Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Projeto de um tradutor de código para estudo de redes automotivas em sistemas embarcados

Lucas Avelino de Lima Jacinto

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. Evandro Leonardo Silva Teixeira

Brasília
2018

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Projeto de um tradutor de código para estudo de redes automotivas em sistemas embarcados

Lucas Avelino de Lima Jacinto

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. Evandro Leonardo Silva Teixeira (Orientador)
FGA/UnB

Prof. Dr. Ricardo Pezzuol Jacobi
Universidade de Brasília

Prof. Dr. Ricardo Zelenovsky
Universidade de Brasília

Prof. Dr. José Edil Guimarães de Medeiros
Coordenador do Curso de Engenharia da Computação

Brasília, 28 de novembro de 2018

Dedicatória

Dedico este trabalho a minha família, em especial minha mãe, Joselene, meu pai, Almir, meus irmãos, Leonardo e Luis Gustavo e a minha namorada e amor da minha vida, Letícia.

Agradecimentos

Agradeço primeiramente a Deus, por ter me concedido saúde, família, amigos e por ter me dado muito mais do que eu preciso.

Agradeço aos meus pais, Almir de L. Jacinto e Joselene V. A. de L. Jacinto, por me criarem com tanto amor, por me apoiarem em todas as etapas da minha vida e por terem se sacrificado por toda a vida para me dar o melhor. Agradeço também pelos conselhos e ensinamentos que me fizeram chegar até aqui e que vou levar para a vida toda.

Agradeço aos meus irmãos Leonardo A. de L. Jacinto e Luis Gustavo A. de L. Jacinto, por terem me incentivado e pelas conversas que me renderam novas ideias e me ajudaram muito na produção deste trabalho.

Agradeço a minha namorada, Letícia R. Miranda, por ter me ajudado de tantas formas, seja através do carinho, apoio moral, conselhos, ou com as risadas que me proporcionou. Por me entender todas as vezes em que não pude sair para poder estudar. Agradeço por ter sido minha companheira, por me fazer tão feliz e por me amar tão intensamente.

Agradeço ao meu orientador, Evandro L. S. Teixeira, por todo o suporte e dedicação dada a este trabalho, por estar sempre disposto a me esclarecer qualquer dúvida e pela confiança depositada em mim.

Resumo

O projeto de redes de comunicação automotivas representa uma etapa importante no desenvolvimento de um automóvel nos dias de hoje. As ferramentas de desenvolvimento e simulação facilitam a prototipação e a análise inicial do desempenho da rede. O custo elevado de plataformas de hardware tem dificultado a verificação do funcionamento da rede pelos estudantes de engenharia. Neste sentido, o presente trabalho apresenta uma solução na forma de um tradutor que realiza a conversão do código gerado por meio de uma ferramenta de simulação de redes virtuais, para redes embarcadas em plataformas acessíveis e de baixo custo. A solução desenvolvida foi validada através de 3 estudos de caso onde redes com diferentes níveis de complexidade puderam ser testadas. Por fim, os resultados de cada estudo de caso são apresentados e apesar das limitações de recursos da plataforma, o estudo de caso 3 comprova o desempenho da solução em uma rede muito próxima de uma real.

Palavras-chave: Sistemas Embarcados, Sistemas Automotivos, Redes Automotivas

Abstract

The design of automotive communication networks represents an important step in the development of an automobile these days. Development and simulation tools facilitate prototyping and initial analysis of network performance. The high cost of hardware platforms has made it difficult to verify the operation of the network by engineering students. In this sense, the present work presents a solution in the form of a translator that performs the conversion of the code generated by means of a simulation tool from virtual networks, to embedded networks, in accessible and low cost platforms. The solution developed was validated through 3 case studies where networks with different levels of complexity could be tested. Finally, the results of each case study are presented and despite the limitations of platform resources, case study 3 proves the performance of the solution in a network very close to a real one.

Keywords: Embedded Systems, Automotive Systems, Automotive Networks

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A evolução tecnológica dos automóveis tem impulsionado o aumento do número de componentes eletroeletrônicos presentes em um veículo[1]. Estes sistemas têm substituído gradualmente os que antes eram puramente mecânicos e hidráulicos e permitiram o surgimento de novas funcionalidades como funções de conforto e segurança. Nos dias de hoje, carros modernos podem ter mais de 50 unidades de controle eletrônico (ECU, *Electronic Control Unit*) e carros de luxo chegam a ter mais de 100[2].

Para atender a demanda do mercado automotivo por novas funcionalidades cada vez mais complexas foi necessário que os sistemas automotivos migrassem da arquitetura centralizada para a distribuída, permitindo a troca de dados e o compartilhamento de ações de controle entre diferentes subsistemas.

Com o elevado número de componentes eletrônicos, tornou-se inviável conexão ponto-a-ponto entre os mesmos. Para simplificar a troca de informações entre os diferentes subsistemas e reduzir o número de fios necessários para interconectá-los, a indústria automotiva passou a utilizar redes de comunicação multiplexada com um único barramento compartilhado.

O projeto dessas redes de comunicação intra-veicular modernas pode ser uma tarefa bastante complexa. Para garantir que um modelo de uma rede automotiva tenha o comportamento esperado, na prática, é necessário que antes exista uma etapa de simulação utilizada para avaliar o comportamento da rede e corrigir eventuais problemas de projeto em sua arquitetura.

O uso de ferramentas de simulação de redes automotivas como CANoe da Vector e o Busmaster da *Robert Bosch Engineering and Business Solutions Limited* (RBEI) e ETAS GmbH permitem que o projetista da rede possa avaliar os canais de comunicação, bem como o comportamento da rede em situações específicas e a seu desempenho, facilitando o correto dimensionamento da rede.

1.1 Definição do Problema

O elevado custo de plataformas de *hardware* para simulação de redes automotivas torna esta etapa, tão fundamental do processo de desenvolvimento de um automóvel, pouco acessível a estudantes de engenharia.

Entretanto, existe atualmente, uma quantidade expressiva de microcontroladores e kits de desenvolvimento para sistemas embarcados mais acessíveis e que podem ser utilizados, com limitações, para o estudo de redes automotivas em plataforma de hardware, dos quais, os microcontroladores da plataforma arduino são um exemplo.

O problema abordado neste trabalho consiste da falta de ferramentas de simulação de redes automotivas em hardware embarcado que se utilizem das plataformas mais acessíveis a estudantes de engenharia.

1.2 Objetivo

O objetivo deste trabalho é propor e contribuir, através do desenvolvimento de um tradutor de código, para uma plataforma distribuída para projeto e simulação de redes automotivas em sistemas embarcados.

Este trabalho tem como objetivos específicos:

- Implementar funcionalidades semelhantes às encontradas em softwares de simulação de redes automotivas virtuais.
- Permitir simulação de uma rede CAN aderente a norma SAE J1939, amplamente utilizado pela indústria automotiva para comunicação intra-veicular.
- Permitir que o usuário da solução possa simular e testar redes CAN de complexidade arbitrária.
- Permitir interação com o ambiente externo, com acesso às interfaces analógicas, digitais e serial de um microcontrolador, de forma a simular aquisição de dados vindos de sensores presentes no automóvel.
- Possibilitar o monitoramento da rede de forma a permitir uma análise e comparação do desempenho com uma rede virtual.

1.3 Estrutura do Trabalho

Este trabalho está organizado em 5 capítulos. O Capítulo 2 apresenta os principais conceitos abordados. Nele é exposta uma introdução a redes automotivas e aos protocolos

que são abordados neste trabalho, bem como uma visão inicial a respeito de softwares de simulação de redes automotivas e de sistemas embarcados. O Capítulo 3 descreve os passos principais para o desenvolvimento da solução de software. No Capítulo 4 há uma descrição de cada um dos estudos de caso deste trabalho e uma análise dos resultados. Por fim, o Capítulo 5 apresenta a conclusão deste trabalho.

1.4 Metodologia

A metodologia utilizada no desenvolvimento deste trabalho é apresentada pela Figura 1.1.

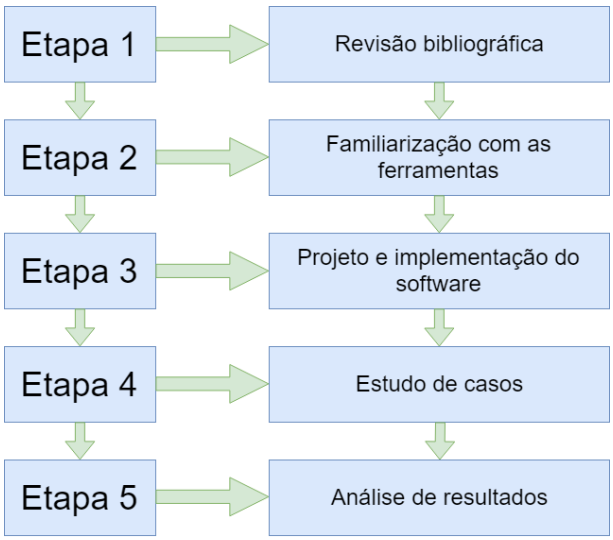


Figura 1.1: Etapas do projeto.

1.4.1 Revisão bibliográfica

Na primeira etapa, foram revisados todos os principais conceitos abordados no trabalho como redes automotivas, características das redes CAN, padrão SAE J1939, arquitetura de software automotivo, estudo de diferentes opções para implementação da solução final como escolha da plataforma de hardware, escolha do software de simulação, entre outros.

1.4.2 Familiarização com as ferramentas

A segunda etapa consistiu do aprendizado das ferramentas adotadas no projeto como o software de simulação de redes automotivas *Busmaster* e o sistema operacional de tempo real *trampoline RTOS*.

1.4.3 Projeto e implementação do software

Nesta etapa os requisitos do software foram traçados, foi realizado um projeto da arquitetura do software e a implementação.

Foram estudadas as diferentes bibliotecas de software utilizadas para auxiliar o desenvolvimento do projeto como as bibliotecas *boost* utilizadas e o framework de interface gráfica *Qt*.

1.4.4 Estudos de casos

Nesta etapa foram selecionados 3 estudos de caso para testar o funcionamento do software. Os estudos de caso foram implementados em ambiente virtual, passaram cada um pela transformação através do software desenvolvido no trabalho e foram testados e avaliados em ambiente embarcado.

1.4.5 Análise dos Resultados

Nesta etapa foi realizada a análise final dos resultados e elaborada a conclusão do projeto.

Capítulo 2

Revisão Bibliográfica

Este capítulo apresenta os principais conceitos que foram abordados ao longo do trabalho como redes de comunicação automotiva, tecnologias de barramento intra veicular, sistemas embarcados, software de simulação de redes, dentre outros.

2.1 Evolução da arquitetura eletroeletrônica

Da década de 1970 para os dias atuais a tecnologia presente nos automóveis evoluiu de sistemas puramente hidráulicos e mecânicos para sistemas eletrônicos e distribuídos.

Os primeiros dispositivos eletrônicos incorporados nos automóveis possuíam uma arquitetura centralizada. Entretanto, aos poucos, mais dispositivos foram substituídos por versões eletrônicas e a complexidade da arquitetura cresceu significativamente.

De forma a simplificar o projeto de cada subsistema, a arquitetura padrão da indústria automotiva migrou da centralizada para a distribuída e as redes de comunicação entre dispositivos chegaram a ter até mais de 150 ECUs [2].

No início da eletrônica veicular embarcada, as funcionalidades eram implementadas como sistemas autônomos, sem nenhuma integração com as outras partes do sistema. O aumento da complexidade das funções fornecida pelos automóveis gerou a necessidade de uma rede de comunicação intra-veicular eficiente para troca de informações e para o compartilhamento de ações de controle entre os diferentes sistemas [1].

Um exemplo de funcionalidade que faz uso da rede de comunicação é o sistema de controle de tração (TCS, *Traction Control System*), que atua buscando reduzir o deslizamento dos pneus quando o veículo está em aceleração. Para isso, ele utiliza a informação de velocidade das rodas para detectar quando uma roda está perdendo tração. Quando detecta que uma roda está mais veloz do que as demais, o TCS ativa o freio nesta roda, de forma a reduzir sua velocidade e aumentar a sua tração. O TCS de alguns automóveis pode, além disso, reduzir a potência do motor para as rodas que estão deslizando. Para

execução desse controle, o sistema necessita portanto, da comunicação com as ECUs, de gerenciamento do motor (EMS, *Engine Management System*) e do sistema de frenagem anti-bloqueio (ABS, *Anti-lock Braking System*) [3].

Outro exemplo de sistema com ação de controle distribuída é o sistema de controle de cruzeiro adaptativo (ACC, Adaptative Cruise Control). Este sistema presente em alguns automóveis modernos utiliza um sensor de radar para capturar informações de distância, velocidade relativa e posição relativa do veículo à frente. O ACC permite que o automóvel mantenha uma distância segura do veículo a frente, através de uma lógica de controle sobre os valores obtidos pelo sensor e do controle da aceleração e desaceleração do veículo por meio da interação com a EMS, para influenciar o torque do motor, com sistema de transmissão para influenciar na transmissão e com a TCS para agir no torque dos freios[4].

A princípio os dados de diferentes ECUs eram trocados por meio de enlaces ponto-a-ponto, entretanto esse modelo logo se mostrou ineficiente para suportar o crescente número de informações que deveriam ser comunicados entre diversos sistemas para criar as novas funcionalidades em um automóvel. A indústria automotiva partiu então para o uso de redes de comunicação onde as ECUs estão mutuamente conectadas em um canal multiplexado [1].

A Figura 2.1 apresenta o crescimento na quantidade de funções, o número de ECUs presentes em um veículo nas últimas décadas e a previsão de crescimento para o futuro. Percebe-se que muitas das novas funcionalidades se tornam possíveis apenas através da interconexão de diversos sistemas, o que indica um aumento na complexidade das redes de comunicação entre os dispositivos eletrônicos.

O uso de redes de comunicação automotivas é muito vantajoso se comparado ao método de conexões ponto-a-ponto. Um fator importante é a utilização de um barramento ao qual diferentes dispositivos eletrônicos podem se conectar para realizar a comunicação, reduzindo assim, a quantidade de fios necessários para interligar os sistemas. Além disso, o compartilhamento de informações também evita que haja redundância de sensores no veículo.

De acordo com as necessidades de velocidade de comunicação das redes automotivas, diferentes protocolos são selecionados. A Tabela 2.1 apresenta a classificação proposta pela *Society of Automotive Engineers* (SAE) que caracteriza as redes automotivas de acordo com as taxas de transmissão:

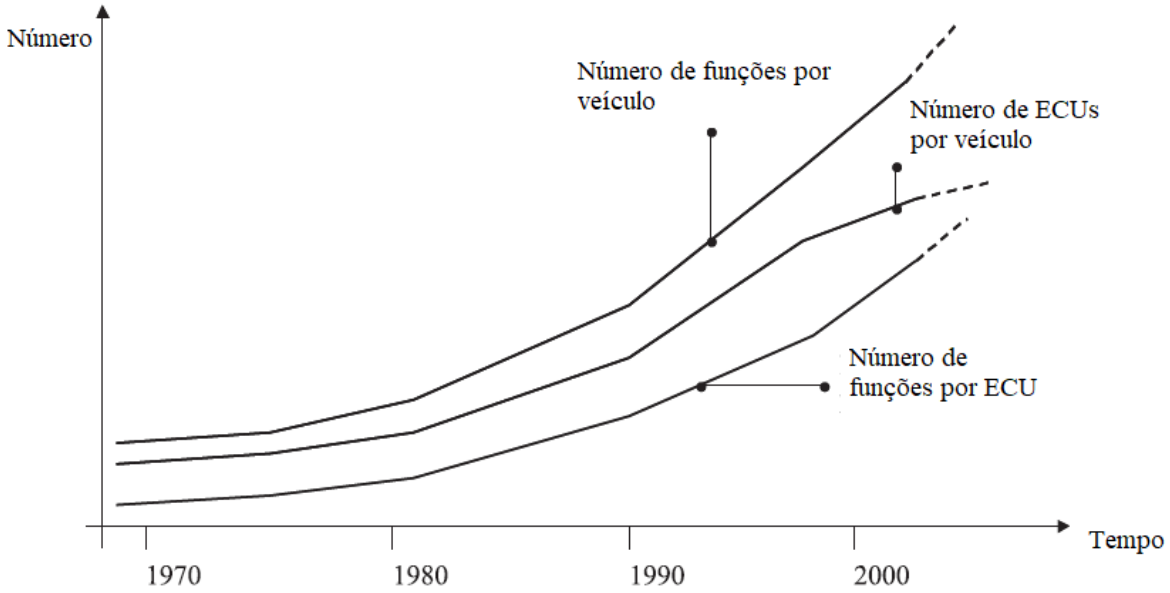


Figura 2.1: Número de funções e ECUs por veículo (Fonte: [4]).

Tabela 2.1: Classificação SAE para redes automotivas.

Classe	Taxa de transmissão	Aplicações	Protocolos
A	Até 10Kb/s	Controle de iluminação, limpador de para-brisas, portas, etc.	LIN
B	10Kb/s a 125Kb/s	Aplicações secundárias como painel de instrumentos, etc.	CAN
C	125Kb/s a 1Mb/s	Controle de ignição, direção, freios, motor, etc.	TTCAN, CAN
D	Acima de 1Mb/s	Funções de entretenimento como internet, TV digital, etc.	Flex-Ray, MOST

2.2 Protocolo CAN

O protocolo *Controller Area Network* (CAN) foi desenvolvido pela Robert Bosch GmbH e lançado em 1986. É um protocolo de comunicação serial síncrono de tempo real e com um alto nível de segurança para integração de ECUs em uma rede de comunicação veicular.

As principais características do protocolo CAN incluem[5]:

- Priorização de mensagens;
- Latência determinística;
- Topologia flexível;

- Recepção de mensagens de múltiplas fontes com sincronização temporal;
- Operação com múltiplos mestres;
- Mecanismos de detecção de erros e sinalização;
- Retransmissão automática de mensagens;
- Outros mais.

Uma rede CAN típica consiste de vários nós interligados por um barramento, onde cada nó representa uma ECU. A arquitetura de um nó no barramento CAN consiste de um controlador e um transceptor CAN. O primeiro implementa uma máquina de estados para execução do protocolo e o segundo implementa a interface física, isto é, transmissão e recepção dos *frames* no meio físico.

O modelo *Open Systems Interconnection* (OSI) define 7 camadas para as pilhas de protocolos de redes de comunicação. São elas: aplicação, apresentação, sessão, transporte, rede, enlace e física. Dentre essas camadas, o protocolo CAN trata apenas das duas camadas mais abaixo na pilha, camada de enlace e camada física, como pode ser observado na Figura 2.2. Essas duas camadas tratam do controle do acesso ao meio, bem como controle do enlace lógico e do meio físico no qual os dados irão trafegar. A implementação das camadas acima fica a cargo do desenvolvedor de acordo com as necessidades da aplicação.

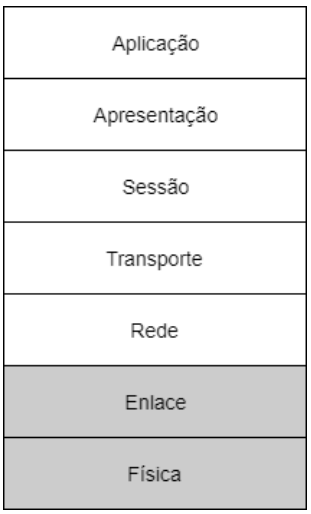


Figura 2.2: Modelo de Pilha de Protocolos OSI. As camadas em cinza são tratadas pelo protocolo CAN.

2.2.1 Codificação dos bits

Os bits no protocolo CAN são codificados segundo o esquema *non-return to zero* (NRZ), no qual o nível lógico permanece constante durante toda a duração de cada um dos bits que trafegam no barramento[5].

Devido ao esquema de codificação adotado pelo protocolo, quando muitos bits de mesmo nível lógico são transmitidos sequencialmente, o valor do estado do sinal no barramento pode ser interpretado erroneamente por outros nós na rede como um problema. Para contornar isso, o protocolo implementa um método de tratamento de erro denominado *bit stuffing*. Após cinco bits consecutivos de mesmo valor, o transmissor envia intencionalmente um bit com o valor oposto para indicar que não há anomalias na transmissão. O receptor deve ser capaz de detectar e remover o bit invertido[5].

2.2.2 Arbitragem

O protocolo CAN utiliza o mecanismo de arbitragem *Carrier Sense Multiple Access with Collision Avoidance* (CSMA/CA) para resolver conflitos no barramento[5]. Este mecanismo conta com a utilização de dois tipos de bits no barramento, dominante e recessivo. De forma que, quando os dois tipos de bits são transmitidos simultaneamente, o bit dominante sobrescreve eletricamente o bit recessivo no barramento.

Quando um nó detecta que não possui o controle do barramento através da leitura de um bit dominante após o envio de um bit recessivo no barramento, este imediatamente interrompe a transmissão e passa para o modo *listen only*. Ele só irá transmitir uma mensagem novamente quando receber uma sinalização de fim da transmissão (*end of frame*).

Assim, o protocolo prioriza mensagens através da disposição dos bits dominantes no campo de arbitragem das mensagens. Na Figura 2.3 observamos que, apesar de existirem 3 nós tentando simultaneamente transmitir uma mensagem no barramento, apenas o nó 3 prossegue com a transmissão, pelo fato dos nós 1 e 2 perceberem que em um dado bit do campo de arbitragem, o nível lógico lido do barramento divergiu do que foi transmitido pelo nó e imediatamente suspenderem a transmissão.

2.2.3 Mensagem no protocolo CAN

A especificação do protocolo CAN publicada em 1991 se subdivide em duas partes, o formato de quadro padrão com o campo identificador de 11 bits, CAN 2.0 A e o formato estendido com o identificador de 29 bits, CAN 2.0 B.

Os quadros no protocolo CAN 2.0 A são classificados em 5 categorias:

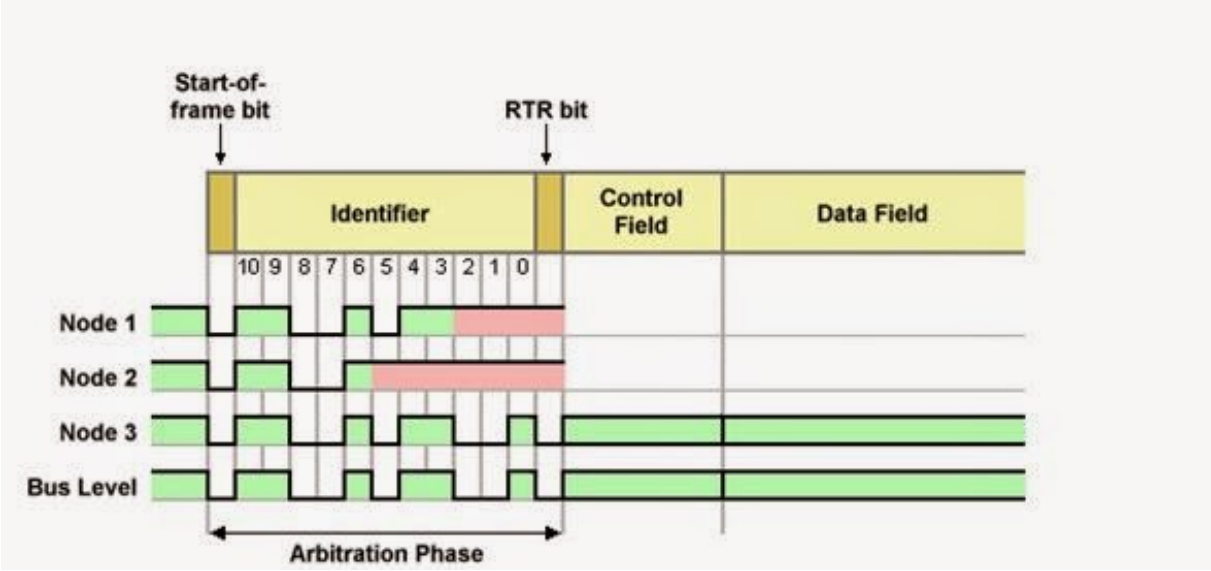


Figura 2.3: Exemplo de arbitragem no protocolo CAN 2.0 A (Fonte: [6]).

Quadros de dados encapsulam e transportam os dados dos transmissores aos receptores.

Quadros remotos são utilizados para realizar uma requisição de um quadro de dados.

Quadros de erros são transmitidos para notificar os nós que um erro foi detectado.

Quadros de sobrecarga servem para indicar que um nó está sobrecarregado no momento e atrasar o envio do próximo quadro de dados ou remoto.

Quadros inter-quadros são utilizados para separar temporalmente os quadros de dados e remotos dos seus anteriores.

2.2.3.1 Quadro de dados

O formato do quadro de dados no protocolo CAN 2.0 A pode ser observado na Figura 2.4. A estrutura desse quadro é formada por [5]:

***Start of Frame* (SOF)** consiste de um bit dominante após um certo período no qual o barramento estava ocioso. Indica o início da transmissão de um quadro.

Campo de arbitragem consiste de um identificador de 11 bits e um bit *Remote Transmission Request* (RTR). Os bits do campo identificador são transmitidos do mais significativo para o menos e os 7 bits mais significativos não podem ser todos recessivos. O bit RTR deve ser dominante em um quadro de dados.

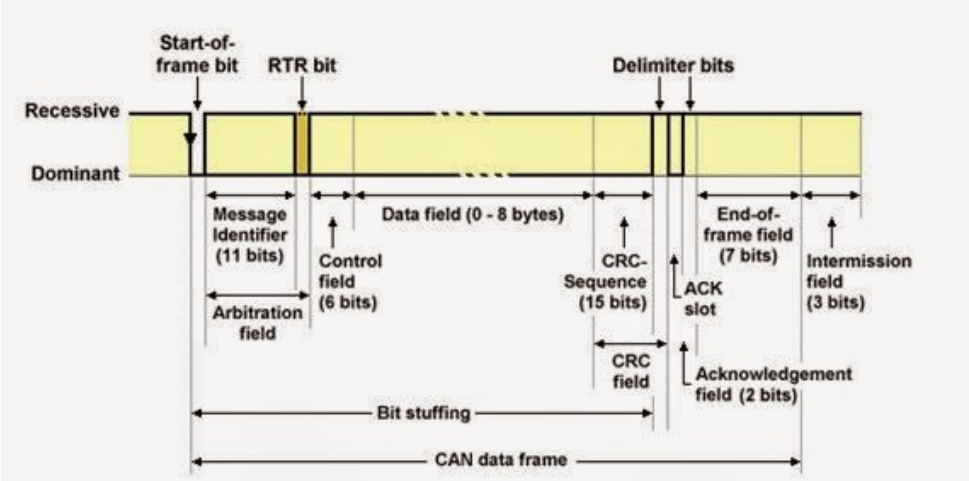


Figura 2.4: Quadro de dados no protocolo CAN 2.0 A (Fonte: [6]).

Campo de controle consiste de um campo de 2 bits reservados para uso futuro e um campo de 4 bits que informa a quantidade de bytes no campo de dados, *Data Length Code* (DLC).

Campo de dados é onde se encontram os dados da mensagem. Transmitidos do bit mais significativo para o menos significativo.

Campo *Cyclic Redundancy Check* (CRC) formado pelos campos: sequência CRC e delimitador CRC. O primeiro serve para verificar a validade da mensagem, enquanto o segundo consiste de um único bit recessivo.

Campo *Acknowledge* (ACK) consiste de dois campos de um bit cada. O primeiro é um bit de resposta dos receptores, no qual transmissor envia um bit recessivo no barramento e todos os nós no barramento que receberam a mensagem e verificaram com sucesso a validade da mensagem respondem com um bit dominante. E o segundo é um bit delimitador recessivo.

***End of Frame* (EOF)** formado por 7 bits recessivos. Indica o fim transmissão de um quadro.

2.2.3.2 Quadro Remoto

O quadro remoto tem estrutura similar ao quadro de dados, entretanto se difere principalmente em dois aspectos: o bit RTR é recessivo, o que serve para indicar uma requisição e o campo de dados é inexistente independente do valor no campo DLC.

