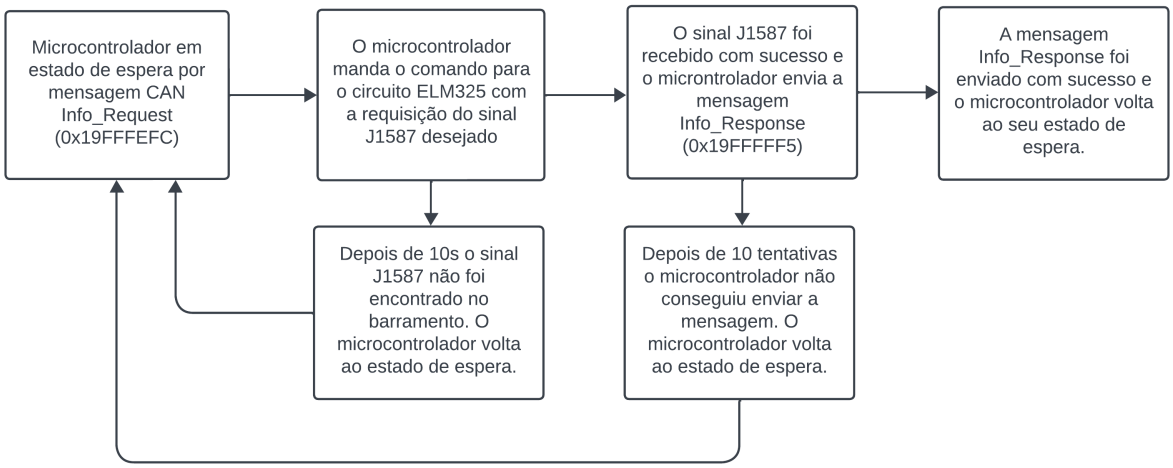
4.1 Modelagem do sistema

Para realizar a conversão do J1587 interpretado pelo ELM325, foi desenvolvido o código implementado no microcontrolador ESP32, cuja lógica está descrita na figura 49.

O primeiro bloco ilustra a inicialização do sistema, onde o microcontrolador está esperando receber os dados provenientes do circuito integrado. Isso é feito através de uma requisição que é enviada pelo VN1610 no barramento J1939.

A partir disso, o microcontrolador manda este comando para o interpretador, que requisita o sinal J1587 equivalente. Dentro de uma janela de dez segundos, o microcontrolador espera receber o sinal solicitado. Caso não o receba, volta para o estado inicial de espera. Quando o sinal é recebido como esperado, o microcontrolador envia a mensagem Info_Response para o VN1610. O ESP32 conta dez tentativas de envio da mensagem com sucesso. Caso contrário, volta ao estado inicial.

Figura 49 – Diagrama de blocos do código.

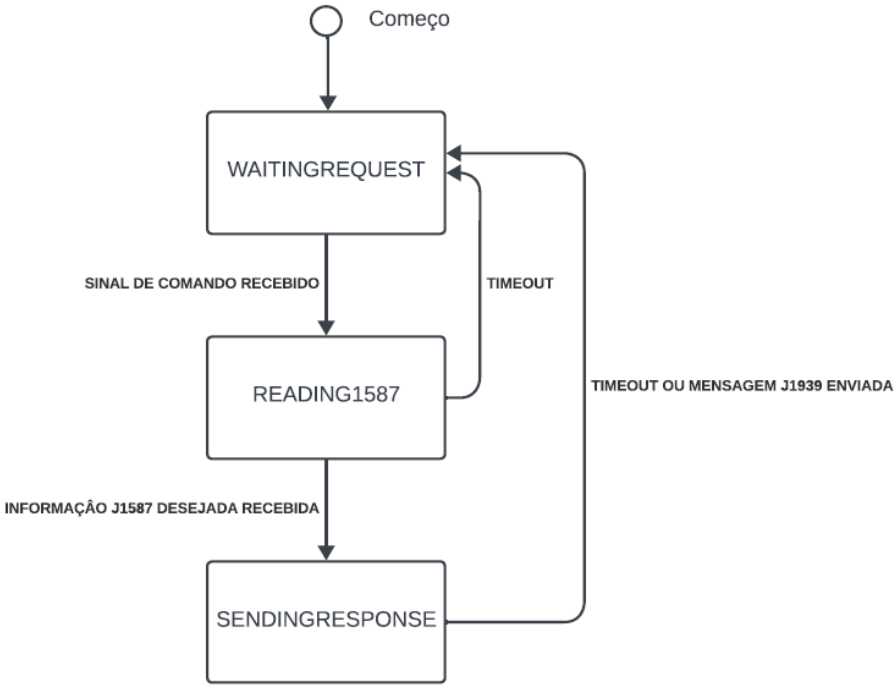


Fonte: Autoria própria (2024).

Essa lógica pode ser resumida através da ilustração da máquina de estados finita desenvolvida, ilustrada na figura 50.

A máquina de estados está separada em estado de espera, estado de requisição dos sinais J1587 e o estado da resposta de J1939.

Figura 50 – Máquina de estados do *software* desenvolvido.



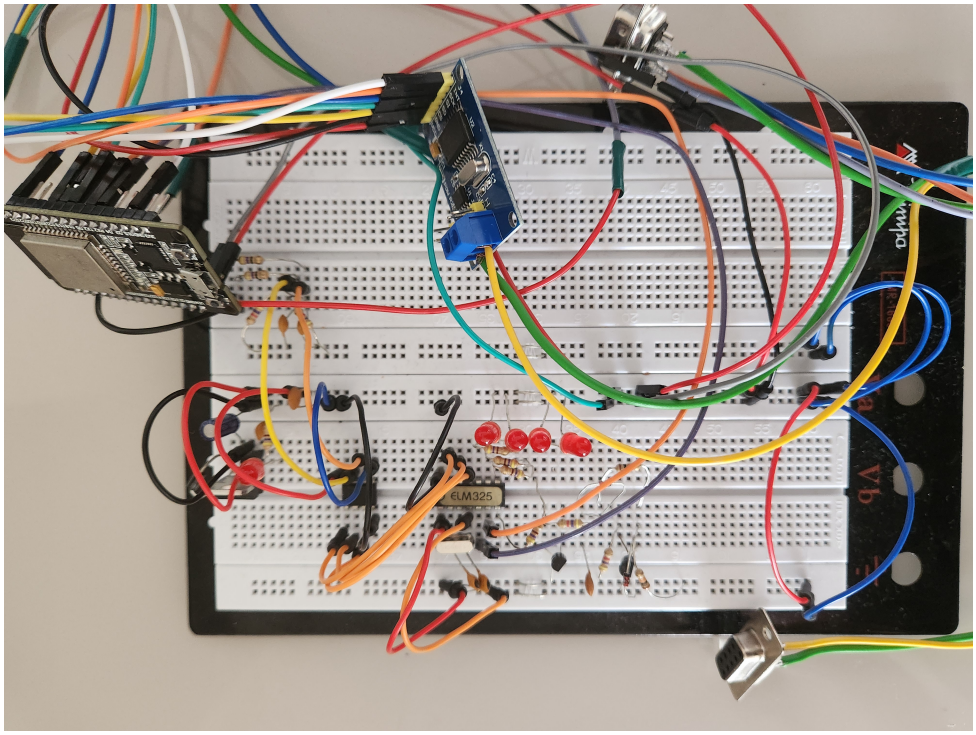
Fonte: Autoria própria (2024).

4.2 Montagem do protótipo

4.2.1 Montagem na *protoboard*

A fase inicial de implementação projeto consistiu na montagem do circuito em uma placa de prototipagem, *protoboard*, a fim de validar o *design* tanto do *hardware* quanto do *software*. A montagem do circuito em *protoboard* é apresentada na figura 51.

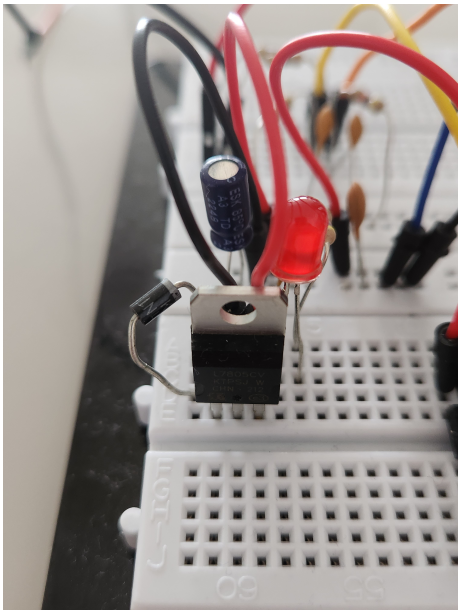
Figura 51 – Dispositivo montado em *protoboard*.



Fonte: Autoria própria (2024).

O circuito foi separado em blocos funcionais. O primeiro a ser descrito é o regulador de tensão, figura 52, implementado com o LM7805. Assim como descrito anteriormente, o protótipo deve operar em uma tensão de 5V a partir da alimentação proveniente da bateria do veículo.

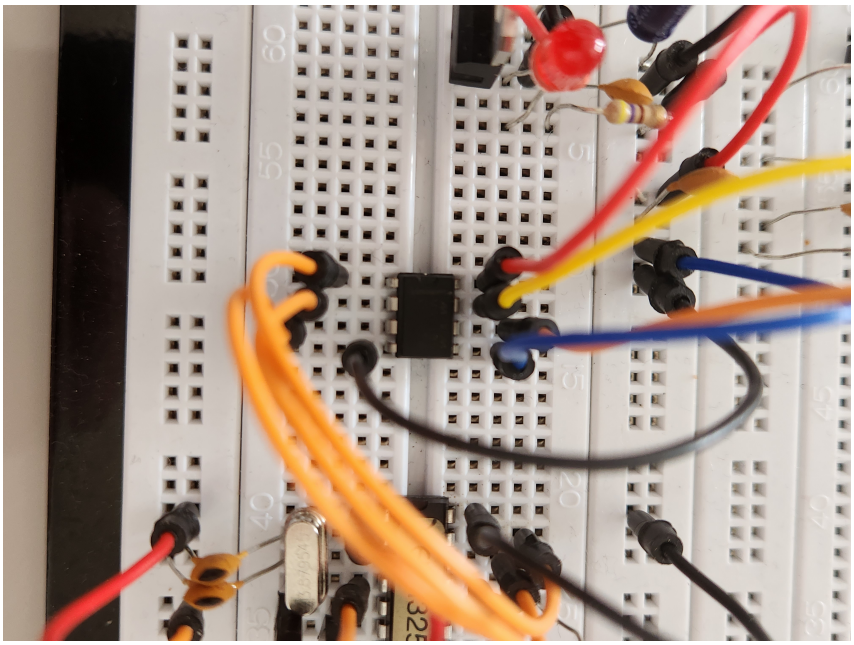
Figura 52 – Circuito Regulador de Tensão com LM7805.



