UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
CURSO DE ENGENHARIA MECATRÔNICA

THIAGO MARTINS

DESENVOLVIMENTO DE SNIFFER CAN APLICADO AO ÔNIBUS ELÉTRICO E EPOSMOTE III

Joinville
2017

THIAGO MARTINS

DESENVOLVIMENTO DE SNIFFER CAN APLICADO AO ÔNIBUS ELÉTRICO E EPOSMOTE III

Trabalho de Conclusão de Curso apresentado ao curso de graduação em Engenharia Mecatrônica, da Universidade Federal de Santa Catarina, Centro Tecnológico de Joinville, como requisito parcial para obtenção do título de Bacharel em Engenharia Mecatrônica.

Orientador: Prof. Dr. Anderson Wedderhoff Spengler

Joinville
2017

RESUMO

O trabalho teve como objetivo, desenvolver um *sniffer* de barramento CAN de um ônibus elétrico utilizando um EposMote III. O *sniffer* foi projetado, levando em consideração, a capacidade de leitura de todos os dados que trafegam pelo barramento, sendo ao todo, 1000 quadros por segundo. O objetivo futuro desta aplicação, é a sua utilização na identificação de dados voltados ao monitoramento do ônibus elétrico. Estes dados são referentes ao consumo de energia, ao estado das baterias, à velocidade de deslocamento do veículo, ao posicionamento (GPS), entre outros. O desenvolvimento do *sniffer*, passou primeiramente por testes em bancada, usando um barramento CAN emulado em dois microcontroladores. Este teste garantiu que o *sniffer* desenvolvido estava apto a interceptar dados em um barramento CAN. A etapa de testes no ônibus elétrico foi caracterizada pela coleta dos dados reais do barramento. Entre os principais resultados, obteve-se o correto funcionamento do gerenciamento de coleta de dados pelo EposMote. Em contrapartida, o resultado de desempenho do *sniffer*, não foi satisfatório, pois alguns requisitos de tempo não foram atendidos. O conflito, entre a obtenção dos dados do barramento pelo EposMote com o envio destes mesmos dados para o computador (utilizando o único processador do EposMote III), fizeram com que o *sniffer* deixasse de coletar alguns dados do barramento CAN do ônibus elétrico. Este problema foi diretamente relacionado ao consumo excessivo do tempo de CPU, pela comunicação serial UART usada para externalizar os dados do EposMoteIII para o computador. Ao final deste trabalho foram apresentados, além dos resultados obtidos, algumas sugestões para solução do consumo de tempo de CPU devido a externalização dos dados. São sugeridas também, melhorias no uso do controlador CAN, utilizado nesse trabalho.

Palavras-chave: *Sniffer*, *Controller Area Network*, Ônibus Elétrico.

ABSTRACT

The objective of the work was to develop a CAN bus sniffer of an electric bus using an EposMote III. The sniffer was designed, taking into consideration, the ability to read all data that travels through the bus. In total, about 1000 frames of messages per second travel. The future objective of this application is its use in the identification of certain data for the monitoring of the electric bus. These data refer to the energy consumption, the state of the batteries, the speed of movement of the vehicle, the positioning (GPS), among others. The development of the sniffer, first passed by laboratory tests, using a CAN bus emulated in two microcontrollers. This test ensured that the developed sniffer was able to intercept data on a CAN bus. The test stage in the electric bus was characterized by the collection of the actual data of the bus. From the main results obtained, the correct operation of the data collection management by EposMote was obtained. In contrast, the sniffer performance result was not satisfactory, as some time requirements were not met. The conflict between obtaining the EposMote CAN bus data and sending the same data to the computer (using the single EposMote III processor) caused the sniffer to stop collecting some data from the CAN bus. This problem was directly related to the excessive consumption of the CPU time, by the serial communication UART used to send the data of EposMoteIII to the computer. At the end of this work were presented, in addition to the results obtained, some suggestions for solving the CPU time consumption due to sending the data out of the device. Also suggested are improvements in the use of the CAN controller used in this work.

Keywords: Sniffer, Controller Area Network, Electric Bus.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

T_q Timing Quantum

ADESD Application Driven Embedded System Design

BRP Baud Rate Prescaler

CAN Controller Area Network

CCS Code Composer Studio

CI Circuito Integrado

EBus Electric Bus/Ônibus Elétrico

EMC Compatibilidade Eletromagnética

EPOS Embedded Parallel Operating System

ESD Descarga eletrostática

FOSC Frequency of Crystal Oscillator

IDE Ambiente de desenvolvimento Integrado

ISO Organização Internacional de Normalização

LIN Local Interconnect Network

LISHA Software/Hardware Integration Lab

LLC Logic Link Control

MAC Medium Acess Control

MOST Media Oriented System Transport

OSI Open System Interconnection

RT Real Time

SPI Serial Peripheral Interface

TTP/C Time-Triggered Protocol/Class C

SUMÁRIO

1 INTRODUÇÃO

A indústria automotiva evolui rapidamente sob a influência de fatores como a pressão por parte de legislações, dos clientes ou até pelo avanço tecnológico, e está atualmente adotando arquiteturas de carros flexíveis em sua estratégia para enfrentar uma concorrência acirrada, alcançando níveis mais altos de inovação em seus produtos (NAVET; SIMONOT-LION, 2008). Dessa forma, é visto a dependência do progresso tecnológico necessário para satisfazer estas necessidades e obrigações do mercado.

As tecnologias utilizadas no setor eletrônico passaram por grandes avanços o qual permitiu que qualquer função de um veículo fosse controlada por estes sistemas (NAVET; SIMONOT-LION, 2008). Exemplos de tais dispositivos incluem sistemas de gerenciamento do motor, suspensão ativa, ABS, controle de câmbio, controle de luzes, ar condicionando, airbag, central de trava elétrica, entre outros. Tudo isso significa mais segurança e conforto para o motorista, somado a redução no consumo de combustível, gases de emissão e custos para a montadora do automóvel.

Atualmente, cerca de 25 fabricantes de chips produzem dispositivos com interfaces CAN e grande parte dos carros de passageiros fabricados hoje em dia estão equipados com pelo menos uma rede CAN. Este protocolo também é usado em outros tipos de veículos, abrangendo desde trens a navios, estando presente até mesmo em sistemas industriais. Isto torna o CAN um dos protocolos de comunicação mais importantes e flexíveis já desenvolvidos (ESACADEMY, 2017).

Navet e Simonot-Lion (2008) apresentam cinco categorias principais utilizadas para classificar os sistemas embarcados em um carro, de acordo com as diferentes funcionalidades, restrições e modelos do sistema. Essas categorias são: powertrain, chassi, corpo, interface homem-máquina e telemetria. O domínio do powertrain está relacionado com os sistemas que participam na propulsão longitudinal do veículo, incluindo o motor, transmissão, e todos os componentes auxiliares. O chassi está relacionado as quatro rodas e sua posição relativa de movimento e, neste domínio, os sistemas são principalmente de direção e frenagem. O corpo inclui elementos que não pertencem à dinâmica do veículo, sendo, portanto, aqueles que apoiam o usuário como o airbag, limpador, iluminação, vidro elétrico, ar condicionado, equipamento de assento, etc. Já a interface homem-máquina inclui equipamentos que permitem a troca de informações entre os sistemas eletrônicos e o motorista através de telas, botões e

o volante. E por fim, o domínio da telemetria está relacionado com componentes que permitem a troca de informações entre o veículo e o monitoramento remoto.

A nova indústria de veículos elétricos trazem consigo uma série de novos desafios. Sendo uma tendência mundial, possui grande significado na segurança energética global, conservação de energia e redução de emissões, e promovendo o desenvolvimento na indústria automobilística. A função do veículo elétrico está aumentando e melhorando, e o sistema eletrônico é mais complexo devido o uso generalizado do sistema de controle elétrico, aumentando assim a dificuldade de manutenção e reduzindo a confiabilidade do veículo (WANG; YAO; SHI, 2011).

Muitas empresas e universidades vêm investindo neste setor e trazem consigo uma série de inovações. O Grupo de Pesquisa Estratégica em Energia Solar da Universidade Federal de Santa Catarina (FOTOVOLTAICA-UFSC, ou FV-UFSC) desenvolve estudos nas diversas áreas de aplicação da energia solar no Brasil e possui projetos também no uso desta energia em veículos elétrico (FV-UFSC, 2017). O ônibus elétrico alimentado por energia solar é um deles, e este possuí uma série de informações de controle de energia que trafegam por seu barramento CAN. Monitorar estes dados de forma remota, é uma necessidade atual do projeto. Para isto, é proposta a instalação de um *sniffer* no barramento CAN do ônibus elétrico. Os *sniffers* são ferramentas que interceptam e analisam os dados que trafegam em uma rede. E o grupo fotovoltaica deseja monitorar, de forma remota, os dados que trafegam no barramento CAN de seu ônibus elétrico. Dados que estão relacionados, por exemplo, ao consumo de energia, ao estado das baterias, dados de torque do motor, velocidade atual do ônibus, posição (GPS), entre outros.

A proposta deste trabalho, é realizar a leitura dos dados que trafegam neste barramento, utilizando um EposMote III. Com o auxílio de um controlador CAN autônomo com interface SPI (Serial Peripheral Interface) e um transceptor CAN de alta velocidade. O sistema operacional EPOS (Embedded Parallel Operating System) embarcado no dispositivo não possui suporte à comunicação CAN, por este motivo, os circuitos integrados que auxiliam no gerenciamento da comunicação são necessários.

Os testes iniciais são realizado com auxilio de dois microcontroladores TIVA C Séries TM4C123G da fabricante Texas Instruments, cada um conectado à um transceptor TCAN1042H, da mesma fabricante, que faz a conversão das saídas de leitura e escrita do microcontrolador em um verdadeiro barramento CAN. O teste final é realizado no barramento CAN do ônibus elétrico citado anteriormente para coletar todos os dados que ali transitam. A seguir são apresentados os objetivos geral e específicos deste trabalho.

1.1 Objetivos

1.1.1 Objetivo Geral

Desenvolver um sniffer de rede CAN utilizando um EposMote III.

1.1.2 Objetivos Específicos

Para atingir o objetivo principal, necessita-se concretizar os seguintes objetivos específicos:

- Gerenciar os CIs MCP2515 (controlador CAN) e MCP2562 (transceptor CAN) através das interfaces do microcontrolador EposMote III;
- Fazer um bancada de testes para experimentos pré campo;
- Instalar o sniffer desenvolvido no barramento CAN do ônibus elétrico e avaliar o funcionamento;

2 FUNDAMENTAÇÃO TEÓRICA

2.1 CAN - Controller Area Network

Controller Area Network (CAN) é o nome dado a uma técnica de comunicação serial, padronizada internacionalmente, desenvolvida para suportar as necessidades de comunicação encontradas em veículos automotores. Por ter nascido para a indústria automobilística, foi projetada para ter imunidade a interferência eletromagnética, grande capacidade de auto-diagnóstico e recuperação de erros (BOSCH et al., 1991). Essas atribuições garantiram a aplicação do CAN também na indústria de automação, médica, manufatura, entre outras.

O meio de transmissão de informações não é explicitamente definido, já que a comunicação pode ocorrer via cabos, rádio-frequência, infra-vermelho, etc. A única restrição para transmissão é que o meio deve conseguir representar bits com dois estados distintos. Na maioria dos sistemas, é necessário conhecer algumas atribuições e o funcionamento dos mecanismos implementados (PARET, 2007). Abaixo são listados as principais elementos que compõem o sistema CAN:

- Nó transmissor: um componente conectado à rede de comunicação é denominado transmissor. O transmissor envia informações ao meio e até que este barramento entre num estado ocioso ou seja ocupado por outro nó transmissor;
- Nó receptor: um componente é chamado de receptor quando não está transmitindo uma mensagem;
- Valores do barramento: em CAN, os estados do barramento são chamados de dominantes (nível lógico baixo) ou recessivos (nível lógico alto);
- Taxa de bits do barramento: é o número de bits transmitidos por unidade de tempo. Ela pode ser diferente de um sistema para outro, mas deve ser uniforme e constante para qualquer nó da rede;
- Mensagens/*Frames*: as mensagens no barramento são enviadas em um formato específico com comprimento e tempo limitados;

- Direcionamento de informações: o conteúdo de mensagens é marcado por um identificador, o qual não informa o destino, mas sim o significado dos dados. Isso significa que cada nó pode ser programado para decidir se a mensagem no barramento é relevante ou não;
- Prioridades: o transmissor define uma prioridade através do identificador da mensagem antes de acessar o barramento;
- *Frame* de dados: é o *frame* que carrega informação;
- *Frame* remoto: sinaliza a necessidade de receber informação na forma de um *frame* de dados correspondente;
- Arbitração: se dois ou mais nós tentarem transmitir ao mesmo tempo, o conflito das mensagens é resolvido pela técnica não destrutiva "*bitwise*", na qual o identificador é analisado e a mensagem de maior prioridade é mantida. Essa técnica garante que não há perda de informações por conflito de transmissão.

Nas seções a seguir, são descritas as camadas de comunicação onde o protocolo CAN opera, em seguida, são apresentados os diferentes tipos de *frames* que fazem parte do protocolo e por fim suas características de detecção e resolução de erros.

2.1.1 Camadas de Comunicação

O protocolo CAN opera em conformidade com o modelo OSI (Open System Interconnection) que é uma referência da ISO para classificação protocolos de comunicação entre os mais diversos sistemas e assim garantir um processo comunicação entre dois terminais. Esta classificação divide o sistema de comunicação em sete camadas de abstração. Cada nível implementa diferentes funcionalidades que compõem um dado sistema de comunicação.

Tabela 1 – Camadas CAN e o modelo OSI.

Num. de Camadas	Modelo ISO/OSI	Protocolo CAN
7	Aplicação	Especificado pelo Usuário
6	Apresentação	em branco
5	Seção	em branco
4	Transporte	em branco
3	Rede	em branco
2	Enlace	Protocolo CAN (com livre escolha de meio)
1	Física	

Fonte: PARET(2007).

Na Tabela 1 podem ser visualizadas as 7 camadas que compõem o modelo OSI. O protocolo CAN corresponde às duas primeiras camadas, ou seja, ele trata apenas das camadas física e de enlace.

2.1.1.1 Camada Física

Geralmente, quando construídas essas subcamadas, elas não necessitam mais serem modificadas e, teoricamente, devem funcionar com qualquer protocolo. Basicamente, a camada física especifica como o sinal será transmitido e deve garantir a transferência entre nós independente do meio. Para operar de forma adequada, todos os nós da rede devem estar se comunicando através do mesmo meio. Sendo assim, a camada física é responsável por:

- Representar os bits de um sinal;
- Sincronizar o processo de transmissão;
- Definir os níveis, elétricos ou ópticos, do sinal;
- Definir o meio de transmissão.

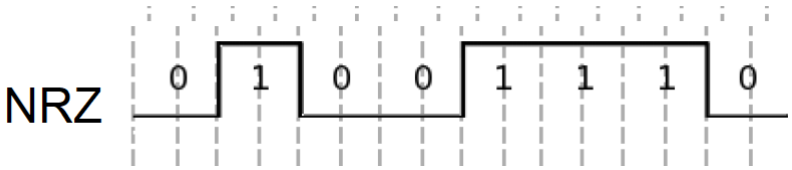
Conforme já informado, nos documentos de padronização, o meio pelo qual os frames de mensagens são transmitidos não são explicitamente definidos. Porém, o meio definido para transmissão de informações pelo protocolo CAN deve ser capaz de:

- Representar a informação em bits com estados dominantes e recessivos;
- Permanecer em um estado recessivo enquanto um nó envia um bit recessivo ou quando nenhum nó está transmitindo;

- Permanecer em um estado dominante enquanto um nó envia um bit dominante, suprimindo quaisquer bits recessivos.

O bit transmitido pela comunicação CAN é codificado de acordo com o princípio NRZ (Non Return Zero). A codificação NRZ é uma forma de garantir que durante a geração do bit seu nível lógico permanece constante independente de ser dominante ou recessivo. A Figura 1 ilustra esse conceito.

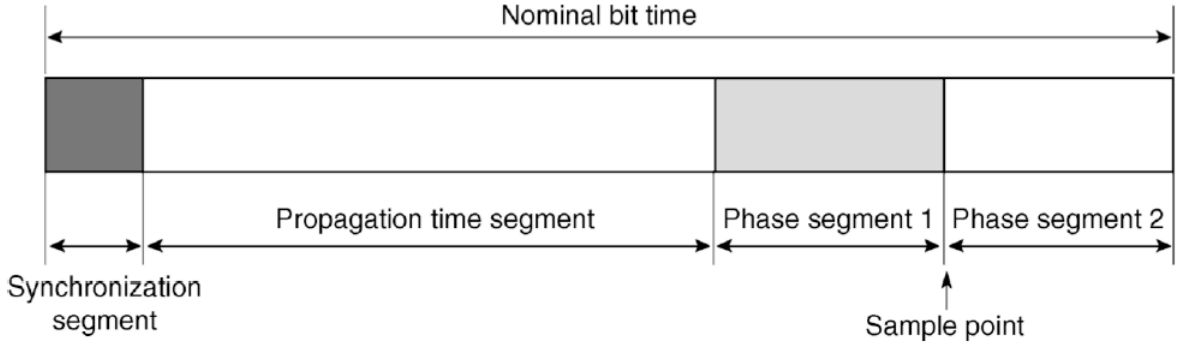
Figura 1 – Codificação NRZ



Adaptado de: PARET(2007).