2.2.3.3 Quadro de Erro

O quadro de erro no protocolo CAN 2.0 A é utilizado para notificar outros nós da detecção de um erro no barramento.

A estrutura deste quadro consiste de um campo de *error flag* e um campo delimitador. O campo de *error flag* é formado pela superposição de diferentes *error flags* fornecidas por diversos nós no barramento. As *error flags* podem ser passivas, caracterizadas pelo envio de 6 bits recessivos consecutivos no barramento ou ativas, com o envio de 6 bits dominantes consecutivos.

O campo delimitador é formado por 8 bits recessivos que são enviados após o envio das *error flags*. Os diversos nós monitoram o barramento à espera de uma transição de um bit dominante para um bit recessivo e a partir desse momento enviam o campo delimitador de erro.

2.2.3.4 Quadro de Sobrecarga

O quadro de sobrecarga é utilizado para indicar que um nó está sendo sobrecarregado por um determinado período e atrasar o envio do próximo quadro de dados ou quadro remoto. Formado por dois campos: *overload flag* que consiste em 6 bits dominantes consecutivos e um delimitador do campo de sobrecarga que consiste de 8 bits recessivos consecutivos.

2.2.3.5 Quadro entre quadros

O quadro entre quadros ou, espaçamento entre quadros, tem o propósito de separar a transmissão de quadros de dados ou remotos dos quadros que os precedem.

Formado pelos campos:

Campo de intermissão consiste na transmissão de 3 bits recessivos, onde nenhum nó tem permissão para iniciar o envio de um quadro de dados ou remoto.

Campo de barramento ocioso no qual o barramento está livre e qualquer nó pode acessar o barramento para transmitir um quadro. O período deste campo pode ser escolhido arbitrariamente.

Campo de suspensão de transmissão que consiste de 8 bits recessivos enviados por um nó em estado de erro passivo após a transmissão de uma mensagem, seguindo o campo de intermissão.

2.2.4 CAN 2.0 B

A especificação CAN 2.0 B, também conhecida como formato estendido, tem o propósito de inserir no protocolo suporte para ampliar os identificadores para 29 bits. Esta extensão possibilita um aumento considerável do espaço de endereçamento de mensagens distintas. Entretanto, também provoca um aumento na sobrecarga relacionada ao envio dos quadros, com o aumento do cabeçalho das mensagens.

A Figura 2.5 compara os formatos de quadros de dados nas especificações CAN 2.0 A e CAN 2.0 B. Como pode ser observado na imagem, os 11 bits do campo identificador da mensagem permanecem sem alteração entre as duas versões do protocolo, o bit de RTR foi movido para ficar após os 18 bits finais do identificador. Além disso, a versão 2.0B faz uso dos bits reservados do campo de controle e do bit que antes era o RTR para adicionar os bits *Substitute Remote Request* (SRR) e *Identifier Extension* (IDE). No formato estendido, ambos os bits são recessivos, o primeiro tem a função clara de dar prioridade a quadros de dados no formato padrão e o segundo indica que o quadro está no formato estendido.

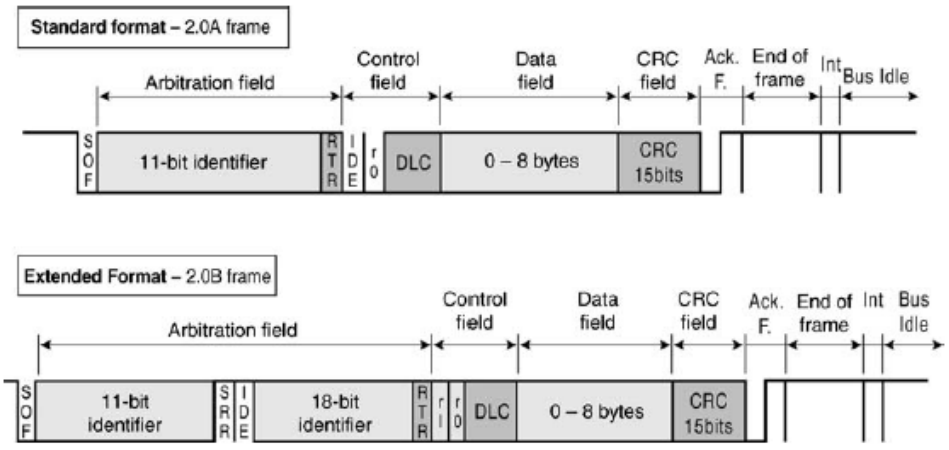


Figura 2.5: Diferença entre o quadro de dados nas especificações CAN 2.0 A e CAN 2.0 B (Fonte: [5]).

2.2.5 O problema da implementação do protocolo CAN

O protocolo CAN define um conjunto de regras que devem ser seguidas para a comunicação entre os nós da rede. Contudo, levanta uma série de questões a respeito da implementação de uma aplicação específica, que não são respondidas pelo próprio protocolo, tais como:

- Como selecionar os endereços de cada nó da rede?

- Como enviar mensagens com mais de 8 bytes de dados, caso necessário?
- Como enviar mensagens com um destinatário específico ou informar o endereço do remetente, caso necessário?
- Como dar significado aos bytes de dados transmitidos?

Estas questões e diversas outras são respondidas por protocolos que estão na camada de aplicação do modelo OSI/ISO. Desta forma, o protocolo CAN pode ser largamente utilizado por um amplo conjunto de aplicações.

Muitas normas abertas e proprietárias foram desenvolvidas para uso do protocolo CAN em aplicações específicas. Algumas delas são[5]:

- CANopen, produzido pela *CAN in Automation* (CiA).
- Device Net, produzida por Allen Bradley.
- SDS (*Smart Distributed System*), produzida pela Honeywell.
- OSEK (*Open systems and interfaces for distributed eletronics in cars*), produzido pela *European car industries group*.
- CANKingdom, produzido pela Kvaser.
- SAE J1939, produzido pela *Society of Automotive Engineers in the USA* (SAE).

Quanto à utilização do protocolo CAN pela indústria automotiva, no entanto, muitas fabricantes optam por implementar sua própria norma.

2.3 Padrão SAE J1939

O comitê de controle de ônibus e caminhões e comunicações da *Society of Automotive Engineers* (SAE) desenvolveu a família de especificações J1939 para uso na rede interna de veículos que se baseia no protocolo CAN 2.0 B.

A norma SAE J1939 implementa as camadas acima do CAN, na pilha de protocolos OSI, trazendo algumas funcionalidades extras para a rede como uso de mensagens de tamanho de até 1785 bytes e o uso de identificadores para os nós [7].

Algumas características gerais da norma incluem [7]:

- Uso do formato estendido do protocolo CAN (2.0B).
- Taxa de transmissão padrão de 250 Kbits/s.

- Máximo de 30 ECUs na rede.
- Máximo de 253 aplicações de controle, onde múltiplas aplicações podem ser gerenciadas por um única ECU.
- Tamanho máximo de mensagem de até 1785 bytes de dados.
- Define o conceito de grupo de parâmetros.
- Comunicação *peer-to-peer* e *broadcast*.
- Suporta mecanismos de gerenciamento da rede.

Um Grupo de parâmetros na norma SAE J1939 é um conjunto de parâmetros que pertence a um mesmo tópico e que compartilha uma mesma taxa de transmissão.

Os pacotes de dados J1939 contém os dados padrão de uma mensagem junto com um cabeçalho que contém o *Parameter Group Name* (PGN). O PGN identifica a função da mensagem e os dados relacionados, isto é, quais tipos de informações a mensagem transporta.

O formato do ID segundo a norma SAE J1939 está ilustrado na Figura 2.6. Percebe-se que o campo ID de 29 bits do protocolo CAN 2.0 B, na norma SAE J1939, é constituído de:

- 3 bits de prioridade.
- 18 bits de PGN, formado pelos campos:
  - 1 bit de página de dados estendida (*extended data page*).
  - 1 bit de página de dados (*data page*).
  - 8 bits de formato do PDU (*PDU format*).
  - 8 bits de dados específico do PDU (*PDU specific*).
- 8 bits de endereço do remetente (*source address*).

O campo de formato PDU, caso possua valor menor que 240 indica PGNs específicos, ou seja, enviados para dispositivos específicos (*peer-to-peer*). Caso contrário, indica que a mensagem foi enviada para todos os dispositivos na rede (*broadcast*) [8].

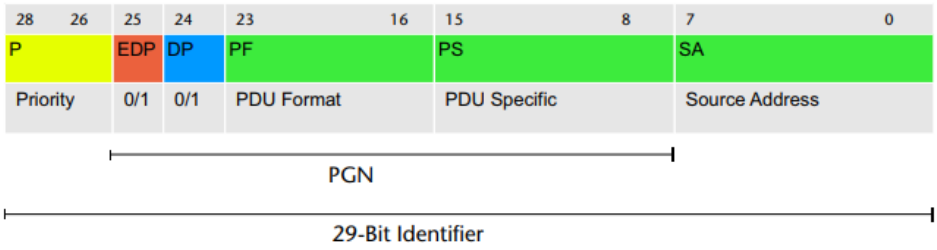


Figura 2.6: Formato do ID na norma SAE J1939 (Fonte: [9]).

2.4 Software para Projeto de Redes Automotivas

Um importante passo para validação de uma rede automotiva ocorre através da sua simulação em ambiente virtual e controlado, e o monitoramento e análise do comportamento da rede nessas condições.

Na indústria automobilística existem diversas ferramentas de simulação de redes automotivas disponíveis, como CANoe da Vector e o Busmaster da *Robert Bosch Engineering and Business Solutions Limited* (RBEI) e ETAS GmbH. O Busmaster tem como aspectos positivos relevantes para uso em trabalhos acadêmicos, além de outras qualidades, o fato de ser uma ferramenta gratuita e de software livre.

2.4.0.1 CANoe

O CANoe é uma ferramenta de software proprietário da empresa Vector, amplamente utilizada por empresas em todo o mundo, que abrange o desenvolvimento, teste e análise de redes de ECUs e de cada ECU individualmente [10]. A Figura 2.7 apresenta a interface do software CANoe.

Dentre as suas principais características, destacam-se:

- Permite a integração de todas as tarefas de desenvolvimento e teste por meio de uma só ferramenta.
- Testes fáceis de automatizar.
- Avaliação de resultados baseada em gráficos e texto de amigáveis ao usuário.
- Permite o uso de diversos protocolos e normas automotivas empregadas na indústria como CAN, LIN, FlexRay, SAE J1939, GPS, entre outros.
- Permite detectar e corrigir erros ainda no processo de planejamento da rede.

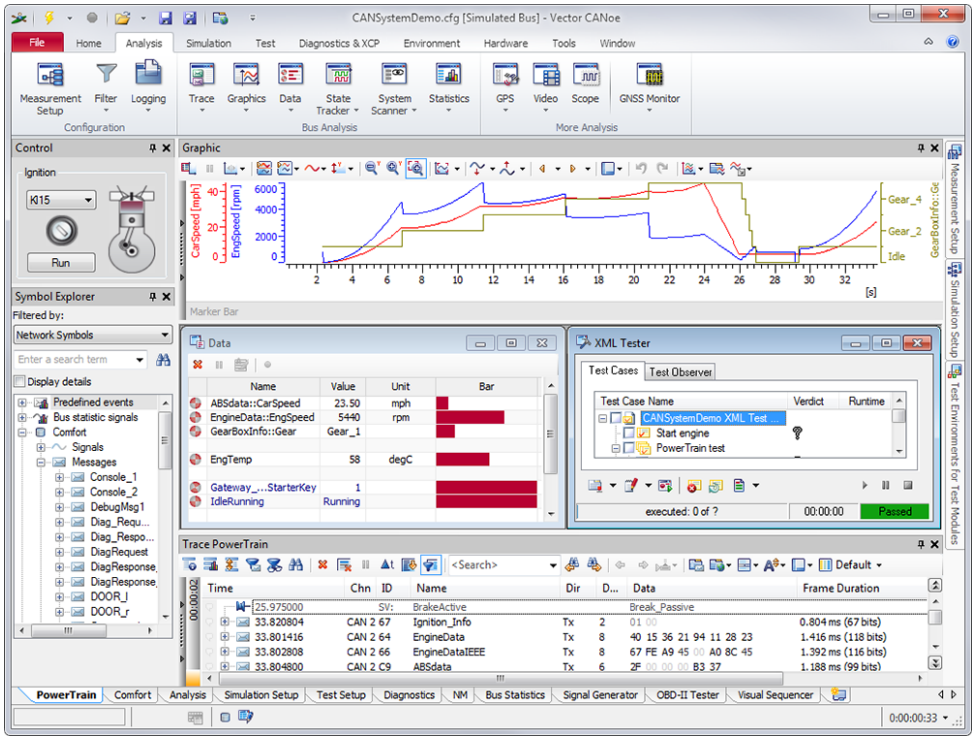


Figura 2.7: Interface do software CANoe da Vector (Fonte: [11]).

2.4.0.2 Busmaster

O Busmaster é uma ferramenta acessível e de interface fácil de usar para desenvolvimento de redes CAN que executa em ambiente Windows. Permite o projeto, o monitoramento, a simulação e a análise de mensagens em um barramento CAN.

O usuário do Busmaster pode programar as diferentes funcionalidades para os nós de uma rede CAN com qualquer nível de complexidade, através das interfaces providas pelo software.

A aplicação fornece um editor de banco de dados de mensagens, tela de monitoramento dos pacotes no barramento com diferentes opções de visualização, podendo escolher entre observar os bytes sem formatação ou no formato lógico definido. A Figura 2.8 apresenta a interface do software Busmaster.

Dentre suas principais características estão:

- Suporte para redes CAN 2.0 A e CAN 2.0 B, Rede LIN, Rede FlexRay, CAN FD.
- Uso de portas USB do computador para conectar múltiplos hardwares em redes CAN.
- Suporte ao padrão SAE J1939.

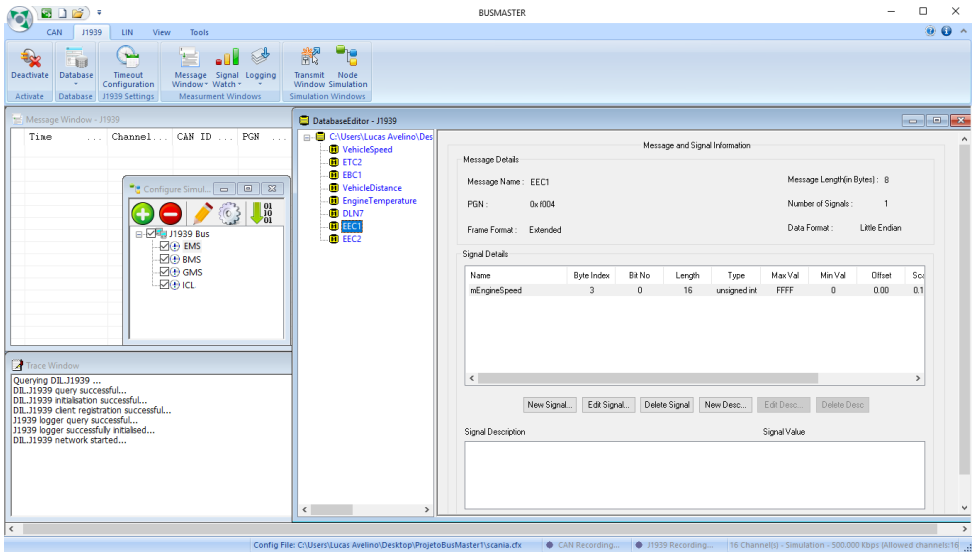


Figura 2.8: Interface do software Busmaster da RBEI e ETAS.

- Análise das estatísticas da rede.
- Tela de visualização de mensagens no barramento.
- Registro de mensagens e possibilidade de repetir a execução de mensagens registradas.

O software conta com suporte a algumas plataformas de hardware embarcado para simulação da rede. Entretanto, as opções de plataformas de *hardware* são bastante limitadas e são em sua maioria de alto custo, o que contribui para restringir sua utilização em ambiente acadêmico.

2.5 Sistemas Embarcados Automotivos

Sistemas embarcados são sistemas computadorizados que são caracterizados por possuírem restrições de recursos disponíveis e projetados para finalidades específicas. Geralmente são compostos por um microcontrolador e dispositivos de entrada e saída analógicos e/ou digitais.

Os sistemas embarcados presentes em um automóvel implementam uma série de funcionalidades com diferentes características e restrições. Algumas dessas funções são de domínio crítico de segurança como controle de ignição, transmissão, suspensão, direção e frenagem e necessitam de garantias de confiabilidade no sistema.

Para tentar atender a essas necessidades dos sistemas embarcados automotivos modernos a indústria automotiva investe no desenvolvimento de novos paradigmas de pro-

jeto, redes de comunicação e camadas de *middleware* em busca de tornar os sistemas confiáveis[12].

2.5.1 Sistemas Operacionais de Tempo Real

Os sistemas operacionais de tempo real (RTOS) se distinguem dos sistemas operacionais de propósito geral por fornecerem mecanismos aos usuários que têm como função possibilitar que as aplicações possam alcançar além da corretude lógica, a corretude temporal. Isto é, garantir o cumprimento das restrições temporais (*deadlines*).

Os RTOSs podem ser classificados em duas categorias quanto ao comprometimento em atender aos *deadlines*: sistemas de *hard real-time* e sistemas de *soft real-time*. No livro [13], o autor define que uma unidade de trabalho (*job*) tem restrições do tipo *hard real-time* se o usuário requer a validação de que o sistema sempre atende a restrição temporal, sendo a validação feita através da demonstração por um procedimento comprovadamente correto e eficiente ou através da realização de exaustiva simulação e testes. O autor ainda define restrições do tipo *soft real-time*, como sendo quando nenhuma validação é necessária, ou quando apenas uma demonstração de que o *job* atende a uma restrição estatística é suficiente.

Sistemas operacionais de tempo real são responsáveis por fornecer abstrações como tarefas concorrentes, mecanismos de comunicação e sincronização entre tarefas, dentre outros serviços e por fornecer garantias temporais nos serviços oferecidos.

2.5.2 ISO 17356: Veículos Rodoviários - Interface aberta para aplicações automotivas embarcadas

A ISO 1736 é um padrão que define o *Offene Systeme und deren Schnittstellen für die Elektronik in Kraftfahrzeugen* (OSEK), composto pelos documentos:

- ***ISO 17356-1:2005 Road vehicles – Open interface for embedded automotive applications – Part 1: General structure and terms, definitions and abbreviated terms*** define a estrutura geral da ISO.
- ***ISO 17356-2:2005 Road vehicles – Open interface for embedded automotive applications – Part 2: OSEK/VDX specifications for binding OS, COM and NM*** define especificações para vinculação do OS, COM e NM.
- ***ISO 17356-3:2005 Road vehicles – Open interface for embedded automotive applications – Part 3: OSEK/VDX Operating System (OS)*** define um ambiente uniforme que suporta utilização eficiente de recursos para software de aplicação de unidades de controle automotivo.

- ***ISO 17356-4:2005 Road vehicles – Open interface for embedded automotive applications – Part 4: OSEK/VDX Communication (COM)*** define interfaces padrões e protocolos para troca de dados em uma rede de comunicação.
- ***ISO 17356-5:2006 Road vehicles – Open interface for embedded automotive applications – Part 5: OSEK/VDX Network Management (NM)*** define a funcionalidade padrão para garantir o funcionamento adequado das redes automotivas.
- ***ISO 17356-6:2006 Road vehicles – Open interface for embedded automotive applications – Part 6: OSEK/VDX Implementation Language (OIL)*** define uma linguagem utilizada para realização de configurações do sistema e descrição de objetos para as implementações de OS e COM.

2.5.2.1 OSEK/VDX OS

O padrão OSEK/VDX OS define um sistema operacional de tempo real que foi projetado com base nos princípios de escalabilidade, portabilidade do software, configurabilidade e alocação estática de recursos do sistema operacional. Suporta aplicações com restrições de *hard real-time* e fornece um conjunto serviços para dar suporte a aplicação, tais como:

- Gerenciamento de tarefas (*tasks*) como, ativação e terminação de tarefas, gerenciamento do estado das tarefas e troca de tarefas.
- Sincronização por meio de duas funcionalidades
  - Recursos, que funcionam como *locks* para garantir a exclusão mútua entre tarefas na execução de uma seção crítica.
  - Controle de eventos, que funciona como variável de condição, permitindo a espera e notificação de tarefas no acontecimento de um evento.
- Gerenciamento de interrupções por meio de serviços para processamento em interrupções.
- Alarmes com tempo relativo ou absoluto.
- Serviços para troca de dados por meio de mensagens intra processador.
- Tratamento de erros por meio de mecanismos de suporte ao usuário em diferentes casos de erros.

O sistema operacional OSEK/VDX fornece dois conceitos diferentes de tarefas:

- ***Basic Task*** pode assumir 3 estados, estes são ilustrados pela Figura 2.9 em conjunto com as transições entre cada um deles e descritos aqui:
  - ***running*** é o estado assumido quando a tarefa está em execução. Apenas uma única tarefa está em execução em qualquer ponto no tempo.
  - ***ready*** é estado assumido quando a tarefa foi ativada e espera ser posta em execução pelo escalonador ou quando sofreu preempção e aguarda ser escalonada novamente para execução.
  - ***suspended*** é o estado assumido quando a tarefa foi terminada e esta permanece neste estado até que seja ativada novamente.
- ***Extended Task*** se difere da *basic task* por poder assumir um estado a mais, o estado *waiting*, que possibilita a tarefa a realizar a espera por um evento. Pode assumir 4 estados, ilustrados na Figura 2.10, dos quais, os estados *running*, *ready* e *suspended* funcionam da mesma forma que na *basic task* e o estado *waiting* é o estado assumido quando a tarefa espera por um evento, através de um serviço do sistema operacional, e fica inapta a continuar a sua execução enquanto o evento não ocorrer. Acontecido o evento, a tarefa é posta novamente no estado *ready*.

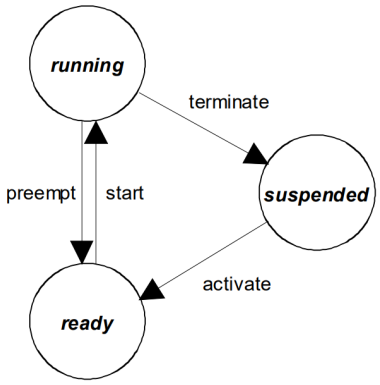


Figura 2.9: Máquina de estados de uma *Basic Task* (Fonte: [14]).

O escalonador decide qual tarefa será transferida para o estado *running* em um dado ponto no tempo com base em um algorítimo de escalonamento de prioridades.

As prioridades das tarefas no OSEK/VDX OS são definidas de forma estática, no momento de concepção da aplicação, e o escalonador não suporta gerenciamento dinâmico de prioridades. O escalonador, no entanto, faz uso do protocolo chamado *priority ceiling* para o gerenciamento de recursos, que pode temporariamente alterar a prioridade de uma tarefa que detém um determinado recurso para evitar problemas de inversão de prioridade

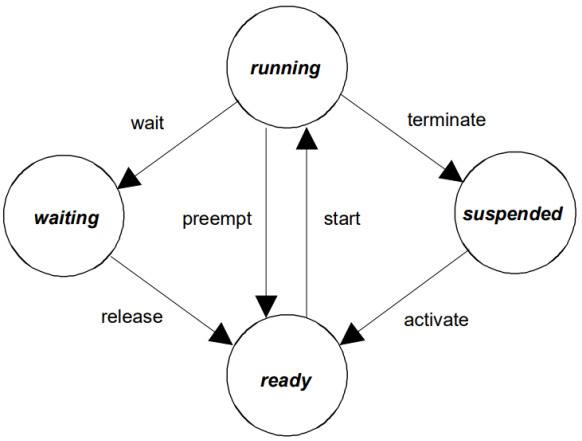


Figura 2.10: Máquina de estados de uma *Extended Task* (Fonte: [14]).

e *deadlock*. Contudo, a alteração de prioridades é definida na concepção da aplicação e não pode ser mudada ao longo da execução.

O algorítimo de escalonamento seleciona para execução, de todas as tarefas nos estados *running* e *ready*, a mais antiga do conjunto de tarefas com a mais alta prioridade.

O escalonador também suporta duas políticas de escalonamento de tarefas:

- **Escalonamento preemptivo completo (*full preemptive scheduling*)** permite que uma tarefa seja interrompida para a execução do escalonamento de uma nova tarefa durante qualquer instrução na ocorrência de uma condição de ativação do escalonador, por exemplo, quando uma tarefa de mais alta prioridade é movida para o estado *ready*.

- **Escalonamento não-preemptivo (*non preemptive scheduling*)** apenas permite que uma tarefa seja interrompida para o escalonamento de uma nova tarefa quando a tarefa em execução requisita explicitamente o escalonamento através de um serviço de sistema. Mecanismo chamado de escalonamento cooperativo.

2.5.2.2 OSEK/VDX Implementation Language (OIL)

A linguagem OIL permite a configuração do sistema operacional por meio da descrição, orientada a objetos dos recursos utilizados. Todos os objetos de sistemas especificados pelo padrão OSEK e seus relacionamentos são descritos através do OIL.

OIL define tipos padrões para os objetos. Cada objeto é descrito por um conjunto de atributos e referências. OIL define o conjunto de atributos padrões para cada objeto.

Os arquivos de configuração em linguagem OIL contêm duas partes:

- **Definição da implementação** para configuração de recursos padrões e de específicos da implementação.
- **Definição da aplicação** para a configuração da estrutura da aplicação localizada em uma CPU em particular.

A Figura 2.11 ilustra o processo de desenvolvimento de uma aplicação utilizando o padrão OSEK. A primeira etapa consiste no desenvolvimento da aplicação em linguagem C, utilizando os serviços do padrão OSEK. Nesta etapa também é necessário realizar a configuração da aplicação por meio da criação de arquivos em linguagem OIL, que pode ser feita através de um software ou manualmente pelo desenvolvedor. Na etapa seguinte utiliza-se um programa para gerar o código fonte em linguagem C, com as configurações do sistema operacional, a partir dos arquivos em linguagem OIL. Em seguida, os arquivos de código fonte produzidos, em conjunto com os arquivos de código da aplicação são compilados, o que resulta em arquivos objeto. Por fim, através do ligador (*linker*), é gerado um código executável a partir dos arquivos objeto da aplicação e dos arquivos objeto do Kernel e serviços do OSEK.

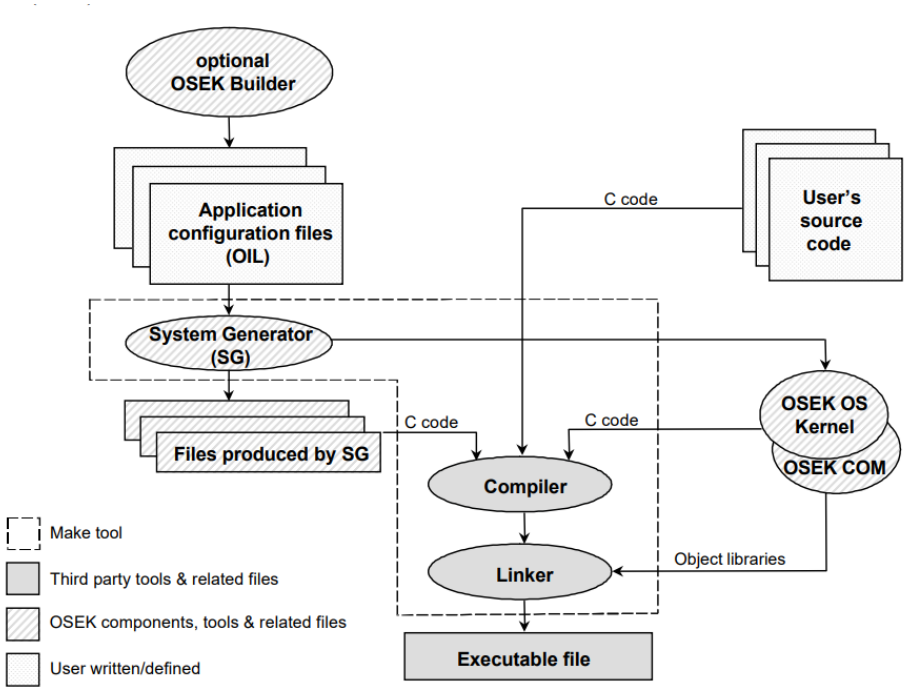


Figura 2.11: Processo de desenvolvimento para aplicações no padrão OSEK (Fonte: [15]).