Fonte: Autoria própria (2024).

A seguir, ilustrado na figura 53, foi montado o transceptor RS-485. Este bloco é responsável por processar a camada física da rede J1708, utilizando o circuito integrado MAX485.

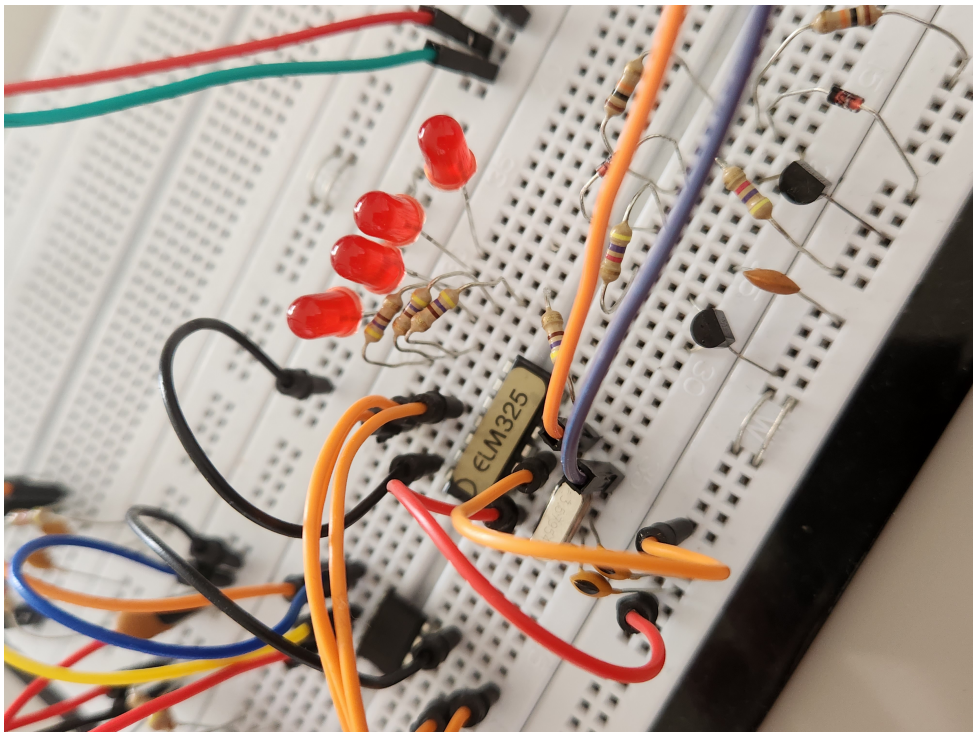
Figura 53 – Transceptor MAX485.



Fonte: Autoria própria (2024).

A figura 54 segue o interpretador J1587, repensável por ler, filtrar e enviar sinais no barramento.

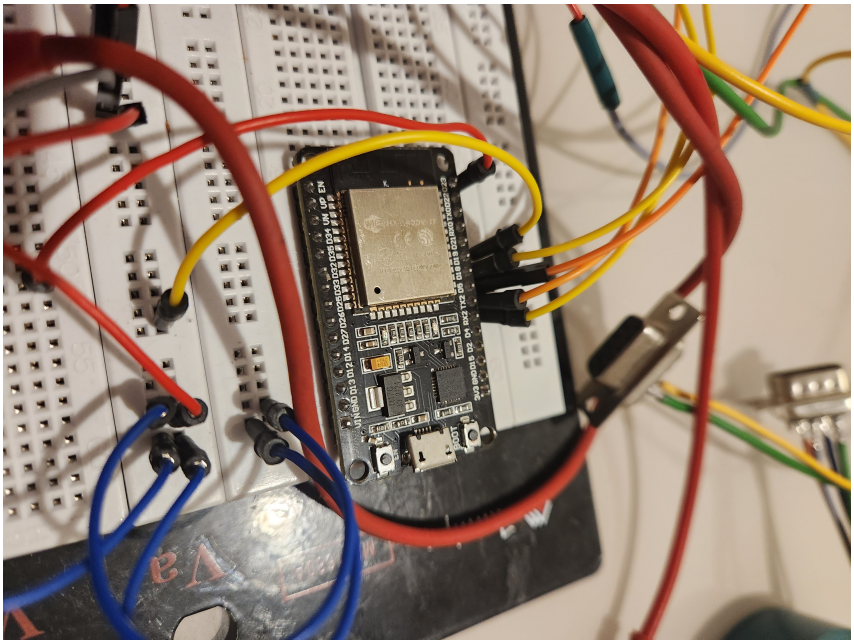
Figura 54 – ELM325 em *protoboard*.



Fonte: Autoria própria (2024).

O próximo bloco consiste no microcontrolador ESP32-WROOM-32d, como ilustra a figura 55.

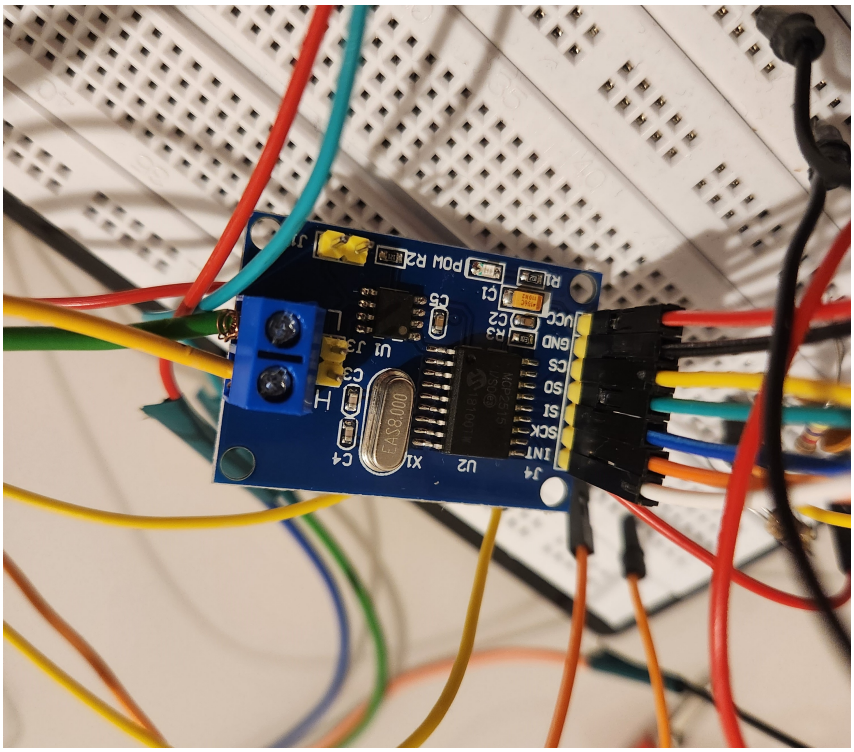
Figura 55 – Microcontrolador ESP32 em *protoboard*.



Fonte: Autoria própria (2024).

Por último, a figura 56 traz o módulo MCP2515 junto TJA1050, responsáveis pela transcepção e controle da rede CAN J1939.

Figura 56 – Circuito MCP2515 e TJA1050 em *protoboard*.



Fonte: Autoria própria (2024).

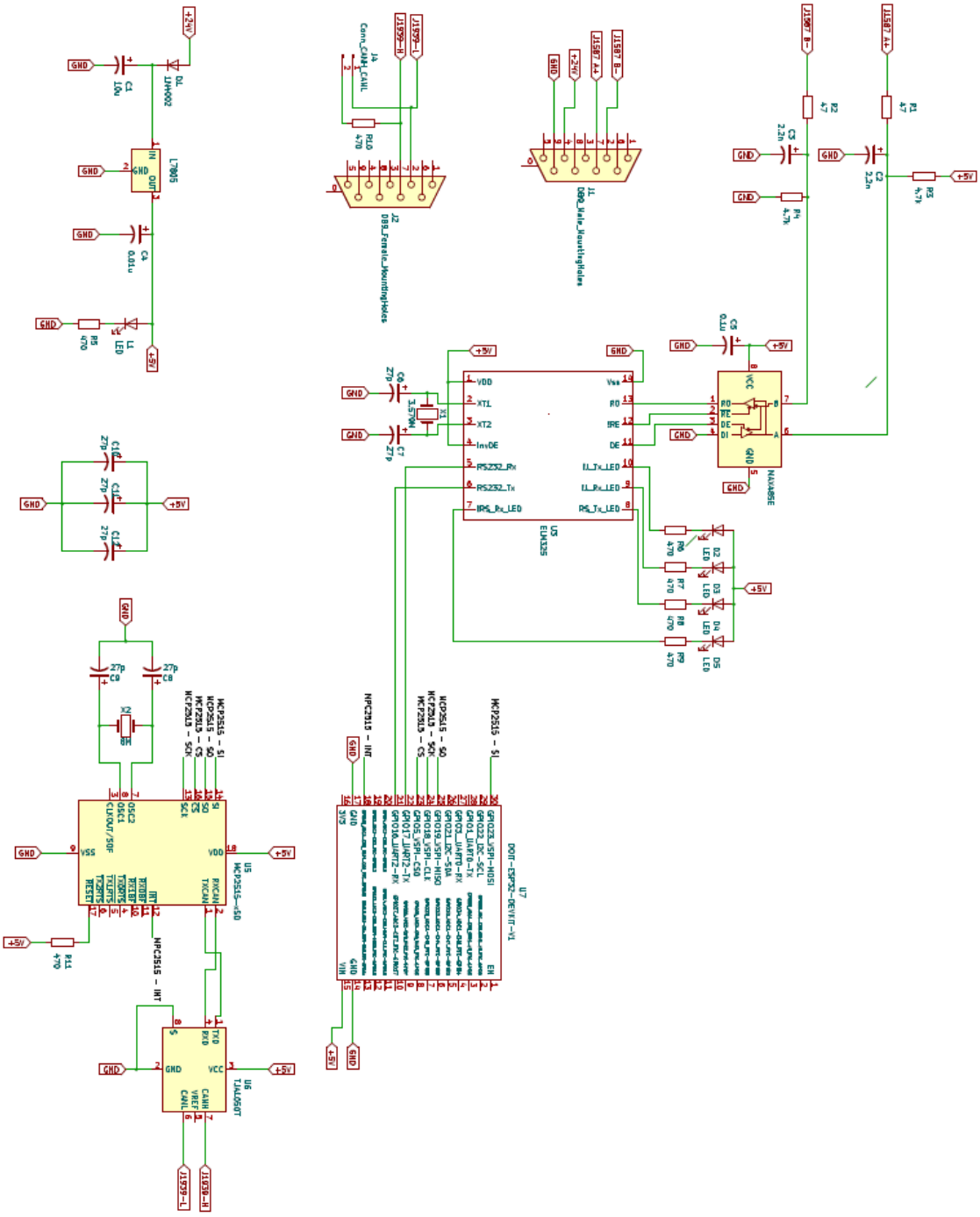
4.2.2 Montagem na placa de circuito impresso

Para facilitar a utilização do usuário, foi realizado o desenvolvimento da PCB (*printed circuit board*) através do *software* KiCad.