O tempo nominal de bit é dividido em um número de segmentos de tempo que não podem ser sobrepostos. Eles estão representados na Figura 2. Por definição, a taxa nominal de bit é inversamente proporcional ao tempo nominal de bit.

Figura 2 – Estrutura do tempo nominal do bit em CAN



Adaptado de: PARET(2007).

Os segmentos que compõem o tempo nominal do bit sob o protocolo CAN são:

- Segmento de sincronização: utilizado para sincronizar os vários nós presentes no barramento. Por definição, a borda de entrada do bit deve aparecer nesse segmento;
- Segmento de propagação: é o tempo necessário para compensar atrasos devido à fenômenos físicos de propagação do sinal;

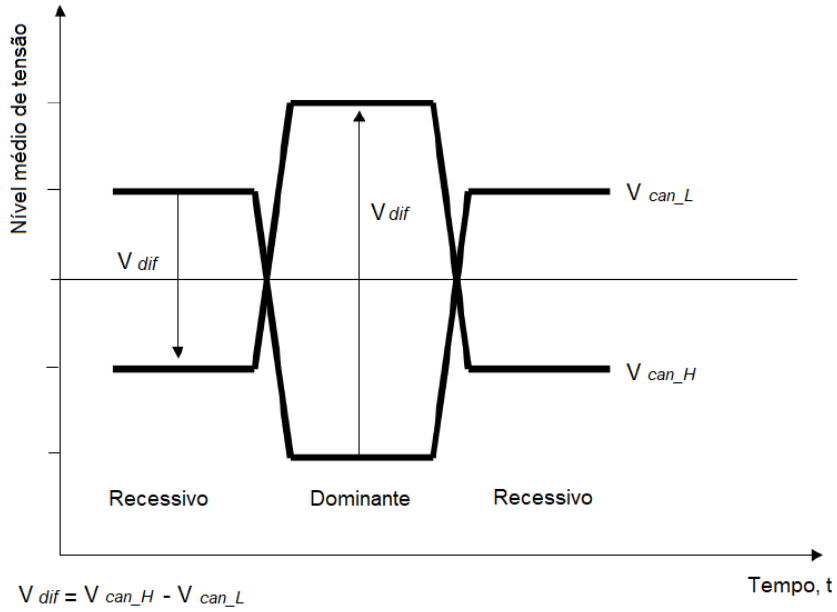
- Segmentos de fase 1 e 2: são utilizados para compensar erros, variações ou mudanças na fase ou posição das bordas dos sinais. Além disso, são os instantes ideais para a amostragem do bit.

Em uma rede CAN o tempo nominal de bit é fixo, específico de seu projeto e cada nó deve ser capaz de operar de acordo com sua definição. Para evitar problemas de construção, o protocolo especifica a duração de cada fase do tempo de bit. A menor unidade que deve ser considerada é o chamado quantum (tq). Ele é definido como a unidade básica do tempo de bit e é obtida através do clock do nó CAN ($fsys$) e de um Baud Rate (BRP) da seguinte forma: $tq = BRP/fsys$ (MICROCHIP TECHNOLOGY, 2005a).

As taxas de transmissão do protocolo CAN tendem a apresentar valores padrão, como: 33.333 bps, 83.333 bps, 50 Kbps, 100 Kbps, 125 Kbps, 250 Kbps, 500 Kbps (mais comum), 800 Kbps e 1.000 Mbps. Taxas de transmissão superiores a 1.000 Mbps não são encontradas, pois isso viola as especificações do protocolo. Tais convenções tomam por base fios condutores como o meio físico de transferência e, como todo processo de comunicação, sofre atenuação de seu sinal ao longo de sua transmissão. Por este motivo, o máximo comprimento de uma rede CAN depende de sua velocidade operação (GUIMARÃES, 200-).

O segmento de sincronização possui tamanho fixo de 1 quantum. Já o segmento de propagação utiliza dados de configuração da própria rede em seu cálculo. Os segmentos de fase 1 e 2 são definidos de acordo com os valores nominais de clock, suas tolerâncias, flutuações de tempo, temperatura de operação e considerações de pior caso. Mesmo com a flexibilidade oferecida pela configuração de sua camada física, a transmissão de informações através de cabos elétricos ainda é predominante. Através deste meio, utiliza-se um mecanismo baseado na diferença de tensão entre o par trançado CAN_L e CAN_H. A diferença de potencial entre os cabos é analisada para interpretação dos dados na rede (GUIMARÃES, 2007). A Figura 4 exemplifica este comportamento.

Figura 3 – Barramento CAN diferencial



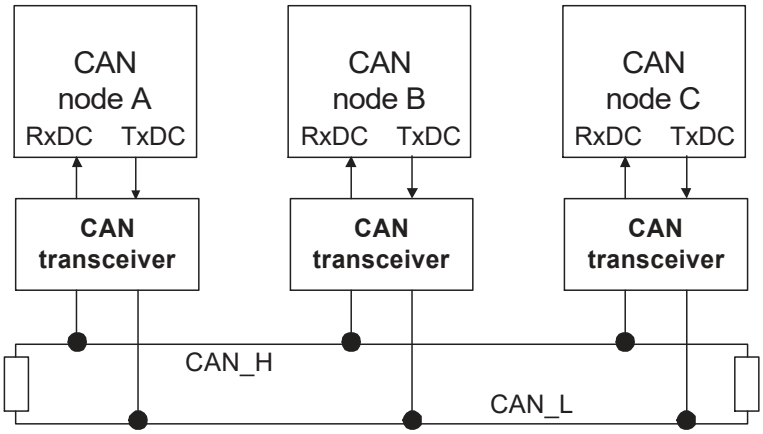
Fonte: VIEIRA et al.(2007).

O valor de tensão no barramento é relativo à tensão de cada cabo CAN_L (CAN-LOW) e CAN_H (CAN-HIGH) em relação ao cabo terra, e variam de acordo com sua implantação. O CAN_L durante o estado dominante possui baixo nível de tensão, e alto no recessivo. O CAN_H tem alto nível de tensão no estado dominante e baixo no recessivo (INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO), 2003).

O fato de este protocolo fazer sua leitura de modo diferencial e o meio de propagação do sinal ser um par trançado, lhe garante bom desempenho na imunidade a ruídos e interferências eletromagnéticas. O par trançado faz com que o ruído se propague igualmente através dos dois condutores e a leitura diferencial é capaz de anular sua influência.

Na prática, quando não há transmissão de dados no barramento, este permanece em estado recessivo devido à presença de resistores “pull-up” que induzem esse estado. Quando um nó deseja transmitir um dado ele emite um bit dominante que sobrescreve o bit recessivo, lhe dando prioridade no envio. Sua configuração permite que, durante qualquer processo de comunicação, todos os seus elementos observem as informações do barramento, conforme indicado pela Figura 4, onde a forma de conexão de elementos ao barramento é demonstrada.

Figura 4 – Conexão de dispositivos ao barramento CAN



Fonte: SIEMENS(1996).

2.1.1.2 Camada de Enlace

A segunda camada do modelo OSI é composta por duas subcamadas: MAC (Medium Access Control) e LLC (Logic Link Control). A primeira é basicamente o núcleo do protocolo. Sua função é apresentar as mensagens recebidas da subcamada LLC e aceitar mensagens para sua transmissão à LLC. De modo resumido, MAC é responsável por:

- Enquadramento de mensagens;
- Arbitração ou decisão sobre as mensagens;
- Reconhecimento;
- Detecção de erros;
- Sinalização de erros.

Já LLC possui a função de filtrar e processar as mensagens e seus estados. Com o objetivo de identificar quais mensagens devem ser transmitidas, a subcamada LLC decide quais mensagens serão utilizadas pelo nó de comunicação e de produzir uma interface entre a camada de aplicação e o hardware do sistema. De um modo geral, suas responsabilidades são:

- Filtrar mensagens;
- Notificar overloads;

- Tratar da recuperação de erros.

Dito isto, as seções a seguir detalharão como essas responsabilidades são desenvolvidas.

2.1.2 Transferência de Mensagens

A transferência de mensagens é composta por tipos de frames, intervalos e estados bem definidos:

- Frame de dados: transportam os dados ou mensagens entre os nós do barramento CAN;
- Frame remoto: enviado por uma unidade ativa no barramento para requisitar a transmissão de um frame de dados cujo identificador tem o mesmo valor que o frame remoto;
- Frames de overload: este frame indica que um nó teve sobrecarga e é utilizado para requisitar mais tempo entre o precedente e o próximo frame de dados ou frame remoto;
- Frame de erro: transmitidos por qualquer nó no barramento sempre que uma malha na comunicação é detectada;
- Interframes: espaço de tempo entre a transmissão dois frames;
- Modo Sleep (Descanso): certos dispositivos CAN possuem a capacidade de cessar suas atividades internas e possibilitar uma economia de energia, entrando no chamado modo sleep.

2.1.3 Características, Detecção e Processamento de Erros

A detecção a o processamento de erros é uma das capacidades mais importantes do protocolo CAN e é essencial compreender o seu mecanismo a fim de beneficiar todas as suas vantagens. Este processo é muitas vezes abstraído por elementos de hardware desenvolvidos em conformidade com este protocolo, de forma com que a interação do usuário não se faz necessária. No entanto, eventualmente pode-se encontrar problemas que necessitam de um melhor entendimento deste processo para solucioná-los. Pode-se, por exemplo, evitar o desperdício de muitas horas de desenvolvimento de software ao reinventar métodos que já foram muito bem implementado no hardware.

Desta forma, esta seção será dividida em duas partes: a primeira tratará dos tipos de erros que podem ocorrer e sua estratégia de processamento e recuperação de erros. A segunda será apresentará mais detalhes sobre os métodos de detecção, sinalização e correção de erros. É importante ressaltar que a ocorrência de qualquer erro ocasiona no envio de um frame de erro, o qual será apresentado no decorrer desta seção. Os erros que podem ocorrer num processo de comunicação são:

Na camada física:

- Efeitos parasitas;
- Erros de preenchimento de bit;
- Falha na detecção do bit ACK, mesmo após uma transmissão de sucesso.

Na violação do formato do frame:

- Erro no delimitador CRC;
- Erro no reconhecimento ACK;
- Erro de fim de frame;
- Erro de delimitador;
- Erro de overload.

Juntamente com os erros, o controlador do sistema CAN deve ter a capacidade de observar se o erro é frequente e afeta de forma significativa o funcionamento da rede ou acontece esporadicamente. Sobre distúrbios frequentes, o controlador pode decidir mudar para um estado denominado ’bus off’ e recarregar suas configurações padrão.

2.2 EPOS - Embedded Parallel Operating System

O protocolo CAN anteriormente apresentado será implementado no EPOS (Embedded Parallel Operating System). Um sistema operacional desenvolvido pelo LISHA (Software/Hardware Integration Lab). Este sistema faz parte de um projeto que visa automatizar o desenvolvimento de sistemas integrados para que os desenvolvedores possam se concentrar no que realmente importa: suas aplicações (LISHA - SOFTWARE/HARDWARE INTEGRATION LAB, 2017c).

O EPOS conta com o método ADESD (Application Driven Embedded System Design) para orientar o desenvolvimento de componentes de software e hardware que podem ser adaptados automaticamente para atender aos requisitos de aplicativos específicos. O EPOS possui um conjunto de ferramentas para auxiliar os desenvolvedores na seleção, configuração e conexão de componentes em sua estrutura

específica da aplicação. A combinação de metodologia, componentes, frameworks e ferramentas possibilita a geração automática de instâncias de um sistema embarcado específico do aplicativo (LISHA - SOFTWARE/HARDWARE INTEGRATION LAB, 2017a).

2.3 Fotovoltaica UFSC

O Grupo de Pesquisa Estratégica em Energia Solar da Universidade Federal de Santa Catarina, FOTOVOLTAICA-UFSC, ou FV-UFSC, fica localizado no Sapiens Parque, Florianópolis-SC, e desenvolve estudos nas diversas áreas de aplicação da energia solar no Brasil, com foco principal em sistemas fotovoltaicos integrados ao entorno construído e interligados à rede elétrica pública, os chamados Edifícios Solares Fotovoltaicos (FV-UFSC, 2017).

Os Edifícios Solares Fotovoltaicos integram à sua fachada e/ou cobertura, módulos solares que geram, de forma descentralizada e junto ao ponto de consumo, energia elétrica pela conversão direta da luz do sol e servem ao mesmo tempo como material de revestimento destas fachadas e coberturas (FV-UFSC, 2017). Diversos projetos são realizados por este grupo de pesquisa, como o carro elétrico, análise economica de painéis solares em edifícios (como por exemplo do Estádio do Maracanã), o laboratório móvel ônibus elétrico e a própria estrutura do laboratório que é toda alimentada com energia solar.

No presente trabalho, o sniffer de rede CAN desenvolvido será diretamente conectado ao barramento CAN do ônibus elétrico para coleta dos dados que nele trafegam.

2.3.1 Projeto Ônibus Elétrico

O ônibus elétrico, alimentado por energia solar, foi introduzido com o intuito de adicionar um transporte com deslocamento produtivo. Foi inaugurado em dezembro de 2016, e iniciou o serviço regular de transporte entre o Campus Trindade e o Sapiens Parque em março de 2017. O ônibus – que é parte de um projeto de deslocamento produtivo com veículos elétricos alimentados por energia solar fotovoltaica – é um ambiente de trabalho, com poltronas confortáveis (somente transporta passageiros sentados), duas mesas de reunião, tomadas 220V e USB, ar-condicionado e wi-fi UFSC (FV-UFSC, 2017). Na Figura 5, a foto do ônibus elétrico retirada Sapiens Parque em seu posto de reabastecimento.

Figura 5 – Ônibus Elétrico estacionado no Sapiens Parque, Florianópolis-SC.

Fonte: Autor (2017).

O veículo elétrico realiza cinco viagens por dia (52 km por viagem do Sapiens Parque à UFSC e retorno, cerca de 5.000 km/mês), prestando serviços regulares e gratuitos para a comunidade UFSC. O projeto contou com financiamento de um milhão de reais pelo Ministério da Ciência, Tecnologia e Inovação (MCTI, atual MCTIC) e conta com a parceria das empresas WEG, Marcopolo, Mercedes e Eletra.

O ônibus de possui carroceria Marcopolo-Torino, um chassi Mercedes - O-500U e tem capacidade para 38 passageiros sentados. Atinge velocidade máxima de 80 km/h. Necessita de um tempo de recarga de 2,5 horas, a Voltagem do sistema de tração é de 600 Vdc com um sistema auxiliar de 12/24 Vdc - 220 Vac. Seu motor de tração é um modelo Trifásico W22 Plus 250L com potência nominal de 200 kW, tensão nominal de 440Vac, 6 polos, 1180 rpm e sua refrigeração é por água (ELETRA, 2016).

3 MATERIAIS E MÉTODOS

Este trabalho teve como objetivo desenvolver um *sniffer* de rede CAN e aplicar o mesmo ao barramento presento no ônibus elétrico do grupo fotovoltaica UFSC, conforme apresentado nos capítulos anteriores. Para realizar esta tarefa, foi escolhida a plataforma de hardware EposMote III, que utiliza o sistema operacional EPOS. O circuito projetado para ser acoplado a esta plataforma contem CIs exclusivos para 4 protocolos de comunicação serial bem utilizados, sendo eles: RS232, RS485, CAN e LIN. A função do Epos é gerenciar os CIs destinados à comunicação CAN.

Para realizar o controle destes circuitos integrados, foi utilizada comunicação serial SPI suportada pelo microcontrolador e implementada no sistema operacional. Foi utilizado um controlador CAN MCP2515 para gerenciar a comunicação através das configurações recebidas via SPI. Também é utilizado um transceptor MCP2562 que transforma a comunicação CAN baseada em TX/RX (proveniente do dispositivo MCP2515) em um verdadeiro nó de barramento CAN com CAN_L e CAN_H.

A implementação é simplificada de forma a utilizar apenas os modos de comunicação de transmissão e recepção. A proposta é implementar apenas um *sniffer*, ou seja, agir de forma passiva no barramento CAN. Sendo assim, algumas características do CAN não foram aplicadas no código gerado no sistema operacional Epos.

