UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

PATRICK NOÉ DOS SANTOS FILGUEIRA

ROBOTIZAÇÃO DE UMA CADEIRA DE RODAS

VITÓRIA
2011

PATRICK NOÉ DOS SANTOS FILGUEIRA

ROBOTIZAÇÃO DE UMA CADEIRA DE RODAS

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Engenharia Elétrica, na área de Automação.

Orientador: Prof. Dr. Teodiano Freire Bastos Filho.

VITÓRIA
2011

PATRICK NOÉ DOS SANTOS FILGUEIRA

ROBOTIZAÇÃO DE UMA CADEIRA DE RODAS

Dissertação submetida ao programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do Grau de Mestre em Engenharia Elétrica.

Aprovada em 29 de Setembro de 2011.

COMISSÃO EXAMINADORA

_____

Prof. Dr. Teodiano Freire Bastos Filho - Orientador
Universidade Federal do Espírito Santo

_____

Prof. Dr. Anselmo Frizera Neto
Universidade Federal do Espírito Santo

_____

Prof. Dr. Humberto Ferasoli Filho
Universidade Estadual Paulista - UNESP/Bauru

À Minha Esposa e Família.

AGRADECIMENTOS

Agradeço o seu valioso exemplo e amizade:

Ao meu orientador Prof. Dr. Teodiano Freire Bastos Filho...

Agradeço pelo seu apoio desde o início desta jornada, o que me proporcionou muito mais que orientação para este trabalho, mas sim, um grande exemplo de autenticidade, valores éticos e amor ao estudo, que serviram de espelho para minha própria construção intelectual. E também pela fantástica oportunidade de desenvolver este trabalho e entrar em contato com esta área de pesquisa.

Aos meus amigos...

Agradeço suas valiosas sugestões nos momentos decisivos e importantes de minha vida, bem como o constante estímulo, dedicação e atenção, oferecendo-me sempre as condições e proteções necessárias para a realização deste trabalho.

Enfim...

Agradeço a todos os que me ajudaram a percorrer a trajetória de construção deste trabalho.

E, em especial, à minha mãe Alzeidi França dos Santos, ao meu padrasto Jõao da Silva Barbosa, à minha irmã Marianna dos Santos Filgueira e a minha amada esposa Danielly Wandermurem Benício que sempre estiveram ao meu lado ao longo de todos esses anos.

RESUMO

Neste trabalho foi desenvolvido um sistema de controle de baixo nível para uma cadeira de rodas robotizada controlada por sinais biológicos, incluindo *hardware* (Placa de Acionamento dos motores) e software (firmware executado em microcontrolador).

O objetivo é dar suporte para um sistema de controle de alto nível que captura, processa e interpreta sinais biológicos a fim de controlar uma cadeira de rodas robótica.

Neste trabalho foi dado um especial foco à eletrônica, ao sistema operacional de tempo real aplicado a cadeira de rodas robótica e ao controlador de baixo nível também aplicado a esta. A eletrônica de controle da cadeira de rodas utilizou uma rede de comunicação industrial, a rede CAN, para suportar toda a aplicação e algoritmos de controle. Todas as tarefas implementadas são geridas por um sistema operacional de tempo real. O controlador de baixo nível implementado é capaz de controlar de forma independente as velocidades angular e linear da cadeira de rodas.

Para alcançar o objetivo proposto neste trabalho fez-se necessário manipular dispositivos de eletrônica de potência, programação, algoritmos de controle e instrumentação, rede CAN, sistema de tempo real, armazenamento de dados e controle automático.

- Palavras-chave: Cadeira de Rodas Robotizada, controlador de baixo nível, rede CAN, sistema de tempo real, supervisório, ponte H, PWM e sinais biológicos.

ABSTRACT

In this work, it is developed a system of low-level control for a robotic wheelchair controlled by biological signals, including hardware (motor driver board) and software (firmware running on microcontroller).

The goal is to support a high-level control system that captures, processes and interprets biological signals to control a robotic wheelchair.

In this work a special focus is given to electronics, real-time operating system applied to robotic wheelchair and also low level controller applied to their. The control electronics of the wheelchair used an industrial communication network, the CAN network to support all application and control algorithms. All tasks implemented are managed by a real-time operating system. The low-level controller implemented allows independently control of the angular and linear speeds of the wheelchair.

To achieve the objective proposed in this work, it was necessary to manipulate devices of power electronics, programming, algorithms, control and instrumentation, CAN network, real-time system, data storage and automatic control.

GLOSSÁRIO

CI - Circuito integrado

Bit - Dado digital que contém uma informação binária, sendo 0 ou 1.

Byte - Conjunto de oito bits

MSB - *Most significant bit*

LSB - *Least significant bit*

I/O - *Input/Output*

PCP - Placa de Controle dos Periféricos

PP - Placa de Processamento

RTC - *Real Time Clock*

CR - *Carrier return*

FAT - *File Allocation Table*

PWM - *Pulse-width modulation*

ASCII - *American Standard Code for Information Interchange* 2

PC - *Personal Computer*

UART - *Universal Asynchronous Receiver/Transmitter*

LED - *light-emitting diode*

IHM – Interface Homem Máquina

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

INTRODUÇÃO

Indivíduos com sérias disfunções de origem neuromotoras apresentam, invariavelmente, graves problemas de fala, comunicação e locomoção. No entanto, pessoas que apresentam estas limitações mantêm um perfeito estado de consciência, o que permite que algumas técnicas sejam desenvolvidas para provê-las de meios pelos quais possam expressar vontades [1].

Uma solução largamente estudada e pesquisada nos dias de hoje é o uso de sinais biológicos para desenvolvimento de canais alternativos de comunicação entre o indivíduo e tudo aquilo que o cerca. Dentre estes sinais biológicos, os mais comumente utilizados são: sinais mioelétricos (eletromiograma - EMG), potenciais córtico-retina (eletrooculograma - EOG) e sinais cerebrais (eletroencefalograma - EEG) [1].

Assim, é possível substituir os dispositivos tradicionais utilizados na comunicação, tais como, apontadores de cabeça, pulsadores, dentre outros, que não seriam eficazes para pessoas acometidas de graves disfunções motoras, por sistemas baseados em sinais biológicos.

Além disso, é possível, juntamente com o uso de recursos computacionais específicos, desenvolver ferramentas que sejam capazes de identificar um desejo ou uma vontade do usuário utilizando esses sinais biológicos. Para tal, entretanto, é necessário que o usuário tenha um grau mínimo de controle de suas funções cerebrais, de tal forma que possibilite a operação do sistema de forma satisfatória.

Objetivando melhor qualidade de vida para pessoas com deficiências motoras severas, o Grupo de Pesquisa do Laboratório de Automação Inteligente (LAI) da Universidade Federal do Espírito Santo (UFES) dedica esforços na pesquisa de soluções tecnológicas que usam os sinais biológicos para comandar dispositivos e equipamentos.

Uma destas soluções é o uso de sinais biológicos para controlar uma cadeira de rodas. Para tal, a cadeira rodas deve ser capaz de interpretar e executar sinais oriundos de sistemas de processamento de diversos sinais biológicos, tais como, sinais mioelétricos

(eletromiograma - EMG), potenciais córtico-retina (eletrooculograma - EOG) e sinais cerebrais (eletroencefalograma - EEG).

Este trabalho tem como objetivo o desenvolvimento de uma plataforma robótica montada sobre uma cadeira de rodas motorizada, que seja capaz de interpretar e executar comandos de alto nível oriundos de sistemas de processamento de sinais biológicos conectados a este. Para tal, foi desenvolvido um sistema de controle de baixo nível para a cadeira de rodas, sendo este composto de: instrumentação para a cadeira de rodas, periféricos de controle, sistema de acionamento dos motores da cadeira de rodas, placa de controle de tais periféricos e instrumentos, placa de processamento, IHM e um supervisório de controle e configuração (Figura 1).

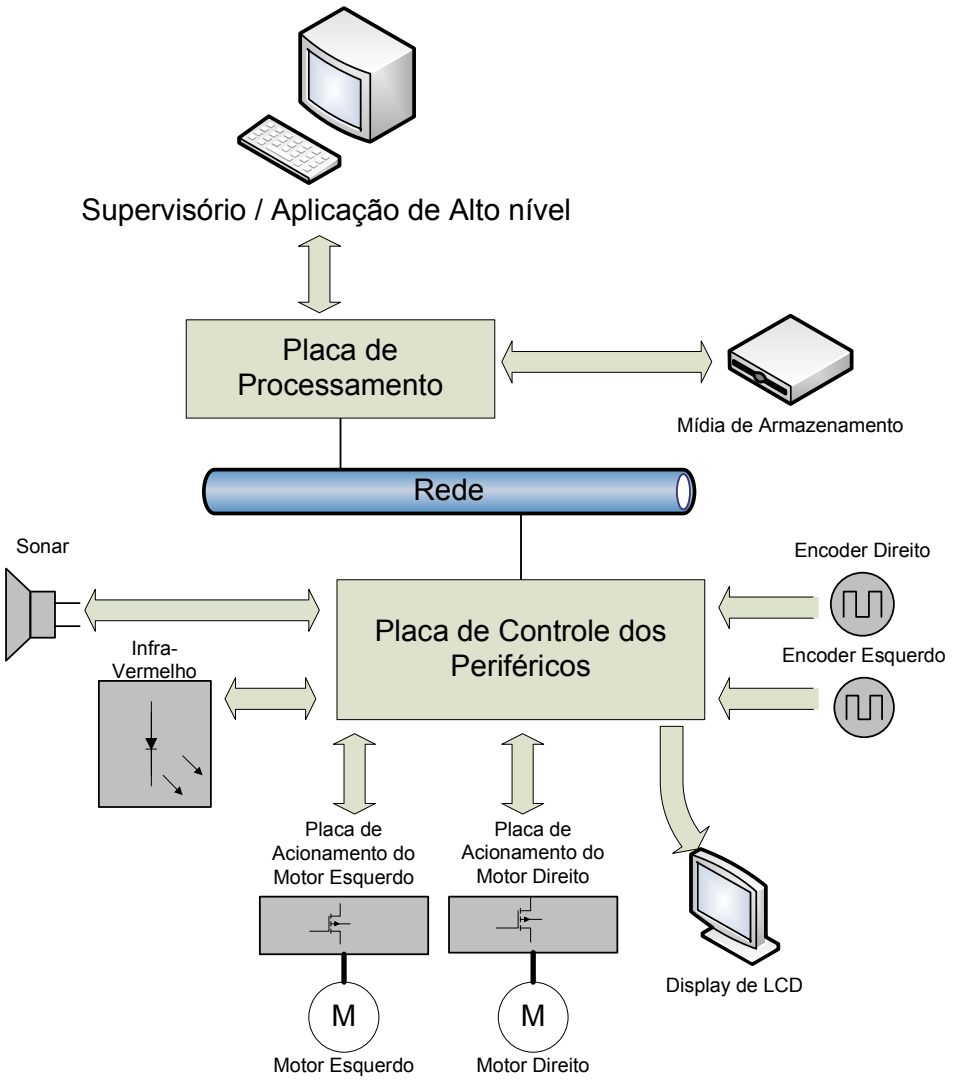


Figura 1: Esquemático conceitual da arquitetura do sistema de controle de baixo nível da cadeira de rodas.

O desenvolvimento deste trabalho iniciou-se com uma cadeira de rodas motorizada comercial, da qual foi retirado o sistema de acionamento dos motores. Assim, usou-se tão somente a estrutura física da cadeira de rodas, seus motores de acionamento e suas baterias.

A partir desta plataforma da cadeira de rodas comercial foi desenvolvido um sistema de acionamento dos motores de corrente contínua, o qual tem por objetivo principal controlar a tensão média nestes.

Este sistema de acionamento dos motores da cadeira de rodas motorizada é comandado por um sistema de processamento e controle dos periféricos da mesma, sendo que estes periféricos são *encoders* instalados no eixo de cada motor, sensores infravermelhos, e sonar. Os *encoders* têm o objetivo de medir as velocidades de cada roda para seu posterior controle. Os sensores infravermelhos e o sonar têm por objetivo o mapeamento do ambiente para navegação segura da cadeira de rodas.

O sistema de processamento implantado na cadeira de rodas robotiza se baseia na rede CAN e na filosofia de variáveis de barramento, ou seja, as informações de variáveis de controle da cadeira de rodas robotizada trafegam pela rede e podem ser processadas por qualquer nó da rede CAN. Neste trabalho também foi desenvolvida uma biblioteca de software que tem por objetivo criar rotinas de manipulação de variáveis de rede de modo automático, ou seja, independente das demais tarefas do nó de rede. Assim, as demais tarefas de um nó de rede não executam qualquer contato direto com a rede, sendo este contato e gestão das atualizações das variáveis de rede realizadas por tarefas especializadas.

Devido a esta arquitetura de variáveis de rede o processamento de controle da cadeira de rodas é dividido em duas placas, sendo possível a implantação de mais funcionalidades apenas adicionando outros nós de rede.

As placas que compõem o sistema de processamento usam um *kernel*[1] de tempo real, o µC-OS II, que é capaz de gerenciar várias tarefas em cada microcontrolador da cadeira de rodas. Neste trabalho é possível entender como foi realizado o estudo de execução das diversas tarefas de controle da cadeira de rodas, e como se pôde garantir suas execuções em tempo real.

Os algoritmos de controle dos microcontroladores da cadeira de rodas foram em sua maioria descritos na linguagem C, sendo que alguns arquivos de configuração do µC-OS II foram escritos no *assembly* nativo da família 2 do MSP430 [2]. O compilador utilizado neste trabalho foi o IAR *workbench* [3]. Todos os códigos fonte dos programas desenvolvidos neste trabalho estão disponíveis nos anexos deste documento, contudo, por questões ambientais e para facilitar sua consulta estes anexos estão disponíveis apenas em formato eletrônico.

O controle das velocidades linear e angular da cadeira de rodas é realizado por um controlador de baixo nível executado no sistema de processamento. O controlador implementado traz inovações personalizadas para seu uso na cadeira de rodas robotizada.

O projeto do sistema de processamento ainda provê uma estrutura de log periódico das variáveis de controle da cadeira de rodas, sendo realizado o armazenamento de tais variáveis em um cartão de memória micro-SD.

Por fim, foi desenvolvido um sistema supervisório para testes e depuração de erros, sendo este executado em um PC que se comunica com o sistema de processamento da cadeira de rodas por intermédio de uma interface UART.

Este trabalho aborda os aspectos de projeto das placas de acionamento dos motores da cadeira de rodas motorizada, o desenvolvimento das placas de processamento, as interfaces com os diversos periféricos desta, a arquitetura das variáveis de rede usando a rede CAN, o desenvolvimento do supervisório, o uso do µC-OS II como sistema de tempo real, e a implementação de um controlador de baixo nível. A cadeira de rodas robotizada juntamente

---

[1] Kernel*:* Algoritmo responsável por garantir que todas as tarefas de um microcontrolador sejam executadas no de modo concorrente e independente.

está ilustrada na Figura 2. A todo sistema ilustrado na Figura 1 juntamente com a estrutura da cadeira de rodas se chamou a Cadeira de Rodas Robotizada, cujo seu desenvolvimento foi objetivo deste trabalho.

Figura 2: Cadeira de Rodas Desenvolvida na UFES.

Estado da Arte

Em [4] também foi utilizada uma cadeira de rodas comercial a qual serviu de base para implementação de uma plataforma robótica. Esta plataforma foi composta por com 27 sensores ultrassônicos Polaroid [5] sendo, os motores da cadeira comercial e um sistema de controle usando o QNX[2] sendo executado em um PC industrial.

Todo o sistema de controle e navegação, desvios de obstáculos, controle de baixo nível e seu controlador está alocado no PC industrial. A arquitetura prevê ainda expansões a partir do PC industrial usando outros PCs interligados ao primeiro por uma rede *Ethernet*.

---

[2] QNX: sistema operacional de tempo real comercial direcionado para aplicações industriais

A arquitetura desenvolvida em [4] utilizou aspectos contemplados neste trabalho: sistema de tempo real, sonar, *encoders*, sistema de controle de baixo nível.

Outro aspecto relevante que este trabalho tem em comum com [4] é a preocupação com a segurança e a prevenção de falhas no sistema de tempo real. Em [4] foi utilizada a metodologia de arvore de falhas para determinar quais são os pontos mais críticos para garantir um *safety-critical system* (sistema crítico seguro). Algo similar foi realizado neste trabalho, sendo avaliados os pontos de possíveis falhas e sistemas de detecção de tais falhas, tudo para garantir que não ocorram eventos indesejados com o usuário

Figura 3: cadeira de rodas *Breman Autonomous Wheelchair* [4].

Em [6], [7] e [8] foi utilizada uma plataforma robótica comercial *ROBUTER* da companhia Francesa *ROBOSOFT* como base para uma cadeira de rodas robotizada. Algumas de suas características se assemelham a este trabalho, isso pois utiliza sonar para navegação e sistema microcontrolado para o controle de baixo nível, contudo, por se tratar de uma cadeira que utiliza um robô comercial estes trabalho focam em algoritmos de navegação e controle de alto nível, deferentemente deste trabalho.

De todos os trabalhos pesquisados o que mais se assemelha a este foi o desenvolvido em [9]. Esta semelhança começa na abordagem do problema, ambos, este trabalho e [9], apresentam uma arquitetura própria desde o acionamento dos motores da cadeira de rodas até o controle de baixo nível. Em [9] assim como neste trabalho foram utilizados protocolos de

rede seriais industriais para permitir de maneira confiável a troca de comandos entre as diversas placas e algoritmos do sistema (Figura 4). Contudo, neste trabalho foi utilizado o protocolo de comunicação CAN [10] e em [9] foi utilizado o protocolo *LonWorks* [11]. As demais funcionalidades tais como: sonar, infravermelho, acionamento dos motores, sensoriamento baseado em *encoders*, sistema de tempo real e controle de baixo nível são semelhantes entre os dois projetos. O uso de uma técnica de programação que permite que tarefas se comuniquem de forma transparente a rede de comunicação, esta aqui chamada de variáveis de barramento, é a principal diferença entre estes trabalhos, visto que este trabalho utilizou estas variáveis de barramento como base de todo sistema de controle de baixo nível, e em [9] não foi mencionado qualquer aplicação de técnica similar.

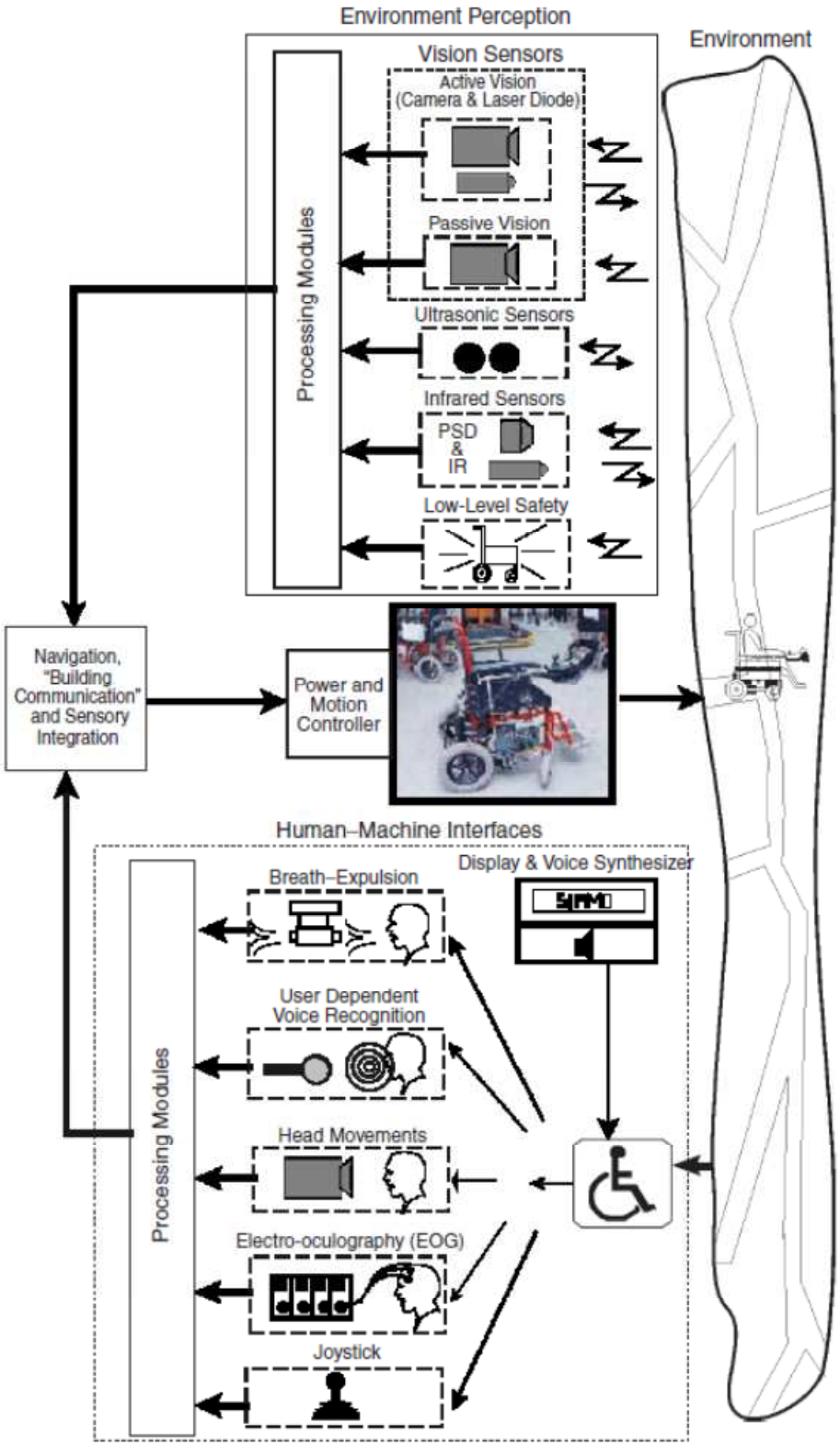


Figura 4: Arquitetura básica do projeto SIAMO [9].

Em [12] foi desenvolvida uma cadeira de rodas robótica para aplicações de navegação autônoma, sendo utilizada uma plataforma robótica de navegação baseada em

sonar, sensores *laser* e chaves de impacto na frente da cadeira de rodas. Este trabalho possui interfaces já implementadas para utilização de sensores infravermelhos e sonar, além disso, pela própria arquitetura em rede é possível adicionar qualquer outro *hardware* sem muitas modificações na plataforma original. A arquitetura utilizada no desenvolvimento de [13] é similar à [12], na qual novamente se utilizou chaves de contato como sensores de colisão na frente da cadeira de rodas e sonar, além de sensores infravermelhos. Contudo, não foi relatado o uso de qualquer protocolo de rede ou uso de sistema de tempo real, itens presentes neste trabalho.

Em pesquisas na literatura técnica internacional, pôde-se constatar grande acervo de aplicações de alto nível, tais como navegação autônoma, mapeamento de ambiente, processamento de imagens e contornos de obstáculos usando cadeira de rodas robóticas, contudo sem grande ênfase na eletrônica e controle de baixo nível de cadeiras de rodas robóticas. Assim, existem poucos trabalhos com foco similar ou igual a este, no qual são enfatizados a eletrônica e o controle de baixo nível de uma cadeira de rodas robótica.

1 O SISTEMA DE ACIONAMENTO DOS MOTORES

Os movimentos da Cadeira de Rodas Robotizada são providos por dois motores de corrente contínua acoplados as rodas por intermédio de correias e polias. Estes motores devem ter sua velocidade e torque controlados para garantir o conforto e a segurança do usuário da Cadeira de Rodas Robotizada. Objetivando controlar as velocidades de tais motores, foi implementado um sistema de acionamento individual para cada motor. Este sistema de acionamento, destacado na Figura 5, provê o controle e monitoramento do estado dos motores da Cadeira de Rodas Robotizada. Este Capítulo aborda o projeto e a construção deste sistema de extrema relevância para a Cadeira de Rodas Robotizada.

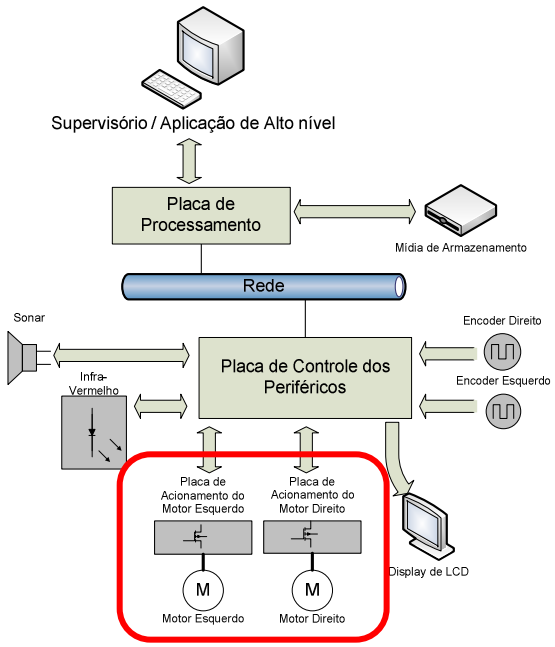


Figura 5: Esquemático básico da arquitetura da Cadeira de Rodas Robotizada com o sistema de acionamento dos motores em destaque.

1.1 O Projeto Básico do Sistema de Acionamento dos Motores da Cadeira de Rodas Robotizada

O sistema de acionamento dos motores da cadeira de rodas é composto por dois motores de corrente contínua de imãs permanentes, sendo um para cada roda, com as seguintes características:

- Potência nominal de 360 W;

- Tensão nominal de 24 V;
- Corrente nominal de 15 A;
- 900 rpm;

Existe a necessidade de controle fino na velocidade da cadeira de rodas, assim o acionamento deve suportar tal demanda fazendo a tensão média nos motores da cadeira variar suave e continuamente.

A reversibilidade é outra necessidade do conjunto motor-roda, visto que a cadeira deve ser capaz de se mover para frente e para trás com exatidão.

A alimentação é outro fator restritivo, visto que foram usadas baterias de chumbo-ácido que acompanham a cadeira comercial. Assim, é disponibilizado apenas um conjunto de baterias de 12 V dispostas em série de modo a prover 24 V.

Diante de tais premissas o modelo de engenharia básica para o acionamento é composto de uma ponte H [14] acionada e controlada por um *gate driver*[3] e isoladas por opto-acopladores das placas de controle. Os sinais de interface entre o acionamento e a placa de controle é tão somente o sinal PWM[4] e outros sinais de controle necessários para a comunicação acionamento e placa de controle (Figura 6).

Por questões de economia de *hardware* e software foi adotado como premissa o uso de apenas um sinal PWM para cada motor. Assim, o ciclo de trabalho é simétrico e o tempo morto é criado via *hardware*.

---

[3] *Gate Driver*: dispositivo capaz de acionar transistores de potencia fazendo interface entre o sistema de potência e microcontroladores.

[4] PWM (*Pulse-Width Modulation*): sinal quadrado com frequência fixa e largura entre de pulsos variável.

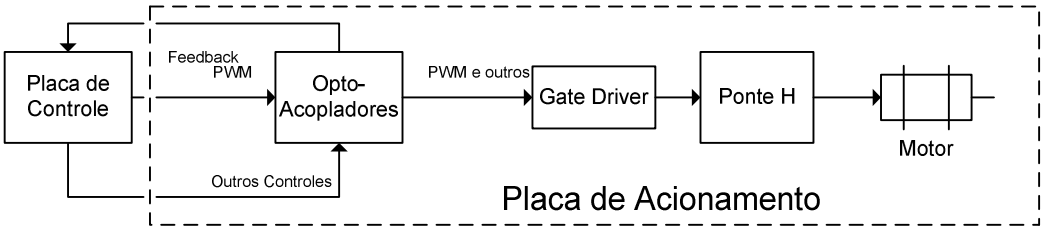


Figura 6: Topologia básica para o acionamento dos motores da cadeira de rodas.

O funcionamento da Placa de Acionamento segue as seguintes etapas:

- O sinal PWM, juntamente com outros sinais de controle, é enviado para a Placa de Acionamento via conector;
- Todos os sinais entre a placa de controle e a Placa de Acionamento são interfaceados por opto-acopladores para minimizar a possibilidade de injeção de ruídos no circuito de controle.
- O acionamento é feito por uma ponte H para permitir o controle da tensão no motor juntamente com a possibilidade de reversão do mesmo.
- O *gate driver* provê o tempo morto no chaveamento entre os transistores do mesmo lado da ponte a fim de eliminar a possibilidade de curto no chaveamento.

Assim, segue o dimensionamento dos itens da Placa de Acionamento.

- Sinal PWM;
- Ponte H;

1.2 O sinal PWM

O sinal PWM tem por característica a modulação por largura de pulso; é um sinal de onda quadrada e possui frequência definida.

Tipicamente o sinal PWM usado para controle o de motores varia de 10 kHz até 100 kHz. Para este projeto optou-se por usar uma frequência de 20 kHz para o sinal PWM (Figura 7). Este valor é justificado pelo fato de estar no limiar da faixa de frequência audível humana

[15] e pelo fato que o dimensionamento do *gate driver* [16] para frequências baixas, em torno de 20 kHz, atende também para frequências mais elevadas, ou seja, o dimensionamento dos componentes de *bootstrap* do *gate driver* para 20 kHz atende também valores mais elevados de PWM, ao passo que, o contrário não se sustenta.

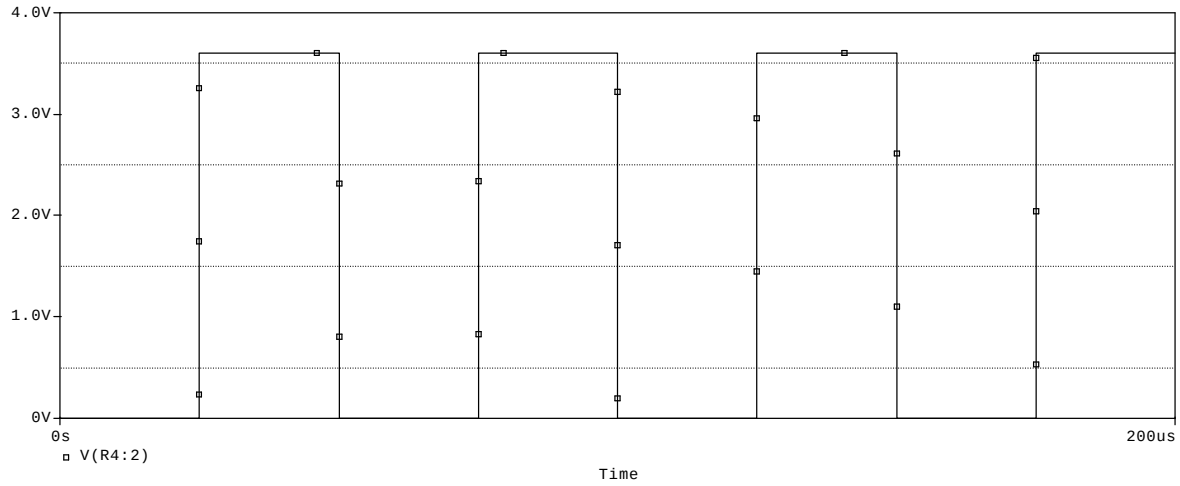


Figura 7: Simulação de sinal PWM de 20 kHz com 50 % de ciclo de trabalho.

1.3 O Dimensionamento da Ponte H

Foi idealizada uma ponte H utilizando transistores MOSFET, por sua baixa resistência (r_{dson}) série e alta velocidade de resposta, visto que a tensão de alimentação é 24 V. Para tensões mais elevadas é recomendado o uso de IGBTs por sua baixa impedância série em dispositivos acima de 100 V. A Figura 8 mostra o esquema da ponte H utilizando transistores MOSFET [17].

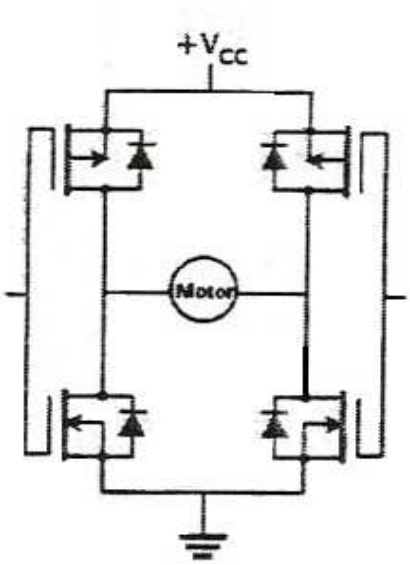


Figura 8:Ponte H composta de transistores dos tipos MOSFET Canal P e N.

Um grande problema da configuração de ponte H mostrada na Figura 8 é o fato dos transistores MOSFET canal P, na parte superior da ponte, ter uma resistência de condução elevada em relação aos transistores MOSFET canal N.

A solução encontrada foi utilizar somente transistores MOSFET canal N, como mostrado na Figura 9.

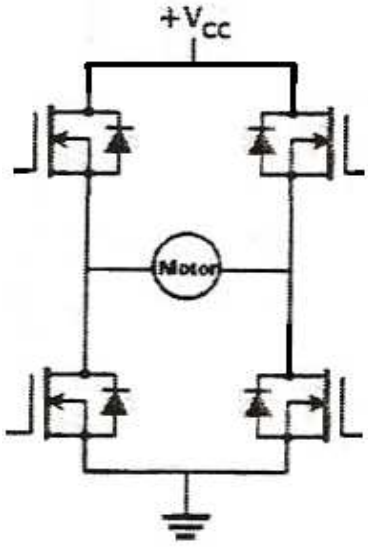


Figura 9:Ponte H composta apenas de transistores do tipo MOSFET Canal N.

Contudo, para usar a configuração da Figura 9 é necessário utilizar um dispositivo que seja capaz de acionar os transistores da parte superior da ponte.

Isso, devido ao fato dos transistores MOSFET canal N necessitarem de uma tensão positiva entre o seu *gate* e fonte. Contudo, em tal configuração, isso não aconteceria com os transistores da parte superior da ponte [17].

Para solucionar esse problema foi idealizado um sistema de acionamento dos transistores da parte superior baseado nos circuitos integrados IR2114SS e no IR2114SS, ambos produzidos pela *International Rectifier* "IRF" [16]. Tais circuitos integrados são capazes de gerar o chamado "terra virtual" na fonte dos transistores da parte superior da ponte. Com isso, os transistores são submetidos a uma tensão positiva entre o *gate* e a fonte, o que leva à sua plena condução.

1.3.1 Os Transistores

Os motores possuem corrente nominais de 15 A, assim, considerando o fato dos motores poderem trabalhar em sobrecarga de até 30 % de sua potência nominal, os transistores escolhidos foram os IRF3205, que são MOSFET canal N, suportam corrente de regime de até 110 A e tem isolamento para até 55 V [18].

Outro dado relevante do IRF3205 é sua curva de corrente de dreno (I_D) por queda de tensão de dreno para fonte (V_{DS}) mostrada na Figura 9, que indica a resistência do canal do transistor.

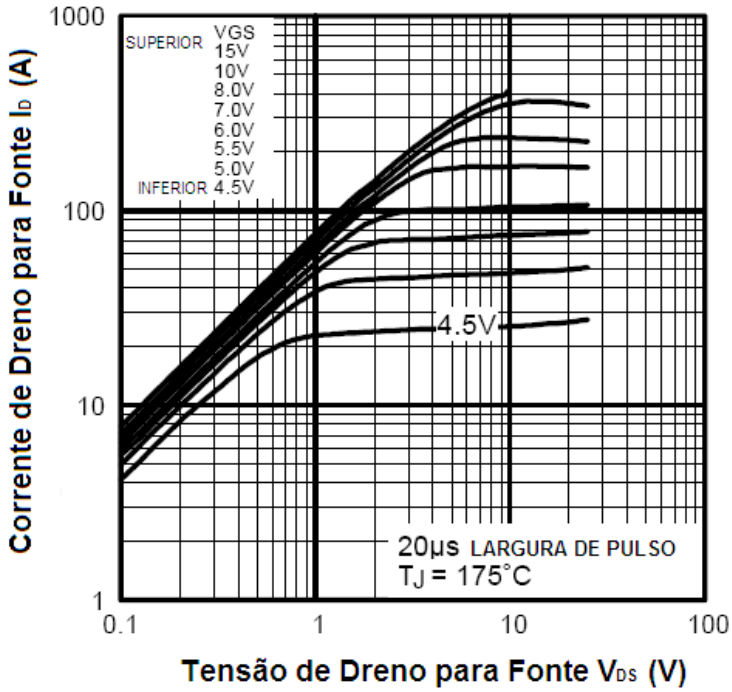


Figura 9 – Comportamento resistivo do IRF3205.

De acordo com a Figura 9, a curva de menor resistência de canal é a correspondente a V_{GS} igual a 15 V, isso sabendo que a tensão máxima suportada pelo IRF3205 entre *gate* e fonte é de 20 V. Portanto, o *gate driver* que aciona o IRF3205 provê 15 V de tensão de chaveamento para este.

1.3.2 Geração do Tempo Morto[5] do PWM

Outro fato relevante para a escolha do IR2114SS é a geração de banda morta implementada em *hardware*. Este dispositivo possui entre suas funcionalidades a geração de uma banda morta de aproximadamente 330 ns entre o tempo chaveamento dos transistores por ele controlados.

---

[5] Tempo morto ou banda morta: Tempo no qual os dois transistores do mesmo lado da ponte permanece fora de condução (off) antes de um deles entrar em condução

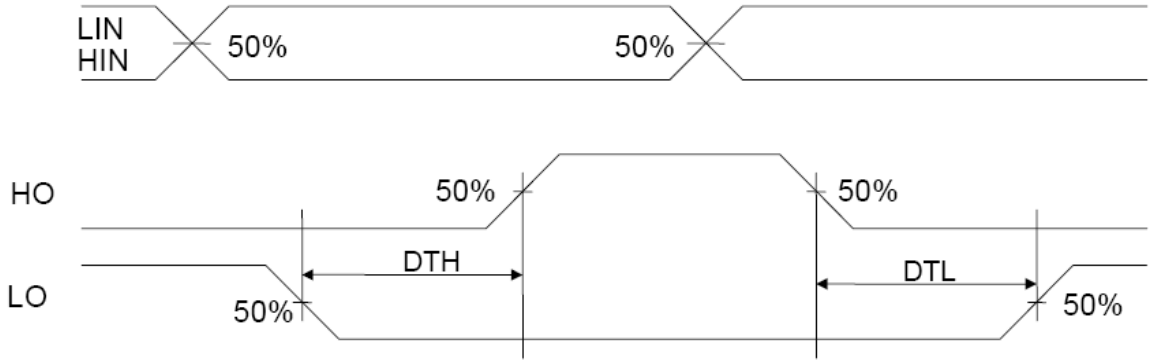


Figura 10: Comportamento típico do IR2114SS entre as entradas HIN e LIN e suas saídas HO e LO, onde DTH e DTL são as bandas mortas e têm valor tipico 330 ns.

Assim, observa-se na Figura 10 que mesmo com as entradas mudando de estado no mesmo instante de tempo existe um tempo fixo, determinado pelo próprio IR2114SS, de 330 ns, no qual as saídas permanecem em estado 0.

Esta funcionalidade do *gate driver* foi comprovada nos ensaios e testes com a ponte H. A Figura 11 ilustra o teste realizado com a ponte H implementada na qual fica evidente a geração da banda morta.

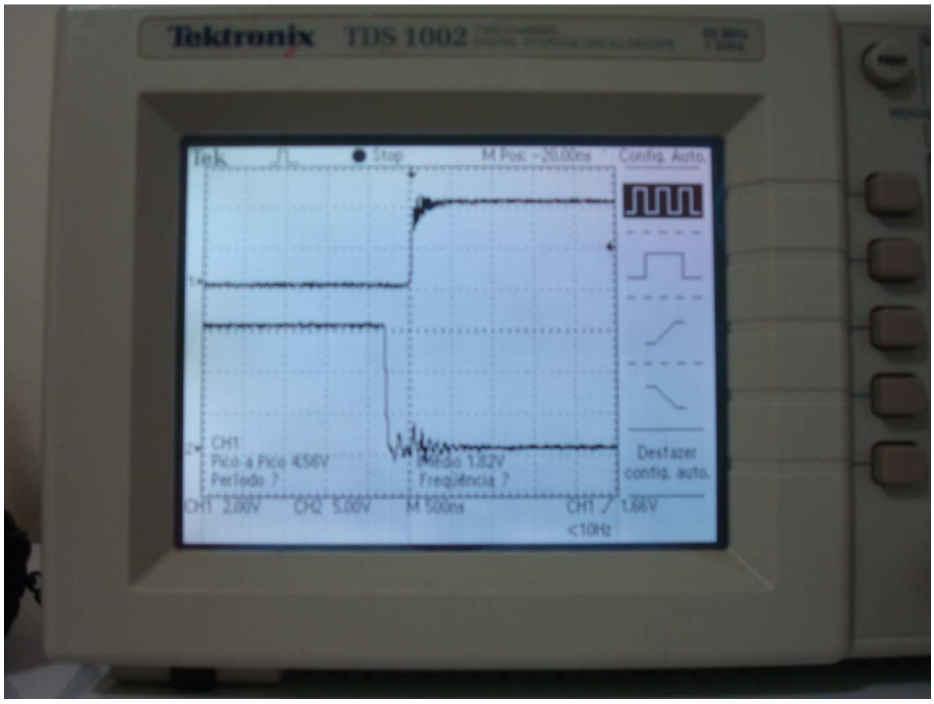


Figura 11: Fotografia de um osciloscópio registrando a banda morta da ponte implementada no momento do chaveamento das saídas HO e LO do IR2114SS.

A *International Rectifier* “IRF” sugere uma topologia de meia ponte H, mostrada na Figura 12, mas que não atende as especificações do projeto. Isso porque necessita de uma fonte simétrica de 24 V e -24 V e, como dito anteriormente, o projeto tem como prerrogativa utilizar as baterias da cadeira de rodas, que geram uma tensão de 24 V.

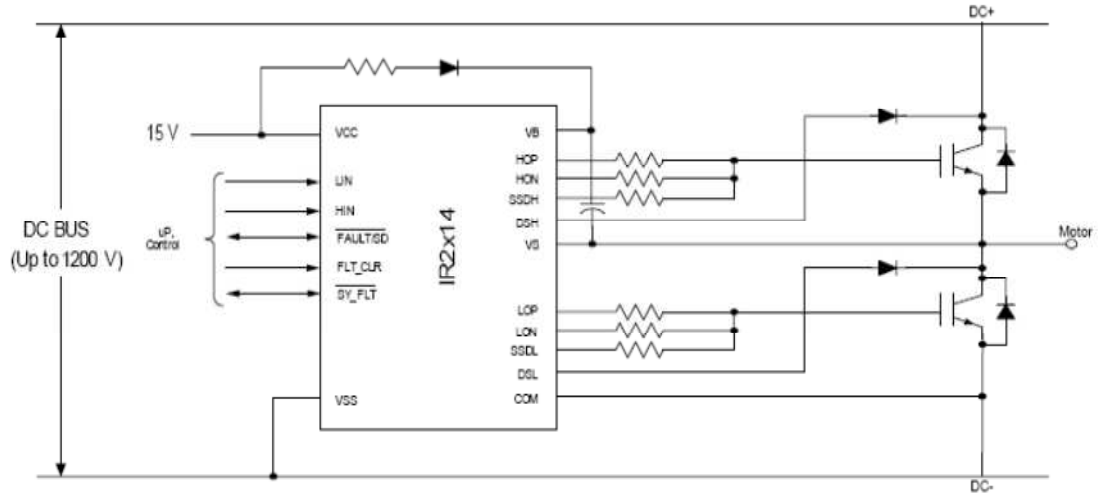


Figura 12: Meia Ponte H sugeridas pela IRF.

Assim, para atender à premissa do uso de apenas uma fonte simples de 24 V foi criado o layout exposto na Figura 13.