O KiCad é um *software* de código aberto para automação de *design* eletrônico. Este programa gera os arquivos das camadas de cobre, legendas, máscara de solda e furação necessários para impressão da placa de circuito (KICAD, 2023). Para gerar a PCB, o primeiro passo é desenhar esquemático. Este pode ser visto na figura 57. A partir do esquemático é feito o posicionamento dos componentes e roteamento da placa.

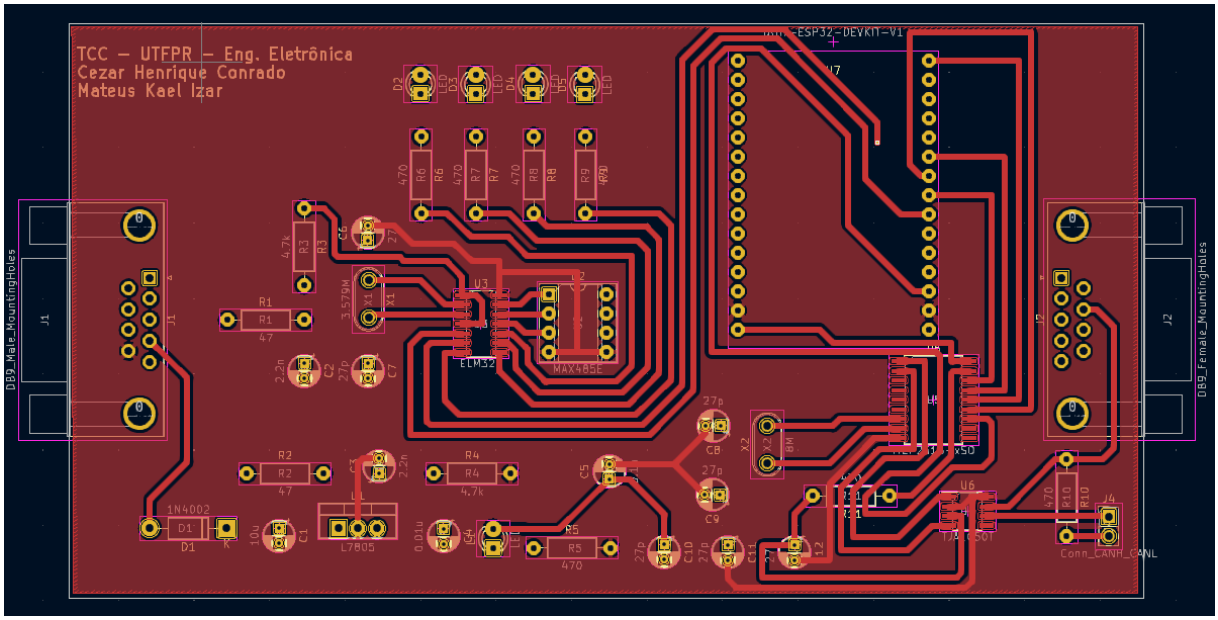
As figuras 58 e 59 ilustram as camadas de cobre, que são as camadas condutivas após a gravação da placa, sendo trilhas ou camadas de energia e aterramento. Para este projeto, foi definida a utilização de duas camadas, devido à facilitação do roteamento. A figura 60 ilustra a furação da placa.

Figura 57 – Esquema elétrico da placa de circuito.



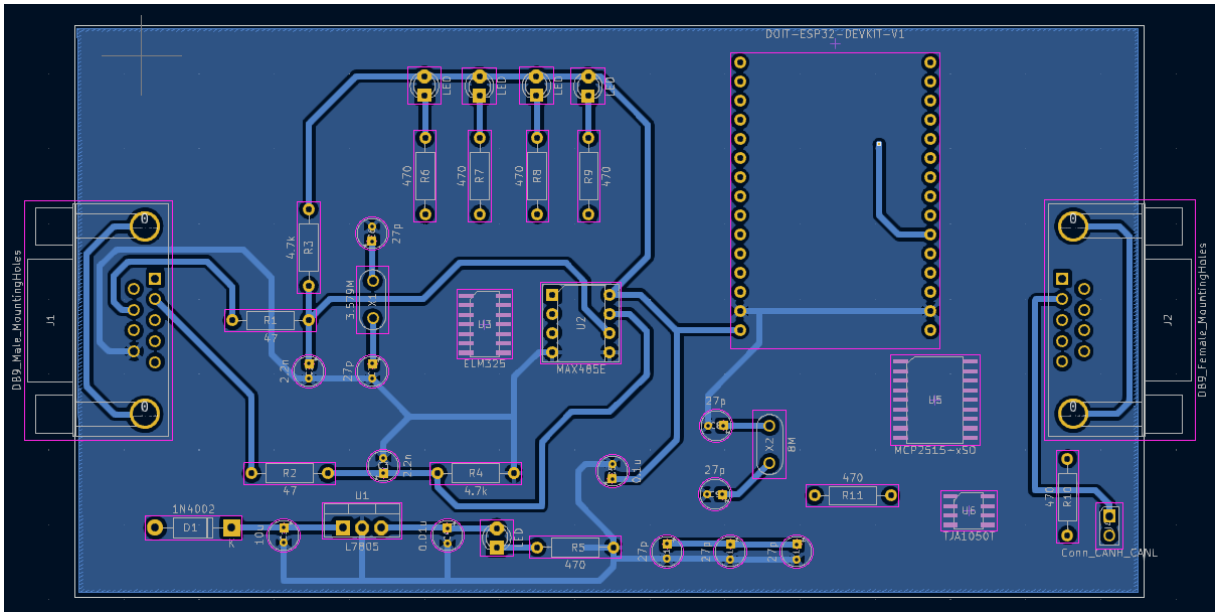
Fonte: Autoria própria (2024).

Figura 58 – Camada de cobre frontal.



Fonte: Autoria própria (2024).

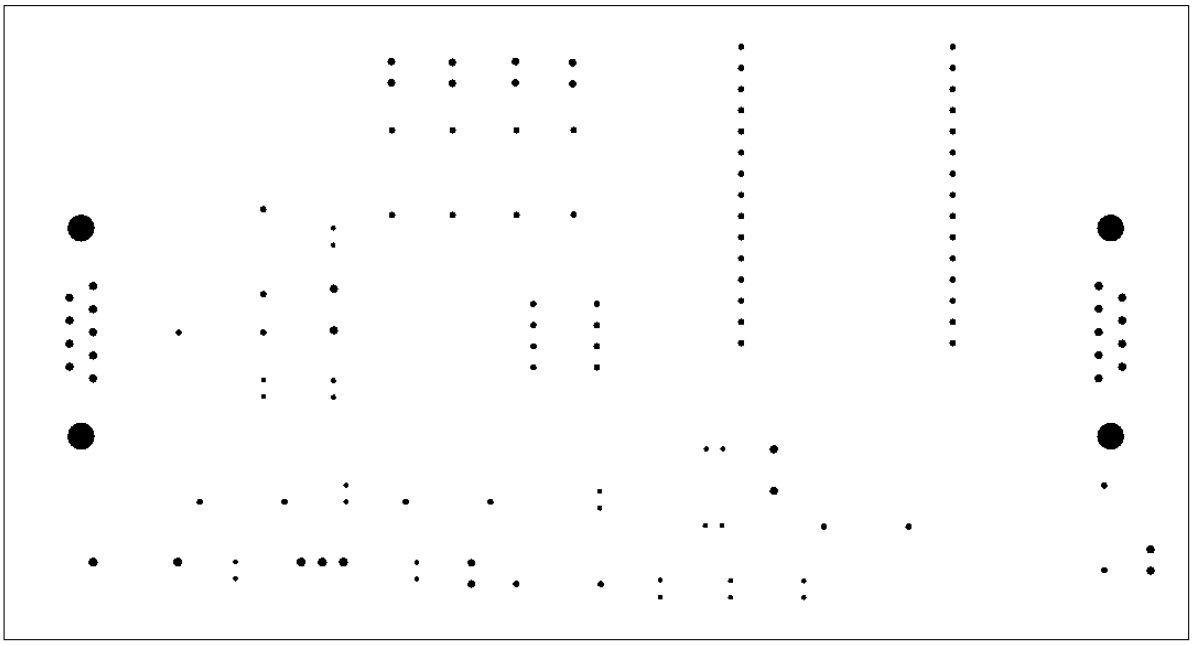
Figura 59 – Camada de cobre traseira.



Fonte: Autoria própria (2024).

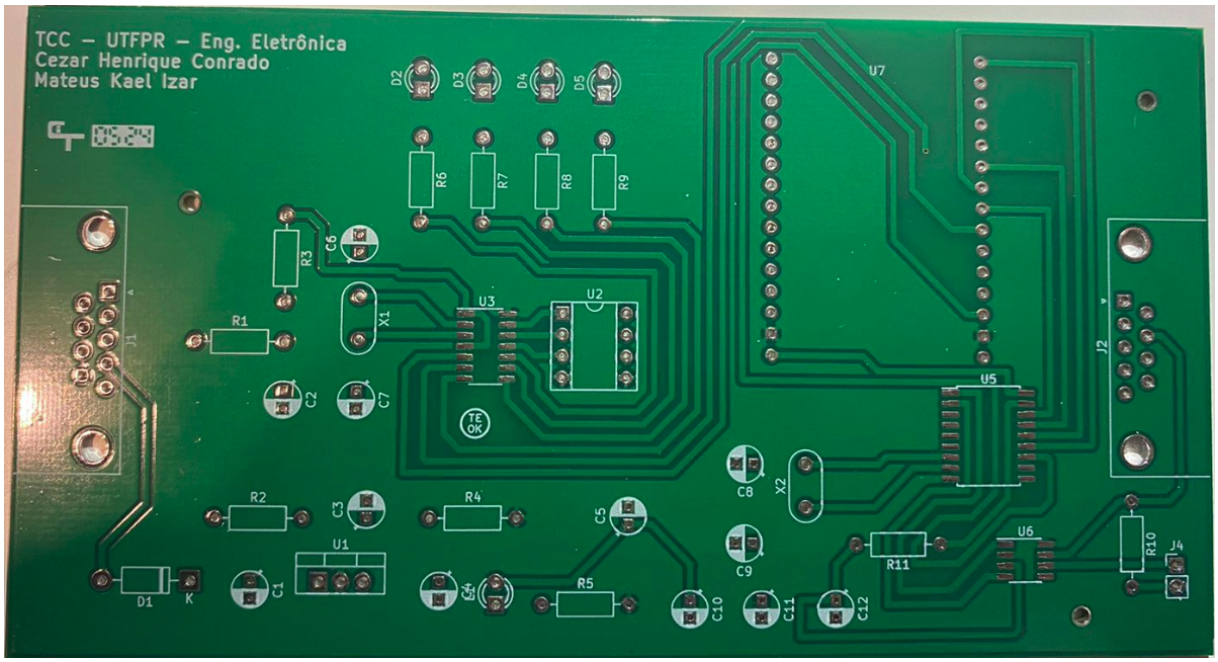
A fim de permitir com que o usuário final atualize o código do microcontrolador conforme demanda, foi decidido integrar o kit de desenvolvimento do ESP32 e conectá-lo a placa via uma barra de *socket*. Na figura 61 podemos observar a placa de circuito impresso e na figura 62 apresentá-se o produto final deste trabalho já com os componentes soldados.