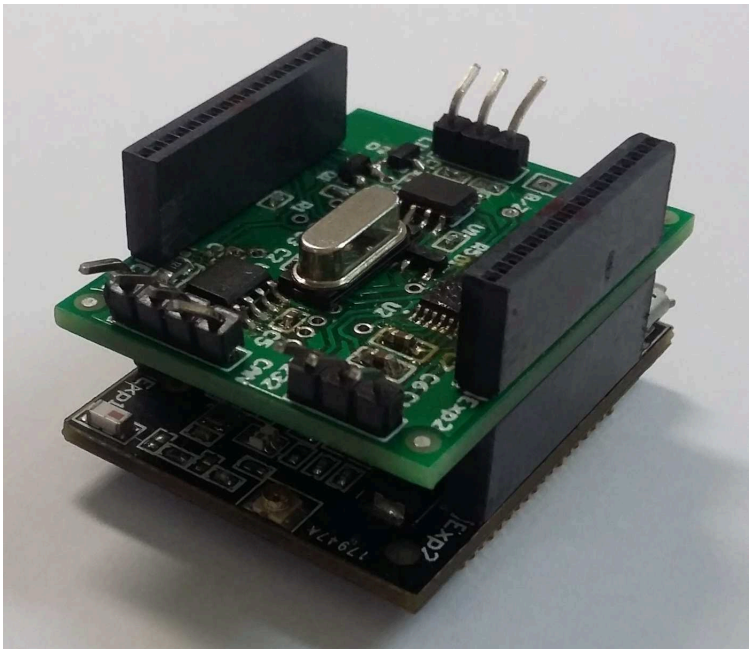
Antes de colocar o *sniffer* na rede CAN do ônibus elétrico, foram realizados testes em bancada para validar o funcionamento do mesmo. Para isto, foram utilizados dois microcontroladores, cada um com um transceptor que, da mesma forma que o CI MCP2562, transforma a comunicação RX/TX em um barramento CAN_L e CAN_H.

3.1 Dispositivos do *Sniffer* CAN

O conjunto de hardware apresentado nesta seção foi utilizado para realizar o recebimento dos dados do barramento CAN. Como já citado anteriormente, o *sniffer* é desenvolvido utilizando um plataforma de hardware conhecida como EposMote III, o qual necessita de circuitos integrados específicos para realizar algumas comunicações seriais, como a comunicação CAN, por exemplo. A imagem apresentada na Figura 6

mostra os circuito de comunicação serial (placa superior verde) acoplada ao EposMote III (placa inferior preta).

Figura 6 – Conjunto EposMote III e Serial Board - *Sniffer*



Fonte: Autor (2017)

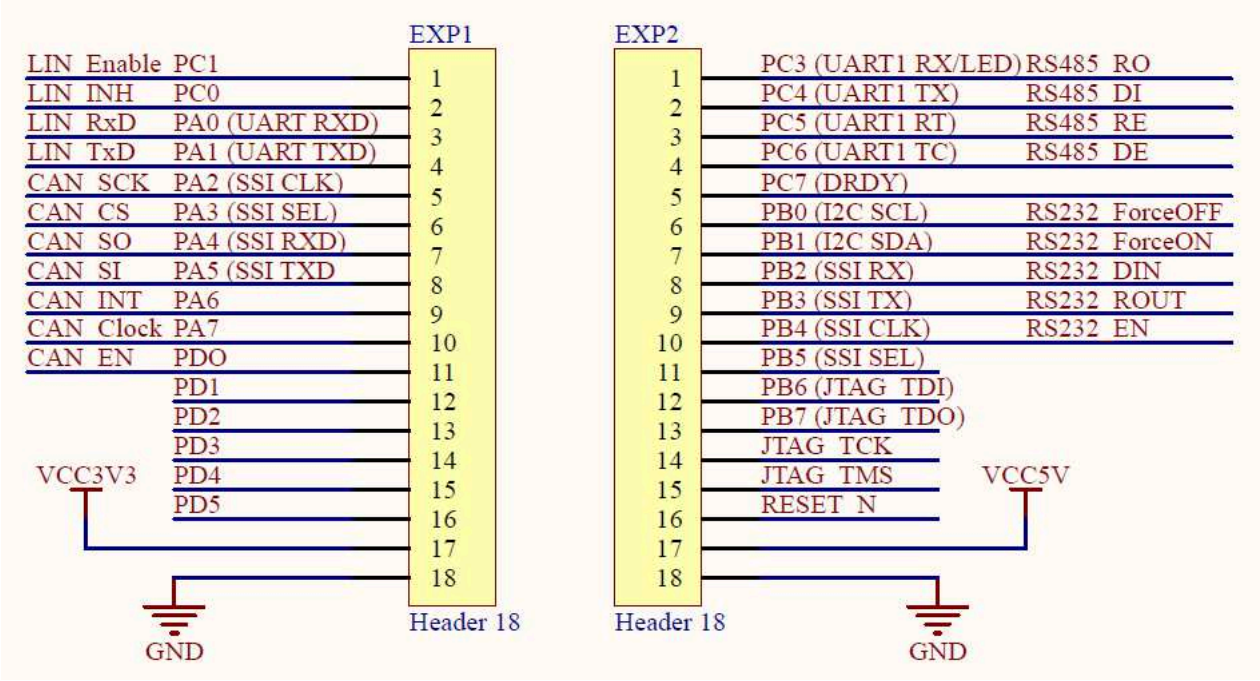
O EposMote III é apresentado no tópico a seguir, e na sequência, serão abordados o circuito da placa de comunicação serial adaptada para utilização junto ao EposMote III, e por fim, são descritos os circuitos integrados para comunicação CAN e suas funcionalidades.

3.1.1 EposMote III

O SoC (System-on-a-chip) escolhido para esta aplicação foi o Texas Instruments CC2538, que combina uma MCU ARM Cortex-M3 com memória RAM de até 32KB e até 512KB de memória *Flash*, e opera à uma frequência de 32 MHz (LISHA - SOFTWARE/HARDWARE INTEGRATION LAB, 2017a).

A placa de desenvolvimento EposMote possui 36 pinos de saída o quais são mostrados na Figura 7. Nesta figura, ainda é possível observar, que as saídas do EposMote estão conectadas com os protocolos de comunicação serial da placa de expansão.

Figura 7 – Conectores externos EposMote III relacionados a Placa de comunicação Serial



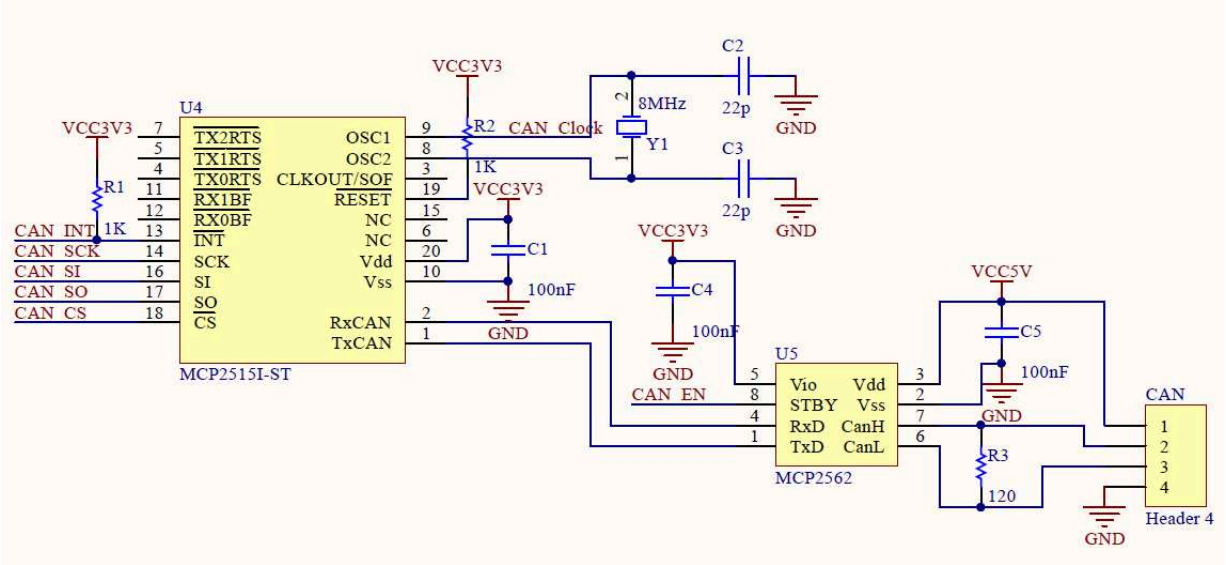
Fonte: LISHA - SOFTWARE/HARDWARE INTEGRATION LAB(2017b)

As GPIOs utilizadas neste trabalho são voltadas a comunicação CAN. Logo os pinos utilizados são: PA2, PA3, PA4, PA5, PA6, PA7 e PD0. A funcionalidade de cada um é também relacionada na Figura 7. Os pinos de PA2 à PA5 são utilizados para comunicação SPI através do módulo SSI do microcontrolador e dois pinos de entrada de interrupção e *clock* PA6 e PA7 respectivamente. O pino PD0 é o EN (CI *enable*). Ao decorrer deste capítulo, principalmente nos tópicos destinados ao CIs, a funcionalidade de cada pino fica mais evidente.

3.1.2 Placa de Comunicação Serial

O circuito de comunicação serial é utilizado como um extensor do EposMote III que utiliza de seus periféricos como SPI, UART e GPIO para gerenciamento e aplicação de comunicações RS232, RS485, LIN e CAN. O esquemático encontra-se em anexo no fim deste trabalho e mostra o circuito completo da placa. Para os fins deste trabalho, apenas o circuito utilizado na comunicação CAN é apresentado na Figura 8

Figura 8 – Circuito CAN da placa de comunicação Serial



Fonte: LISHA - SOFTWARE/HARDWARE INTEGRATION LAB(2017b)

Conforme apresentado na Figura 8, dois CIs são utilizados para a comunicação CAN. Um deles, é o controlador MCP2515, que faz o papel de gerenciador CAN, e possui um interface SPI diretamente conectada aos periféricos do EposMotes III. O outro, trata-se de um transceptor MCP2562, que transforma o sinal Rx/Tx, obtido na conexão com o MCP2515, em sinais CAN_L/CAN_H para o barramento CAN.

Nota-se que os pinos de interface serial SPI (PA2 à PA5) são conectados ao controlador MCP2515. Assim como os pinos de interrupção (PA6) e *clock* (PA7). O pino PA6 permite que o microcontrolador receba os sinais de interrupção gerados pelo controlador. E o pino PA7 permite que o microcontrolador forneça um sinal de *clock* para o controlador CAN, caso necessário. Apenas o pino EN (PD0) é conectado no pino STBY do MCP2562.

Durante o teste inicial, um erro de hardware da placa de comunicação serial foi identificado. O circuito projetado não permitia o funcionamento do cristal oscilador, e sendo assim, o MCP2515 não estava operante. Observou-se uma ligação inadequada do cristal com o pino PA7 do EposMote. Esta ligação, à principio, serviria para redundância de *clock*, porém, esta impedia o seu funcionamento. A solução encontrada para o problema, foi o rompimento dessa trilha.

3.1.2.1 Controlador MCP2515

O MCP2515 é um controlador CAN que suporta a versão CAN2.0B, ou seja, é um nó CAN completo que tem duas máscaras e seis filtros de aceitação os quais são usados para filtrar mensagens indesejadas, reduzindo assim a sobrecarga dos

microcontroladores. O MCP2515 interage com microcontroladores através de uma interface periférica serial (SPI) padrão (MICROCHIP TECHNOLOGY, 2005a). Possui ao todo 5 *buffers*, sendo 3 de escrita (TX *buffer*) e 2 de leitura (RX *buffer*). Ao todo, são 18 pinos de gerenciamento do CI, mas conforme apresentado na Figura 8, nem todos são utilizados no circuito de comunicações seriais. O MCP2515 suporta 5 modos de operação sendo eles:

1. Configuration Mode;
2. Normal Mode;
3. Sleep Mode;
4. Listen-Only Mode;
5. Loopback Mode.

Dos modos de operação acima listados, apenas o modo de Configuração e Listen-Only foram utilizados. O CI ainda permite realizar o RESET do dispositivo através do pino (que esta diretamente ligado na alimentação do EposMote) ou via interface SPI. E logo após este reinicio ele é prontamente colocado em modo de Configuração.

Alguns registradores do MCP2515 só podem ser modificados durante o modo de configuração, conforme especifica o datasheet. Estes registradores são relacionados as mascaras e filtros de mensagens, as interrupções de escrita gerada pelos *buffer*s TX quando estes enviam a mensagem, e as configurações de bit timing (CNF). O dispositivo ainda exige um *clock* externo necessário para sincronismo e calculo do Timing Quantum (Tq), setado nos registradores CNF. Este calculo foi brevemente abordado na Seção 2.1.1.1 e necessita de alguns dados para obtenção do Tq desejado. Por exemplo, supondo que seja desejado um baude rate de $125kHz$ com $FOSC = 20MHz$ e escolhendo um BRP=4 onde $T_q = 2.(BRP + 1)/FOSC$:

$$T_q = \frac{2(4+1)}{20000000} T_q = 500ns \tag{1}$$

E para determinar o tempo de bit necessário para configurações dos registradores CNF, deve realizar a seguinte operação:

$$T_{qs} = \frac{\frac{1}{T_q}}{BaudeRate} T_{qs} = \frac{\frac{1}{500ns}}{125KHz} T_{qs} = 16TQ \tag{2}$$

Este valor obtido de é o numero de time quantum usado para configurar os registradores CNF1, CNF2 e CNF3 e devem ser distribuídos entre os mesmos seguindo os critérios de TQ dos registradores (MICROCHIP TECHNOLOGY, 2005a). O valor de Time Quantum de 16TQ foi mantido em todo o projeto, apenas adequando a frequência do oscilador (por necessidade em atingir maiores taxas de transmissão) e/ou o valor

de BRP. Estas escolhas serão novamente abordadas ainda nesta seção. Para mais informações da distribuição deste Time Quantum ver datasheet.

A configuração necessária é realizada pela MCU através da interface SPI. Esta interface possuí alguns instruções pré definidas para o MCP2515. Estas instruções são apresentadas na Tabela 2 à seguir:

Tabela 2 – Instruções SPI

Nome da Instrução	Formato da Instrução	Descrição
RESET	1100 0000	Faz o Reset dos Registradores para o estado Default, set o modo de Configuração
READ	0000 0011	Lê os valores do registrador
Read RX Buffer	1001 0nm0	Comando de leitura dos buffers RX0 e RX1. O conjunto n e m determinam de qual buffero dado ou o identificador será lido. Mantendo os sinal CS baixo, esta função irá realizar a leitura de todos os 8 bytes do buffer.
WRITE	0000 0010	Comando de escrita nos registradores
Load TX Buffer	0100 0abc	Escrita nos buffers TX0, TX1 e TX2. O conjunto a, b e c determinal em qual buffer o dado ou o identificador será escrito.
RTS (Message Request-To-Send)	1000 0nnn	Requisita o envio dos dados contidos em um ou mais buffers TX. Os bis 'n' correspondem aos buffer em ordem decrescente: TX2 - TX1 - TX0.
Read Status	1010 0000	Lê o registrador de estados de escrita e leitura dos buffers
RX Status	1011 0000	Comando de pesquisa rápida que indica a correspondência do filtro e o tipo de mensagem (padrão, estendido e/ou remoto) da mensagem recebida
Bit Modify	0000 0101	Realiza a modificação bit-a-bit dos registradores (nem todos os registradores permitem esta ação. Para saber quais registradores suportam esta ação, ver a Seção 11 do datasheet)

Adaptado de: MICROCHIP TECHNOLOGY (2005a)

As instruções acima listadas foram utilizadas como base de concepção do *sniffer* CAN deste trabalho. Vale ressaltar que o objetivo é apenas realizar a coleta dos dados de forma passiva, ou seja, as configurações, instruções e modos de operação que não são necessários para esta proposta não forma implementadas.

3.1.2.2 Transceptor MCP2562

O MCP2562 é um transceptor CAN de alta velocidade da Microchip Technology. Ele serve como uma interface entre um controlador de protocolo CAN (Rx/Tx) e o barramento físico de dois fios CAN (CAN_L e CAN_H). O dispositivo é compatível com a norma ISO-11898-5 e atende aos requisitos automotivos para alta velocidade (até 1 Mb/s), baixa corrente quiescente, compatibilidade eletromagnética (EMC) e descarga eletrostática (ESD) (MICROCHIP TECHNOLOGY, 2005b).

Normalmente, cada nó em um sistema CAN deve ter um dispositivo para converter os sinais digitais gerados por um controlador em sinais adequados para transmissão através do barramento (saída diferencial). Ele também fornece um *buffer* entre o controlador CAN e os picos de alta voltagem que podem ser gerados no barramento CAN por fontes externas.