Um exemplo de configuração de objetos utilizados na definição da aplicação por meio da linguagem OIL é a configuração de tarefas, onde são definidos os atributos de prioridade, tipo de escalonamento, tipo de ativação, referência a recursos (*locks*) utilizados pela tarefa, eventos a qual a tarefa pode reagir e mensagens acessadas pela tarefa.

Além disso, no contexto de definição da aplicação, é possível configurar serviços de alarmes que podem, por exemplo, ativar tarefas quando atingirem uma contagem determinada. Pode-se também declarar recursos e eventos utilizados na aplicação, configurar rotinas de tratamento de interrupção, dentre outras configurações.

Capítulo 3

Projeto do Software

Este capítulo aborda diversos aspectos que foram tratados no desenvolvimento da solução de software para realizar a simulação de redes automotivas em plataformas embarcadas.

3.1 Principais Requisitos do Software

Dentre os principais requisitos do software desenvolvido neste trabalho destacam-se:

Permitir desenvolvimento da lógica de funcionamento dos nós em um barramento para que seja possível simular diferentes redes automotivas com complexidade arbitrária e compostas por nós com funções diversas.

Executar em uma plataforma embarcada de baixo custo facilitando o acesso em ambiente acadêmico.

Possuir uma interface simples e amigável possibilitando ser utilizada com poucos passos e sem necessidade de treinamento.

Permitir comparação dos resultados com redes virtuais para avaliar o desempenho das redes automotivas nas plataformas embarcadas.

O software que será exportado para o microcontrolador deve atender a restrições de tempo real para permitir uma simulação próxima da realidade, onde algumas das funções exercidas pelas unidades de controle eletrônico possuem de fato restrições de tempo real.

3.2 Plataforma de Desenvolvimento

A utilização da plataforma de desenvolvimento proposta neste trabalho consiste de 3 passos. O primeiro é a criação e análise da rede automotiva virtual através do uso de uma

ferramenta de simulação. O segundo passo consiste na utilização de um assistente, desenvolvido neste trabalho, para configurar o processo de geração de código para plataforma embarcada. Por fim, o último passo consiste da análise de resultados da rede embarcada e comparação com a rede virtual. Esses passos estão representados pela Figura 3.1.

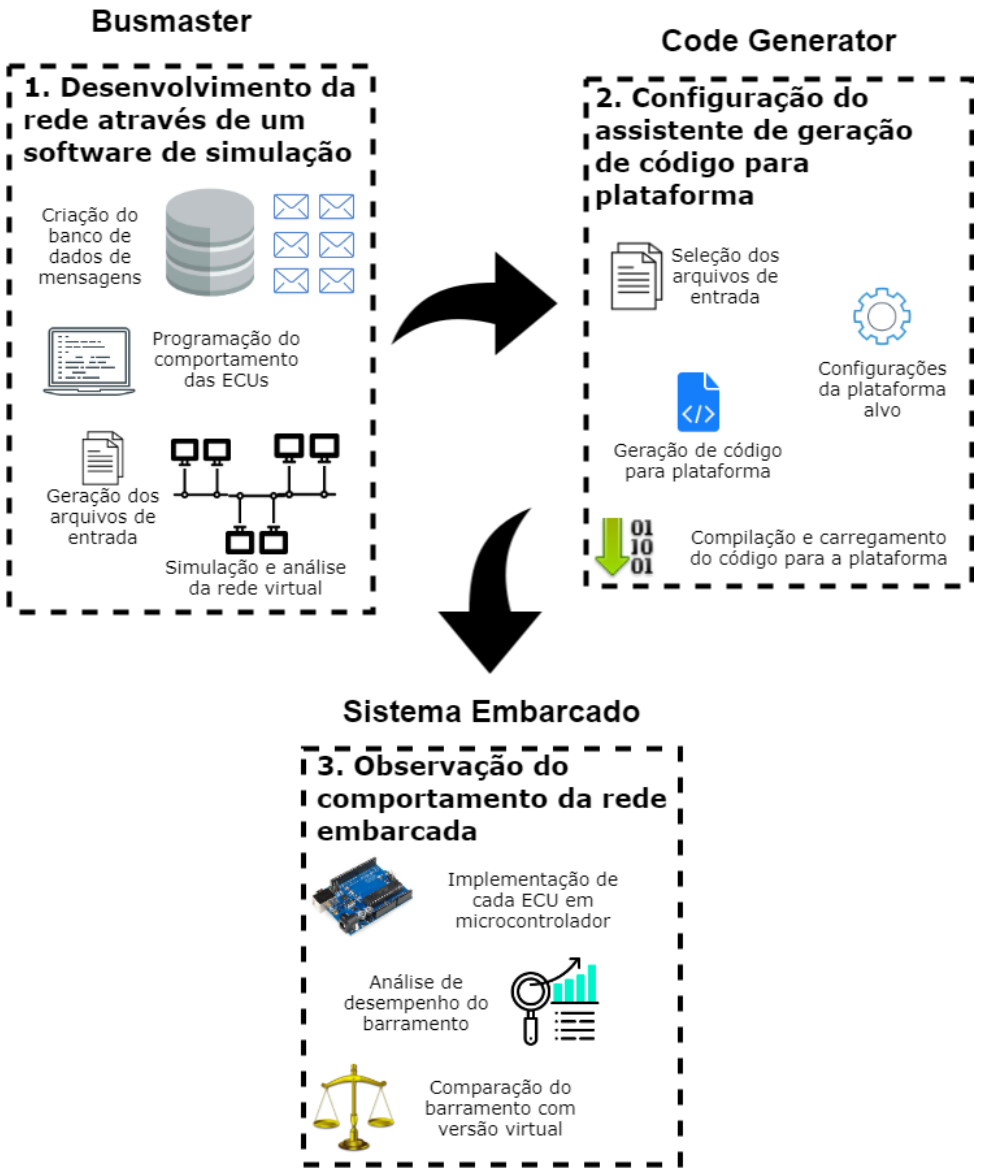


Figura 3.1: Processo de desenvolvimento da rede automotiva embarcada através da plataforma de desenvolvimento .

3.2.1 Software para projeto e análise da rede automotiva

A plataforma de desenvolvimento conta com o uso de uma ferramenta de projeto e análise de redes automotivas, confiável, gratuita, de software livre e que possui uma interface

simples e amigável ao desenvolvedor. A ferramenta selecionada foi o Busmaster.

A Figura 3.2 apresenta os passos para desenvolvimento de uma rede automotiva CAN SAE J1939.

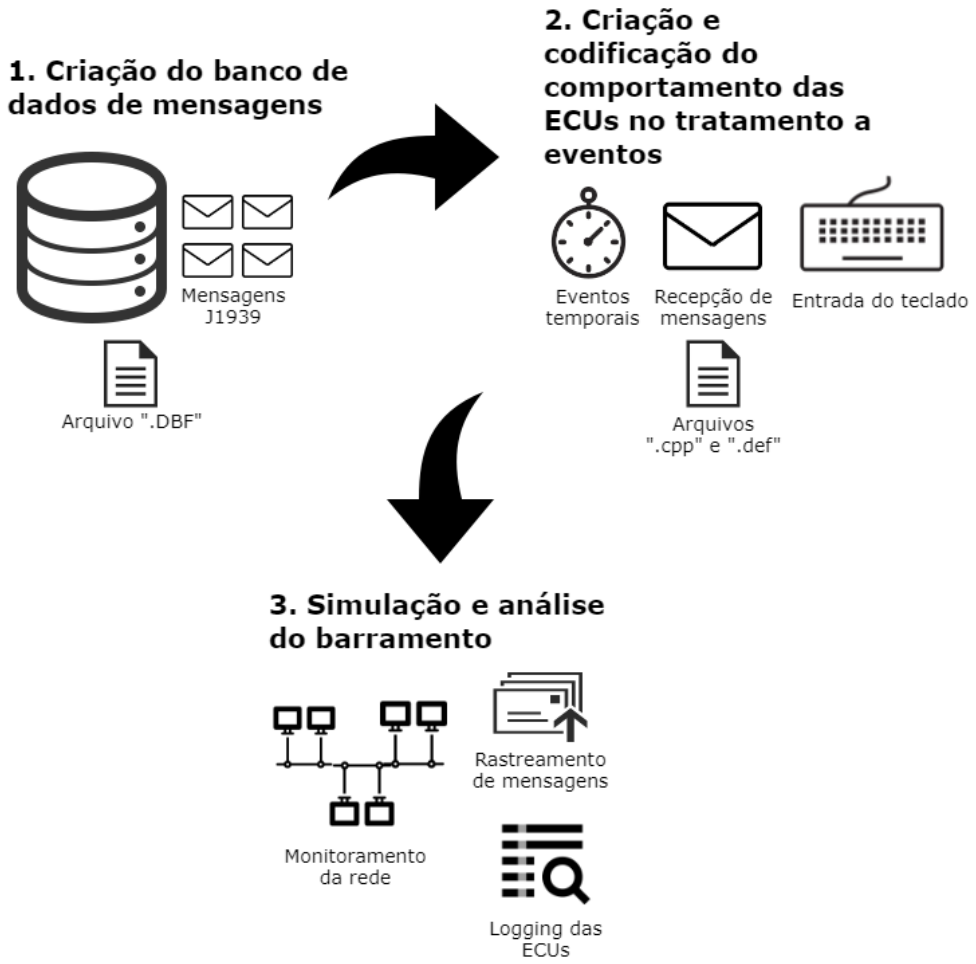


Figura 3.2: Passos para projeto e análise de uma rede automotiva no Busmaster.

O primeiro passo consiste da criação de mensagens no formato estabelecido na norma SAE J1939 e da geração do banco de dados para sua persistência. Através do Busmaster, diversos parâmetros das mensagens podem ser especificados, como tamanho, PGN, e especificações de cada um dos tipos de dados contidos na mensagem, como nome, largura, posicionamento dentro da mensagem, entre outros.

Em seguida, o projetista da rede pode desenvolver o software que será executado em cada nó do barramento através da janela de edição de código. Nela é possível programar em linguagem C++ o comportamento a ser executado por cada nó na ocorrência de diversos eventos. As funções às quais o usuário deve programar para tratar os eventos são

chamadas pelo software busmaster de *handlers*. Dentre eles, os seguintes foram abordados neste trabalho:

- ***Message Handler*:** é uma função a ser executada pela ECU ao receber uma nova mensagem CAN e que tem como parâmetro a mensagem recebida. Os *message handlers* são divididos em:
  - ***OnPGN_All*:** lida com qualquer mensagem CAN recebida.
  - ***OnPGNRange_<LimiteInferior>_<LimiteSuperior>*:** lida com mensagens CAN recebidas cujo ID se encontra dentro de um intervalo definido pelos parâmetros *LimiteInferior* e *LimiteSuperior*.
  - ***OnPGNID_<PGN>*:** lida com mensagens CAN recebidas com um PGN específico definido pelo parâmetro *PGN*.
  - ***OnPGNName_<NomeDaMensagem>*:** lida com um tipo específico de mensagem CAN recebida, que possui um nome definido no banco de dados de mensagens, este nome é definido através do parâmetro *NomeDaMensagem*.
- ***Timer Handler*:** é uma função a ser executada pela ECU periodicamente. O nome de cada *Timer Handler* segue o formato ***OnTimer_<NomeDoTimer>_<Período>***, onde o parâmetro *NomeDoTimer* nomeia o temporizador e *Período* define o intervalo periódico em milissegundos no qual a função será chamada.
- ***Key Handler*:** é uma funcionalidade que permite a interação com o ambiente externo, na forma de uma função a ser executada pela ECU no momento em que o usuário pressionar uma tecla específica. O nome de cada *Key Handler* segue o formato ***OnKey_<Tecla>***, onde o parâmetro *Tecla* define a qual tecla o *handler* corresponde.
- ***DLL Handler*:** é uma função a ser executada pela ECU no momento em que a DLL referente a ECU é carregada e/ou descarregada no software de simulação. Os nomes dos *DLL Handlers* invocados no momento do carregamento e do descarregamento da DLL são, respectivamente, *OnDLL_Load* e *OnDLL_Unload*.

Ao final do desenvolvimento da rede, o desenvolvedor pode observar e analisar o comportamento da rede através do ambiente de simulação. As mensagens trafegadas no barramento virtual podem ser observadas em tempo real por meio da janela de visualização de mensagens. O desenvolvedor pode analisar os registros realizados por cada ECU na janela de *logging* e gerar estatísticas através de uma janela própria.

A Figura 3.3 apresenta uma captura da tela principal do software Busmaster com as janelas de edição do banco de dados de mensagens e edição do código da ECU abertas. Nota-se que o programa possui uma interface simples e autoexplicativa.

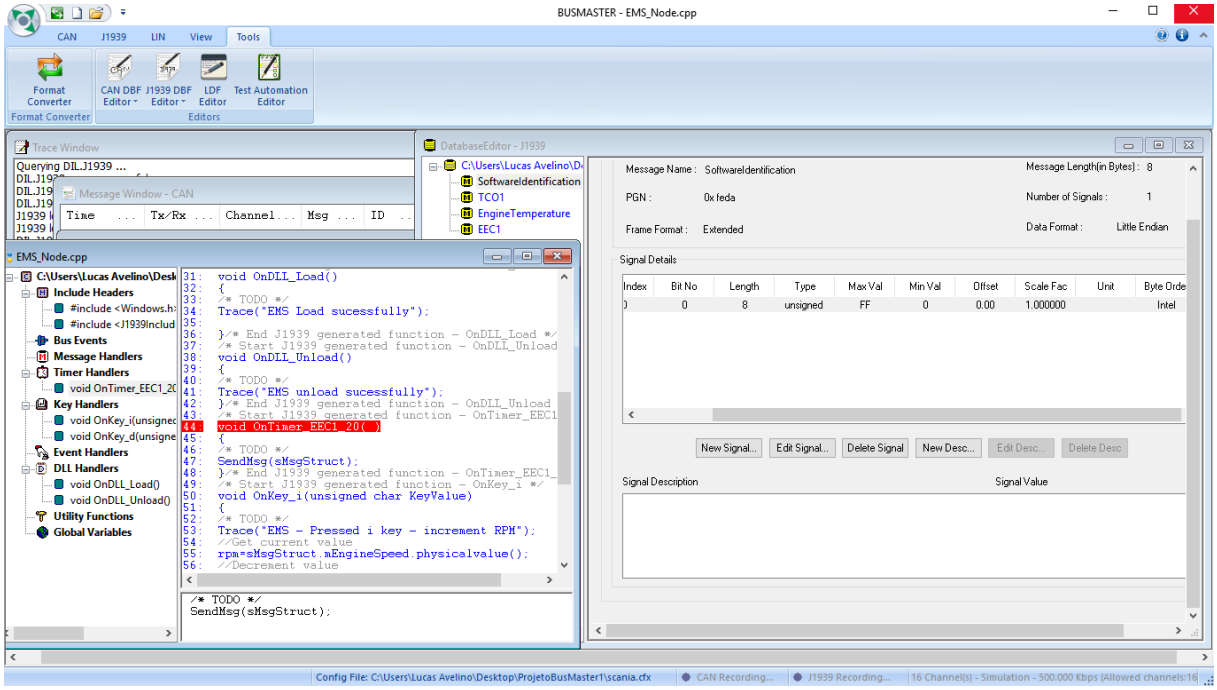


Figura 3.3: Captura da tela do software busmaster com a janela de edição do nó e a janela do editor do banco de dados de mensagens abertas.

As funcionalidades presentes no software Busmaster, que foram descritas nesta seção, foram utilizadas para implementação das ECUs em ambiente embarcado.

Fazendo uso do Busmaster é possível aproveitar o ambiente de desenvolvimento de rede automotiva fornecido pelo mesmo e concentrar o desenvolvimento realizado neste trabalho apenas na aplicação encarregada da adaptação do código gerado para a plataforma embarcada.

3.2.2 Arquitetura do tradutor de código

Através do assitente desenvolvido neste trabalho, o usuário realiza as configurações para a tradução do código fonte da aplicação, produzidos através do busmaster, para a plataforma de *hardware*. Estas configurações são, por exemplo, informar os arquivos de entrada e descrever o mapeamento das teclas para os pinos do microcontrolador. Em seguida, os arquivos de entrada são lidos e através das informações extraídas são gerados os arquivos de código fonte para a plataforma embarcada. Ocorre o processo de compilação e carregamento dos binários executáveis para a plataforma alvo e por fim, pode-se for-

mar uma plataforma de hardware distribuída onde o software produzido para plataforma pode ser testado. A Figura 3.4 apresenta o processo de geração de código para plataforma embarcada proposto neste trabalho.

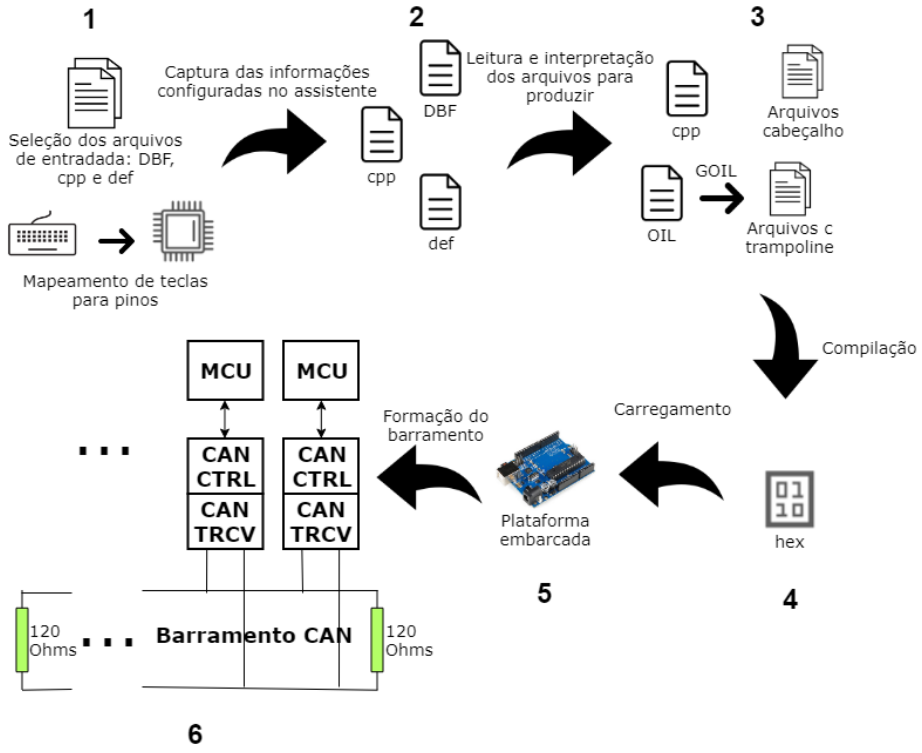


Figura 3.4: Processo de geração de código.

O software projetado tem como principal função, realizar a leitura e conversão dos arquivos de código produzidos pelo Busmaster para execução em plataforma de hardware fazendo uso do trampolineRTOS. Além de facilitar a compilação e o carregamento do binário executável, através de uma única interface.

3.2.2.1 Arquivos de cabeçalho utilizados pelo Busmaster

Ao desenvolver o código de emulação de nó da rede CAN J1939 através do Busmaster o código fonte gerado faz uso de dois arquivos cabeçalho, *"Windows.h"* e *"J1939Includes.h"*.

O arquivo *"Windows.h"* define tipos específicos para a plataforma *desktop* e que são usados por todo o código do busmaster. Para permitir a utilização desses tipos no *target* é necessário a implementação de uma versão semelhante do arquivo para a plataforma de hardware.

O arquivo *"J1939Includes.h"* juntamente com os outros arquivos que são incluídos pelo mesmo definem um conjunto de funções e estrutura de dados importantes utilizadas pelo

código gerado do Busmaster. Dentre essas funções e estrutura de dados, apenas algumas foram re-implementadas especificamente para a plataforma Arduino e grande parte delas foi removida do arquivo a ser utilizado pelo programa. Um exemplo de estrutura de dados que foi re-implementada foi a *J1939_MSG* do busmaster que instancia um *array* de 1785 posições no código do busmaster e devido a restrições de espaço da plataforma arduino, foi implementado para possuir um tamanho parametrizável e por padrão de apenas 8 posições, que engloba grande parte das mensagens no formato SAE J1939.

3.2.2.2 Leitura dos arquivos gerados pelo Busmaster

A criação do banco de dados de mensagens e dos códigos das ECUs por meio do Busmaster gera como saída alguns arquivos, dos quais, os seguntes foram utilizados como entrada para o software desenvolvido: arquivo de definições com extensão ".def", arquivo do banco de dados de mensagens J1939 com extensão ".DBF"e o arquivo de código fonte C++ com extensão ".cpp".

3.2.2.2.1 Arquivo do Banco de Dados de Mensagens

O arquivo com extensão ".DBF"é produzido pelo Busmaster após a criação e edição do banco de dados de mensagens J1939. A formatação do arquivo pode ser observada através do exemplo representado pela Figura 3.5. O arquivo descreve todas as informações inseridas através da tela de edição do software, por meio de uma sintaxe própria.

As estruturas de dados das mensagens são definidas em linguagem C++ e a gramática do *parser* para realização de leitura do arquivo e preenchimento das estruturas de dados é definida através da biblioteca boost spirit.

3.2.2.2.2 Arquivo de Definições

Para cada arquivo em linguagem C++ produzido pelo busmaster para emulação de um nó da rede, também é criado um arquivo com o nome das funções utilizadas. Esse arquivo é utilizado para extração dos nomes dos *handlers*.

3.2.2.2.3 Arquivo de código fonte em linguagem C++

O arquivo de código fonte em linguagem C++ é produzido através da edição do código que especifica o comportamento da ECU, onde o usuário escolhe os eventos aos quais o software irá responder, as mensagens do bancos de dados enviadas ou recebidas, e programa a estratégia de controle realizada pela ECU.

A Figura 3.6 apresenta a formatação do código gerado. Nota-se que o arquivo possui uma estrutura bem definida, com comentários que delimitam blocos de código específicos.

```
//******************************BUSMASTER Messages and signals Database ******************************//

[DATABASE_VERSION] 1.3

[PROTOCOL] J1939

[BUSMASTER_VERSION] [3.2.1]

[NUMBER_OF_MESSAGES] 4

[START_MSG] EEC2,61443,8,1,1,X
[START_SIGNALS] mAcceleratorPedalPosition,8,2,0,U,255,0,1,0.000000,0.395000,%,
[END_MSG]

[START_MSG] EEC1,61444,8,1,1,X
[START_SIGNALS] mEngineSpeed,16,4,0,U,65535,0,1,0.000000,0.125000,RPM,
[END_MSG]

[START_MSG] DLN7,65415,8,1,1,X
[START_SIGNALS] mFuelLevel,8,5,0,U,255,0,1,0.000000,0.395000,l,
[END_MSG]

[START_MSG] EngineTemperature,65262,8,1,1,X
[START_SIGNALS] mEngineOilTemperature,16,3,0,U,65535,0,1,-273.000000,0.031000,degC,
[END_MSG]
```

Figura 3.5: Exemplo de arquivo do banco de dados produzido pelo Busmaster.

A elaboração da gramática para leitura dos arquivos C++ através da biblioteca boost spirit é facilitada pela presença dos comentários, permitindo a separação em diversos módulos para leitura de cada bloco.

3.2.2.3 Geração de Arquivos de Saída

Após realizar a extração das informações dos arquivos para as estruturas de dados, o software segue para a geração dos arquivos de códigos para a plataforma alvo. Para geração desses arquivos adaptados, o software faz uso de arquivos de *template*. O programa conta com diversos arquivos com trechos de códigos fixos e partes parametrizáveis que permitem a geração de um código para o *target*.

3.2.2.3.1 Arquivo em linguagem OIL

O arquivo de descrição dos recursos do sistema operacional em linguagem OIL gerado pelo assistente tem as seguintes características:

- Para garantir a exclusão mútua no acesso ao driver CAN é definido um recurso, o qual é adquirido antes do acesso ao driver e liberado após o mesmo, pelas *threads* de envio e recepção de mensagens.
- Para cada *timer handler* definido no código fonte de entrada são definidos um alarme e uma tarefa. O alarme tem a função de ativar a tarefa toda vez que a contagem

```
/* This file is generated by BUSMASTER */
/* VERSION [1.2] */
/* BUSMASTER VERSION [3.2.1] */
/* PROTOCOL [J1939] */

/* Start J1939 include header */
#include <Windows.h>
#include <J1939Includes.h>
/* End J1939 include header */

/* Start J1939 global variable */
EEC1 msg;
double engineSpeed;
/* End J1939 global variable */

/* Start J1939 Function Prototype  */
GCC_EXTERN void GCC_EXPORT OnKey_i(unsigned char KeyValue);
GCC_EXTERN void GCC_EXPORT OnKey_d(unsigned char KeyValue);
GCC_EXTERN void GCC_EXPORT OnTimer_SendEEC1_20( );
GCC_EXTERN void GCC_EXPORT OnDLL_Load();
/* End J1939 Function Prototype  */

/* Start J1939 generated function - OnTimer_SendEEC1_20 */
void OnTimer_SendEEC1_20( )
{
    SendMsg(msg);
}/* End J1939 generated function - OnTimer_SendEEC1_20 */
```

Figura 3.6: Exemplo de arquivo C++ produzido pelo Busmaster.

chegar a zero, o que ocorre em um período equivalente ao estipulado no *timer handler*. A tarefa consiste apenas de uma *thread* que será encarregada de executar a função do *timer handler*.

- Para todos os *key handlers* são criados um único alarme e uma única tarefa. O alarme ativa a tarefa toda vez que a contagem chega a zero e o período do alarme indica o ciclo de varredura das entradas digitais e analógicas. A tarefa consiste de uma *thread* que terá a função de executar a leitura de cada uma das entradas digitais e analógicas e chamar os respectivos *key handlers* sequencialmente.

- Para todas os *message handlers* são criadas uma única tarefa, uma única rotina de tratamento a interrupção classe 2 e um único evento. A rotina de tratamento de interrupção serve para indicar o recebimento de uma mensagem através do disparo do evento. A tarefa, por sua vez, realiza uma espera bloqueante pelo evento e ao

recebê-lo, realiza a leitura de todas as mensagens sequencialmente designando para cada um dos *message handlers* responsáveis pelas mensagens recebidas.

- Uma tarefa foi criada para o envio das mensagens pelo barramento CAN. Uma fila de mensagens foi utilizada para enviar as mensagens para a tarefa designada. Para a fila de mensagens, um evento para indicar fila cheia foi definido em cada uma das demais tarefas e um evento para indicar fila vazia foi definido na tarefa que envia mensagens pelo barramento CAN, além de um recurso para proteger o acesso a fila de mensagens.

3.2.2.3.2 Arquivos de código em linguagem C++

O arquivo de cabeçalho em linguagem C++ *msg_types.h* gerado é onde são definidas todas as estruturas de dados das mensagens criadas no banco de dados J1939 para serem usadas pelo código fonte. Cada uma das mensagens consiste de uma estrutura de dados baseada em uma *J1939_MSG* e de membros que representam as informações contidas na mensagem.

O arquivo de código fonte em linguagem C++ contém o código de cada uma das funções criadas no Busmaster, juntamente com o código das tarefas auxiliares. A função *SendMsg* do Busmaster é definida como uma simples inserção da mensagem J1939 na fila de mensagens.