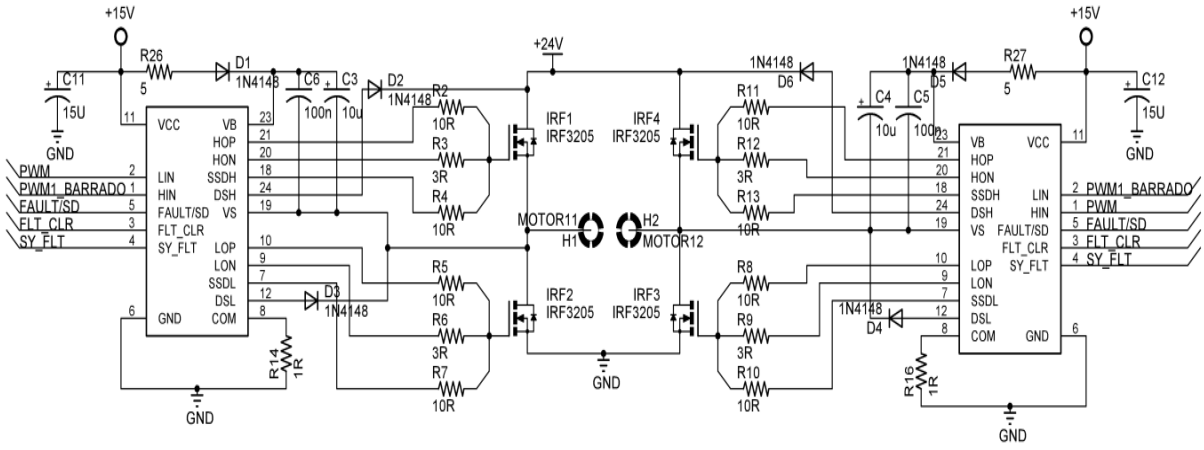


Figura 13: Circuito da ponte H implementada no projeto.

1.3.3 O *Gate Driver* IR2114SS / IR2214SS

O *gate driver* utilizado no projeto foi o IR2114SS. O IR2114 / IR2214 são adequados para conduzir uma única meia ponte em aplicações de comutação de energia. Estes *drivers* fornecem para a porta superior capacidade de 2 A e necessitam de baixa corrente quiescente, o que permite o uso de técnicas *bootstrap* para fornecimento de energia em sistemas de média potência. Estes possuem ainda proteção de curto circuito completo por meio de detecção de dessaturação e gerencia todas as falhas da meia-ponte por meio de desligamento suave. Outras características que fizeram do IR2114SS o *gate driver* escolhido para a aplicação deste trabalho são:

- Geração de banda morta de 330 ns via *hardware;*
- Desligamento suave em condição de surtos;
- Possibilidade de sincronismo de desligamento suave dos dois braços da ponte;
- Tensão máxima de trabalho de 600 V para IR2114SS e 1200 V para o IR2214SS;

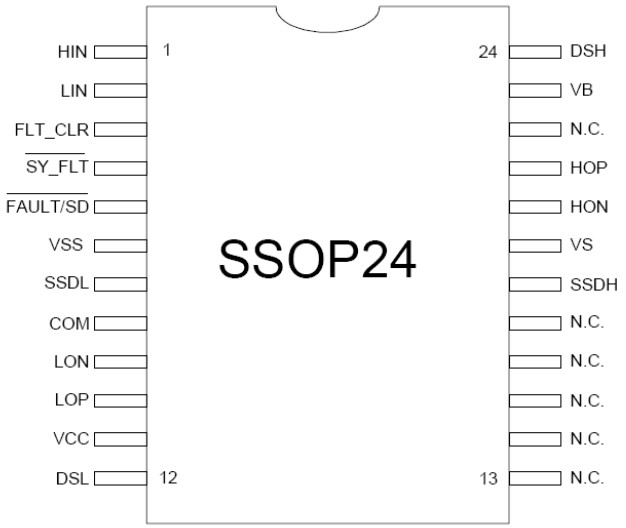


Figura 14: *Layout* e pinagem do IR2114SS / IR2214SS.

O IR2114SS (Figura 14) [16] possui alguns pinos específicos para a o acionamento e controle da meia ponte. Estes são:

- V_{cc} Alimentação do *driver*;
- V_{ss} Terra do *driver*;
- HIN Entrada lógica de controle do transistor superior da meia ponte;
- LIN Entrada lógica de controle do transistor inferior da meia ponte;
- $\overline{FAULT/SD}$ Dupla função (in / out) ativo baixo. Como uma saída, indica condição de falha. Como uma entrada, desliga as saídas do controlador de porta independentemente HIN / LIN.
- $\overline{SY\_FLT}$ Dupla função (*in* / *out*) ativo baixo. Como uma saída, indica que a seqüência SSD[6] está ocorrendo. Como uma entrada, um sinal ativo baixo congela tanto o *status* de saída;
- FLT_CLR Limpa falha, ativo alto;
- LOP Descarga do *gate* do transistor inferior da meia ponte;
- LON Carga do *gate* do transistor inferior da meia ponte;
- DSL Entrada de detecção de desaturação do transistor inferior da meia ponte;
- $SSDL$ Descarga do desligamento suave do *gate* do transistor inferior da meia ponte;
- COM Comum do lado de acionamento dos transistores;
- V_B Alimentação flutuante para acionamento do *gate* do transistor superior da meia ponte;
- HOP Descarga do *gate* do transistor superior da meia ponte;
- HON Carga do *gate* do transistor superior da meia ponte;
- DSH Entrada de detecção de desaturação do transistor superior da meia ponte;
- $SSDH$ Descarga do desligamento suave do *gate* do transistor superior da meia ponte;
- V_S Terra flutuante para acionamento do *gate* do transistor superior da meia ponte;

---

[6] SSD: *Soft Shutdown*, desligamento suave em condição de falha.

1.3.4 Dimensionamento dos Componentes de *Bootstrap*

Bootstrap é a técnica que permite proporcionar uma diferença de potencial de V_{GS} entre o *gate* e a fonte de um transistor mesmo que a fonte deste transistor esteja em um potencial mais elevado que a alimentação do *gate driver*.

Esta técnica consiste em carregar um capacitor e colocá-lo, após sua carga, entre o *gate* e a fonte do transistor acionado. Isso é possível fazendo o chaveamento deste capacitor de modo a carregá-lo e conectá-lo ao *gate* do transistor de modo repetitivo.

Na Figura 12 está ilustrado o circuito sugerido pela IRF para implementação da meia ponte. O projeto da meia ponte inclui o dimensionamento de alguns componentes presentes nesta Figura. São estes (Figura 15):

- Capacitor de *bootstrap;*
- Diodo de carga do capacitor de *bootstrap;*
- Resistor de *bootstrap;*

A Figura 15 representa a disposição física dos componentes de *Bootstrap.*

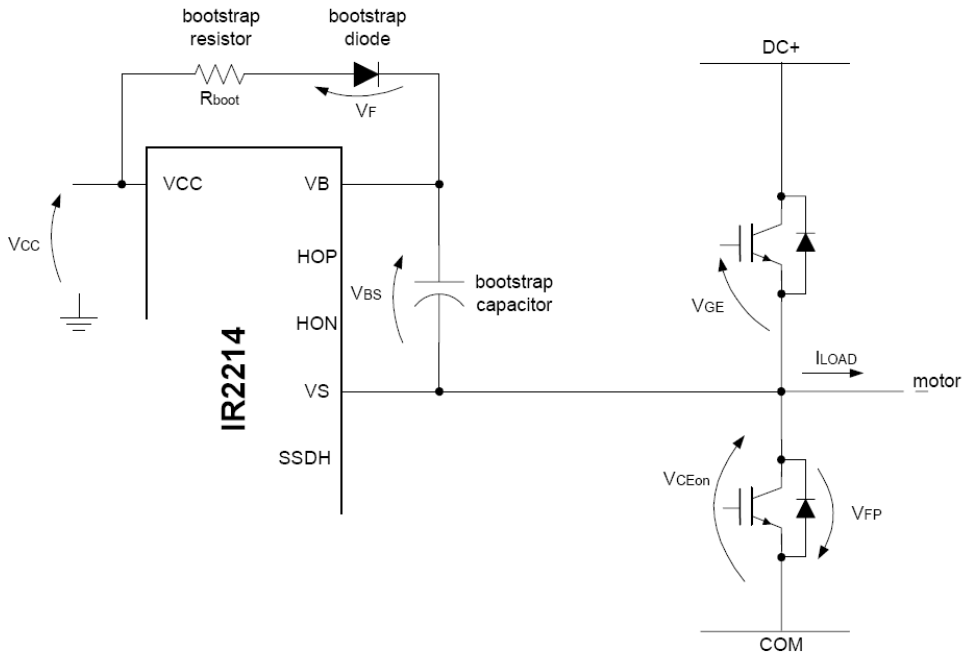


Figura 15: Componentes de *bootstrap* da meia ponte.

O diodo de *bootstrap* deve ter a tensão máxima de ruptura maior que a tensão de alimentação da ponte. Visto que a tensão da ponte é de 24 V, a tensão de ruptura do diodo deve ser maior que esta. A corrente máxima do diodo dever ser maior que 100 mA e, o tempo de recuperação reversa do diodo deve ser menor que 100 ns.

Visando atender estes parâmetros de projeto, o diodo escolhido é o 1N4148 que tem tempo de recuperação reverso de 4 ns, tensão máxima reversa de 100 V e corrente máxima de 200 mA [19].

O resistor de *bootstrap*, por recomendação do fabricante do *gate driver* não deve exceder 10 Ω. Portanto o valor dimensionado é 5 Ω.

De acordo com as relações estabelecidas para o capacitor de *bootstrap* em [16], utilizou-se a topologia de dois capacitores em paralelo, sendo um eletrolítico e outro cerâmico devido a alta indutância série do capacitor eletrolítico. Os valores dos mesmos são respectivamente 10 µF e 100 nF, eletrolítico e cerâmico .

Assim, a meia ponte projetada contemplando todos os componentes necessários para o seu funcionamento está ilustrada na Figura 16 e a ponte completa está representada na Figura 13.

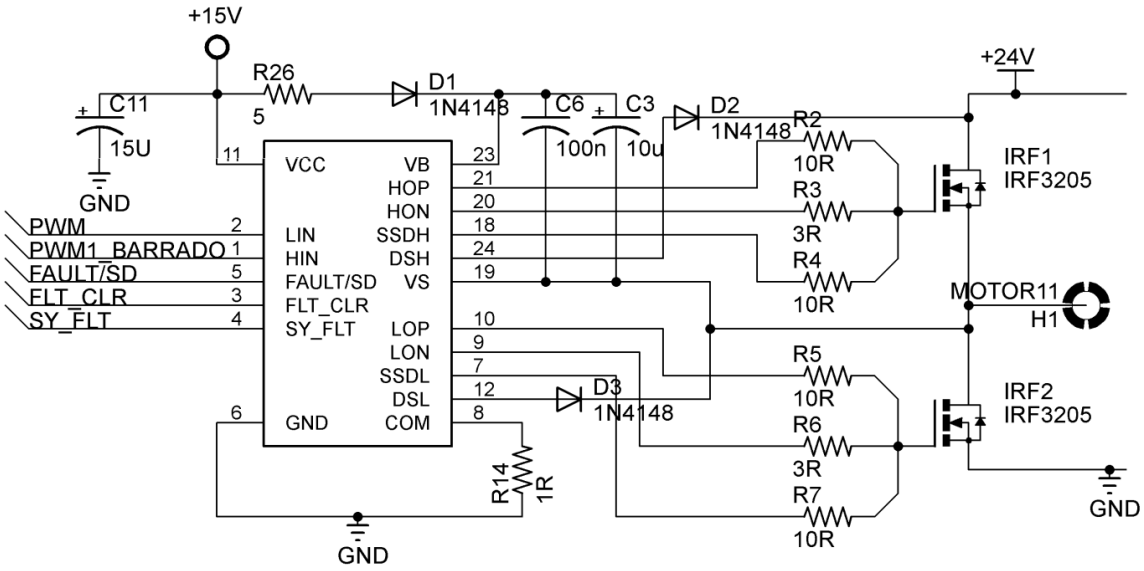


Figura 16: Projeto da meia ponte implementada no trabalho.

1.3.5 O Sistema de Opto-Acoplagem

Os motores CC, aliados ao fato de existir um chaveamento de 20 kHz de corrente da ordem de 30 A, geram ruídos de alta amplitude. Isso faz com que o sistema microcontrolado ligado à mesma bateria dos motores não funcione [17]. A solução implementada foi a separação elétrica entre os circuitos microcontrolados e os circuitos de potência. Faz-se necessário então o uso de acopladores ópticos para comunicação entre o microcontrolador e a ponte H.

Logo, para fazer com que o sinal PWM gerado pelo microcontrolador chegue à ponte H é necessário passá-lo por um sistema de opto-acopladores para então chegar às entradas dos *gate drivers*, ou seja, as entradas dos circuitos integrados IR2114SS que comandam a ponte H.

O acoplador óptico utilizado é o HCPL2530, cujo circuito montado está mostrado na Figura 17. O motivo da escolha deste opto-acoplador é sua boa resposta em frequência e tempos de subida e descida dos sinais de saída baixos e equalizados entre si. Os valores dos resistores usados seguem referencia [20].

O uso de uma porta lógica NOT na saída do circuito de acoplagem óptica é devido ao fato do *gate driver* necessitar do PWM e do mesmo PWM invertido para gerar comutação dos dois transistores de cada meia ponte.

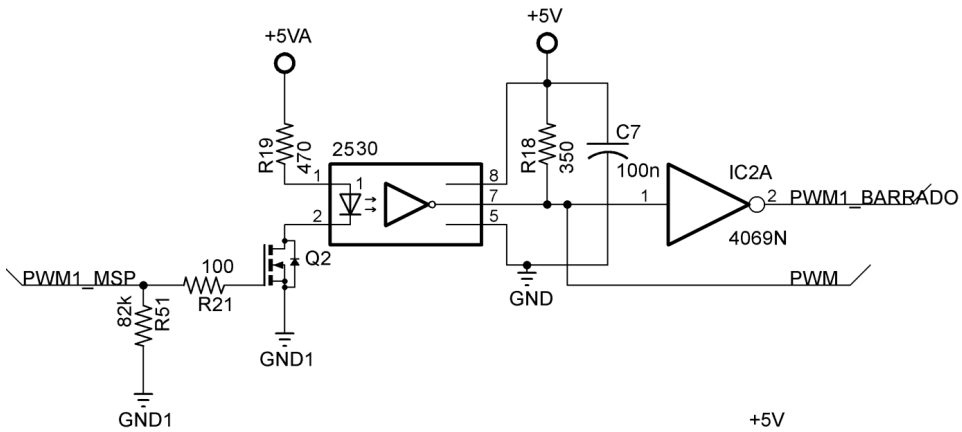


Figura 17: Circuito de acoplagem óptica implementado.

1.4 O Circuito Completo da Placa de Acionamento

Para melhor sinalização foram disponibilizados LEDs e conectores coloridos para facilitar as conexões e diagnóstico. Assim, as conexões e sinalizações são:

- Conector preto: pólo negativo da bateria (Terra);
- Conector vermelho: pólo positivo da bateria;
- Conector amarelo e azul: ligações dos motores;
- LED amarelo: alimentação de 15 V da Placa de Acionamento ligada;
- LED verde: alimentação de 5 V da Placa de Acionamento ligada;

O esquemático do circuito final de acionamento dos motores de CC da cadeira de rodas está ilustrado na Figura 18. Na Figura 19 está ilustrada a fotografia da Placa de Acionamento e seus subsistemas.

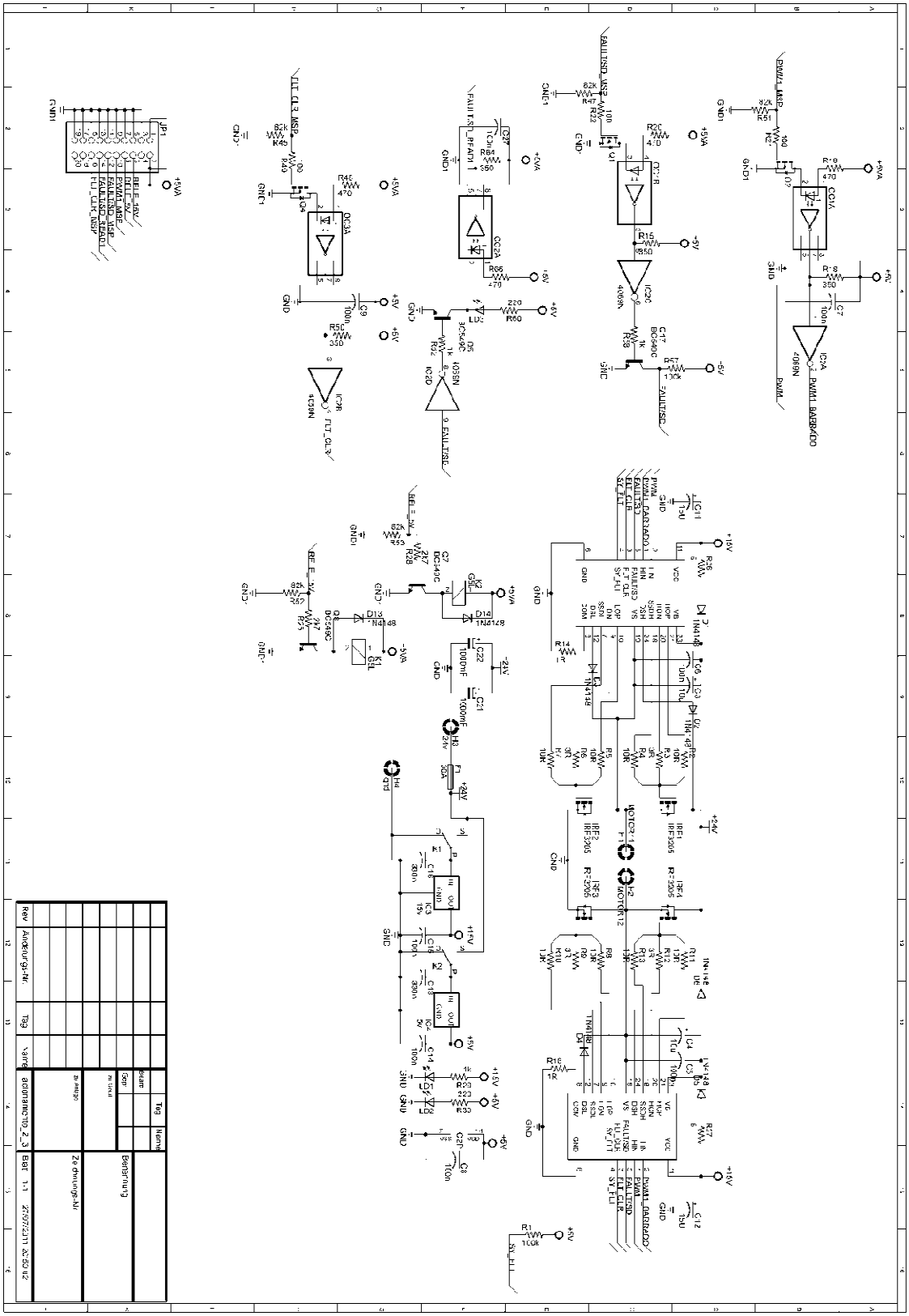


Figura 18: Circuito completo da Placa de Acionamento dos motores da cadeira de rodas.

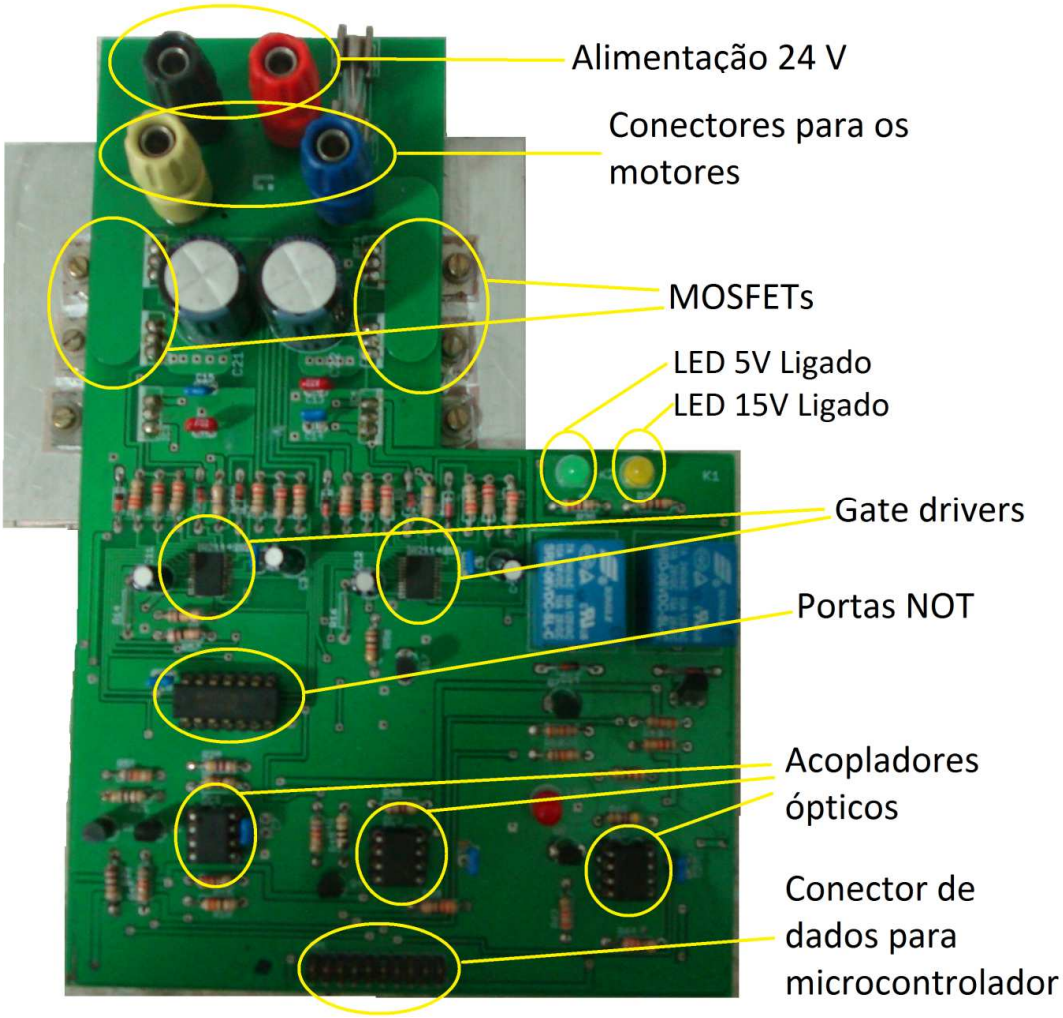


Figura 19: Fotografia da Placa de Acionamento da cadeira de rodas com os subsistemas destacados.

A Placa de Acionamento tem como função básica a geração de PWM de alta potência para os motores. Assim, o PWM gerado pela placa está ilustrado na Figura 20.

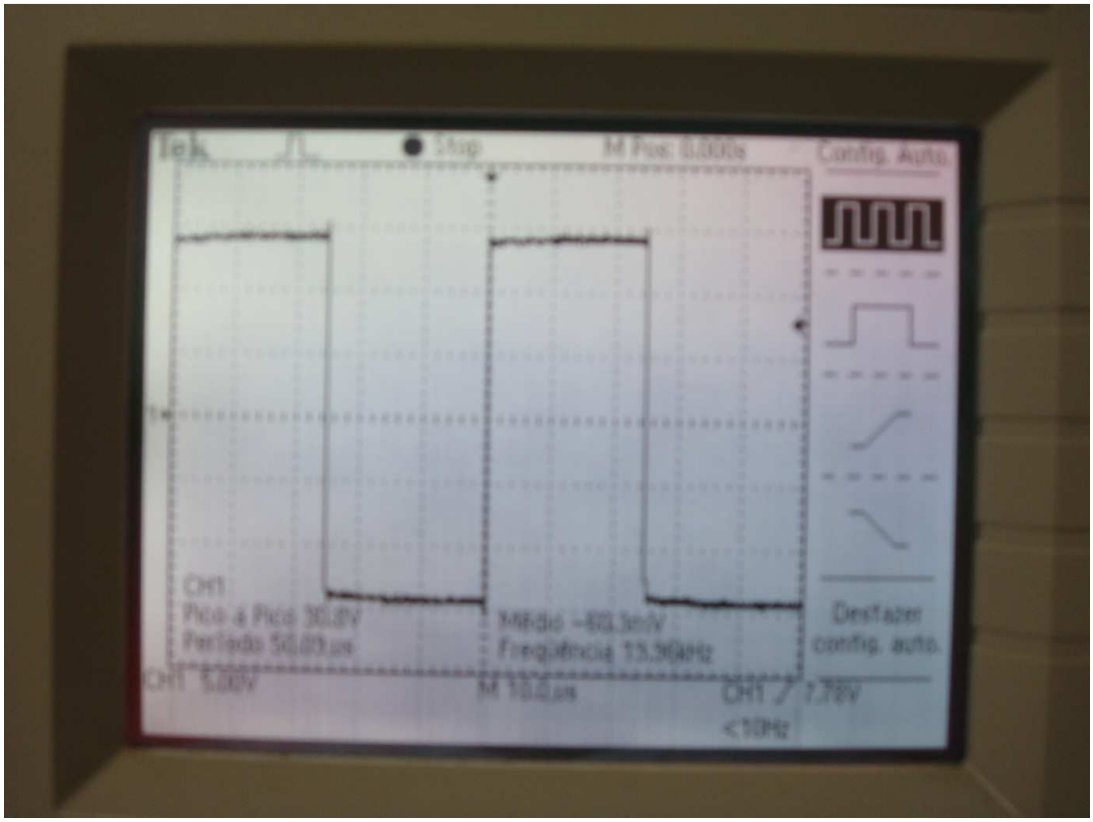


Figura 20: Fotografia do PWM de saída da Placa de Acionamento.

1.4.1 O Conector de Dados da Placa de Acionamento

Para a comunicação entre a Placa de Acionamento e a placa de controle foi disponibilizado um conector de 14 pinos (Figura 21). A Placa de Acionamento necessita de alimentação externa de 5 V para os opto-acopladores e todos os sinais desta são níveis TTL[7], ou seja, 0 → 0 V e 1 → 5V. A função de cada pino está descrita na Tabela 1.

[7] TTL: *Transistor–transistor logic*. Tecnologia de integração de circuitos.

Tabela 1: Funções dos pinos do conector de interface da Placa de Acionamento.

Pino	Função
1	Entrada de alimentação de 5 V.
2	Entrada de alimentação de 5 V.
3	GND.
4	Entrada de alimentação de 5 V.
5	GND.
6	Entrada de acionamento do Relé de 15 V. 1 → liga Relé; 0 → Desliga Relé.
7	GND.
8	Entrada de acionamento do Relé de 5 V. 1 → liga Relé; 0 → Desliga Relé.
9	GND.
10	Entrada do PWM.
11	GND.
12	Entrada de SSD para o *gate driver*. 1 → os *gate drivers* entram em SSD; 0 → os *gate drivers* saem do estado de SSD.
13	GND.
14	Saída de leitura de FAULT/SD dos *gate drivers*. 0 → os *gate drivers* estão no estado de FAULT/SD; 1 → estão livres de falhas.
15	GND.
16	Entrada de comando de reset de falha. 1 → Reset da falha existente. 0 → *Gate*s sem comando de reset de falha.
17	GND.
18	Não conectado.
19	GND.
20	Não conectado.

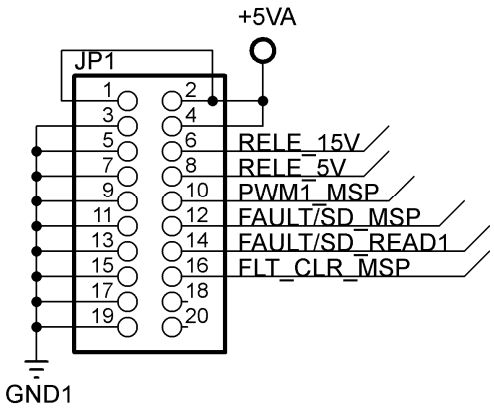


Figura 21: Esquemático do conector de interface da cadeira com a placa de controle.

1.5 Conclusão

As placas de acionamento apresentaram desempenho satisfatório, não apresentando qualquer falha durante um longo período de teste que compreendeu todo o desenvolvimento do projeto, visto que estas foram as primeiras placas a serem produzidas. A arquitetura deste projeto que divide os sistema de potencia do sistema de controle se mostrou bastante versátil, visto que em caso de falha de qualquer de suas placas, a substituição, ou até mesmo alteração do projeto é simplificada.

Alguns detalhes construtivos destas placas se destacam. O desenvolvimento das pontes H utilizando apenas um PWM para cada motor se mostrou muito eficaz, contudo, exige o uso *gate drivers* com geração de tempo morto próprio. A topologia dos componentes de potência na placa de acionamento é outro ponto de destaque, visto que o capacitor em paralelo a cada lado da ponte deve estar fisicamente próximo a este. Isso devido ao fato que qualquer cabo ou trilha adicionado entre os capacitores e os transistores gera uma indutância que a 20 kHz resulta em perda de capacidade de corrente da ponte H. A adoção do sistema optoacoplante foi de suma importância, visto que em testes realizados sem os mesmo, o sistema de controle paralisava quando os motores eram ligado, isso devido à circulação de ruídos de alta potência no sistema de controle.

2 HARDWARE DE CONTROLE E GERENCIAMENTO DE BAIXO NÍVEL DA CADEIRA DE RODAS

Para suportar os algoritmos de baixo nível que incluem controlador de baixo nível, controle de sensores e controle de periféricos a Cadeira de Rodas Robotizada dispõe de duas placas de processamento. Estas placas atuam de forma complementar para desempenhar os serviços de controle da Cadeira de Rodas Robotizada. Este Capítulo aborda o projeto de tais placas, sendo possível entender o projeto, premissas e toda a topologia de *hardware* projetada para suportar os algoritmos de controle de baixo nível e suas interfaces com os demais sistemas e periféricos. Na Figura 22 é possível identificar a atuação das placas de controle e processamento objetos deste Capítulo na arquitetura básica do projeto geral.

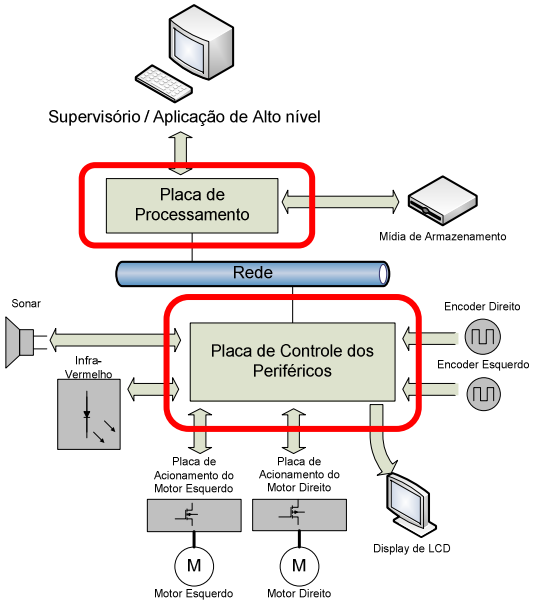


Figura 22: Arquitetura básica da Cadeira de Rodas Robotizada com as placas de controle e processamento destacadas.

2.1 Arquitetura Geral

O projeto do *hardware* de controle de baixo nível é responsável por suportar os algoritmos de controle e também todos os periféricos da cadeira e ainda centralizar as informações provenientes destes. Sua arquitetura suporta ainda futuras ampliações e adição de novos dispositivos. Outra característica importante é a disponibilização, de modo seguro, de

variáveis de controle em rede, isso para permitir que futuras adições de *hardware* não necessitem modificar a estrutura já existente. Assim, algumas premissas de projeto devem ser seguidas:

- Interfaces de comunicação com os diversos periféricos da cadeira;
- Capacidade de processamento para os algoritmos de controle;
- Suporte para aplicação de um *kernel* de tempo real;
- Atuação em rede, permitindo a adição de novos *hardwares*;
- Suporte para *log*, em mídia de armazenamento de dados, de variáveis previamente selecionadas;
- Gestão de todo o sensoriamento da cadeira de rodas;

A rede de comunicação entre as placas de controle dos periféricos e Placa de Processamento (PP) deve ter da capacidade de expansão. Outra premissa que a rede deve contemplar é que o acesso, o endereçamento e priorização dos pacotes de informação não se façam por dispositivo, mas sim por relevância de suas informações. Isto se justifica pela arquitetura do sistema, que não obedece à hierarquia de dispositivos, mas sim das informações provenientes de cada um, portanto, permitindo adição de novas funcionalidades apenas adicionando novos *hardware*s e não atualizando os já existentes.

Diante de tais características, além das sempre requeridas detecção de erros e robustez do protocolo, a rede escolhida foi a CAN (*Control Area Network*).

O detalhamento da implementação e outras justificativas para o usa da rede CAN estão no Capítulo 4. Assim, este Capítulo abordará apenas justificativas de projeto de *hardware*.

A divisão do controle em duas placas, sendo uma para o processamento chamado de Placa de Processamento (PP) e outra para o controle dos periféricos chamada de Placa de Controle dos Periféricos (PCP), interligadas por rede de comunicação, justifica-se por ser mais segura e barata para a manutenção que a implementação de apenas uma placa com todas as funções.

O aumento da segurança é justificado pela divisão de responsabilidades, ou seja, a Placa de Controle dos Periféricos (PCP) detém o controle de mais alta prioridade da cadeira. Se por qualquer motivo houver mais de um comando para a cadeira, perda de comunicação ou mesmo se a Placa de Processamento (PP) por qualquer motivo parar o processamento, a Placa de Controle dos Periféricos, ao detectar a situação de risco, irá parar a cadeira.

A redução de custos de manutenção é justificada pelo barateamento das placas, ou seja, no caso de queima de uma das placas a troca é apenas da mesma e não de todo o sistema.

Também por questões de redução de custos adotou-se como microcontrolador padrão o MSP430F2618 da *Texas Instruments* [21]. No caso de produção em larga escala, a padronização de componentes se mostra atrativa financeiramente.

2.2 O Microcontrolador MSP430F2618

O dimensionamento do microcontrolador considerou os seguintes aspectos:

- Periféricos internos;
- *Clock* suficiente para a aplicação;
- Quantidade de memória RAM[8];
- Quantidade de memória *Flash*[9];
- Depuração de programa durante sua execução;

O MSP430F2618 da *Texas Instruments* [21] possui diversos periféricos internos implementados em *hardware* que otimizam o processamento das rotinas. Seu *clock* é de até 16 MHz. Possui ainda memória RAM de 8 kB e *Flash* de 116 kB. Oferece ainda uma

---

[8] Memória RAM: *Random Access Memory*. Memória de dados digitais volátil e de acesso não seqüenciado.

[9] Memória *Flash*: Memória de dados digitais não volátil de alta velocidade de escrita.

interface JTAG[10] para depuração de programa. Abaixo está a lista de periféricos e características do MSP430F2618:

- Faixa de Tensão de alimentação de baixa, de 1,8 V a 3,6 V;
- Consumo ultra-baixo de potência:
  -- Modo Ativo : 365 μA em 1 MHz, 2.2 V
  -- Modo *Standby* (VLO): 0.5 μA
  -- Modo *Off* (retenção da RAM): 0.1 μA
- Arquitetura RISC de 16 *Bits* e ciclo de instrução de 62,5 ns;
- Conversor analógico para digital de 12 Bits;
- *Timer* A de 16 *Bits* e três canais de amostragem / temporização;
- *Timer* B de 16 *Bits* e sete canais de amostragem / temporização;
- Comparador analógico interno;
- Quatro interfaces seriais universais (USCIs) com suporte para I2C, SPI™ [22] e UART;
- Monitor de tensão de alimentação;
- *Bootstrap loader*;
- 116KB+256B de memória *Flash* e 8KB de memória RAM;
- Memória RAM de acesso plano;
- Registrador de *Stack pointer*.
- Pilha localizada na memória;

Estas duas ultimas características são importantes para a implementação do *kernel* de tempo real detalhado no Capítulo 5.

A seção seguinte abordará a detalhes do projeto da Placa de Controle dos Periféricos.

---

[10] JTAG: *Joint Test Action Group.* Interface de programação e depuração de microcontroladores que permite acesso à memória deste durante a execução.

2.3 O Projeto da Placa de Controle dos Periféricos (PCP)

O projeto conceitual da PCP inclui além das interfaces representadas na Figura 22 uma fonte de alimentação com capacidade para suportar todos os periféricos que necessitam de alimentação e que seja capaz de ser suprida pela bateria principal da cadeira de rodas, além de interface serial com o PC de contingência e configuração específica e capacidade de conexão com o barramento CAN.

Na Figura 23, está representado o esquema conceitual da PCP da cadeira de rodas. Nele ficam estabelecidas as relações entre as diversas interfaces, fonte de alimentação e microcontrolador.

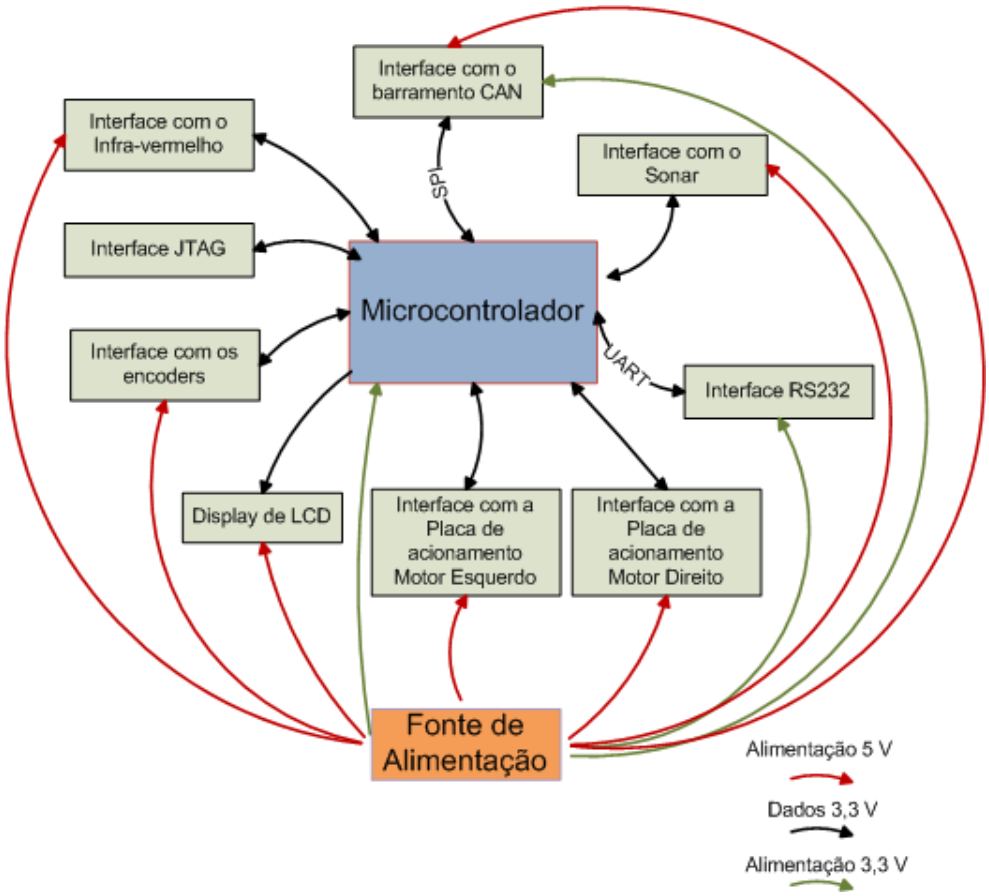


Figura 23: Esquema conceitual da Placa de Controle dos Periféricos (PCP).

A seguir está o detalhamento de cada bloco representado na Figura 23.

2.3.1 A Fonte de Alimentação da PCP

A fonte da PCP, como mostra a Figura 23, deve prover alimentação para todas as interfaces e subsistemas presentes nesta placa, além dos periféricos conectados à mesma.

Para o dimensionamento da fonte alguns fatos devem ser considerados:

- A entrada para a fonte, e a tensão a qual ela é encarregada de estabilizar, é a bateria principal da cadeira de rodas, cujo valor nominal é de 24 V. Contudo, em operação a plena carga a bateria, devido a sua impedância interna, apresenta em seus terminais uma oscilação de tensão com picos de até 30 V, e fonte deve suportar tal oscilação. Esta constatação foi possível apenas com experimentos práticos;
- Existem cargas de 5 e 3,3 V de alimentação;
- A fonte primária de energia é uma associação em série de duas baterias, assim, a fonte é escassa fazendo que a economia seja premissa.

O dimensionamento ainda deve considerar o estudo de cargas a serem alimentadas pela fonte em questão. Assim, a Tabela 2 ilustra o estudo de cargas considerando dados de fabricantes e experimentos práticos.

Tabela 2: Estudo de cargas alimentadas pela fonte da Placa de Controle dos Periféricos da Cadeira de Rodas

Circuito	Quantidade	Tensão (V)	Corrente por unidade (mA)	Corrente Total (mA)
Interface com o Infravermelho **[23]**	4	5	35	140
Interface com a Placa de Acionamento de Motor (Capítulo 1)	2	5	150	300
Interface com *encoder* **[24]**	2	5	50	100
Display de LCD **[25]**	1	5	20	20
Interface RS232 **[26]**	1	3,3	1	1
Interface com o Sonar	1	5	100	300

[27]				
Interface com o barramento CAN **[28][29]**	1	5 e 3,3	20	20
Microcontrolador **[21][2]**	1	3,3	20	20
Carga Total 5V	**11**	**5**	**---**	**901**
Carga Total 3,3V	**3**	**3,3**		**51**

Existem ainda nas interfaces dos *encoders*, sonar, Barramento CAN e Placas de Acionamento, CIs 74HC244 alimentados em 3,3V, contudo, sua demanda de carga não é significativa [30].

Para atender as exigências acima citadas a topologia escolhida foi a de uma fonte chaveada de 5 V em série com um regulador de tensão de 3,3 V (Figura 26). Esta topologia permite alimentar tanto os componentes de 5 V quanto os de 3,3 V. Devido ao uso de uma fonte chaveada a configuração possui ainda bom tempo resposta a surtos e oscilações de tensão, larga faixa de tensão de entrada e ótimo rendimento [31][32].

O circuito projetado se baseia no circuito integrado TPS5430 da *Texas Instruments* [32] e sua configuração típica está ilustrado na Figura 24.

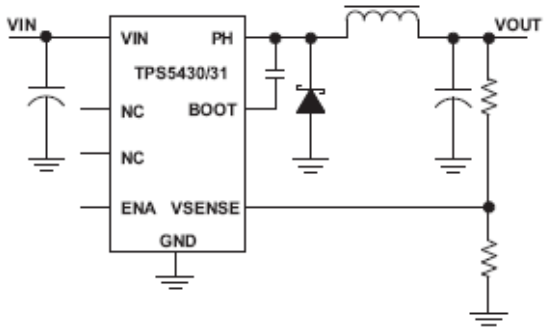


Figura 24: Circuito típico de uma fonte chaveada usando o TPS5430/31.

O TPS5430 [32] possui parâmetros elétricos satisfatórios para o projeto:

- Faixa de tensão de entrada de 5,5 até 36 V;
- Corrente de saída máxima de 3 A com picos de até 4 A;

- Rendimento de 95 %;
- Tensão de saída ajustável com o mínimo de 1,22 V;
- Frequência de chaveamento de 500 kHz;
- MOSFET interno com impedância interna de 110 mΩ;

Assim, objetivando atender as premissas estabelecidas nesta seção para dimensionamento da fonte, a configuração adotada para a fonte chaveada de 5 V está ilustrada na Figura 25.

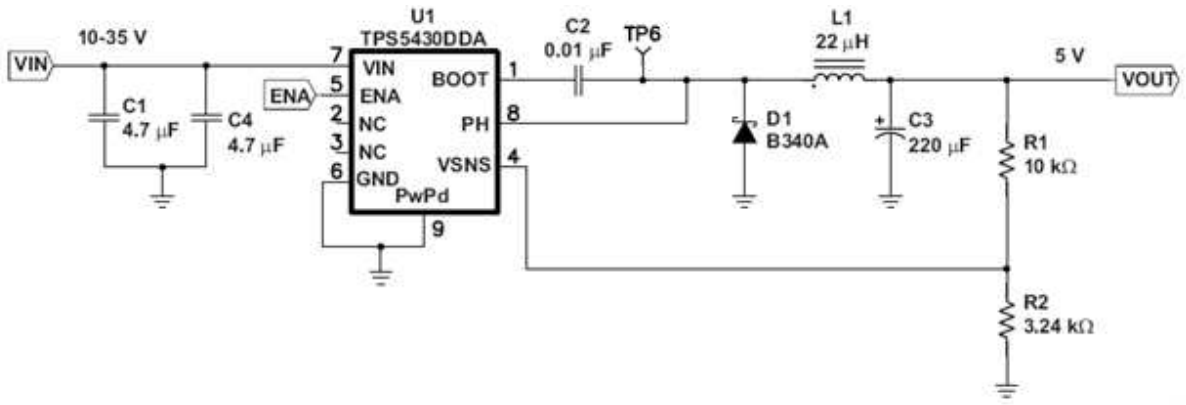


Figura 25: Configuração de uma fonte chaveada baseada no TPS5430 com tensão de saída de 5V e entra de 10-35 V.