Conforme especificado pela MICROCHIP TECHNOLOGY 2005b, o transceptor possui dois modos de operação, sendo determinados como NORMAL e STANDBY. O modo normal é selecionado aplicando um nível de tensão baixo ao pino STBY. Neste estado o driver está operacional e pode gerenciar os pinos do barramento. As encostas dos sinais de saída em CANH e CANL são otimizadas para produzir emissões eletromagnéticas mínimas. O receptor diferencial de alta velocidade é ativo neste estado. Para selecionar o modo standby o pino STBY é colocado em um nível alto. Neste modo, o transmissor e a parte de alta velocidade do receptor são desligados para minimizar o consumo de energia. O receptor de baixa potência e o bloco de filtro de despertar (Wake-up) são habilitados para monitorar as atividades do barramento. O pino de recebimento (RXD) mostrará uma representação atrasada do barramento CAN, devido o filtro wake-up.

Conforme mostrado no circuito da placa de comunicação serial, o pino STBY esta diretamento conectado a saída GPIO D0 (representada por CAN EN) do EposMoteIII. E para o trabalho em questão, este pino é mantido em nível lógico baixo e dispensa o filtro wake-up.

3.2 Epos - Implementação

Com base nas informações levantadas nas seções anteriores foi possível estabelecer um plano de trabalho para implementação do código no sistema operacional

EPOS. Desta forma, foram estabelecidas as funcionalidades da aplicação e os métodos a ela atribuídos.

O uso da comunicação SPI disponível no EPOS é fundamental para o envio das instruções. A interface serial SPI, possui algumas funções as quais foram analisadas e selecionadas para a aplicação CAN. As funções de escrita bloqueante e leitura não bloqueante foram utilizadas e o construtor de classe SPI recebeu diretamente da aplicação CAN os dados de configuração.

Uma modificação necessário para a aplicação, foi o gerenciamento do pino CS que agora fica a critério da implementação CAN. Esta modificação foi necessária para o correto gerenciamento do MCP2515 que pode receber um conjunto de dados de até 10bytes consecutivos, e neste período CS deve ser mantido em nível lógico baixo. Como a implementação anterior do SPI gerenciava o CS, toda vez que 1 byte era enviado, o CS era colocado novamente em nível lógico alto e o MCP2515 não era capaz de interpretar o conjunto de dados enviados pela aplicação CAN.

Uma nova função de leitura bloqueante foi implementada para gerir o recebimento dos dados. Esta função basicamente verifica se existe algum dado no *buffer* de leitura, e se acaso essa afirmação continuar sendo valida mesmo após a leitura dos dados, um laço com esta mesma verificação faz o papel de limpeza do *buffer*, fazendo com que apenas o ultimo valor (o valor válido desejado) seja armazenado na variável de retorno da função.

Esta função se fez necessária logo nos primeiros testes onde foi identificada a leitura errada dos valores que eram observados pelo osciloscópio. O problema ocorria na leitura sucessiva de bytes, e apesar dos dados que estavam transitando nos fios SPI serem os esperados, o valor retornado pela função de leitura não correspondia aos mesmo e desta forma a função para "limpar"o *buffer* foi implementada, corrigindo o erro até então observado. A seguir, é apresentado o trecho de código implementado.

```
1 Reg32 get_datamod(){
2         Reg32 data;
3         if((ssi(SSI_SR) & RNE)){
4             while((ssi(SSI_SR) & RNE)){
5                     data=ssi(SSI_DR);
6             }
7         }
8         return data;
9 }
```

Com as modificações acima realizadas, foi possível implementar o código responsável pelo gerenciamento CAN. Esta implementação é detalhada no tópico a seguir.

3.2.1 Implementação do Gerenciamento CAN

Para aplicar as funcionalidades idealizadas para o sistema operacional EPOS, foi determinado que, em primeiro momento, dois arquivos fossem implementados. O primeiro, o arquivo *can.h*, traz os métodos, membros, e definições utilizados na classe definida como *class CAN*. Já no segundo, o arquivo *can.cc*, os métodos são devidamente implementados para atender as exigências do trabalho em questão.

Muitas das funcionalidades do controlador MCP2515 (e consequentemente do protocolo CAN) não foram implementadas. O objetivo nesta etapa foi colocar o *sniffer* em funcionamento. Sendo um *sniffer* de rede, ele deve coletar todos os dados que trafegam no barramento, sendo assim, as máscaras e filtros não foram configurados. Além disso, variados modos de operação não foram implementados, e desta forma apenas os estados de configuração, modo normal de operação, e *listen-only* (destinado ao *sniffer*) são atingidos. Neste caso, o modo normal de operação se fez necessário para realização de testes de envio de mensagem para verificação do correto *setup* do *baud rate* do controlador, nada além disso.

Funcionalidade exclusivas do CI, como pinos de interrupção destinas individualmente para cada *buffer*, também não forma utilizados. Isto seria esperado, uma vez que o circuito apresentado na Figura 8 mostra que não há ligações físicas entre o CI e o EposMote III para estes pinos. Para fins de melhorias futuras, funções não utilizadas também foram declaradas, porém, não trazem relevância para este trabalho.

Os códigos completos dos arquivos apresentados nesta seção podem ser observados nos Apêndices A (*can.h*) e B (*can.cc*) deste trabalho. Abaixo, na Tabela 3 são descritas as variáveis (membros da classe CAN) utilizadas nesta solução:

Tabela 3 – Tabela de Membro de Classe CAN

Tipo	Variável	Funcionalidade
SPI	spi_	Utilizada para acesso aos métodos da Classe SPI.
GPIO	cs_can	Pin-out faz o gerenciamento do sinal CS destinado a comunicação SPI
GPIO	en_	Pin-out de gerenciamento de modo Standby do MCP2562
GPIO	interrupt_	Pin-in que recebe um sinal de interrupção gerado pelo MCP2615
GPIO	can_clock	Pin-in recebe o clock gerado pelo oscilador da placa de comunicação serial

Fonte: Autor (2017)

Estes membros são declarados como privados na classe CAN, e além destes, a implementação conta com algumas definições que remetem as instruções apresentadas na Tabela 2. Com pouca flexibilidade e faltando algumas funções para gerenciamento do controlador, essas definições podem e devem ser melhor implementadas utilizando técnicas como enumerações e funções para seleção de multiplas variações (como no caso das instruções *Read RX Buffer*, *Load TX Buffer*e *RTS* que dependem dos valores passados como parâmetro de preenchimento.

O construtor da classe foi criado de forma objetiva, onde não há parâmetros de entrada. O trecho de código desta implementação é mostrado abaixo:

```
1 CAN::CAN():spi_(0, Traits<CPU>::CLOCK, SPI::FORMAT_MOTO_0, SPI::
     MASTER, 2000000, 8),
2           en_('D', 0, GPIO::OUT, GPIO::UP, 0),//Enable MCP2562
3      interrupt_('A', 6, GPIO::IN),//Receives Interrupt
4      cs_can('A',3,GPIO::OUT),
5      can_clock('A', 7, GPIO::IN)
6 {
7      disable_EN();
8      init_MCP2515();
9 };
```

Sendo construtor responsável por setar as variáveis da Classe CAN, nota-se que a comunicação de interface SPI recebe valores como o clock do sistema (Epos opera à 32MHz), o Formato SPI (0,0 - Polaridade=0, Fase=0) uma vez que o MCP2515 suporta apenas os modos (0,0) ou (1,1), modo Master de operação, 2MHz para taxa de dados, e *data byte* como sendo 8.

Os métodos implementados e utilizados para testes e no *sniffer* de rede can são apresentados com uma breve descrição abaixo:

void init_MCP2515(): Chama a função **reset()**. Futuramente pode realizar outras configurações relacionadas ao baude rate e ao tipo de aplicação (para definir o modo de operação);

void reset(): Envia a instrução de RESET;

void config(): Configura os registradores CNFs para baude rate de 500KHz e clock de 32MHz;

void void config_filters(int MODE); Recebe o modo de operação em que o dispositivo irá atuar. Se for em modo Listen_Only todos os filtros e mascaras são desabilitados orientando os *buffers* para receberem qualquer tipo de mensagem. Se o modo for Normal, os filtros e mascaras são orientados a receber qualquer tipo de mensagem;

void void read_rx_buffer(bool n, bool m,char* data): Envia a instrução de leitura dos buffers. O parâmetro n=0 indica uma leitura no Buffer RX0, enquanto n=1 representa leitura no Buffer RX1. O parâmetro m=0, indica uma leitura dos 8bits mais significativos do identificador de mensagem Standard. Por sua vez, m=1 indica a leitura da primeira posição do Buffer (RXBnD0). A função mantem CS habilitado para receber os 8 valores contidos no buffer caso essa opção seja escolhida. O parâmetro *char* data* retorna esses valores;

void read_register(char* status_receive, char _register): Esta função envia a instrução READ e em seguida o endereço do registrador (parâmetro passado como *_register*). O parâmetro *status_receive* retorna os 8 bits lidos do registrador;

void write(char adress, char data): Envia a instrução WRITE para escrever no registrador desejado. Os parâmetros recebidos são o endereço do registrador (*adress*) e valor a ser escrito nele (*data*);

void load_tx_buffer(bool a, bool b, bool c, char* data): Função criada para teste de escrita no barramento através do buffer TX0. Os únicos parâmetros que esta função reconhece são: *bool c* que determina o carregamento do identificador ou dos dados (8 bytes no maximo), e o parâmetro *char* data*, que são os valores que serão escritos no buffer TX0;

void request_to_send(bool tx_0,bool tx_1,bool tx_2): Conforme a função anterior, esta função foi criada apenas para teste, e reconhece os comandos de requisição de envio para TX0, TX1 ou TX2, mas não para multiplas requisições (funcionalidade do MCP2515).

void bit_modify(char adress,char mask, char data): Envia a instrução BIT MODIFY seguida do endereço do registrador que se deseja modificar, a mascara de modificação, e o bits modificados. Alguns registradores não suportam esta ação. Ver datasheet do controlador MCP2515 para informações sobre esses registradores.

void spi_send(unsigned int): Faz o envido de um único byte toda vez que é chamada. Usa a função *void put_data(Reg32 data)* da classe SPI.

unsigned int spi_receive(): Recebe um byte em cada chamada, faz uso da função *Reg32 get_datamod()* antes de realizar a leitura através da função *Reg32 get_data(). Antes desta segunda chamada, envia um byte 0x00 para manter o clock SPI ativo durante a leitura.*

Conforme mencionado anteriormente a implementação destas funções podem ser vistas nos Apêndices deste trabalho. No tópico à seguir, é apresentada a implementação de aplicação do *sniffer* can.

3.2.2 Aplicação *Sniffer*

Após a definição e implementação das funcionalidades CAN, é estabelecido uma função principal para aplicação do *sniffer*. O arquivo que recebe a aplicação é chamado *can_sniffer_test.cc* e encontra-se no Apêndice C deste trabalho.

Esta aplicação possui duas funções auxiliares, sendo elas:

- void loop_can_getstatus();
- void read_buffer(CAN *can);

A função *main* apenas executa um *delay* inicial para atrasar as primeiras leituras do sniffer (tempo julgado importante para acompanhamento dos primeiros dados coletados) e posteriormente chama a função *loop_can_getstatus()* que além de possuir o loop principal da aplicação, realiza as configurações necessárias para seu correto funcionamento.

No início desta função são declaradas as variáveis de controle de Status e Interrupções e em seguida cria um tipo CAN e chama as funções de configuração (para CNFs e Mascaras e Filtros). Com as configurações iniciais realizadas, o modo de operação é modificado para Listen-Only e as flags de interrupção são limpas e habilitadas. O próximo passo é o loop principal de execução do sniffer que é apresentado abaixo:

```
1  while(1) {
2      can.read_register(&Status,0x0F); //read CANCTRL
3      can.read_register(&Status,0x0E); //read CANSTAT
4      can.read_register(&Interrupt, 0b00101100); // CANINTF
5      if((Interrupt & 0x01)== 0b00000001){
6          read_buffer(&can);
7          cout<<hex<<(unsigned int) Interrupt<<'\n';
8          can.bit_modify(0b00101100,0b11111111,0b00000000);
9      }
10 }
```

Os registradores de controle (CANCTRL) e status (CANSTAT) são lidos com a mesma finalidade, observação do comportamento no osciloscópio. Já a leitura do registrador de flags de interrupção (CANINTF) é realizada para fins de identificação de dados nos buffer RX. Em seguida é verificado se o buffer RX0 gerou alguma interrupção, ou seja, recebeu algum dado. A comparação é realizada com o ultimo bit do byte de CANINTF que remete ao buffer RX0, mas isso não significa que outras flags não possam ter sido geradas.

Caso a flag seja verdadeira, a função read_buffer(CAN*) é chamada. Essa função por sua vez realiza 3 tipos de leitura em sequência. Primeiro lê os 8 bits mais

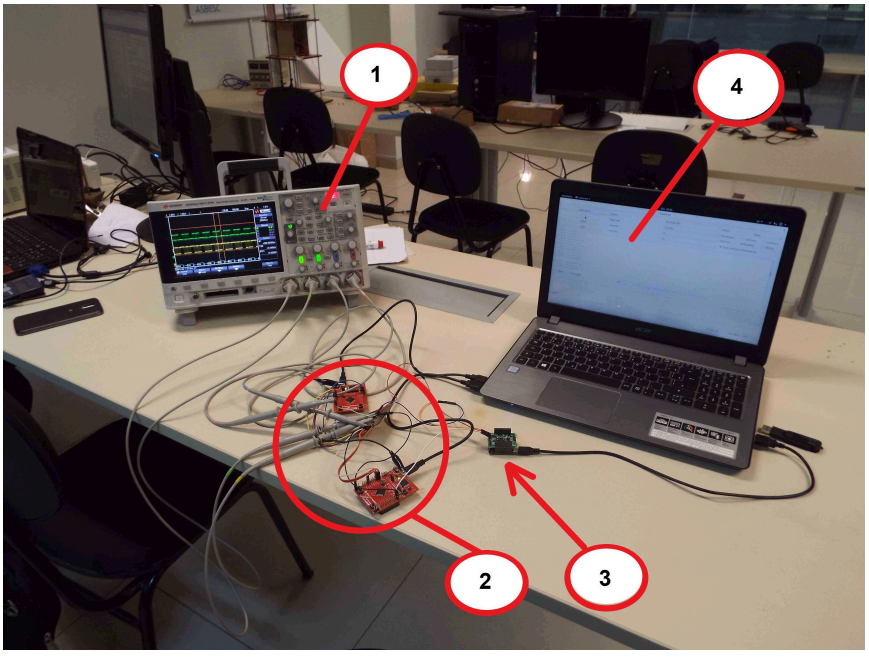
significativos do identificador da mensagem contida no buffer (RXB0SIDH) através da função *read_rx_buffer*, em seguida lê os 3 bits menos significativos diretamente no registrador responsável (RXB0SIDL) com auxilio da função *read_register*. Esse dois valores obtidos são armazenados na mesma variável do tipo *unsigned int* e compõem o identificador completo da mensagem. Os 8 bytes de dados são lidos por ultimo usando a função *read_rx_buffer*, com acesso direto a primeira posição do buffer RXO (ou seja RXB0D0) e os demais dados são obtidos realizando as leituras de forma consecutiva mantendo CS habilitado. Esta última parte é gerida pela própria função de leitura, conforme mencionado no tópico anterior.

Antes de finalizar esta função, o ID (identificador) e os dados obtidos são enviados via USB para o computador e somente após esta ação, retorna para o loop principal. E no loop principal, o valor anteriormente obtido das flags de interrupção, são também enviados para o computador. Essas flags podem inticar não somente interrupções dos buffers, mas também dos diversos tipos de erros identificados na comunicação CAN. Por ultimo, antes de reiniciar o laço, as flags de interrupção são limpas.

3.3 Bancada de Testes - Emulação de Barramento CAN

A Figura 9 mostra o conjunto completo utilizado para o experimento na bancada de testes.

Figura 9 – Conjunto Experimental Completo



Fonte: Autor (2017)

Na Figura 9, são apresentados os seguintes itens: 1. Osciloscópio: usado para verificação dos dados que trafegam no barramento; 2. Conjunto CAN/Tiva: usado para emulação de barramento CAN; 3. Conjunto EposMote/Serial Board: suporte hardware para implementação do *sniffer*; 4. Computador Pessoal: usado como interface de verificação dos dados recebidos pelo *sniffer*.

O papel do nó mestre neste barramento experimental consistem em ficar enviando uma série de *frames standard* com identificadores e dados conhecidos para posterior validação do correto recebimento pelo *sniffer*. O nó escravo é aplicado com intuito de gerar o sinal de recebimento (ACK). Caso isto não ocorra, por definição, o nó mestre tentara sempre enviar o mesmo *frame* de dados uma vez que a falta de ACK é relacionada como sendo um erro na recepção da mensagem. Pode-se ressaltar que o dispositivo *sniffer* age de forma passiva no barramento e por isso não envia quaisquer tipo de sinal de verificação.

O código completo gerado nesta etapa tanto para o nó mestre, como para o nó escravo, pode ser observado nos Apêndices deste trabalho. São então enviados 12 *frames* com identificadores de 0x6D0 à 0x6DB com dados diferentes para cada identificador. A saída desses *frames* do microcontrolador ocorrem via CAN TX e para que um barramento seja efetivamento comparado ao esperado no ônibus elétrico, transceptores são alocados nas saídas destes microcontroladores, conforme mencionado anteriormente.