A tarefa *can_send_task* é responsável pelo envio das mensagens J1939 através do controlador CAN e executa um loop infinito. As mensagens presentes na fila são lidas e enviadas pelo barramento sequencialmente. A fila permite que a tarefa seja bloqueada quando a fila se encontrar vazia e permite também que sejam bloquadas as demais tarefas que desejem enviar mensagens enquanto a fila se encontra cheia. A Figura 3.7 apresenta o fluxograma do código executado pela tarefa.

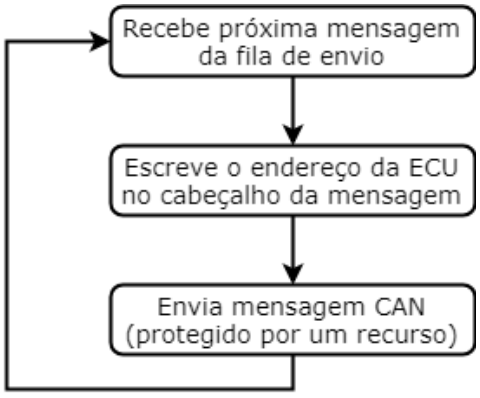


Figura 3.7: Fluxograma do código executado pela tarefa *can_send_task*.

As tarefas relativas aos *timer handlers* consistem simplesmente do código dos mesmos, sem alterações. Um alarme é responsável por ativá-la de tempos em tempos, com período definido através do próprio protótipo do *timer handler* que indica o seu valor em milissegundos. A Figura 3.8 apresenta uma representação do fluxograma e do alarme que ativa a tarefa.

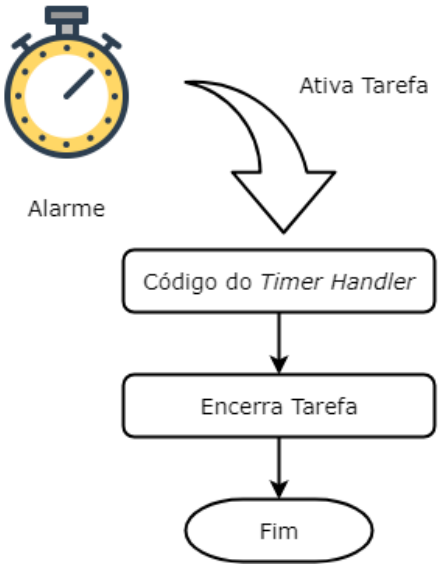


Figura 3.8: Fluxograma do código executado pela tarefa *timer_task*.

A tarefa *pins_reader_task* realiza a leitura dos pinos digitais mapeados aos *key handlers* sequencialmente e caso se encontrem no estado ativo, o *key handler* associado é invocado. Além da leitura dos pinos digitais, também é realizada a leitura dos pinos analógicos e o valor lido é passado como parâmetro para o respectivo *key handler*. Esta tarefa é ativada de tempos em tempos por um alarme, de acordo com o período definido no assistente. A Figura 3.9 apresenta uma representação do fluxograma e do alarme que ativa a tarefa.

A tarefa *can_recv_task* é responsável pela leitura das mensagens recebidas pelo controlador CAN. O controlador CAN utiliza um pino para indicar a recepção de uma nova mensagem. Esse pino por sua vez é associado a uma interrupção no microcontrolador e a rotina de tratamento indica esse fenômeno à tarefa *can_recv_task* através de um evento. A tarefa executa um loop infinito onde realiza a espera pelo evento e em seguida, a leitura de todo o buffer de mensagens, com o encaminhamento para os *message handlers* adequados. A Figura 3.10 apresenta o fluxograma da rotina de tratamento a interrupção e da tarefa que realiza a leitura das mensagens CAN.

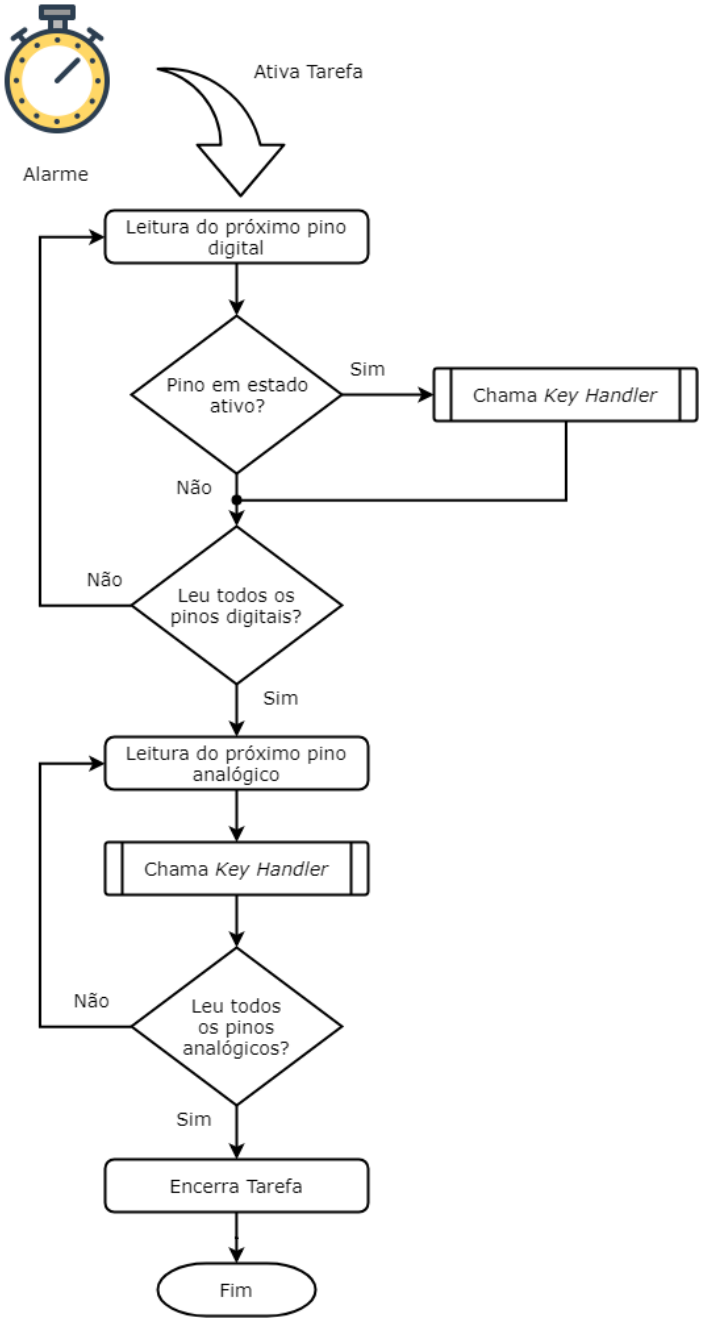


Figura 3.9: Fluxograma do código executado pela tarefa *pins_reader_task*.

3.2.2.4 Fluxograma do software do assitente *Code Generator*

O fluxograma representado na Figura 3.11 apresenta os principais passos realizados pelo software do assistente, desde a leitura dos arquivos de entrada e armazenamento das estruturas de dados, até a geração do binário executável e carregamento na plataforma alvo.

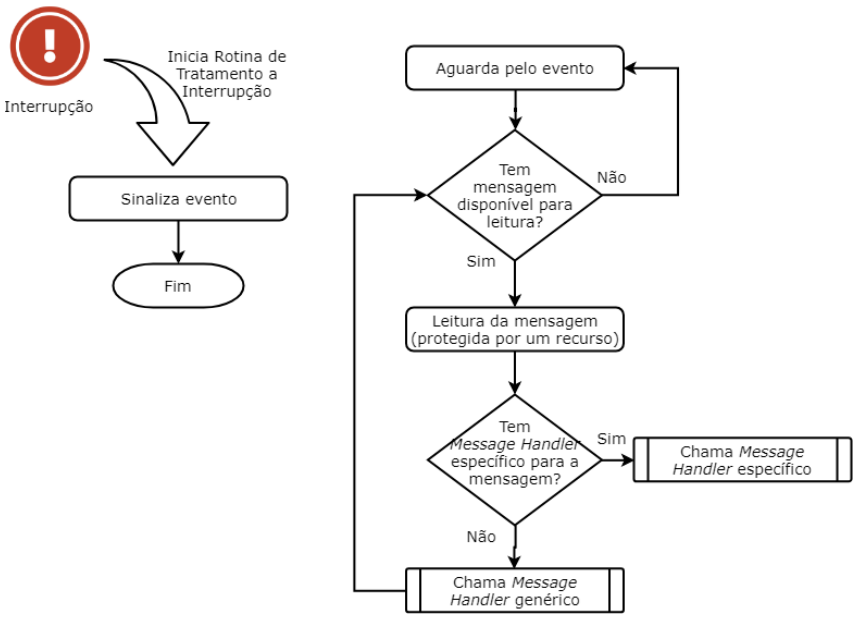


Figura 3.10: Fluxograma do código executado pela tarefa *can_recv_task* juntamente com a rotina de tratamento a interrupção associada a recepção de mensagem CAN pelo controlador.

3.2.3 Arduino

Uma importante decisão de projeto foi a seleção dos componentes de hardware. Os microcontroladores da plataforma Arduino foram escolhidos principalmente pelo baixo custo e disponibilidade para compra. Além disso, outros pontos importantes considerados foram: o fato de possuírem uma interface de programação que torna a prototipagem simples e eficaz e a existência de bastante suporte pela comunidade online.

Arduino é uma plataforma eletrônica que integra hardware e software projetada para ser simples, fácil de usar, de baixo custo e extensível[16]. Existem várias placas de desenvolvimento com diferentes microcontroladores na lista de placas Arduino. As duas selecionadas para o projeto foram as placas Arduino UNO e Arduino NANO, principalmente, por possuírem maior disponibilidade de compra e pelo baixo custo.

As Figuras 3.12 a 3.13 representam as placas Arduino UNO e Arduino NANO, respectivamente. Ambas as placas são baseadas no microcontrolador da Microchip ATmega328p. Possuem 14 pinos de entrada e saída digitais, 6 pinos de entrada analógica, cristal de quartzo de 16 MHz e conector USB.

Além disso, conta com uma comunidade de usuários online bastante extensa, documentação da plataforma e da interface de programação através da página web e uma quantidade muito grande de módulos de hardware e software disponíveis, construídos para interoperar com placas arduino.

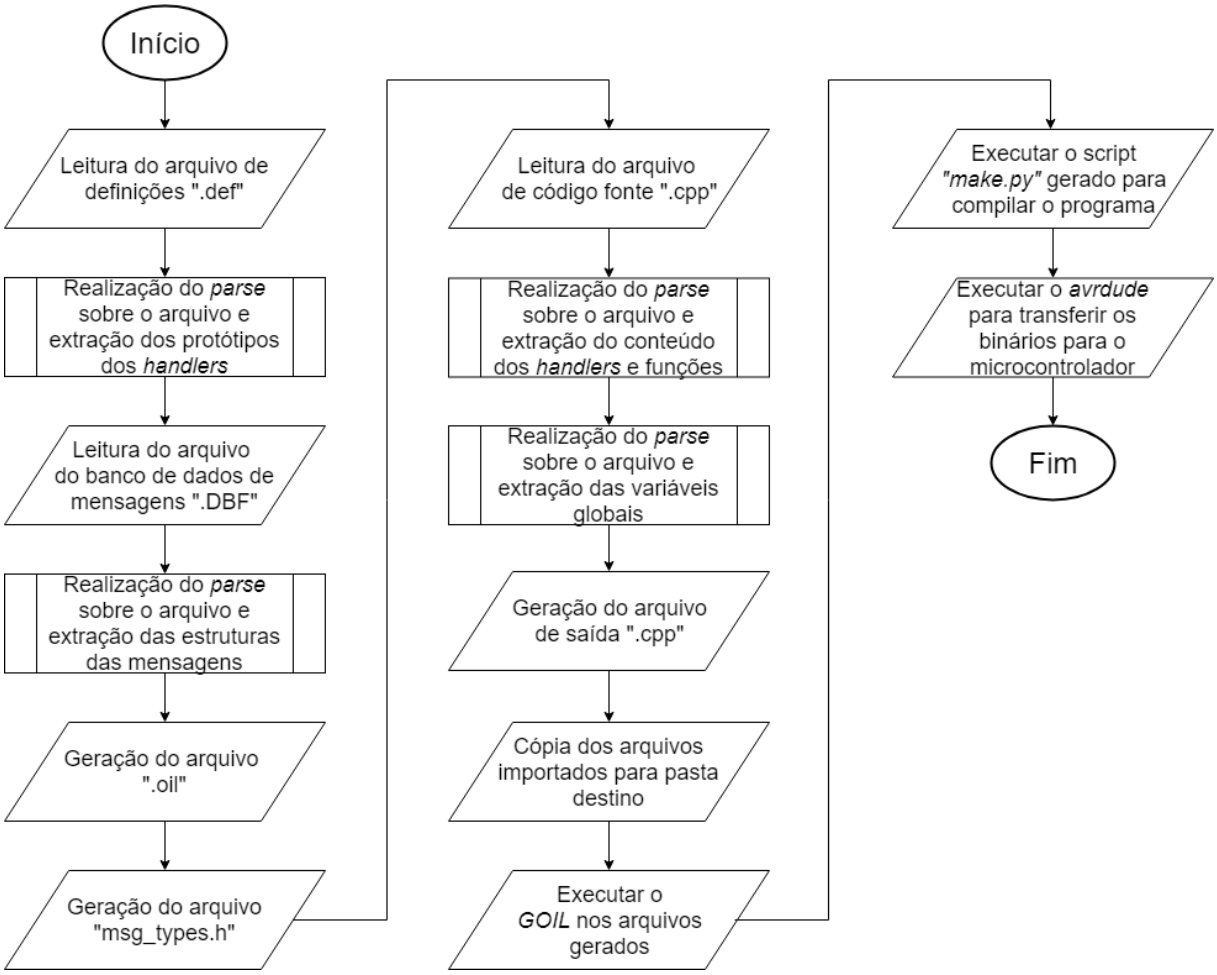


Figura 3.11: Fluxograma correspondente a lógica de funcionamento do software.

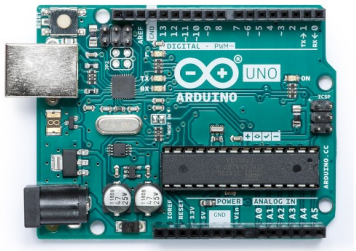


Figura 3.12: Placa de desenvolvimento Arduino UNO.

Contudo, a plataforma arduino conta com algumas limitações. Entre elas, o ambiente de desenvolvimento que é bastante limitado em recursos de *debug*, uma vez que não possui uma interface BDM (*Background Debug Mode*), a ineficiência de grande parte das bibliotecas fornecidas e APIs bastante restritivas no modo de usar os recursos disponíveis.

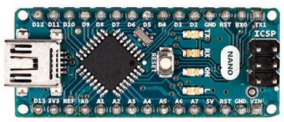


Figura 3.13: Placa de desenvolvimento Arduino NANO.

3.2.4 Módulo CAN MCP2515

O microcontrolador ATmega328p, apesar de possuir uma ótima relação custo-benefício, apresenta uma limitação importante para a realização deste trabalho, que consiste da ausência de um controlador CAN embutido. Esta limitação torna necessária a utilização de um módulo de hardware em conjunto para fornecer essa funcionalidade.

O módulo MCP2515 representado pela Figura 3.14 foi selecionado para realizar a interface com o barramento CAN. Consiste de uma placa de circuito impresso que inclui o controlador CAN MCP2515 da Microchip e o transceptor CAN TJA1050 da NXP Semiconductors. Possui interface SPI para controle através de um microcontrolador externo, suporta a especificação CAN 2.0 B com velocidade de comunicação de até 1Mb/s e possui alimentação de 5V.

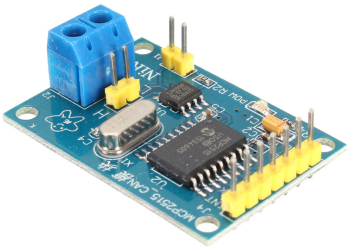


Figura 3.14: Módulo CAN MCP2515.

O módulo foi escolhido principalmente pelo excelente custo-benefício, disponibilidade em lojas online para compra e pela existência de bibliotecas de software já desenvolvidas e testadas para plataforma arduino. Entretanto, a interface SPI pode ser um fator limitante da velocidade de comunicação se comparado a controladores CAN integrados ao microcontrolador.

A Figura 3.15 ilustra a plataforma de hardware utilizada, exibindo as conexões entre o módulo MCP2515 e a placa de desenvolvimento arduino NANO.

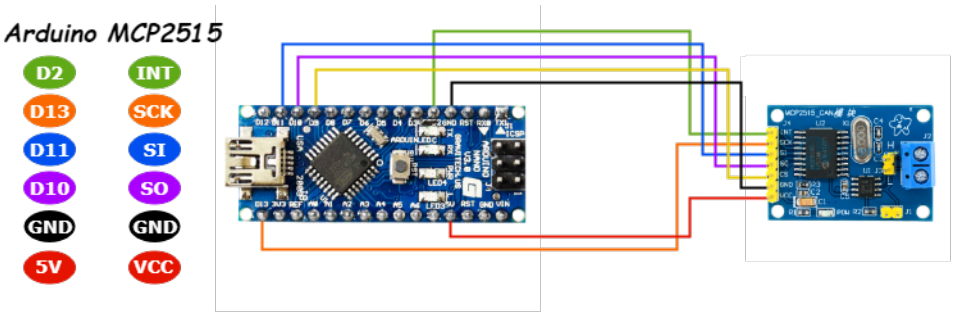


Figura 3.15: Esquemático das conexões entre arduino e o módulo MCP2515.

3.2.5 Trampoline RTOS

Optou-se pela utilização do Trampoline RTOS que é um sistema operacional de tempo real, com escalonamento estático de tarefas, para sistemas embarcados de pequeno porte e com APIs alinhadas com os padrões da indústria automotiva OSEK/VDX OS e AUTOSAR OS 4.2. Possui suporte a diferentes microcontroladores, incluindo arquitetura AVR de 8 bits presente nas placas Arduino UNO e NANO.

Através do sistema operacional trampoline é possível descrever diversas características do sistema como propriedades das tarefas, alarmes, mecanismos de sincronização, entre outras, por meio de arquivos escritos em linguagem OIL. O sistema implementa um escalonador de prioridades estático onde cada tarefa deve possuir uma prioridade única e as tarefas podem ser preemptíveis ou não.

Trampoline é uma iniciativa acadêmica, de software livre, para implementação do padrão OSEK/VDX RTOS. Foi desenvolvido visando principalmente portabilidade e eficiência no uso de memória [17]. Como resultado, até o momento em que este trabalho foi desenvolvido, trampoline já havia sido portado para 11 plataformas, incluindo arquiteturas de microcontroladores com memória bastante escassa, como ATmega 328p que possui apenas 2KB de memória RAM e 32KB de memória Flash [18].

A utilização de um RTOS contudo, também tem algumas limitações como o espaço ocupado pelo sistema operacional na memória limitada dos microcontroladores, a sobrecarga na troca de contexto que pode reduzir o desempenho do sistema, a complexidade de problemas de sincronização de tarefas, dentre outros.

3.2.6 Linguagem, bibliotecas e frameworks utilizados

C++ foi a linguagem escolhida para o desenvolvimento do projeto principalmente pelo fato de ser a linguagem utilizada pelo busmaster e pela plataforma de hardware, assim facilitando a portabilidade do código fonte.

Para desenvolvimento da interface gráfica, permitindo o uso do software de forma simples e prática, optou-se pelo framework Qt[19], que possui uma licença gratuita, documentação bastante extensa, ampla comunidade online, uma grande quantidade de exemplos disponíveis, dentre diversos outros benefícios.

Para realização da leitura dos códigos gerados pelo software busmaster e armazenamento em estruturas de dados próprias e internas ao software, a biblioteca Boost Spirit X3[20] foi utilizada. Esta biblioteca consiste de um *parser* descendente recursivo orientado a objetos para C++. Permite a descrição de *parsers* através de uma linguagem de domínio específico semelhante a *Extended Backus Naur Form* (EBNF), embutida e escrita em C++ sem necessidade de utilização de outros passos de compilação ou outros softwares.

Capítulo 4

Estudos de Caso

Esta seção apresenta a descrição dos estudos de caso realizados. Foram realizados três estudos de caso para testar a implementação da plataforma de desenvolvimento proposta. O primeiro é o caso mais simples com apenas duas ECUs. O segundo estudo de caso, trata-se de uma rede um pouco mais complexa com três ECUs, com variáveis divididas entre duas ECUs. Por fim, o terceiro estudo de caso é o mais complexo dos três e consiste de uma rede automotiva com quatro ECUs, onde uma delas realiza cálculos de dinâmica veicular.

4.1 Estudo de Caso 1

A rede automotiva desenvolvida para o estudo de caso 1 consiste de duas ECUs (aqui denominadas ECU1 e ECU2). A ECU1 é responsável por obter valores de rotação do motor que variam no tempo e transmití-los para a ECU2 através de um barramento CAN J1939. A ECU2 simplesmente monitora o barramento e realiza a recepção das mensagens transmitidas. A Figura 4.1 apresenta a rede automotiva proposta.

A informação de rotação do motor em RPM é transmitida através da mensagem *EEC1*. Esta é uma mensagem que segue o padrão SAE J1939 e possui 8 bytes de dados. No quarto byte de dados, ela apresenta a informação de rotação do motor em RPM. Esta mensagem está representada na Figura 4.2. A rotação do motor foi o único parâmetro utilizado da mensagem, os demais parâmetros foram especificados como não disponíveis.

A mensagem com o valor atual da rotação do motor é transmitida a cada 20 milissegundos pela ECU1. A rede pode ser avaliada através da análise do tempo de recepção dessa mensagem pela ECU2.

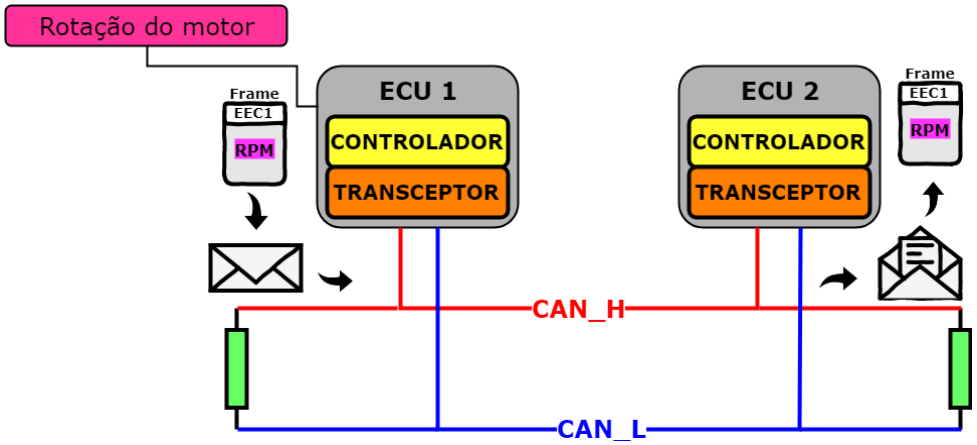


Figura 4.1: Rede automotiva do estudo de caso 1.

Figura 4.2: Mensagem EEC1 no padrão J1939. Foi utilizada para esse estudo de caso apenas a informação de rotação do motor, portanto as informações presentes nos outros bytes não foram descritas.

4.1.1 Desenvolvimento da rede automotiva virtual

Para o desenvolvimento da rede virtual no software Busmaster o primeiro passo foi a criação da mensagem EEC1 no banco de dados. As configurações da mensagem podem ser observadas na Figura 4.3 que apresenta a tela de edição de mensagens do Busmaster.

Após a geração do banco de dados, o próximo passo é a programação do comportamento das ECUs através da tela de *Node Simulation*.

Para descrever o comportamento da ECU1 foram utilizados as seguintes funcionalidades do Busmaster.

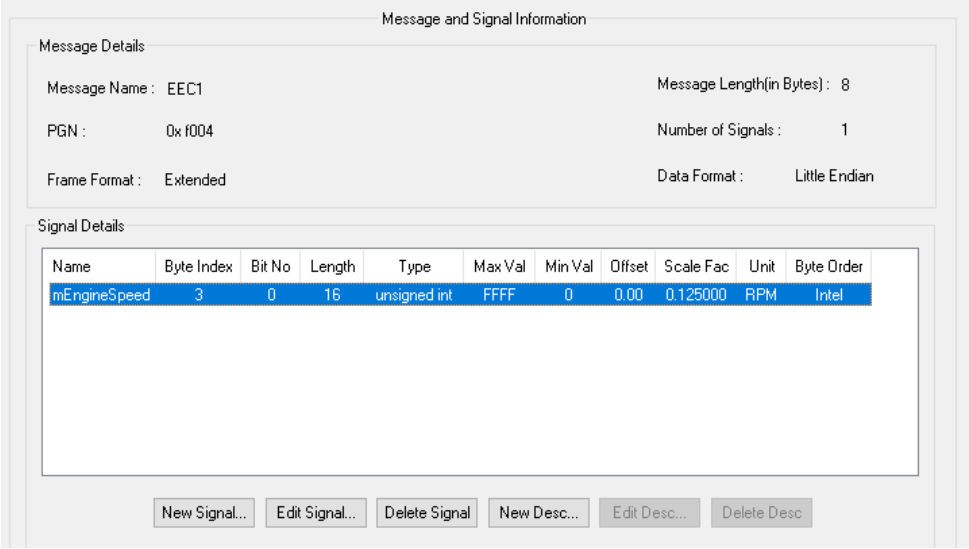


Figura 4.3: Mensagem EEC1 vista no editor de mensagens J1939 do Busmaster.

- **Variáveis globais** para instanciar a mensagem e o valor de rotação do motor a serem manipulados.
- **OnDLL_Load** para inicializar a mensagem com a prioridade adequada.
- **OnTimer** para garantir o envio da mensagem a cada 20 milissegundos.
- **OnKey** para manipular o valor da rotação do motor. Ao pressionar a tecla i do teclado, seu valor é incrementado em 100 RPM, ao pressionar a tecla d, seu valor é decrementado de 100 RPM. Permitindo a simulação da variação do parâmetro.

A Figura 4.4 apresenta um diagrama com os fluxogramas das principais funções definidas no código da ECU1 que definem o comportamento dela na rede.

A função da ECU2 está ligada ao monitoramento da rede quando em ambiente embarcado, já que o Busmaster provê os meios de analisar o barramento no ambiente virtual. No ambiente embarcado é necessário que uma ECU receba as mensagens e as exiba por meio do envio para uma máquina *host* utilizando a interface serial. O código para a ECU2 então, utiliza as seguintes funcionalidades.