Esta configuração possui os seguintes parâmetros de projeto:

- Tensão de entrada de 10 até 35V;
- Tensão de saída regulada para 5V;
- Corrente máxima de saída de 3A;
- Rendimento de 93%;
- *Ripple* na tensão de saída de 30 mV;

Esta configuração é um dos exemplos de aplicação do TPS5430 e está presente em [32]. Neste projeto, optou-se por escolher dispositivos cujos parâmetros elétricos têm valores acima dos parâmetros do TPS5430, ou seja, este é o dispositivo que determina os parâmetros máximos de operação da fonte. Maiores detalhes sobre o dimensionamento dos demais componentes da fonte chaveada encontram-se em [32].

O regulador de 3,3V escolhido para o projeto é o REG1117-33 [33]. A justificativa para esta escolha é seu limite mínimo de tensão de entrada, 4,4V, e sua capacidade de corrente é de 800 mA. Assim, o circuito final da fonte de alimentação da PCP com suas duas saídas de tensão, 3,3 V e 5 V, está representado na Figura 26.

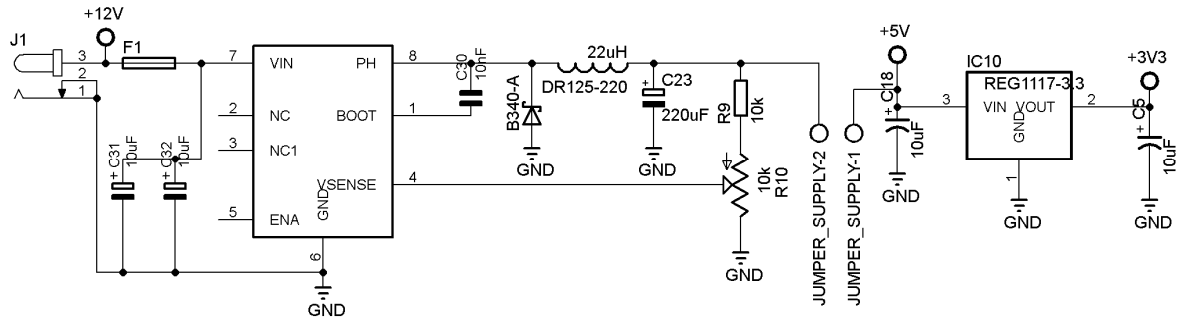


Figura 26: Esquemático do projeto da fonte de alimentação da Placa de Controle dos Periféricos.

Para ajuste de tensão de saída da fonte chaveada foi disponibilizado um potenciômetro e um *jumper* (Figura 26). O procedimento de *setup* é:

- Ainda com a placa desligada deve-se abrir o *jumper supply;*
- Com o *jumper supply* aberto deve-se ligar a placa conectando a fonte de alimentação de 10 até 35 V CC ao conector J1;
- Com a placa energizada deve-se acoplar um voltímetro na saída do *jumper supply* 2 e ajustar o potenciômetro até o valor de 5 V;
- Com o valor de saída da fonte chaveada ajustada para 5 V, deve-se então desligar a fonte, fechar o o *jumper supply* e a placa estará pronta para ser ligada novamente em sua plenitude*;*

2.3.2 Interface Com os *Encoders*

Os *encoders* incrementais, usados neste projeto, são dispositivos capazes de fornecer um sinal com frequência variável de acordo com a sua velocidade de giro. Maiores detalhes sobre estes encontram-se na seção 3.1 deste trabalho.

Para esta seção as informações pertinentes sobre os *encoders* são seus parâmetros elétricos, que para esta aplicação são níveis TTL;

O nível da tensão do microcontrolador é 3,3 V. Assim, para interface entre os *encoders* e o microcontrolador é usado o CI 74HC244 [34] alimentado em 3,3 V.

Portanto, o circuito final de interface do microcontrolador MSP430F2618 com os *encoders* está ilustrado na Figura 27 e o esquema de ligação dos conectores para os *encoders* na Placa de Controle dos Periféricos está ilustrado na Figura 28.

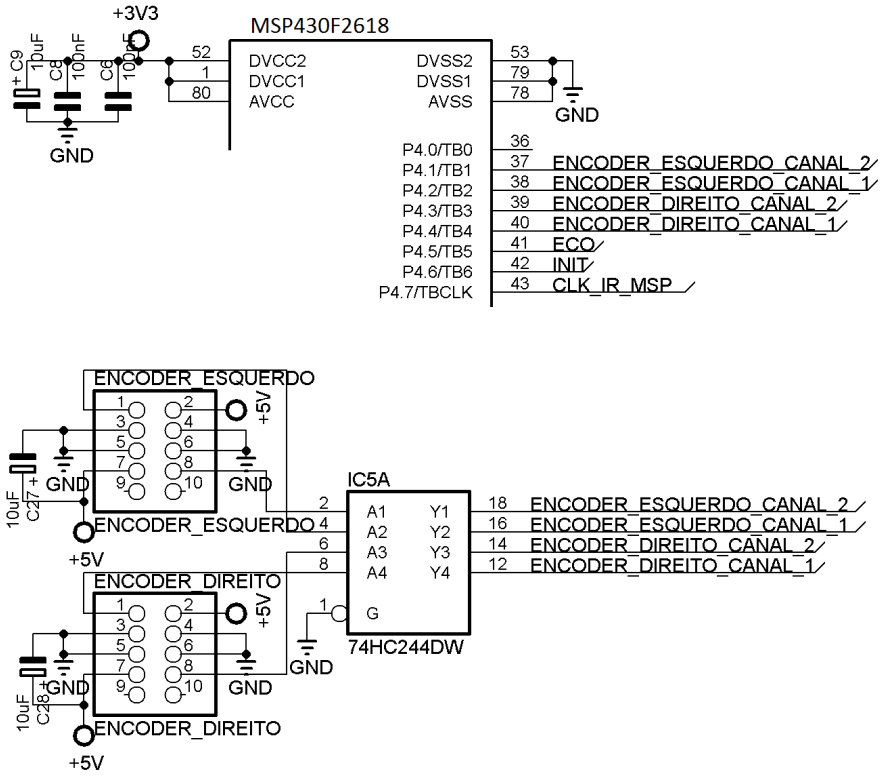


Figura 27: Circuito de interface do microcontrolador com os *encoders* da cadeira de rodas.

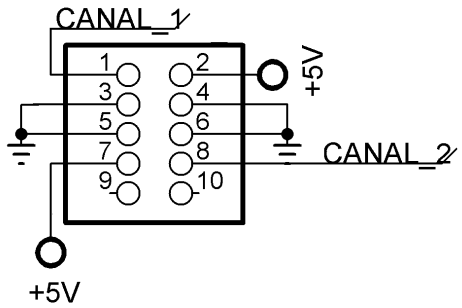


Figura 28: Esquemático de ligação do conector dos *encoders* na Placa de Controle dos Periféricos.

2.3.3 Interface Com o Display de LCD

Visando uma interface com usuário rápida de variáveis importantes da cadeira de rodas foi disponibilizado suporte para um display de LCD. Mais detalhamento sobre este encontram-se na seção 3.5.

Para esta seção é pertinente apenas salientar que os níveis elétricos são TTL, sendo que a comunicação entre microcontrolador e display é unidirecional e segue apenas do microcontrolador para o display. Portanto, não se faz necessária mudança de nível nos sinais de comunicação, visto que o valor mínimo de tensão do display é 2,5 V para nível lógico 1, e o microcontrolador fornece 3,3 V. O esquemático de ligação do conector do display está representado na Figura 29.

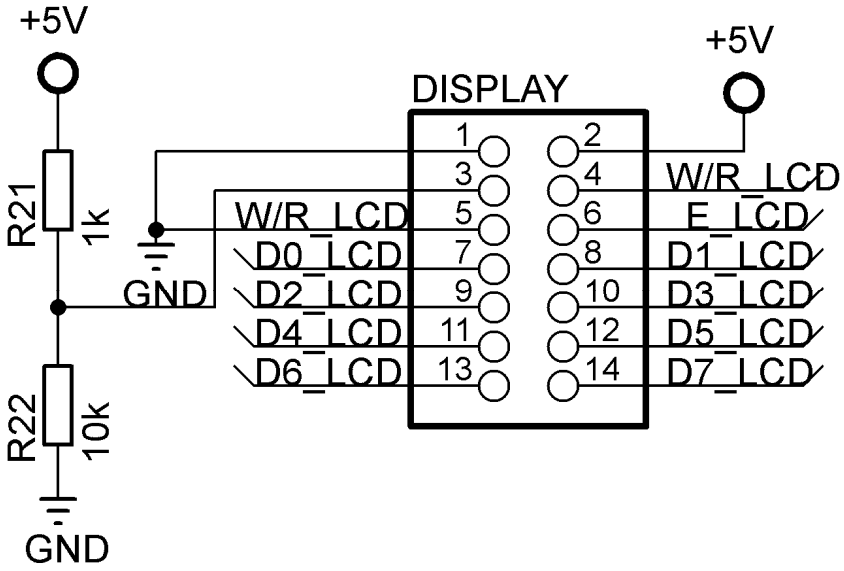


Figura 29: Esquemático de ligação do conector do display de LCD.

2.3.4 Interface Com as Placas de Acionamento dos Motores

Para conexão com as Placas de Acionamento dos motores da cadeira foram disponibilizados dois conectores, sendo um para cada motor (Figura 30). Os níveis elétricos destas placas são TTL, sendo assim, para os sinais oriundos das Placas de Acionamento em direção ao microcontrolador foi usado novamente o 74HC244, este sendo alimentado em 3,3 V. maiores detalhes sobre a Placa de Acionamento dos motores no Capítulo 1.

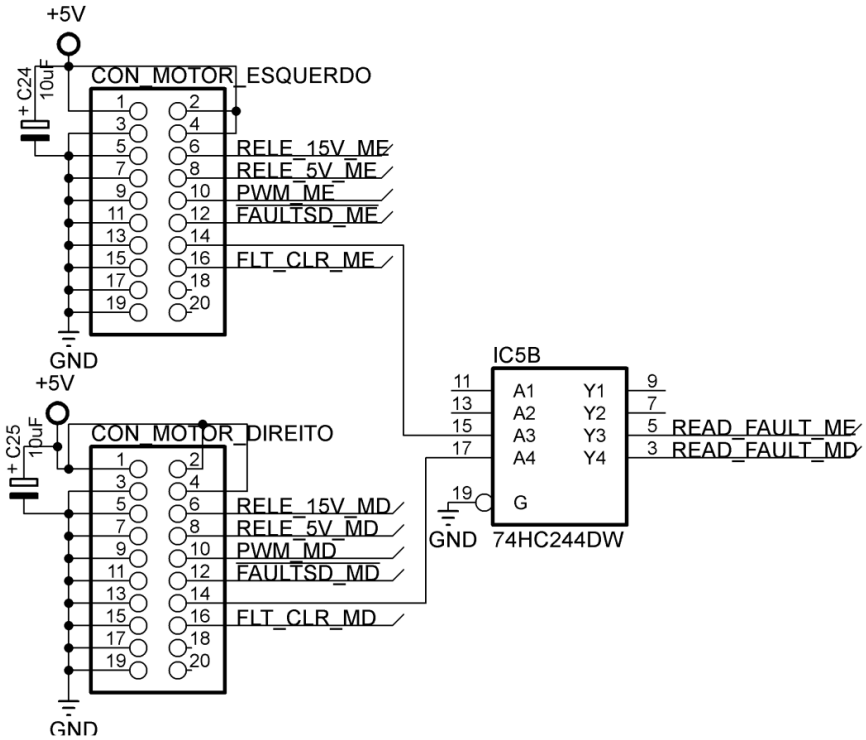


Figura 30: Esquemático de ligação dos conectores da Placa de Controle dos Periféricos para as Placas de Acionamento dos motores.

2.3.5 Interface RS-232 de Comunicação e *Bootstrap Loader* (BSL) [35][36]

Visando prover comunicação de *setup* da PCP com um PC foi disponibilizada uma interface tradicional.

Outra funcionalidade atribuída a esta interface é prover a gravação do microcontrolador pelo protocolo *bootstrap loader* [35][36]. Este protocolo permite que os usuários se comuniquem com a memória no microcontrolador MSP430 durante a fase de prototipagem, a produção final, e em serviço. Tanto a memória programável (memória *flash*) quanto a memória de dados (RAM) pode ser modificado, conforme a necessidade.

Assim, o circuito implementado está mostrado Figura 31 e tem como base o circuito típico do MAX3243 que está disponível em [26].

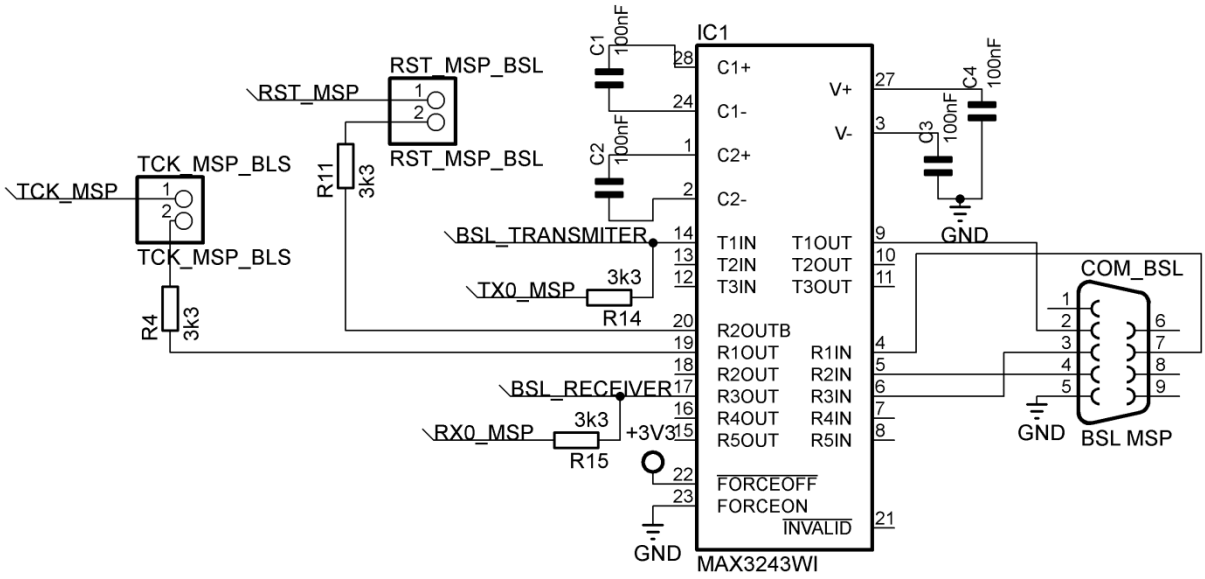


Figura 31: Circuito de comunicação RS-232 e BSL implementado naPlaca de Controle dos Periféricos.

Por questões de proteção elétrica e visando separação de funcionalidade foram disponibilizados dois *jumpers* para configurar a interface RS-232 em interface serial ou BSL. Estes *jumpers* são os conectores TCK_MSP_BLS e RST_MSP_BLS, sendo que, para configurar a interface serial comum estes conectores devem ser mantidos em aberto, e para configurar como interface BSL estes devem ser fechados. O esquema de ligação do conector serial desta interface está representado na Figura 32.

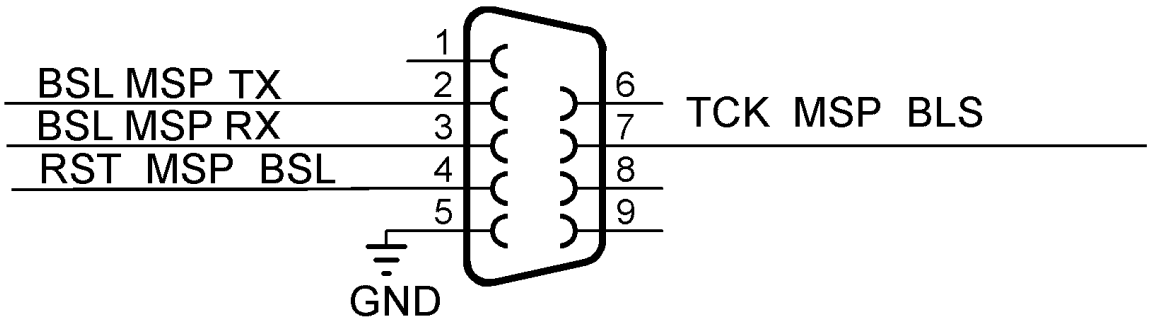


Figura 32: Esquemático de ligação do conector RS-232 da Placa de Controle dos Periféricos.

2.3.6 Interface Com o Sonar

Para comunicação do microcontrolador com o sonar idealizado para o projeto da cadeira de rodas foi disponibilizado uma interface baseada novamente no 74HC244, visto que o sonar usa níveis TTL. Maiores detalhes sobre o uso do sonar encontram-se na seção 3.2 e também em [27].

Portanto, a implementação desta interface está representada na Figura 33.

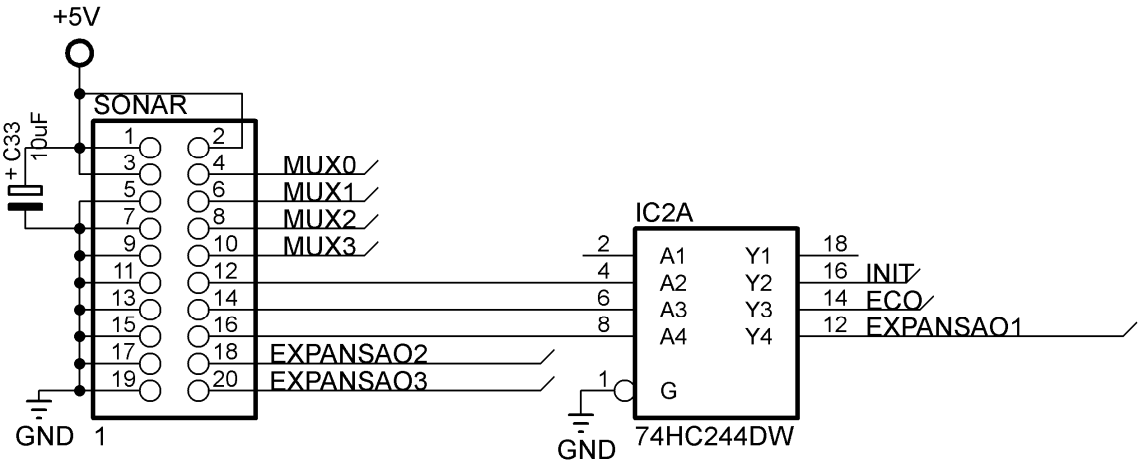


Figura 33: Circuito de interface com o sonar.

2.3.7 Interface Com os Sensores Infravermelho

Os sensores infravermelho usados no projeto foram os GP2D02 da SHARP [23]. Estes sensores possuem a característica de se comunicarem via protocolo próprio. Mais detalhes sobre este protocolo e sobre os sensores em questão encontram-se na seção 3.3.

Para esta seção, basta considerar o fato que os sensores usam níveis TTL. Assim, o circuito projetado está representado na Figura 34. Contudo, em testes práticos e ensaios em laboratório este circuito se mostrou ineficaz, visto que não era estabelecida comunicação entre o microcontrolador e os sensores. Vale ressaltar que este circuito está de acordo com o circuito de referência do sensor [37]. O circuito recomendado e usado como referência esta na Figura 35.

Visando corrigir o problema citado acima o projeto foi revisado, contudo, não foi possível implementar a nova placa com o projeto revisado. Assim, foi feito um ajuste na placa já existente, na qual foi retirado o diodo do sinal CLK_IR e colocado um divisor de tensão (Figura 36) no mesmo sinal.

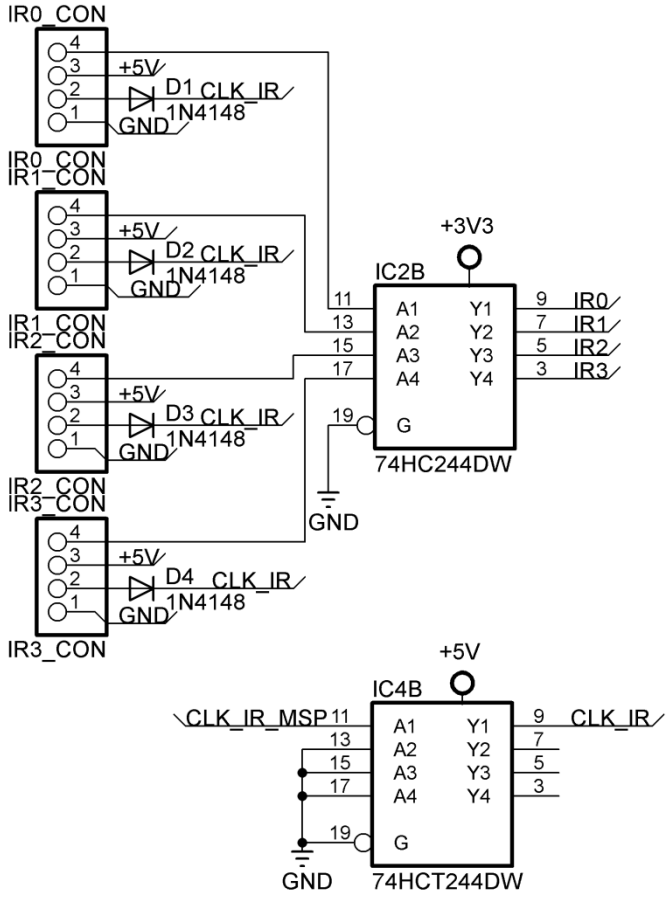


Figura 34: Circuito de interface com os sensores Infravermelho projetado, contudo, ineficaz.

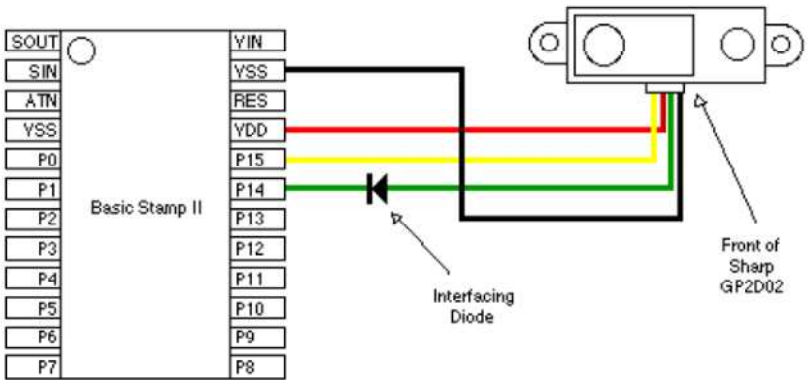


Figura 35: Circuito de referência para elaboração do projeto da Figura 34.

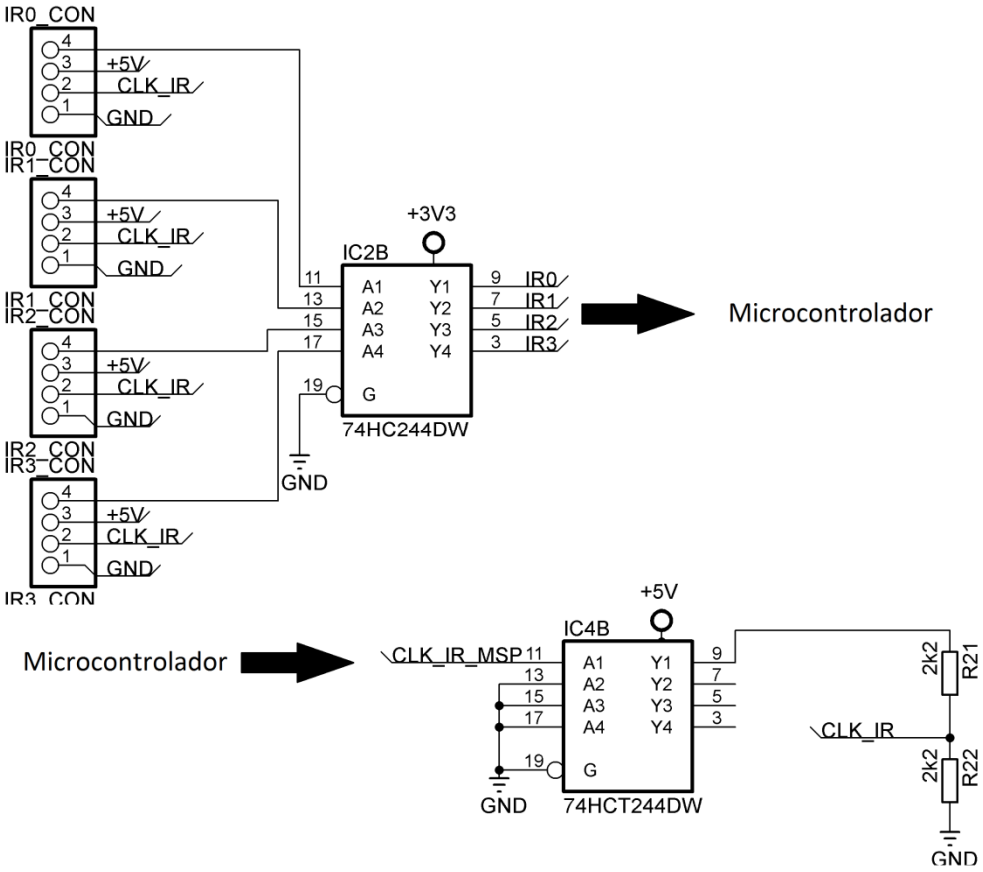


Figura 36: Circuito adaptado para interface com os sensores *infravermelho*.

Após a adaptação realizada no circuito da Figura 36 foi possível estabelecer comunicação com os sensores infravermelhos.

2.3.8 Interface JTAG do Microcontrolador

JTAG (*Joint Test Access Group*) é um protocolo para gravação e depuração de programas para microcontroladores e CIs programáveis de modo geral. O MSP430F2618 permite o uso de tal protocolo, e para usá-lo no projeto foi disponibilizado o circuito e conector ilustrado na Figura 37.

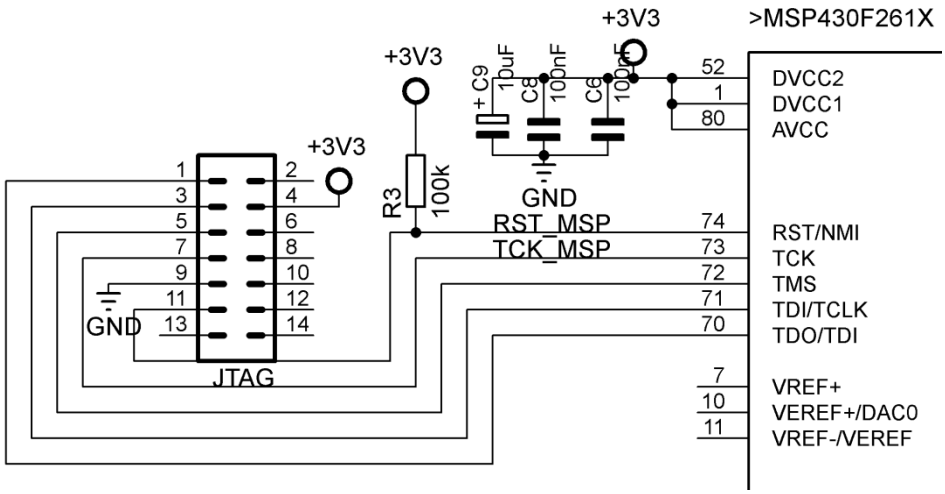


Figura 37: interface de gravação e depuração implementada no projeto.

2.3.9 Interface Com o Barramento da Rede CAN

Esta seção tem por objetivo detalhar o *hardware* projetado para interface com o barramento CAN (*Control Area Network*) utilizado para comunicação entre placas. Detalhes sobre o protocolo podem ser encontrados em [29]. A configuração, detalhes dos componentes e rotinas de software usadas para controle do barramento CAN serão detalhadas no Capítulo 4.

Para possibilitar o acesso da PCP ao barramento CAN foi projetada uma interface autônoma no que se refere ao processamento do protocolo. Esta interface tem como base o CI controlador CAN MCP2515 da *Microchip* [29].

A Figura 38 ilustra a atuação dos dispositivos CAN no barramento, na qual o *node controller* é o microcontrolador, o XCVR o transceiver e o MCP2515 é o controlador CAN.

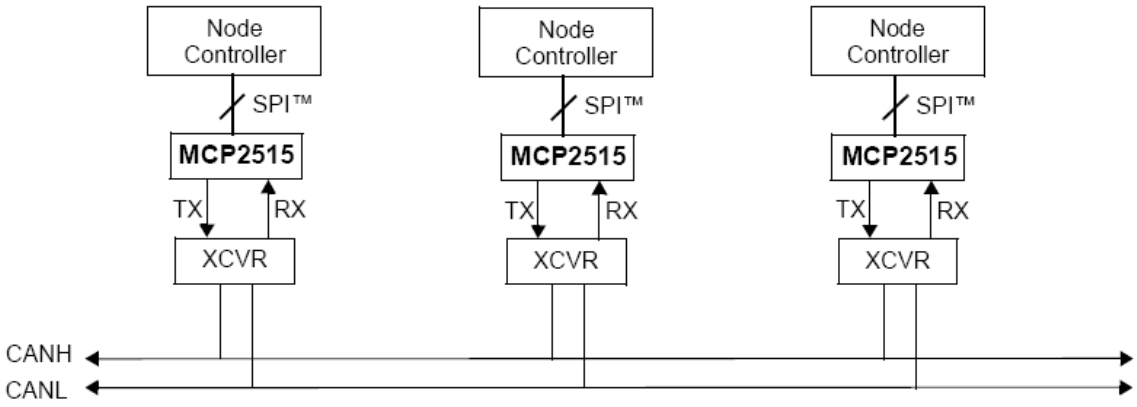


Figura 38: layout do barramento CAN com as respectivas atuações dos dispositivos integrantes.

Assim, o circuito implementado para desempenhar a função de interface do microcontrolador com o barramento CAN deve conter o MCP2515 como controlador de barramento, e o MCP2551 como *transceiver* para o meio físico do barramento. Contudo, por opção de projeto o MCP2515 foi alimentado em 3,3 V e o *transceiver* deve ter alimentação padrão do barramento CAN, 5 V. Portanto, fez-se necessária a utilização do 74HC244 como conversor de níveis entre o MCP2515 e o MCP2551. O circuito final de interface está ilustrado na Figura 39, na qual todas as linhas que contêm em seu nome o texto “MSP2” são sinais conectados diretos ao MSP430.

Por questões de opção de projeto o conector padrão é do tipo USB, isso para permitir fácil conexão e uso de hub para o barramento. A alimentação do barramento está conectada diretamente à entrada de alimentação da PCP, isso para alimentar as demais placas do barramento.

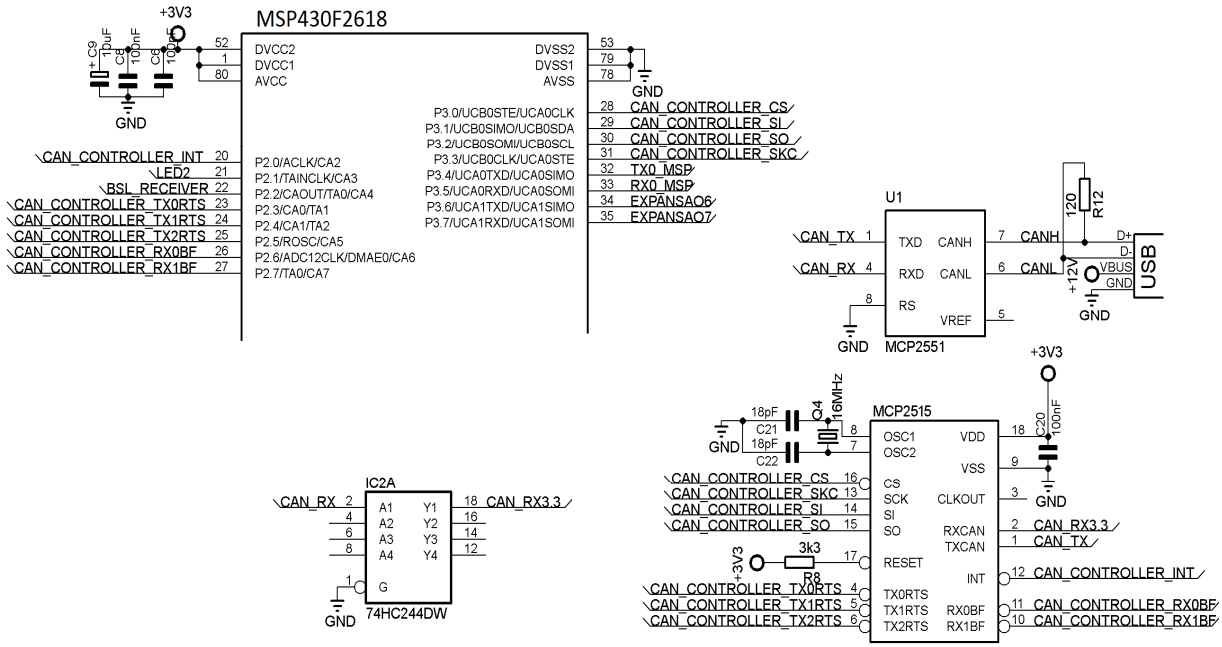


Figura 39: Circuito de interface do microcontrolador com o barramento CAN.

2.3.10 Considerações Finais Sobre a Placa de Controle dos Periféricos

O projeto da Placa de Controle dos Periféricos se mostrou dentro das expectativas, visto que todas as interfaces desempenharam bem suas funções nos testes e ensaios. Como relatado na seção 2.3.7, a interface com os sensores infravermelhos teve de ser ajustada para funcionar plenamente.

Outro fato ocorrido durante o projeto foi a inversão dos pinos 1 e 8 do TPS5430. A primeira versão da fonte que foi a implementada continha o erro supracitado. Diante de tal fato a placa fabricada foi manualmente ajustada e o projeto refeito. A Figura 26 mostra o circuito corrigido.

As características elétricas de alimentação da placa são limitadas pelas características da fonte chaveada, visto que esta é responsável pela alimentação de todos os dispositivos da placa e seus periféricos. Assim:

- Tensão de entrada de 10 a 35 V;

- Fusível de 1 A ligado antes da fonte chaveada. Obedecendo ao estudo de carga da seção 2.3.1;
- Em caso de mudança no potenciômetro, deve-se seguir o procedimento descrito na seção 2.3.1 para ajustar a tensão da fonte chaveada em 5 V;

Para sinalização foram disponibilizados dois LED, sendo que o LED verde indica atividade do barramento CAN e o amarelo indica erro de comunicação no barramento CAN.

Para futuras implementação e evolutivas no projeto, foi disponibilizado um conector de expansão. Este conector está representado na Figura 40 e suas características são:

- Pinos EXPANSAO1, EXPANSAO4 e EXPANSAO5 são entradas TTL;
- Pinos EXPANSAO2, EXPANSAO3, EXPANSAO6 e EXPANSAO7 são I/O 3,3 V direto nos pinos do microcontrolador;

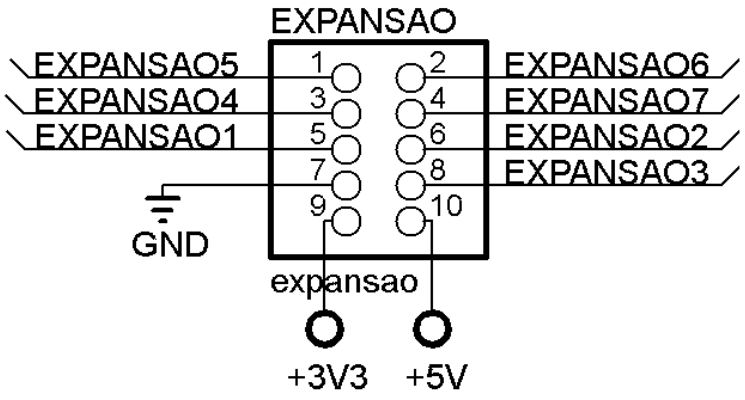


Figura 40: Conector de expansão e evolutivas da Placa de Controle dos Periféricos.

O esquemático final da Placa de Controle dos Periféricos já com os ajustes supracitados está ilustrado na Figura 41 a fotografia da placa com seus sistemas destacados está na Figura 42 e a disposição física dos componentes da placa está representada na Figura 43.

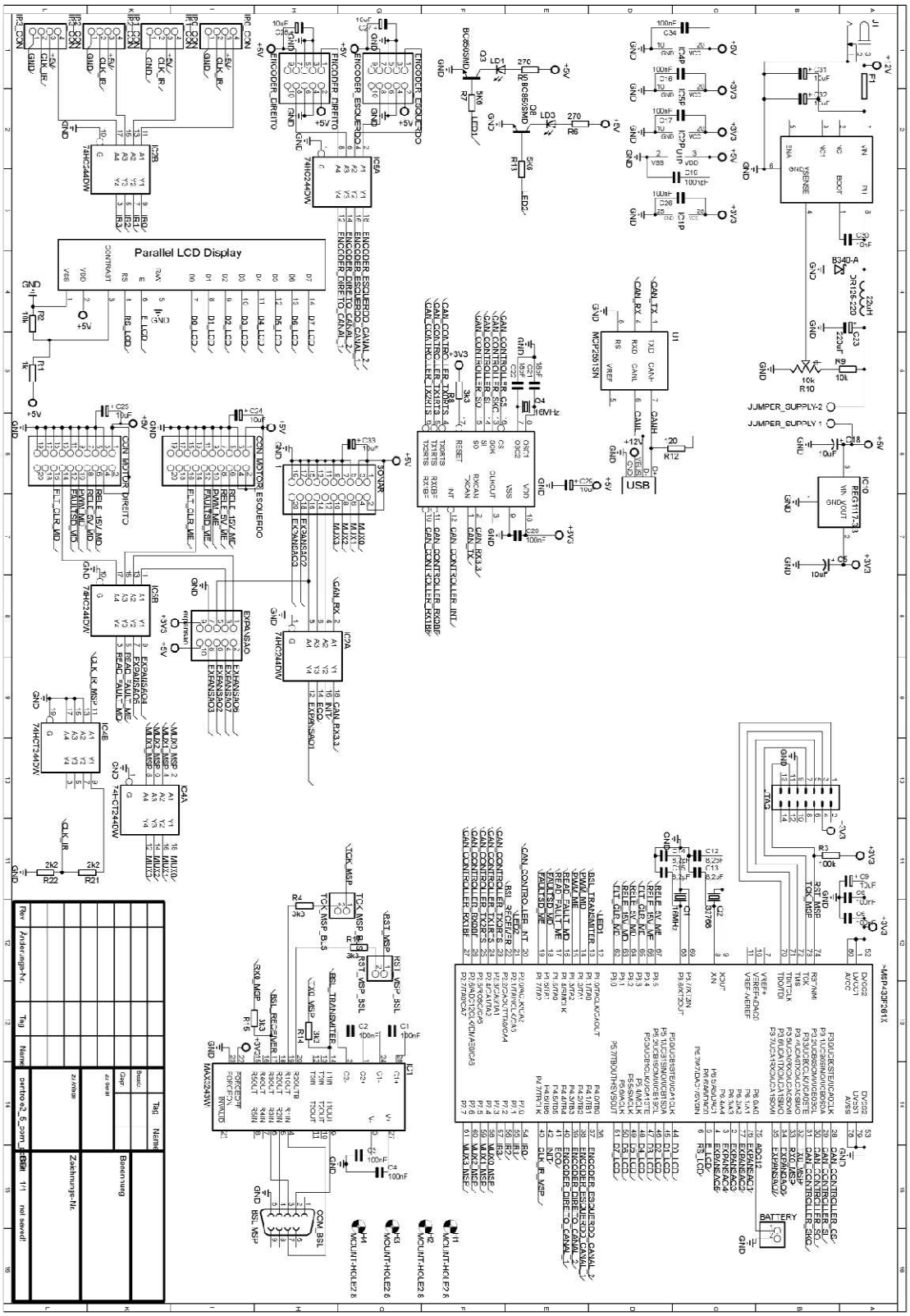


Figura 41: Esquemático final do projeto da Placa de Controle dos Periféricos.

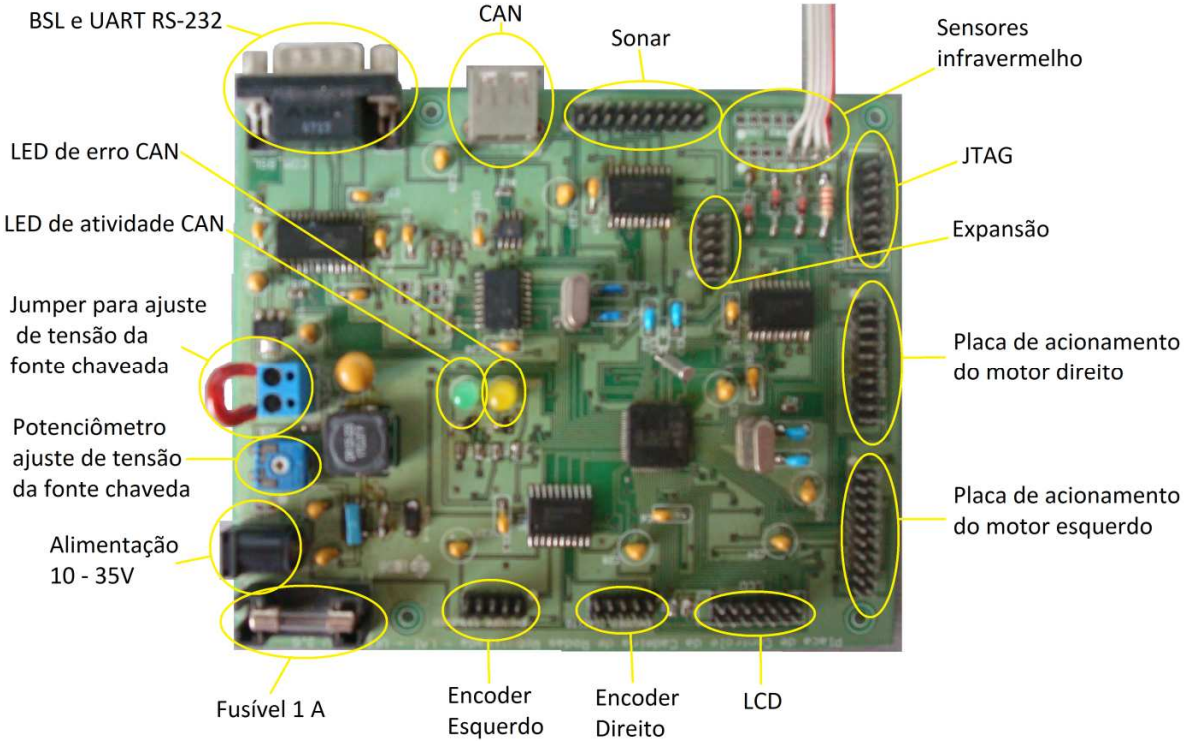


Figura 42: Fotografia da Placa de Controle dos Periféricos com seus sistemas destacados.