A seguir são apresentados o conjunto de dispositivos utilizados para emulação barramento CAN.

3.3.1 Microcontrolador Tiva C Séries

A série TM4C, assim como para a maioria dos microcontroladores, são anexadas a placas Lauchpad. Todas as placas LaunchPad desta série possuem emulação on-board para código de programação e depuração, botões e LEDs, bem como um conector que aceita módulos plug-in BoosterPack, que pode adicionar novas funcionalidades ao LaunchPad, como conectividade sem fio, LEDs, sensores, entre outros (TEXAS INSTRUMENTS, 2017a).

O LaunchPad TM4C123G é uma plataforma de baseada em um ARM Cortex-M4F da Texas Instruments. O design do TM4C123G LaunchPad usa o microcontrolador TM4C123GH6PM com uma interface de dispositivo USB 2.0 e um módulo de hibernação. Os cabeçalhos da interface facilitam e simplificam a expansão da funcionalidade do TM4C123G ao conectar-se a outros periféricos (TEXAS INSTRUMENTS, 2017a).

Sendo este microcontrolador bem estabelecido no mercado com bibliotecas que auxiliam no desenvolvimento de um barramento CAN, faz com que este dispositivo

se torne ideal para validação do sniffer proposto neste trabalho. Esta implementação é facilitada também por sua biblioteca TivaWare, que fornece as funções do driverlib o qual também possui abstrações para implementação de redes CAN.

3.3.2 Code Composer Studio

O Code Composer Studio (CCS) é um ambiente de desenvolvimento integrado (IDE) que suporta o portfólio de microcontroladores e processadores da Texas Instruments. Inclui um conjunto de ferramentas usadas para desenvolver e depurar aplicações embarcadas. Ele inclui um compilador C otimizado, editor de código-fonte, ambiente de criação de projeto, depurador, entre outros recursos (TEXAS INSTRUMENTS, 2017b).

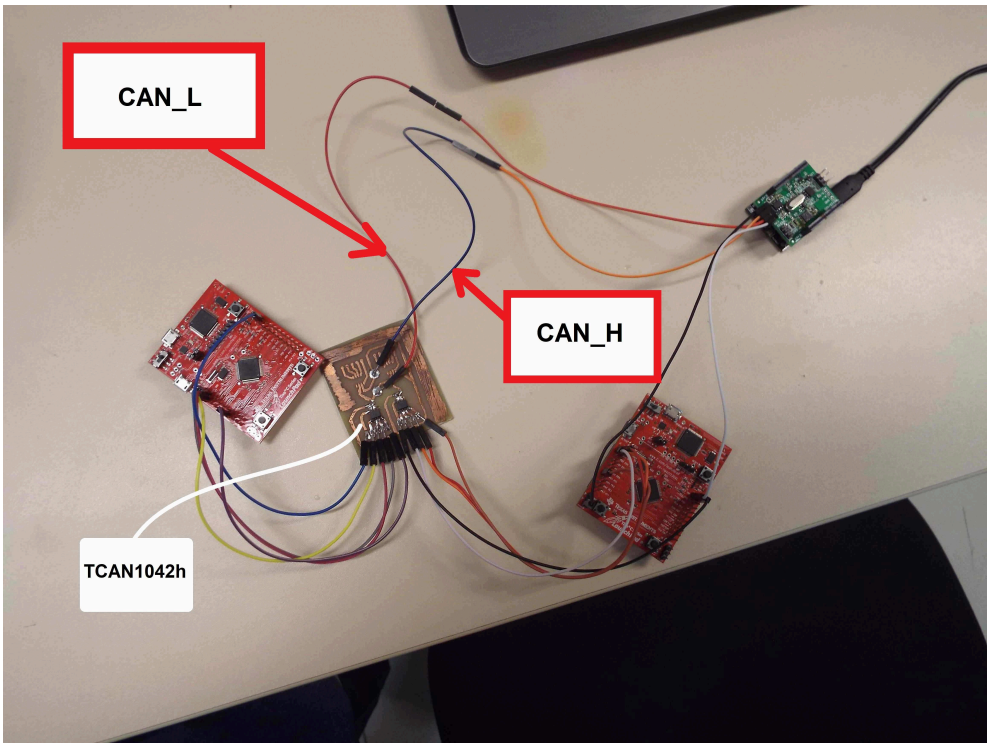
Os microcontroladores utilizados no conjunto experimental foram programados usando CCS com auxilio da biblioteca TivaWare que possui uma série de implementações que abstraem o uso de suas funcionalidade/periféricos.

3.3.3 Transceptor CAN - TCAN1042h

Esta família de transceptores do CAN atende ao padrão de camada física de alta velocidade CAN (TEXAS INSTRUMENTS, 2007). Todos os dispositivos são projetados para uso em redes CAN FD de até 2 Mbps. Possui um modo de baixo consumo de energia com recurso de solicitação remota para *wake-up*.

Em relação as suas características principais, o CI possui dois modos de operação que seriam o modo Normal e Standby. Os modos de operação ocorrem de forma idêntica ao MCP2562 utilizado no circuito de comunicação serial CAN acoplado ao EposMote. Na Figura 10 é apresentada a conexão entre os dispositivos.

Figura 10 – Conjunto EposMote III e Serial Board - *Sniffer*



Fonte: Autor (2017)

Na imagem, os fios são interligados por uma placa. O objetivo de sua concepção, foi alocar o dispositivo apresentado neste tópico e também usa-la como ponte de conexão para o barramento CAN de teste.

4 RESULTADOS E DISCUSSÕES

Neste capítulo são discutidos os resultados obtidos na implementação do *sniffer* de barramento CAN proposto para este trabalho. Em primeiro momento, foram realizados testes em bancada conforme apresentado na Seção 3.3. Após correções, o teste no ônibus elétrico foi realizado. Essas etapas são apresentadas à seguir juntamente com a análise dos resultados obtidos.

4.1 Testes em Bancada

Os primeiros testes do *sniffer* CAN, foram aplicados ao conjunto experimental de emulação de barramento CAN. Utilizando uma aplicação simples de envio de mensagens no modo normal de operação do MCP2515, verificou-se o *baude rate* definido nas configurações do dispositivo. Seguindo as configurações de BRP=0 para o cristal oscilador de 8 MHz (conforme apresentado no esquemático do circuito CAN da Figura 14 no Capítulo 3) para atingir 16 Tq (*Time Quantum*), obteve-se 250 KHz para taxa de dados (seguindo as equações 1 e 2 do Capítulo 3).

O barramento CAN dos microcontroladores, foram então configurados para trabalhar à uma taxa de 250 KHz. Para o primeiro teste prático apenas 1 identificador de mensagem foi definido. A mensagem escolhida teve seu ultimo bit modificado a cada envio (em modo crescente). Os valores obtidos pelo *sniffer* desenvolvido para esta taxa de dados obteve o resultado esperado conforme mostra a Tabela 4.

Tabela 4 – Primeiro Data Log do Sniffer CAN

N	Data Log
1	13,1122334455667700
2	13,1122334455667701
3	13,1122334455667702
4	13,1122334455667703
5	13,1122334455667704
6	13,1122334455667705
7	13,1122334455667706
8	13,1122334455667707
9	13,1122334455667708
10	13,1122334455667709
11	13,112233445566770a
12	13,112233445566770b
13	13,112233445566770c
14	13,112233445566770d
15	13,112233445566770e
16	13,112233445566770f
17	13,1122334455667700
18	13,1122334455667701

Fonte: Author(2017)

Com os teste de bancada realizado, a próxima etapa foi aplicar o sniffer no barramento CAN do ônibus elétrico. Esta etapa é apresentada no tópico à seguir

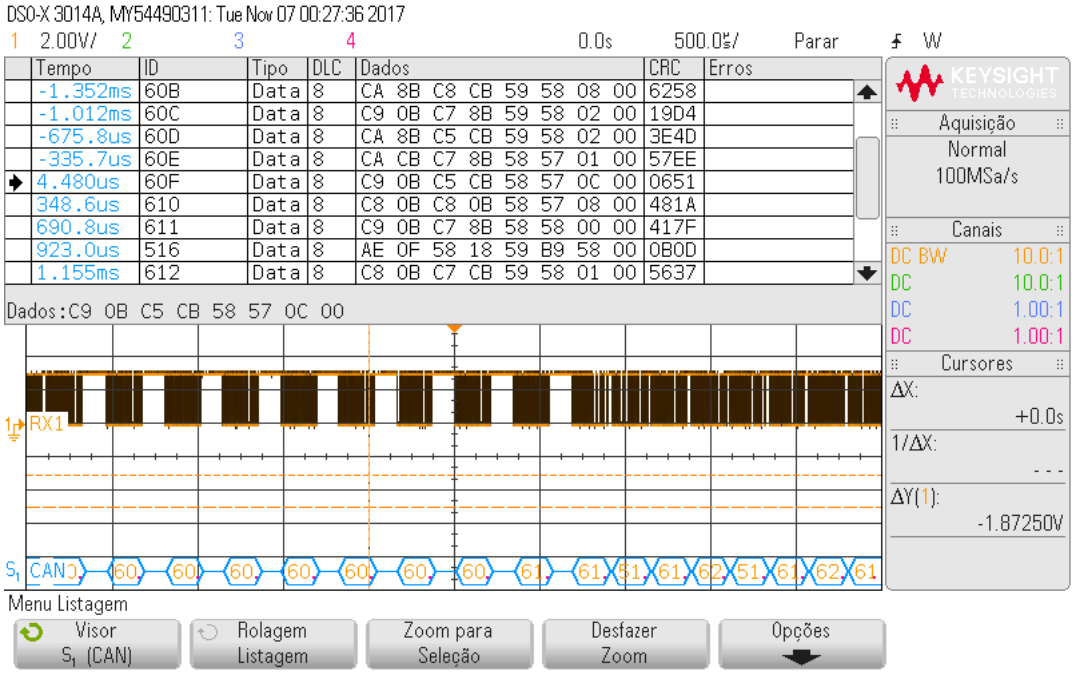
4.2 Testes no ônibus elétrico

Para fins de verificação, o osciloscópio foi utilizado para identificar o conjunto de dados que transitavam no barramento. O primeiro teste realizado, foi para identificar qual era a taxa de dados real que o barramento operava. A Figura 11, mostra o osciloscópio realizando a interceptação dos dados do barramento CAN, durante os testes em campo. Na Figura 12, é apresentada a imagem obtida da interface do osciloscópio durante este processo.

Figura 11 – Osciloscópio conectado ao barramento do Ônibus

Fonte: Autor (2017)

Figura 12 – Imagem dos dados do barramento CAN do EBus capturada pelo osciloscópio



Fonte: Autor (2017)

As informações coletadas através do osciloscópio foram salvas para posterior comparação com *sniffer* desenvolvido.

A leitura apresentada foi importante para determinar algumas informações desejadas para a aplicação. Primeiramente, foi observado que uma grande quantidade de ruído se faz presente no barramento do ônibus elétrico. Porém, uma das principais características do barramento é a sua imunidade ao ruído, devido a sua estrutura CAN_L e CAN_H. Os dois fios transmitem os valores de forma diferencial. E o par trançado faz com que o ruído se propague igualmente pelo dois fios do barramento. Com a leitura diferencial, o ruído é anulado, e o valores transmitidos são obtidos, mesmo com a presença de ruído. Parte dos dados obtidos nesta leitura são apresentados na Tabela 5 abaixo:

Legenda de Erros:

CRC : erro no *checksum*. (CRC) *Cyclic Redundancy Check*,

Form : Se um nó detectar um bit dominante em um dos quatro segmentos (incluindo *end-of-frame*, *interframe*, o delimitador de confirmação ou o delimitador CRC), ocorreu um erro de formato e gerou um *frame* de erro. A mensagem é repetida (*Form error*).

Frame : mensagem de erro gerada. Pode ser proveniente de um erro detectado pelo transmissor (*Bit error*) ou detecção de mais de 6 bits recessivos consecutivos (*Stuff error*);

Fo Fr : erro de confirmação, nenhum nó recebeu corretamento o *frame*. Algum nó solicita o reenvio (*Acknowledge Error*).

Tabela 5 – Dados obtidos pelo Osciloscópio

Time	ID	Type	DLC	Data	CRC	Errors
-175.0ms	415	Data	2	00 00	2F2D	
-174.1ms	556	Data	8	A6 0F 58 26 58 2F 58 00	6C87	
-173.9ms	6A7	Data	8	C1 0B BD 4B 56 AE 00		Fo Fr
-173.6ms	1EF	Data	8	00 00 00 00 00 00 00 00	59D9	
-173.4ms	557	Data	8	E8 A3 0F 08 A9 0F 56 00	3D7B	Fo Fr
-170.3ms	557	Data	8	E8 A3 0F 08 A9 0F 56 00		Fo Fr
?						Form
-169.9ms	6A7	Data	8	C1 0B BD 4B 57 57 00 00	00C6	
-169.5ms	6A8	Data	8	C0 CB BD 8B 57 57 00 00	08A6	

-169.2ms	6A9	Data	8	C1 0B BF 4B 57 56 08 00	57F7	Fo Fr
-168.9ms	6A9	Data	8	C1 0B BF 4B 57 56 08 00		Fo Fr
-168.7ms	6A9	Data	8	C1 0B BF 4B 57 56 08 00	57B7	CRC
-168.5ms	401	Data	8	A7 FF B1 FF 00 00 00 00	3B0D	
-168.2ms	6AA	Data	8	C0 8B BC 0B 58 57 02 00	3D02	
-167.9ms	6AB	Data	8	C1 0B BD CB 58 58 00 00	5C4F	
-167.5ms	6AC	Data	8	C0 8B BD 0B 58 57 01 00		Frame
-167.3ms	6AC	Data	8	C0 8B BD 0B 58 57		Fo Fr

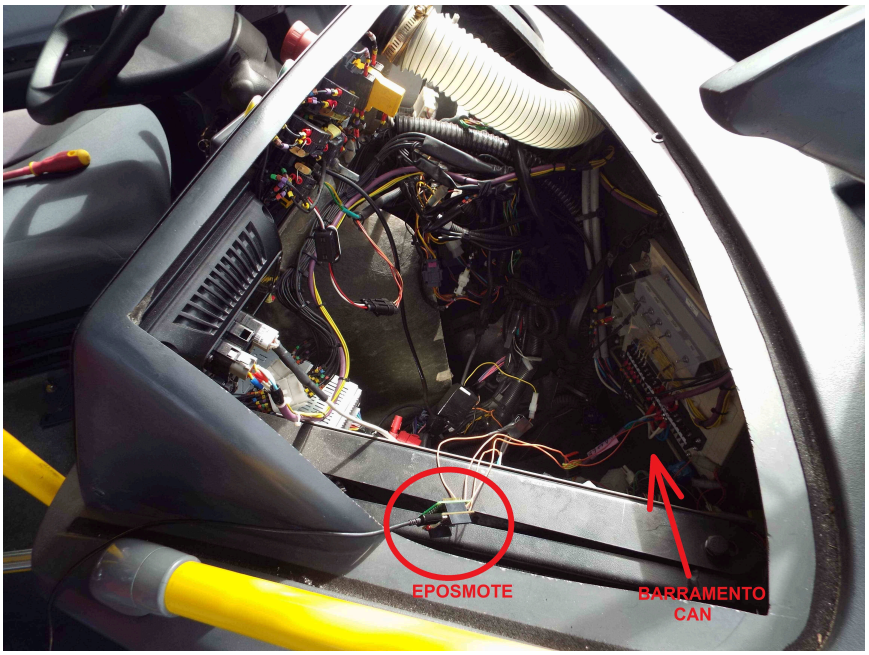
Fonte: Autor (2017)

Outro ponto importante nessa etapa, foi a identificação da taxa de dados do barramento CAN presente no ônibus. Foi verificado que o barramento opera à uma frequência de 500 KHz. Desta forma, foi necessário reconfigurar os registradores que determinam o *Time Quantum* do controlador CAN. Porém, o cristal de 8MHz, anexado à placa de comunicação serial, não é capaz de atender a este requisito. Logo, foi necessário substituí-lo por um novo cristal, de 32MHz. Esta escolha, surgiu com o intuito de realizar uma modificação permanente. Sendo que a máxima frequência de operação do protocolo CAN, é de 1 MHz. E este cristal é capaz de fornecer esta frequência.

Com o novo cristal anexado à placa de comunicação serial, os novos cálculos para obtenção da taxa de dados, de 500 KHz, foi realizada. E o resultado mostrou que, a escolha de um BRP=1, para o oscilador de 32 MHz, para atingir um tempo de bit nominal de 16Tq, gerou a taxa de dados desejada.

Conectando o EposMote III ao barramento CAN, foram realizados os testes do *sniffer* conforme proposta do projeto. A imagem da Figura 13 apresenta o dispositivo conectado ao barramento CAN do EBus.