- **OnDLL_Load** para enviar pela interface serial o cabeçalho da tabela que será formada em tempo real ao receber as mensagens CAN.
- **OnPNG_All** para realizar a recepção de qualquer mensagem transmitida no barramento e possibilitar a impressão das informações pela interface serial. Para o envio de informações pela interface serial é preciso inserir código C++ que será

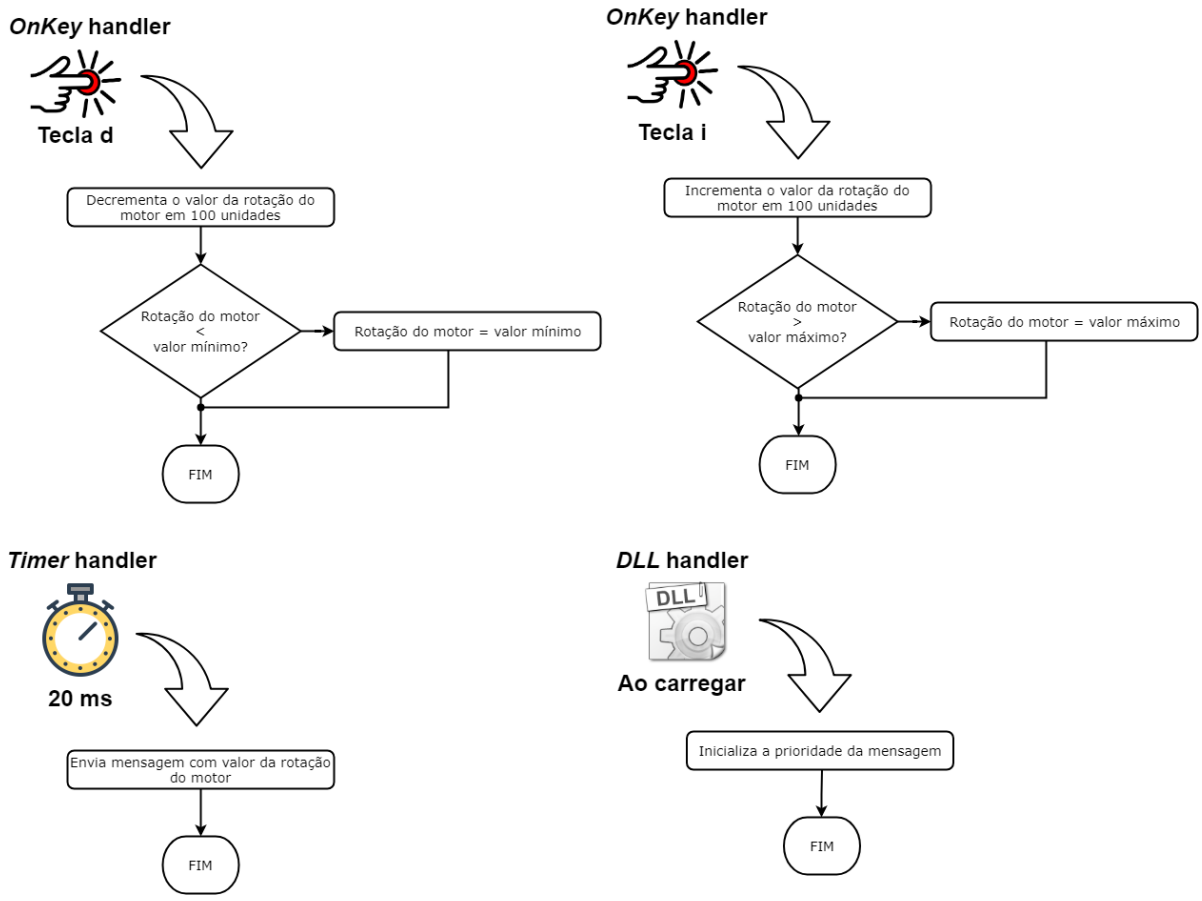


Figura 4.4: Diagramas com as funcionalidades da ECU1.

executado somente no *target*. Para isso, podem ser utilizadas as diretivas de pré-processamento *#ifdef ou #ifndef* em conjunto com a macro *CG_ON_TARGET*, definida na plataforma alvo.

A Figura 4.5 apresenta um diagrama com as principais funcionalidades exercidas pela ECU2.

Observa-se na Figura 4.6 o código produzido para a ECU1 e ECU2 com o auxílio da ferramenta de edição do Busmaster.

Por fim, o último passo consiste em analisar o barramento virtual e se certificar de que a rede se comporta conforme o esperado. A Figura 4.7 apresenta a tela de mensagens do barramento J1939 no Busmaster e a tela com os sinais observados. A informação de tempo relativo da mensagem pode ser vista no canto esquerdo indicando que o período com que a mensagem passa pelo barramento é, de fato, de aproximadamente 20 milissegundos. O id correto também pode ser observado, com o valor do PGN e prioridades definidas no código do programa.

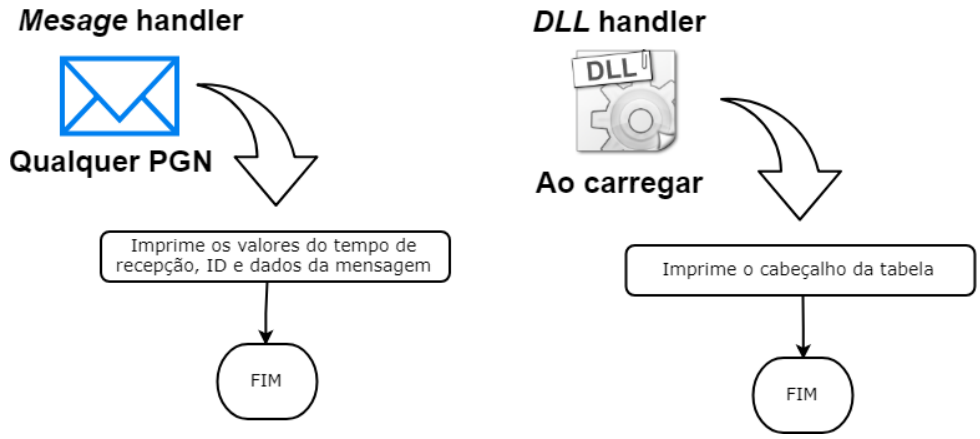


Figura 4.5: Diagramas com as funcionalidades da ECU2.

```
/* Start J1939 global variable */
EEC1 msg;
double engineSpeed;
/* End J1939 global variable */

/* Start J1939 generated function - OnKey_i */
void OnKey_i(unsigned char KeyValue)
{
    engineSpeed = msg.mEngineSpeed.physicalvalue() + 100;
    if(engineSpeed > 8031.875)
    {
        engineSpeed = 8031.875;
    }
    msg.mEngineSpeed.physicalvalue(engineSpeed);
}/* End J1939 generated function - OnKey_i */
/* Start J1939 generated function - OnKey_d */
void OnKey_d(unsigned char KeyValue)
{
    engineSpeed = msg.mEngineSpeed.physicalvalue() - 100;
    if(engineSpeed < 0)
    {
        engineSpeed = 0;
    }
    msg.mEngineSpeed.physicalvalue(engineSpeed);
}/* End J1939 generated function - OnKey_d */
/* Start J1939 generated function - OnTimer_SendEEC1_20 */
void OnTimer_SendEEC1_20( )
{
    SendMsg(msg);
}/* End J1939 generated function - OnTimer_SendEEC1_20 */
/* Start J1939 generated function - OnDLL_Load */
void OnDLL_Load()
{
    msg.id.setPriority(3);
}/* End J1939 generated function - OnDLL_Load */
```

```
/* Start J1939 global variable */
UINT8* buf;
UINT16 rcvTime;
unsigned int len;
UINT32 rxId;
/* End J1939 global variable */

/* Start J1939 generated function - OnPGN_All */
void OnPGN_All(J1939_MSG RxMsg)
{
    rxId = RxMsg.id;
    buf = RxMsg.data;
    len = RxMsg.dlc;
#ifdef CG_ON_TARGET
    rcvTime = millis();
    Serial.print(rcvTime);
    Serial.print("\t\t");
    Serial.print("0x");
    Serial.print(rxId, HEX);
    Serial.print("\t");
    for(unsigned int i = 0; i<len; i++){
        if(buf[i] > 15)
        {
            Serial.print("0x");
        } else
        {
            Serial.print("0x0");
        }
        Serial.print(buf[i], HEX);
        Serial.print("\t");
    }
    Serial.println();
#endif
}/* End J1939 generated function - OnPGN_All */
/* Start J1939 generated function - OnDLL_Load */
void OnDLL_Load()
{
#ifdef CG_ON_TARGET
    Serial.println("CAN BUS Module Initialized!");
    Serial.println("Time\t\tId\t\tByte0\tByte1\tByte2"
                   "\tByte3\tByte4\tByte5\tByte6\tByte7");
#endif
}/* End J1939 generated function - OnDLL_Load */
```

Figura 4.6: Código para a ECU1 e ECU2 produzido com o auxílio da ferramenta de edição do Busmaster. Algumas partes do arquivo gerado foram omitidas para simplificação.

4.1.2 Conversão da rede virtual para rede embarcada

O processo de conversão do código gerado para a plataforma embarcada é feito através do assistente *Code Generator* desenvolvido neste trabalho. Para cada ECU presente na

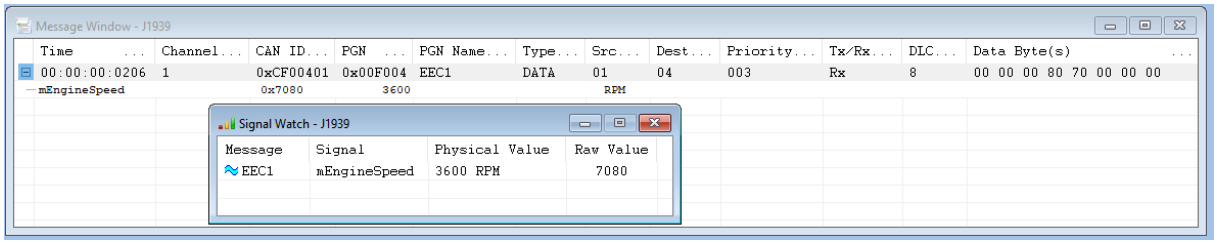


Figura 4.7: Tela de mensagens do barramento J1939 para o estudo de caso 1.

rede, deve-se invocar o assistente e realizar as configurações para geração do binário para a plataforma embarcada.

Para a conversão do código gerado para a ECU1 as teclas associadas aos *key handlers* precisam ser mapeadas, através do assistente, para pinos digitais no microcontrolador. O assistente permite que o usuário escolha entre a lógica de ativação em nível alto ou baixo, que permite a utilização de chaves em circuito de pull-down ou pull-up, respectivamente. Para o estudo de caso 1, serão utilizadas duas chaves em circuito de pull-up, ou seja, ativas em nível lógico baixo.

Para a ECU1 é necessário indicar através do assistente que a ECU envia mensagens pelo do barramento CAN, permitindo que sejam instanciados a tarefa de envio de mensagens e a fila de envio. O endereço $0x00$ será associado a ECU1. O código gerado para a ECU1 é apresentado na Figura 4.8.

Já para a geração de código para o microcontrolador referente a ECU2, é necessário indicar que a ECU faz uso da interface serial, para que seja gerado o código da inicialização dessa interface com taxa de transmissão de 115200 bits por segundo. O endereço $0x01$ será associado a ECU2. O código gerado para a ECU2 está representado pela Figura 4.8.

A Figura 4.9 apresenta todas as telas do assistente de tradução de código com as configurações utilizadas para a geração de código para a ECU1.

4.1.3 Análise dos resultados

Para a análise dos estudos de caso, utilizou-se do monitor serial presente na IDE do arduino. Um importante aspecto a ser analisado neste estudo de caso é o intervalo entre recepções de mensagens CAN.

Pode-se observar através da Figura 4.10 a inicialização da ECU2, evidenciado pela presença do cabeçalho da tabela de mensagens. Nota-se que o intervalo entre recepções de mensagens CAN é de aproximadamente 20 milissegundos, como esperado. O Id da mensagem também corresponde ao esperado, dado que a prioridade da mensagem foi configurada como 3, o PGN como $0xF004$ e o endereço da ECU1 como $0x00$.

```
EEC1 msg;
double engineSpeed;

void setup()
{
    while (CAN_OK != CAN.begin(CAN_250KBPS)){}
    for(unsigned int i = 0; i < N_DIGITAL_KEY_HANDLERS; ++i)
    {
        pinMode(digital_key_handlers[i].key, INPUT);
    }
    OnDLL_Load();
}

TASK(can_send_task)
{
    J1939_MSG to_send_msg;
    while(1)
    {
        can_send_msg_queue.receive(to_send_msg);
        to_send_msg.id.setSourceAddress(0);
        GetResource(can_hardware);
        CAN.sendMsgBuf(to_send_msg.id, 1, 8, to_send_msg.data);
        ReleaseResource(can_hardware);
    }
}

TASK(OnTimer_SendEEC1_20)
{
    SendMsg(msg);
    TerminateTask();
}

void OnKey_i(unsigned char KeyValue)
{
    engineSpeed = msg.mEngineSpeed.physicalvalue() + 100;
    if(engineSpeed > 8031.875)
    {
        engineSpeed = 8031.875;
    }
    msg.mEngineSpeed.physicalvalue(engineSpeed);
}

void OnKey_d(unsigned char KeyValue)
{
    engineSpeed = msg.mEngineSpeed.physicalvalue() - 100;
    if(engineSpeed < 0)
    {
        engineSpeed = 0;
    }
    msg.mEngineSpeed.physicalvalue(engineSpeed);
}

TASK(pins_reader)
{
    for(unsigned int i = 0; i < N_DIGITAL_KEY_HANDLERS; ++i)
    {
        if(digitalRead(digital_key_handlers[i].key) ==
                        digital_key_handlers[i].active_level)
        {
            (*digital_key_handlers[i].handler)(digital_key_handlers[i].key);
        }
    }
    for(unsigned int i = 0; i < N_ANALOG_KEY_HANDLERS; ++i)
    {
        (*analog_key_handlers[i].handler)(analogRead(analog_key_handlers[i].key));
    }
    TerminateTask();
}

void OnDLL_Load()
{
    msg.id.setPriority(3);
}
```

```
UINT8* buf;
UINT16 rcvTime;
unsigned int len;
UINT32 rxId;

void setup()
{
    Serial.begin(115200);
    while (CAN_OK != CAN.begin(CAN_250KBPS)){}
    OnDLL_Load();
}

void OnPGN_All(J1939_MSG& RxMsg)
{
    rxId = RxMsg.id;
    buf = RxMsg.data;
    len = RxMsg.dlc;
#ifdef CG_ON_TARGET
    rcvTime = millis();
    Serial.print(rcvTime);
    Serial.print("\t\t");
    Serial.print("0x");
    Serial.print(rxId, HEX);
    Serial.print("\t");
    for(unsigned int i = 0; i<len; i++){
        if(buf[i] > 15)
        {
            Serial.print("0x");
        } else
        {
            Serial.print("0x0");
        }
        Serial.print(buf[i], HEX);
        Serial.print("\t");
    }
    Serial.println();
#endif
}

ISR(can_recv_isr)
{
    SetEvent(can_recv_task, can_recv_evt);
}

TASK(can_recv_task)
{
    J1939_MSG received_msg;
    while(1)
    {
        WaitEvent(can_recv_evt);
        ClearEvent(can_recv_evt);
        GetResource(can_hardware);
        while(CAN_MSGAVAIL == CAN.checkReceive())
        {
            CAN.readMsgBuf((byte*)&received_msg.dlc,
                           (byte*)received_msg.data);
            received_msg.id = CAN.getCanId();
            ReleaseResource(can_hardware);
            OnPGN_All(received_msg);
            GetResource(can_hardware);
        }
        ReleaseResource(can_hardware);
    }
}

void OnDLL_Load()
{
#ifdef CG_ON_TARGET
    Serial.println("CAN BUS Module Initialized!");
    Serial.println("Time\t\tId\t\tByte0\tByte1\tByte2"
                   "\tByte3\tByte4\tByte5\tByte6\tByte7");
#endif
}
```

Figura 4.8: Código gerado para o microcontrolador referente a ECU1 e ECU2. Alguns trechos do arquivo foram omitidos para simplificação.

É possível observar na Figura 4.11 que em um dado instante, o valor presente nos bytes 3 e 4 da mensagem EEC1 aumentou seu valor de $0x6D60$ para $0x7080$. Este incremento equivale a 800 em base decimal. Se multiplicarmos o valor do incremento pelo fator definido na mensagem, que é de 0.125 e somarmos o *offset* que é de 0 para converter o valor da mensagem para o valor real, obtemos $800 * 0.125 + 0 = 100$. Este valor corresponde ao programado no *key handler* da ECU1.

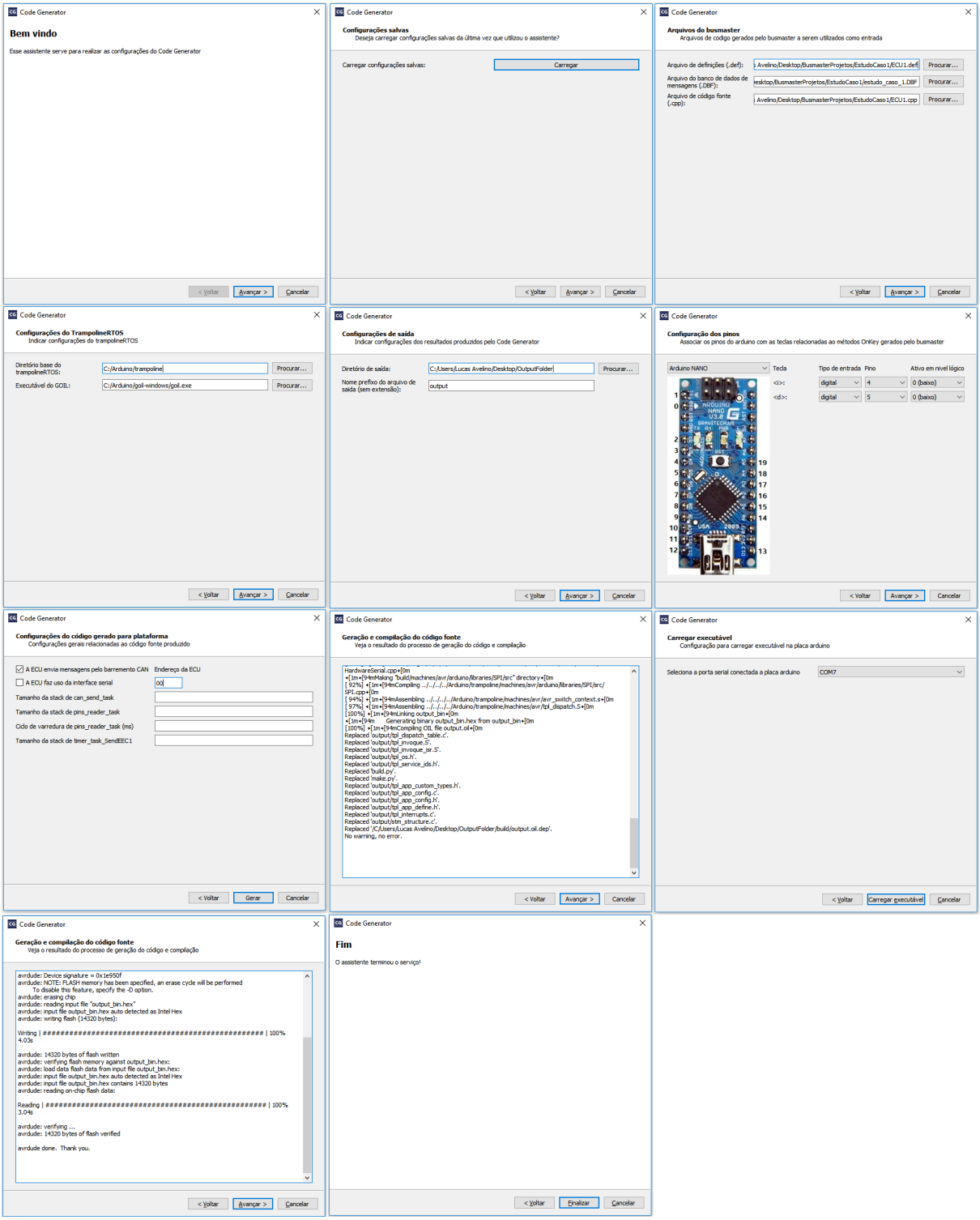


Figura 4.9: Telas do assistente *Code Generator*.

Com base nos resultados obtidos é possível verificar que a rede embarcada se comportou de forma semelhante à rede virtual para o estudo de caso 1. A ECU2 permitiu a realização de uma análise do barramento CAN através do registro em tempo real pela

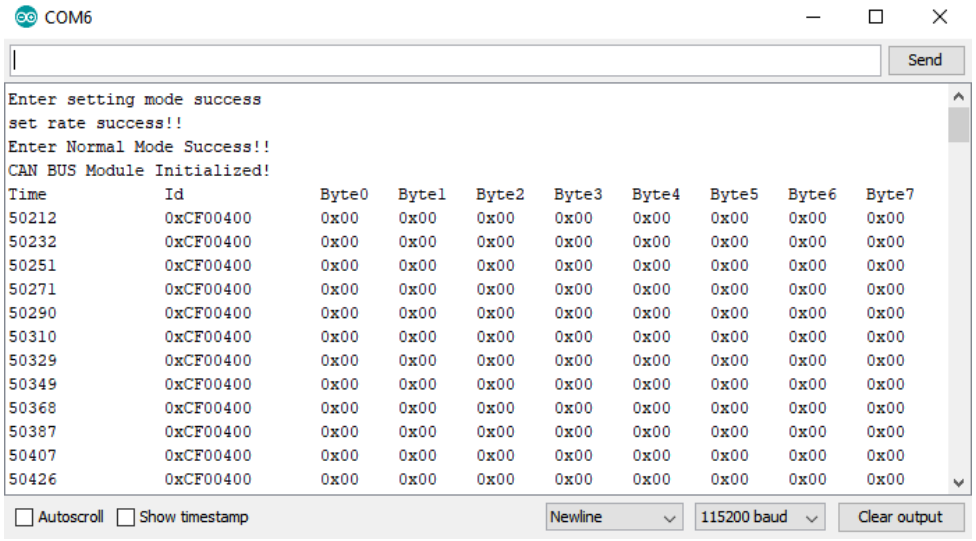


Figura 4.10: Tela do monitor serial da IDE do arduino. A imagem revela as informações obtidas através da ECU2 sobre as mensagens recebidas no momento em que a ECU1 é inicializada.

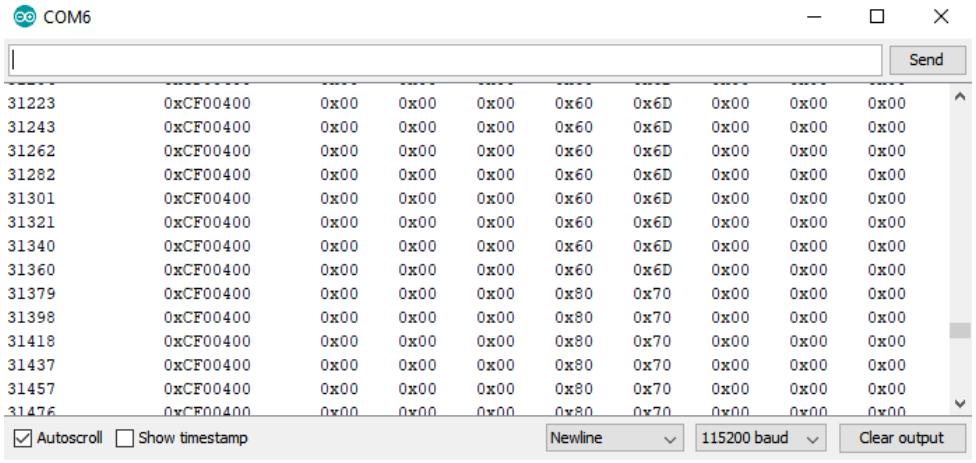


Figura 4.11: Tela do monitor serial da IDE do arduino no momento em que o valor da presente nos bytes 3 e 4 aumentou de $0x6D60$ para $0x7080$.

interface serial de todas as mensagens que trafegaram na rede. Nota-se que o intervalo de transmissão da mensagem EEC1 correspondeu ao esperado, assim como a resposta ao acionamento das chaves de incremento e decremento da rotação do motor.

O estudo de caso 1, comprovou o correto funcionamento da função *message handler* do tipo *OnPGN_All* que trata todas as mensagens recebidas. Também permitiu verificar o comportamento dos *DLL handlers* do tipo *OnDLL_Load* e dos *key handlers* que foram utilizados para incrementar e decrementar a variável de rotação do motor através do pressionar de dois botões pelo usuário. O funcionamento adequado do *timer handler*

também pode ser observado através da medição do intervalo de recepção das mensagens que indicou que o envio ocorreu no tempo esperado.

O desempenho da rede embarcada para o estudo de caso 1 foi equivalente ao da rede virtual. Por se tratar de uma rede simples e de baixa complexidade não houve complicações no funcionamento de nenhuma das ECUs.

4.2 Estudo de Caso 2

Para o estudo de caso 2, a rede automotiva representada pela Figura 4.12 foi desenvolvida. 3 ECUs estão presentes na rede, EMS (do inglês, *Engine Management System*, Sistema de Gerenciamento do Motor), GMS (do inglês, *Gearbox Management System*, Sistema de Transmissão) e ICL (do inglês, *Instrument Cluster System*, Sistema do Painel de Instrumentos). A nomenclatura das ECUs aqui utilizada, se baseia na documentação da Scania.

A EMS é responsável pelo controle da variável de rotação do motor e da variável de temperatura do líquido de arrefecimento. A GMS controla a variável da marcha atual. Por fim, a ICL serve para apresentar dados de relevância para o condutor. Controla as funcionalidades do painel de instrumentos de um veículo. E aqui é responsável pela recepção das mensagens e sua exibição.

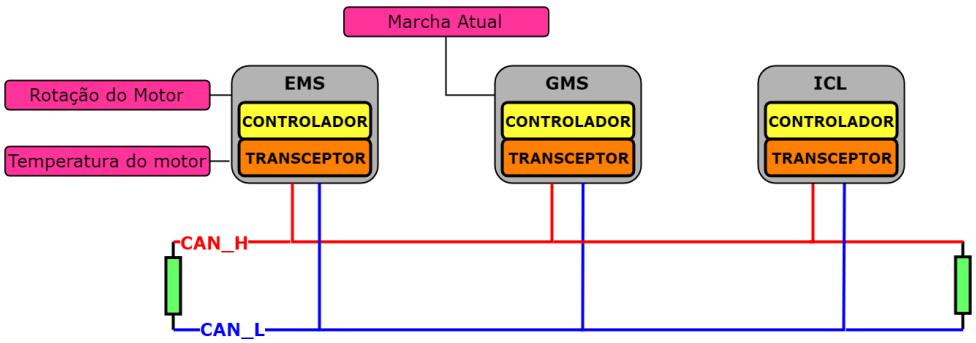


Figura 4.12: Rede automotiva do estudo de caso 2.

Através deste estudo de caso, pode-se verificar o comportamento da rede embarcada em um cenário mais complexo, com mais de uma ECU que envia informações pelo barramento e mais variáveis. As funcionalidades de *key handler* analógico e *message handler* do tipo *OnPGNName* também serão utilizadas e avaliadas, além das utilizadas no estudo de caso 1.

A Figura 4.13 apresenta o formato dos 3 diferentes *frames* utilizados no estudo de caso 2 para transmitir os parâmetros de rotação do motor, temperatura do líquido de arrefecimento do motor e marcha atual.

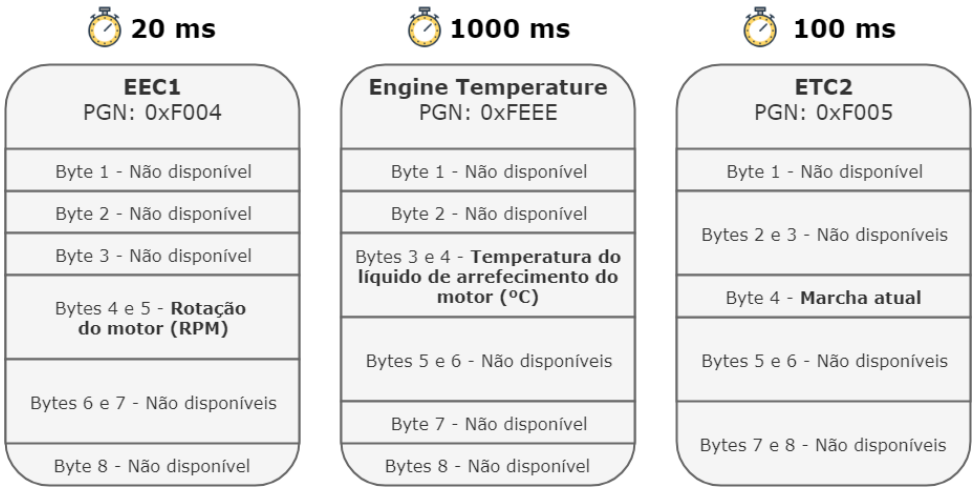


Figura 4.13: Formato das mensagens J1939 utilizadas no estudo de caso 2.