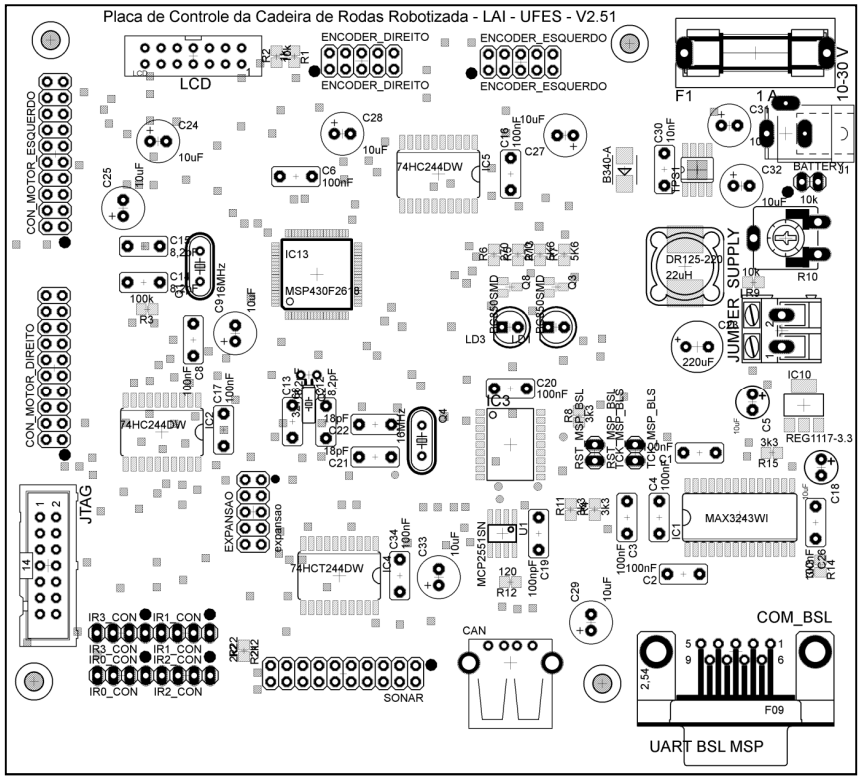


Figura 43: Disposição Física dos componentes na Placa de Controle dos Periféricos.

2.4 O Projeto da Placa de Processamento (PP)

O objetivo da PP é:

- Prover suporte computacional para os algoritmos de controle de baixo da cadeira.
- Armazenar informações pertinentes ao controle da cadeira;
- Comunicar com o PC que executa o controle de alto nível e interpretar seus comandos;

Para prover o suporte computacional necessário, foi utilizado o mesmo microcontrolador da PCP o MSP430F2618. Outros motivos para esta escolha estão na seção 2.2.

Para realizar o armazenamento das variáveis pertinentes ao controle da cadeira de rodas optou-se por usar um dispositivo comercial de *log* industrial que usa como base um cartão micro-SD. Este dispositivo é o DOSonChip CD17B10 [38] que será detalhado na seção 3.4.

A comunicação entre a PP com o PC que executa o controle de alto nível da cadeira é realizada por uma interface serial, visto sua facilidade de operação e simplicidade de implementação.

Para permitir rastreabilidade dos dados armazenados no micro-SD, foi implementado um RTC (*Real Timer Clock*). Assim, é possível determinar o momento de cada leitura de dados.

Assim, o esquema conceitual da Placa de Processamento dos dados está na ilustrado na Figura 44. Nela é possível observar a existência da interfaces RS-232 de BSL e UART, interface JTAG e interface com o barramento CAN. Estas interfaces são idênticas às mesmas interfaces da PCP. Portanto, maiores detalhes a respeito destas estão nas seções 2.3.5, 2.3.8 e 2.3.9. A seguir serão detalhados os demais módulos que compõe este esquema.

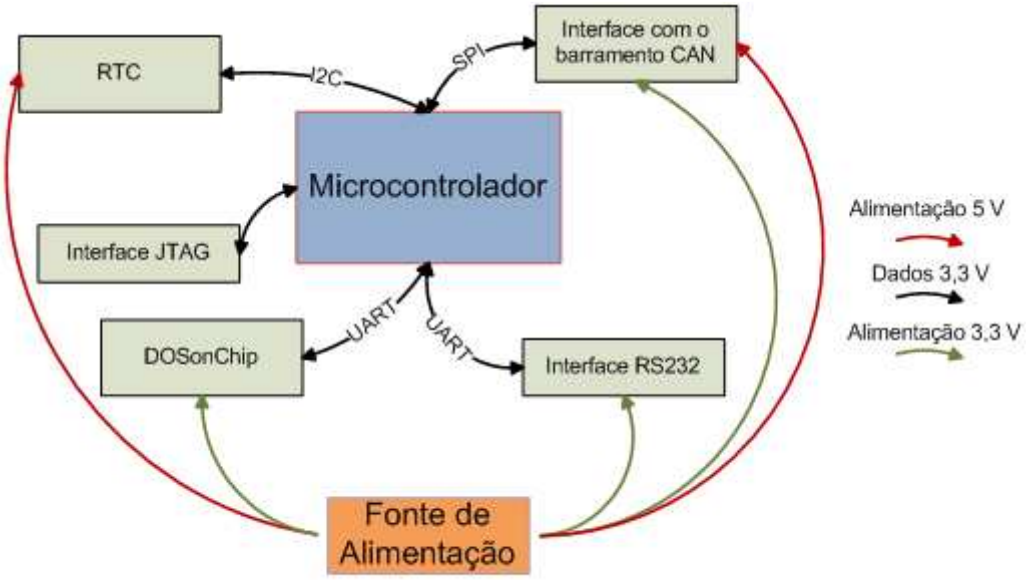


Figura 44: Esquema conceitual da Placa de Controle de Processamento.

2.4.1 A Fonte de Alimentação da Placa de Processamento

Observa-se na Figura 44 que no projeto conceitual existem componentes alimentados em 3,3 V e 5 V. Similar à Placa de Controle dos Periféricos optou-se pela arquitetura de duas fontes. Contudo, nesta placa optou-se por um regulador de tensão para a fonte de 5 V ao invés de uma fonte chaveada. Esta opção é devida ao fato da Placa de Processamento apresentar baixo consumo esperado (Tabela 3), não justificando o uso de uma fonte chaveada. A topologia adotada para as fontes novamente é o regulador de 3,3 V em série com o regulador de 5 V (Figura 45).

A Placa de Processamento usa a alimentação do barramento CAN como fonte de energia. Assim, a tomada de força da placa é oriunda do conector CAN (Figura 45).

O regulador de tensão de 5 V usado é o KA7805 [39] que tem como características elétricas:

- Tensão de saída de 5 V;
- Tensão de entrada máxima 35 V;
- Corrente de saída 800 mA;

O regulador de 3,3 V é o REG1117-33 [33] e segue o mesmo circuito detalhado na seção 2.3.1.

Tabela 3: Estudo de carga da Placa de Processamento

Circuito	Quantidade	Tensão (V)	Corrente por unidade (mA)	Corrente Total (mA)
Interface com DOSonChip	4	5	35	5
RTC	2	5	50	1,5
Interface RS232 **[26]**	1	3,3	1	1
Interface com o barramento CAN **[28][29]**	1	5 e 3,3	20	20
Microcontrolador **[21][2]**	1	3,3	20	20
Carga Total 5V	**11**	**5**	**---**	**47,5**
Carga Total 3,3V	**3**	**3,3**	**---**	**25**

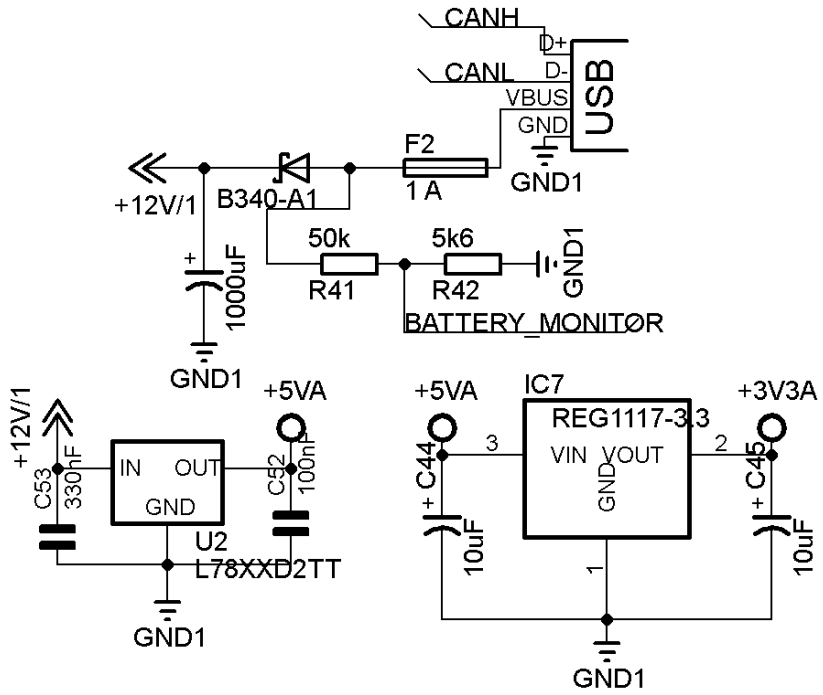


Figura 45: Circuito da Fonte de alimentação da Placa de Processamento.

2.4.2 Interface Com o DOSonChip

O CD17B10 DOSonCHIP [38] fornece uma interface para cartões de memória, incluindo microSDHC, microSD, miniSDHC, miniSD, SDHC e cartões SD. Ambas as interfaces física e interface com sistemas de arquivamento são incluídos no CD17B10.

Comandos junto com os dados do usuário são enviados para o CD17B10 sobre qualquer um de seus protocolos UART, SPI™ [22], ou I2C. Estes comandos são usados para navegar no diretório de sistema de arquivos, leitura de dados, ou gravar dados no cartão de memória usando formato padrão do sistema de arquivos FAT, que é diretamente compatível com PC, mídia digital players, câmeras digitais, etc.

Nesta aplicação foi usada uma revenda da *SparkFun Eletronics* do CD17B10 DOSonCHIP que utiliza cartão micro-SD e tem como interface os protocolos UART e SPI™ [22] (Figura 46). O protocolo escolhido para comunicar com o DOSonChip é o UART. Como a tensão de alimentação do DOSonChip é 3,3 V não há necessidade de qualquer interface de níveis de tensão entre este e o microcontrolador. Desta maneira a interface entre estes se resume a um conector (Figura 47).

Maiores detalhes sobre o controle e protocolo de comunicação com o DOSonChip estão adiante.

Figura 46: Dispositivo de revenda da *SparkFun Eletronics* do DOSonChip.

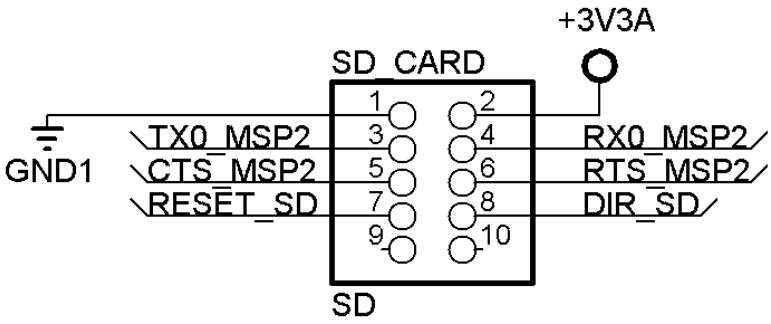


Figura 47: Conector de interface entre DOSonChip e o MSP430.

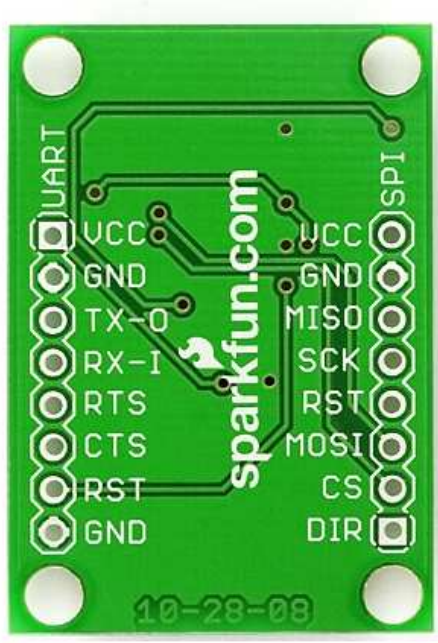


Figura 48: Conexões da placa DOSonChip usada no projeto.

A relação entre os pinos do conector da Placa de Processamento e os pinos da placa do DOSonChip da *SparkFun* está descrita na Tabela 4.

Tabela 4: Tabela de correspondência entre os pinos do DOSonChip *SparkFun* e o Conector da Placa de Processamento.

Pino Conector da Placa de Processamento	Pino interface UART DOSonChip *SparkFun*
1	GND
2	Vcc
3	RX-I

4	TX-O
5	CTS
6	RTS
7	RST
8	Não conectado
9	Não conectado
10	Não conectado

2.4.3 *Hardware* de Interface Com o RTC

O relógio de tempo real (RTC - *Real Time Clock*) projetado tem como base o CI DS1307 [40]. Mais detalhes sobre a comunicação e controle do DS1307 encontram-se nas seções adiante. Para esta seção as informações relevantes são as características elétricas e de protocolo de comunicação camada física com o RTC.

O DS1307 necessita de alimentação de 5 V mais uma bateria ou pilha de 3 a 3,3 V para manter seu *clock* quando a alimentação é interrompida. Seu protocolo de comunicação é o I2C de 100 kHz. O *clock* do DS1307 é proveniente de um cristal de 32768 Hz. Existe ainda uma saída de 1 Hz do DS1307 que é enviada para o microcontrolador. Assim, o circuito de tempo real implementado está ilustrado na Figura 49.

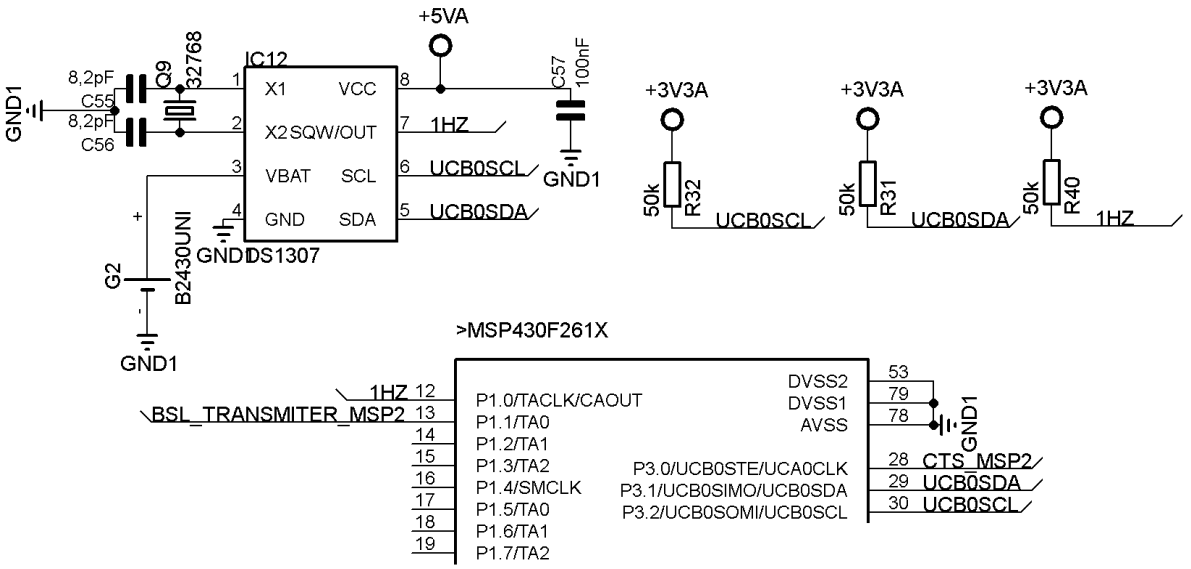


Figura 49: Circuito do RTC implementado no projeto.

2.4.4 Considerações Gerais Sobre a Placa de Processamento

As características elétricas da Placa de Processamento seguem as limitações impostas pelo regulador de tensão de 5 V, visto que este alimenta toda a placa, inclusive o regulador de 3,3 V. Assim, as características elétricas de entrada da Placa de Processamento são:

- Tensão de entrada de 10 a 35 V;
- Corrente nominal de 47,5 mA;
- Fusível de 200 mA;

Para prover melhor sinalização foram adicionados quatro LEDs vermelhos, sendo suas funções detalhadas na Figura 50.

Similar à Placa de Controle dos Periféricos, a Placa de Processamento dispõe de duas interfaces de expansão, “Porta 6” e “Entradas Auxiliares” da Figura 50, e suas características elétricas são:

- Pinos de “Entradas Auxiliares” são entradas TTL;
- Pinos de “Porta 6” são I/O de 3,3 V direto nos pinos do microcontrolador;

As conexões das interfaces de expansão, “Porta 6” e “Entradas Auxiliares” estão detalhadas na Figura 51 e a disposição final dos componentes na Placa de Processamento está ilustrada na Figura 52.

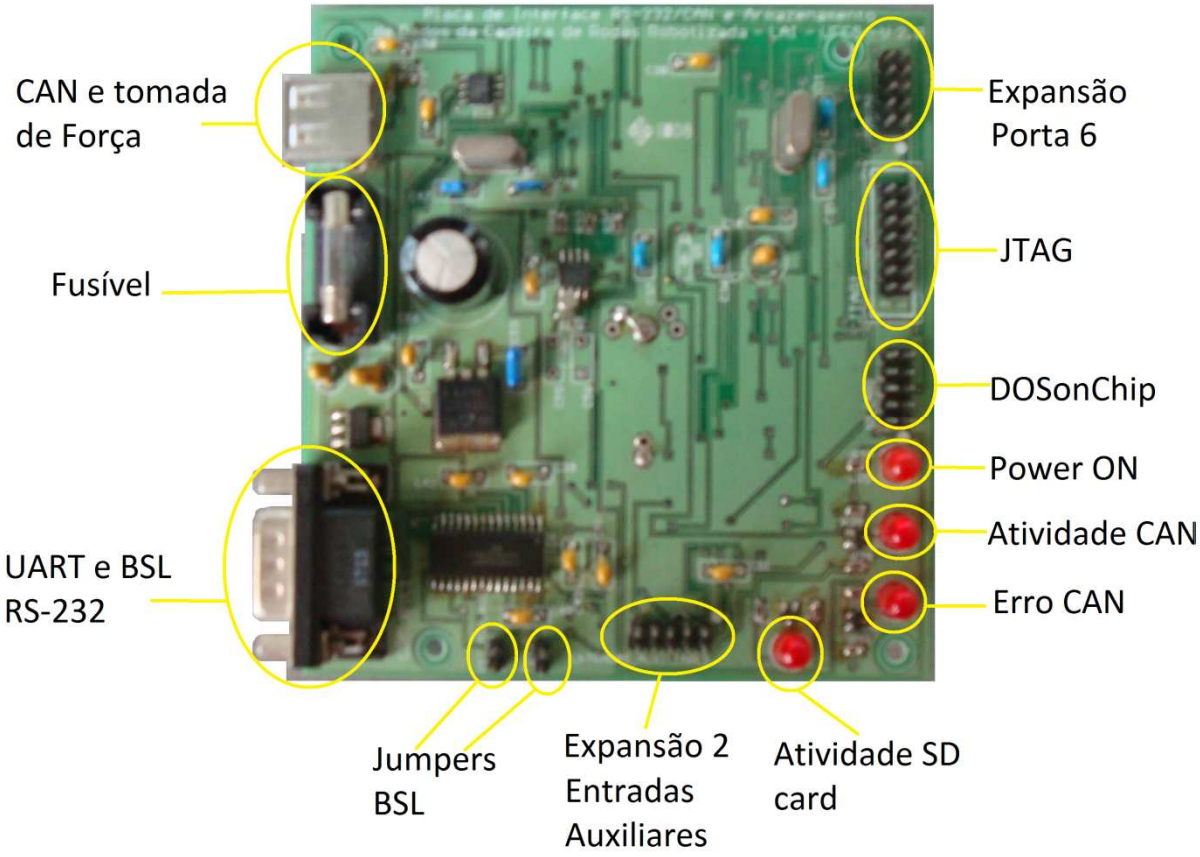


Figura 50: Fotografia da Placa de Processamento com seus sistemas destacados.

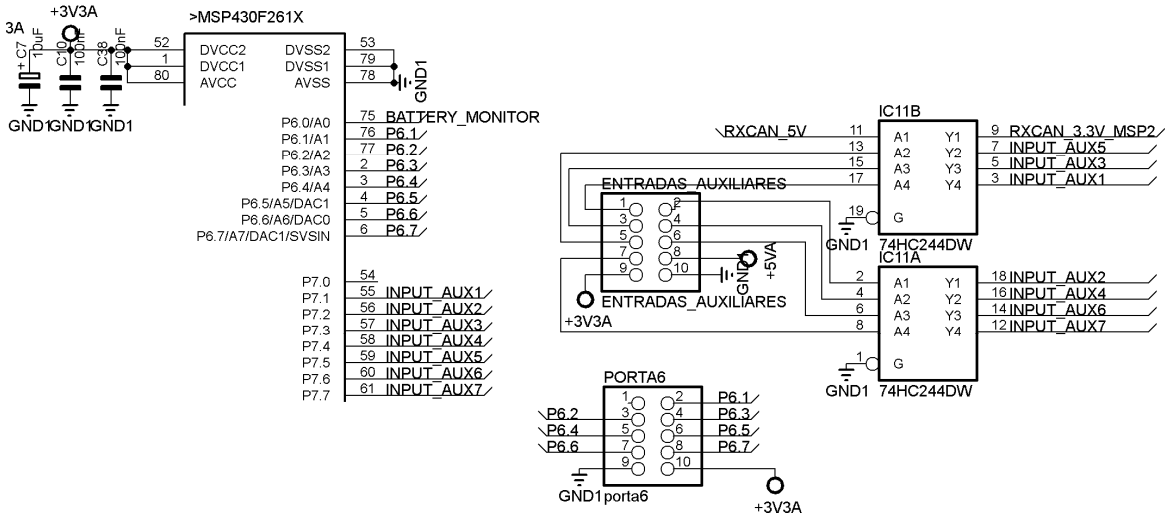


Figura 51: Conexões das interfaces de expansão da Placa de Processamento.

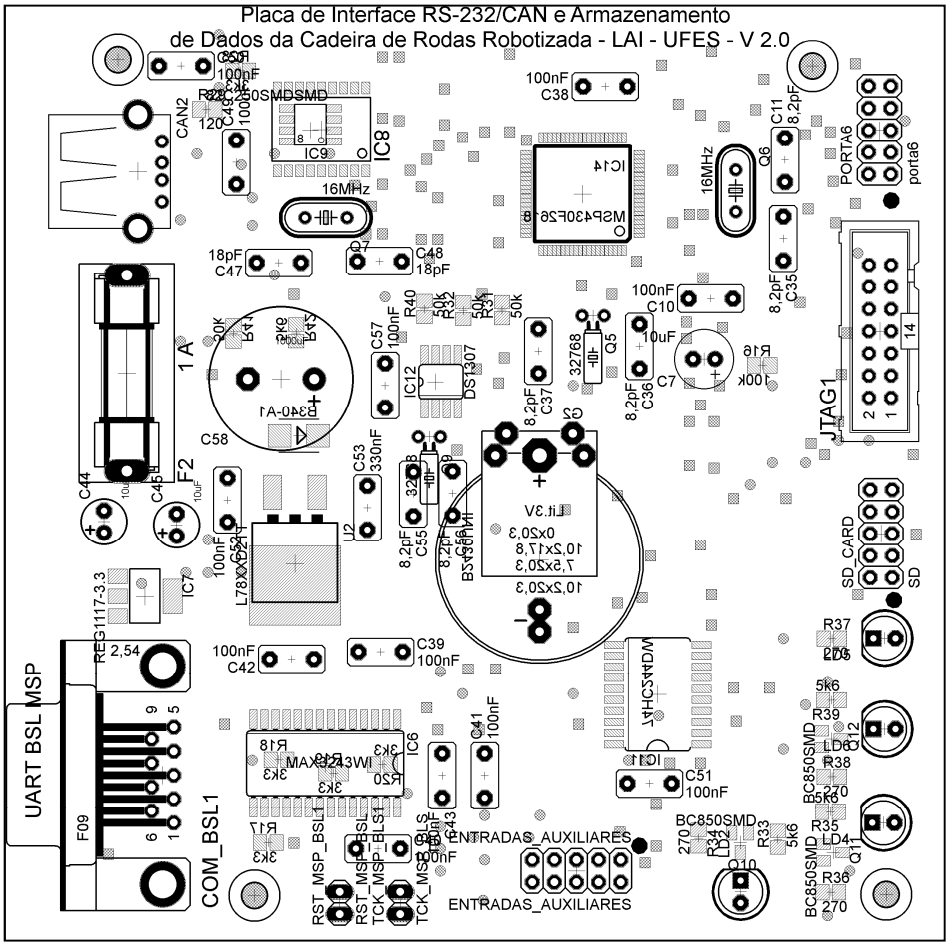


Figura 52: Disposição Física dos componentes na Placa de Processamento.

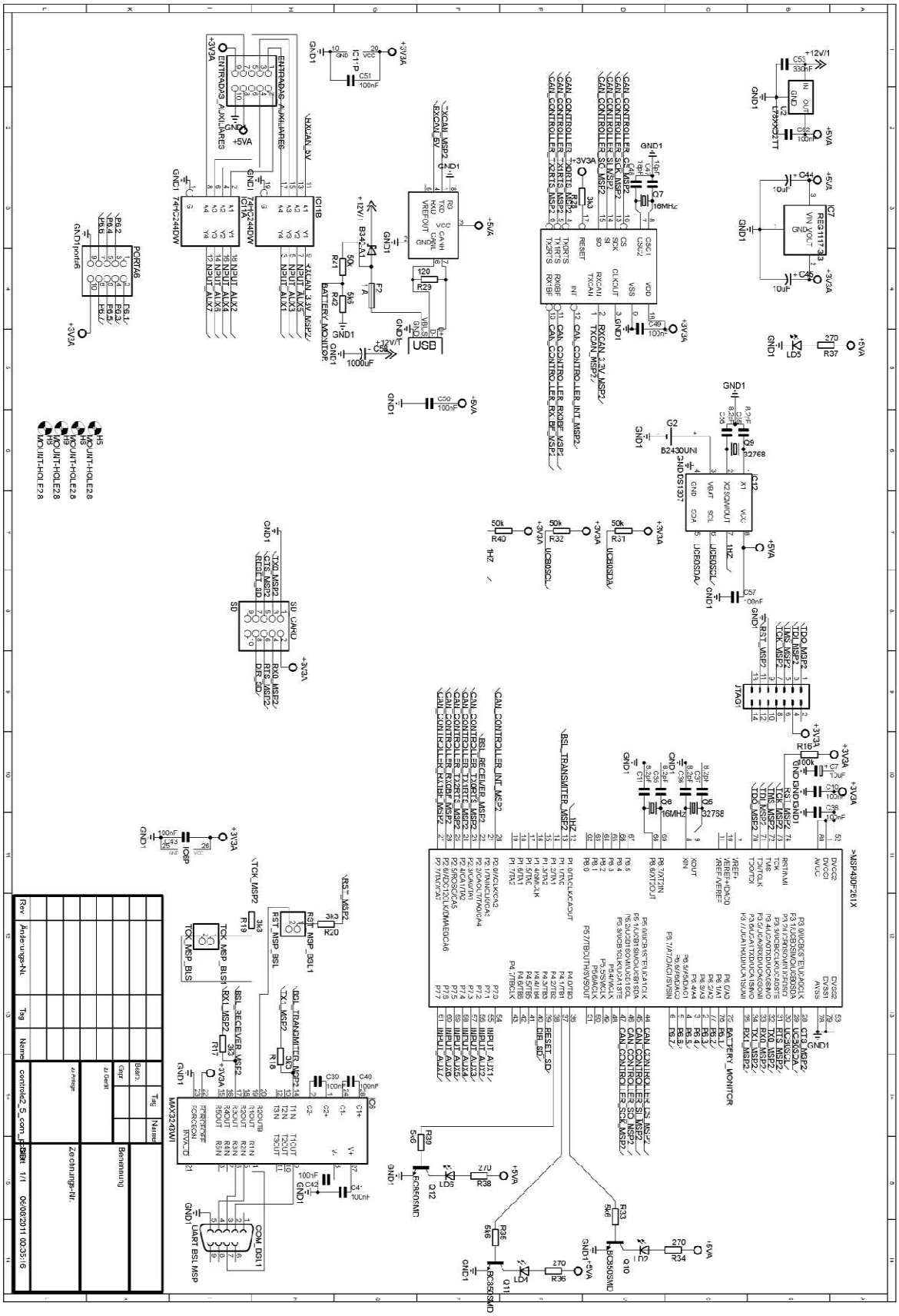


Figura 53: Esquemático final do Projeto da Placa de Processamento.

2.5 Conclusão

A arquitetura de placas dedicadas, sendo uma para processamento e outra para controle dos sensores e periféricos se mostrou eficaz, segura e versátil. Isto porque com a divisão foi possível melhor escalonamento dos processos. O ponto de vista de segurança tal topologia permite que ocorram verificações de funcionamento entre as placas, assim, em caso de falha em uma das placas a outra possui subsídios para detectar e minimizar os efeitos decorrentes de tal falha. A versatilidade desta topologia é proveniente da possibilidade de se adicionar a qualquer momento mais placas para desempenhar outras funções.

3 SENSORIAMENTO E PERIFÉRICOS

Para realizar a função de percepção do ambiente a Cadeira de Rodas Robotizada dispõe de uma série de sensores e periféricos (Figura 54). Neste Capítulo serão abordadas suas rotinas de controle, interfaces, funções e parametrizações.

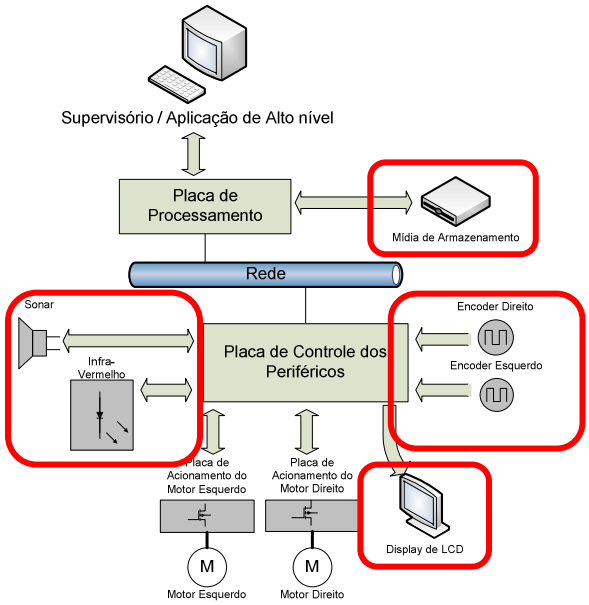


Figura 54: Arquitetura básica da Cadeira de Rodas Robotizada com seus periféricos em destaque.

3.1 Os Encoders e a Rotina de Mensuração de Velocidades

A base da hodometria[11] da cadeira de rodas são os *encoders*. Estes dispositivos são utilizados para medir a velocidade da cadeira de rodas, e por conseqüência estimam sua posição. Em seguida serão detalhados os tipos de *encoders* existentes e suas características.

3.1.1 Tipos de *Encoders* e Suas Características

O *encoder* é um transdutor que converte um movimento angular ou linear em uma série de pulsos digitais elétricos. Esses pulsos gerados podem ser usados para determinar velocidade, taxa de aceleração, distância, rotação, posição ou direção [41].

---

[11] Hodometria: Arte de medir as distâncias percorridas e velocidades de um objeto.

O sistema de leitura é baseado em um disco (*encoder* rotativo), formado por janelas radiais transparentes e opacas alternadas. Este é iluminado perpendicularmente por uma fonte de luz infravermelha, quando então, as imagens das janelas transparentes são projetadas no receptor. O receptor converte essas janelas de luz em pulsos elétricos conforme as ilustrações presentes na Figura 55 e na Figura 56. Os *encoders* podem ser divididos em *encoders* incrementais e absolutos.

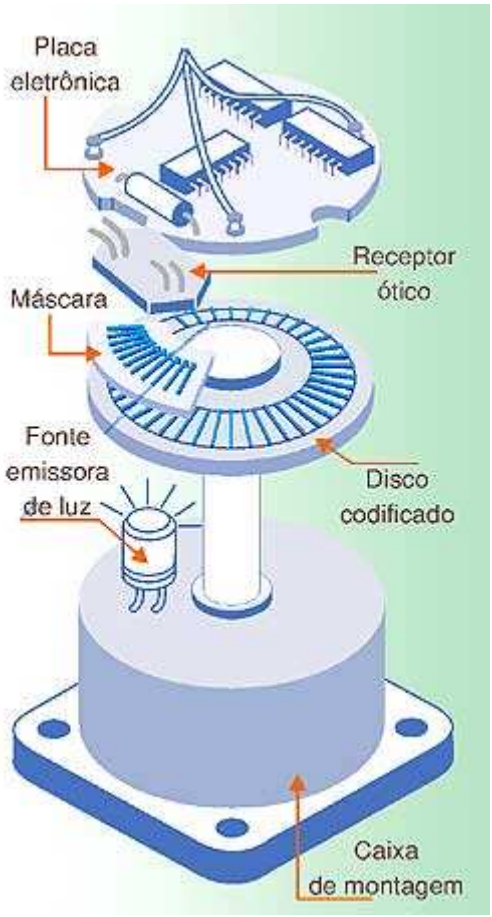


Figura 55:Princípio de funcionamento de um *encoder* rotativo incremental [41].

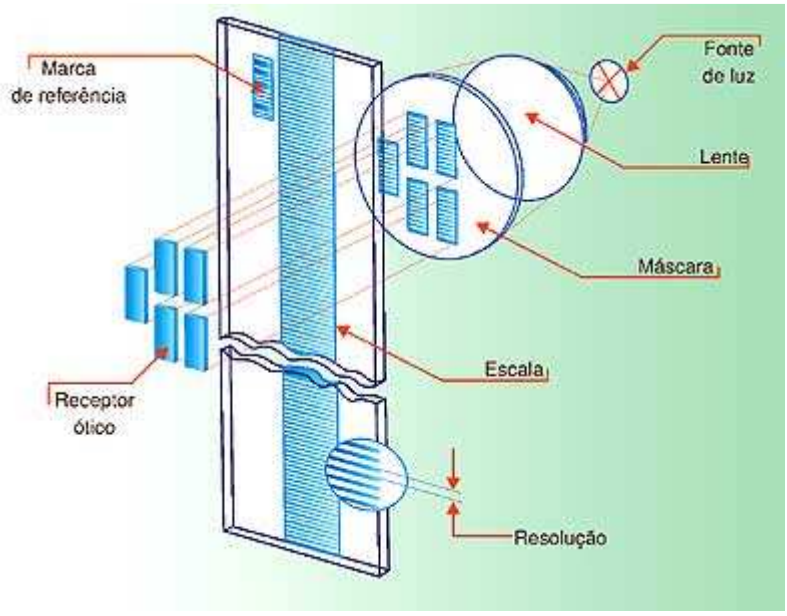


Figura 56:Princípio de funcionamento de um *encoder* absoluto [41].

O *encoder* incremental em quadratura fornece normalmente dois pulsos quadrados defasados em 90º, que são chamados usualmente de canal A e canal B, de acordo com a Figura 57. A leitura de somente um canal fornece apenas a velocidade, enquanto que a leitura dos dois canais fornece também o sentido do movimento [41].

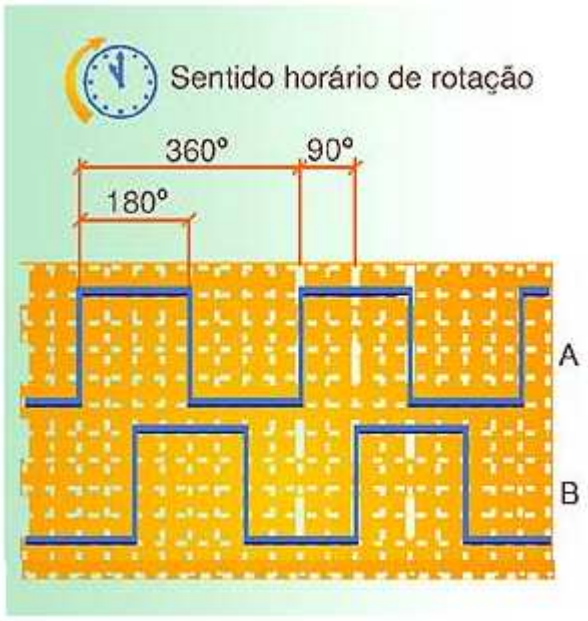


Figura 57: Representação gráfica dos sinais A e B de um *encoder* incremental [41].

Os *encoders* absolutos são capazes de fornecer um código binário contendo a informação de posição angular na qual o mesmo se encontra, ou seja, media com relativa precisão o ângulo que seu eixo se encontra [41].

3.1.2 Os *Encoders* Usados na Cadeira de Rodas

Os *encoders* utilizados neste trabalho são do tipo incremental (Figura 58), cujo fabricante é a *Computer Optical* e o modelo é o CP-350 *Incremental, Digital* [24]. Suas principais características são:

- Dois canais de saída digitais em quadratura;
- Níveis de tensão TTL;
- 50 mA de consumo;
- Incremental;
- 100 pulsos por revolução;

Os *encoders* estão acoplados aos eixos dos motores, sendo um para cada motor. Sabendo que os motores estão ligados às rodas por um sistema de polias e correias, e ainda desprezando qualquer tipo de folga, escorregamentos e distensão no sistema de polias e correias, é possível afirmar que os períodos dos sinais de cada *encoder* são proporcionais ao período de sua respectiva roda.

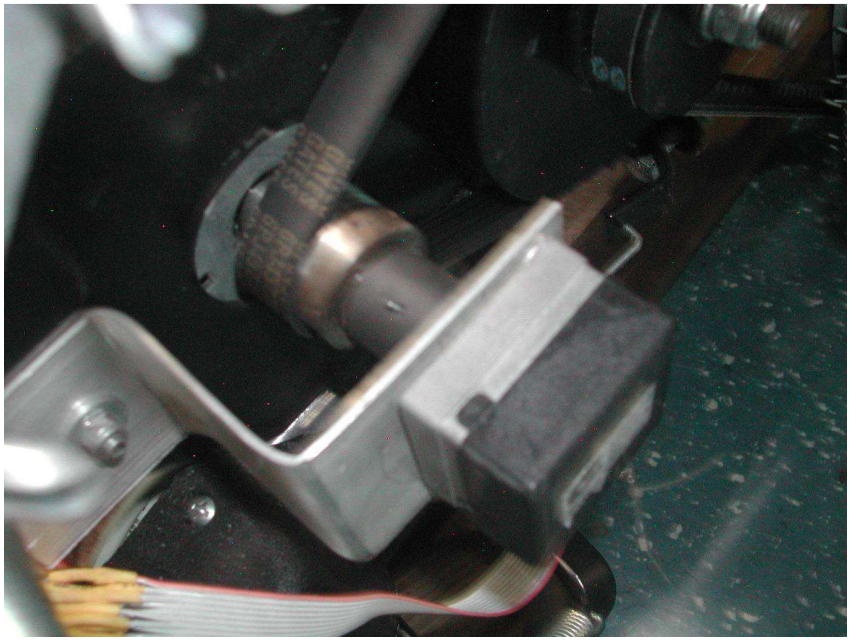


Figura 58: *Encoder* da roda direita da cadeira de rodas motorizada.

3.1.3 O Processo de Determinação das Velocidades das Rodas da Cadeira de Rodas

O processo de leitura de velocidade das rodas começa com a captura dos períodos dos sinais gerados pelos *encoders*. Estes períodos são capturado pelo microcontrolador usando uma de suas interfaces de captura de período a CCP (*capture/compare blocks*) *timer* B canais 1,2,3 e 4. A Figura 27 ilustra as ligações físicas entre os *encoders* e o microcontrolador.

O processo de captura do período dos *encoders* começa com uma transição positiva[12] nos canais A dos *encoders*. Neste trabalho usou-se apenas o canal A de cada um dos *encoders* para medir sua velocidade. Os canais B foram usados apenas como referência de sentido. O motivo desta opção de projeto é que em testes o uso dos dois canais dos dois *encoders* para medir seus períodos, e conseqüentes velocidades, demandou recurso computacional excessivo. O evento de transição positiva em um dos canais dispara uma interrupção no

[12] Transição Positiva: transição de estado lógico 0 para estado lógico 1 de um sinal digital.

microcontrolador da PCP que, ao atender a interrupção, apenas mede o período do sinal do canal A e usa o canal B para determinar o sentido de rotação. Estas informações são dispostas em uma estrutura de variáveis globais que posteriormente serão processadas para determinar as velocidades de cada roda. A rotina de tratamento da interrupção de captura dos períodos dos *encoders* está descrita no anexo 5.

O processo de determinação das velocidades das rodas da cadeira de rodas tem como base a relação básica de velocidade:

$$V = \frac{dD}{dt}$$

Equação 1

Onde D é o espaço percorrido, V é a velocidade e t é o tempo.

Como mencionado anteriormente, os *encoders* proveem 100 pulsos por revolução e, como eles estão acoplados nos eixos dos motores da cadeira de rodas, cada revolução dos mesmos é codificados por 100 pulsos. As rodas da cadeira de rodas são conectadas aos motores através de dois conjuntos de polias e correias, portanto, desprezando-se os deslizamentos pode-se afirmar que para cada rotação das rodas são dadas K_{Acop} rotações nos motores. Como a distância percorrida por cada roda, D_{Roda}, pode ser representada em função do número de rotações de desta, N_{Roda}, e pelo seu raio, R_{Roda}, então:

$$D_{Roda} = 2\pi \times N_{Roda} \times R_{Roda}$$

Equação 2

Sendo,

$$N_{Roda} = K_{Acop} \times N_{Motor} => D_{Roda} = 2\pi \times R_{Roda} \times K_{Acop} \times N_{Motor}$$

Onde N_{Motor} é o número de rotações do motor de cada roda e K_{Acop} é a relação de redução de todo sistema de acoplamento (polias e correias).

Como os *encoders* têm 100 pulsos por revolução a D_{Roda} pode ser dada em função do número de pulso n_{pulsos}:

$$D_{Roda} = 2\pi \times R_{Roda} \times K_{Acop} \times 100 \times n_{pulsos} =>$$

$$V_{Roda} = \frac{dD_{Roda}}{dt} = 2\pi \times 100 \times R_{Roda} \times K_{Acop} \times \frac{\mathrm{d}n_{pulsos}}{\mathrm{dt}}$$

Onde $\frac{\mathrm{d}n_{pulsos}}{\mathrm{dt}}$ é a frequência do sinal dos *encoders*. Logo:

$$V_{Roda} = \frac{2\pi \times 100 \times R_{Roda} \times K_{Acop}}{T_{Pulsos}}$$

$2\pi \times 100 \times R_{Roda} \times K_{Acop}$ pode ser considerada como constante de calibração dos *encoders*, K_{cal}. Esta constante foi determinada experimentalmente com o uso de um tacômetro e seu valor é $K_{cal} = 3840$. Assim, a velocidade de cada uma das rodas da cadeira de rodas pode ser expressa como:

$$V_{Roda} = \frac{K_{cal}}{T_{Pulsos}} = \frac{3840}{T_{Pulsos}}\ m/s$$

Equação 3

Este método de mensuração de velocidade é extremamente dependente das tolerâncias construtivas dos *encoders*, assim, para sua eficácia, os *encoders* devem prover uma largura física entre os pulsos com oscilação adequada. No caso dos CP-350 esta oscilação é de ± 2,5%, sendo adequado para a aplicação.

De acordo com a Equação 3 pode-se medir as velocidades das duas rodas separadamente, sendo a velocidade da roda direita chamada V_d e a velocidade da roda esquerda chamada de V_e.

Para hodometria da cadeira de rodas outras duas velocidades são importantes: a velocidade linear V_l e a velocidade angular ω. A Figura 59 ilustra as relações entre as velocidades presentes na cadeira de rodas.

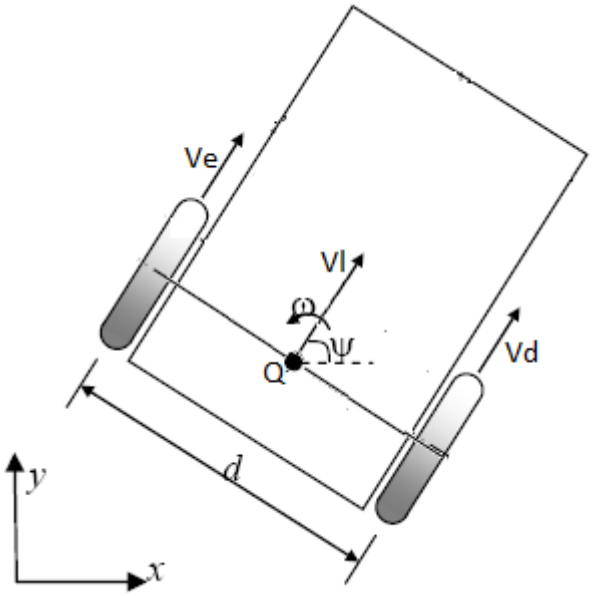


Figura 59: Representação das velocidades atuantes na cadeira de rodas.