Figura 13 – Sniffer no barramento CAN do EBus



Fonte: Autor (2017)

Alguns dados coletados pelo *sniffer* CAN são apresentados na Tabela 6 abaixo, e representam de forma parcial os testes realizados no ônibus elétrico.

Tabela 6 – Dados coletados pelo sniffer no barramento CAN do EBus

ID	Mensagem	Interrupção*
0x563	c832c883fbaa454	21
0x1e3	007f7070b452	1
0x1eb	00020000	1
0x603	10000000	1
0x533	43969659b526e80	1
0x6ab	484a44ca585620	a1
0x6b3	434a41ca595820	1
0x543	1888c235800	21
0x6cb	43a428a595880	1

* CANINT (INTERRUPT FLAG). Ver capítulo 7 no datasheet do dispositivo MCP2515.

Fonte: Autor (2017)

Comparando os dados apresentados na Tabela 6 com a imagem obtida pelo osciloscópio (mostrados na Figura 12 e na Tabela 5), é observado uma incoerência nos dados coletados pelo *sniffer*. O primeiro ponto a ser tratado, é o fato de que, ao contrário dos valores obtidos no osciloscópio, não existe uma sequência de IDs real nos dados, como por exemplo: 6A7, 6A8, 6A9. Além disso, por definição do protocolo,

a cada erro identificado, o *frame* deve ser reenviado, como é observado nos dados coletados no osciloscópio. Os dados apresentados na Tabela 6 não seguem esses princípios.

Desta forma, o próximo tópico apresenta o resultado de um novo teste proposto para identificar as falhas reconhecidas na aplicação do *sniffer* de rede CAN.

4.3 Desempenho do *Sniffer*

Em análise, observou-se que, o *sniffer* realizava o envio de cada dado interceptado para o computador, assim que estes eram obtidos. O EposMote faz a externalização do dados via comunicação UART, a uma taxa de 115200 bits/s, ou seja, 4 vezes menor que a frequência de operação do barramento CAN. A comunicação SPI, entre EposMote e MCP2515, foi configurada à uma frequência de 2 MHz.

Para ocorrer a transferência dos dados do controlador MCP2515 para o EposMote é necessário o envio de 3 *bytes* de comando, usando a SPI. Sendo que o primeiro *byte*, requisita os bits mais significativos do identificador da mensagem, que retorna *1 byte*. O segundo *byte* de comando, requisita os bits menos significativos do mesmo identificador, que retorna 1 *byte*. Por fim, o ultimo byte de comando, faz a requisição dos dados da mensagem, que podem ser de até 8 bytes. Além destes, 2 bytes de endereço e leitura são enviados, para leitura do registrador que gerencia as flags de interrupção. Esta leitura, retorna 1 byte com as interrupções geradas. Desta forma, é necessário transferir 16 *bytes* através da SPI, para que a aquisição completa dos dados de uma única mensagem ocorra.

O controlador recebe aproximadamente 1000 quadros/seg do barramento CAN, ou seja, é necessário a transferência de 16000 bytes por segundo (128 kbps), na SPI. Isso implica em um 1 bit transferido à cada 0,064 segundos, ou seja, a comunicação consome certa de 6,4% do tempo de processamento da CPU.

O EposMoteIII opera a UART, à uma taxa de 115200 bits/s. Sendo essa, usada na externalização dos dados. Nesta transferência, é realizada a transmissão de 2 bytes referentes ao ID da mensagem, 8 bytes de dados e 1 byte referente à flag de interrupção. Um total de 11 bytes são externalizados, para cada mensagem interceptada pelo sniffer. Desta forma, aproximadamente 76% do tempo de processamento da CPU é consumido, para esta comunicação.

Os valores apresentado evidenciam o problema encontrado na externalização dos dados. Considerando uma configuração de 500000 kbits/s para a comunicação UART, este consumo de CPU reduziria para aproximadamente 17%. Porém, não foram encontrados, nos códigos do Epos, como a UART sobre a USB é configurada. E desta forma, esta configuração não pode ser realizada.

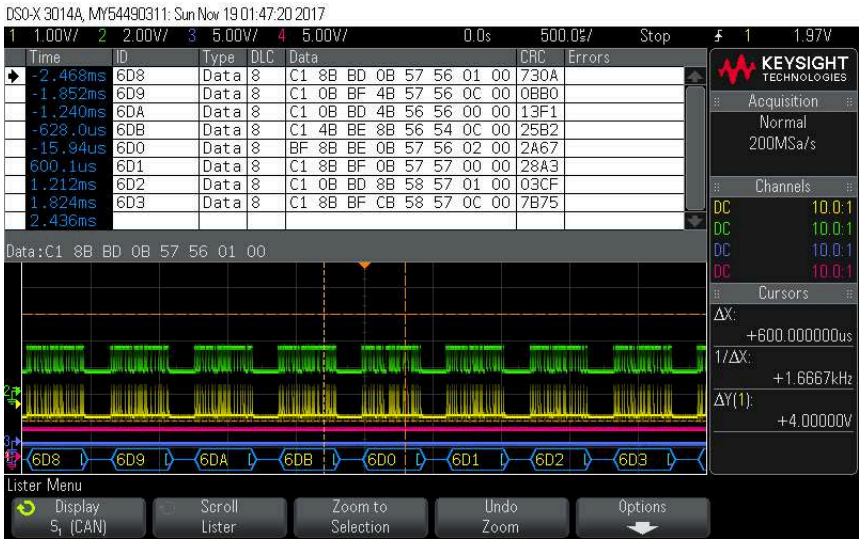
Outra questão para ser abordada, é o porque este problema não havia sido

identificado nos testes de bancada. A principio, para que os dados que trafegavam no barramento CAN simulado pudessem ser facilmente identificados (instruções de leitura e escrita, inicio e fim de *frame*), foi colocado um pequeno *delay* antes de cada *frame* enviado pelo microcontrolador configurado como nó mestre. Este *delay* era suficientemente grande para que os dados fossem enviados para o computador sem que ocorressem perdas na leitura.

Sendo assim, um novo teste foi realizado. O teste consistiu em remover o *delay* do nó mestre. E ao invés de enviar sempre o mesmo identificador e mensagem, ele enviara uma sequência de 12 *frames* com IDs diferentes e com variados valores de dados em suas mensagens. Estes novos *frames*, foram baseados em um conjunto pequeno de dados retirados da leitura efetuada pelo osciloscópio. Esta mensagens, correspondem aos identificadores de 6D0 à 6DB.

Desta forma, o *sniffer* de rede CAN realizou um total de 12 leituras no barramento CAN e somente ao término dessas leituras ele enviou os dados para o computador. Todos os códigos apresentados nos apêndices deste trabalho correspondem as últimas modificações realizadas, no código de aplicação (*"can_sniffer_test.cc"*) os trechos de código removidos encontram-se comentados. A imagem obtida pelo osciloscópio, durante esse teste, é apresentada na Figura 14.

Figura 14 – Nova sequência de Frames para Testes - Osciloscópio



Fonte: Autor (2017)

O novo teste foi realizado e os dados obtidos são apresentados na Tabela 7.

Tabela 7 – Dado do sniffer com aplicação corrigida

ID	Mensagem	Interrupção
0x6d5	c1bc0cb5856c0	1
0x6d6	c2cbbeb5754830	1
0x6d7	c14bc0b575610	1
0x6d8	c18bbdb575610	1
0x6d9	c1bbf4b5756c0	1
0x6da	c1bbd4b565600	1
0x6db	c14bbe8b5654c0	1
0x6d0	bf8bbeb575620	1
0x6d1	c18bbfb575700	1
0x6d2	c1bbd8b585710	1
0x6d3	c18bbfcb5857c0	1
0x6d4	c18bbf4b585710	1
0x6d8	c18bbdb575610	1

Fonte: Autor (2017)

Conforme esperado, os valores obtidos em lotes de 12 frames correspondem à uma sequência real dos dados trafegando no barramento CAN. Também era esperado, que durante o envio destas leituras, para o computador, os demais dados (reenviados) seriam perdidos (intervalo entre o último e o penúltimo dado apresentado na Tabela 7).

Uma possível solução para o problema encontrado, seria a redução no numero de mensagens à serem externalizadas. Em geral, as aplicações de *sniffer* buscam interceptar um determinado conjunto de dados. Através da priorização dos IDs de interesse, é possível reduzir o numero de dados interceptados pelo *sniffer*. Como o objetivo da implantação do *sniffer* é o monitoramento remoto do ônibus elétrico, nem todos os dados que trafegam no barramento CAN são interessantes para a aplicação. Alguns identificadores podem ser priorizados através das mascaras e filtros de mensagens do controlador CAN. Por exemplo, o *sniffer* pode receber apenas os dados referentes ao consumo atual de energia, ao estado das baterias, a velocidade, rotação do motor e posição (GPS). Isto faz com que o numero de quadros obtidos seja drasticamente reduzido. Além disso, as mensagens são repetidas a cada segundo. Para um monitoramento de consumo de energia, por exemplo, não haveria necessidade de verificar os dados a cada segundo. Isto reduziria ainda mais a influencia da externalização no tempo de processamento no EposMoteIII.

Os IDs são de caráter proprietário, e até a conclusão deste trabalho, não foram identificados. Os catalogos SAE J1939-x, trazem informações sobre os dispositivos que utilizam interface CAN, em diversas aplicações, porém não apresentam os valores dos IDs. Devido este fato, a relação das mensagens obtidas com as suas respectivas aplicações, não são apresentadas neste trabalho.

5 CONCLUSÕES E TRABALHOS FUTUROS

O trabalho foi proposto com o objetivo de habilitar o sistema embarcado EposMotes III para realizar a interceptação dos dados que trafegam no barramento CAN do ônibus elétrico do grupo Fotovoltaica UFSC, localizada no Sapiens Parque em Florianópolis, Santa Catarina.

Inicialmente foram apresentados ao leitor os fundamentos e conjunto experimental utilizado no desenvolvimento do *sniffer* de rede CAN. O gerenciamento do controlador e transceptor CAN foi estabelecido, atingindo o primeiro objetivo proposto. A bancada de testes desenvolvida, mostrou-se eficiente a ponto de identificar a maior parte dos problemas, antes que o experimento no ônibus elétrico fosse realizado. O *sniffer* desenvolvido foi instalado no barramento CAN do ônibus elétrico, e a avaliação do seu funcionamento foi realizada. O problema da aplicação, onde os dados foram perdidos devido ao consumo de tempo de CPU, foi devidamente verificado e justificado. A externalização dos dados, utilizando uma comunicação UART, com taxa de transmissão de 115200 bits/s, impossibilita o uso do *sniffer* desenvolvido. Desta forma, o objetivo geral deste trabalho não foi atingido.

A continuidade deste projeto está relacionada ao monitoramento remoto do ônibus elétrico. Mesmo que uma nova versão do *sniffer* não utilize a comunicação serial UART (sobre a USB), em sua configuração atual, o EposMote deverá realizar a externalização desses dados por outros meios (comunicação sem fio, por exemplo), e o problema deverá persistir. Sendo assim, a externalização dos dados será um desafio a ser estudado.

Para trabalhos futuros, é sugerido realizar as melhorias no código de implementação CAN do Epos, podendo estender as funções já utilizadas para deixar o gerenciamento mais flexível. Aplicar as funcionalidades não exploradas do controlador MCP2515 como, por exemplo, o uso de mascaras e filtros de mensagens. Desenvolver uma aplicação que realiza o envio e recebimento de frames de dados e frames de requisições em um barramento CAN qualquer (ou seja, um nó CAN ativo). Sendo assim, é estimulada a reformulação dos códigos aplicados ao sistema operacional Epos utilizados neste trabalho. E no futuro, esta nova implementação do Epos poderá se valer de todos as vantagens do SO, como modelar threads em tempo real.

REFERÊNCIAS

BOSCH, R. et al. **CAN specification version 2.0**. [S.l.]: Stuttgart, 1991.

ELETRA. **Manual de manutenção: Veículo Elétrico, Universidade Federal de Santa Catarina 12 metros**. [S.l.], 2016.

ESACADEMY. **Everything about the CAN bus or Controller Area Network**. 2017. Https://www.canbus.us/. [Acesso em: 09 nov 2017].

FV-UFSC. **Grupo de Pesquisa Estratégica em Energia Solar da Universidade Federal de Santa Catarina**. 2017. <http://fotovoltaica.ufsc.br>. [Acesso em: 09 nov 2017].

GUIMARÃES, A. de A. **CAN bus: Conceituação**. [S.l.]: de Almeida Guimarães, Alexandre, 200–. <http://www.alexag.com.br/CAN_Bus_Parte_2.html>.

GUIMARÃES, A. de A. **Eletrônica embarcada automotiva**. [S.l.]: Érica, 2007.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO). **Road Vehicles: controller area network (CAN)**. [S.l.]: ISO 11898-1:2003, 2003. <https://www.iso.org/standard/33422.html>.

LISHA - SOFTWARE/HARDWARE INTEGRATION LAB. **EPOS - Embedded Parallel Operating System**. 2017. <http://epos.lisha.ufsc.br/EPOS+Hardware>. [Acesso em: 10 nov 2017].

LISHA - SOFTWARE/HARDWARE INTEGRATION LAB. **LISHA - Serial communication board**. 2017. <https://lisha.ufsc.br>. [Acesso em: 15 nov 2017].

LISHA - SOFTWARE/HARDWARE INTEGRATION LAB. **LISHA - Software/Hardware Integration Lab**. 2017. <https://lisha.ufsc.br/HomePage>. [Acesso em: 10 nov 2017].

MICROCHIP TECHNOLOGY. **Datasheet: MCP2515**: Stand-alone can controller with spi interface. [S.l.]: Microchip Technology, 2005. <http://ww1.microchip.com/downloads/en/DeviceDoc/21801e.pdf>.

MICROCHIP TECHNOLOGY. **Datasheet: MCP2562**: High-speed can transceiver. [S.l.]: Microchip Technology, 2005. <http://ww1.microchip.com/downloads/en/DeviceDoc/20005167C.pdf>.

NAVET, N.; SIMONOT-LION, F. **Automotive embedded systems handbook**. [S.l.]: CRC press, 2008.

PARET, D. **Multiplexed networks for embedded systems: CAN, LIN, Flexray, Safe-by-Wire...** [S.l.]: John Wiley & Sons, 2007.

SIEMENS. **AP 2921 On-Board Communication via CAN without Transceiver**. [S.l.]: Siemens Microcontrollers ApNote, 1996. <https://www.mikrocontroller.net/attachment/28831/siemens_AP2921.pdf>.

TEXAS INSTRUMENTS. **Datasheet: TCAN1042**. [S.l.]: Texas Instruments Datasheets, 2007. <http://www.ti.com/lit/ds/sllses7c/sllses7c.pdf>.

TEXAS INSTRUMENTS. **ARM Cortex-M4F Based MCU TM4C123G LaunchPad Evaluation Kit**. 2017. <http://www.ti.com/tool/EK-TM4C123GXL>. [Acesso em: 10 nov 2017].

TEXAS INSTRUMENTS. **Code Composer Studio (CCS) Integrated Development Environment (IDE)**. 2017. <http://www.ti.com/tool/ccstudio>. [Acesso em: 15 mar 2017].

VIEIRA, A. J. F. F. et al. **Breve descrição procolo de comunicações CAN**. [S.l.]: Microchip Technology, 2007. <https://web.fe.up.pt/~ee99058/projecto/>.

WANG, X.; YAO, W.; SHI, G. A control system of electric vehicle based on can bus. In: IEEE. **Advanced Mechatronic Systems (ICAMechS), 2011 International Conference on**. [S.l.], 2011. p. 580–582.