Figura 60 – Furação da placa.



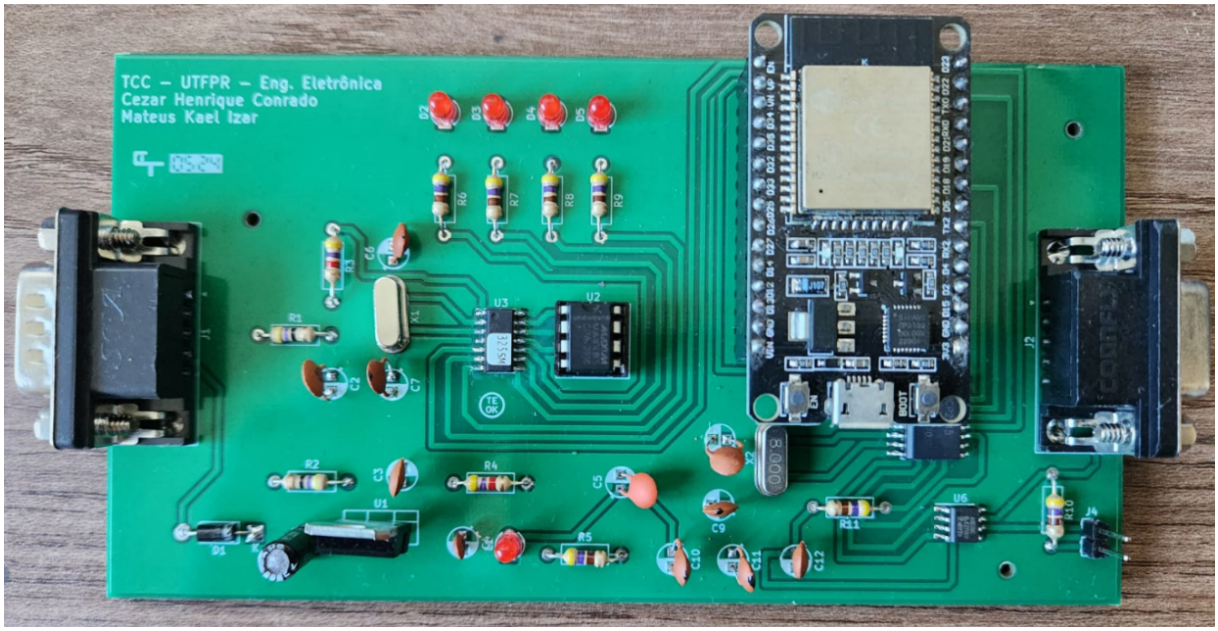
Fonte: Autoria própria (2024).

Figura 61 – Placa de circuito impresso.



Fonte: Autoria própria (2024).

Figura 62 – Placa de circuito impresso com os componentes soldados.



Fonte: Autoria própria (2024).

4.2.3 *Database* de comunicação *CAN*

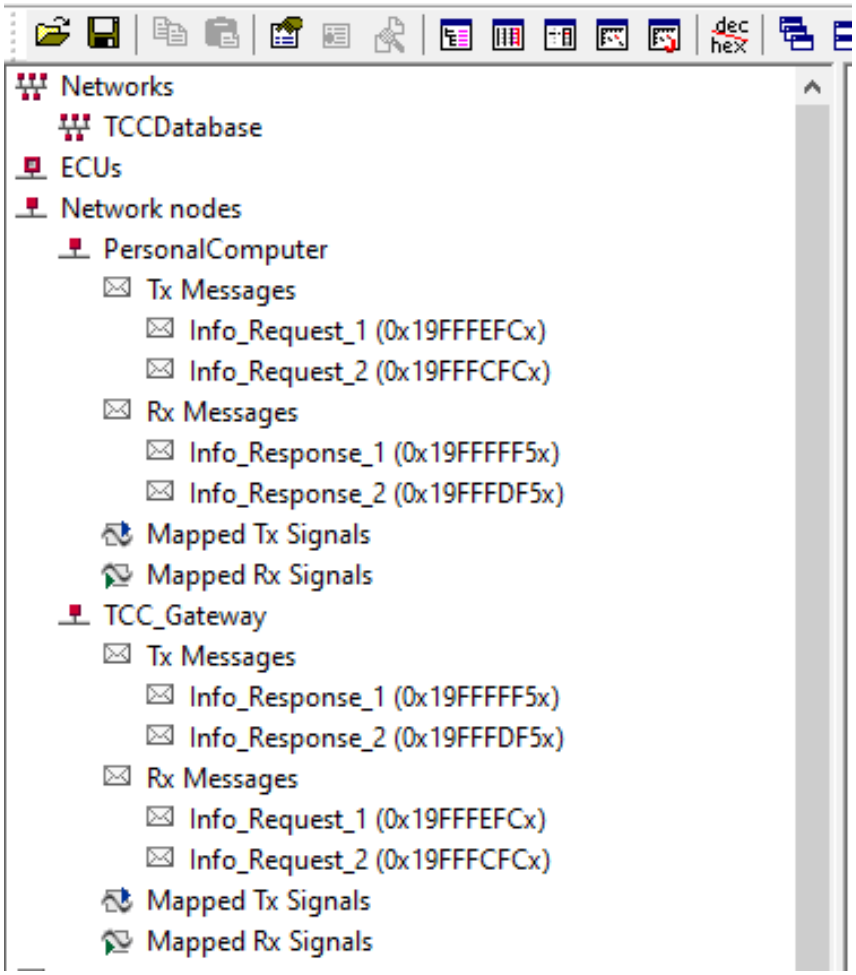
A fim de permitir a compreensão dos dados, foi definida uma *Database CAN* utilizando o *software CANdb++* da *Vector*. Este documento é descritivo das mensagens disponíveis no barramento de comunicação. Desta forma é possível como que o usuário final possa utilizar os dados da rede de forma compreensível.

Como é possível observar na figura 63, foram definidos dois nós de comunicação. O nó *PersonalComputer* contem as mensagens de requisição de informações para o protótipo, enquanto o nó *TCC_Gateway* contem as mensagens de resposta do protótipo para o computador do usuário final.

As figuras 64 e 65 ilustram o posicionamento dos sinais dentro da mensagem. Nas mensagens de requisição temos os sinais de *MID*, tipo de sinal (*PID*, *SID*, *PPID*, *PSID*), definições se o sinal está incluso dentro dos sinais de falha para o painel de instrumentos (PID *194* ou *PPID* 208) e número do sinal de acordo com as definições do protocolo *J1587*. Já as mensagens de resposta contem o valor encontrado na rede de comunicação lenta.

É importante salientar que os endereços das mensagens foram escolhidos de forma a terem a menor prioridade possível diante aos demais *frames* da rede. Desta forma é possível restringir o impacto na comunicação, tendo em vista que estas mensagens estão fadadas a sempre perder a arbitragem e somente serem enviadas quando o barramento estiver completamente livre.

Figura 63 – Descritivo dos nós do *Database*.

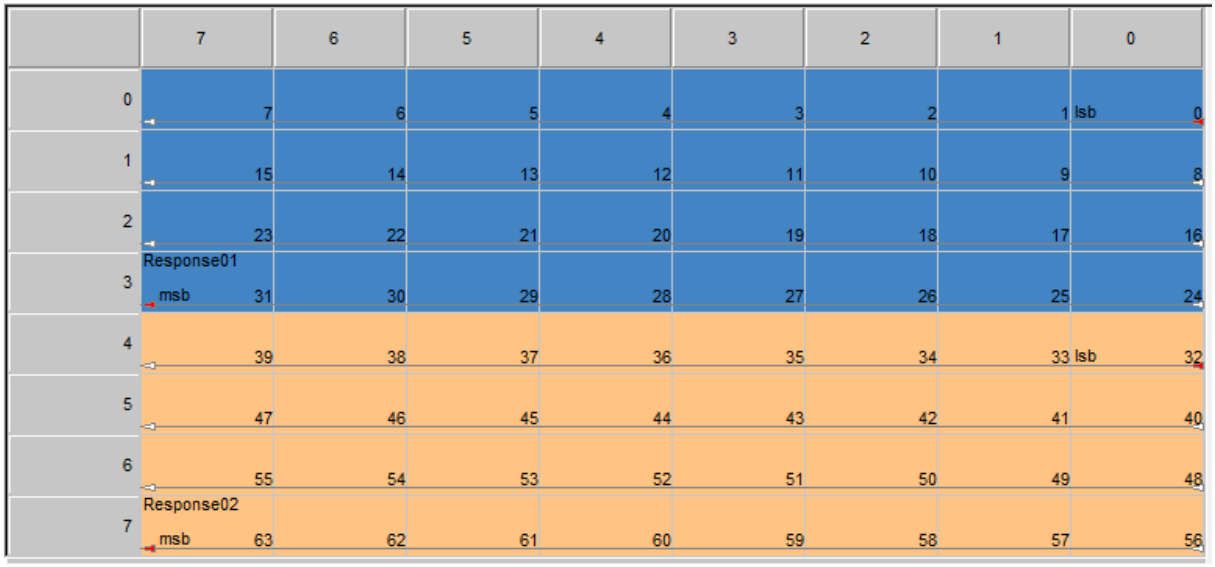


Fonte: Autoria própria (2024).

Figura 64 – Descritivo da mensagem de requisição de dados.

Fonte: Autoria própria (2024).