De acordo com [42] as velocidades linear e angular da cadeira de rodas podem ser expressas em função das velocidades das rodas, assim:

$$\begin{bmatrix} V_i \\ \omega \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{d} & -\frac{1}{d} \end{bmatrix} \begin{bmatrix} V_d \\ V_e \end{bmatrix}$$

Equação 4

Onde a distância entre as rodas “d” para a cadeira de rodas é de 600 mm.

O posicionamento espacial da cadeira de rodas é composto por suas coordenadas nos eixos x e y e por sua orientação, representada na Figura 59 por Ψ. De acordo com [42] as relações entre as velocidades da cadeira e sua posição espacial pode ser dada por:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\Psi} \end{bmatrix} = \begin{bmatrix} V_l \cos \Psi \\ V_l \operatorname{sen} \Psi \\ \omega \end{bmatrix}$$

Equação 5

Neste trabalho é assumido que as coordenadas x, y e Ψ são referentes ao ponto médio entre os centros das duas rodas, representado pela letra Q na Figura 59.

Reescrevendo a Equação 5 se tem:

$$x = \int V_l \cos \Psi \, dt + \, x_0$$

Equação 6

$$y = \int V_l \operatorname{sen} \Psi \, dt + \, y_0$$

Equação 7

$$\Psi = \int \omega \, dt + \, \Psi_0$$

Equação 8

Assumindo-se o uso de métodos numéricos para operação de integração, e assumindo-se que todas as mensurações de velocidades são efetuadas de maneira discreta, é possível mensurar V_i e ω a cada período de amostragem, sendo: $(V_i{}^k, \omega^k)$, $(V_i{}^{k-1}, \omega^{k-1})$, $(V_i{}^{k-2}, \omega^{k-2})$, $(V_i{}^{k-3}, \omega^{k-3})$, $(V_i{}^{k-4}, \omega^{k-4})$ ... valores de velocidades linear e angular, respectivamente, nos instantes de tempo: k, k-1, k-2 ...

Portanto, é possível reescrever a Equação 6, Equação 7 e Equação 8 como:

$$x = T\left(\sum_{k=0}^{n} \frac{V_{i_k} + V_{i_{k-1}}}{2} \cos\left(\frac{\Psi_k + \Psi_{k-1}}{2}\right)\right)$$

Equação 9

$$y = T\left(\sum_{k=0}^{n} \frac{V_{i_k} + V_{i_{k-1}}}{2} \operatorname{sen}\left(\frac{\Psi_k + \Psi_{k-1}}{2}\right)\right)$$

Equação 10

$$\Psi = T\left(\sum_{k=0}^{n} \omega_k\right)$$

Equação 11

Para realizar os cálculos de hodometria da cadeira de rodas são usadas as Equação 9, Equação 10 e Equação 11, sendo T o período no qual a rotina de execução da hodometria, a *TaskControleDosEncoders*, é executada, que neste trabalho é de 5 ms.

Contudo, ao incluir as operações de posicionamento e orientação na tarefa *TaskControleDosEncoders* a ocupação do processamento do microcontrolador ficou elevada, levando a perda de tempo real em alguns momentos. Isto se deve ao fato do uso de funções trigonométricas e variáveis do tipo *float.* Diante do exposto optou-se por não efetuar tais cálculos na estrutura montada, contudo, pela arquitetura de variáveis de rede, e sabendo que as velocidades de ambas as rodas estão disponíveis nesta, é possível efetuar os cálculos de posicionamento e orientação da cadeira de rodas em futuras ampliações usando-se outro processador ligado a rede CAN.

O código C da tarefa de hodometria está descrito no anexo 9 e seu arquivo *header* no anexo 8. Outras considerações sobre a tarefa de hodometria são suas variáveis de saída. Estas variáveis são: velocidade da roda direita (V_d) e velocidade da roda esquerda (V_e).

Estas variáveis estão disponibilizadas no barramento CAN e seus parâmetros de rede estão descritos na Tabela 14. Outra informação relevante sobre estas variáveis são seus tipos, sendo que: V_d e V_e são do tipo *long int* de 32 *bits* e sua dimensão é mm/s.

A tarefa de hodometria foi desenvolvida seguindo a metodologia *State Chart* [43], e tem como funcionalidade o atendimento a comandos externos, leitura dos *encoders* e *infravermelho* e processamento da Equação 3 para cada roda. Ela dispõe de um *byte* de comando e um *byte* de *status*, ambos variáveis de rede. O *byte* de comando indica à tarefa qual procedimento deve ser executado, e o *byte* de *status* indica para as demais tarefas da rede se a rotina de hodometria sofreu uma falha de tempo real ou não.

Seus comandos podem ser:

- *Run*, executa as funções de leitura dos *encoders* e infravermelho e processamento da Equação 3 para cada roda;
- *Limpa Falhas*, apaga o *bit* indicador de perda de tempo real no *byte* de *status* da tarefa de hodometria;

A tarefa que utilizar as informações oriundas da rotina de hodometria deve antes verificar se o *bit* indicador de perda de tempo real está ativo, ou seja, quando ocorre uma falha de tempo real seu *bit* correspondente no *byte* de *status* assume o valor lógico 1, e quando nenhuma falha e tempo real foi detectada o *bit* assume o valor lógico 0.

3.2 O Sonar

Este trabalho visa prover uma infraestrutura de suporte de sensoriamento para a cadeira de rodas. Contudo, não é escopo deste trabalho o estudo de posicionamento dos sensores e a colocação dos mesmos, visto que para a inserção de qualquer sensor deve ser acompanhado de um estudo de posicionamento, área de abrangência, possíveis falhas e pontos cegos dos sensores. Esta decisão é fundamentada no fato que os posicionamentos e os algoritmos de controle destes estão diretamente ligados à aplicação específica que será dada a cadeira de rodas.

Neste trabalho foi disponibilizada uma estrutura de *hardware* preparada para receber o sonar desenvolvido em [27] (Figura 60) para um robô móvel. Tal sonar necessita de uma estrutura de temporização, multiplexação e *bits* de controle. Para a multiplexação e *bits* de controle foram disponibilizados pinos de I/O do microcontrolador, e para a temporização foram disponibilizados dois dos seis canais de CCP do *timer B* [21] do microcontrolador. A interface física está detalhada no Capítulo 2.

O sonar desenvolvido em [27] usou como base o modulo (*6500 Series Sonar Range Module*) fabricado pela Polaroid. A série 6500 é um módulo ultra-sônico de medição de distâncias que pode acionar todos os tipos de transdutores eletrostáticos da Polaroid com uma simples interface adicional [5]. Este módulo é capaz de medir distâncias entre 15 cm e 10m. A precisão sobre toda a faixa de medidas é de ± 1% do valor medido.

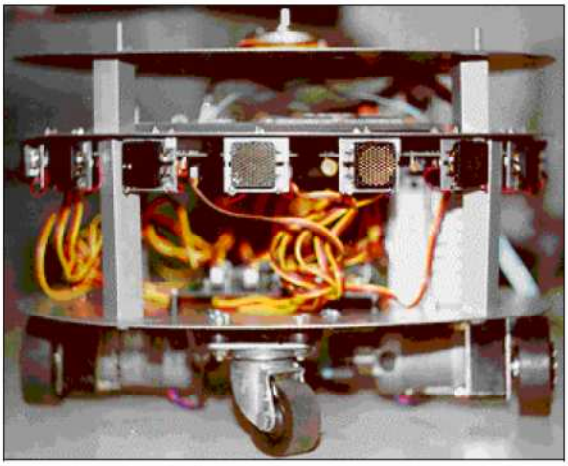


Figura 60: Robô Móvel “Brutus” desenvolvido no LAI/PPGEE.

Este módulo tem uma entrada denominada *blanking* que permite a exclusão seletiva de ecos para operação em um modo multi-eco (detecção de múltiplos ecos). O módulo é capaz de diferenciar ecos de objetos que distem ao menos 7,5cm um do outro. O amplificador com largura de faixa variável e controle digital de ganho (Figura 61 e Figura 62 - componente U1 - SN28784), minimiza os efeitos da atenuação do sinal acústico, o ruído e a detecção dos lóbulos laterais (secundários) dos transdutores ultra-sônicos [27][5].

O módulo série 6500 possui um ressonador cerâmico controlado para gerar uma base de tempo de 420 kHz muito precisa. Uma saída baseada nestes 420 kHz é disponível para uso externo. A excitação do transdutor é realizada com um trem de 16 pulsos cuja frequência é de 49.4 kHz. O módulo opera com uma tensão de alimentação que varia na faixa de 4.5V a 6.8V, e é caracterizado para operar sob uma faixa de temperatura que vai de 0 ºC a 40 ºC. Todas estas informações estão contidas nas referências [27][5].

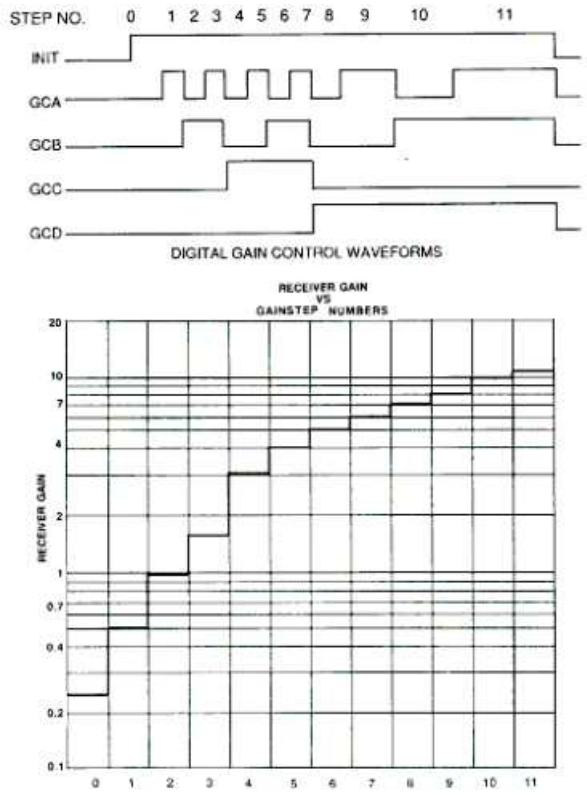


Figura 61:Variação do ganho do SN28784 com o tempo [5].

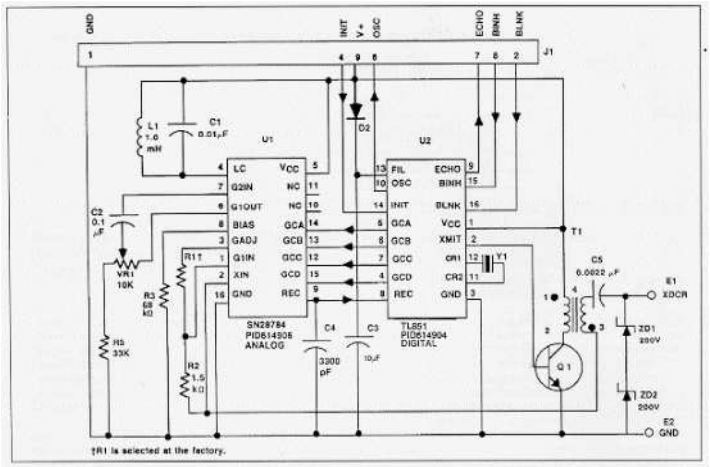


Figura 62: Esquema elétrico da placa 6500 Series Sonar Range Module da Polaroid [5].

A PCP possui interface implementada com a placa de controle dos módulos Polaroid série 6500 desenvolvida em [27], sendo esta placa um sistema de controle e multiplexação de sinais de eco oriundos dos 16 módulos Polaroid desenvolvidos em [27].

3.3 O Sistema de Detecção de Obstáculos Baseado em Sensores Infravermelho

Para sensoriamento mais preciso e mensuração de distâncias pequenas foram disponibilizadas interfaces para os sensores de infravermelho GP2D02 da Sharp [23]. A infraestrutura física está detalhada no Capítulo 2. Este trabalho se restringe ao desenvolvimento da infraestrutura física e algoritmo de comunicação entre microcontrolador e GP2D02. Novamente, a implementação do posicionamento e algoritmos de navegação não são escopo deste trabalho.

O GP2D02 possui protocolo de comunicação próprio. Este protocolo se caracteriza por ser serial, síncrono e do tipo mestre-escravo, sendo o microcontrolador mestre e o GP2D02 escravo. Neste protocolo o microcontrolador envia um sinal de *clock* e o GP2D02 responde enviando uma sequência de dados. A forma de onda do sinal de *clock* para o GP2D02 está ilustrada na Figura 63.

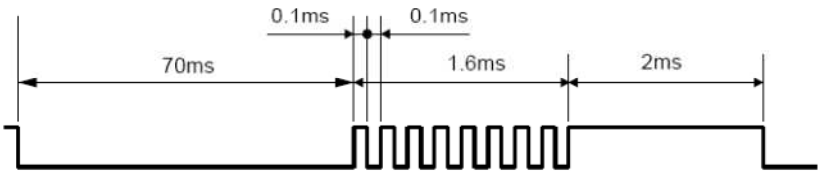


Figura 63: Forma de onda do *clock* do GP2D02.

Para a geração do *clock* ilustrado acima foi usada uma interrupção de tempo do *Timer B* [21]. O algoritmo de geração deste sinal está descrita no anexo 5.

Outro fato relevante sobre o GP2D02 é o formato dos dados de saída, que são codificados por um valor de 8 *bits*, relacionado com a distância medida entre o sensor e o objeto refletor pelo gráfico ilustrado na Figura 64.

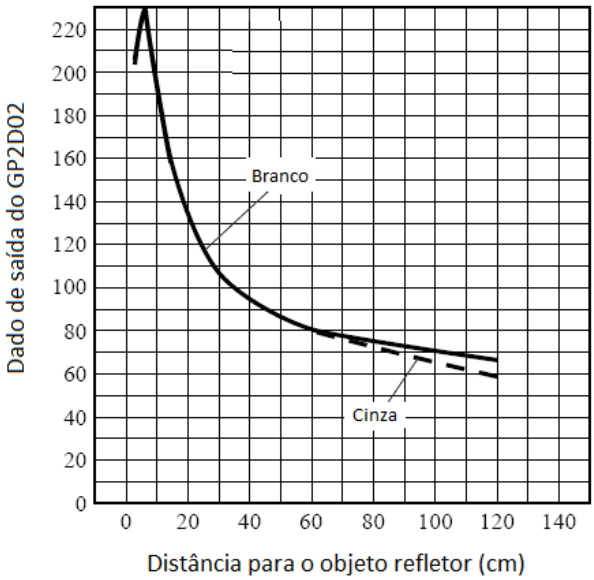


Figura 64: Gráfico de relação entre o valor de saída do GP2D02 com a distância medida.

A interpolação da curva descrita pela Figura 64 é representada pela equação:

$$D = \frac{1580}{L + 0{,}5} + C$$

Equação 12

Onde: D é o valor de saída do sensor, L é a distância medida e C é uma constante de calibração, sendo esta determinada via experimentos práticos com o valor $C = 69{,}4$.

Foram disponibilizadas quatro interfaces para os sensores infravermelhos, conforme Figura 36, sendo que todos compartilham o mesmo sinal de *clock*. Após a recepção dos dados pela rotina de temporização do sinal de *clock*, um *bit* indicador de novo dado infravermelho é setado. A rotina de hodometria é também responsável pela execução da Equação 12 a partir dos dados recebidos pela rotina de temporização do *clock* infravermelho. Assim, quando o *bit* de novo dado infravermelho é setado, a rotina de hodometria executa a Equação 12 para os quatro sensores infravermelhos, e insere os resultados em quatro variáveis de barramento. Estas variáveis são: *IR.Distancia0, IR.Distancia1, IR.Distancia2 e IR.Distancia3*, sendo todas do tipo *unsigned int* de 16 *bits* e suas dimensões são expressas em *mm*. Os parâmetros destas variáveis de barramento estão descritos na Tabela 14.

3.4 A Interface Com Cartão Micro-SD

Objetivando armazenamento de variáveis de controle importantes da cadeira de rodas, e futuras aplicações, tais como: navegação em ambiente mapeado, reconhecimento de ambiente, aquisição e armazenamento dos sinais vitais do usuário, foi disponibilizada uma infraestrutura de armazenamento de dados composta de um cartão de memória micro-SD e uma tarefa de controle e inserção dos dados no cartão.

A gestão e controle do cartão micro-SD é de responsabilidade de um CI dedicado, o CD17B10, cujo fabricante é a DOSonCHIP, sendo esta uma interface dedicada para cartões de memória.

O CD17B10 DOSonCHIP fornece uma interface para cartões de memória removíveis indústriais, incluindo micro-SDHC, micro-SD, mini-SDHC, mini-SD, SDHC, SD, MMC e outros cartões. Ele implementa tanto a interface física quanto a interface de sistema de arquivamento. O CD17B10 disponibiliza uma série de comandos simples para manipulação dos arquivos e dados presentes no cartão de memória, podendo estes ser enviados para o CD17B10 sobre o protocolo UART, SPI, ou I2C. Estes comandos são usados para navegar nos arquivos ou diretórios, leitura ou escrita de dados no cartão de memória. O formato padrão de arquivamento de arquivos é o FAT, que é diretamente compatível com PC, mídia digital players, câmeras digitais e etc [38].

Neste trabalho foi usada uma implementação comercial do CD17B10 fabricada pela *Sparkfun Eletronics* (Figura 46). Este dispositivo é composto do próprio CD17B10, um *socket* para cartão micro-SD e um cristal de 32768 Hz para o RTC interno do CD17B10.

A comunicação entre CD17B10 e microcontrolador escolhida foi a UART operando em 19,2 kbps. Foi então utilizada a USCI_A0 do microcontrolador da Placa de Processamento, sendo esta configurada para a taxa referida e 1 *bit* de *stop*, 8 *bits* de dados e 1 *bit* de *start*. O CD17B10 possui um mecanismo de *auto baud rate* para sua interface UART, ou seja, detecção automática de taxa de sinalização, de modo que para sincronizar o CD17B10 com o microcontrolador, este deve enviar um pacote padrão composto de dois *bytes* CR (0x0D) para o CD17B10. Após este procedimento o CD17B10 reconhece a taxa de

sinalização e está pronto para operar. A interface física entre o CD17B10 e o microcontrolador está descrita no Capítulo 2.

3.4.1 O Protocolo de Comunicação do DOSonCHIP CD17B10

Como mencionado anteriormente o CD17B10 possui uma série de comandos para sua operação. Os comandos se dividem em comandos simples e complexos. Todos os comandos simples consistem de um único *byte* identificador seguido de quatro *bytes* de argumento do comando. A resposta a um comando básico consiste de um *byte* indicador de resposta, podendo indicar sucesso ou insucesso, seguido de um argumento de resposta de quatro *bytes* (Figura 65). As respostas bem sucedidas possuem o *byte* indicador DOS_RES_NOERROR seguido por um argumento de quatro *bytes* que é específico para o comando. As respostas sem êxito retornam um código de erro seguido por um valor de erro de quatro *bytes*, que podem ser úteis para depuração [38].

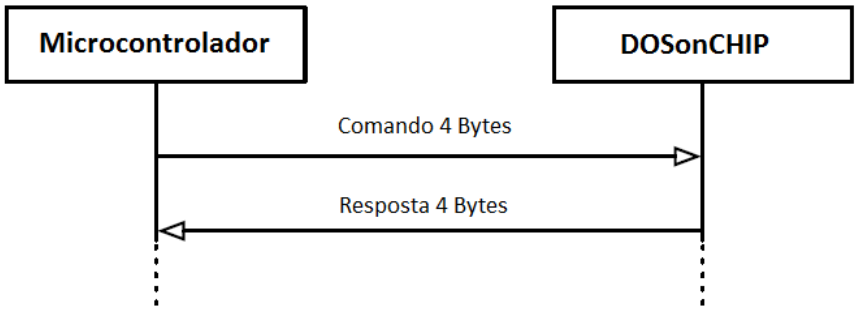


Figura 65: Ilustração do protocolo de comunicação de comando simples DOSonCHIP – microcontrolador.

Tabela 5: Comandos Simples do DOSonCHIP [38]

Comando	Descrição do Comando	Argumento do Comando	Argumento de Resposta
DOS_DOS_CMD_GET_ID	Retorna a versão do do CI CD17B10	Ignorado	*Bytes* 3-1: ignorado *Byte* 0: versão do DOSonCHIP
DOS_CMD_GET_VERSION	Retorna a versão do *firmware* do CI CD17B10	Ignorado	*Byte* 3: DOSonCHIP boot*load*er MSB *Byte* 2: DOSonCHIP boot*load*er LSB *Byte* 1: DOSonCHIP firmware MSB

			Byte 0: DOSonCHIP firmware LSB
DOS_CMD_GET_TIME	Retorna a data/hora atual do IC DOSonCHIP. Só tem sentido quando a data/hora foi definido e o RTC foi ativado (através do comando DOS_CMD_SET_TIME_ON_OFF).	Ignorado	*Bytes* 3-0: Data/Hora Atuais.
DOS_CMD_SET_TIME	Define a data/hora atual do IC DOSonCHIP. Só tem sentido quando a data/hora foi definido e o RTC foi ativado (através do comando DOS_CMD_SET_TIME_ON_OFF).	*Bytes* 3-0: Data/Hora Atuais.	Ignorado
DOS_CMD_SET_TIME_ON_OFF	Habilita o RTC do DOSonCHIP	*Bytes* 3-1: Ignorados *Byte* 0: DOS_ON ou DOS_OFF	Ignorado
DOS_CMD_MOUNT	Monta o Cartão, sendo seu argumento preferencial deve ser 0xFFFFFFFF	*Bytes* 3-0: Número de setores livres a serem encotrados para modo de escrita	*Bytes* 3-0: indentificador único do cartão.
DOS_CMD_GET_FREE_SECTORS	Retorna o número de setores livres encontrados do cartão. Cada sertor tem 512 *bytes*	Ignorado	*Bytes* 3-0: número de setores livres.
DOS_CMD_DIR	Inicializa ou incrementa o diretório	*Bytes* 3-1: Ignorado *Byte* 0: DOS_FIRST ou DOS_NEXT	*Bytes* 3-1: Ignorado *Byte* 0: atributo do arquivo atual.
DOS_CMD_DIR_GET_PROPERTY	Pesquisa as propriedades do diretório ou arquivo atual	*Bytes* 3-1: Ignorado *Byte* 0: DOS_PROPERTY_SIZE, DOS_PROPERTY_TIME_CREATED ou DOS_PROPERTY_TIME_MODIFIED	Para DOS_PROPERTY_SIZE: tamanho em *bytes* do arquivo, sendo 0 para diretório. Para DOS_PROPERTY_TIME_CREATED e DOS_PROPERTY_TIME_MODIFIED: retorna a data de criação ou modificação do diretório.
DOS_CMD_MAKE_DIR	Cria um novo diretório como o nome especificado pelo *buffer name.*	Ignorado	Ignorado
DOS_CMD_SET_DIR	Abre o diretório especificado pelo *buffer name.*	Ignorado	Ignorado

DOS_CMD_DELETE	Deleta o arquivo especificado por *buffer name*. A exclusão de diretórios não é suportada.	Ignorado	Ignorado
DOS_CMD_SET_HANDLE	Define um *handle* para o arquivo atual	*Bytes* 3-1: valor do novo *handle*(0-3)	Ignorado
DOS_CMD_OPEN_READ	Abre o arquivo especificado por *buffer name* no modo de leitura.	*Bytes* 3-0: 0	*Bytes* 3-0: tamanho do arquivo em *bytes*.
DOS_CMD_OPEN_WRITE	Abre o arquivo especificado por *buffer name* no modo de escrita.	*Bytes* 3-0: 0	*Bytes* 3-0: tamanho do arquivo em *bytes*.
DOS_CMD_SEEK	Incrementa o ponteiro de escrita e leitura o arquivo em uma quantidade de *bytes*	*Bytes* 3-0: número de *bytes* de increment. 0 para inicio do arquivo	Ignorado
DOS_CMD_WRITE_PREALLOCATE	Aloca espaço em *bytes* no arquivo aberto no modo de escrita.	*Bytes* 3-0: número de *bytes* para alocar.	*Bytes* 3-0: novo tamanho do arquivo em *bytes*.
DOS_CMD_CLOSE	Fecha o arquivo especificado pelo *handle* atual	Ignorado	Ignorado

Os comando complexos do DOSonCHIP têm seu próprio protocolo de comunicação. Estes comandos são para escrita de dados, leitura de dados e leitura e escrita do *buffer name*, sendo descritos em seguida.

Comando DOS_CMD_SET_NAME: Define o valor do *buffer name* atual. Antes de enviar o nome, o microcontrolador deve informar ao DOSonCHIP o comprimento do nome que será enviado, sendo o máximo de 12 *bytes* para cada nome (Figura 66).

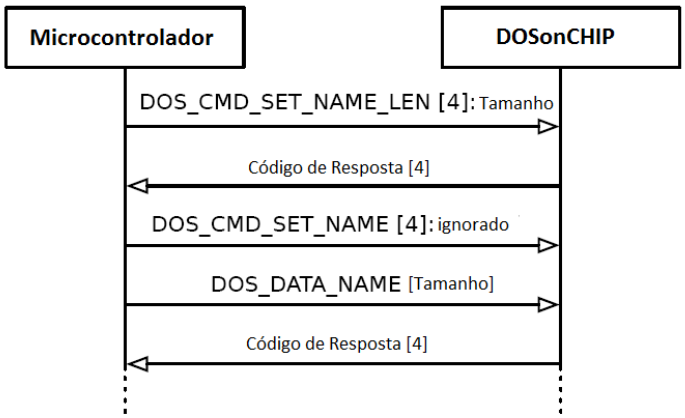


Figura 66: Protocolo de comunicação do comando DOS_CMD_SET_NAME DOSonCHIP.

Comando DOS_CMD_GET_NAME: Recupera o valor do *buffer name* atual. Antes de receber o nome, o DOSonCHIP informa ao microcontrolador o comprimento do nome que será o enviado, sendo o máximo de 12 *bytes* para cada nome.

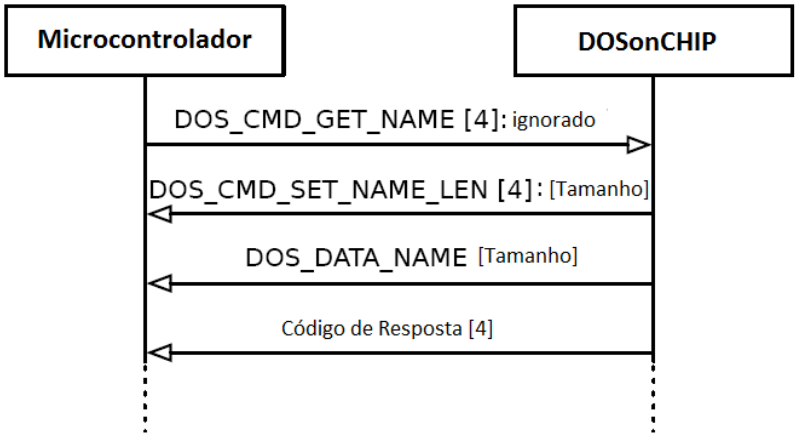


Figura 67: Protocolo de comunicação do comando DOS_CMD_GET_NAME DOSonCHIP.

Comando DOS_CMD_READ: Lê dados de um arquivo aberto para leitura (indicado pelo *handle* atual). O DOSonCHIP envia dados em pacotes com tamanho variável. Antes de enviar o primeiro bloco, o DOSonCHIP informa ao microcontrolador o tamanho do pacote que será o enviado. Sempre que o tamanho do pacote mudar, o DOSonCHIP vai informar ao microcontrolador o novo tamanho.

Para receber um bloco, o microcontrolador deve enviar o comando DOS_HANDSHAKE_PAK_NEXT. Em resposta, o DOSonCHIP envia o comando DOS_DATA_BLOCK seguido do bloco de dados. Ao final da transmissão de dados o DOSonCHIP envia uma resposta DOS_DATA_BLOCK_END. Se qualquer outra resposta é recebida indicará um erro durante a leitura.

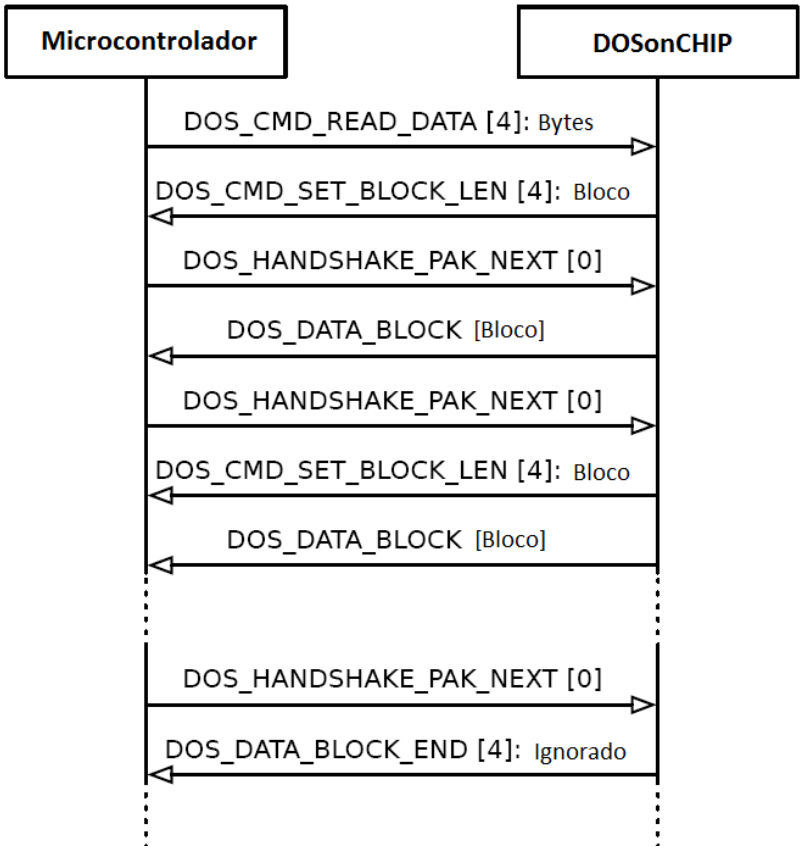


Figura 68: Protocolo de comunicação do comando DOS_CMD_READ DOSonCHIP.

Comando DOS_CMD_WRITE: Grava dados em um arquivo aberto no modo de escrita (indicado pelo *handle* atual). O microcontrolador envia os dados ao DOSonCHIP em uma série de pacotes. Antes que qualquer dado seja escrito, o DOSonCHIP informa ao microcontrolador o tamanho do pacote de dados que é esperado. Se houver mudanças no tamanho do pacote de dados o DOSonCHIP informa o microcontrolador o novo tamanho do pacote.

Antes de enviar os dados para o DOSonCHIP, o microcontrolador consulta a sua disponibilidade através de uma série de comandos *handshake*. Se o microcontrolador recebe o comando DOS_HANDSHAKE_GO, ele envia um bloco de dados usando o tamanho do bloco anteriormente informado pelo DOSonCHIP. Se o microcontrolador recebe o comando DOS_CMD_SET_BLOCK_LEN, deve ler quatro *bytes* do argumento de resposta e ajustar o comprimento do bloco atual antes de enviá-lo (Figura 69).

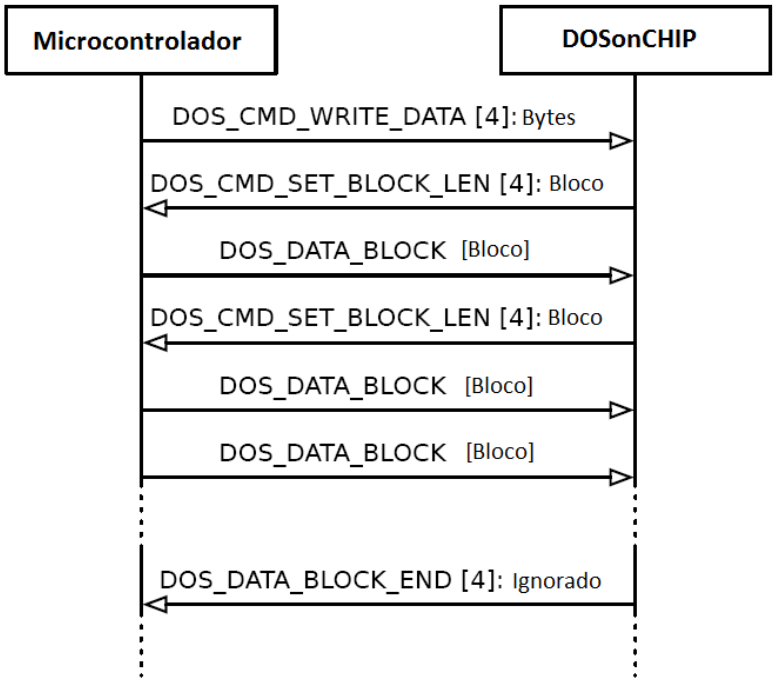


Figura 69: Protocolo de comunicação do comando DOS_CMD_WRITE DOSonCHIP.

Todos os comandos, resposta e erros do DOSonCHIP estão detalhados no anexo 9 e as funções de comando implementadas neste trabalho estão descritas na linguagem de programação C no anexo 10.

3.4.2 As Tarefas de Controle Arquivamento de Dados

Objetivando o armazenamento de dados de maneira periódica e contínua foram criadas duas tarefas que coletam os dados e os armazenam no micro-SD através do DOSonCHIP.

Estas duas tarefas são a *SDControl* e a *Amostrador*, que são responsáveis, repectivamente, por inserir os dados no cartão de memória através do DOSonCHIP e por realizar a amostragem periódica dos dados.

A tarefa *Amostrador* realiza a amostragem dos dados monitorados e as armazena em um *buffer* previamente disponibilizado pela *SDControl*, sendo que esta possui dois *buffers* de armazenamento, nos quais um é disponibilizado para a amostragem enquanto o outro é enviado para o DOSonCHIP. Assim, a *SDControl*, ao terminar o envio de dados para o

DOSonCHIP, verifica se o outro *buffer* está preenchido. Se sim, ela disponibiliza para a tarefa *Amostrador* o *buffer* que foi recém enviado para o DOSonCHIP e começa a enviar o *buffer* preenchido. Esta arquitetura de dois *buffers* permite que as duas tarefas sejam executadas de modo independente um da outra. Obviamente, esta independência é limitada, visto que se a *SDControl* por qualquer razão parar sua execução a tarefa *Amostrador* não pode continuar sua execução.

A tarefa *Amostrador* possui arquitetura muito simples, sendo que sua funcionalidade é amostrar os dados a cada 50 ms, tempo este definido como período de amostragem.

As variáveis armazenadas pela tarefa *Amostrador* são:

- PWM do motor da roda direita em valor de -1000 a 1000 proporcionais a -100 % e 100 % respectivamente;
- PWM do motor da roda esquerda em valor de -1000 a 1000 proporcionais a -100 % e 100 % respectivamente;
- Velocidade da roda direita em mm/s;
- Velocidade da roda esquerda em mm/s;
- *Set-point* de velocidade linear em mm/s;
- *Set-point* de velocidade angular em $mrad/s$;
- *Byte* 0x0D e 0x0A indicadores de mudança de parágrafo;

Os dados são armazenados nesta seqüência no cartão micro-SD, assim, cada linha de dados no arquivo de gravação corresponde a seqüência do PWM do motor da roda direita até o fim do parágrafo.

Todos os dados são escritos no cartão micro-SD usando a formatação ASCII, assim, antes de armazenar no *buffer*, a tarefa *Amostrador* converte os dados em decimal ASCII.

A *SDControl* foi desenvolvida usando a metodologia *State Chart* [43], sendo que sua função é controlar toda a comunicação com o DOSonCHIP. Ela também é responsável pela configuração do CD17B10, montagem e desmontagem do cartão micro-SD, abertura e criação do diretório “LOG”, abertura do arquivo de armazenamento e escrita dos dados.

O inicio da operação da *SDControl* é marcado pelo reset físico do CD17B10 através de pino específico. Feito o reset, a *SDControl* liga o CD17B10, atualiza e liga seu relógio interno, monta o cartão, cria ou abre o diretório “LOG”. Selecionado o diretório “LOG”, abre ou cria o arquivo no modo de escrita, monitora se existe comando específico, verifica qual *buffer* está preenchido e pronto para escrita e escreve os dados no cartão. Na Figura 70 está a representação da máquina de estados na *SDControl*.

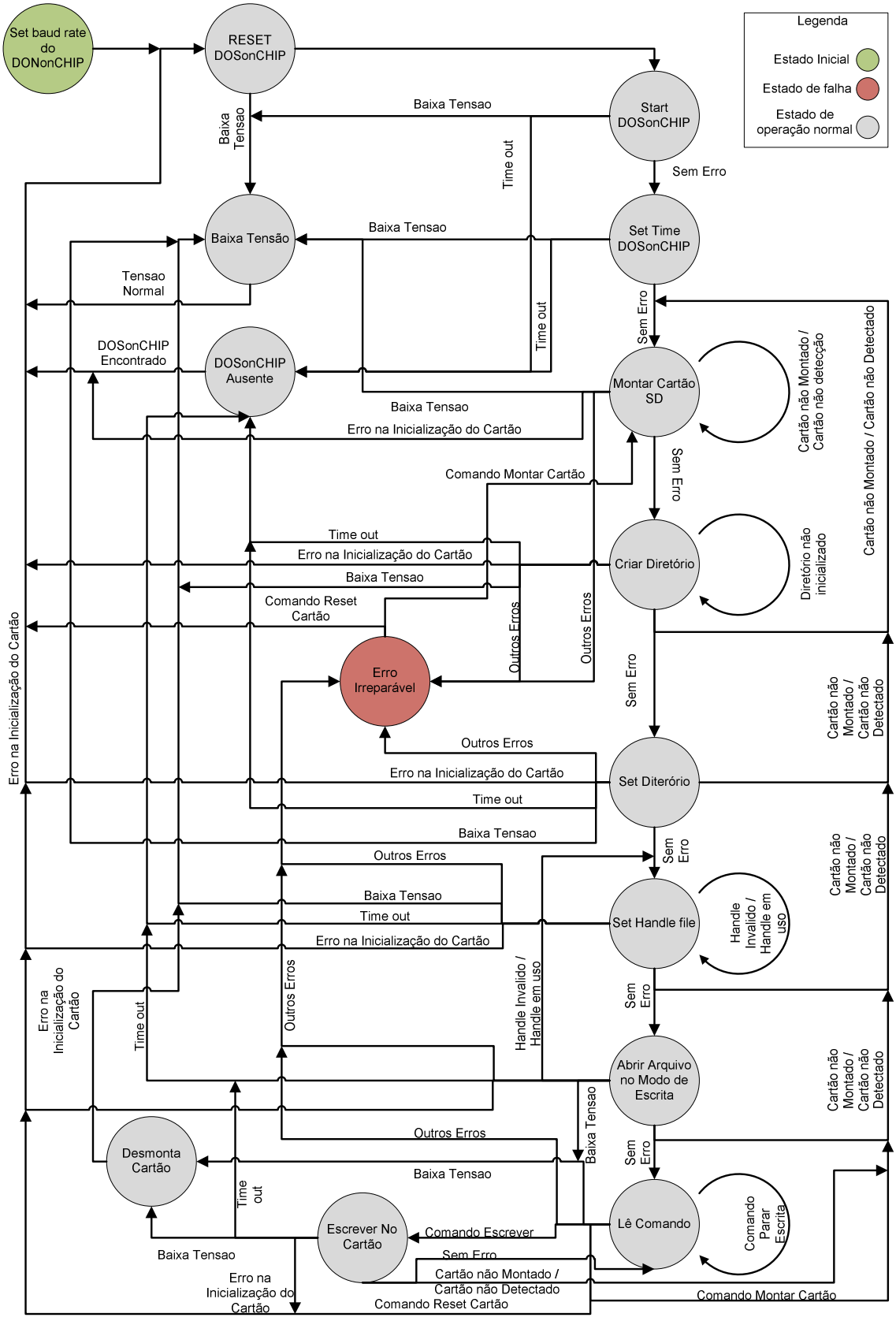


Figura 70: Representação da máquina de estados da tarefa *SDControl*.

3.4.3 Comandos Para o Cartão de Memória

A *SDControl* possui um *byte* que recebe os comandos externos de controle do cartão micro-SD. Neste trabalho foram implementados os comandos:

- Escrever dados; comando no qual ocorre a gravação normal dos dados;
- Parar escrita; comando no qual a escrita é paralisada;
- Remover cartão; comando que fecha o arquivo e desmonta o cartão de memória;
- Montar cartão; comando no qual é realizada a montagem do cartão;
- Reset DOSonCHIP; comando de reset do CD17B10;

O comando escrever é padrão ao iniciar a tarefa *SDControl*. Todos os comandos podem ser dados escrevendo seu respectivo valor no *byte SDCardComando*. Foi implementado outro *byte* de controle com a função de *status* do cartão micro-SD. Este *byte* é o *SDCardStatus*, sendo atribuído diversos valores a este *byte* conforme os *status* da *SDControl*. Mais informações sobre os códigos de programação das tarefas citadas nesta seção e valores assumidos pelo *byte SDCardStatus* e *SDCardComando* encontram-se nos anexos 11 e 12.

3.4.4 Proteção de Integridade do Cartão Micro-SD

A principal falha que pode danificar permanentemente o cartão de memória é a sub-tensão no momento de uma escrita. Assim, objetivando minimizar esta falha foi implementado um circuito de reserva de energia e detecção de sub-tensão, sendo valores abaixo de 7,3 V considerados como sub-tensão. Observando-se a Figura 70 nota-se que se ocorrer uma queda na alimentação da Placa de Processamento, a tarefa *SDControl* assume o estado de baixa tensão, no qual o arquivo de escrita é fechado e todas as operações com o cartão são encerradas.

O funcionamento do sistema de proteção de integridade do cartão micro-SD começa com o monitoramento constante da tensão de alimentação da Placa de Processamento. Se esta tensão cair abruptamente, a energia da placa é mantida por um breve tempo por intermédio de

um capacitor carregado previamente. Ao detectar a queda de tensão a *SDControl* assume o estado de baixa tensão. Na Figura 71 está a representação do circuito de tomada de força da Placa de Processamento, no qual é possível observar o ponto de monitoramento de tensão da alimentação e o capacitor que retém a energia suficiente para o mecanismo de proteção atuar.

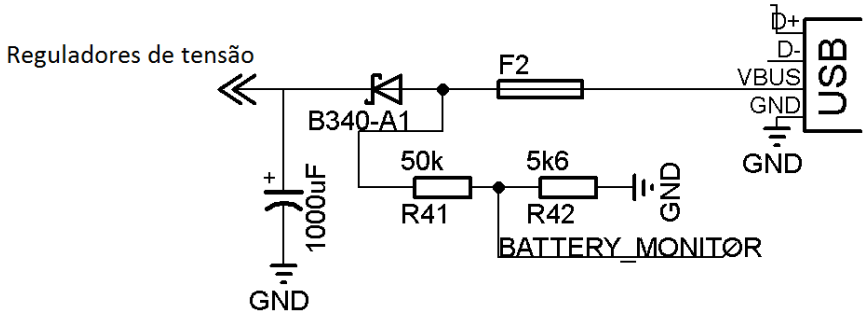


Figura 71: Circuito de tomada de força da Placa de Processamento.

3.5 O Display de LCD

Objetivando prover uma interface com usuário rápida e eficiente, foi implementado um display de LCD de quatro linhas e vinte colunas baseado no CI HD44780 [25]. Este display mostra informações de controle e *status* da cadeira de rodas. A Figura 72 ilustra a disposição dos campos de informações do display de LCD

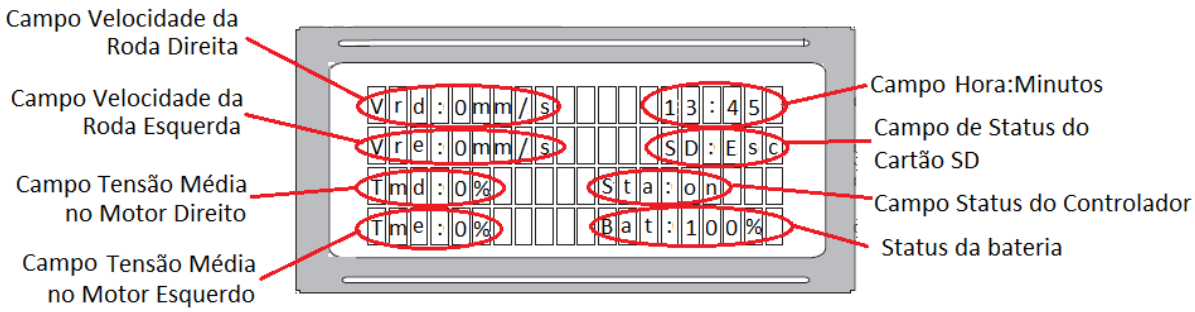


Figura 72: Posição das informações no display de LCD.