APÊNDICE A – CÓDIGO DE GERENCIAMENTO CAN - CAN.H

```
1  // can.h
2
3  // EPOS - CAN (Gerenciamento MCP2515 em conjunto com MCP2562)
4
5  #include <alarm.h>
6  #include <gpio.h>
7  #include <spi.h>
8  #include <machine.h>
9  #include <cpu.h>
10 #include <utility/ostream.h>
11
12
13 /***********SPI INSTRUCTION FOR CAN COMMUNICATION *********/
14 /* Para que as instrucoes sejam reconhecidas o PIN CS referente a
      comunicacao SPI deve estar estar em nivel baixo '0' durante a
      instrucao e logo em segui seu nivel logico deve retornar '1' */
15 #define RESET 0b11000000
16 #define READ  0b00000011
17 //#define READ_RX_BUFFER 0b10010nm0
18 #define WRITE 0b00000010
19 //#define LOAD_TX_BUFFER 0b01000abc
20 //#define RTS 0b10000nnn
21 #define READ_STATUS 0b10100000
22 #define RX_STATUS 0b10110000
23 #define BIT_MODIFY 0b00000101
24 /*********************************************************/
25
26
27 __BEGIN_SYS
28 class CAN{
29     public:
30         CAN(); //Construtor
31         ~CAN(); //Destrutor
32
33         void enable_EN();
34         void disable_EN();
35
```

```
36          void init_MCP2515();
37          void reset();
38          void config();
39          void config_filters(int MODE);
40          void read_rx_buffer(bool n, bool m,char* data);
41          void read_register(char* status_receive, char _register);
42          void write(char adress, char data);
43          void load_tx_buffer(bool a, bool b, bool c, char* data);
44          void request_to_send(bool tx_0,bool tx_1,bool tx_2);
45          void bit_modify(char adress,char mask, char data);
46
47          //Functions SPI
48          void spi_send(unsigned int);
49          unsigned int spi_receive();
50      private:
51          // Variaveis do Protocolo
52          SPI     spi_;
53          GPIO    en_; //Enable (PIN-OUT)
54          GPIO    interrupt_;//Interrupt (PIN-IN)
55          GPIO    cs_can;
56          GPIO    can_clock;
57  };
58  __END_SYS
```

APÊNDICE B – CÓDIGO DE GERENCIAMENTO CAN - CAN.CC

```
1  // can.cc
2
3  // EPOS - Can Abstraction Implementation
4  #include <can.h>
5
6  __BEGIN_SYS
7  // Methods
8  CAN::CAN():spi_(0, Traits<CPU>::CLOCK, SPI::FORMAT_MOTO_0, SPI::
       MASTER, 2000000, 8),
9          en_('D', 0, GPIO::OUT, GPIO::UP, 0),//Enable MCP2562
10         interrupt_('A', 6, GPIO::IN),//Receives Interrupt
11         cs_can('A',3,GPIO::OUT),
12         can_clock('A', 7, GPIO::IN)
13 {
14     disable_EN();
15     init_MCP2515();
16 };
17
18 //****** SET CAN FUNCTIONS WITH SPI INSTRUCTIONS *********/
19
20 //The MCP2515 must be initialized prior to activation. Its
       initialization will occur after RESET and SELECT MODE
21 void CAN::init_MCP2515(){
22     cs_can.set(0);
23     reset(); // Configuration Mode Select
24     while(spi_.is_busy()){}
25     cs_can.set(1);
26 }
27
28 // configuracao exclusiva para Oscilador com 32MHz e uma taxa de
       500KHz
29 void CAN::config(){
30     char CNF1=0b00101010;
31     char CNF2=0b00101001;
32     char CNF3=0b00101000;
33     /* According to CAN with baud rate of 500 Khz; Oscillator with
          32MHz; BRP = 0 to determine Tqs=16*/
```

```
34      write(CNF1, 0b00000001); //SJW=1*Tqs='00'; BRP=2='000001' //
           for 1MHz of baude set BRP = 0
35      write(CNF2, 0b10110001); //BTLMODE= Length of PS2=Definido em
           CNF3='1'; SAM='0'; PHSEG1=PS1 Lenght='000'; PRSEG=
           Propagation='001'
36      write(CNF3, 0b00000101); //SOF='0'; WAKFIL='0'; PHSEG2
           <2:0>='101'
37
38  }
39
40  void CAN::config_filters(int MODE){
41      if(MODE==1){ //Listen-Only mode
42          // Edit RXB0 (Buffer 0)
43          char RXB0CTRL=0b01100000;
44          char ADRESS_RXB0CTRL=0b01100000;
45          write(ADRESS_RXB0CTRL, RXB0CTRL);
46
47          // Edit RXB1 (Buffer 1)
48          char RXB1CTRL=0b01100000;
49          char ADRESS_RXB1CTRL=0b01110000;
50          write(ADRESS_RXB1CTRL, RXB1CTRL);
51          //Edit Mask and Filter off
52      }
53      else{ //Normal mode
54          /********** SET BUFFER0 **********/
55          char RXM0SIDH_ADRESS=0b0000000;//RXM0SIDH -> MASK 0
               STANDARD IDENTIFIER HIGH
56          write(RXM0SIDH_ADRESS,0);
57
58          char RXM0SIDL_ADRESS=0b00100001;//RXM0SIDL -> MASK 0
               STANDARD IDENTIFIER LOW
59          write(RXM0SIDL_ADRESS,0);
60
61          char RXB0CTRL=0b00100000;
62          char ADRESS_RXB0CTRL=0b01100000;
63          write(ADRESS_RXB0CTRL, RXB0CTRL);
64
65
66          /********** SET BUFFER1 **********/
67          char RXM1SIDH_ADRESS=0b00100100;//RXM1SIDH -> MASK 1
               STANDARD IDENTIFIER HIGH
68          write(RXM1SIDH_ADRESS,0);
```

```
69
70          char RXM1SIDL_ADRESS=0b00100101;//RXM1SIDL -> MASK 1
               STANDARD IDENTIFIER LOW
71          write(RXM1SIDL_ADRESS,0x0);
72
73          char RXB1CTRL=0b00100000;
74          char ADRESS_RXB1CTRL=0b01110000;
75          write(ADRESS_RXB1CTRL,RXB1CTRL);
76      }
77  }
78
79  //************ TRANSMISSION/RECEIVE MODE ***************/
80
81  // Reinicia MCP2515
82  void CAN::reset(){
83      cs_can.set(0);
84      spi_send(RESET);
85      while(spi_.is_busy()){}
86      cs_can.set(1);
87  }
88
89  // Instrucao de leitura dos buffers RX (nm select RX buffer ->
       RXBnm, onde m (SIDH ou D0))
90  void CAN::read_rx_buffer(bool n, bool m,char* data){
91      char READ_RX_BUFFER;
92      if(n==0){
93          if(m==0){
94              READ_RX_BUFFER=0b10010000; // Buffer 0 - RXB0SIDH
95          }
96          else{
97              READ_RX_BUFFER=0b10010010; // Buffer 0 - RXB0D0
98          }
99      }
100     else{
101         if(m==0){
102             READ_RX_BUFFER=0b10010100; // Buffer 1 - RXB1SIDH
103         }
104         else{
105             READ_RX_BUFFER=0b10010110; // Buffer 1 - RXB1D0
106         }
107     }
108     cs_can.set(0);
```

```
109     spi_send(READ_RX_BUFFER);
110     while(spi_.is_busy()){}
111     if(m==0){
112         *data=spi_receive(); // recebe apenas o identificador de
               msg Standard (8 bits mais significativos - faltam 3 bits
                que devem ser lidos de RXB1SIDL)
113     }
114     else{
115         for(int i=0;i<8;i++){
116             *data=spi_receive();
117             data++;
118         }
119     }
120     while(spi_.is_busy()){}
121     cs_can.set(1);
122 }
123
124 // Escreve no registrador especificado
125 void CAN::write(char adress, char data){
126     cs_can.set(0);
127     spi_send(WRITE);
128     spi_send(adress);
129     spi_send(data);
130     while(spi_.is_busy()){}
131     cs_can.set(1);
132 }
133
134 // Instrucao de escrita nos buffers TX (abc='110' ou '111' sao
       invalidos)
135 void CAN::load_tx_buffer(bool a, bool b, bool c,char *data){
136     char Instruction=0;
137     if(a==0 && b==0 && c==0){
138         //Envia para TXB0SIDH
139         Instruction=0b01000000;
140     }
141     else if(a==0 && b==0 && c==1){
142         //Envia para TXB0D0
143         Instruction=0b01000001;
144     }
145     cs_can.set(0);
146     spi_send(Instruction);
147     if(c==0){
```

```
148         spi_send(*data);
149     }
150     else{
151         for(int i = 0; i <8; i++)
152         {
153             spi_send(data[i]);
154         }
155     }
156     while(spi_.is_busy()){}
157     cs_can.set(1);
158 
159 }
160 
161 // Requisita o envio dos dados contidos nos buffers TX (todos
    ='111', o valor '000' é ignorado)
162 // No momento a aplicacao suporta apenas requisicao de envio do
    BUFFER0
163 void CAN::request_to_send(bool tx_0,bool tx_1,bool tx_2){
164     char RTS;
165     if (tx_0==true) {
166         RTS=0b10000001;
167     }
168     else if (tx_1==true) {
169         RTS=0b10000010;
170     }
171     else if (tx_2==true) {
172         RTS=0b10000100;
173     }
174     else{
175         RTS=0b10000001; //TX0 - default
176     }
177     cs_can.set(0);
178     spi_send(RTS);
179     while(spi_.is_busy()){}
180     cs_can.set(1);
181 }
182 
183 void CAN::read_register(char* status_receive, char _register){
184     cs_can.set(0);
185     spi_send(0x03);
186     spi_send(_register);
187     while(spi_.is_busy()){}
```

```
188     *status_receive = spi_receive();
189     cs_can.set(1);
190 }
191 
192 // Devem ser passados o endereco do registrados, a mascara dos bits
        a serem modificados e a modificacao
193 void CAN::bit_modify(char adress,char mask,char data){
194     cs_can.set(0);
195     spi_send(BIT_MODIFY);
196     spi_send(adress);
197     spi_send(mask);
198     spi_send(data);
199     while(spi_.is_busy()){}
200     cs_can.set(1);
201 }
202 
203 /*******************SPI Functions************************/
204 
205 // Envio de byte pela SPI
206 void CAN::spi_send(unsigned int data){
207     spi_.put_data(data);
208 }
209 
210 // Recebimento de byte pela SPI
211 unsigned int CAN::spi_receive(){
212     int t;
213     t=spi_.get_datamod();
214     spi_send(0x00);
215     unsigned int data = spi_.get_data();
216     while(spi_.is_busy()){}
217     return data;
218 }
219 
220 
221 
222 /****************** OTHERS FUNCTIONS*********************/
223 
224 //Habilita MCP2562
225 void CAN::enable_EN(){
226     en_.set(0);
227 }
228 
```

```
229 //Desabilita MCP2562
230 void CAN::disable_EN(){
231     en_.set(1);
232 }
233 
234 __END_SYS
```

APÊNDICE C – CÓDIGO DE APLICAÇÃO - CAN_SNIFFER_TEST.CC

```
1  // can\_sniffer\_test.cc
2
3  #include <machine.h>
4  #include <alarm.h>
5  #include <can.h>
6  #include <gpio.h>
7  #include <cpu.h>
8  #include <utility/ostream.h>
9
10 using namespace EPOS;
11 OStream cout;
12
13 void loop_can_getstatus();
14 void read_buffer(CAN *can);
15
16 int main()
17 {
18     Alarm::delay(10000);
19     loop_can_getstatus();
20     return 0;
21 }
22
23 void loop_can_getstatus()
24 {
25     int mode=1;
26     char Status=0, Interrupt=0;
27     CAN can;
28     can.config();
29     can.config_filters(mode); //Listen-Only Config Filters
30
31     can.write(0x0F,0b01100100); //set Listen-Only MODE
32
33     //clear all Interrupt flags
34     can.bit_modify(0b00101100,0b11111111,0b00000000);
35     //Enable Interrupt for RXB0
```

```
36      can.bit_modify(0b00101011, 0b11111111,0b00000001);
37      can.enable_EN(); // MCP2562
38      while(1) {
39          can.read_register(&Status,0x0F);//read register CTRL
40          can.read_register(&Status,0x0E);//read register STAT
41          can.read_register(&Interrupt, 0b00101100);
42          if((Interrupt & 0x01)== 0b00000001){
43              read_buffer(&can);
44              //cout<<hex<<(unsigned int) Interrupt<<'\n';
45              can.bit_modify(0b00101100,0b11111111,0b00000000);
46          }
47      }
48  }
49
50  void read_buffer(CAN *can){
51      char data[8];
52      char ID;
53      static int n=0;
54      static unsigned int IDS[12];
55      //unsigned int standardID=0;
56      can->read_rx_buffer(0,0,&ID); //read  RXB0SIDH
57      //standardID |= (ID << 3);
58      IDS[n]|= (ID << 3);
59      can->read_register(&ID, 0b01100010); // read RXB0SIDL
60      //standardID |= (ID >> 5);
61      IDS[n]|= (ID >> 5);
62      can->read_rx_buffer(0,1,data);
63      //cout<<hex<<IDS[n]<<'\n';
64      if(n>=11){
65          for(int i=0;i<12;i++){
66              cout<<hex<<IDS[i]<<'\n';
67          }
68          n=0;
69          for(int i=0;i<12;i++){
70              IDS[i]=0;
71          }
72          cout<<"------------------------------"<<'\n';
73      }
74      else{
75          n++;
76      }
77      /*for(int i=0;i<8;i++){
```

```
78          cout <<hex<<(unsigned int) data[i];
79          cout << " ";
80      }
81      cout<<"    ";*/
82  }
```