4.2.1 Desenvolvimento da rede automotiva virtual

Para o desenvolvimento da rede virtual do estudo de caso 2, utilizou-se o mesmo procedimento especificado no estudo de caso 1. O primeiro passo foi a criação do banco de dados com todas as mensagens no padrão J1939 que serão utilizadas no editor do Busmaster. A descrição das mensagens na ferramenta foi feita de acordo com a Figura 4.13.

O segundo passo foi a programação do comportamento das ECUs EMS, GMS e ICL no editor de código do Busmaster. O comportamento da EMS foi programado em termos das seguintes funcionalidades do Busmaster:

- **Variáveis globais** para definir as mensagens EEC1 e EngineTemperature que são transmitidas pela ECU. Além das constantes que representam os valores máximo, mínimo, e o valor de incremento das variáveis rotação do motor e temperatura do motor.

- **OnDLL_Load** para inicializar as mensagens com as prioridades adequadas e os valores iniciais das variáveis.

- **OnTimer** para enviar a mensagem EEC1 a cada 20 milissegundos. E outro para calcular o valor de temperatura e enviar a mensagem EngineTemperature a cada 1 segundo.

- **OnKey** para permitir incrementar ou decrementar a variável de rotação do motor. No sistema embarcado é utilizada leitura de um pino analógico para calcular o valor atual de rotação do motor.

A Figura 4.14 apresenta um diagrama contendo os fluxogramas das principais funcionalidades exercidas pela EMS.

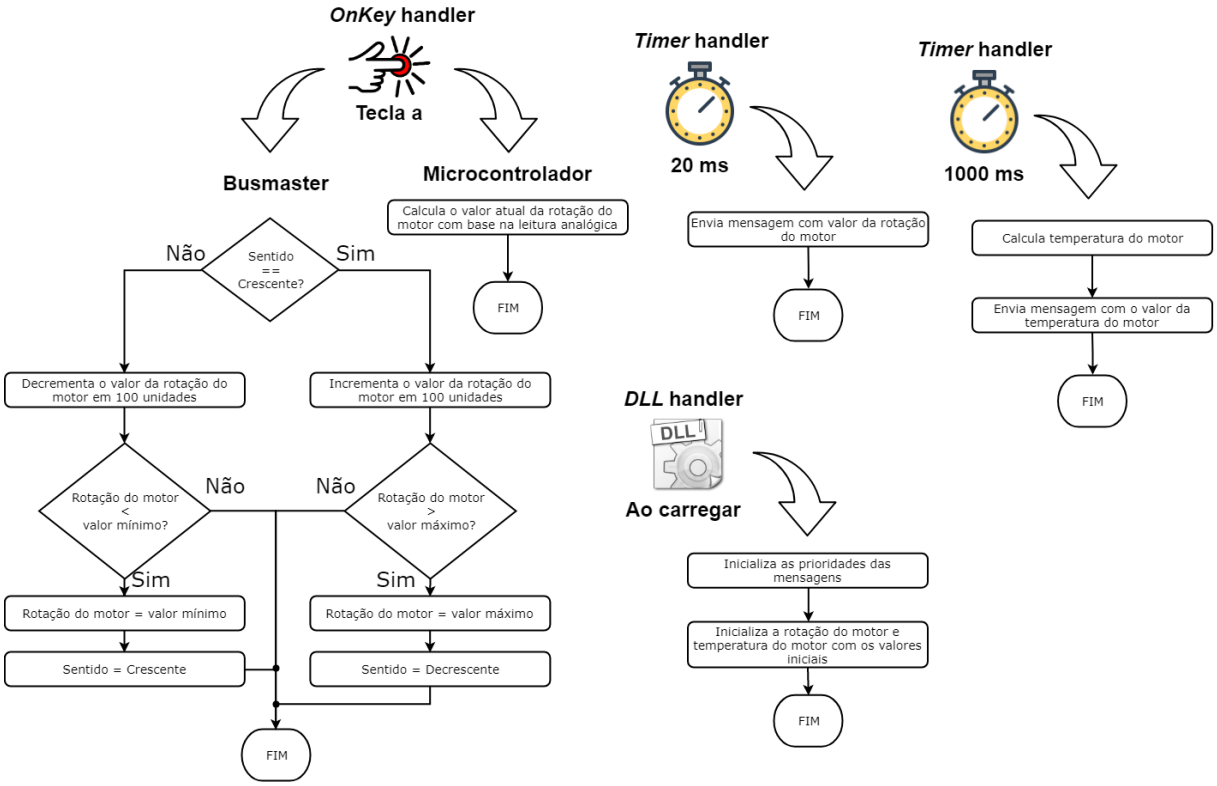


Figura 4.14: Diagramas com as funcionalidades da EMS.

Para a programação da GMS, as seguintes funcionalidades foram utilizadas:

- **Variáveis globais** para instanciar a mensagem ETC2 e as constantes de valor inicial, valor mínimo, valor máximo e o valor de incremento da variável marcha atual.
- **OnDLL_Load** para inicializar a mensagem com a prioridade adequada e o valor inicial da marcha atual.
- **OnTimer** para enviar a mensagem ETC2 a cada 100 milissegundos.
- **OnKey** para manipular o valor da marcha atual. Ao pressionar a tecla i do teclado, seu valor é incrementado de uma marcha, ao pressionar a tecla d, seu valor é decrementado de uma marcha.

A Figura 4.15 apresenta um diagrama contendo os fluxogramas das principais funcionalidades exercidas pela GMS.

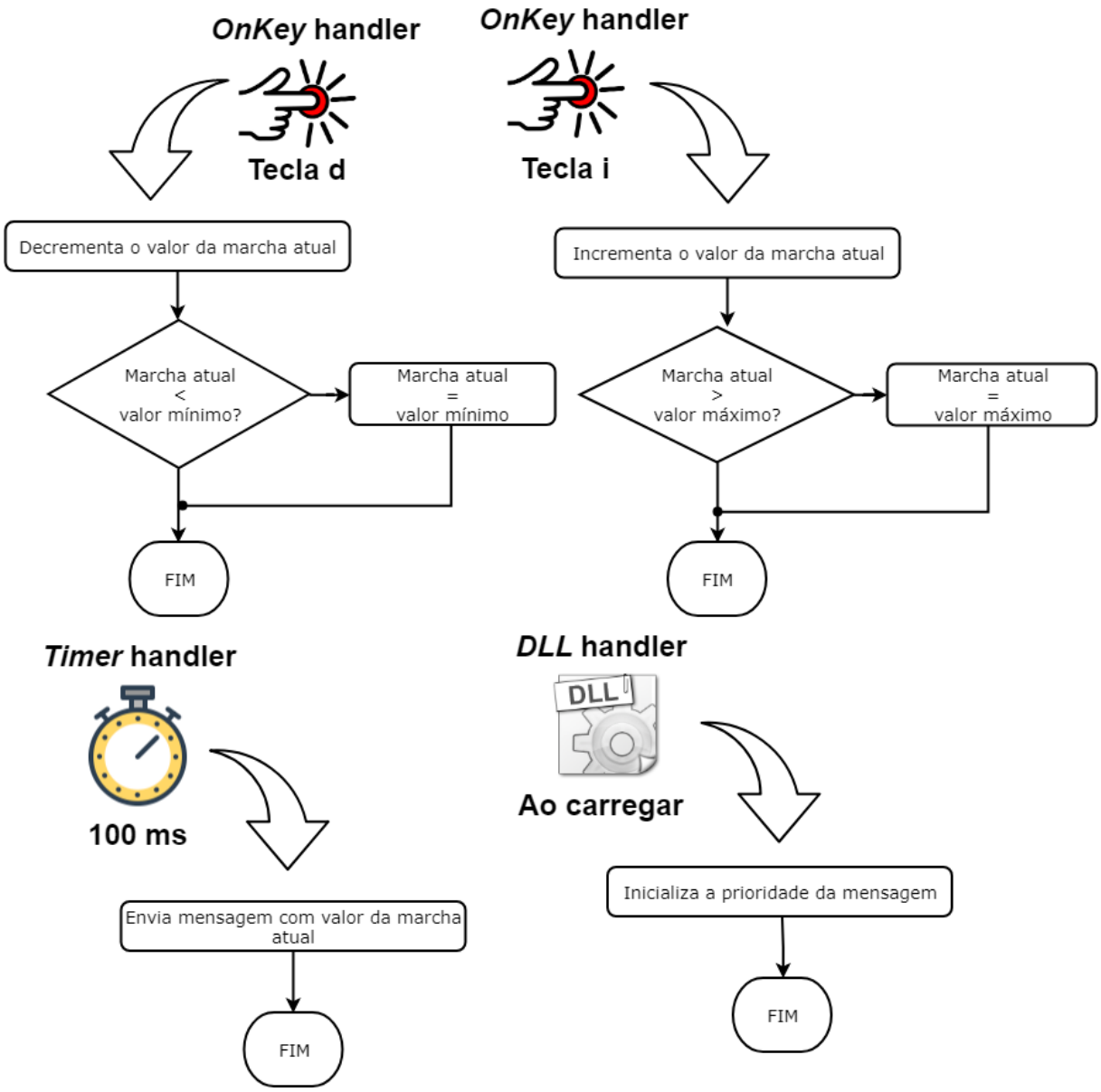


Figura 4.15: Diagramas com as funcionalidades da GMS.

Por fim, a última ECU, ICL, exerce o mesmo papel que a ECU2 no primeiro estudo de caso, isto é, monitora a rede quando em ambiente embarcado e não tem papel relevante no contexto do Busmaster. O seu comportamento é idêntico ao da ECU2 no primeiro estudo de caso com a o acréscimo da funcionalidade de enviar através da interface serial o valor real da variável sendo transmitida em cada mensagem. Para isso, faz uso de um *Message handler* do tipo *OnPGNName* que possui a diferença de fornecer como parâmetro para a função, a mensagem no formato original, facilitando a obtenção dos valores de cada sinal.

A Figura 4.16 apresenta um diagrama contendo os fluxogramas das principais funcionalidades exercidas pela ICL.

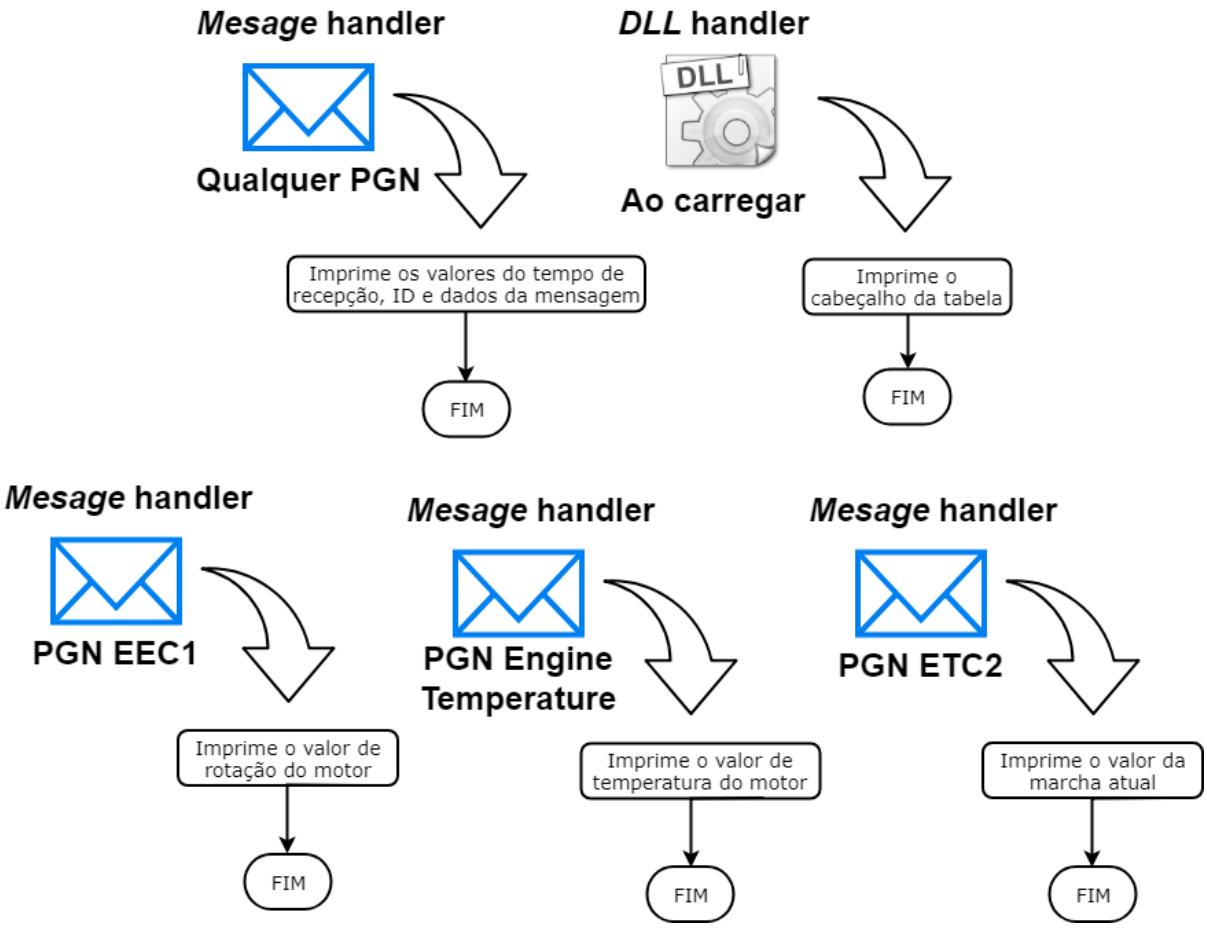
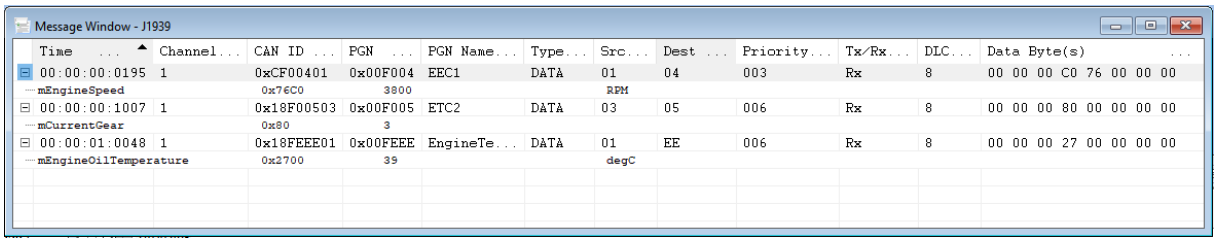


Figura 4.16: Diagramas com as funcionalidades da ICL.

A Figura 4.17 apresenta a tela de mensagens J1939 que trafegam pelo barramento CAN J1939 no ambiente de simulação do Busmaster. Pode-se verificar através da Figura 4.17 que o tempo relativo entre transmissões de cada uma das mensagens se comporta como esperado, isto é, para a mensagem EEC1 um intervalo aproximado de 20 milissegundos, para a mensagem Engine Temperature um intervalo de 1000 milissegundos e para a mensagem ETC2 um intervalo de 100 milissegundos. A prioridade das mensagens também confere com o programado em cada uma das ECUs.



Time	Channel	CAN ID	PGN	PGN Name	Type	Src	Dest	Priority	Tx/Rx	DLC	Data Byte(s)
00:00:00:0195	1	0xCF00401	0x00F004	EEC1	DATA	01	04	003	Rx	8	00 00 00 C0 76 00 00 00
mEngineSpeed		0x76C0	3800			RPM					
00:00:00:1007	1	0x18F00503	0x00F005	ETC2	DATA	03	05	006	Rx	8	00 00 00 80 00 00 00 00
mCurrentGear		0x80	3								
00:00:01:0048	1	0x18FEEE01	0x00FEEE	EngineTe...	DATA	01	EE	006	Rx	8	00 00 00 27 00 00 00 00
mEngineOilTemperature		0x2700	39			degC					

Figura 4.17: Tela de mensagens do barramento J1939 para o estudo de caso 2.

4.2.2 Conversão da rede virtual para rede embarcada

Nesta etapa o assistente Code Generator foi utilizado para realizar a transformação no código fonte de cada uma das ECUs para produzir os binários para a plataforma embarcada.

Para a EMS foi necessário mapear o *key handler* que manipula a variável rotação do motor para um pino analógico do microcontrolador e indicar através do assistente essa configuração. Isso permite a utilização de um potenciômetro para simular a mudança da rotação do motor. É necessário também indicar que a ECU transmite mensagens pelo barramento para habilitar essa funcionalidade no código produzido.

Para a GMS também é necessário mapear os *key handlers* para pinos no microcontrolador, porém, para pinos digitais ao invés de analógicos, para permitir o funcionamento utilizando botões da mesma forma como foi feito no estudo de caso 1. Também deve-se indicar que a ECU transmite mensagens CAN.

Por fim, para a ICL, que exerce o papel de monitorar a rede e exibir os dados para o usuário, basta indicar que a ECU utiliza a interface serial.

A Figura 4.18 apresenta três telas do Code Generator. A tela de configuração dos pinos para a EMS no canto superior esquerdo. Nota-se a configuração de um pino analógico. A tela de configuração dos pinos para a GMS no canto superior direito. Neste caso, a tela apresenta a configuração de dois pinos digitais ativos em nível lógico baixo. E por fim, a tela de configuração do código gerado para a plataforma para a ICL. Neste caso, é indicado na tela que a ECU faz uso da interface serial e o endereço $0x17$ é associado a ECU.

4.2.3 Análise dos resultados

As Figuras 4.19 a 4.20 apresentam algumas informações a respeito das mensagens recebidas pela ICL e impressas através de sua interface serial no monitor serial da IDE do arduino. Observa-se que os intervalos entre as mensagens do mesmo id correspondem ao esperado, pois a verifica-se um período de aproximadamente 20 milissegundos entre mensagens do tipo *EEC1*, 100 milissegundos entre mensagens do tipo *ETC2* e 1 segundo entre mensagens do tipo *Engine Temperature*.

O experimento possibilitou o controle da variável de rotação do motor através do uso do potenciômetro e da variável de marcha atual através de dois botões para incrementar e decrementar o valor da marcha. Também foi possível verificar o crescimento do valor da variável de temperatura de arrefecimento do motor conforme esperado.

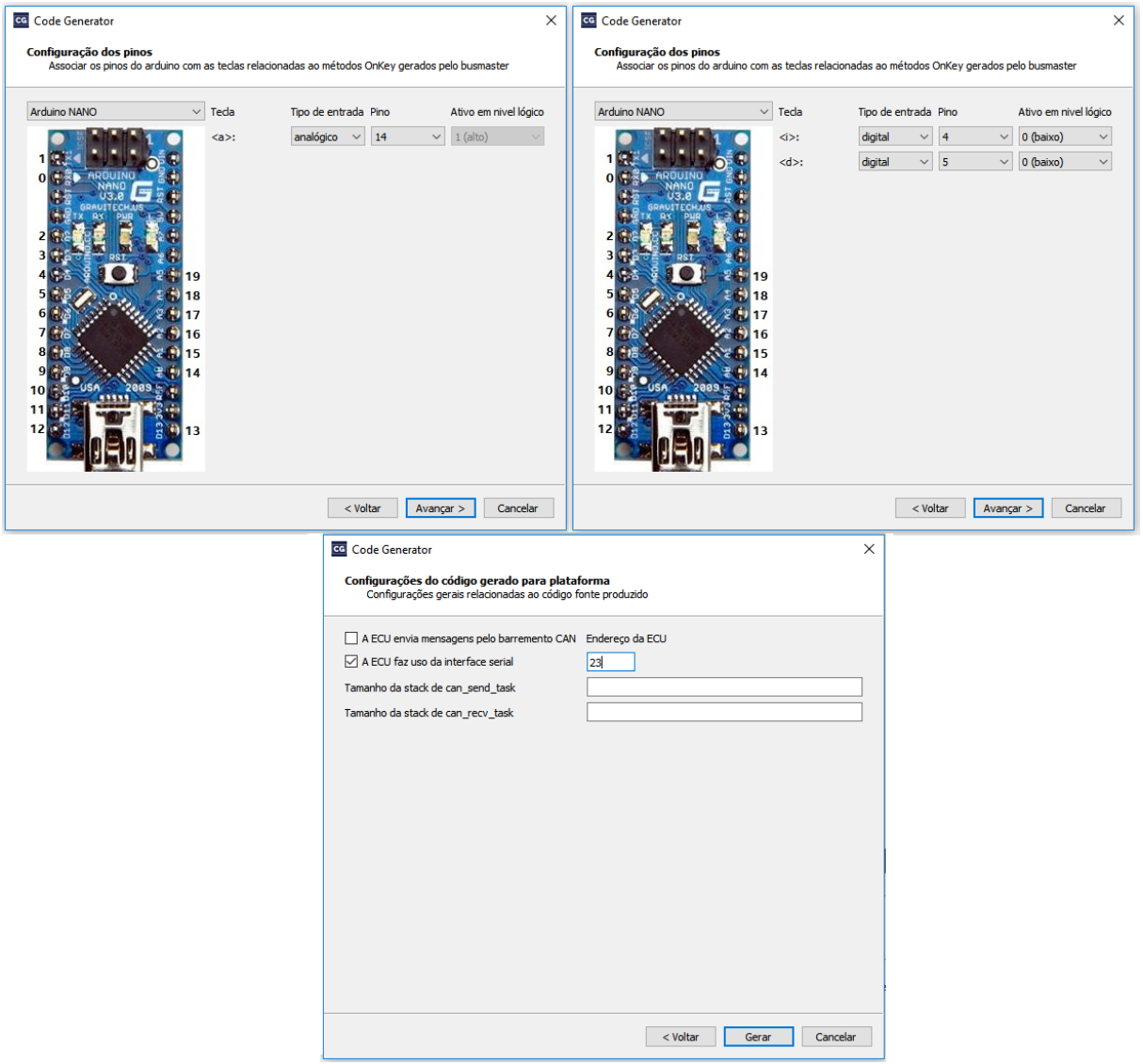


Figura 4.18: Telas do Code Generator de configuração dos pinos para a EMS no canto superior esquerdo e GMS no canto superior direito. E tela do Code Generator da configuração do código gerado para a plataforma para a ICL no canto inferior..

Nota-se que a ICL pôde capturar corretamente as mensagens que trafegaram no barramento CAN e extrair os valores de cada uma das variáveis para exibir os corretos valores através da interface serial.

Conclui-se que os resultados obtidos no estudo de caso 2 se mostraram satisfatórios, uma vez que foram muito similares aos observados no ambiente de simulação do Busmaster. Os intervalos entre as mensagens permaneceram dentro da faixa de tempo esperada.

Os *handlers*: *timer handler*, *DLL handler*, *key handler* com mapeamento para pino digital e analógico, e *message handler* dos tipos *OnPGN_All* e *OnPGNName* puderam ser verificados nesse estudo de caso.

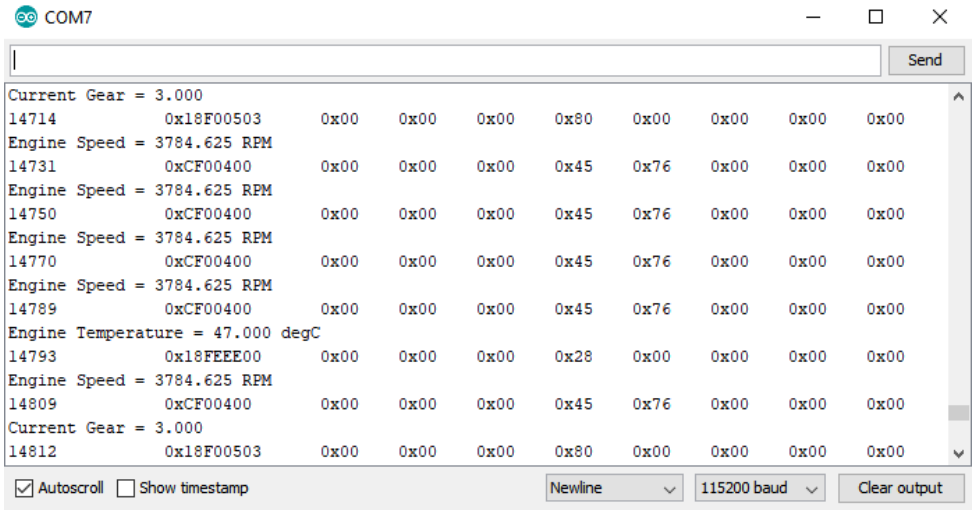


Figura 4.19: Tela do monitor serial da IDE do arduino. A imagem revela as informações obtidas através da ICL sobre as mensagens que trafegam no barramento CAN.

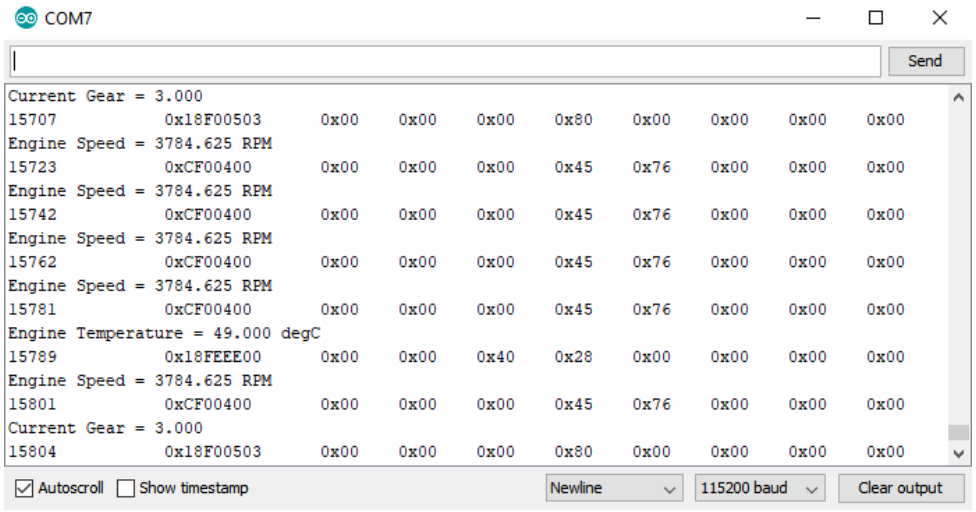


Figura 4.20: Tela do monitor serial da IDE do arduino com uma diferença de aproximadamente 1 segundo em relação a Figura 4.19.

4.3 Estudo de Caso 3

O último estudo de caso consiste de uma rede automotiva com um grau muito maior de complexidade. Nesta rede, 4 ECUs estão presentes: EMS, BMS (do inglês, *Brake Management System*, Sistema de Gerenciamento de Freio), GMS e ICL. Assim como no estudo de caso 2, a nomenclatura das ECUs utilizada aqui é baseada na documentação da Scania.

Nesta rede, a BMS é responsável pelo controle do sistema de frenagem do veículo. A

GMS é responsável pelo controle do sistema de transmissão. A EMS realiza a leitura de valores de posição do pedal de aceleração, a recepção dos valores das variáveis enviadas pelas ECUs BMS e GMS e implementa um modelo dinâmico de um veículo para calcular valores para as variáveis de velocidade do veículo, nível de combustível, rotação do motor, temperatura do líquido de arrefecimento e distância percorrida. Todas as variáveis tratadas pela EMS também são transmitidas pelo barramento. Por fim, a ICL é responsável pela recepção e monitoramento de todas as mensagens que trafegam pela rede.

A Figura 4.21 ilustra a rede automotiva do estudo de caso 3 com todas as ECUs e variáveis do sistema.