Figura 65 – Descritivo da mensagem de resposta do sistema.



Fonte: Autoria própria (2024).

4.3 Validação

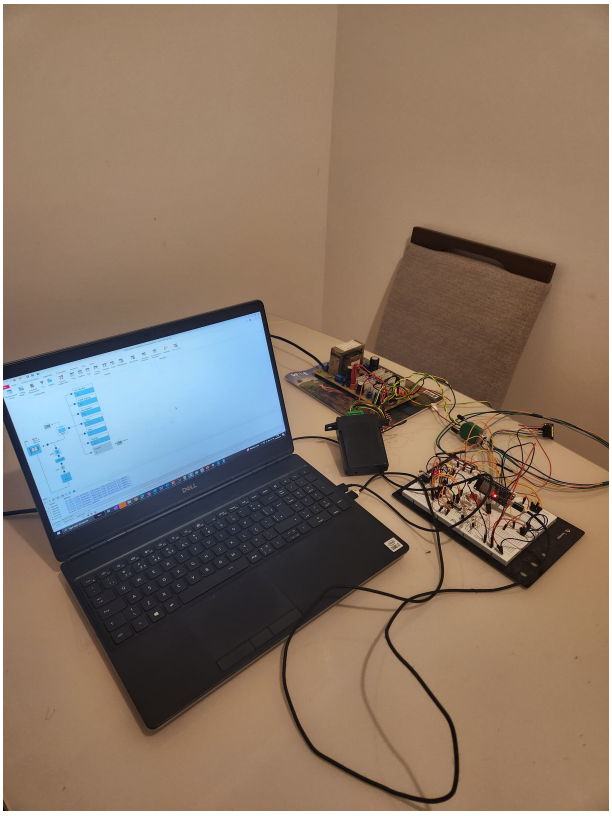
Este capítulo apresenta os resultados da validação do sistema proposto. O processo de verificação do projeto foi escalonado em três partes:

1. Validação do protótipo em *protoboard* com uma única unidade de controle.
2. Validação do protótipo em placa de circuito impresso com uma única unidade de controle.
3. Validação do protótipo em placa de circuito impresso conectado a uma bancada de veículo completo.

O objetivo da primeira etapa é verificar o funcionamento básico dos componentes e estruturar um mínimo produto viável a fim de servir de base para as etapas posteriores de desenvolvimento. A validação em placa de circuito impresso visa validar o *design* da placa e considerar o escopo completo do protótipo. Por fim, a última etapa de validação objetiva confrontar a proposta deste trabalho em um ambiente similar ao ambiente real, se utilizando de uma bancada de sistema completo.

4.3.1 Validação do protótipo em *protoboard* com uma única unidade de controle

Figura 66 – Bancada de teste com protótipo em *protoboard*.



Fonte: Autoria própria (2024).

O protótipo foi validado em bancada utilizando uma unidade de controle real, figura 66. A fonte regulada foi configurada em 24V, simulando a tensão de bateria padrão para veículos pesados. Para esta validação, foi escolhido uma unidade de gerenciamento de frota, *FMS (Fleet Management System) Gateway*, apresentada na figura 67.

Figura 67 – FMS.

Fonte: Autoria própria (2024).

É possível observar o resultado do programa tanto via a opção de *debug* na comunicação serial (figura 68), quanto pelo *software* de interação com rede *CAN*, *CANAlyzer* (figura 69). Verifica-se o funcionamento do protótipo em captar com sucesso as informações da rede J1587 sendo as tais:

1. *MID* 179, 0xB3 em hexadecimal - número de identificação da unidade de controle *FMS Gateway*.
2. *Signal Type* 0 - a informação requisitada é um *PID*.
3. *Siganal Number* 44, 0x2C em hexadecimal - A informação requisitada é o *PID* 44.
4. *Data* 196, 0xC4 em hexadecimal - dado retornado pelo protótipo.

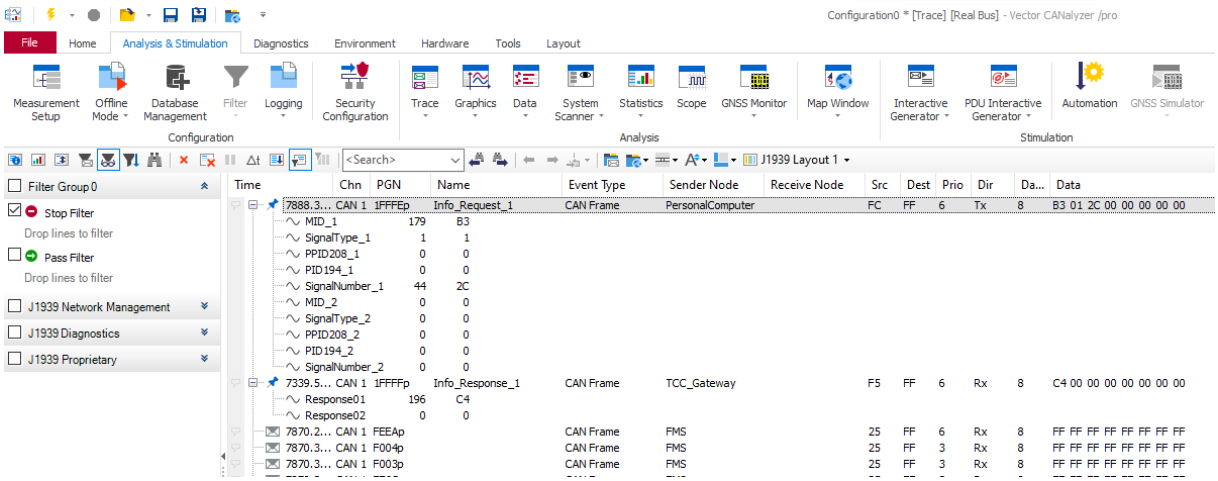
Também foi possível verificar o sistema contra um *software* de analise J1587 (Figura 70), e dessa forma conferir que o valor do PID 44 é 196 (11000100b). Validou-se o protótipo em *protoboard* alcançando o mínimo produto viável como objetivado.

Figura 68 – Janela de *debug* do programa.

```
CAN ID: 19FFFEFC
MID1: 179 -- SignalType: 0 -- SignalNumber: 44
State Reading 1587
MID: B3 PID: 2C Data: C4
Terminou Leitura J1587
State Sending CAN Response
Message Sent Successfully!
Communication Ok! Go back to waiting.
```

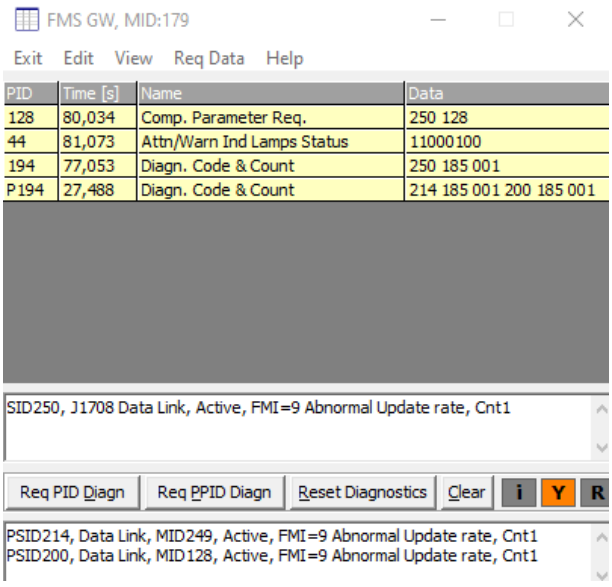
Fonte: Autoria própria (2024).

Figura 69 – Janela de dados do *CANAlyzer*.



Fonte: Autoria própria (2024).

Figura 70 – Software de Análise J1587

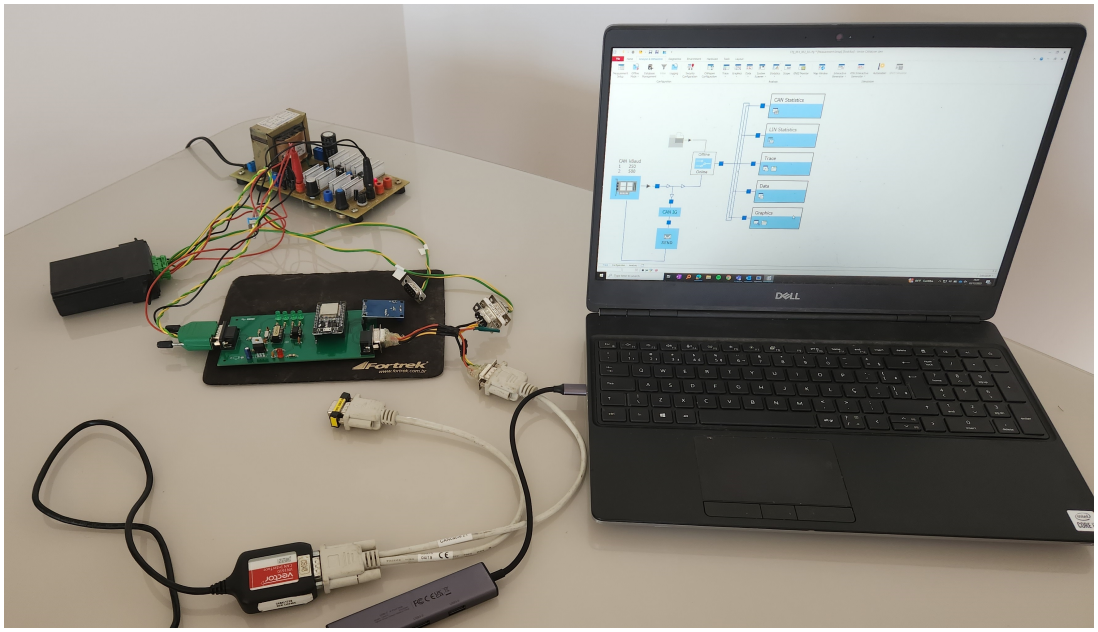


Fonte: Autoria própria (2024).

4.3.2 Validação do protótipo em placa de circuito impresso com uma única unidade de controle

Ao final do desenvolvimento do protótipo e com os componentes soldados em placa de circuito impresso, validou-se o projeto novamente em uma bancada com a unidade *FMS GW*, figura 71. Aqui buscou-se avaliar se todas as funcionalidades almejadas estavam de acordo com os requisitos desejados.

Figura 71 – Bancada de teste da verificação em *PCB*.