APÊNDICE D – CÓDIGO DE SIMULAÇÃO CAN - MASTER

```
1
2  #include <stdint.h>
3  #include <stdbool.h>
4  #include "inc/hw_memmap.h"
5  #include "inc/hw_types.h"
6  #include "inc/hw_can.h"
7  #include "inc/hw_ints.h"
8  #include "driverlib/can.h"
9  #include "driverlib/interrupt.h"
10 #include "driverlib/sysctl.h"
11 #include "driverlib/gpio.h"
12 #include "driverlib/pin_map.h"
13 #include "driverlib/debug.h"
14
15 volatile unsigned long g_ulIntCount = 0;
16 volatile unsigned long g_ulMsg1Count = 0;
17 volatile unsigned long g_ulMsg2Count = 0;
18 volatile unsigned long g_bErrFlag = 0;
19
20 tCANMsgObject g_sCANMsgObject1;
21 tCANMsgObject g_sCANMsgObject2;
22 tCANMsgObject g_sCANMsgObject3;
23 tCANMsgObject g_sCANMsgObject4;
24 tCANMsgObject g_sCANMsgObject5;
25 tCANMsgObject g_sCANMsgObject6;
26 tCANMsgObject g_sCANMsgObject7;
27 tCANMsgObject g_sCANMsgObject8;
28 tCANMsgObject g_sCANMsgObject9;
29 tCANMsgObject g_sCANMsgObject10;
30 tCANMsgObject g_sCANMsgObject11;
31 tCANMsgObject g_sCANMsgObject12;
32
33 uint8_t g_ucMsg1[8]={0xBF, 0x8B, 0xBE, 0x0B, 0x57, 0x56, 0x02,0x00
      };
34 uint8_t g_ucMsg2[8]={0xC1, 0x8B, 0xBF, 0x0B, 0x57, 0x57, 0x00,0x00
      };
35 uint8_t g_ucMsg3[8]={0xC1, 0x0B, 0xBD, 0x8B, 0x58, 0x57, 0x01,0x00
```

```
    };
36 uint8_t g_ucMsg4[8]={0xC1, 0x8B, 0xBF, 0xCB, 0x58, 0x57, 0x0C,0x00
    };
37 uint8_t g_ucMsg5[8]={0xC1, 0x8B, 0xBF, 0x4B, 0x58, 0x57, 0x01,0x00
    };
38 uint8_t g_ucMsg6[8]={0xC1, 0x0B, 0xC0, 0xCB, 0x58, 0x56, 0x0C,0x00
    };
39 uint8_t g_ucMsg7[8]={0xC2, 0xCB, 0xBE, 0x0B, 0x57, 0x54, 0x83,0x00
    };
40 uint8_t g_ucMsg8[8]={0xC1, 0x4B, 0xC0, 0x0B, 0x57, 0x56, 0x01,0x00
    };
41 uint8_t g_ucMsg9[8]={0xC1, 0x8B, 0xBD, 0x0B, 0x57, 0x56, 0x01,0x00
    };
42 uint8_t g_ucMsg10[8]={0xC1, 0x0B, 0xBF, 0x4B, 0x57, 0x56, 0x0C,0x00
    };
43 uint8_t g_ucMsg11[8]={0xC1, 0x0B, 0xBD, 0x4B, 0x56, 0x56, 0x00,0x00
    };
44 uint8_t g_ucMsg12[8]={0xC1, 0x4B, 0xBE, 0x8B, 0x56, 0x54, 0x0C,0x00
    };
45
46 void CANIntHandler(void);
47 void InicializaCAN(void);
48 void set_msg(void);
49
50 int main(void) {
51     tCANMsgObject sCANMessage;
52     //Configuracao do clock!
53
54     SysCtlClockSet(
55     SYSCTL_SYSDIV_5 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ |
          SYSCTL_OSC_MAIN); //System clock: 40MHz
56
57     InicializaCAN();
58
59     set_msg();
60
61     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
62     GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2
          | GPIO_PIN_3);
63     GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 |
          GPIO_PIN_3, 0x00);
64
```

```
65      //message vector
66      tCANMsgObject g_sCANMsgObject[12] = { g_sCANMsgObject1,
           g_sCANMsgObject2,
67              g_sCANMsgObject3, g_sCANMsgObject4, g_sCANMsgObject5,
68              g_sCANMsgObject6, g_sCANMsgObject7, g_sCANMsgObject8,
69              g_sCANMsgObject9, g_sCANMsgObject10, g_sCANMsgObject11,
70              g_sCANMsgObject12 };
71
72      sCANMessage.ui32MsgID = 0x11;// CAN msg ID
73      sCANMessage.ui32MsgIDMask = 0xfffff;// mask, all 20 bits must
           match
74      sCANMessage.ui32Flags = MSG_OBJ_RX_INT_ENABLE |
           MSG_OBJ_USE_ID_FILTER |
75                              MSG_OBJ_EXTENDED_ID;
76      sCANMessage.ui32MsgLen = 8;// allow up to 8 bytes
77      sCANMessage.ui32MsgID = 0x2001;
78
79      while (1) {
80          static int n = 0;
81          GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x08);
82          CANMessageSet(CAN0_BASE, 1 ,&g_sCANMsgObject[n],
               MSG_OBJ_TYPE_TX);
83          GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x00);
84          SysCtlDelay(SysCtlClockGet()/5000);
85          if (n >= 11) {
86              n = 0;
87          } else {
88              n++;
89          }
90      }
91  }
92
93  void InicializaCAN(void) {
94
95      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
96      GPIOPinConfigure(GPIO_PB4_CAN0RX);
97      GPIOPinConfigure(GPIO_PB5_CAN0TX);
98      GPIOPinTypeCAN(GPIO_PORTB_BASE, GPIO_PIN_4 | GPIO_PIN_5);
99      SysCtlPeripheralEnable(SYSCTL_PERIPH_CAN0);
100     while (!SysCtlPeripheralReady(SYSCTL_PERIPH_CAN0)) {
101     }
102     CANInit(CAN0_BASE);
```

```
103     CANBitRateSet(CAN0_BASE, SysCtlClockGet(), 250000);
104     CANIntRegister(CAN0_BASE, CANIntHandler);
105         // if using dynamic vectors
106     CANIntEnable(CAN0_BASE, CAN_INT_MASTER | CAN_INT_ERROR |
           CAN_INT_STATUS);
107     IntEnable(INT_CAN0);
108     CANEnable(CAN0_BASE);
109 }
110
111 void CANIntHandler(void) {
112     unsigned long ulStatus;
113     GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0x04);
114     ulStatus = CANIntStatus(CAN0_BASE, CAN_INT_STS_CAUSE);
115
116     if (ulStatus == CAN_INT_INTID_STATUS) {
117         ulStatus = CANStatusGet(CAN0_BASE, CAN_STS_CONTROL);
118         g_bErrFlag = 1;
119     }
120
121     else if (ulStatus == 1) {
122         CANIntClear(CAN0_BASE, 1);
123         g_ulMsg1Count++;
124         g_bErrFlag = 0;
125     }
126
127     else if (ulStatus == 2) {
128         CANIntClear(CAN0_BASE, 2);
129         g_ulMsg2Count++;
130         g_bErrFlag = 0;
131     }
132
133     else {
134     }
135 }
136
137 /**Set all */
138 void set_msg() {
139 //
140 // Initialize the message object that will be used for sending CAN
141 // messages.  The message will be 4 bytes that will contain an
       incrementing
142 // value.  Initially it will be set to 0.
```

```
143 //
144     g_sCANMsgObject1.ui32MsgID = 0x6D0;
145     g_sCANMsgObject1.ui32MsgIDMask = 0;
146     g_sCANMsgObject1.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
147     g_sCANMsgObject1.ui32MsgLen = sizeof(g_ucMsg1);
148     g_sCANMsgObject1.pui8MsgData = g_ucMsg1;
149 
150     g_sCANMsgObject2.ui32MsgID = 0x6D1;
151     g_sCANMsgObject2.ui32MsgIDMask = 0;
152     g_sCANMsgObject2.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
153     g_sCANMsgObject2.ui32MsgLen = sizeof(g_ucMsg2);
154     g_sCANMsgObject2.pui8MsgData = g_ucMsg2;
155 
156     g_sCANMsgObject3.ui32MsgID = 0x6D2;
157     g_sCANMsgObject3.ui32MsgIDMask = 0;
158     g_sCANMsgObject3.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
159     g_sCANMsgObject3.ui32MsgLen = sizeof(g_ucMsg3);
160     g_sCANMsgObject3.pui8MsgData = g_ucMsg3;
161 
162     g_sCANMsgObject4.ui32MsgID = 0x6D3;
163     g_sCANMsgObject4.ui32MsgIDMask = 0;
164     g_sCANMsgObject4.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
165     g_sCANMsgObject4.ui32MsgLen = sizeof(g_ucMsg4);
166     g_sCANMsgObject4.pui8MsgData = g_ucMsg4;
167 
168     g_sCANMsgObject5.ui32MsgID = 0x6D4;
169     g_sCANMsgObject5.ui32MsgIDMask = 0;
170     g_sCANMsgObject5.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
171     g_sCANMsgObject5.ui32MsgLen = sizeof(g_ucMsg5);
172     g_sCANMsgObject5.pui8MsgData = g_ucMsg5;
173 
174     g_sCANMsgObject6.ui32MsgID = 0x6D5;
175     g_sCANMsgObject6.ui32MsgIDMask = 0;
176     g_sCANMsgObject6.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
177     g_sCANMsgObject6.ui32MsgLen = sizeof(g_ucMsg6);
178     g_sCANMsgObject6.pui8MsgData = g_ucMsg6;
179 
180     g_sCANMsgObject7.ui32MsgID = 0x6D6;
181     g_sCANMsgObject7.ui32MsgIDMask = 0;
182     g_sCANMsgObject7.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
183     g_sCANMsgObject7.ui32MsgLen = sizeof(g_ucMsg7);
184     g_sCANMsgObject7.pui8MsgData = g_ucMsg7;
```

```
185
186     g_sCANMsgObject8.ui32MsgID = 0x6D7;
187     g_sCANMsgObject8.ui32MsgIDMask = 0;
188     g_sCANMsgObject8.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
189     g_sCANMsgObject8.ui32MsgLen = sizeof(g_ucMsg8);
190     g_sCANMsgObject8.pui8MsgData = g_ucMsg8;
191
192     g_sCANMsgObject9.ui32MsgID = 0x6D8;
193     g_sCANMsgObject9.ui32MsgIDMask = 0;
194     g_sCANMsgObject9.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
195     g_sCANMsgObject9.ui32MsgLen = sizeof(g_ucMsg9);
196     g_sCANMsgObject9.pui8MsgData = g_ucMsg9;
197
198     g_sCANMsgObject10.ui32MsgID = 0x6D9;
199     g_sCANMsgObject10.ui32MsgIDMask = 0;
200     g_sCANMsgObject10.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
201     g_sCANMsgObject10.ui32MsgLen = sizeof(g_ucMsg10);
202     g_sCANMsgObject10.pui8MsgData = g_ucMsg10;
203
204     g_sCANMsgObject11.ui32MsgID = 0x6DA;
205     g_sCANMsgObject11.ui32MsgIDMask = 0;
206     g_sCANMsgObject11.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
207     g_sCANMsgObject11.ui32MsgLen = sizeof(g_ucMsg11);
208     g_sCANMsgObject11.pui8MsgData = g_ucMsg11;
209
210     g_sCANMsgObject12.ui32MsgID = 0x6DB;
211     g_sCANMsgObject12.ui32MsgIDMask = 0;
212     g_sCANMsgObject12.ui32Flags = MSG_OBJ_TX_INT_ENABLE;
213     g_sCANMsgObject12.ui32MsgLen = sizeof(g_ucMsg12);
214     g_sCANMsgObject12.pui8MsgData = g_ucMsg12;
215 }
```

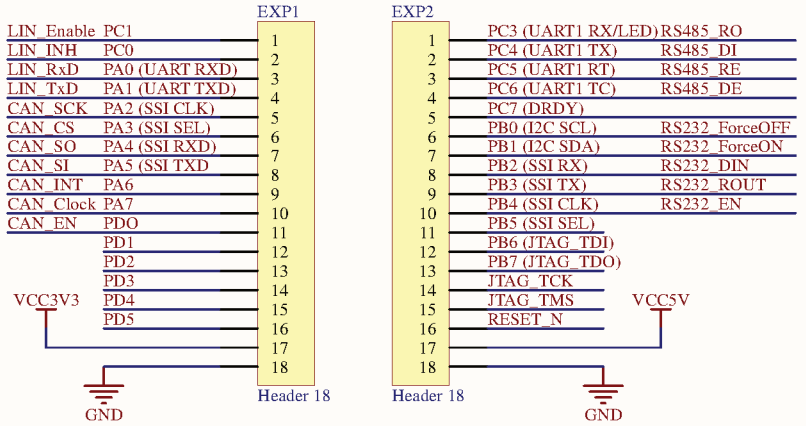
APÊNDICE E – CÓDIGO DE SIMULAÇÃO CAN - SLAVE

```
1  #include <stdint.h>
2  #include <stdbool.h>
3  #include "inc/hw_memmap.h"
4  #include "inc/hw_can.h"
5  #include "inc/hw_ints.h"
6  #include "driverlib/can.h"
7  #include "driverlib/interrupt.h"
8  #include "driverlib/sysctl.h"
9  #include "driverlib/gpio.h"
10 #include "driverlib/pin_map.h"
11
12 volatile unsigned long g_ulIntCount = 0;
13 volatile unsigned long g_ulMsg1Count = 0;
14 volatile unsigned long g_ulMsg2Count = 0;
15 volatile unsigned long g_bErrFlag = 0;
16
17 volatile bool rxFlag = 0; // msg recieved flag
18 volatile bool errFlag = 0; // error flag
19
20 void CANIntHandler(void);
21 void InicializaCAN(void);
22
23 int main(void) {
24     tCANMsgObject sCANMessage;
25     unsigned char ucMsgData[8];
26
27     SysCtlClockSet(
28     SYSCTL_SYSDIV_5 | SYSCTL_USE_PLL | SYSCTL_XTAL_16MHZ |
          SYSCTL_OSC_MAIN);
29          //System clock: 40MHz
30
31     InicializaCAN();
32
33     SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
34     GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE,
35     GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3);
36     GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1 | GPIO_PIN_2 |
```

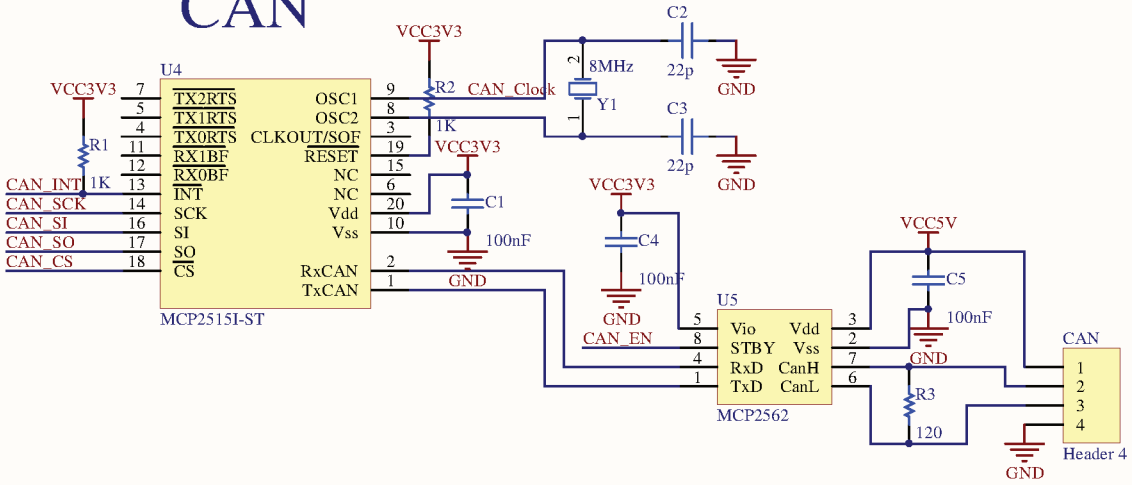
```
        GPIO_PIN_3, 0x00);
37
38      sCANMessage.ui32MsgID = 0;// CAN msg ID
39      sCANMessage.ui32MsgIDMask = 0;
40          // mask, all 20 bits must match
41      sCANMessage.ui32Flags = MSG_OBJ_RX_INT_ENABLE |
           MSG_OBJ_USE_ID_FILTER
42              | MSG_OBJ_EXTENDED_ID;
43      sCANMessage.ui32MsgLen = 8;// allow up to 8 bytes
44
45      CANMessageSet(CAN0_BASE, 1, &sCANMessage, MSG_OBJ_TYPE_RX);
46
47      while (1) {
48          if (rxFlag) { // rx interrupt has occured
49              sCANMessage.pui8MsgData = ucMsgData;
50                          // read CAN message object 1 from CAN
                               peripheral
51              CANMessageGet(CAN0_BASE, 1, &sCANMessage, 0);
52              rxFlag = 0; // clear rx flag
53              if (sCANMessage.ui32Flags & MSG_OBJ_DATA_LOST) {
54                          // check msg flags for any lost messages
55              }
56          }
57          GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1, 0x00);
58          SysCtlDelay(SysCtlClockGet() / 50000);
59      }
60  }
61
62  void InicializaCAN(void) {
63
64      SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOB);
65      GPIOPinConfigure(GPIO_PB4_CAN0RX);
66      GPIOPinConfigure(GPIO_PB5_CAN0TX);
67      GPIOPinTypeCAN(GPIO_PORTB_BASE, GPIO_PIN_4 | GPIO_PIN_5);
68      SysCtlPeripheralEnable(SYSCTL_PERIPH_CAN0);
69      while (!SysCtlPeripheralReady(SYSCTL_PERIPH_CAN0)) {
70      }
71      CANInit(CAN0_BASE);
72      CANBitRateSet(CAN0_BASE, SysCtlClockGet(), 250000);
73      CANIntRegister(CAN0_BASE, CANIntHandler);
74          // if using dynamic vectors
75      CANIntEnable(CAN0_BASE, CAN_INT_MASTER | CAN_INT_ERROR |
```

```
        CAN_INT_STATUS);
76      IntEnable(INT_CAN0);
77      CANEnable(CAN0_BASE);
78  }
79
80  void CANIntHandler(void) {
81      unsigned long status = CANIntStatus(CAN0_BASE,
            CAN_INT_STS_CAUSE); // read interrupt status
82      // controller status interrupt
83      if (status == CAN_INT_INTID_STATUS) {
84
85          status = CANStatusGet(CAN0_BASE, CAN_STS_CONTROL);
86          errFlag = 1;
87          g_bErrFlag = 1;
88      } else if (status == 1) { // msg object 1
89          CANIntClear(CAN0_BASE, 1); // clear interrupt
90          rxFlag = 1; // set rx flag
91          errFlag = 0; // clear any error flags
92          g_ulMsg1Count++;
93          g_bErrFlag = 0;
94      } else if (status == 2) {
95          CANIntClear(CAN0_BASE, 2);
96          g_ulMsg2Count++;
97          g_bErrFlag = 0;
98      }
99  }
```

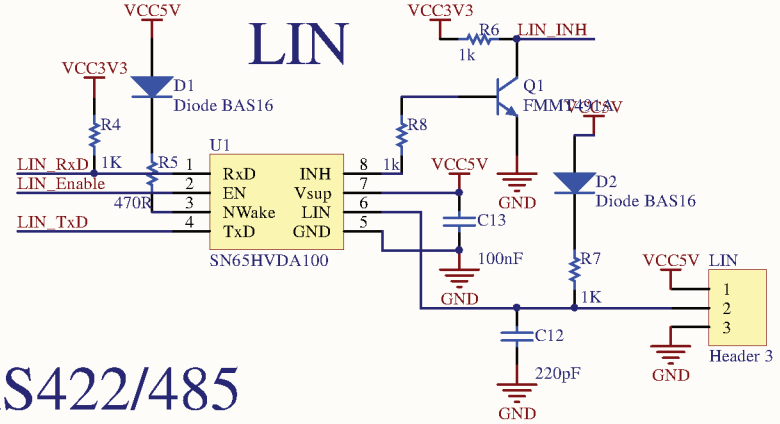
External Connectors



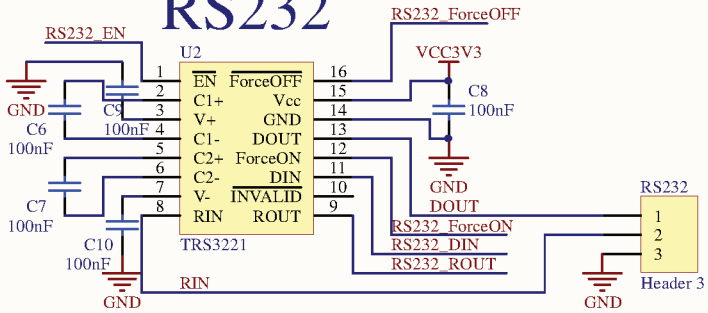
CAN



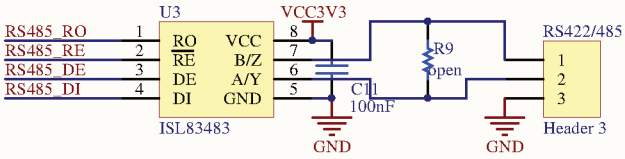
LIN



RS232



RS422/485



Title		
Size: A4	Number	Revision
Date: 07/03/2017	Sheet of	
File: C:\Users\..\SerialCom.SchDoc	Drawn By:	