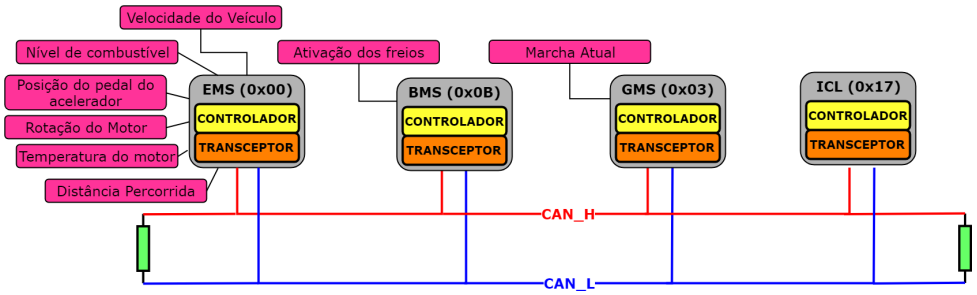


Figura 4.21: Rede automotiva do estudo de caso 3.

Com este estudo de caso, podem ser testados os *handlers*: *timer handler*, *key handler* dos tipos digital e analógico, *DLL handler*, *Message handlers* do tipo *PGNName handler* como no estudo de caso 2. Além disso, também é possível verificar o comportamento da rede quando existe um número maior de mensagens que trafegam o barramento. Outro aspecto importante a analisar neste estudo de caso, é a sobrecarga de tarefas realizada pela a EMS e a correta execução de todas elas através da plataforma embarcada.

A Figura 4.22 ilustra o formato das mensagens utilizadas no estudo de caso 3. Todas elas são baseadas no formato das mensagens descritas na norma SAE J1939. Apenas os sinais utilizados no estudo de caso 3, estão representados na figura. Os intervalos de tempo entre transmissões de cada mensagem estão indicados acima de cada uma delas na figura.

A Figura 4.23 apresenta um diagrama que ilustra, com abstrações para simplificação, a dinâmica de controle realizada pela EMS e a relação entre as variáveis do sistema. Observa-se que existem apenas 3 variáveis que servem apenas de entrada no sistema, a posição do pedal do acelerador, a ativação dos freios e a marcha atual.

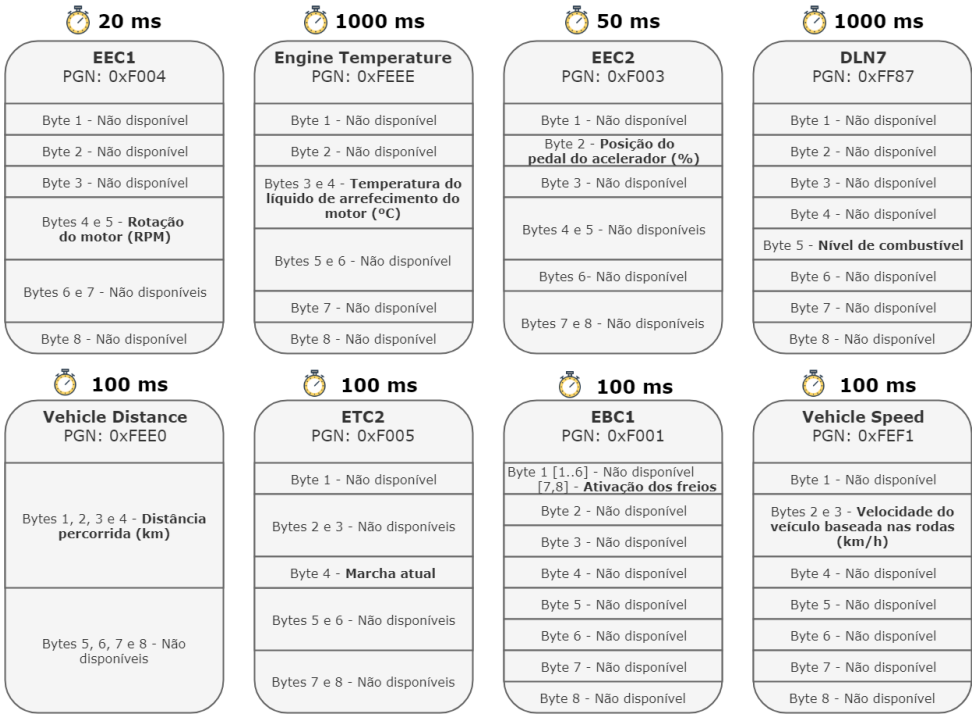


Figura 4.22: Formato das mensagens J1939 utilizadas no estudo de caso 3.

4.3.1 Desenvolvimento da rede automotiva virtual

O mesmo procedimento realizado nos outros dois estudos de caso, foi seguido no estudo de caso 3. Iniciando com a criação das mensagens J1939 no banco de dados do Busmaster. A Figura 4.22 ilustra o formato utilizado para descrição de cada uma delas no editor do Busmaster.

O passo seguinte é a programação do comportamento das ECUS. A Figura 4.24 ilustra um diagrama que representa o comportamento da GMS.

Para a descrição do comportamento da GMS, as seguintes funcionalidades do Busmaster foram utilizadas:

- **Variáveis globais** para instanciar a mensagem ETC2 e as constantes de valor inicial, valor mínimo, valor máximo e o valor de incremento da variável marcha atual.
- **OnDLL_Load** para inicializar a mensagem com a prioridade adequada e o valor inicial da marcha atual.
- **OnTimer** para enviar a mensagem ETC2 a cada 100 milissegundos.

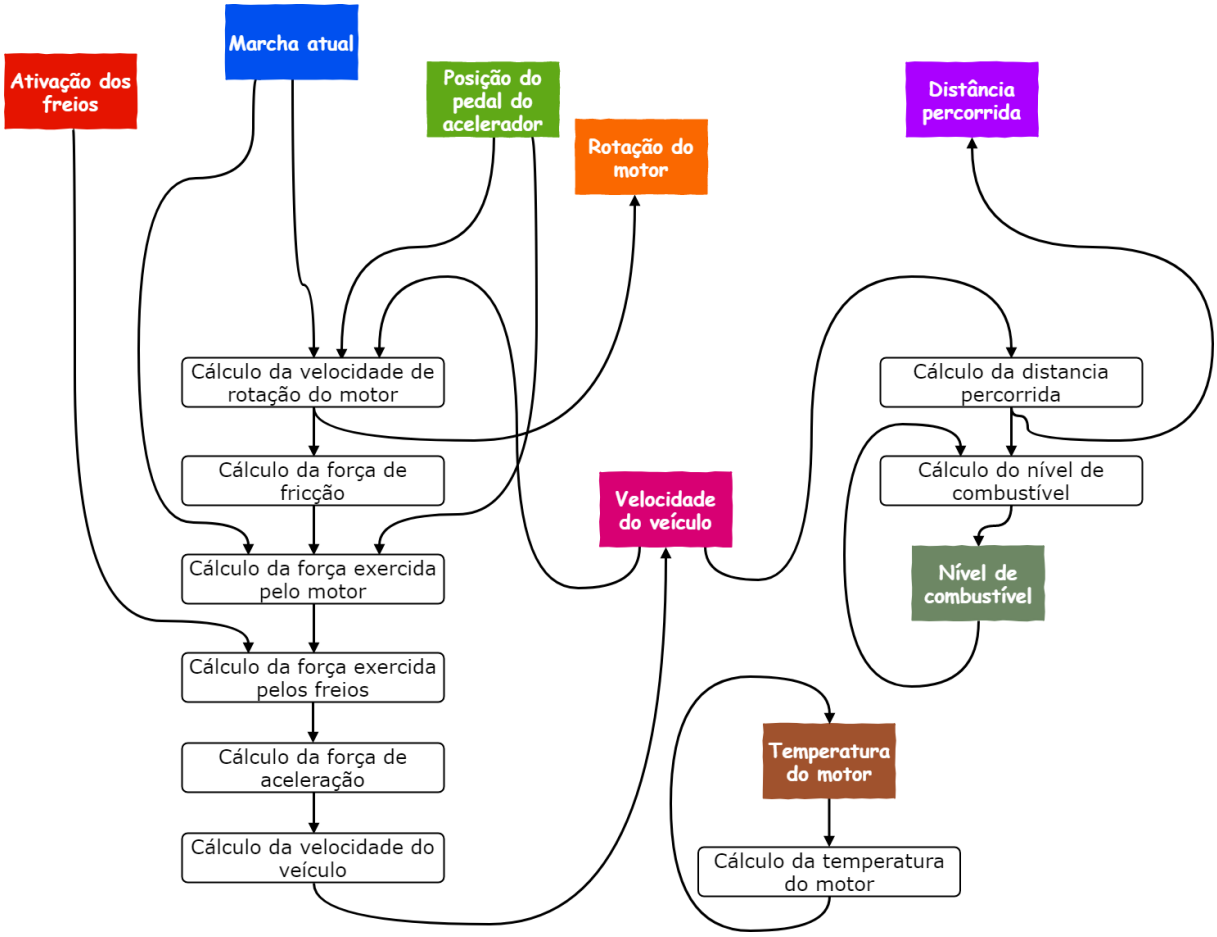


Figura 4.23: Diagrama da dinâmica veicular do estudo de caso 3. As variáveis do sistema aparecem em caixas coloridas. As caixas brancas indicam um processamento realizado pela EMS. A seta saindo de uma variável indica que ela está servindo de entrada e a seta entrando em uma caixa contendo uma variável indica que ela é o resultado de um processamento.

- **OnKey** para manipular o valor da marcha atual. Ao pressionar a tecla i do teclado, seu valor é incrementado de uma marcha, ao pressionar a tecla d, seu valor é decrementado de uma marcha.

A BMS segue o comportamento descrito pela Figura 4.25. No Busmaster, seu comportamento foi descrito em termos das funcionalidades:

- **Variáveis globais** para instanciar a mensagem EBC1 e o valor atual do tipo booleano que indica o estado dos freios, ativados ou desativados.
- **OnDLL_Load** para inicializar a mensagem com a prioridade adequada.
- **OnTimer** para enviar a mensagem EBC1 a cada 100 milissegundos.

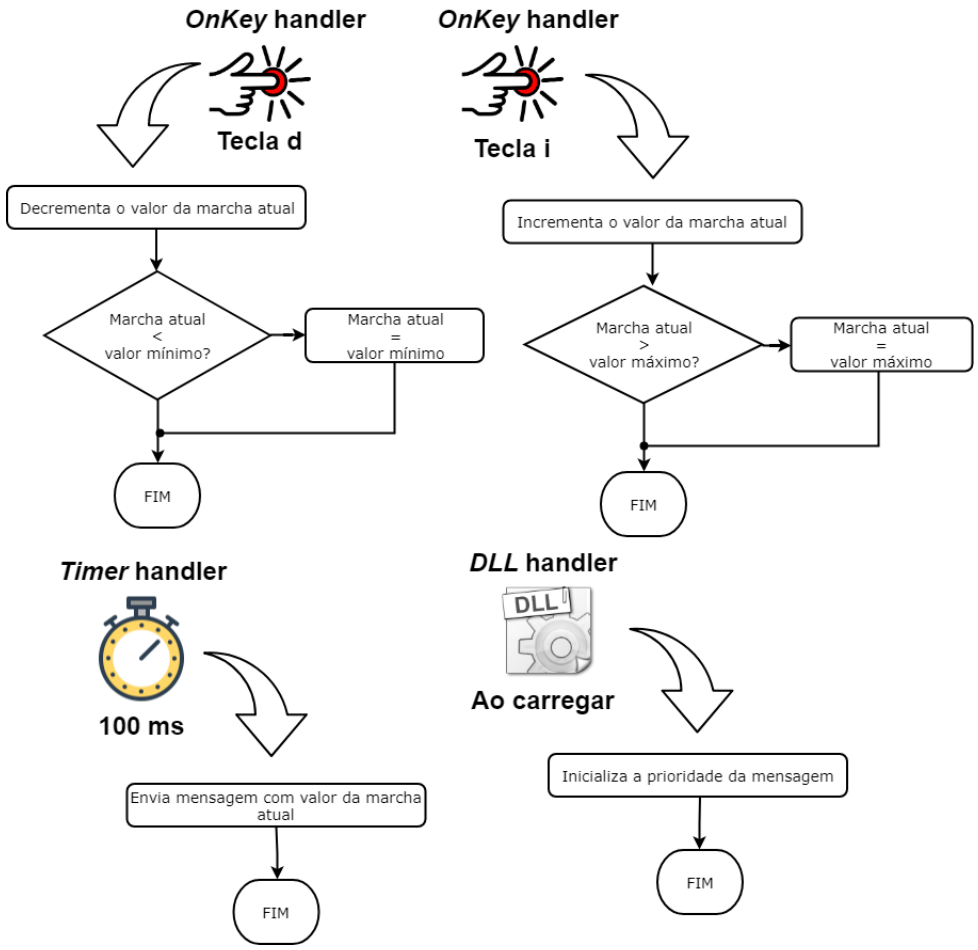


Figura 4.24: Diagramas com as funcionalidades da GMS.

- **OnKey** para permitir que o usuário altere o estado dos freios ao pressionar a tecla b.

A ICL implementa o comportamento descrito pela Figura 4.26. Novamente nesse estudo de caso, a função da ICL é de realizar o monitoramento da rede, quando a simulação ocorrer em ambiente embarcado. No contexto de simulação da rede virtual no Busmaster, a ECU não exerce papel importante para a rede.

A ICL, nesse estudo de caso, foi programada duas vezes. A primeira foi para a geração de uma planilha em formato CSV, através do envio pela interface serial de mensagens formatadas como linhas da tabela. Facilitando assim, a obtenção de estatísticas da rede a partir do arquivo gerado.

O segundo programa executado pela ICL, tem como objetivo a utilização do software Instrument Cluster para a exibição em tempo real das informações transmitidas em uma interface gráfica semelhante a um painel de um automóvel. A função da ICL continua

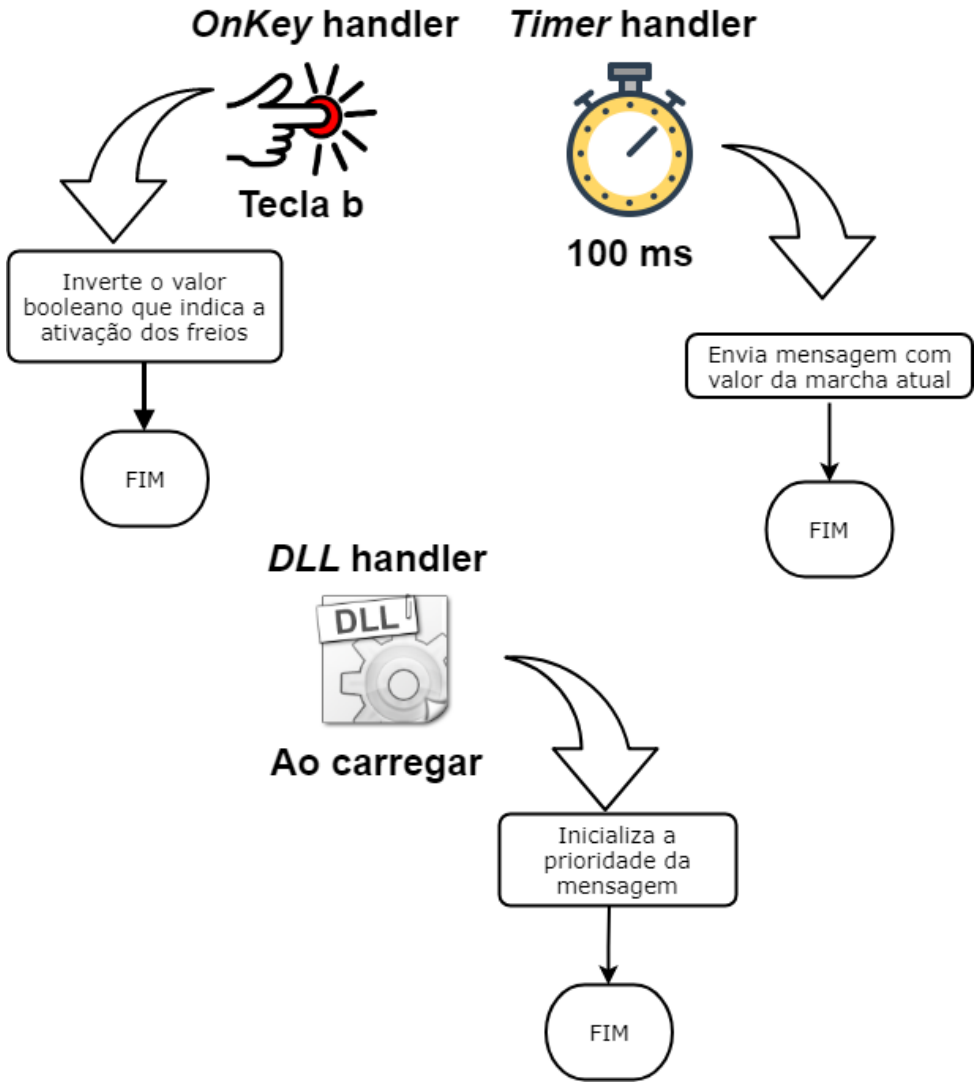


Figura 4.25: Diagramas com as funcionalidades da BMS.

sendo a transmissão pela interface serial das informações trafegadas na rede porém, seguindo o protocolo estabelecido pelo software Instrument Cluster.

Por fim, a descrição do comportamento da EMS em termos das funcionalidades do Busmaster está representada pela Figura 4.27. Para a implementação do comportamento esperado para a ECU utilizou-se as seguintes funcionalidades:

- **Variáveis globais** para a instanciação de todas variáveis de controle que representam estado e das mensagens EEC1 e EEC2. Também foram utilizadas diversas constantes globais.

- **OnKey** para permitir incrementar ou decrementar a variável de posição do pedal do acelerador. No sistema embarcado é utilizada a leitura de um pino analógico

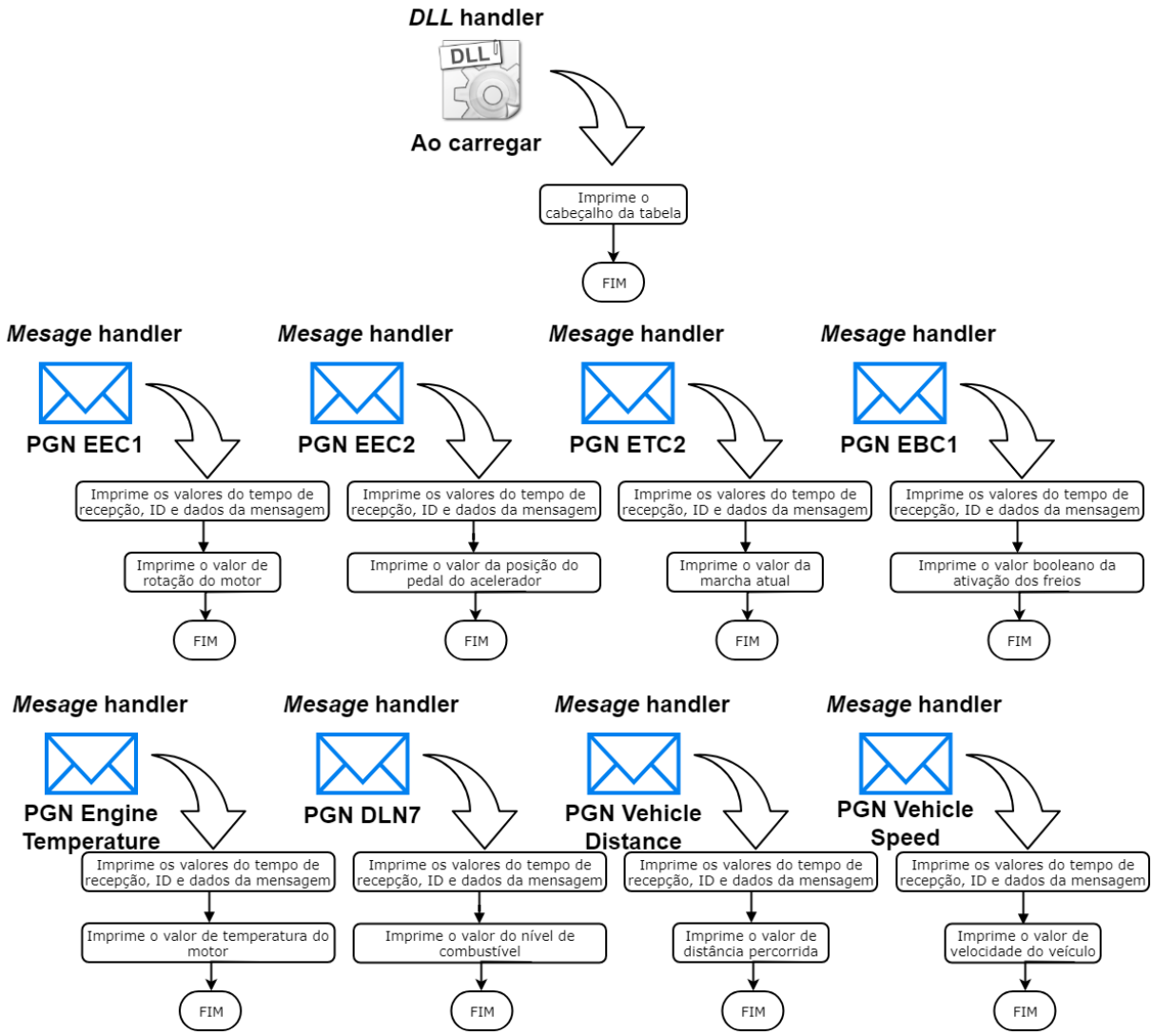


Figura 4.26: Diagramas com as funcionalidades da ICL.

para para calcular o valor atual de posição do pedal do acelerador.

- **OnDLL_Load** para inicializar as prioridades das mensagens EEC1 e EEC2.
- **OnPGNName** para realizar a recepção das mensagens EBC1 e ETC2 e extrair os valores de ativação dos freios e marcha atual para as variáveis globais que as representam.
- **OnTimer** para realizar as operações que ocorrem entre intervalos de tempo específicos como envio das mensagens Vehicle Distance e Vehicle Speed a cada 100 milissegundos, o cálculo e envio da mensagem Engine Temperature e o envio da mensagem DLN7 que ocorrem a cada 1000 milissegundos, o envio da mensagem EEC1 que ocorre a cada 20 milissegundos, o envio da mensagem EEC2 que ocorre a

cada 50 milissegundos, o cálculo do nível de combustível e distância percorrida que ocorre em tempo variável e múltiplo de 5 milissegundos e o cálculos da dinâmica veicular que ocorrem a cada 10 milissegundos.

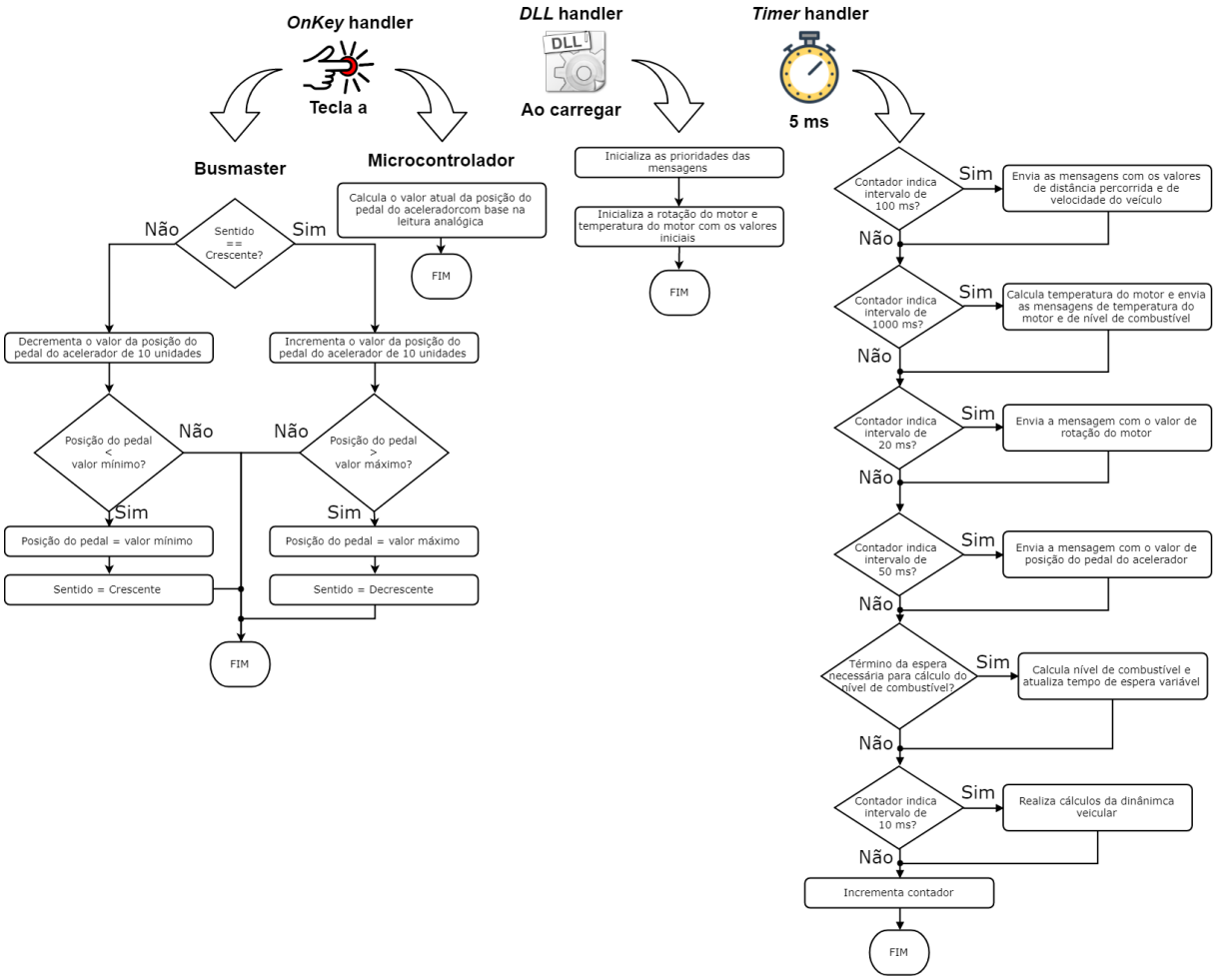
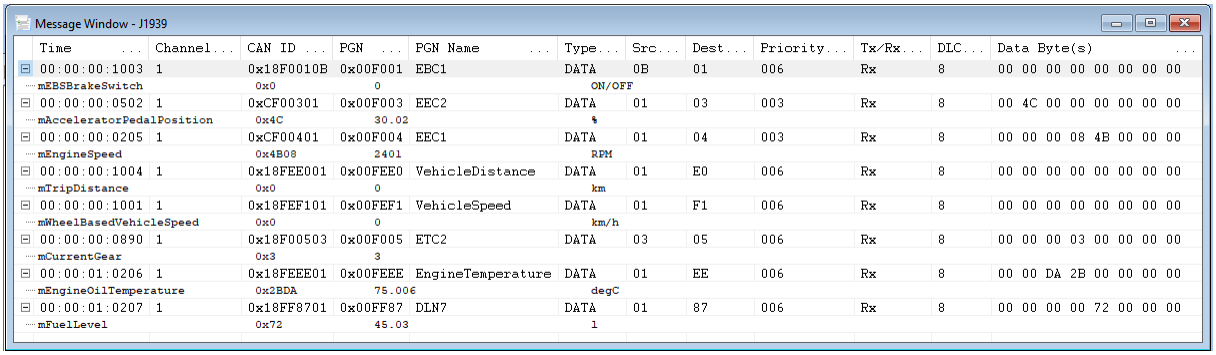


Figura 4.27: Diagramas com as funcionalidades da EMS.

É importante notar na Figura 4.27 que diversas funções baseadas em eventos temporais foram descritos em termos de um único *timer handler*. O código da ECU EMS foi projetado dessa forma tendo em vista características da implementação do código gerado pelo software Code Generator para a plataforma embarcada e as restrições de espaço encontradas no microcontrolador. Ao utilizar um único *timer handler* todas essas funções podem utilizar da mesma thread de execução e portanto da mesma pilha de execução, evitando problemas como estouro da pilha por falta de espaço disponível. O código da ECU EMS sofreu diversas alterações de forma a reduzir o espaço utilizado pelo programa e minimizar os impactos na performance.