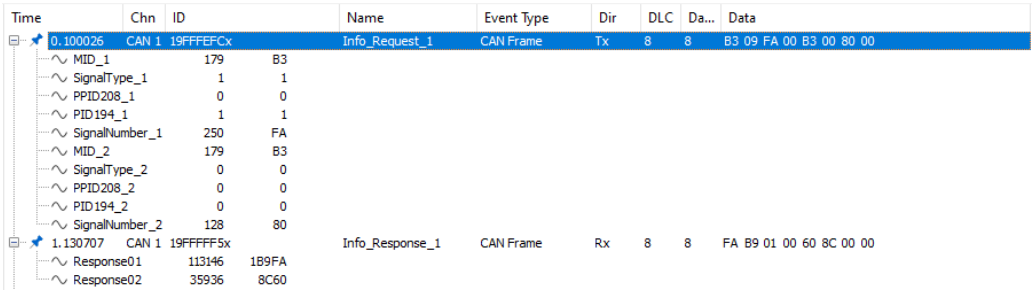
Fonte: Autoria própria (2024).

Objetivos específicos dessa fase de teste foram os seguintes:

1. Testar o sistema em placa de circuito impresso
2. Testar sistema requisitando duas mensagens ao mesmo tempo.
3. Testar a requisição de dados do *PID 194*.

É possível verificar na figura 72 os resultados conforme o esperado. Ainda assim nessa fase do projeto foram encontrados erros no *design* da placa de circuito e foram precisos 3 iterações de desenvolvimento para chegar no produto final.

Figura 72 – Resultados da verificação em placa de circuito impresso.



Time	Chn	ID	Name	Event Type	Dir	DLC	Da...	Data
0.100026	CAN 1	19FFFEFCx	Info_Request_1	CAN Frame	Tx	8	8	B3 09 FA 00 B3 00 80 00
MID_1		179	B3					
SignalType_1		1	1					
PPID208_1		0	0					
PID194_1		1	1					
SignalNumber_1		250	FA					
MID_2		179	B3					
SignalType_2		0	0					
PPID208_2		0	0					
PID194_2		0	0					
SignalNumber_2		128	80					
1.130707	CAN 1	19FFFFF5x	Info_Response_1	CAN Frame	Rx	8	8	FA B9 01 00 60 8C 00 00
Response01		113146	1B9FA					
Response02		35936	8C60					

Fonte: Autoria própria (2024).

A versão de *software* aqui testada também se mostrou mais robusta e capaz de manter a ciclicidade da mensagem de resposta em aproximadamente 1000ms. Ainda comprovou-se a capacidade dos componentes de requisitar mais de um sinal ao mesmo tempo sem perdas significativas de performance.

A requisição de dados via *PID 194* se mostrou um desafio nessa etapa. No protocolo *J1587* o sinal *PID 194* entrega os códigos de falha das unidades eletrônicas. Entretanto, o sinal só é enviado quando a falha faz uma transição entre os estados ativo e inativo. Assim sendo, foi necessário requisitar a informação da *ECU*, em vez de apenas monitorar o barramento, como tinha sido feito até então para os demais sinais.

A estrutura do *PID 194* também é única em relação a outros *PIDs*. Considerando o campo de dados da resposta a requisição recebemos 0xFA 0xB9 0x01. Segue a interpretação do sinal:

1. O primeiro *byte*, 0xFA, se refere ao numero do *PID* ou *SID* recebido. Nesse caso é o 250. Como requisitado.

2. O segundo *byte*, 0xB9, é o *diagnostic code character* ou DCC. 0xB9 equivale em binário a 10111001b.

    - Bit 8 (1b): Contagem de falha inclusa. Neste caso indica que a contagem está presente.
    - Bit 7 (0b): Estado atual da falha. 0b indica que a falha está ativa.
    - Bit 6 (1b): Tipo do diagnostico. 1b indica que é um diagnóstico padrão, já 0b indicaria que é um diagnostico estendido (*PIDs/SIDs* com valores maiores a 255).
    - Bit 5 (1b): Indica se a falha é um identificador de subsistema, SID (1b), ou um identificador de parametro, PID (0b).
    - Bits 4-1 (1001b): Indica o modo de falha. Nesse caso o valor 1001b indica que esta é uma falha de taxa de atualização do sinal (mais rápido ou mais lento que o normal).

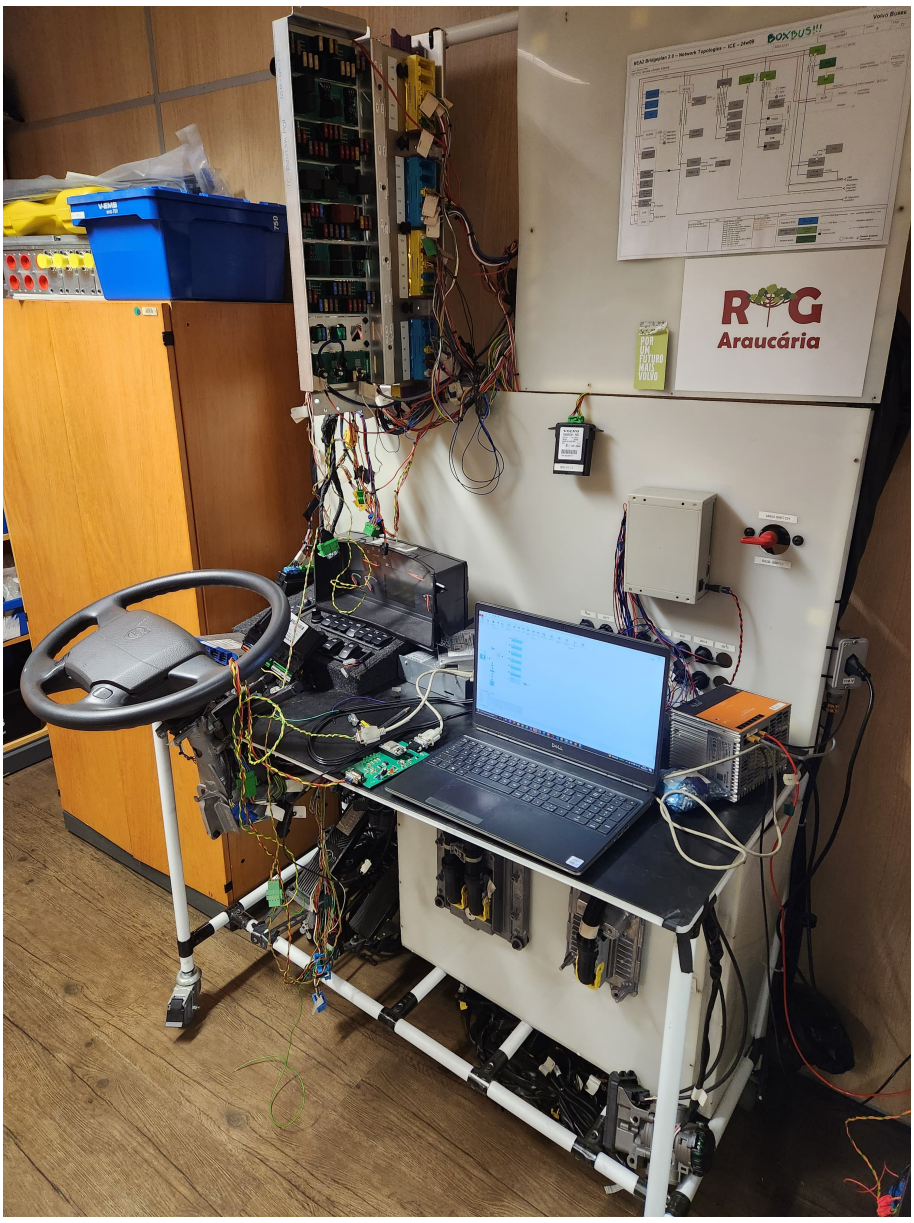
3. O terceiro *byte*, 0x01, indica a contagem de falha. Neste caso a falha ocorreu apenas uma vez.

Desta maneira, foram alcançados todos os objetivos dessa fase de testes. Foi considerado concluído o desenvolvimento do protótipo dentre do escopo almejado neste trabalho. Assim, o produto está pronto para ser testado em uma aplicação real.

4.3.3 Validação do protótipo em placa de circuito impresso conectado a uma bancada de veículo completo

Finalmente o protótipo foi levado à bancada de veículo completo (figura 73) onde foi possível verificar a robustez do sistema em barramentos de comunicação mais congestionados, assim como foi possível testar o sistema requisitando informações de mais unidades de controle. Uma bancada de veículo completo é uma representação da arquitetura eletrônica de um automóvel. Neste caso a bancada representa um ônibus.

Figura 73 – Verificação em bancada de veículo completo

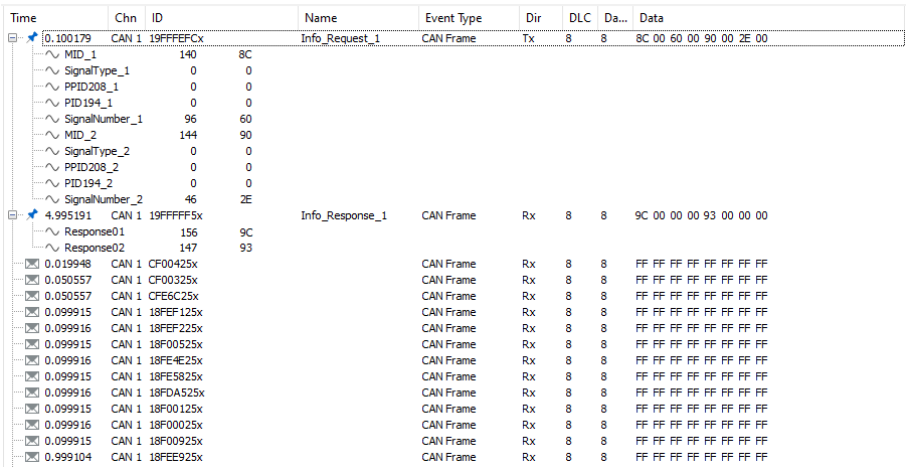


Fonte: Autoria própria (2024).

Foi observado que o protótipo seguiu funcional, mesmo conectado a mais unidades. Para exemplificar o resultado a figura demonstra o protótipo requisitando os dados de nível de combustível do painel de instrumentos (MID 140 PID96) e de pressão do tanque de ar da unidade de veículo (MID 150 PID 46).

Nessa imagem também pode-se visualizar o que era almejado com a proposta de projeto. Analisar tanto dados CAN quanto J1587 na mesma ferramenta e sincronizados. Também é importante mencionar que é necessário que o usuário tenha conhecimento do barramento J1587 para interpretar os dados. Neste exemplo, a resposta para o nível de combustível foi 196. Este dado é enviado pela unidade com um fator de 0,5. Logo o valor físico representado é 98%.