Os campos do LCD são:

- Campo de velocidade da roda direita. Este campo mostra a velocidade da roda direita em mm/s

- Campo de velocidade da roda esquerda. Este campo mostra a velocidade da roda esquerda em mm/s
- Campo tensão no motor direito. Este campo mostra a tensão média no motor direito, sendo dada em valores percentuais referentes ao PWM do motor direito.
- Campo tensão no motor esquerdo. Este campo mostra a tensão média no motor direito, sendo dada em valores percentuais referentes ao PWM do motor esquerdo.
- Campo Hora: minutos. Mostra “hora : minutos” da cadeira de rodas.
- Campo Status da bateria. Este campo mostra em uma escala percentual a carga das baterias;
- Campo *status* do cartão SD. Mostra em qual *status* se encontra o cartão SD. Estes *status* podem ser:
    - “Res” → resetando o DOSonCHIP;
    - “BT” → baixa tensão detectada;
    - “Esc” → escrevendo dados no cartão micro-SD;
    - “Ini” → iniciando DOSonCHIP;
    - “Des” → cartão desmontado;
    - “Mon” → montando cartão micro-SD;
    - “Len” → lendo dados do cartão micro-SD;
    - “ErF” → erro fatal ocorrido;
    - “Par” → DOSonCHIP paradado;
    - “AbD” → abrindo diretório;
    - “SDO” → DOSonCHIP não detectado;
    - “AbH” → abrindo handle;
    - “AbA” → abrindo arquivo;
    - “Dem” → cartão desmontado;
    - “NSD” → cartão micro-SD não detectado;
    - “XXX” → sem conexão com DOSonCHIP;
- Campo *status* do controlador. Este campo mostra em qual *status* se encontra o controlador da cadeira. O *status* do controlador é disponibilizado em uma palavra de 13 *bits* e quando apenas os *bits* 0 e 6 são 1 é mostrada a palavra “*on*”. Quando todos os *bits* são 0 é mostrada a palavra “*off*”. Quando ocorre a perda de comunicação com a Placa de Processamento é mostrada a palavra “Desc”. Para todos os outros casos é

mostrado o número inteiro correspondente a palavra de *status*, sendo necessária a decomposição em binário para verificação do ocorrido. Assim, esta palavra é composta pelos seguintes *bits*:

- *Bit* 0 → controlador *run.* Este *bit* indica que o controlador da cadeira de rodas está ativo, sendo 1 controlador ativo, e 0 controlador inativo
- *Bit* 1 → controlador salvando parâmetros. Indica que o controlador está desempenhando a tarefa de salvar os parâmetros, sendo 1 controlador salvando parâmetros, e 0 controlador não está salvando parâmetros;
- *Bit* 2 → atualizando parâmetros. Indica que o controlador está atualizando os parâmetros, sendo 1 controlador atualizando os parâmetros, e 0 controlador não atualizando os parâmetros
- *Bit* 3 → erro no salvamento dos parâmetros. Indica a ocorrência de erro no salvamento dos parâmetros, sendo 1 ocorreu um erro e 0 não ocorreu este erro.
- *Bit* 4 → perda de tempo real do controlador. Indica que o controlador foi executado um ou mais períodos acima que o permitido, ou seja, com atraso. 1 indica a perda de tempo real e 0 tempo real obedecido;
- *Bit* 5 → alerta de colisão infravermelho. Indica que um dos sensores infravermelho está indicando distancia menor que 300 mm.
- *Bit* 6 → *driver* da cadeira *run.* Indica que o *driver* da cadeira está ativo, sendo 1 para *driver* ativo e 0 para *driver* inativo.
- *Bit* 7 → *encoders* esquerdo desconectado. Indica que o *encoder* do lado esquerdo está desconectado, sendo 1 para *encoder* desconectado e 0 para *encoder* conectado.
- *Bit* 8 → *encoder* direito desconectado. Indica que o *encoder* do lado direito está desconectado, sendo 1 para *encoder* desconectado e 0 para *encoder* conectado.
- *Bit* 9 → falha no motor direito. Indica falha no motor do lado direito, sendo 1 para falha ativa e 0 para ausência de falhas;
- *Bit* 10 → falha no motor esquerdo. Indica falha no motor do lado esquerdo, sendo 1 para falha ativa e 0 para ausência de falhas;

- *Bit* 11 → perda de sincronismo. Indica que as variáveis de rede ficaram indisponíveis, sendo 1 para variáveis de rede indisponíveis e 0 para variáveis disponíveis;
- *Bit* 12 → perda de tempo real do *driver* da cadeira. Indica que o *driver* da cadeira de rodas foi executado um ou mais períodos acima que o permitido, ou seja, com atraso. 1 indica a perda de tempo real e 0 tempo real obedecido;
- *Bit* 13 → perda de tempo real do hodômetro. Indica que o hodômetro foi executado um ou mais períodos acima que o permitido, ou seja, com atraso. 1 indica a perda de tempo real e 0 tempo real obedecido;

A tarefa que atualiza o display é executada a cada 100 ms, sendo uma tarefa simples de execução direta, ou seja, não assume diferentes estados. Mais detalhes da implementação da tarefa de atualização do display encontram-se no anexo 13.

3.6 Controle das Placas de Acionamento da Cadeira de Rodas.

As Placas de Acionamento dos motores da cadeira de rodas necessitam de um controle contínuo que monitore a ocorrência de falhas e atualize os valores dos PWMs. Objetivando prover tal monitoramento das Placas de Acionamento dos motores e ainda garantir a segurança do usuário diante de possíveis falhas foi criada a tarefa, ou rotina, *driver* motores. Esta tarefa possui funcionalidades importantes para todo o sistema, visto que:

- Atualiza os valores dos temporizadores de PWMs com informações oriundas da rede CAN;
- Monitora falhas nos motores, através das falhas nos *gate drivers*;
- Monitora a conexão dos *encoders*;
- Monitora a disponibilidade das variáveis de barramento necessárias para seu bom funcionamento;
- Interpreta comandos oriundos da rede CAN;

A *driver* motores inicia sua execução configurando os canais 0, 1 e 2 do *timer* A do microcontrolador como base de tempo PWM de 20 kHz, largura de pulsos do PWM do motor

direito e largura de pulsos do PWM do motor esquerdo, respectivamente. Após este procedimento e a configuração das variáveis da rede CAN, ela inicia a execução da máquina de estados principal.

Novamente foi usada a técnica de programação *State Chart* [43] para implementar a máquina de estados principal (Figura 73).

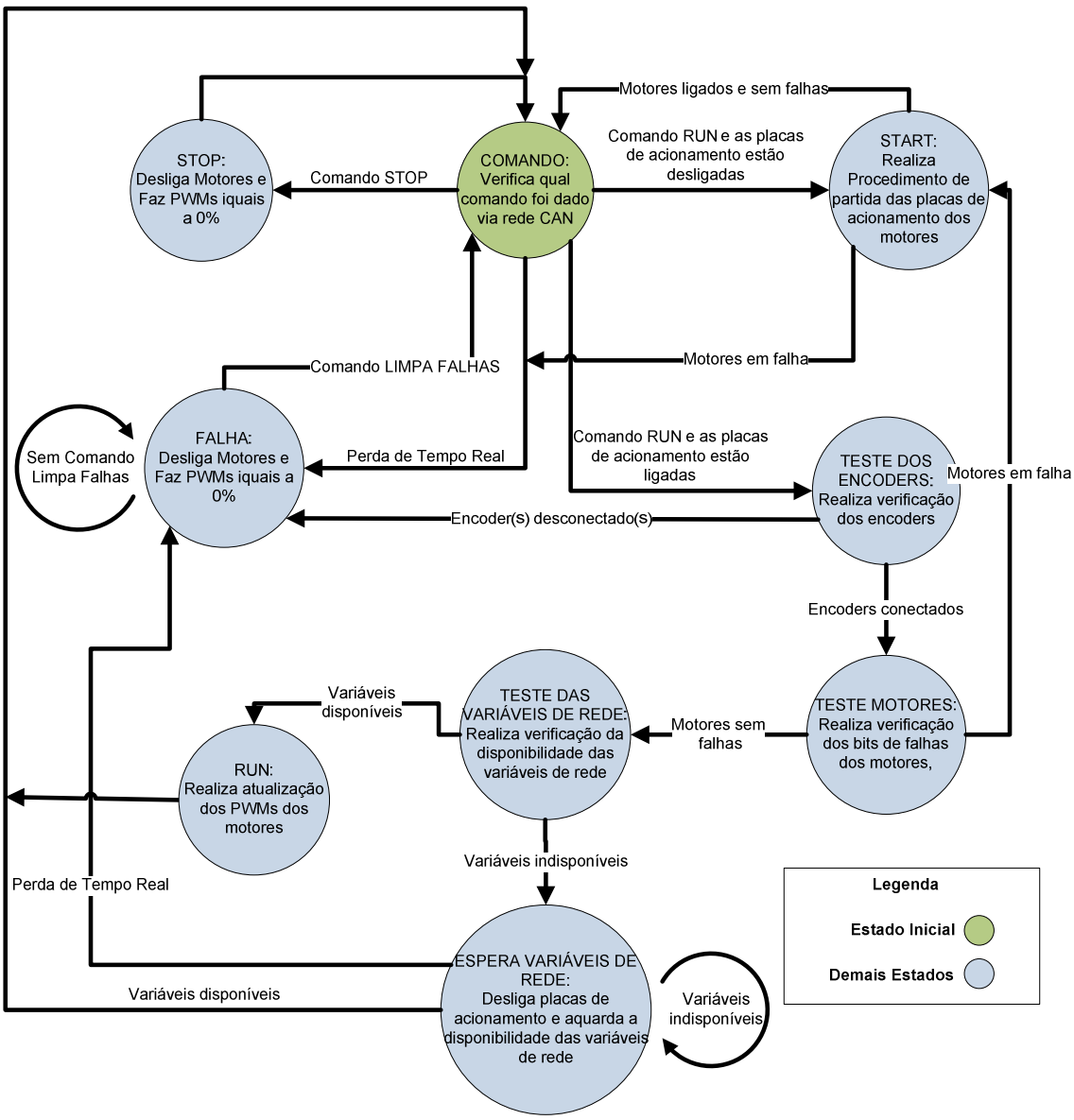


Figura 73: Representação da máquina de estados da rotina *driver* motores.

A segurança é tratada pela *driver* motores como prioridade, visto que é ela que aciona os motores da cadeira de rodas. As verificações de segurança incluem:

- Testes dos *encoders*; é realizada no estado TESTE DOS ENCODERS (Figura 73) e verifica se os PWMs estão em mais de 30 % ou menos de -30 %, e as velocidades das rodas estão em 0 mm/s. Se esta condição se mantiver por mais de 1 segundor é setado o *bit* de *encoder* desconectado para a roda em questão e os motores são desligados;
- Teste dos motores; é realizado no estado TESTE MOTORES e consiste na verificação dos *bits* de FAULT/SD (Capítulo 1) das Placas de Acionamento. Se ocorrer falha, é setado o *bit* de falha do motor nesta condição, os motores são desligados e a máquina de estados assume o estado START;
- Teste de disponibilidade das variáveis de rede; esta verificação é realizada no estado TESTE DAS VARIÁVEIS DE REDE, e consiste na verificação da atualização das variáveis de comando e PWMs, visto que são variáveis oriundas da rede CAN. Se uma ou mais destas variáveis permanecer sem atualização por mais de 30 ms os motores são desligados e a máquina de estados vai para o estado ESPERA VARIÁVEIS DE REDE, no qual permanece até que seja restabelecida a atualização das variáveis.
- Teste de tempo real; esta verificação é realizada nos estados COMANDO e ESPERA VARIÁVEIS DE REDE. Este teste consiste na verificação do número de chamadas desta rotina que estão aguardando execução. Se houver mais de uma chamada é setado o *bit* de perda de tempo real, os motores são desligados e a máquina de estados assume o estado de falha. As chamadas da rotina *driver* motores ocorrem por uma única fonte, a interrupção de tempo do *timer* B canal 0, e assim, se houver mais de uma chamada pendente implica que o *timer* estourou duas vezes e a rotina não foi executada. Esta condição caracteriza perda de tempo real.

Para ligar os motores existe um procedimento específico usado para minimizar a possibilidade de queima dos componentes da ponte H. Este procedimento inicia ajustando os PWMs para 0 %, liga o regulador de 15 V e aciona-se o sistema de *soft shutdown* dos *gate drivers* de cada Placa de Acionamento dos motores. Após 400 ms, é ligado o regulador de 5 V de cada placa. Novamente aguarda-se mais 400 ms e ao término deste tempo o comando de *soft shutdown* é retirado. Aguarda-se 10 ms e é enviado uma comando de *fault clear* [16] para

cada Placa de Acionamento. Aguardam-se 50 ms e os comandos de *fault clear* são retirados e as Placas de Acionamento estão ligadas. Este procedimento é realizado pelo *driver* motores no estado START.

Os comandos interpretados pela *driver* motores são inseridos em uma palavra de comandos de 2 *bits* disponível na rede CAN, sendo;

- *Bit* 0 → comando limpa falhas. Este comando é dado fazendo este *bit* iqual 1 e após 30 ms retornando-o para 0;
- *Bit* 1 → comando liga motores, sendo 1 para ligar motores e 0 para desligá-los. Com este *bit* em 1 é dado o comando RUN e com este *bit* em 0 é dado o comando STOP.

O *status* da *driver* motores está inserido em uma palavra de comandos de 7 *bits* disponível na rede CAN, sendo;

- *Bit* 0 → *Run*, sendo 1 indica que a *driver* motores está em execução normal, e sendo 0 indica que a *driver* motores está em STOP.
- *Bit* 1 → *encoder* esquerdo desconectado, sendo 1 para indicar *encoder* esquerdo desconectado, e 0 para o *encoder* esquerdo conectado.
- *Bit* 2 → *encoder* direito desconectado, sendo 1 para indicar *encoder* direito desconectado, e 0 para o *encoder* direito conectado.
- *Bit* 3 → falha no motor direito, sendo 1 indica que a falha FAULT/SD da Placa de Acionamento do motor direito está ativa, e sendo 0 indica que a falha FAULT/SD da Placa de Acionamento do motor direito está inativa;
- *Bit* 4 → falha no motor esquerdo, sendo 1 indica que a falha FAULT/SD da Placa de Acionamento do motor esquerdo está ativa, e sendo 0 indica que a falha FAULT/SD da Placa de Acionamento do motor esquerdo está inativa;
- *Bit* 5 → perda de sincronismo, sendo 1 indica que ocorreu a perda de sincronismo devido a indisponibilidade das variáveis de barramento, e sendo 0 indica que as variáveis de barramento estão disponíveis.
- *Bit* 6 → perda de tempo real, sendo 1 indica a ocorrência de perda de tempo real, e sendo 0 indica que as restrições de tempo real estão sendo obedecidas.

Os parâmetro CAN das palavras de comando e status da driver motores estão disponíveis na Tabela 14 e o código C da sua implementação está no anexo 14.

3.7 O Controlador do RTC DS1307

Objetivando prover informações de relógio para as tarefas de controle do cartão micro-SD e demais aplicações, foi implementado, na Placa de Processamento, um RTC usando o CI DS1307 [40]. Este CI é capaz de prover informações de ano, mês, dia, dia da semana, horas, minutos e segundos. Possui capacidade de se manter em funcionamento após o desligamento da Placa de Processamento usando uma bateria de $3{,}3\ V$ acoplada diretamente a ele. O DS1307 disponibiliza um pino de $1\ Hz$ e uma interface de comunicação I2C [40] de $100\ kHz$.

Para interface com o DS1307 foi criada uma rotina específica chamada Relógio, que inicia sua execução configurando a USCI_B0 do microcontrolador da Placa de Processamento para o modo mestre I2C a $100\ kHz$. Após esta etapa a Relógio lê do DS1307 a data e hora atualizadas e as registra em suas variáveis internas de tempo. Feita a atualização inicial a tarefa Relógio passa a monitorar a cada $100\ ms$ o pino do microcontrolador ligado ao sinal de $1\ Hz$ do DS1307, sendo que ao detectar uma transição positiva, a tarefa Relógio incrementa a sua variável de segundos, que ao atingir 60 faz a Relógio ler novamente os dados do DS1307 e assim atualizar suas variáveis de tempo.

As variáveis de tempo da rotina Relógio são: ano, mês, dia, dia da semana, horas, minutos e segundos. Todas as variáveis estão disponibilizadas na rede. Os parâmetros CAN destas variáveis estão na Tabela 14, o código C da implementação da tarefa Relógio está no anexo 15.

3.8 Protocolo de Comunicação Com Supervisório / Aplicação de Alto Nível

Como este trabalho tem por objetivo criar uma infraestrutura para aplicações de alto nível para a cadeira de rodas, para lograr êxito neste objetivo a cadeira de rodas deve estabelecer comunicação com esta aplicação. A interface escolhida para tal foi o protocolo UART RS-232 [44]. Os motivos para esta escolha são resumidos em:

- Ampla disponibilidade em PCs industriais;
- Facilidade de manipulação;
- Boa imunidade a ruídos a curtas distâncias;

3.8.1 A Rotina de Controle da UART

A função de comunicação com a aplicação de alto nível é executada pela tarefa UART na Placa de Processamento. Esta tarefa inicia suas funções configurando a USCI_A1 do microcontrolador da Placa de Processamento para:

- Modo UART;
- Taxa de sinalização de 57,6 $kbps$;
- 8 *bits* de dados;
- 1 *bit* de *stop*;
- 1 *bit* de *start*;
- Sem paridade;

Após a configuração da interface serial do microcontrolador a tarefa UART inicia seu funcionamento normal. A tarefa UART opera sob demanda da aplicação externa, ou seja, fica aguardando os comandos para então respondê-los. Ela fica aguardando a liberação de um semáforo próprio que é somente liberado pela recepção de um dado qualquer pela rotina de interrupção de recepção. Logo que este semáforo é liberado pela interrupção de recepção, a tarefa UART avalia o dado recém recebido e se for um comando válido o executa.

O semáforo atua também como contador de *bytes* recebidos, visto que quando um *byte* novo é recebido pela interface serial, a rotina de tratamento de interrupção de recepção UART armazena em um *buffer* o *byte* recebido e incrementa o semáforo. Logo, ao verificar a disponibilidade de dados pelo semáforo, a tarefa UART decrementa o semáforo e lê o *byte* do *buffer* de recepção, interpreta o comando e envia uma resposta se o comando pedir. Se antes de decrementar o semáforo seu valor for 0 a tarefa UART é então suspensa pelo *kernel* até que outro dado seja recebido.

3.8.2 Os Comandos de Alto Nível

Todos os dados trocados com a cadeira de rodas pela interface serial são codificados como decimais ASCII. Foram criados diversos pacotes de informações, sendo todos seguindo o seguinte padrão:

- Identificador do comando →Primeiro *byte*, ou primeiro e segundo *bytes*;
- Argumento → *bytes* de dados
- Delimitador de comando padrão ‘@’ → último *byte*;

Assim, o comando tem a seguinte estrutura:

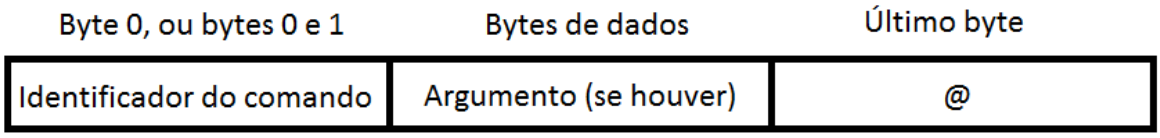


Figura 74: Estrutura dos comandos seriais.

Os comandos seriais podem ser de atualização de parâmetros, comandos simples e requisição de informações. A Tabela 6 ilustra as relações entre os comandos, argumentos e as respostada enviadas pela cadeira de rodas. A implementação na linguagem C da tarefa UART esta descrita no anexo 16.

Tabela 6: Comandos de alto nível da cadeira de rodas.

Descrição	Identificador	Argumento	Resposta
Informações do controlador	"Da"	-	"Da" + "valor da palavra de *status* do controlador" + ":" + "Velocidade da roda esquerda em mm/s" + ":" + "Velocidade da roda direita em mm/s" + ":" + "valor percentual multiplicado pó 10 do PWM do motor esquerdo" + ":" + "valor percentual multiplicado pó 10 do PWM do motor direito" + "@"
***Set-point* de velocidade linear**	"Db"	"Valor de *set-point* de velocidade linear em mm/s"	-
***Set-point* de diferença de velocidade entre as rodas**	"Dc"	"Valor de *set-point* de diferença de velocidade entre as rodas em mm/s"	-
Atualiza valor de Kp do controlador	"Dd"	"valor de Kp multiplicado por 100"	-
Atualiza valor de Kid (ganho do laço integral entre as rodas) do controlador	"De"	"valor de Kid multiplicado por 100"	-
Atualiza valor de Ki (ganho do integrador de cada roda) do controlador	"Df"	"valor de Ki multiplicado por 100"	-
Informações de ganhos do controlador e *set-points* ativos	"Dg"	-	"Dg" + "Valor de *set-point* de velocidade linear em mm/s" + ":" + "Valor de *set-point* de diferença de velocidade entre as rodas em mm/s" + ":" + "valor de Kp multiplicado por

			100” + “:” + “valor de Kid multiplicado por 100” + “:” + “valor de Ki multiplicado por 100” + “@”
Comando para o cartão micro-SD	“C1”	“Comando”	-
Informações de *status* do cartão micro-SD	“C2”	-	“C” + “valor da palavra de *status* do cartão micro-SD” + “:” + “Valor máximo atingido pelo *buffer* 1 de escrita” + “:” + “Valor máximo atingido pelo *buffer* 2 de escrita” + “:” + “Valor atual do *buffer* 1 de escrita” + “:” + “Valor atual do *buffer* 2 de escrita” + “última resposta do DOSonCHIP” + “@”
Informações de data/hora da cadeira de rodas	“h”	-	“h” + “horas” + “:” + “minutos” + “:” + “segundos” + “ ” + “dia da semana” + “ ” + “dia do mês” + “/” + “mês” + “/” + “ano” + “@”
Configura informações de data/hora da cadeira de rodas	“H”	“horas” + “:” + “minutos” + “:” + “segundos” + “ ” + “dia da semana” + “ ” + “dia do mês” + “/” + “mês” + “/” + “ano”	-
***Status* Geral da cadeira de rodas**	“P”	-	“P” + “TEC” + “:” + “REC” + “:” + “valor da palavra de erros do MC2515” + “:” + “valor medido pelo *infravermelho* 0” + “:” + “valor medido pelo *infravermelho* 1” + “:” + “valor medido pelo *infravermelho* 2” + “:” + “valor medido pelo *infravermelho* 3” + “:” + “valor da palavra de *status* do controlador” + “:” + “Velocidade da roda esquerda em mm/s” + “:” + “Velocidade da roda direita em mm/s” + “:” + “valor percentual multiplicado pó 10 do

			PWM do motor esquerdo” + “:” + “valor percentual multiplicado pó 10 do PWM do motor direito” + “:” + “valor da palavra de *status* do cartão micro-SD ” + “:” + “tensão de alimentação em mV” + “@”
Ping cadeira de rodas	“T”	-	“T@”

3.9 Conclusão

Os sensores infravermelhos e ultrassônicos são de modo geral complementares entre si, visto que o sonar não pode mensurar distancias menores que aproximadamente 15 cm, e o infravermelho entre 0 e 60 cm. Portanto o uso de ambos neste projeto se justifica. Os *encoders* incrementais são os mais amplamente utilizados para aplicações de medição de velocidade, e neste trabalho apresentaram medição confiável e precisa. O cartão micro-SD, o RTC, o LCD e a interface com serial RS-232 são periféricos que tornaram possível uma maior interação com o usuário. Tal interação é justificada, pois este trabalho tem o objetivo de servir de base para aplicações comerciais da Cadeira de Rodas Robotizada.

4 O BARRAMENTO DE COMUNICAÇÃO CAN (CONTROL AREA NETWORK)

A arquitetura eletrônicas e de algoritmos da Cadeira de Rodas Robotizada é baseada em uma estrutura de rede, sendo a troca de dados e comandos entre a PP, PCP e outros dispositivos suportada pela rede CAN implementada (Figura 75). Este Capítulo retrata os princípios de funcionamento da rede CAN, os dispositivos e as rotinas que controlam o barramento CAN. Mostra também as relações de dependência entre as rotinas de comunicação executadas pelo microcontrolador e as demais rotinas, e como são efetuadas as atualizações de variáveis chamadas de variáveis de barramento usando a rede CAN.

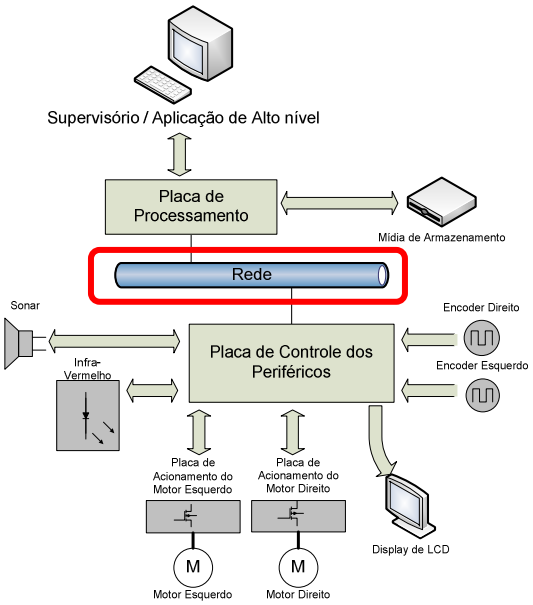


Figura 75: Arquitetura básica da Cadeira de Rodas Robotizada com a Rede CAN em destaque.

4.1 A Rede CAN (Arquitetura de rede)

Esta seção ilustra aspectos básicos da rede CAN, como histórico, protocolo, detecção de erros e taxa de sinalização.

4.1.1 Histórico

CAN é um sistema de comunicação serial internacionalmente padronizado (ISO 11898 e ISO 11519/1) [45] que oferece funcionalidade da camada de enlace de dados do

modelo de referência OSI/ISO. Foi desenvolvido por Robert Bosch (da BOSCH), uma companhia fornecedora de componentes e subsistemas para a indústria automotiva, e estendeu-se seu uso para outras aplicações, tais como sistemas médicos, instrumentação náutica, maquinaria de processamento de papel, sistemas de controle de elevador, produção têxtil, sistema de controle de linha de produção em geral [46].

Especificamente na indústria automotiva, CAN se presta não apenas para facilitar o atendimento dos desejos dos clientes em termos de maior segurança, conforto e conveniência, mas também para alcançar os crescentes requisitos governamentais de controle de poluição e redução de consumo de combustível. Muitos sistemas eletrônicos já equipam os carros atuais (alguns exemplos: sistema de freios ABS, sistema de gerenciamento de motor, controle de tração, controle de ar condicionado, suspensão ativa, controle central de portas, vidros, luminosidade, etc.) cuja complexidade, junto com a necessidade de flexibilidade e expansão de rede, favorecem o uso deste protocolo. No CAN, os controladores, sensores e atuadores se comunicam a velocidades de até 1 $Mbps$ sobre um barramento de dois fios – tipicamente par trançado, blindado ou não [46].

4.1.2 Princípios

A rede CAN é um protocolo do tipo *multi-master*, sistema de transmissão de mensagens que especifica uma taxa máxima de *bit* de sinalização de até 1 $Mbps$. Ao contrário de uma rede tradicional, tais como USB ou Ethernet, CAN não envia grandes blocos de dados ponto-a-ponto, ou seja, de um nó para outro nó, sob a supervisão de um mestre. Em uma rede CAN muitas mensagens curtas como temperatura ou rpm são transmitidas para toda a rede, que permite a consistência dos dados em cada nó do sistema. Outra característica da rede CAN é sua forma de endereçamento, que não enfatiza os dispositivos, mas sim as mensagens,. Assim, em uma rede CAN são as mensagens que são endereçadas e não os dispositivos.

A arquitetura típica para um nó CAN contém basicamente um controlador de barramento CAN, um *transceiver* para interface com o meio físico e a aplicação que é tipicamente um microcontrolador (Figura 76).

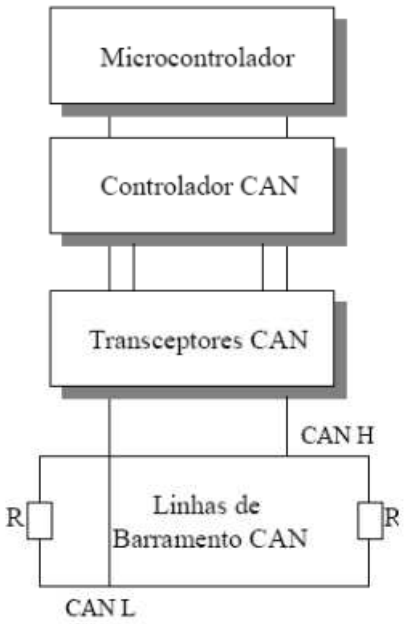


Figura 76: Arquitetura típica de um nó CAN.

Se o nó deseja transmitir dados a um ou mais nós, dados e identificador são passados do microcontrolador ao controlador CAN. O controlador CAN constrói e transmite a mensagem, e cada estação CAN que receba a mensagem corretamente faz um teste de aceitação, verificando se a mensagem está livre de erros ou não. Como o protocolo de transmissão de dados não requer endereço de destinatário físico, ele suporta tanto múltipla recepção (*broadcast, multicast*) quanto sincronização de processos distribuídos, ou seja, mensurações necessárias para vários controladores podem ser transmitidos via a rede [46].

O protocolo CAN se divide basicamente em duas versões: o *Standard* e o *Extended.* A diferença entre estes se limita ao comprimento do campo de identificação da mensagem, que no caso do *Standard* é de 11 *bits* e no caso do *Extended* são 29 *bits*. Nas seções seguintes serão detalhadas estas versões.

Por compatibilidade com o modelo OSI/ISSO, e para alcançar transparência e flexibilidade de implementação, o CAN é dividido em 3 camadas: camada física, camada de link de dados ou enlace e camada de aplicação. A Figura 77 ilustra a participação dos atores no protocolo CAN e suas camadas.

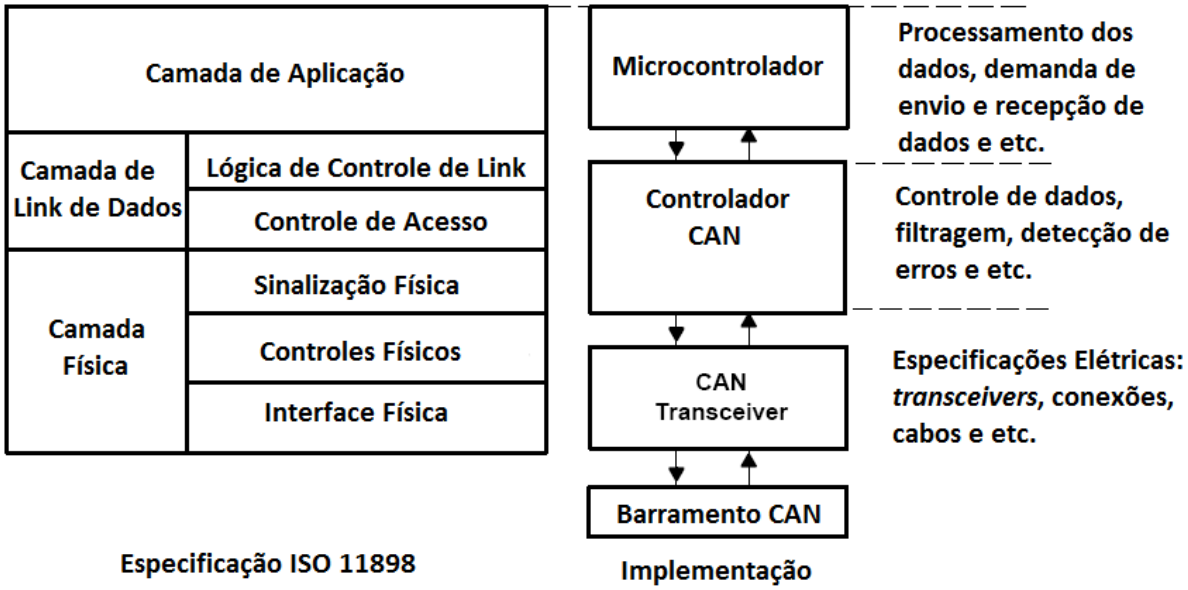


Figura 77: Camadas OSI da rede CAN [46].

4.1.3 CAN Camada Física

CAN especifica dois estados lógicos: recessivo e dominante. A ISO-11898 [45] define uma tensão diferencial para representar os estados recessivo e dominante como mostrado na Figura 78. O estado recessivo, ou *bit* recessivo, é a representação da lógica '1' e o estado dominante, ou *bit* dominante, é a representação da lógica '0' [47].

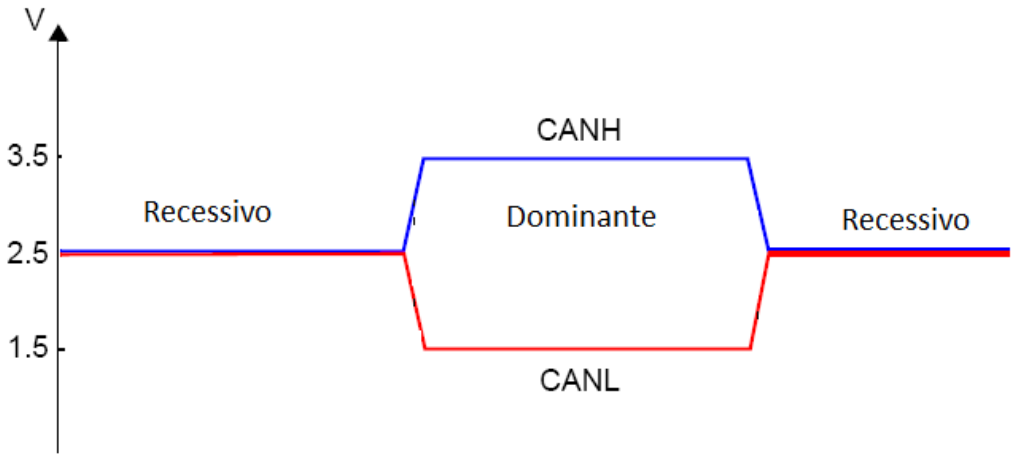


Figura 78: Níveis elétricos do barramento CAN.

A ISO-11898-2 [45] não especifica os fios mecânicos e conectores, contudo, exige resistores de terminação de 120Ω (nominal) em cada extremidade do barramento. A Figura 79 mostra um exemplo de uma rede CAN com base na ISO-11898 [45] [47].

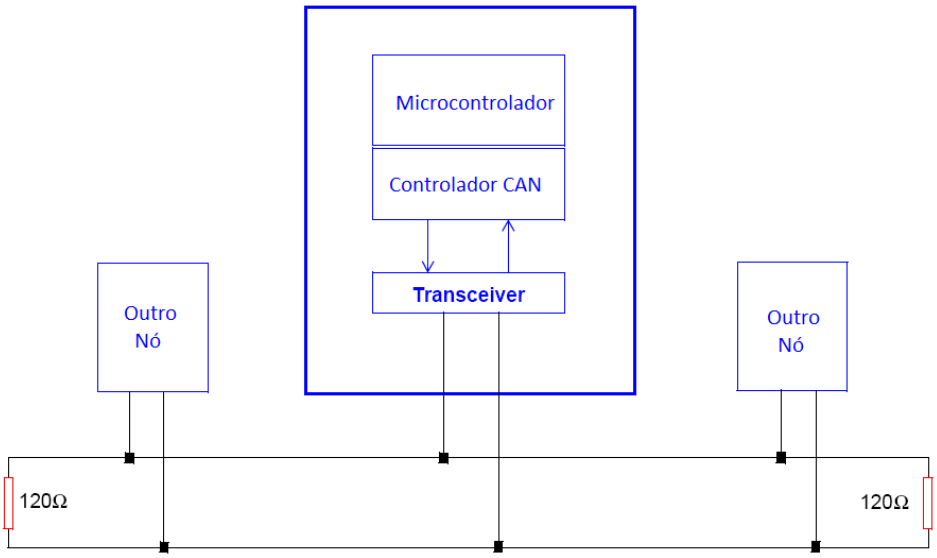


Figura 79: Representação de uma rede CAN camada física.

4.1.4 O Protocolo CAN *Standard*

O protocolo de comunicação CAN é do tipo *carrier-sense multiple-access*, ou seja, com detecção de colisão e de arbitragem na prioridade da mensagem (CSMA / CD + AMP). CSMA significa que cada nó no barramento tem que esperar por um determinado período de inatividade antes de tentar enviar uma mensagem. CD + AMP significa que as colisões são resolvidas por meio de uma arbitragem *bit* a bit, com base em uma pré-programação de prioridade de cada mensagem no campo identificador das mensagens. O identificador de maior prioridade sempre ganha acesso ao barramento quando existe mais de um nó tentando transmitir ao mesmo tempo [10].

A primeira versão CAN, cujas normas estão listadas na Tabela 7, é a ISO 11519 (*Low-Speed* CAN) que é para aplicações de até 125 *kbps* com um identificador de 11 *bits*. A segunda versão, ISO 11898 (1993), também com 11 *bits* identificadores prevê taxas de sinalização de 125 kbps a 1 *Mbps*, enquanto o mais recentes ISO 11898 emenda (1995) introduz o identificador de 29 *bits*. O ISO 11898 (1993) 11 *bits* é muitas vezes referido como

Standard CAN ou CAN Versão 2.0A, enquanto a ISO 11898 (1995) é referida como *Extended* CAN ou CAN Versão 2.0B. O *Standard* tem identificador de 11 *bits*, assim permite até 2^{11} ou 2048 identificadores de mensagens diferentes, enquanto o *Extended* CAN 29-*bit* fornece até 2^{29}, ou seja, 537 milhões de identificadores [10].

Tabela 7: Versões da rede CAN e suas características.

Nome	Padrão	Taxa Máxima	Identificador
Low–Speed CAN	ISO 11519	125 kbps	11 *bits*
CAN 2.0A	ISO 11898:1993	1 Mbps	11 *bits*
CAN 2.0B	ISO 11898:1995	1 Mbps	29 *bits*

4.1.5 O Pacote CAN *Standard*

Figura 80: Pacote de comunicação do protocolo CAN 2.0A.

O significado dos campos de *bits* da Figura 80 são:

- SOF: *bit* dominante de início de *frame* (SOF). Este *bit* marca o início de uma mensagem e é usado para sincronizar os nós do barramento depois de ficar ocioso.
- *Identifier*: identificador de 11 *bits* que estabelece a prioridade da mensagem. Quanto menor seu valor binário maior será sua prioridade.
- RTR: 1 *bit* de requisição de transmissão (RTR). É dominante quando a informação é requerida de outro nó. Todos os nós recebem o pedido, mas o identificador determina o nó especificado. Os dados de resposta também são recebidos por todos os nós e usados por qualquer nó interessado.
- IDE: *bit* que indica o tipo de identificador que está sendo transmitido. O *bit* (IDE) dominante significa que um identificador CAN *Standard* está sendo

transmitido, e recessivo indica que um identificador *Extended* CAN está sendo transmitido;

- r0: reservado (para possível uso, por alteração futura da norma);
- DLC: 4 *bits* de comprimento de dados. Os *bits* (DLC) contêm o número de *bytes* de dados que estão sendo transmitidos.
- Data: Até 64 *bits* de dados de aplicativos podem ser transmitidos.
- CRC: 16 *bits* (15 *bits* mais delimitador), verificação de redundância cíclica (CRC) contém o *checksum* dos dados de aplicação anterior para detecção de erros.
- ACK: Cada nó receptor precisa substituir esse *bit* recessivo original da mensagem por um *bit* dominante, indicando que uma mensagem sem erros foi enviada. Se um nó receptor detectar um erro e deixar este *bit* recessivo descartará a mensagem e o transmissor irá repetir a mensagem em seguida. Desta forma cada nó reconhece (ACK) a integridade dos seus dados. ACK é possui 2 *bits*, sendo que o primeiro *bit* é o reconhecimento e o segundo é um delimitador.
- EOF: Indica fim de *frame* (EOF). Os 7 *bits* deste campo marcam o fim de um *frame* CAN, ou seja, fim da mensagem, e desativa *bit-stuffing*, indicando um erro de enchimento quando dominante. Quando 5 *bits* da mesmo nível lógico ocorrem em sucessão durante a operação normal, um *bit* de lógica invertida é enxertado após a seqüência anterior .
- IFS: é um conjunto de 7 *bits* de espaço *inter-frame* (IFS), que contém a quantidade de tempo exigido pelo controlador CAN para mover um *frame* recebido corretamente à sua posição correta em uma área de *buffer* de mensagens.

4.1.6 O Padrão Extended CAN

Figura 81: Pacote de comunicação do protocolo CAN 2.0B.

Conforme mostrado na Figura 81, o *frame* da mensagem *Extended* é o mesmo da mensagem *Standard* com a adição de:

- SRR: o *bit* (SRR) substitui o *bit* RTR do *Extended* CAN.
- IDE: se este *bit* é recessivo no identificador E*xtended* (IDE) indica que há mais 18 *bits* identificadores em seguida.
- r1: *bit* de reserva adicional [10].

4.1.7 Mecanismo de Arbitração e Prioridades

A prioridade de uma mensagem CAN é determinada pelo valor binário do identificador sendo que, o menor valor numérico corresponde a mais alta prioridade. A arbitração é feita do *bit* mais significativo em direção ao menos significativo, de acordo com o mecanismo *bit-wise arbitration* [10], onde o *bit* no estado dominante (‘0’ lógico) sobrescreve o *bit* no estado recessivo (‘1’ lógico). A Figura 82 ilustra a seqüência de arbitração na qual o nó 2 ganha a arbitração.

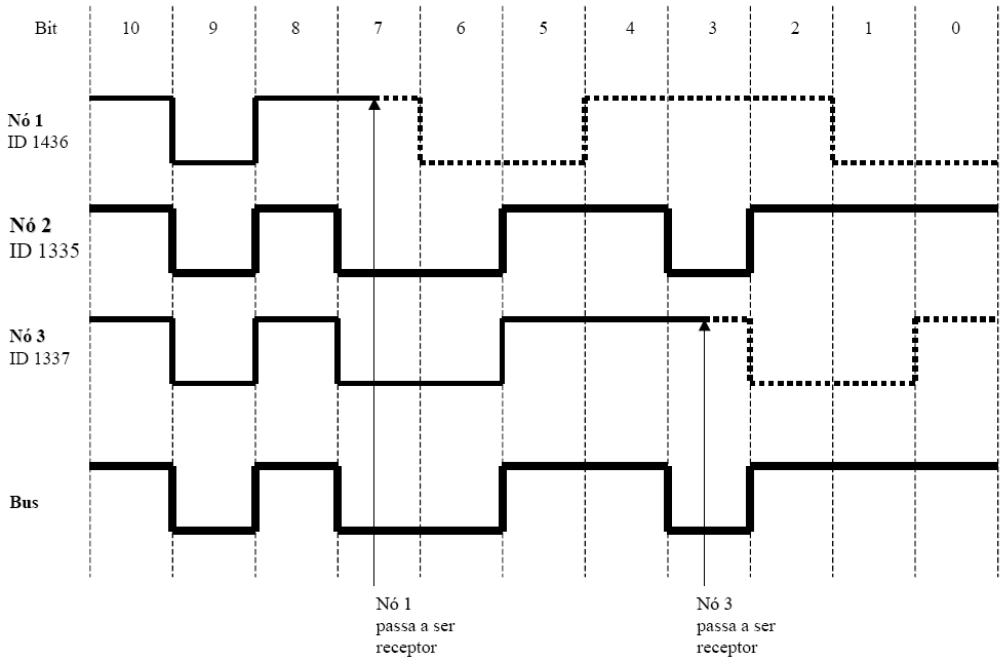


Figura 82: Arbitração Bitwise Não Destrutiva.