O último passo no desenvolvimento da rede automotiva do estudo de caso 3 foi a análise da rede no ambiente de simulação do Busmaster. A Figura 4.28 apresenta a tela de mensagens J1939 do software, onde verifica-se os valores de intervalo de tempo relativos de cada mensagem, os valores reais de cada variável, os IDs de cada mensagem e a disposição dos bytes de dados de cada uma delas.



Time	Channel	CAN ID	PGN	PGN Name	Type	Src	Dest	Priority	Tx/Rx	DLC	Data Byte(s)
00:00:00:1003	1	0x18F0010B	0x00F001	EBC1	DATA	0B	01	006	Rx	8	00 00 00 00 00 00 00 00
mEBSBrakeSwitch		0x0	0		ON/OFF						
00:00:00:0502	1	0xCF00301	0x00F003	EEC2	DATA	01	03	003	Rx	8	00 4C 00 00 00 00 00 00
mAcceleratorPedalPosition		0x4C	30.02		%						
00:00:00:0205	1	0xCF00401	0x00F004	EEC1	DATA	01	04	003	Rx	8	00 00 00 08 4B 00 00 00
mEngineSpeed		0x4B08	2401		RPM						
00:00:00:1004	1	0x18FEE001	0x00FEE0	VehicleDistance	DATA	01	E0	006	Rx	8	00 00 00 00 00 00 00 00
mTripDistance		0x0	0		km						
00:00:00:1001	1	0x18FEF101	0x00FEF1	VehicleSpeed	DATA	01	F1	006	Rx	8	00 00 00 00 00 00 00 00
mWheelBasedVehicleSpeed		0x0	0		km/h						
00:00:00:0890	1	0x18F00503	0x00F005	ETC2	DATA	03	05	006	Rx	8	00 00 00 03 00 00 00 00
mCurrentGear		0x3	3								
00:00:01:0206	1	0x18FEEE01	0x00FEEE	EngineTemperature	DATA	01	EE	006	Rx	8	00 00 DA 2B 00 00 00 00
mEngineOilTemperature		0x2BDA	75.006		degC						
00:00:01:0207	1	0x18FF8701	0x00FF87	DLN7	DATA	01	87	006	Rx	8	00 00 00 00 72 00 00 00
mFuelLevel		0x72	45.03		l						

Figura 4.28: Tela de mensagens do barramento J1939 para o estudo de caso 3.

O resultado obtido nessa etapa não correspondeu ao esperado. Apesar dos tempos de transmissão de cada mensagem se manterem na faixa adequada, houveram problemas na recepção de mensagens ETC2 por parte de ECU EMS o que pode indicar alguma circunstância relacionada a implementação da funcionalidade *OnPGNName* do Busmaster que não foi tratada.

4.3.2 Conversão da rede virtual para rede embarcada

Nesta etapa o assistente Code Generator foi utilizado para realizar a transformação no código fonte de cada uma das ECUs para produzir os binários para a plataforma embarcada.

Para a EMS foi necessário mapear o *key handler* que manipula a variável de posição do pedal do acelerador para um pino analógico do microcontrolador de forma a permitir a utilização de um potenciômetro. Ainda para a EMS, foi necessário a realização de vários testes para fazer a escolha correta dos tamanhos das pilhas de execução de cada thread de forma a evitar problemas de estouro da pilha.

Para a GMS também foi necessário mapear os *key handlers* para pinos no microcontrolador, porém, para pinos digitais ao invés de analógicos, para permitir o funcionamento utilizando botões com circuito em configuração de pull-up.

Para a BMS também foi necessário mapear o *key handler* para um pino digital no microcontrolador de forma a permitir ativar e desativar os freios através de um botão.

Por fim, para a ICL, que exerce o papel de monitor da rede, deve-se indicar a utilização da interface serial. A escolha de um tamanho de pilha adequado para cada uma das threads de execução, também foi uma etapa importante para essa ECU.

A Figura 4.29 apresenta algumas telas do Code Generator que evidenciam o uso do mesmo. No canto superior esquerdo observa-se a tela de configuração dos pinos para a BMS. A configuração de um pino digital ativo em nível lógico baixo é observada. A tela na posição superior central da Figura 4.29 é a de configuração dos pinos para a GMS, onde a configuração de dois pinos digitais ativos em nível lógico baixo é exibida. No canto superior direito encontra-se a tela de configuração do código gerado para a plataforma referente a ICL. O endereço 23 escolhido para a ICL, a marcação da opção que indica a utilização da interface serial e a configuração do tamanho da pilha da tarefa *can_recv_task* com o valor de 512 é observado. As duas telas abaixo das demais são, da esquerda para a direita, a tela de configuração dos pinos e a tela de configuração do código gerado para a plataforma, ambas referentes a EMS. Nota-se nestas duas telas, a configuração de um pino analógico, a configuração do endereço 0, a marcação da opção que habilita o envio de mensagens CAN e o valor da pilha da tarefa *OnTimer_TimeBasedLogic* de 256.

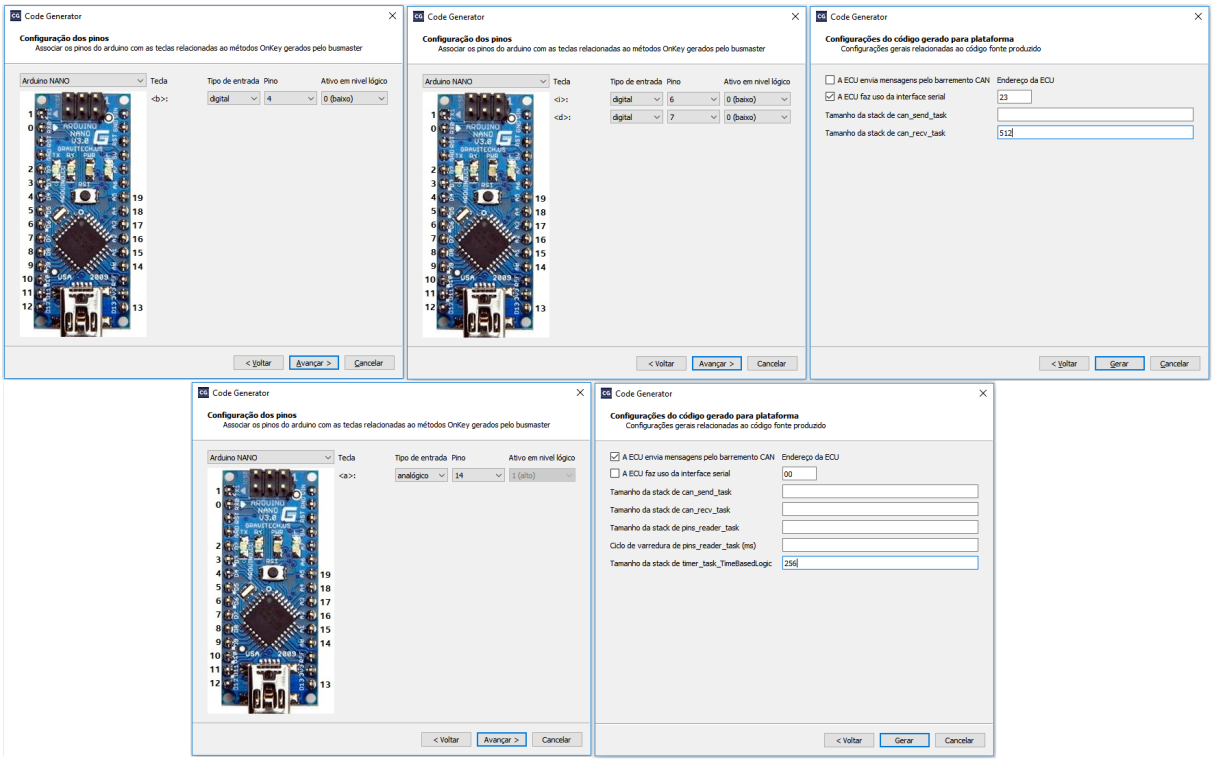


Figura 4.29: Telas do Code Generator para o estudo de caso 3.

A Figura 4.30 apresenta a rede embarcada utilizada. Observa-se a presença dos microcontroladores que implementam cada uma das 4 ECUs, dos botões para incrementar

e decrementar a marcha, do potenciômetro para regular a posição do pedal do acelerador e do botão que ativa e desativa os freios.

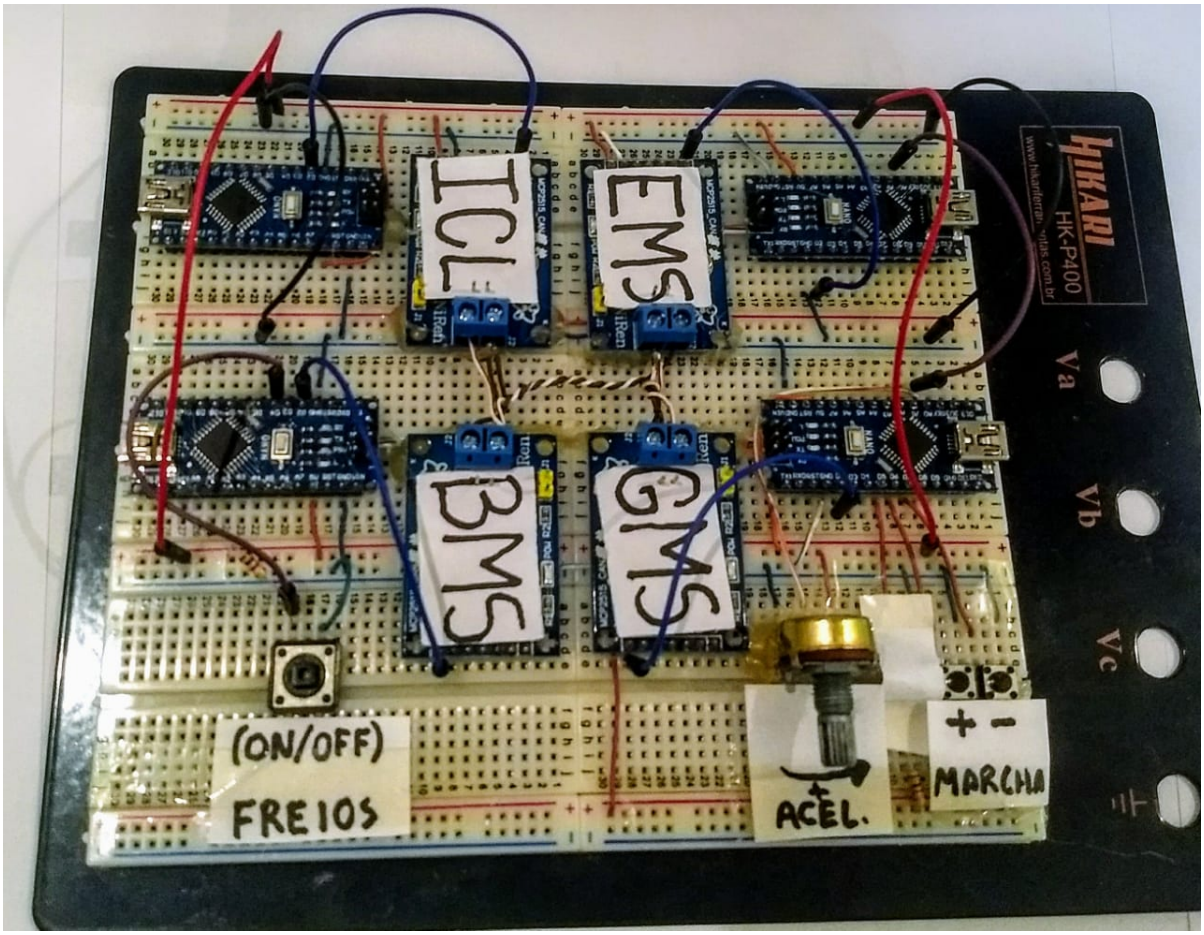


Figura 4.30: Foto da rede embarcada utilizada no estudo de caso 3.

4.3.3 Análise dos resultados

A Figura 4.31 apresenta uma captura da tela do Instrument Cluster quando a rede embarcada do estudo de caso 3 estava em execução. Através do Instrument Cluster pode-se observar em tempo real o funcionamento da rede embarcada para o estudo de caso 3.

A Figura 4.32 apresenta as linhas iniciais da tabela gerada a partir das mensagens transmitidas pela ICL. Pode-se observar a presença dos campos de tempo em milissegundos, id da mensagem, disposição dos bytes de dados e o valor das informações específicas de cada mensagem.

A Figura 4.33 revela momentos após a regulação das variáveis de posição do pedal do acelerador e marcha atual através do potenciômetro e dos botões. É possível notar que no momento observado, a posição do pedal do acelerador na ante-penúltima coluna

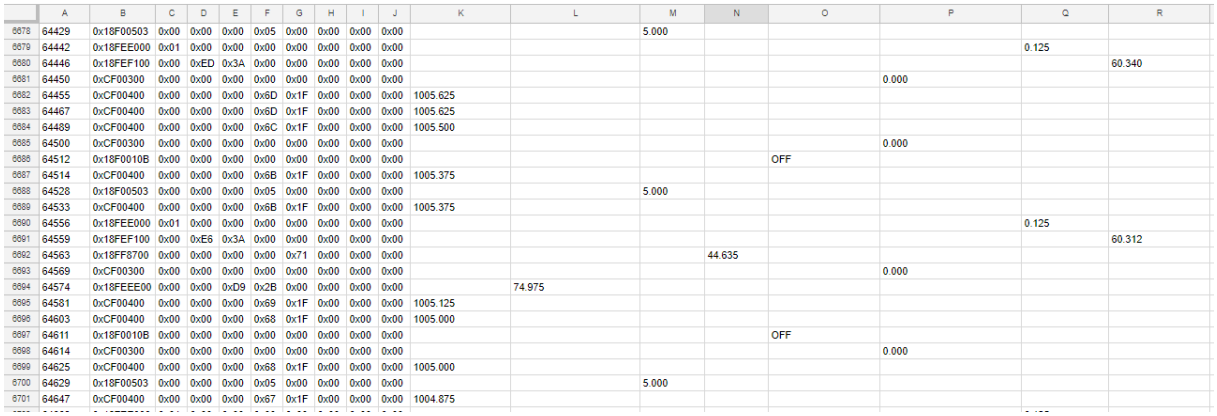
Figura 4.31: Tela do software Instrument Cluster exibindo os dados recebidos da ICL para o estudo de caso 3.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Time (ms)	Id	Byte0	Byte1	Byte2	Byte3	Byte4	Byte5	Byte6	Byte7	Engine Speed (RPM)	Engine Temperature (degC)	Current Gear	Fuel Level (l)	Brake Switch (ON/OFF)	Accelerator Pedal Position (%)	Trip Distance (km)	Vehicle Speed (km/h)
2	2	0xCF00300	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00						0.000		
3	7	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
4	17	0x18F0010B	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00					OFF			
5	20	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
6	40	0x18FEE000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00							0.000	
7	44	0x18FEF100	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00								0.000
8	48	0xCF00300	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00						0.000		
9	53	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
10	61	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
11	78	0x18F00503	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00			0.000					
12	81	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
13	92	0xCF00300	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00						0.000		
14	102	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
15	115	0x18F0010B	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00					OFF			
16	122	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
17	143	0x18FEE000	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00							0.000	
18	146	0x18FEF100	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00								0.000
19	150	0xCF00300	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00						0.000		
20	155	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
21	163	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
22	178	0x18F00503	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00			0.000					
23	184	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
24	194	0xCF00300	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00						0.000		
25	204	0xCF00400	0x00	0x00	0x00	0xC0	0x12	0x00	0x00	0x00	600.000							
26	216	0x18F0010B	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00					OFF			

Figura 4.32: Planilha gerada com as informações transmitidas pela ECU ICL.

se encontrava em 0% e a velocidade apresentada sofria uma redução gradual. Também é possível notar a transmissão de valores de cada uma das mensagens na figura.

Através da planilha produzida, as estatísticas de média e desvio padrão dos intervalos



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
6678	64429	0x18F00503	0x00	0x00	0x00	0x05	0x00	0x00	0x00	0x00			5.000					
6679	64442	0x18FEE000	0x01	0x00	0x00	0x00	0x00	0x00	0x00	0x00							0.125	
6680	64446	0x18FEF100	0x00	0xED	0x3A	0x00	0x00	0x00	0x00	0x00								60.340
6681	64450	0xCF00300	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00						0.000		
6682	64455	0xCF00400	0x00	0x00	0x00	0x6D	0x1F	0x00	0x00	0x00	1005.625							
6683	64467	0xCF00400	0x00	0x00	0x00	0x6D	0x1F	0x00	0x00	0x00	1005.625							
6684	64489	0xCF00400	0x00	0x00	0x00	0x6C	0x1F	0x00	0x00	0x00	1005.500							
6685	64500	0xCF00300	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00						0.000		
6686	64512	0x18F0010B	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00					OFF			
6687	64514	0xCF00400	0x00	0x00	0x00	0x6B	0x1F	0x00	0x00	0x00	1005.375							
6688	64528	0x18F00503	0x00	0x00	0x00	0x05	0x00	0x00	0x00	0x00			5.000					
6689	64533	0xCF00400	0x00	0x00	0x00	0x6B	0x1F	0x00	0x00	0x00	1005.375							
6690	64556	0x18FEE000	0x01	0x00	0x00	0x00	0x00	0x00	0x00	0x00							0.125	
6691	64559	0x18FEF100	0x00	0xE6	0x3A	0x00	0x00	0x00	0x00	0x00								60.312
6692	64563	0x18FF8700	0x00	0x00	0x00	0x00	0x71	0x00	0x00	0x00				44.635				
6693	64569	0xCF00300	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00						0.000		
6694	64574	0x18FEEE00	0x00	0x00	0xD9	0x2B	0x00	0x00	0x00	0x00		74.975						
6695	64581	0xCF00400	0x00	0x00	0x00	0x69	0x1F	0x00	0x00	0x00	1005.125							
6696	64603	0xCF00400	0x00	0x00	0x00	0x68	0x1F	0x00	0x00	0x00	1005.000							
6697	64611	0x18F0010B	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00					OFF			
6698	64614	0xCF00300	0x00	0x00	0x00	0x00	0x00	0x00	0x00	0x00						0.000		
6699	64625	0xCF00400	0x00	0x00	0x00	0x68	0x1F	0x00	0x00	0x00	1005.000							
6700	64629	0x18F00503	0x00	0x00	0x00	0x05	0x00	0x00	0x00	0x00			5.000					
6701	64647	0xCF00400	0x00	0x00	0x00	0x67	0x1F	0x00	0x00	0x00	1004.875							

Figura 4.33: Captura de linhas da planilha que representam momentos depois do apresentado na Figura 4.32.

Tabela 4.1: Estatísticas sobre resultados da rede embarcada do estudo de caso 3.

Mensagem	**Intervalo estipulado**	**Número de Amostras**	**Média Aritmética**	**Desvio Padrão**
EEC1	20 ms	8892	22.170 ms	8.784 ms
Engine Temperature	1000 ms	181	1084.403 ms	24.191 ms
ETC2	100 ms	1980	99.507 ms	5.824 ms
DLN7	1000 ms	157	1250.166 ms	431.935 ms
EBC1	100 ms	1974	99.828 ms	7.467 ms
EEC2	50 ms	3631	54.286 ms	7.469 ms
Vehicle Distance	100 ms	1817	108.430 ms	2.608 ms
Vehicle Speed	100 ms	1817	108.430 ms	5.631 ms

de tempo de transmissão de cada uma das mensagens puderam ser obtidas. A Tabela 4.1 apresenta essas informações.

Verifica-se através da Tabela 4.1 que os valores de média e desvio padrão de intervalos de tempo entre transmissões de cada uma das mensagens se manteve próximo do esperado.

A Tabela 4.2 apresenta os valores esperado e obtido para a vazão da rede embarcada. Nota-se que a rede real, sofreu uma perda de aproximadamente 7% da vazão.

É importante ressaltar que a performance da EMS foi penalizada devido ao foco da

Tabela 4.2: Vazão da rede embarcada do estudo de caso 3.

	Esperada	**Obtida**
frames por segundo	112	103.766

implementação estar em minimizar o espaço de memória utilizado, de forma a evitar um comportamento indefinido gerado por um possível estouro da pilha de execução.

Conclui-se que os resultados obtidos na etapa de simulação da rede automotiva em sistema embarcado foram satisfatórios. Foi possível observar o comportamento esperado para as variáveis do sistema em circunstâncias distintas e, apesar de não ter sido possível a validação da rede no ambiente de simulação do Busmaster o comportamento adequado da rede pôde ser verificado na plataforma embarcada.

Capítulo 5

Conclusão

O avanço da tecnologia automotiva fez com que o número de dispositivos eletrônicos encontrados em um automóvel aumentasse consideravelmente ao longo dos últimos anos.

Redes de comunicação automotivas surgiram com o objetivo de permitir a descentralização dos diferentes sistemas que compõem o veículo e simplificar o desenvolvimento de cada subsistema.

Apesar disso, o projeto de redes de comunicação que atenda aos requisitos impostos aos automóveis nos dias de hoje é uma tarefa extremamente desafiadora. Por esse motivo, o estudo de redes automotivas é primordial para a evolução dos automóveis.

Existem algumas ferramentas de simulação que auxiliam no projeto e análise de redes automotivas como o Busmaster. Contudo, o elevado custo das plataformas para teste em hardware embarcado faz com que o acesso a esse nível de simulação por parte de estudantes ou entusiastas da área seja dificultado.

Este trabalho apresentou uma solução que visa permitir a simulação de redes automotivas em sistemas embarcados de baixo custo por meio da tradução do código gerado através software Busmaster para a plataforma arduino utilizando o TrampolineRTOS.

Os resultados obtidos nos 3 estudos de caso comprovam o desempenho da solução proposta. Os estudos de caso serviram para verificar a implementação de cada uma das funcionalidades disponíveis para especificar o comportamento de uma ECU no software Busmaster. Foi possível observar em tempo real, através da interface serial do microcontrolador, a atualização das diferentes variáveis dos sistemas em cada um dos estudos de caso.

Apesar das restrições de recursos da plataforma embarcada escolhida possuírem um impacto considerável na performance da rede automotiva, o fácil acesso e o baixo custo justificam a escolha.

O desenvolvimento deste trabalho, possibilitou um maior entendimento da área de redes automotivas e da importância delas no projeto dos veículos atuais. A concepção

do tradutor permitiu um aprofundamento nas questões de implementação das redes em sistemas embarcados, na compreensão do funcionamento de softwares de projeto e simulação das mesmas e dos protocolo de comunicação CAN e da norma SAE J1939, padrões da indústria automotiva.

O software Code Generator produzido neste trabalho pode servir para uso em disciplinas acadêmicas de forma a possibilitar aos alunos uma experiência com simulação de redes automotivas em *hardware* embarcado acessível. Como visto nos estudos de caso, o contato com o hardware permite avaliar o projeto das redes considerando restrições reais de implementação, como a indisponibilidade de recursos físicos do sistema.

A aplicação desenvolvida, todavia, possui algumas limitações. Dentre elas, se destacam, as restrições de recursos disponíveis na plataforma alvo, como pouca memória para código e dados, o que traz inúmeras consequências, especificamente, preocupações com a quantidade de variáveis globais, com o tamanho da pilha de execução, com a quantidade de tarefas, entre outros. O hardware utilizado também possui baixa frequência do relógio interno e dos controladores das interfaces de comunicação, além da indisponibilidade de alocação dinâmica de memória.

Além disso, os programas desenvolvidos pelos usuários do Busmaster também podem estar sujeitos a condições de corrida quando convertidos para a plataforma embarcada, pelo fato da implementação fazer uso de múltiplas tarefas concorrentes. Condição de corrida é uma situação em que há acesso concorrente a um recurso compartilhado e a ordem dos acessos pode afetar o resultado das operações.

Ainda, algumas funções auxiliares providas pelo Busmaster não foram implementadas na plataforma embarcada, por motivos variados, como o alto custo de recursos e de tempo de execução gerados pela implementação, a não disposição das bibliotecas utilizadas pela implementação do busmaster, no microcontrolador, dentre outros.

Por fim, como trabalhos futuros, propõe-se a extensão do software para portar outras plataformas com mais recursos disponíveis como os microprocessadores da plataforma Raspberry Pi de forma a viabilizar o desenvolvimento de redes mais complexas e robustas e a adição de suporte a outros sistemas operacionais de tempo real com mais funcionalidades disponíveis, como o FreeRTOS da Amazon.

O código fonte do software Code Generator produzido neste trabalho se encontra em um repositório público no GitHub[1].

[1] `https://github.com/lucasavelino/code_generator`

Referências

[1] Navet, Nicolas, Yeqiong Song, Françoise Simonot-Lion e Cédric Wilwert: *Trends in automotive communication systems*. Proceedings of the IEEE, 93(6):1204–1223, 2005. 1, 5, 6

[2] Staron, Miroslaw: *Automotive Software Architectures - An Introduction*. Springer, 2017. 1, 5

[3] THE EDITORS OF PUBLICATIONS INTERNATIONAL, LTD: *Traction control explained*. `https://auto.howstuffworks.com/28000-traction-control-explained.htm`, acesso em 2018-11-17. 6

[4] Schäuffele, Jörg e Thomas Zurawka: *Automotive Software Engineering - Grundlagen, Prozesse, Methoden und Werkzeuge effizient einsetzen (3. Aufl.)*. Vieweg, 2006. 6, 7

[5] Paret, Dominique: *Multiplexed networks for embedded systems: CAN, LIN, Flexray, Safe-by-Wire...* John Wiley & Sons, 2007. 7, 9, 10, 13, 14

[6] Wissen, Aktie e Bekommen Mehr Wissen: *Can basics2*. `http://www.embeddedc.in/p/arbitration-in-cannetworks-thestandard.html`, acesso em 2018-09-17. 10, 11

[7] *A brief introduction to the sae j1939 protocol*. `https://copperhilltech.com/a-brief-introduction-to-the-sae-j1939-protocol`, acesso em 2018-09-18. 14

[8] *Introduction to sae j1939*. `https://vector.com/portal/medien/cmc/application_notes/AN-ION-1-3100_Introduction_to_J1939.pdf`, acesso em 2018-09-18. 15

[9] Control, Micro: *J1939 protocol stack*. `http://www.microcontrol.net/download/manual/hb_j1939_v3r00_en.pdf`, acesso em 2018-11-18. 16

[10] Vector: *Canoe*. `https://www.vector.com/int/en/products/products-a-z/software/canoe/`, acesso em 2018-11-09. 16

[11] *Canoe*. `https://www.mathworks.com/products/connections/product_detail/vector-canoe.html`, acesso em 2018-11-19. 17

[12] Navet, Nicolas e Francoise Simonot-Lion: *Automotive embedded systems handbook*. CRC press, 2008. 19

[13] Liu, Jane W.-S.: *Real-time systems*. Prentice Hall, 2000, ISBN 978-0-13-099651-0. 19

[14] Osek, OSEK: *Vdx operating system specification 2.2.3*. OSEK Group, 2005. 21, 22

[15] Group, OSEK *et al.*: *Osek/vdx system generation oil: Osek implementation language version 2.5*, 2004. 23

[16] *What is arduino*. `https://www.arduino.cc/en/Guide/Introduction`, acesso em 2018-09-21. 37

[17] Bechennec, Jean Luc, Mikaël Briday, Sébastien Faucou e Yvon Trinquet: *Trampoline an open source implementation of the osek/vdx rtos specification*. Em *Emerging Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on*, páginas 62–69. IEEE, 2006. 40

[18] *Trampoline rtos*. `https://github.com/TrampolineRTOS/trampoline`, acesso em 2018-09-24. 40

[19] *Framework qt*. `https://www.qt.io/`, acesso em 2018-09-24. 41

[20] *Boost spirit x3*. `https://www.boost.org/doc/libs/develop/libs/spirit/doc/x3/html/index.html`, acesso em 2018-09-24. 41