Figura 74 – Resultado do requisição de dados de múltiplas ECUs



Time	Chn	ID		Name	Event Type	Dir	DLC	Da...	Data
0.100179	CAN 1	19FFFEFCx		Info_Request_1	CAN Frame	Tx	8	8	8C 00 60 00 90 00 2E 00
MID_1		140	8C						
SignalType_1		0	0						
PPID208_1		0	0						
PID194_1		0	0						
SignalNumber_1		96	60						
MID_2		144	90						
SignalType_2		0	0						
PPID208_2		0	0						
PID194_2		0	0						
SignalNumber_2		46	2E						
4.995191	CAN 1	19FFFFF5x		Info_Response_1	CAN Frame	Rx	8	8	9C 00 00 00 93 00 00 00
Response01		156	9C						
Response02		147	93						
0.019948	CAN 1	CF00425x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.050557	CAN 1	CF00325x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.050557	CAN 1	CFE6C25x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.099915	CAN 1	18FEF125x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.099916	CAN 1	18FEF225x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.099915	CAN 1	18F00525x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.099916	CAN 1	18FE4E25x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.099915	CAN 1	18FE5825x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.099916	CAN 1	18FDA525x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.099915	CAN 1	18F00125x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.099916	CAN 1	18F00025x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.099915	CAN 1	18F00925x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF
0.999104	CAN 1	18FEE925x			CAN Frame	Rx	8	8	FF FF FF FF FF FF FF FF

Fonte: Autoria própria (2024).

5 CONCLUSÃO

As arquiteturas eletrônicas, conforme (RUSHTON; MERCHANT, 2000), definem as funções e os módulos de um veículo. A arquitetura dos sistemas eletroeletrônicos de um veículo define o número de módulos eletrônicos, as funções atribuídas a cada módulo e as interfaces elétricas entre estes módulos, em termos de:

- Relações e interconexões de subsistemas e componentes;
- Interfaces para outros sistemas;
- Características do meio em torno deste sistema;
- Fluxos de dados no sistema;
- Arquiteturas de *software* e dados.

Na última década os avanços da eletrônica embarcada implementada nos veículos cresceu exponencialmente. Várias funções eletrônicas foram implementadas nos veículos presentes no parque automotor, como navegação, controle adaptativo, controle de tração, controle de estabilização e sistemas de segurança ativa (NAVET, 2009). Hoje, até 2500 mensagens são trocadas por meio de até 70 Unidades de Controle Eletrônico (Natevet,2017).

O presente trabalho teve como principal objetivo criar uma ponte de ligação entre redes automotivas veiculares, disponibilizando meios para que uma rede automotiva veicular obsoleta possa ser convertida em uma rede automotiva veicular que é amplamente difundida e utilizada como padrão atualmente na indústria automobilística. A ideia principal do projeto foi requisitar as informações no protocolo obsoleto (J1587) e realizar a conversão para um protocolo utilizado como padrão na indústria automobilística (J1939). Essa conversão permite que uma rede automotiva veicular obsoleta seja interpretada através de um protocolo de comunicação padrão sem a necessidade de troca das ECU's de todos os veículos que ainda possuam a rede obsoleta.

Uma grande dificuldade do projeto foi encontrar o componente ELM325, que é responsável pela leitura e interpretação do protocolo J1587. O ELM325 é um componente que não é mais fabricado e não é possível comprar diretamente do fabricante. É necessário encontrar pessoas que tenham comprado o ELM325 e estejam dispostas a vender o componente. Isso poderia ser um agravante, caso seja decidido futuramente realizar a produção em larga escala do protótipo.

Um ponto de atenção é que cada um dos protocolos é pré-definido com valores que representam cada um dos sinais que podem ser transmitidos pelas ECU's. O presente projeto mapeou apenas os valores dos sinais enviados pelo *Fleet Management Gateway*. Para uma abordagem completa das redes automotivas, deve ser realizado um mapeamento para converter todos os sinais que são transmitidos em um barramento.

Os resultados do trabalho provaram que a abordagem é eficiente e que é possível converter com precisão o protocolo J1587 para o protocolo J1939. A premissa do projeto é de enorme importância, já que são poucas as pessoas com conhecimento e experiência para operar uma rede J1587. Com o protótipo, é possível padronizar a leitura de uma rede automotiva, independente se os veículos a serem requisitados estejam operando com o protocolo J1587.

REFERÊNCIAS

ALAM, M. S. U. *et al.* Securing vehicle ecu communications and stored data. *In*: **ICC 2019 - 2019 IEEE International Conference on Communications (ICC)**. [*S.l.*: *s.n.*], 2019. p. 1–6.

ANALOG. **Octal High-Voltage-Protected, Low-Power, Low-Noise Operational Amplifiers**. 2023. Disponível em: https://www.analog.com/en/products/max4805.html#product-overview.

AUTOPI. **ECU 101: The Ultimate Electronic Control Units Guide**. 2023. Disponível em: https://www.autopi.io/blog/what-is-electronic-control-unit-definition/.

BASIC, F. Vehicle network gateway (vng). Citeseer, 2004.

BLACKMAN, J.; MONROE, S. Overview of 3.3 v can (controller area network) transceivers. **Texas Instruments, Application Note. SLLA337**, 2013.

BOSCH, R. Can specification. **Robert Bosch GmbH, Postfach**, v. 50, 1991.

CENTRALCHIP. **Regulador de tensão linear LM7805**. 2023. Disponível em: https://www.centralchip.com.br/MLB-3030226792-50pc-lm7805-regulador-de-tenso-lm-7805-st-7805.

COOK, J.; FREUDENBERG, J. Controller area network (can). **EECS**, v. 461, p. 1–5, 2007.

DALMOLIN, R. **DISPOSITIVO DE AQUISIÇÃO E ARMAZENAMENTO DE MENSAGENS SAE J1708 NO FORMATO SAE J1587 PARA VEÍCULOS PESADOS**. Curitiba, 2022.

DAWOUD, D. S.; DAWOUD, P. **Serial communication protocols and standards**. [*S.l.*]: River Publishers, 2020.

EBERT, C.; JONES, C. Embedded software: Facts, figures, and future. **Computer**, IEEE Computer Society Press, Washington, DC, USA, v. 42, n. 4, p. 42–52, apr 2009. ISSN 0018-9162. Disponível em: https://doi.org/10.1109/MC.2009.118.

EISELE, A. O. A. H. **The Benefits of CAN for In-Vehicle Networking**. Hambach Castle, Germany, 2012.

ELMELECTRONICS. **ELM325 - Datasheet**. 2023. Disponível em: https://www.elmelectronics.com/DSheets/ELM325DSB.pdf.

ELMELECTRONICS. **ELM325 - J1708 Interpreter (v2.0)**. 2023. Disponível em: https://www.elmelectronics.com/obdic.html#ELM325.

FREDRIKSSON, L. **Distributed Embedded Control Systems in Robotics: Ten years of Development.** [*S.l.*], 1997.

GAUJAL, B.; NAVET, N. **Fault confinement mechanisms of the CAN protocol: Analysis and improvements**. 2002. Tese (Doutorado) — INRIA, 2002.

GUIMARÃES, A. de A. **ANÁLISE DA NORMA ISO11783 E SUA UTILIZAÇÃO NA IMPLEMENTAÇÃO DO BARRAMENTO DO IMPLEMENTO DE UM MONITOR DE SEMEADORA**. 2003. Dissertação (Mestrado) — Universidade de São Paulo, São Paulo, 2003.

GUIMARÃES, A. de A.; SARAIVA, A. M. **O Protocolo CAN: Entendendo e Implementando uma Rede de Comunicação Serial de Dados baseada no Barramento "Controller Area Network"**. [*S.l.*], 2002.

HPL, S. C. Introduction to the controller area network (can). **Application Report SLOA101**, Texas instruments, p. 1–17, 2002.

INSIDER, M. **What is an Electronic Control Unit?** 2020. Disponível em: https://www.aptiv.com/en/insights/article/what-is-an-electronic-control-unit.

KICAD. **KiCad EDA - A Cross Platform and Open Source Electronics Design Automation Suite**. 2023. Disponível em: https://www.kicad.org/about/kicad/.

KUMAR, S.; DALAL, S.; DIXIT, V. The osi model: Overview on the seven layers of computer networks. **International Journal of Computer Science and Information Technology Research**, v. 2, n. 3, p. 461–466, 2014.

MAKERHERO. **Módulo CAN BUS MCP2515 TJA1050**. 2023. Disponível em: https://www.makerhero.com/produto/modulo-can-bus-mcp2515-tja1050/.

NAVET, F. S.-L. N. **Automotive Embedded Systems Handbook**. CRC Press, 2009. ISBN 9780849380266. Disponível em: https://d1.amobbs.com/bbs_upload782111/files_38/ourdev_629261ASTZIF.pdf. Acesso em: 08 jan. 2024.

PAZUL, K. Controller area network (can) basics. **Microchip Technology Inc**, v. 1, 1999.

POWERS, C.; KIRSON, A.; ACTON, D. Today's electronics in today's vehicles. *In*: ENGINEERS, S. of A. (Ed.). **International Congress on Transportation Electronics**. [*S.l.*: *s.n.*], 1998.

RUSHTON, G.; MERCHANT, V. **Vehicle Electrical/Electronic System Design Considerations**. [*S.l.*]: SAE International, 2000.

SAASTAMOINEN, R. Analysis of the sae j1708 protocol. **Mälardalen University. Västerås, Sweden**, 2008.

SARAVATI. **Placa ESP32 WiFi / Bluetooth**. 2023. Disponível em: https://www.saravati.com.br/placa-esp32-wifi-bluetooth-devkit-v1-30-pinos.html.

SIMMASOFTWARE. **J1587 Introduction**. 2023. Disponível em: https://www.simmasoftware.com/j1587-introduction.pdf.

STEPPER, M. R. Data link overview for heavy duty vehicle applications. **SAE transactions**, JSTOR, p. 723–737, 1990.

STRAUSS, C.; CUGNASCA, C.; SARAIVA, A. **Protocolos de Comunicação para Equipamentos Agrícolas**. São Paulo, 1998.

THARAD, V. **Electronic Control Unit (ECU)**. 2019. Disponível em: https://www.linkedin.com/pulse/electronic-control-unit-ecu-vijay-tharad/?trackingId=7grABG4bTaK6PF6FUwYBPg%3D%3D.

TOLKIEN, J. R. R. **The Fellowship of the Ring**. UK: George Allen and Unwin, 1954.

TRUCK; COMMITTEE, B. L. S. C. N. *et al.* **Electronic Data Interchange Between Microcomputer Systems in Heavy-Duty Vehicle Applications**. [*S.l.*]: SAE International, 2008.