Qualquer atividade na linha é monitorada por todos os nós, inclusive o(s) transmissor(es). Logo que o transmissor de prioridade mais baixa detecta a perda na arbitração do bit, torna-se um receptor e não tenta retransmissão até que o barramento volte à inatividade [46].

4.1.8 Tipos de Mensagens CAN

Existem quatro tipos diferentes de mensagens, ou *frames* que podem ser transmitidos em uma rede CAN: o *frame* de dados, o *frame* remoto, o *frame* de erro, e o *frame* de sobrecarga. A mensagem é considerada livre de erros, quando o último *bit* do campo EOF final de uma mensagem é recebido no estado recessivo. Um *bit* dominante no campo EOF faz com que o transmissor repita uma transmissão [10].

Cada tipo de mensagem CAN tem um número máximo e mínimo de *bits*. Estes números dependem também do tipo de endereçamento, *Standard* ou *Extended.* A Tabela 8 ilustra a quantidade de *bits* transmitidos em cada um dos tipos de pacotes, *ou frames*, CAN.

Tabela 8: Tabela de correlação entre os tipos de mensagens CAN.

Tipo de Mensagem	**Quantidade de *Bits* Por Pacote (Número de Bits)**	
	Standard	*Extended*
***Frame* de Dados**	De 47 à 111	De 67 à 131
***Frame* de Erro**	14	14
***Frame* Remoto**	47	67
***Frame* de Sobrecarga**	14	14

4.1.8.1 *Frame* de Dados CAN

O *frame* de dados é o tipo de mensagem mais comum, e é composto pelo campo de arbitragem, o campo de dados, o campo de CRC [48], e o campo de reconhecimento. O campo de arbitragem determina a prioridade de uma mensagem quando dois ou mais nós estão lutando pelo barramento. O campo de arbitragem contém um identificador de 11 *bits* para o CAN 2.0A e o *bit* RTR (Figura 80), que é dominante para os *frames* de dados. O CAN 2.0B (Figura 81) contém o identificador de 29-*bit* e o *bit* RTR. Em seguida vem o campo de dados, que contém de zero a oito *bytes* de dados, e o campo que contém o CRC [48] de 16 *bits* de verificação utilizados para a detecção de erro. Por último, existe o campo de confirmação.

Qualquer controlador CAN que recebe uma mensagem corretamente envia um *bit* dominante ACK que substitui o *bit* transmitido recessivo no final da transmissão da mensagem correta. O transmissor verifica a presença do *bit* dominante ACK e retransmite a mensagem se não o detectar [10].

4.1.8.2 *Frame* Remoto

A finalidade do *frame* remoto é solicitar a transmissão de dados de outro nó. O *frame* remoto é semelhante ao *frame* de dados, com duas diferenças importantes. Em primeiro lugar, este tipo de mensagem é explicitamente marcado como um *frame* remoto por um *bit* RTR recessivo no campo de arbitragem, e em segundo lugar, não há dados [10].

4.1.8.3 O *Frame* de Erro

O *frame* de erro é uma mensagem especial que viola as regras de formatação de uma mensagem CAN. É transmitida quando um nó detecta um erro em uma mensagem, e faz com que todos os outros nós na rede enviem um *frame* de erro também. O transmissor original, ao detectar o *frame* de erro, automaticamente retransmite a mensagem. Há um elaborado sistema de contadores de erro no controlador CAN que garante que um nó não pode segurar o barramento repetidamente com *frames* de erro [10].

4.1.8.4 O *Frame* de Sobrecarga

É semelhante ao *frame* de erro no formato, e é transmitido por um nó que se torna muito ocupado. É usado principalmente para prever um atraso extra entre as mensagens [10].

4.1.9 O Mecanismo de Detecção de Erros

A robustez da rede CAN pode ser atribuída em parte à sua abundante verificação de erros. O Protocolo CAN incorpora cinco métodos de verificação de erros: três no nível de mensagem e dois no nível de bit. Se uma mensagem não atender a qualquer um destes métodos de detecção de erro, ela não é aceita e um *frame* de erro é gerado a partir dos nós de recebimento, fazendo com que o nó de transmissão reenvie a mensagem até que esta seja recebida corretamente. No entanto, se um nó com defeito ocupa o barramento continuamente repetindo um erro, sua capacidade de transmissão é removida por seu controlador após um número de erros limite ser atingido [10].

No nível da mensagem é usado para detecção de erros o CRC [48] e os *slots* ACK, exibidos na Figura 80 e Figura 81. O CRC de 16 *bits* contém a soma de verificação dos dados de aplicação para detecção de erros anteriores com 15 *bits* de *checksum* e 1 *bit* delimitador. O campo ACK possui dois *bits* de comprimento e consiste em reconhecer o *bit* e um *bit* delimitador. Finalmente, no nível da mensagem há outra forma de *check* de erros que consiste na verificação dos campos da mensagem que devem ser sempre recessivos. Se um *bit* dominante é detectado, é gerado um erro. Os *bits* de verificação são SOF, EOF, delimitador de ACK, e o *bit* delimitador de CRC [10].

No nível de bit, cada *bit* transmitido é monitorado pelo transmissor da mensagem. Se um *bit* de dados é escrito no barramento e seu oposto é lido, é gerado um erro. As únicas exceções a esta regra são o campo identificador de mensagem que é usado para arbitragem, e os ACK que requerem que um *bit* recessivo seja substituído por um *bit* dominante. Outro método de detecção de erros usado pela rede CAN é a regra de inserção de *bits* (*bit stuffing*), onde depois de cinco *bits* consecutivos do mesmo nível lógico, se o *bit* seguinte for de mesmo nível lógico um erro é gerado. Logo, a técnica *bit stuffing* consiste em após a inserção de cinco *bits* de mesmo valor lógico um sexto *bit* é adicionado com valor lógico invertido dos anteriores. Este *bit* não tem qualquer valor para a mensagem, apenas é inserido para detecção de erros e sincronismo. O *bit stuffing* garante subida de bordas disponíveis para sincronização em curso da rede, e que um fluxo de *bits* recessivos não sejam confundidos com um *frame* de erro, ou o espaço *inter-frame*, que são sete *bits* que significam o fim de uma mensagem. Os *bit stuffing* são removidos pelo controlador do nó receptor antes dos dados serem encaminhados para a aplicação [10].

Com esta lógica, um *frame* de erro ativo é composto por seis *bits* dominantes que violam a regra *bit stuffing*. Isso é interpretado como um erro por todos os nós que podem gerar seus próprios *frames* de erros. Isto significa que um *frame* de erro pode ser gerado a partir do original, assim, o *frame* de erros resultantes pode ter de seis *bits* a doze *bits* com todas as respostas. Este *frame* de erro é seguido por um delimitador de campo de oito *bits* recessivos, que é o período ocioso do barramento antes da mensagem corrompida ser retransmitida. É importante salientar que a mensagem retransmitida ainda tem de lutar para a arbitragem no barramento [10].

4.1.10 Relação Entre Taxa de Transmissão e Comprimento de Rede CAN

O maior problema físico do barramento CAN se refere à técnica de arbitragem de bit, que exige que onda frontal do primeiro *bit* de uma mensagem alcance todos os nós mais remotos em uma rede e volte ao transmissor antes que o este *bit* seja substituído por outro pelo próprio transmissor. Com esta limitação, a duração máxima de um *bit* e a taxa de sinalização são determinados por parâmetros de rede, tais como comprimento e impedância.

Os fatores a serem considerados no projeto de rede são: atraso de propagação (≈ 5 ns/m) típico de cabo de par trançado de barramento, perda de amplitude de sinal devido à impedâncias do cabo, e impedância de entrada do transceptores do barramento. Sob análise rigorosa, as variações entre os diferentes osciladores em um sistema também precisam ser contabilizadas como ajustes na taxa de sinalização e de um barramento. As taxas máximas típicas de sinalização alcançadas pelo barramento CAN em relação ao comprimento de rede estão listadas na Tabela 9.

Tabela 9: Taxas máximas de sinalização de uma rede CAN em relação a seu comprimento.

Comprimento da Rede CAN (m)	Taxa Máxima de Sinalização (kbps)
30	1000
100	500
250	250
500	125
1000	62,5

4.2 A Arquitetura de Atualização de Variáveis e Comunicação CAN Implementada

O objetivo da estrutura de comunicação de dados usando a rede CAN é a atualização de forma contínua de variáveis presentes nos diversos nós do barramento CAN. As variáveis geridas pela rede CAN assumem o conceito de variáveis de barramento, ou variáveis de rede, ou seja, a mesma informação está presente nos diversos nós e é atualizada continuamente através da rede CAN. Assim, as variáveis como velocidade das rodas são atualizadas por uma tarefa em um nó e algum tempo depois a mesma informação poderá ser lida por outra tarefa em outro nó. Esta funcionalidade é somente possível devido ao modo de endereçamento da rede CAN, que atribui identificador para as mensagens e não para os nós de rede.

Outra característica é a transparência do método de atualização. Assim, as tarefas que usam as informações de variáveis de barramento não têm qualquer participação na atualização

destas, ou seja, o processo de atualização das variáveis de barramento ocorre de modo transparente às demais rotinas e processos. Esta arquitetura pode ser utilizada com ou sem o uso de um *kernel* ou sistema de escalonamento de tarefas. Contudo, para otimização do recurso computacional disponível e para garantir resposta em tempo real é necessário o uso de um sistema de tempo real. Neste projeto utilizou-se o µC-OS II que será detalhado no Capítulo 5.

Esta arquitetura permite também que tarefas que compartilham decisões e informações críticas não estejam sendo processadas no mesmo microcontrolador. Assim, as tarefas de controle de um processo ou sistema podem estar hospedas em diversos microntroladores diferentes. Isso permite inúmeras vantagens no que tange à segurança, custo e modularização.

Como mencionado no Capítulo 2, o controlador CAN usado neste trabalho é o MCP2515 com encapsulamento tipo SOIC, e o transceiver CAN é o MCP2551 também com encapsulamento tipo SOIC. Estes são responsáveis pela operação do protocolo CAN na camada de controle de dados e na camada física respectivamente (Figura 77). O projeto conceitual da interação entre MCP2551, MCP2515, rotinas de controle das variáveis e demais tarefas de um nó de rede está ilustrado na Figura 83.

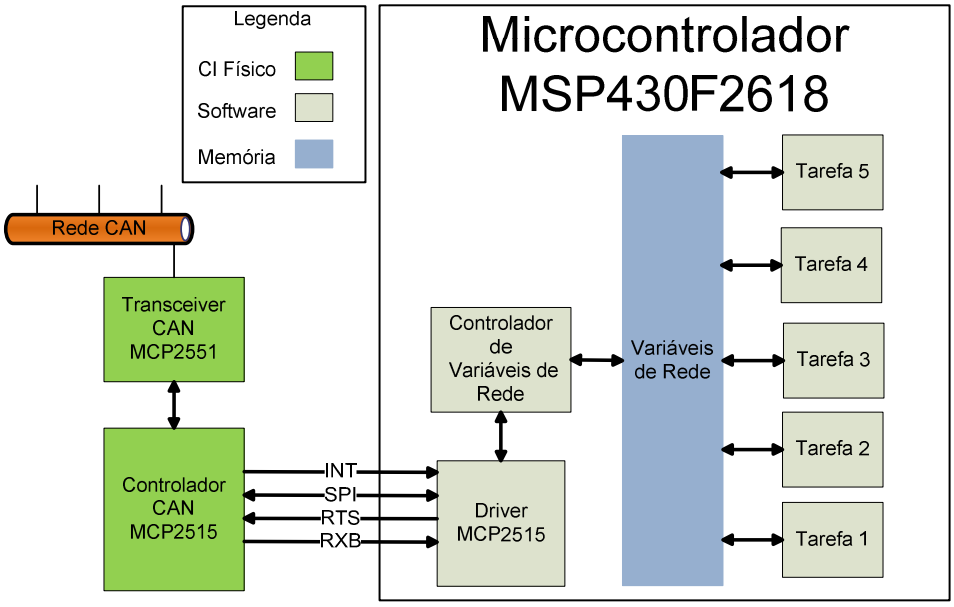


Figura 83: Projeto conceitual do método de atualização das variáveis de barramento usando rede CAN.

Na Figura 83 estão ilustradas as relações que compõe a arquitetura de variáveis de barramento. Nota-se a existência de CIs físicos, tarefas e memória. Os CIs físicos são o controlador CAN MCP2515 e o *transceiver* CAN MCP2551. A memória é a própria memória do microcontrolador e as tarefas são as diversas rotinas executadas no microcontrolador. Entre as tarefas, duas são destacadas:

- *Driver* MCP2515 ou apenas *Driver*– responsável pela comunicação direta com o controlador CAN.
- Controlador de Variáveis de Rede – responsável pela atualização e gestão de modo geral das variáveis de barramento.

A comunicação entre o *driver* e o controlador CAN é realizada por intermédio de uma interface SPI™ [22] e alguns pinos de controle especiais que se conectam diretamente aos pinos do microcontrolador (Figura 39). Estes pinos são:

- INT – pino de interrupção do controlador CAN. Este pino sinaliza ao microcontrolador quando um evento de interrupção, previamente configurado, ocorreu no controlador CAN;
- SPI – interface de comunicação SPI™ [22] utiliza 4 pinos, sendo o CS, SCK, SI e SO;
- RTS – conjunto de 3 pinos, sendo TX0RTS, TX1RTS e TX2RTS. Têm a função de sinalizar que os três *buffers* de transmissão do controlador CAN, TX0BF, TX1BF e TX2BF estão prontos para transmitir;
- RXB – conjunto de dois pinos, sendo RX0BF e RX1BF. Têm a função de sinalizar para o microcontrolador que os *buffers* de recepção contem novo conteúdo;

Todas as ligações físicas entre microcontrolador, controlador CAN e *transceiver* CAN foram abordadas no Capítulo 2. A seguir, serão detalhados cada um os dispositivos, físicos ou *software* (tarefas), que compõe a arquitetura representada na Figura 83.

4.2.1 O *Transceiver* CAN MCP2515

O MCP2551 é um *transceiver* CAN de alta velocidade e tem como função a interface entre o protocolo CAN e o controlador do barramento. O MCP2551 permite transmitir e receber mensagens CAN e é totalmente compatível com a norma ISO-11898 [45], incluindo os requisitos de 24 V. Pode operar a velocidades de até 1 $Mbps$. Normalmente, cada nó de um sistema CAN deve ter um dispositivo como este para converter os sinais digitais gerados por um controlador CAN em sinais adequados para transmissão através dos cabos do barramento (saída diferencial). Ele também fornece uma proteção entre o controlador CAN e altas tensões que podem ser geradas no barramento CAN por fontes externas (EMI, ESD, transientes elétricos, etc.). A Figura 84 ilustra o esquemático do circuito interno e pinos externos do MCP2551 [28].

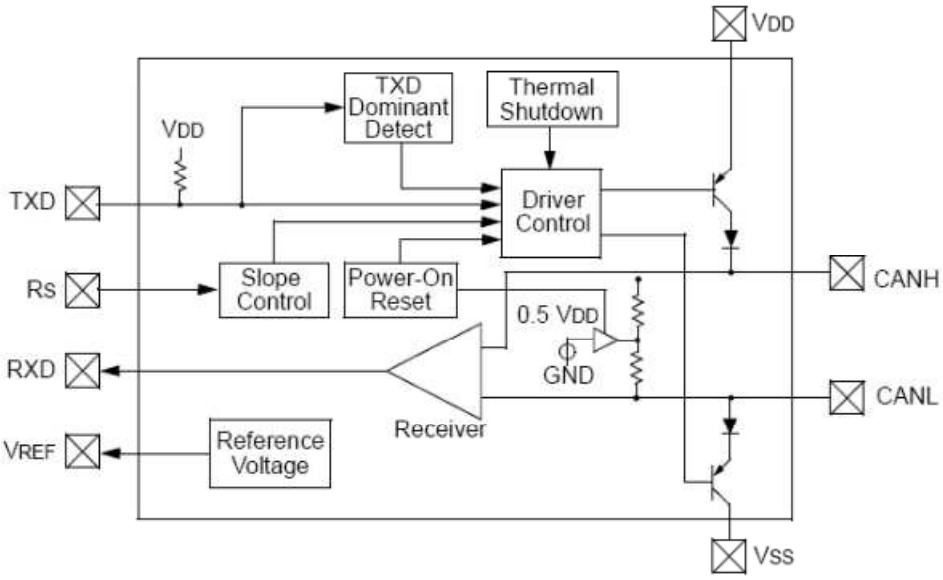


Figura 84: Diagrama de ligação interna dos pinos do MCP2551 e seus circuitos internos.

Suas principais características são:

- Suporta uma operação em até 1Mbps;
- Implementa a norma ISO-11898 [45] camada física;
- Adequado para sistemas de 12V e 24V;
- Detecção de falta à terra (permanente dominante) na entrada TXD
- *Power-on reset* e proteção *brown-out* [28];

- Um evento *brown-out* em um nó não perturba o barramento CAN
- Baixa corrente de consumo no modo *Standby;*
- Proteção contra danos causados por curto-circuito;
- Proteção contra transientes de alta tensão;
- Proteção térmica com desligamento automático do dispositivo;
- Até 112 nós podem ser conectados;
- Alta imunidade a ruídos devido ao barramento diferencial;
- Faixas de Temperatura: Industrial (I): -40 ° C a +85 ° C - *Extended* (E): -40 ° C a +125 ° C

Como mencionado anteriormente, o barramento CAN tem dois estados: dominante e recessivo. Um estado dominante ocorre quando a tensão diferencial entre CANH e CANL é maior do que uma tensão definida (por exemplo, 1,2 V) (Figura 78). O estado recessivo ocorre quando a tensão diferencial é menor que uma tensão definido (normalmente 0 V)(Figura 78). Os estados dominante e recessivo correspondem ao estado de baixa e alta do pino de entrada TXD, respectivamente. No entanto, um estado dominante iniciado por outro nó CAN vai substituir uma estado recessivo no barramento CAN [28].

O MCP2551 tem capacidade de operar em barramento CAN com uma carga mínima de 45 Ω. Assim, este permite um máximo de 112 nós conectados ao mesmo barramento, sendo que, a resistência da entrada diferencial mínima para o MCP2515 é 20 $k\Omega$ e um resistor de terminação nominal de 120 Ω [28].

O pino de saída RXD reflete a tensão do barramento diferencial entre CANH e CANL. Os estados de baixa e alta do pino de saída RXD correspondem aos estados dominantes e recessivos do barramento CAN, respectivamente.

4.2.2 O Controlador CAN MCP2515

MCP2515 *Microchip Technology* [29] é um *stand-alone Controller* para a CAN que implementa a especificação CAN 2.0B. Ele implementa em *hardware* todas as funcionalidades e serviços oferecidos pela rede CAN. Possui duas máscaras e seis filtros de

aceitação que são utilizados para filtrar mensagens indesejadas, assim, reduzindo o *overhead* do microcontrolador. O MCP2515 faz interface com o microcontrolador através de uma interface SPI ™ [22] e por alguns pinos de controle específicos. Suas características principais são [29]:

- Implementa o padrão CAN V2.0B com taxa máxima de 1 $Mbps$:
- Comprimento de dados de 0 a 8 *bytes* por pacote;
- Implementa todos os tipos de mensagens CAN nos formatos *standard* e *Extended;*
- Contém dois *buffers* de recepção (RXBF) e três de transmissão (TXBF);
- Contém duas mascaras e seis filtros de aceitação de mensagem;
- SPI ™ de alta velocidade (10 MHz) e modos de operação 0,0 e 1,1;
- Pino de interrupção INT que permite sinalizar ao microcontrolador que um evento, previamente configurado, ocorreu;
- Pinos de RTS (*Request To Send*) que permitem solicitar a transmissão do conteúdo de cada um dos três *buffers* de transmissão de modo independente;
- Pinos de RXBF (*Receivers Buffers*) que permitem sinalizar ao microcontrolador de maneira independente que os *buffers* de recepção possuem novas informações;
- Tecnologia CMOS de baixo consumo;
- Funciona a partir de 2,7 V a 5,5V;
- Consumo de 5 mA de corrente no modo ativo (típico) e 1 mA de corrente no modo de espera (típico) (modo Sleep);
- Temperaturas suportadas: Industrial (I): -40 ° C a +85 ° C, e *Extended* (E): -40 ° C a +125 ° C

O MCP2515 é um controlador CAN desenvolvido para simplificar aplicações que requerem interface com um barramento CAN. Um diagrama de blocos simples do MCP2515 é mostrado na Figura 85. O dispositivo consiste em três blocos principais [29]:

- O bloco do protocolo SPI;

- O módulo CAN, que inclui a implementação do protocolo CAN, máscaras, filtros e *buffers* de transmissão e de recepção;
- A lógica de controle e registros que são usados para configurar o dispositivo e o seu funcionamento;

A seguir serão detalhados estes blocos:

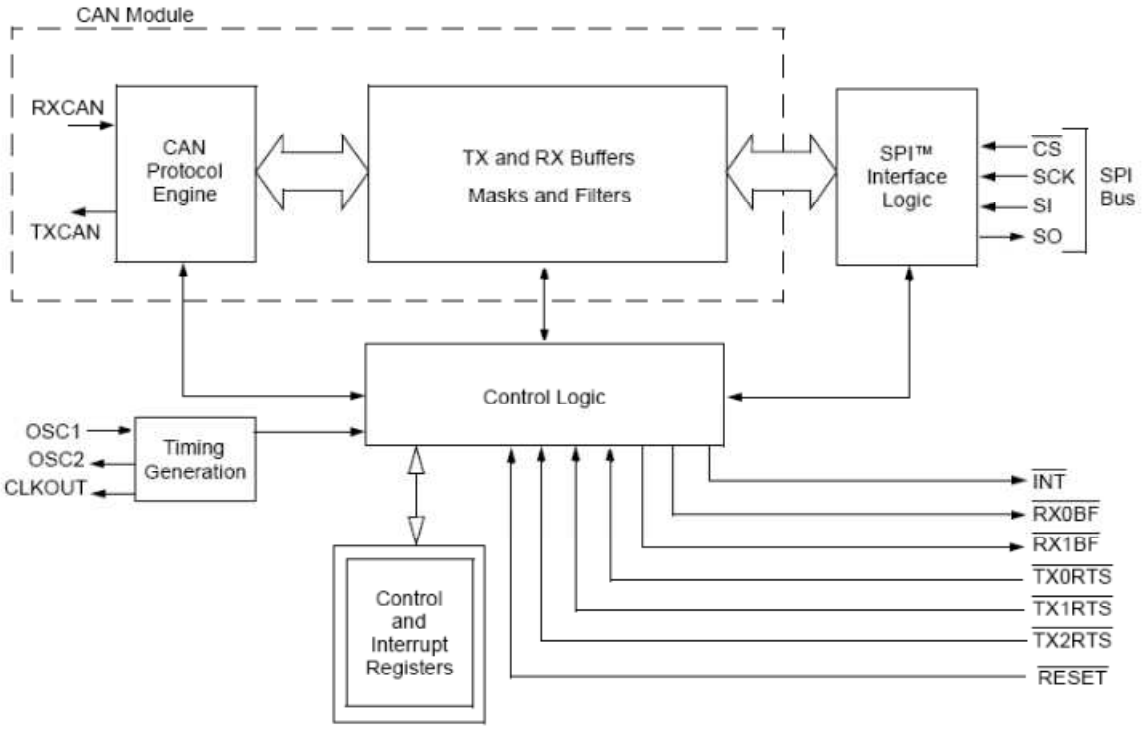


Figura 85: Diagrama de blocos ilustrativo dos sistemas internos do MCP2515.

4.2.2.1 A Interface SPI do MCP2515

Como mencionado anteriormente, a interface do MCP2515 com o microcontrolador é feita através de uma interface serial síncrona SPI. Através desta é possível ler e escrever em qualquer registro do MCP2515, enviar e receber comandos do microcontrolador e verificar o *status* do controlador CAN. Os registros do MCP2515 serão detalhados adiante.

A interface SPI é um protocolo de comunicação síncrono que usa a arquitetura mestre escravo, ou seja, um mestre envia um sinal de controle chamado de *clock* juntamente com outro sinal chamado CS (*chip select*) para os escravos. Estes permanecem inativos até

que o mestre o selecione e envie o sinal de *clock*. No MCP2515 os sinais de controle SPI estão disponíveis nos pinos:

- $\overline{CS}$ - *chip select* quando 0 seleciona o MCP2515 e quando 1 este permanece inativo;
- SCK - entrada de *clock* SPI do MCP2515;
- SO - *slave-out*, saída de comunicação SPI do MCP2515;
- SI – *slave-in* entrada de dados SPI para o MCP2515;

O MCP2515 somente opera em modo escravo, assim, obrigatoriamente o microcontrolador é o mestre SPI. As formas de onda dos sinais SPI estão presentes nas para o fluxo do microcontrolador para o MCP2515 estão ilustrados na Figura 86, e o fluxo MCP2515 para microcontrolador está ilustrado na Figura 87. A ligação física entre microcontrlador e MCP2515 está representada na Figura 39.

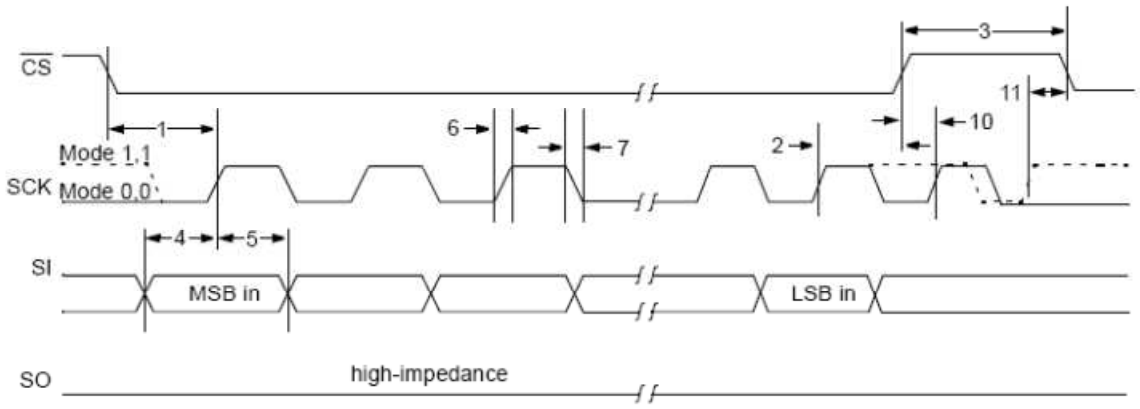


Figura 86: Temporização dos sinais SPI do MCP2515 no modo recepção.

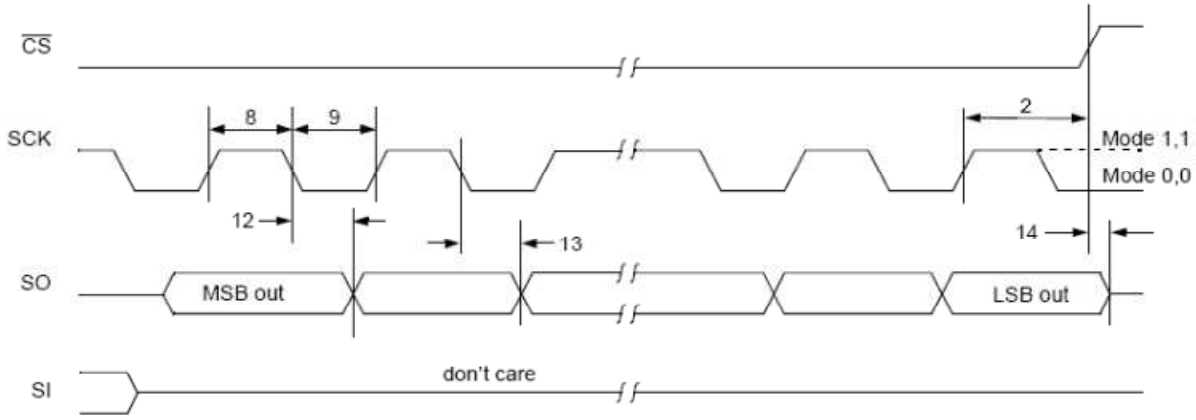


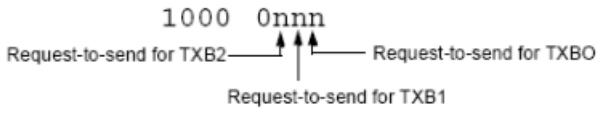
Figura 87: Temporização dos sinais SPI do MCP2515 no modo transmissão.

4.2.2.2 Comandos SPI do MCP2515

Visando prover uma comunicação segura e eficiente com o microcontrolador o MCP2515 implementa um conjunto de instruções e comandos resumidos e objetivos. Estas instruções e comandos são protocolados sobre a interface SPI. Estes comandos estão descritos na Tabela 10.

Tabela 10: Tabela dos comandos SPI do MCP2515.

Nome do comando	Formato Binário do comando	Descrição
RESET	1100 0000	Reiniciar os registros internos para o estado *default* e coloca o MCP2515 no estado de configuração
READ	0000 0011	Lê os dados de um registro selecionado por um endereço subseqüente
Read RX *Buffer*	1001 0nm0	Lê um *buffer* de recepção inteiro reduzindo a sobrecarga de um comando normal de leitura. Coloca o ponteiro de endereço em um dos quatro locais como indicado por 'n, m' na Figura 89.
WRITE	0000 0010	Escreve um dado em um registro devido por um endereço subseqüente

Load* TX *Buffer	0100 0abc	Carrega os *buffers* de transmissão reduzindo a sobrecarga do uso do comando de escrita normal. O ponteiro de escrita é atualizado com o valor definido por 'a,b,c' que estão explicitados Figura 91.
RTS (Request to send)	1000 0nnn	Comando que dá inicio ao processo de transmissão de um dos três *buffers* 
Read *Status*	1010 0000	Comando que lê um *byte* de *status* gerais definido na Figura 94
RX *Status*	1011 0000	Comando que lê um *byte* de *status* gerais da recepção. A definição deste *byte* está na Figura 95
***Bit* Modify**	0000 0101	Permite ao usuário definir ou limpar *bits* individuais em um registro particular.

As figuras ilustram o desenvolvimento temporal de cada um dos comandos descritos da Tabela 10.

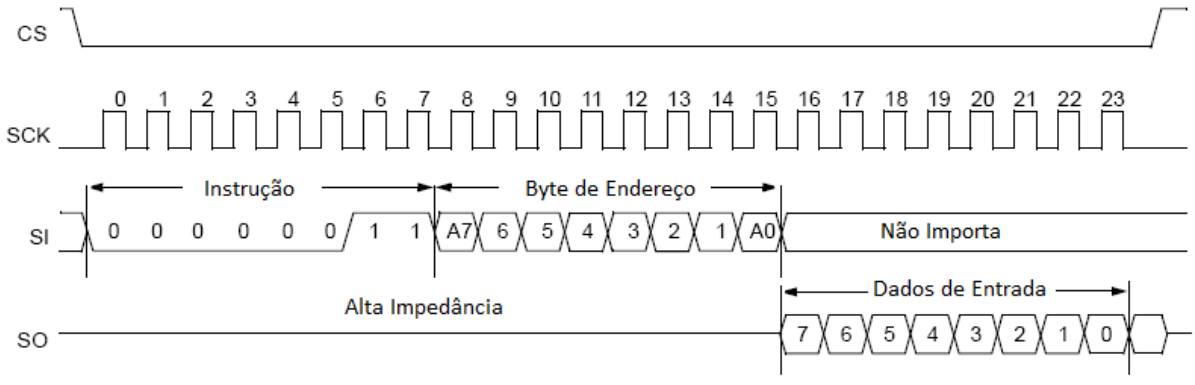
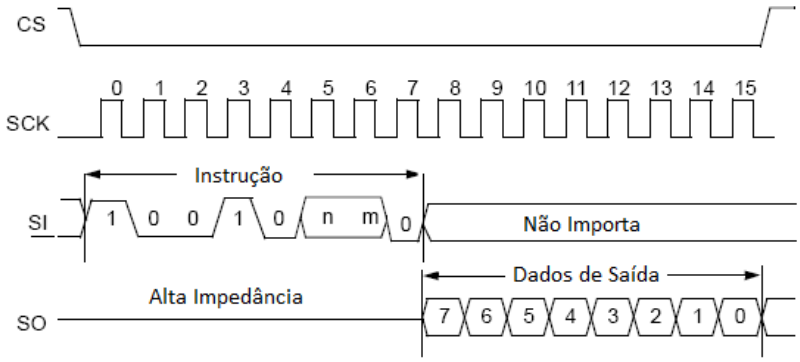


Figura 88: Temporização do Comando READ do MCP2515.



n	m	Address Points to	Address
0	0	Receive Buffer 0, Start at RXB0SIDH	0x61
0	1	Receive Buffer 0, Start at RXBD0	0x66
1	0	Receive Buffer 1, Start at RXB1SIDH	0x71
1	1	Receive Buffer 1, Start at RXBD1	0x76

Figura 89: temporização do comando Read RX *Buffer* do MCP2515.

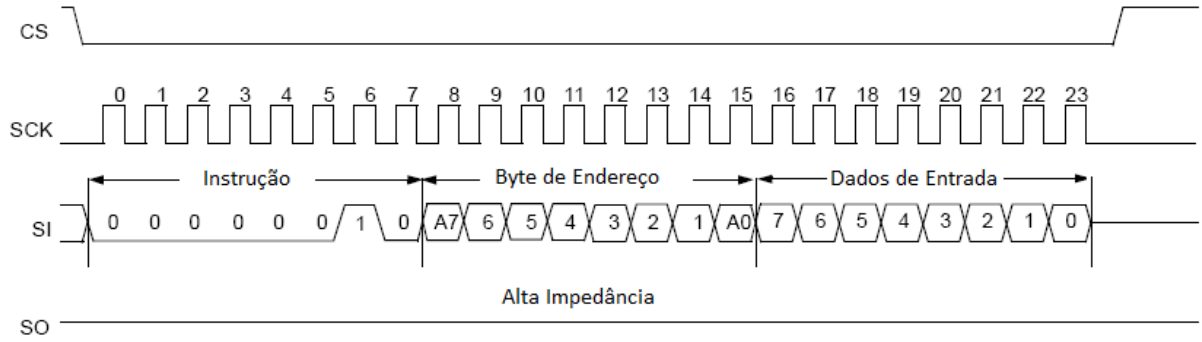
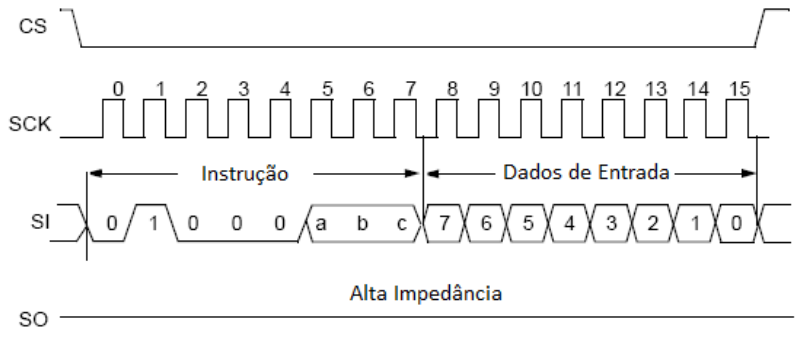


Figura 90: Temporização do comando WRITE do MCP2515.



a	b	c	Address Points to	Addr
0	0	0	TX buffer 0, Start at TXB0SIDH	0x31
0	0	1	TX buffer 0, Start at TXB0D0	0x36
0	1	0	TX buffer 1, Start at TXB1SIDH	0x41
0	1	1	TX buffer 1, Start at TXB1D0	0x46
1	0	0	TX buffer 2, Start at TXB2SIDH	0x51
1	0	1	TX buffer 2, Start at TXB2D0	0x56

Figura 91: Temporização do comando *Load* TX *buffer* do MCP2515.

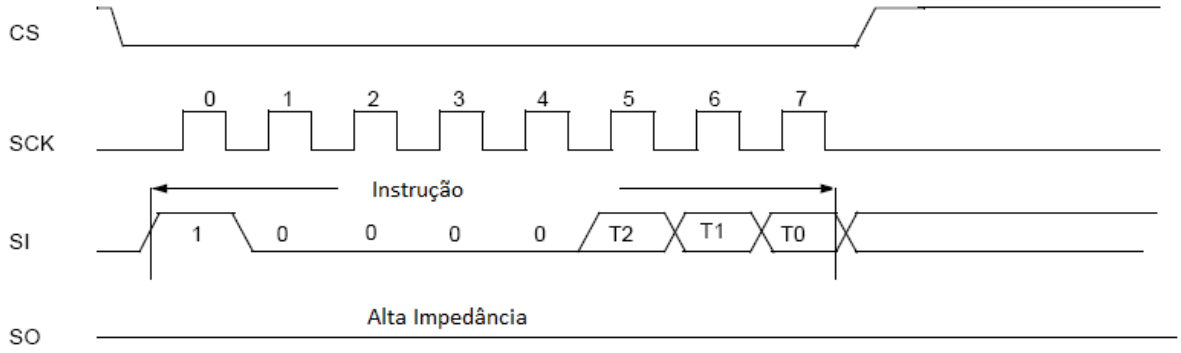


Figura 92: Temporização do comando RTS do MCP2515.

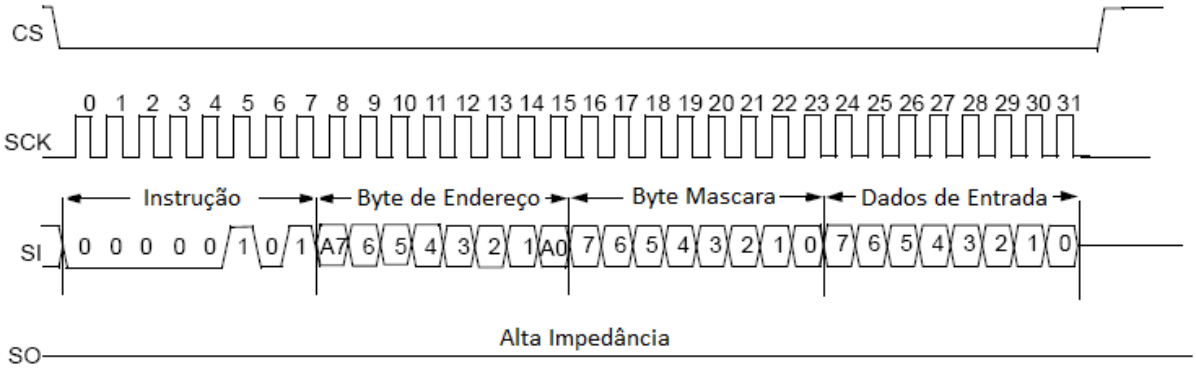


Figura 93: Temporização do comando *Bit* Modify do MCP2515.

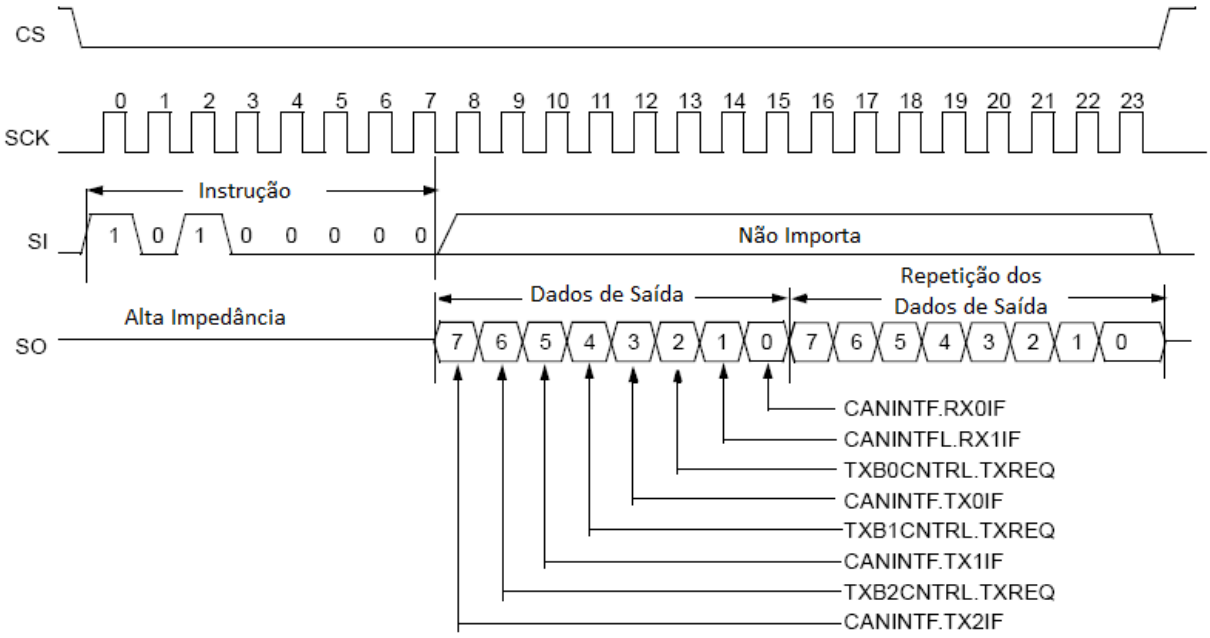
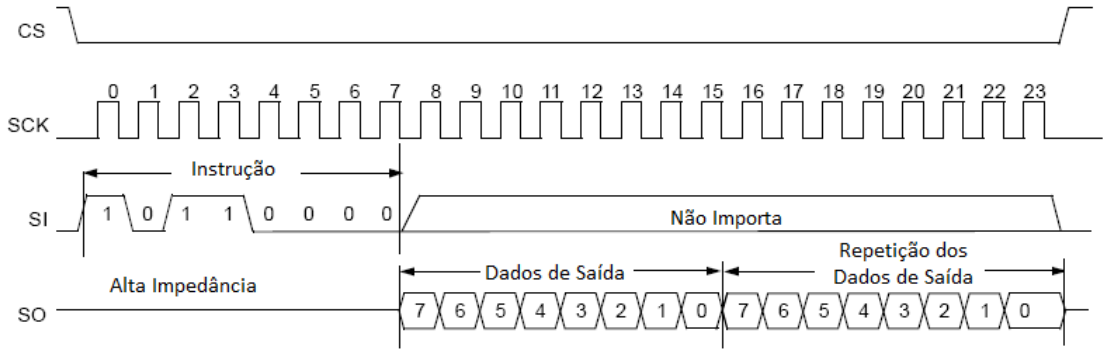


Figura 94: Temporização do comando Read *Status* do MCP2515.



7	6	Msg Recebida
0	0	Sem Msg Recebida
0	1	Msg no RXB0
1	0	Msg no RXB1
1	1	Msg em ambos os Buffers

4	3	Tipo de Msg Recebida
0	0	Standard data frame
0	1	Standard remote frame
1	0	Extended data frame
1	1	Extended remote frame

2	1	0	Filtro Associado
0	0	0	RXF0
0	0	1	RXF1
0	1	0	RXF2
0	1	1	RXF3
1	0	0	RXF4
1	0	1	RXF5
1	1	0	RXF0 (rollover to RXB1)
1	1	1	RXF1 (rollover to RXB1)

Figura 95: temporização do comando RX *Status* do MCP2515.

4.2.2.3 O Módulo de Controle CAN do MCP2515

O módulo pode manipular todas as funções para receber e transmitir mensagens no barramento CAN. Para transmitir uma mensagem, primeiro deve-se carregar um dos três *buffers* de transmissão de mensagens. A transmissão pode ser iniciada usando-se um dos três registros de controle de transmissão: TXB0CTRL, TXB1CTRL e TXB2CTRL, sendo um para cada *buffer*. Pode-se usar para transmissão comando específicos através da interface SPI ou usando um dos pinos TX0RTS, TX1RTS e TX2RTS do MCP2515. O estado e os erros podem ser verificados através da leitura dos registros apropriados. Qualquer mensagem detectada no barramento CAN é verificada quanto a erros e em seguida comparada com os filtros para verificação dos critérios de aceitação da mensagem. A mensagem somente é movida para um dos *buffers* de recepção se estiver livre de erros e se os critérios de aceitação com os filtros e mascaras forem validados.