VECTOR. **Open Standard for Networking and Communication in the Commercial Vehicle Sector**. 2023. Disponível em: https://www.vector.com/int/en/know-how/protocols/sae-j1939/#.

VOSS, W. **A comprehensible guide to controller area network**. [*S.l.*]: Copperhill Media, 2008.

VOSS, W. **A comprehensible guide to J1939**. [*S.l.*]: Copperhill Media, 2008.

APÊNDICE A – Código desenvolvido

```
#include "Arduino.h"
#include <mcp_can.h>
#include <SPI.h>
#include <HardwareSerial.h>

//Hardware defines
#define RXD2 16 // Pinos RX e TX da comunicação serial com o ELM325
#define TXD2 17
#define ONBOARD_LED  2 // Led de feedback visual da placa.
#define INT_CAN0 15 // PIno de interrupção do MCP2515.
#define MCP2515_CS_PIN 5 // Pino CS do MCP2515

#define MAX_BYTES 4 // Maxímo número de bytes

enum gatewayStateEnum {WAITINGREQUEST, READING1587, SENDINGRESPONSE};
    // Estados do protótipo.
enum failureTypeEnum {PID, SID, PPID, PSID}; // Tipos de falha.

MCP_CAN CAN0(MCP2515_CS_PIN);
HardwareSerial SerialPort(2); // use UART2

hw_timer_t *Timer0_Cfg = NULL; // Timer 0

char rxData[100]={0};
char rxIndex=0;

long unsigned int rxId;
unsigned char len = 0;
unsigned char rxBuf[8];
long unsigned int MID1;
long unsigned int MID2;
failureTypeEnum SignalType1;
failureTypeEnum SignalType2;
long unsigned int SignalNumber1;
```

```
34 long unsigned int SignalNumber2;
35 unsigned int sendingTentative = 0;
36 int numBytes = 0;
37 int messageReceived = 0;
38 int j1587Received = 0;
39 int requestedInfo[4] = {0};
40 char *dataArray[8];
41
42 gatewayStateEnum gatewayState;
43
44 void setup() {
45
46   // Inicia a Seria connectada ao Computador
47   Serial.begin(9600);
48   while(!Serial);
49   Serial.println("\nUART0 Init");
50
51   // Inicia a serial conectada ao CI ELM325.
52   SerialPort.begin(57600, SERIAL_8N1, RXD2, TXD2);
53   while(!SerialPort);
54   Serial.println("UART2 Init");
55
56   // Configuração do mode de trabalho das portas do ESP32
57   pinMode(ONBOARD_LED, OUTPUT);
58   pinMode(INT_CAN0, INPUT);
59
60   // Inicializa o ELM325
61   ELMInit();
62
63   // Inicializa o MCP2515
64   MCP2515Init();
65
66   // Inicializa a maquina de estados.
67   gatewayState = WAITINGREQUEST;
```

```
68   Serial.println("State Waiting Request");
69 }
70
71 void loop() {
72
73   if( gatewayState == WAITINGREQUEST )
74   {
75     if(!digitalRead(INT_CAN0))  // Caso o pino 15 esteja baixo, lemos
       o buffer de chegada.
76     {
77       CAN0.readMsgBuf(&rxId, &len, rxBuf); // Leitura do Buffer: len
     = Numero de Bytes, buf = Buffer
78
79       Serial.print("CAN ID: ");
80       rxId = rxId & 0x1FFFFFFF;
81       Serial.println(rxId, HEX);
82       switch(rxId)
83       {
84         case 0x19FFFEFC:
85
86           if( rxBuf[0] >= 128 && rxBuf[0] <= 249)
87           {
88             MID1 = rxBuf[0];
89             requestedInfo[0] = rxBuf[1] & 0x3;
90             SignalNumber1 = rxBuf[3]*256 + rxBuf[2];
91             messageReceived = 1;
92
93             set1587Filter(MID1, SignalNumber1, 1);
94           }
95
96           if( rxBuf[4] >= 128 && rxBuf[4] <= 249)
97           {
98             MID2 = rxBuf[4];
99             requestedInfo[1] = rxBuf[5] & 0x3;
```

```
100             SignalNumber2 = rxBuf[7]*256 + rxBuf[6];
101             messageReceived = 1;
102 
103             set1587Filter(MID2, SignalNumber2, 0);
104           }
105 
106           break;
107         default:
108           messageReceived = 0;
109       }
110 
111       if( MID1 >= 128 && MID1 <= 249)
112       {
113         Serial.print("MID1: ");
114         Serial.print(MID1);
115         Serial.print(" -- SignalType: ");
116         Serial.print(SignalType1);
117         Serial.print(" -- SignalNumber: ");
118         Serial.println(SignalNumber1);
119       }
120 
121       if( MID2 >= 128 && MID2 <= 249)
122       {
123         Serial.print("MID2: ");
124         Serial.print(MID2);
125         Serial.print(" -- SignalType: ");
126         Serial.print(SignalType2);
127         Serial.print(" -- SignalNumber: ");
128         Serial.println(SignalNumber2);
129       }
130     }
131 
132     if(messageReceived){
133 
```

```
134       SerialPort.flush();
135       SerialPort.print("AT MA\r");
136       SerialPort.flush();
137
138       Serial.println("State Reading 1587");
139       gatewayState = READING1587;
140
141       messageReceived = 0;
142     }
143   }
144   else if( gatewayState == READING1587) {
145
146     j1587Received = ELMread();
147
148     if(j1587Received)
149     {
150       Serial.println("Terminou Leitura J1587");
151
152       SerialPort.flush();
153       SerialPort.print("\r");
154       SerialPort.flush();
155
156       Serial.println("State Sending CAN Response");
157       gatewayState = SENDINGRESPONSE;
158
159       sendingTentative = 0;
160       j1587Received = 0;
161     }
162   }
163   else if( gatewayState == SENDINGRESPONSE) {
164     byte resp = sendCAN();
165
166     if(resp == CAN_OK)
167     {
```

```
168       Serial.println("Communication Ok! Go back to waiting.");
169       gatewayState = WAITINGREQUEST;
170     }
171     else
172     {
173       sendingTentative++;
174     }
175
176     if(sendingTentative >= 10)
177     {
178       Serial.println("Failed. Go Back to waiting.");
179       gatewayState = WAITINGREQUEST;
180     }
181
182   }
183   else {
184     gatewayState = WAITINGREQUEST;
185   }
186
187 }
188
189 // Inicializa o modulo ELM325
190 void ELMInit(void) {
191
192   SerialPort.flush();
193   SerialPort.print("ATZ\r");
194   SerialPort.flush();
195   delay(2000);
196
197   Serial.println("ELM325 Init");
198 }
199
200 // Inicializa o modulo MCP2515
201 void MCP2515Init(void) {
```

```
202   if(CAN0.begin(MCP_STDEXT, CAN_250KBPS, MCP_8MHZ) == CAN_OK)
203     Serial.print("MCP2515 Init Okay!!\r\n");
204   else
205     Serial.print("MCP2515 Init Failed!!\r\n");
206
207   CAN0.init_Mask(0,1,0x1FFFFFFF);                // Init first mask
        ...
208   CAN0.init_Filt(0,1,0x19FFFEFC);                // Init first filter
        ...
209
210   CAN0.init_Mask(1,1,0x1FFFFFFF);                // Init second mask
        ...
211   CAN0.init_Filt(2,1,0x1FFFFFFF);                // Init third filter
        ...
212
213
214   CAN0.setMode(MCP_NORMAL);
215 }
216
217 int ELMread(void)      //Faz a leitura de dados da rede e armazena no
      array de dados.
218 {
219   char tmpData[100]={0};
220   rxIndex=0; //Resetando o contador para a prox. leitura.
221
222   char c;
223
224   do{
225     if(SerialPort.available() > 0)
226     {
227       c = SerialPort.read();
228
229       if((c!='Z') && (c!='M') && (c!='\n')) //Mantem isto fora do
      array.
```

```
230       {
231         tmpData[rxIndex] = c; //Adiciona as informações recebidas no
     array.
232         rxIndex = rxIndex + 1;
233       }
234
235     }
236   }while(c != '\r' && c != '<'  && c != 'L'  && c != 'T'); //O CI
     ELM325 termina a sua resposta com esse char, usamos para sair.
237
238   if( rxIndex > 9 )
239   {
240     char* d = strtok(tmpData, " ");
241
242     numBytes = (rxIndex-1)/3 - 2;
243     if( numBytes > MAX_BYTES )
244       numBytes = MAX_BYTES;
245
246     char *tmparray[50];
247     int i = 0;
248
249     while (d != NULL) {
250         tmparray[i++] = strdup(d);
251         d = strtok(NULL, " ");
252     }
253
254     if(strcmp(tmparray[0], "F1") && strcmp(tmparray[0], "F2") &&
     numBytes > 0)
255     {
256       int dataNumber = 0;
257
258       Serial.print("MID: ");
259       Serial.print(tmparray[0]);
260       Serial.print(" PID: ");
```

```
261       Serial.print(tmparray[1]);
262       Serial.print(" Data: ");
263 
264       if( tmparray[0] == MID1 && tmparray[1] == SignalNumber1 )
265       {
266         dataNumber = 0;
267       }
268       else
269       {
270         dataNumber = 1;
271       }
272 
273       for(int n = 0; n < numBytes; n++)
274       {
275         Serial.print(tmparray[n+2]);
276         dataArray[dataNumber*4 + n] = strdup(tmparray[n+2]);
277         Serial.print(" ");
278       }
279       Serial.println("");
280 
281       return 1;
282     }
283 
284   }
285   //Serial.println(tmpData);
286   return 0;
287 }
288 
289 byte sendCAN(void)
290 {
291   byte data[8] = {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00};
292 
293   for(int i = 0; i < 4; i++)
294   {
```

```
295     if
296     data[i] = strtoul(dataArray[i], NULL, 16);
297   }
298
299   byte sndStat = CAN0.sendMsgBuf(0x19FFFFF5, 1, 8, data);
300   if(sndStat == CAN_OK){
301     Serial.println("Message Sent Successfully!");
302   } else {
303     Serial.println("Error Sending Message...");
304   }
305
306   return sndStat;
307 }
308
309 void set1587Filter(long unsigned int fMID, long unsigned int
      fSignalNumber, bool filter1)
310 {
311   SerialPort.flush();
312   if(filter1)
313   {
314     SerialPort.print("AT F1 ");
315   }
316   else
317   {
318     SerialPort.print("AT F2 ");
319   }
320   SerialPort.print(fMID, HEX);
321   SerialPort.print(" ");
322   SerialPort.print(fSignalNumber, HEX);
323   SerialPort.print(" XX XX XX\r");
324   SerialPort.flush();
325
326   delay(2000);
327 }
```