O principal do modulo de controle CAN é uma Máquina de Estado Finito (MEF). A MEF funciona como um seqüenciador que controla o fluxo de dados seqüencial entre a mudança de estado nos registros TX e RX. Controla também a lógica de gestão de erro e o

fluxo de dados paralelos entre a mudança dos pinos TX e RX. A MEF garante que os processos de recepção, transmissão, arbitragem, transporte e sinalização de erro são realizados de acordo com o protocolo CAN. A retransmissão automática de mensagens também é tratada pela MEF.

A verificação de redundância ou CRC (*Cyclic Redundancy Check code*) ocorre na MEF da seguinte maneira: o registro CRC gera um código com o mesmo nome que é transmitido com a mensagem CAN. Quando a mensagem é recebida o receptor gera novamente o mesmo código que é então comparado com o transmitido. Assim, se ambos os códigos forem iguais, a mensagem é valida; caso contrário, a mensagem é descartada e um *frame* de erro e transmitido.

Existem dois registros de contagem e erro internos ao MCP2515: o REC (*Receive Erro Counter*) e TEC (*Transmit Erro Counter*) que funcionam, respectivamente, contando os erros de recepção e transmissão ocorridos no barramento.

O MCP2515 dispõe de uma lógica de temporização de *bit* (BTL) que monitora o barramento e gera toda a temporização necessária para o envio e recebimento dos *bits* CAN. É a BTL que gera a taxa de comunicação à medida que faz a divisão de tempo de cada *bit* que é transmitido e recebido.

Todos os nós CAN devem operar na mesma taxa de comunicação. O protocolo CAN é do tipo NRZ (*Non Return to Zero*), ou seja, sem retorno para zero, e a taxa nominal f_{bit} , chamada também de NBR (Nominal *Bit* rate) ou taxa de *bit* nominal, é determinada pela equação abaixo:

$$NBR = f_{bit} = \frac{1}{t_{Bit}}$$

Equação 13

Sendo:

$$t_{bit} = t_{SyncSeg} + t_{PropSeg} + t_{PS1} + t_{PS2}$$

Equação 14

Onde:

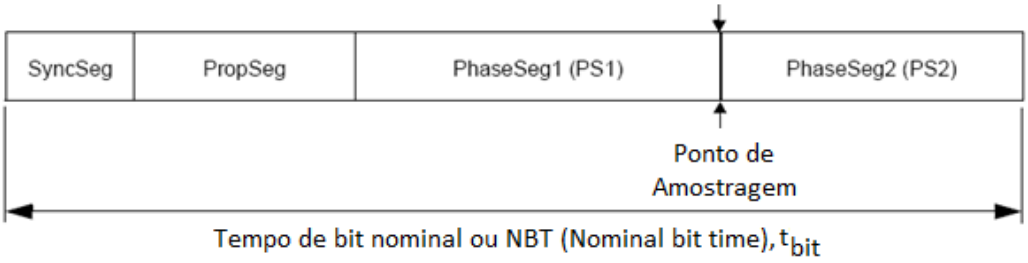


Figura 96: Representação temporal do tempo de *bit* nominal t_{bit} ou NBT do protocolo CAN.

Cada um dos tempos representados acima podem ser divididos em quantuns de tempo ou TQ, que é o período mínimo de temporização interna do MCP2515. O TQ tem ligação direta com o *clock* do CI e é definido pela equação:

$$TQ = 2 \times BRP \times T_{OSC} = 2 \times \frac{BRP}{F_{OSC}}$$

Equação 15

Onde:

- BRP é o registro de pré escala do *clock* do MCP2515
- T_{OSC} *e* F_{OSC} são respectivamente, o período e a frequência do cristal oscilador acoplado ao MCP2515.

Assim, a representação no tempo dos sinais oriundos do BRP, TQ e demais tempos está ilustrada na Figura 97.

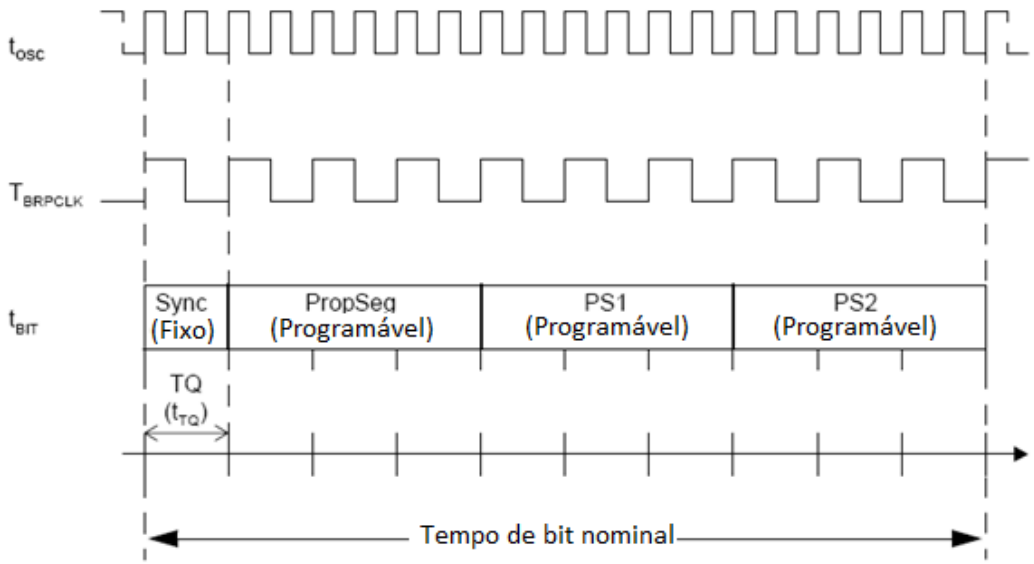


Figura 97: Representação dos sinais TQ, *clock* do BRP em relação do tempo de *bit* nominal do controlador CAN.

Algumas recomendações e limitações são impostas pelo MCP2515, sendo:

- $t_{SyncSeg}$ tem valor fixo em 1 TQ;
- $t_{PropSeg}$ é programável e pode assumir valores de 1 a 8 TQ;
- t_{PS1} é programável e pode assumir valores de 1 a 8 TQ;
- t_{PS2} é programável e pode assumir valores de 2 a 8 TQ;
- Largura de sincronização (SJW) é uma faixa de ajuste de amostragem que é configurável com 1 a 4 TQ.

4.2.2.4 A Lógica de Controle e Registros do MCP2515

A lógica de controle do MCP2515 gerencia todos os registros do CI. É responsável também por sua configuração, gestão de erros, transmissão, recepção e geração de interrupções para o microcontrolador.

Os registros são endereçados por um *byte* de endereço e tem comprimento fixado também em um *byte*. Assim, a tabela de endereços dos registros do controlador CAN é:

Tabela 11: Tabela de endereços dos registros do MCP2515.

***Bits* Menos Significativos do Endereço**	***Bits* Mais Significativos do Endereço**							
	x000 xxxx	x001 xxxx	x010 xxxx	0011 xxxx	x100 xxxx	x101 xxxx	x110 xxxx	x111 xxxx
0000	RXF0SIDH	RXF3SIDH	RXM0SIDH	TXB0CTRL	TXB1CTRL	TXB2CTRL	RXB0CTRL	RXB1CTRL
0001	RXF0SIDL	RXF3SIDL	RXM0SIDL	TXB0SIDH	TXB1SIDH	TXB2SIDH	RXB0SIDH	RXB1SIDH
0010	RXF0EID8	RXF3EID8	RXM0EID8	TXB0SIDL	TXB1SIDL	TXB2SIDL	RXB0SIDL	RXB1SIDL
0011	RXF0EID0	RXF3EID0	RXM0EID0	TXB0EID8	TXB1EID8	TXB2EID8	RXB0EID8	RXB1EID8
0100	RXF1SIDH	RXF4SIDH	RXM1SIDH	TXB0EID0	TXB1EID0	TXB2EID0	RXB0EID0	RXB1EID0
0101	RXF1SIDL	RXF4SIDL	RXM1SIDL	TXB0DLC	TXB1DLC	TXB2DLC	RXB0DLC	RXB1DLC
0110	RXF1EID8	RXF4EID8	RXM1EID8	TXB0D0	TXB1D0	TXB2D0	RXB0D0	RXB1D0
0111	RXF1EID0	RXF4EID0	RXM1EID0	TXB0D1	TXB1D1	TXB2D1	RXB0D1	RXB1D1
1000	RXF2SIDH	RXF5SIDH	CNF3	TXB0D2	TXB1D2	TXB2D2	RXB0D2	RXB1D2
1001	RXF2SIDL	RXF5SIDL	CNF2	TXB0D3	TXB1D3	TXB2D3	RXB0D3	RXB1D3
1010	RXF2EID8	RXF5EID8	CNF1	TXB0D4	TXB1D4	TXB2D4	RXB0D4	RXB1D4
1011	RXF2EID0	RXF5EID0	CANINTE	TXB0D5	TXB1D5	TXB2D5	RXB0D5	RXB1D5
1100	BFPCTRL	TEC	CANINTF	TXB0D6	TXB1D6	TXB2D6	RXB0D6	RXB1D6
1101	TXRTSCTRL	REC	EFLG	TXB0D7	TXB1D7	TXB2D7	RXB0D7	RXB1D7
1110	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT	CANSTAT
1111	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL	CANCTRL

Tabela 12: Tabela contendo os *bits* de cada registro do controlador CAN.

Registro	**Endereço (Hex)**	***Bit* 7**	***Bit* 6**	***Bit* 5**	***Bit* 4**	***Bit* 3**	***Bit* 2**	***Bit* 1**	***Bit* 0**	**Valor default no POR/RST**
BFPCTRL	0C	—	—	B1BFS	B0BFS	B1BFE	B0BFE	B1BFM	B0BFM	xx00 0000
TXRTSCTRL	0D	—	—	B2RTS	B1RTS	B0RTS	B2RTSM	B1RTSM	B0RTSM	xxxx x000
CANSTAT	xE	OPMOD2	OPMOD1	OPMOD0	—	ICOD2	ICOD1	ICOD0	—	100x 000x
CANCTRL	xF	REQOP2	REQOP1	REQOP0	ABAT	OSM	CLKEN	CLKPRE1	CLKPRE0	1110 0111
TEC	1C	Valor do TEC								0000 0000
REC	1D	Valor do REC								0000 0000
CNF3	28	SOF	WAKFIL	—	—	—	PHSEG22	PHSEG21	PHSEG20	00xx x000
CNF2	29	BTLMODE	SAM	PHSEG12	PHSEG11	PHSEG10	PRSEG2	PRSEG1	PRSEG0	0000 0000
CNF1	2A	SJW1	SJW0	BRP5	BRP4	BRP3	BRP2	BRP1	BRP0	0000 0000
CANINTE	2B	MERRE	WAKIE	ERRIE	TX2IE	TX1IE	TX0IE	RX1IE	RX0IE	0000 0000
CANINTF	2C	MERRF	WAKIF	ERRIF	TX2IF	TX1IF	TX0IF	RX1IF	RX0IF	0000 0000
EFLG	2D	RX1OVR	RX0OVR	TXBO	TXEP	RXEP	TXWAR	RXWAR	EWARN	0000 0000
TXB0CTRL	30	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	x000 x000
TXB1CTRL	40	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	x000 x000
TXB2CTRL	50	—	ABTF	MLOA	TXERR	TXREQ	—	TXP1	TXP0	x000 x000
RXB0CTRL	60	—	RXM1	RXM0	—	RXRTR	BUKT	BUKT	FILHIT0	x00x 0000
RXB1CTRL	70	—	RSM1	RXM0	—	RXRTR	FILHIT2	FILHIT1	FILHIT0	x00x 0000

4.3 O driver do Controlador CAN

O *driver* MCP2515 é a tarefa que controla o fluxo de comunicação entre microcontrolador e controlador CAN. É executado pelo microcontrolador e usa seus periféricos como meio físico para gestão da comunicação, interrupções e pinos de controle do controlador CAN.

O *driver* MCP2515 tem o objetivo de prover uma interface entre o controlador de variáveis e o MCP2515. Como ilustrado anteriormente, o controlador CAN possui dois *buffers* de recepção e três *buffers* de transmissão, então, com estes recursos, e se fosse possível garantir que o controlador de variáveis sempre executasse sua rotina antes do MCP2515 sobrescreve os dados dos *buffers* de recepção, não seria necessária a implementação do *driver* do MCP2515, visto que os recursos do controlador CAN seriam suficientes. Contudo, não é possível fazer tal inferência, isso devido ao fato da arquitetura implementada usar um *kernel* de tempo real e, assim, pode ocorrer uma situação típica de tal *kernel*: a inversão de prioridade, que neste caso iria atrasar a retirada dos dados dos *buffers* do MCP2515, o que pode acarretar sobre-escrita dos dados e consequente perda de informação. Este situação é muito crítica, visto que para o emissor da informação ela foi recebida; no entanto, o receptor a ignorou.

O similar ocorreria também da transmissão, visto que, com apenas três *buffers* disponíveis, se quatro mensagens estivessem prontas para serem enviadas ao mesmo tempo a rotina de controle das variáveis teria de esperar para enviar a quarta, o que pode ocasionar o problema de inversão de prioridade ou o bloqueio semafórico das variáveis presentes na quarta mensagem. Esta situação deixaria a rotina de controle das variáveis de barramento, e possivelmente outras, em estado de aguardo do MCP2515.

Para eliminar o problema supracitado, foi desenvolvido o *driver* MCP2515 que é uma tarefa que objetiva atender às demandas do MCP2515 em tempo real, provendo para o controlador de variáveis de barramento *buffers* maiores de recepção, transmissão e gestão e sinalização dos erros do MCP2515.

O *driver* MCP2515 efetua três funcionalidades básicas:

- Configuração da interface SPI do microcontrolador;
- Configuração do barramento CAN;
- Rotina de comunicação CAN e verificação de *status* do MCP2515;

Estas funcionalidades serão detalhadas a seguir.

4.3.1 Configuração da Interface SPI do Microcontrolador MSP430F2618

Como mencionado anteriormente o MCP2515 utiliza para comunicação de dados uma interface SPI, na qual este é escravo e consequentemente o MSP430F2618 é mestre. O MSP430F2618 possui quatro interfaces chamadas de USCI (*Universal Serial Communication Interface*) ou Interface Serial Universal, sendo a USCI_A0, USCI_A1, USCI_B0 e USCI_B1 nas quais as USCI_A0 e USCI_A1 suportam os protocolos: UART, SPI, IrDA e LIN. As interfaces USCI_B0 e USCI_B1 suportam os protocolos: I2C e SPI. Mais detalhes sobre as interfaces USCI do MSP430F2618 em [2].

Assim, as interfaces utilizadas para comunicação com o controlador CAN foram as USCI_B0 e USCI_B1, sendo que para a Placa de Controle dos Periféricos a interface utilizada foi a USCI_B0 e, para a placa de controle, foi utilizada a USCI_B1. Nos Anexos 1 e 3 é possível encontrar os arquivos de configuração das USCIs da Placa de Processamento e da Placa de Controle dos Periféricos, respectivamente. Em ambos os casos a configuração das USCIs é:

- Taxa de sinalização de 8 $Mbps$;
- 8 *bits* de dados;
- Padrão 0,0;
- Configuração SPI a quatro pinos, sendo o microcontrolador mestre;
- Interrupções desligadas;
- MSB enviado primeiro;

As interrupções desligadas são justificadas pela taxa de sinalização, 8 $Mbps$, e durante os testes o uso de interrupções para atualização do registro de deslocamento usado para a transmissão se mostrou menos eficiente que o *polling* de verificação de termino de

transmissão. Assim, a técnica usada, descrita no Anexo 2, usa um *polling* que verifica o *bit* de *status* de atividade da interface SPI. Detalhes das conexões físicas entre os pinos do MCP2515 e MSP430 estão no Capítulo 2.

O *driver* MCP2515 também é responsável pela configuração dos pinos de interface entre o controlador CAN e microcontrolador. Estes pinos são: pino $\overline{INT}$ de interrupção do controlador CAN conectado a um pino do microcontrolador com capacidade de geração de interrupção, e pinos de controle digitais do controlador CAN ,RX0BF, RX1BF, TX0RTS, TX1RTS e TX2RTS conectados a pinos de I/O digitais do microcontrolador [2][29].

4.3.2 Configuração do Barramento CAN

A configuração do barramento se resume à determinação dos tempos mencionados na seção 4.2.2.3 que, juntos, determinam a taxa de *bits* do barramento CAN. Assim, para diversas configurações destes tempos se têm diversas velocidades de comunicação na rede CAN. Objetivando facilidades nos testes, foi criada a Tabela 13 [2][29].

Tabela 13: Tabela de configurações e tempo do barramenyto CAN.

Parâmetro	Configurações de Taxas de Sinalização MCP2515			
	1 Mbps	500 kbps	250 kbps	125 kbps
Fosc (Hz)	16.000.000	16.000.000	16.000.000	16.000.000
BRP	1	1	2	4
Tsyncseg	1	1	1	1
TprogSeg	1	3	3	3
PS1	3	6	6	6
PS2	3	6	6	6
TQ (s)	1,250E-07	1,250E-07	2,500E-07	5,000E-07
T*bit* (s)	1,000E-06	2,000E-06	4,000E-06	8,000E-06
F*bit* ou taxa (bps)	1.000.000	500.000	250.000	125.000

A taxa de sinalização foi fixada em 500 $kbps$ na rede CAN e os parâmetros configurados nos controladores CAN seguem a Tabela 13 [2][29]. Na Figura 98 está ilustrado o barramento CAN implementado na cadeira de rodas operando a 500 $kbps$.

Figura 98: Rede CAN da cadeira de rodas operando a 500 $kbps$.

4.3.3 Rotina de Comunicação CAN e Verificação de *Status* do MCP2515

O *driver* MCP2515 é responsável pela gestão de interrupção, recepção e gestão de erros do MCP2515. Também é responsável pela transmissão de mensagens quando estas se encontrarem em espera, ou seja, existe uma função chamada TXCAN, ilustrada no anexo 2, que tem a função de transmitir um mensagem CAN, contudo, como mencionado anteriormente, o MCP2515 possui apenas três *buffers* de transmissão que, se estiverem cheios no momento do uso da TXCAN, a quarta mensagem será alocada em um *buffer* de transmissão, e será transmitida quando um dos três *buffers* estiver disponível.

As interrupções do MCP2515 podem ser oriundas de: erros ou falhas, nova mensagem disponível em um dos dois *buffers* de recepção, *buffer* de transmissão vazio e se o MCP2515 saiu do estado de espera.

Para esta aplicação foram usadas apenas as interrupções de recepção e transmissão. A gestão de falhas e erros é realizada por *polling* e a interrupção de saída do estado de espera foi desativada, pois nesta aplicação o MCP2515 não é colocado neste estado.

Os pinos RX0BF e RX1BF foram configurados como saídas sinalizadoras de recepção de mensagens, e os pinos TX0RTS, TX1RTS e TX2RTS como entradas de sinais de requisição de transmissão para cada um dos três *buffers*.

O *driver* MCP2515 é executado quando uma interrupção do controlador CAN é ativada ou quando o seu temporizador de *polling* chega ao seu fim.

Na primeira situação o *driver* MCP2515 verifica qual é a interrupção. Sendo uma interrupção de nova mensagem recebida em um dos *buffers*, ele copia os dados desta nova mensagem para um *buffer* de recepção interno do microcontrolador e libera um semáforo de sinalização para o controlador de variáveis. Se a interrupção é de transmissão, ou seja, existe *buffer* de transmissão disponível, o *driver* MCP2515 verifica se existe mensagem aguardando para ser transmitida, se sim, ele a transmite, e se não ele ignora a interrupção.

O temporizador do *driver* MCP2515 é periodicamente atualizado por uma interrupção de tempo do microcontrolador, anexos 4 e 5, e tem a função de temporizar a execução da leitura dos registros de *status* do MCP2515.

O algoritmo do *driver* MCP2515 foi desenvolvido com base na técnica *State Chart* [43], que consiste na criação de máquinas virtuais de estados finitos. Esta técnica permite melhor correlação e tratamento das diversas partes de um programa. A representação da máquina de estados virtual do *driver* MCP2515 está na Figura 99.

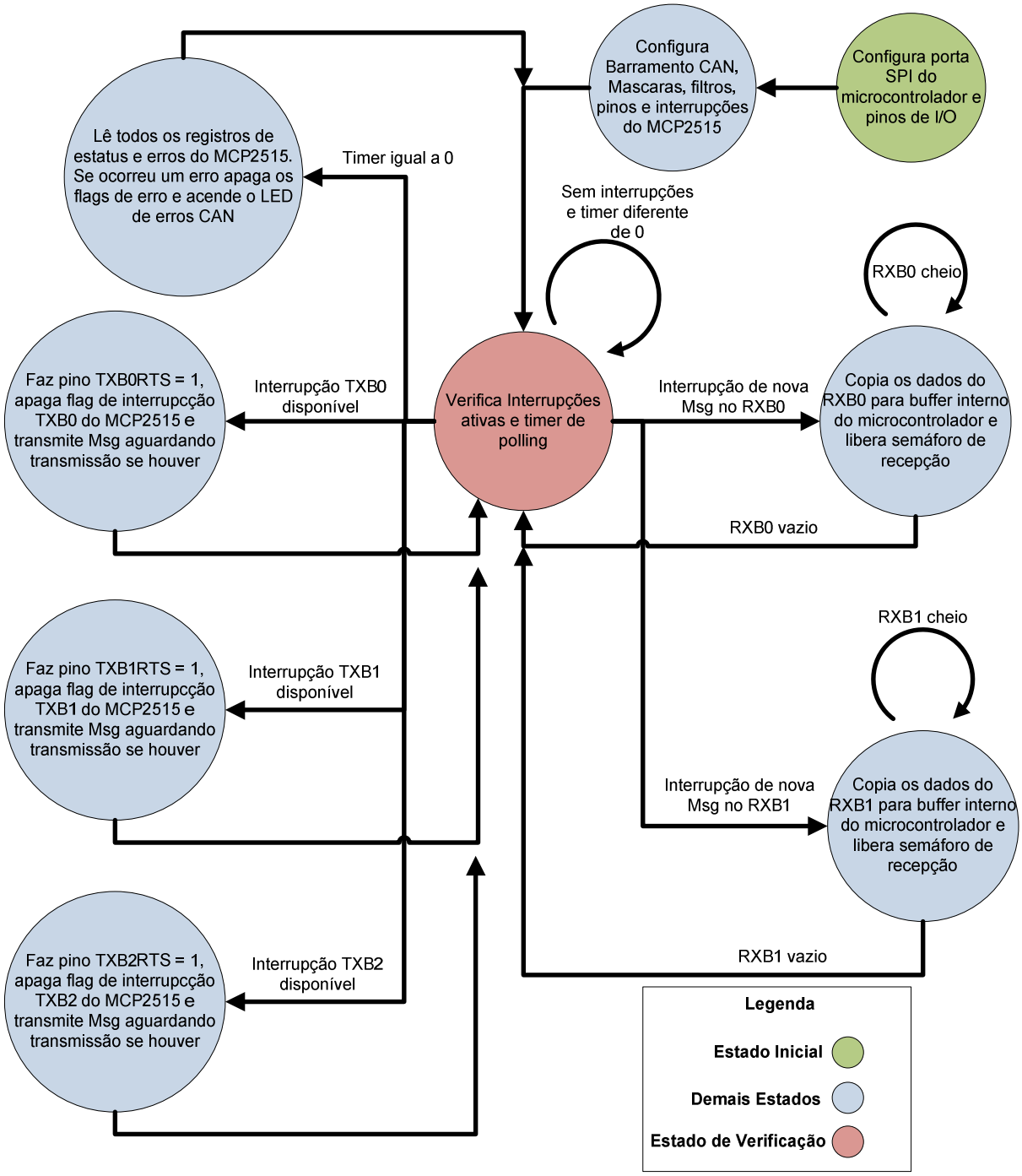


Figura 99: Ilustração da máquina de estado do *driver* MCP2515.

4.4 O Controlador de Variáveis

O controlador de variáveis de barramento é a tarefa responsável pela gestão destas variáveis. Esta tarefa é executada sempre que ocorre a recepção de uma nova mensagem, recebida e sinalizada pelo *driver* MCP2515, ou quando ocorre o estouro do temporizador de *TICK* do controlador de variáveis. Sua função básica é monitorar mensagens previamente declaradas como variáveis de barramento.

O *TICK* do controlador de variáveis é um temporizador controlado por interrupção, que tem a função de temporizar a execução da rotina de envio de RTS pelo controlador de variáveis de barramento.

Para declarar uma variável como variável de barramento, foi criada uma função chamada de *InsertOnBus*, a qual tem a função de alocar em um *buffer* de controle as informações destas mensagens. Estas informações estão destacadas na declaração da função em C abaixo:

***HANDLE_CAN_MSG InsertOnBus** (unsigned int **Id**,unsigned int **TimerTrigger**,void ***DataPointer**,char **DataLength**, char **Envia**)*

- *Id* → identificador de 11 *bits* da mensagem CAN;
- *TimerTrigger* → período de atualização da variável de barramento em unidades de *TICK* do controlador de variáveis, se 0 a variável não é atualizada;
- **DataPointer* → ponteiro para variável física;
- *DataLength* → comprimento da mensagem CAN em número de *bytes* de 0 a 8;
- *Envia* → *bit* que indica: 1 envia a mensagem no momento da chamada desta função, e 0 não envia;
- *HANDLE_CAN_MSG* → retorno da função e indica qual a posição no *buffer* de controle a variável ocupa.

Ao usar a função acima a variável passa a ser monitorada pelo controlador de variáveis. Isso implica que seu uso fica condicionado a este, ou seja, esta variável passa a ser acessada ou atualizada também pelo controlador de variáveis.

O *buffer* de controle que recebe as informações das variáveis de barramento é um *buffer* de estruturas que contém para cada mensagem os itens listados abaixo:

- Um semáforo, que tem a função de bloquear acessos a variável instáveis, ou seja, variáveis em processo de modificação;
- Um temporizador, cuja função é temporizar o envio de RTS para atualização da variável;
- Um período de atualização, que estabelece um período no qual a variável é atualizada. Se 0, a variável não é atualizada, mas é disponibilizada no barramento para leitura de outros nós;
- Um *bit* sinalizador de variável não atualizada, que tem a função de indicar que a variável não foi atualizada no período programado;
- Um *bit* sinalizador de mensagem em atualização, que indica ao controlador de variáveis que um RTS desta variável foi enviado e mensagem de retorno ainda não chegou;

Quando ocorre o evento de recepção de uma nova mensagem pelo *driver* MCP2515, este libera o semáforo de controle do controlador de variáveis, que, ao ser executado, verifica se a mensagem que foi recém recebida é monitorada por ele comparando o identificador da mensagem com os identificadores inseridos no *buffer* de controle. Se a mensagem não foi monitorada, é ignorada. Se a mensagem for monitorada, o controlador de variáveis verifica se é um RTS. Se for RTS ele envia os dados da variável identificada. Se não for RTS ele atualiza os dados da variável com os valores recebidos e reseta o *bit* de mensagem em atualização e o *bit* de variável não atualizada.

Quando ocorre um *TICK* o controlador de variáveis de barramento decrementa os temporizadores de todas as mensagens que são monitoras por ele e, se uma mensagem tem seu temporizador igualado a zero, um RTS é enviado com o identificador desta mensagem, seu temporizador é reinicializado com o período da mensagem e o *bit* de mensagem em

atualização é setado. Se o *TICK* ocorre e o temporizador de uma mensagem é igualado a zero e o *bit* de mensagem em atualização está setado, significa que ocorreu um novo *TICK* e a mensagem requerida anteriormente não chegou, ou seja, o RTS da mensagem anterior não foi atendido. Assim, o controlador de variáveis seta o *bit* sinalizador de variável não atualizado. Este mecanismo garante que as variáveis de barramento são checadas quanto a sua atualização.

Novamente foi usada a técnica *State Chart* [43] para implementar o controlador de variáveis. Portanto, foi criada uma Máquina de Estados Virtual que desempenha as funções do controlador de variáveis (Figura 100), e o programa desenvolvido em C que descreve o controlador de variáveis está disponível nos anexos 6 e 7.

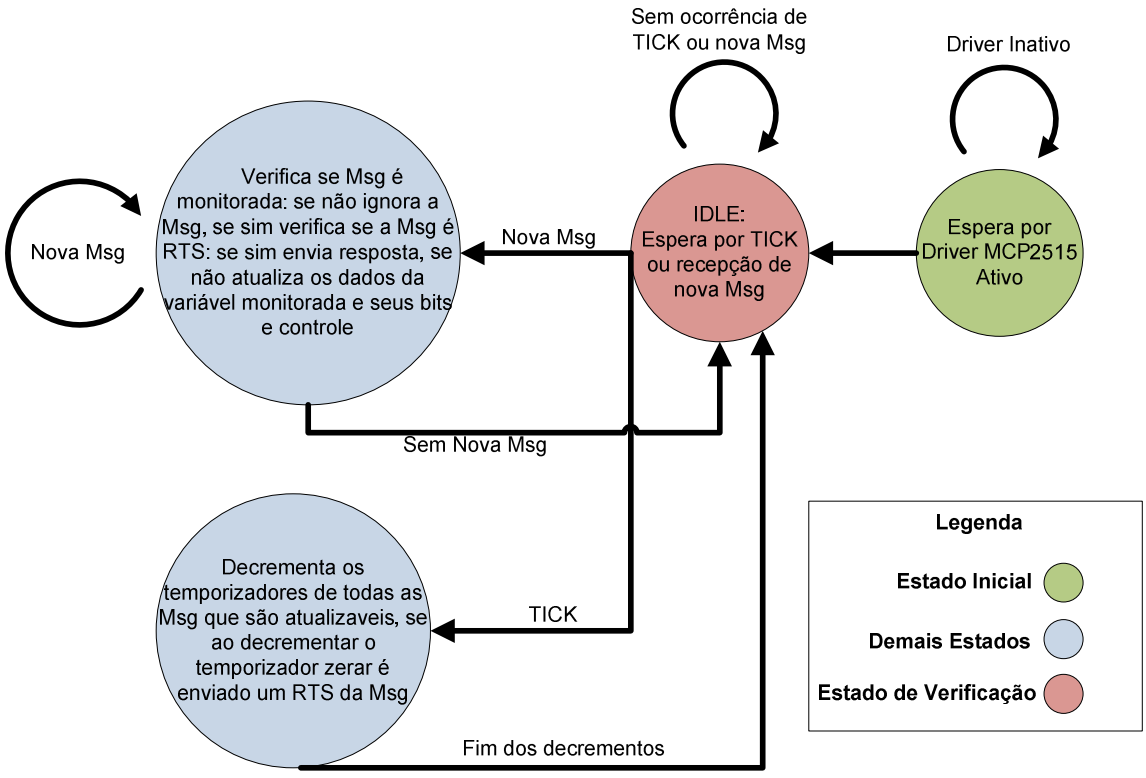


Figura 100: Máquina de estados do controlador de variáveis.

4.5 Como Utilizar Uma Variável de Barramento?

A utilização de uma variável de barramento por uma tarefa qualquer é realizada por intermédio de funções especializadas na manipulação de tais variáveis. Estas funções são:

- *HANDLE_CAN_MSG* ***InsertOnBus*** *(unsigned int* ***Id****,unsigned int* ***TimerTrigger****,void* ****DataPointer****,char* ***DataLength****, char* ***Envia****), detalhada anteriormente;*
- *void* ***CANMSGSemPost*** *(HANDLE_CAN_MSG* ***HandleMSG****):* Função que libera semáforo da mensagem identificada pelo ***HandleMSG,*** que é um número inteiro que corresponde ao índice que a mensagem tem no *buffer* de controle. Esta função não retorna valor;
- *void* ***CANMSGSemPend*** *(HANDLE_CAN_MSG* ***HandleMSG****)*: Função que marca o semaforo da mensagem identificada pelo ***HandleMSG,*** que é um número inteiro que corresponde ao índice que a mensagem tem no *buffer* de controle. Esta função não retorna valor;
- *char* ***CANDataAvaliable*** *(HANDLE_CAN_MSG* ***HandleMSG****)*: Função que verifica se a variável, cuja mensagem é identificada pelo ***HandleMSG*** no *buffer* de controle, está disponível, ou seja, se está atulizada. Esta função retorna 1 se o valor da variável está disponível, e 0 se o valor não foi atualizado;

Portanto, para uma tarefa qualquer compartilhar na rede suas variáveis, primeiramente deve-se disponibilizar na rede as variáveis que ela possui e que outros nós necessitam, usando a função ***InsertOnBus***, Assim, quando for recebido um RTS com identificador destas mensagens o controlador de variáveis irá enviá-la pela rede. Para esta situação deve-se declarar o ***TimerTrigger*** desta função como 0, visto que esta variável não será atualizada pela rede.

Em seguida, devem-se inserir as variáveis com atualização pela rede também com o uso da função ***InsertOnBus***, contudo, deve-se alocar a variável a ser atualizada exatamente como ela foi alocada pelo nó fonte. O ***TimerTrigger*** desta variável deve ser preenchido com o período de atualização desta.

Feita a inserção das variáveis no barramento, o seu uso passa a ser compartilhado pelo controlador de variáveis de barramento. Diante disso, antes do acesso a estas variáveis, deve-se usar a função ***CANDataAvaliable*** para verificar a atualização de seus dados. Se a variável não estiver atualizada a tarefa deve tomar as providencias necessárias. Se a variável estiver atualizada a tarefa deve usar primeiramente a função ***CANMSGSemPend***, que impede que outras tarefas modifiquem os dados enquanto estão sendo lidos, assim evitando processamento de dados instáveis, e depois a função ***CANMSGSemPost***, que libera a variável para ser atualizada por outras tarefas.

Seguindo esta seqüência de instruções, todas as tarefas podem usufruir da atualização automática de suas variáveis, e assim, controlar e ler dados de outros nós de rede independente de onde eles estejam hospedados.

4.5.1 Parâmetros da Implementação CAN Para a Cadeira de Rodas

Como mencionado anteriormente, a taxa da rede CAN implementada neste trabalho é de 250 $kbps$. Outro parâmetro relevante é o *TICK* do controlador de variáveis, que tanto na PP quanto na PCP é de 2 ms. Em ambas as placas foram disponibilizadas *buffers* de controle de variáveis de 16 posições. Isso implica que esta aplicação pode suportar até 16 mensagens CAN. Os *buffers* de recepção e transmissão do *driver* MCP2515 têm 16 posições em ambas as placas, ou seja, pode acumular até 16 mensagens recebidas e 16 mensagens a serem transmitidas. Não foram utilizadas máscaras ou filtros nesta aplicação, visto o pequeno número de mensagens. Estes parâmetros são configuráveis e podem sofrer alterações em outras aplicações. Foram implementadas 9 mensagens e suas descrições estão na Tabela 14.

Tabela 14: Tabela das mensagens CAN implementadas e suas descrições.

ID CAN (Hex)	Variáveis	Descrição das variáveis	Comprimento (*Bytes*)	Periodo de Atualização (ms)	Tarefa Origem	Placa Origem
0x0200	TensaoAlimentacao	Tensão de alimentação das placas de controle dos periféricos e processamento	2	50	interrupção do microcontrolador da Placa de Processamento	PP
0x02AA	Segundos, minutos, Horas, Dia,Mes e Ano	dados o RTC da PP	7	200	Relógio	PP
0x0111	Vre,Vrd	Velocidade das rodas em mm/s	8	2	TaskControleDos*Encoders*	PCP
0x0113	IR.Distancia0, IR.Distancia1, IR.Distancia2 e IR.Distancia3	Dados de distancia lidos pelos sensores infravermelho	8	100	TaskControleDos*Encoders*	PCP
0x0123	Posicao.X e Posicao.Y	posição da cadeira de rodas medida pela tarefa hodômetro em (mm,mm)	8	50	TaskControleDos*Encoders*	PCP
0x0124	Teta	Angulo de orientação da cadeira de rodas em radianos	4	50	TaskControleDos*Encoders*	PCP
0x0110	UeCAN, UdCAN	valores de PWM para motor esquerdo e direito	8	2	Controlador	PP
0x0109	MotoresComando.*Byte*	Comando diretos para os motores	1	20	Controlador	PP
0x0115	Motores*Status.Byte*	*Status* dos motores	1	20	*Driver*Motores	PCP

4.6 Conclusão

A rede CAN é muito robusta e confiável, visto que possui vários métodos de detecção de erros e meio físico apropriado para aplicações na indústria. Sua arquitetura de endereçamento de mensagens, e não de nós físicos, permite a criação das variáveis de barramento.

Para aplicações embarcadas, nas quais vários módulos diferentes se comunicam entre si e desempenham tarefas distintas, a arquitetura de variáveis de barramento se mostrou muito

eficaz, visto que especializa as tarefas, ou seja, os serviços de controle do barramento, controle de erros e gestão de comunicação são desempenhados por uma única tarefa especializada. Isso permite que o programador desenvolva as demais tarefas de modo independente do protocolo de rede.

As funcionalidades da arquitetura apresentadas somente foram possíveis devido ao *kernel* de tempo real, que garante o chaveamento das tarefas, sinalização de eventos e gestão de variáveis em tempo real. Detalhes sobre o *kernel* serão discutidos no Capítulo 5.

5 O SISTEMA OPERACIONAL DE TEMPO REAL

A cadeira de rodas é composta por vários sistemas que usam sensores e algoritmos que, respectivamente, interagem entre si e são executados de modo aleatório. Contudo, alguns dos algoritmos que compõe a cadeira de rodas têm restrições severas de tempo de execução, ou seja, devem ser executadas em tempo real, sendo estas restrições impostas de acordo com sua função. Objetivando dar garantida de execução a tais algoritmos em tempo real, foi usado um *kernel* de tempo real. Neste Capítulo o *kernel* utilizado é abordado assim como suas funções na cadeira de rodas e como foi garantida a execução das tarefas em tempo real.

5.1 Os Sistemas de Tempo Real

Na medida em que o uso de sistemas computacionais se prolifera na sociedade atual, aplicações com requisitos de tempo real tornam-se cada vez mais comuns. Essas aplicações variam muito em relação à complexidade e às necessidades de garantia no atendimento de restrições temporais. Entre os sistemas mais simples, estão os controladores inteligentes embutidos em utilidades domésticas, tais como lavadoras de roupa e DVD *players*. Na outra extremidade do espectro de complexidade estão os sistemas militares de defesa, os sistemas de controle de plantas industriais (químicas e nucleares) e o controle de tráfego aéreo e ferroviário. Algumas aplicações de tempo real apresentam restrições de tempo mais rigorosas do que outras; entre esses, encontram-se os sistemas responsáveis pelo monitoramento de pacientes em hospitais, sistemas de supervisão e controle em plantas industriais e os sistemas embarcados em robôs de automóveis até aviões e sondas espaciais. Entre aplicações que não apresentam restrições tão críticas, normalmente são citados os *videogames*, as teleconferências através da Internet e as aplicações de multimídia em geral. Todas essas aplicações que apresentam a característica adicional de estarem sujeitas a restrições temporais, são agrupados no que é usualmente identificado como Sistemas de Tempo Real [49].

A cadeira de rodas se enquadra nesta categoria, visto que possui vários algoritmos que têm restrição temporal. Outro fato importante é o tipo de usuário da cadeira de rodas, que de modo geral são pessoas com deficiências motoras graves, e assim uma falha de controle pode ocasionar graves consequências para o usuário.

Metodologias e ferramentas convencionais são usadas, em uma prática corrente, no projeto e implementação de sistemas de tempo real. A programação dessas aplicações é feita com o uso de linguagens de alto nível, em geral eficientes, mas com construções não deterministas ou ainda, com linguagens de baixo nível. Em ambos os casos, sem a preocupação de tratar o tempo de uma forma mais explícita, o que torna difícil a garantia de implementação das restrições temporais. Os sistemas operacionais ou núcleos de tempo real, que gerenciam interrupções e tarefas e permitem a programação de temporizadores e de *timeout*, são para muitos projetistas as ferramentas suficientes para a construção de sistemas de tempo real. Embora esses suportes apresentem mecanismos para implementar escalonamentos dirigidos a prioridades, essas prioridades nunca refletem as restrições temporais definidas para essas aplicações [49].

Essas prioridades são determinadas usualmente a partir da importância das funcionalidades presentes nessas aplicações; o que não leva em conta, por exemplo, que o grau de importância relativa de uma função da aplicação nem sempre se mantém igual durante todo o tempo de execução desta. Essas práticas correntes têm permitido resolver de forma aceitável e durante muito tempo certas classes de problemas de tempo real nas quais as exigências de garantia sobre as restrições temporais não são tão rigorosas. Entretanto, essas técnicas e ferramentas convencionais apresentam limitações. Por exemplo, a programação em linguagem *Assembly* produz programas com pouca legibilidade e de manutenção complexa e cuja a eficiência está intimamente ligada à experiência do programador [49].

Assim, para este trabalho optou-se por uso de linguagem de programação de alto nível e uso de *kernel* de tempo real de prioridade fixa e acesso direto às interrupções do microcontrolador.

A maior parte das tarefas da cadeira de rodas é foi desenvolvida na linguagem de programação C ANSI, sendo outras poucas sendo desenvolvias em *Assembly*. Diante de tal

fato e considerando a familiaridade existente, o sistema operacional escolhido para gerir as tarefas da cadeira de rodas foi o µC/OS-II desenvolvido pela Micrium [50]. Este sistema operacional está disponibilizado para *download* no *website* da Micrium, sendo que os termos de uso incluem que para aplicações comerciais deverá ser pago licença e para aplicações não comerciais seu uso é gratuito.

5.2 O µC/OS-II

µC/OS-II *kernel* é um sistema operacional portátil, preemptivo, de prioridade fixa, determinístico em tempo real e multitarefa para microprocessadores, microcontroladores e DSPs. Está disponível em código fonte ANSI C e documentação detalhada. É executado por amplo número de arquiteturas de processadores, com *ports* disponíveis para down*load* no site da Micrium.

5.2.1 Visão geral

O µC/OS-II pode gerenciar até 250 tarefas e inclui: semáforos, *flags* de evento, exclusão mútua de semáforos que eliminam inversões de prioridades, *message mailbox* e *queues*.

O µC/OS-II pode ser escalado entre 5 k*bytes* a 24 k*bytes* de ocupação de memória *flash* do microcontrolador, visando conter apenas os recursos necessários para uma aplicação específica. O tempo de execução para a maioria dos serviços prestados pelo *kernel* é determinístico e constante, sendo estes tempos de execução independentes do número de tarefas em execução na aplicação. Possui ampla documentação necessária para apoiar seu uso em sistemas de segurança crítica, sendo atualmente implementado em uma ampla gama de dispositivos de alto nível de segurança crítica. Inclui certificações em:

- *Avionics*-178B
- SIL3/SIL4 IEC para o transporte e sistemas nucleares.
- É compatível com 99% da *Motor Industry Software Reliability Association (MISRA-C:1998) C Coding Standards*

Tem ampla aplicação em: indústria de aviação, equipamentos médicos, comunicação de dados, eletrodomésticos, telefonia móvel e fixa, controles de processos industriais, indústria automobilística;

O µC/OS-II se resume a uma série de arquivo de compilação que devem ser compilados juntos com a aplicação. Feito isto, o acesso às funcionalidades do *kernel* passa a ser feita por intermédio de funções especializadas.

Neste trabalho, utilizou-se o *port* do µC/OS-II para o MSP430 família 5, contudo, como mencionado anteriormente, o microcontrolador utilizado foi o MSP430F2618 da família 2. Foi então realizada a adaptação no arquivo *Assembly* do *port* da familia 5 para a familia 2. Este arquivo está disponível no anexo 17.

A configuração do µC/OS-II inclui também a origem da sua interrupção de *tick*, sendo esta vinculada à interrupção de tempo do *watchdog* do microcontrolador. Assim, o *watchdog* foi configurado como *timer* cujo período é de 1 *ms* e a rotina de tratamento desta interrupção é o *tick* do µC/OS-II. Um dos grandes problemas dos sistemas operacionais preemptivos é sua elevada demanda por memória, visto que usa uma pilha para cada tarefa e no µC/OS-II não é diferente. Assim, foi disponibilizado para cada tarefa o número padrão de 300 *bytes* de memória alocados para suas pilhas.

5.2.2 Recursos do µC/OS-II Usados na Cadeira de Rodas

O µC/OS-II oferece uma série de recursos de manipulação de tempo, variáveis globais e de tarefas. Neste trabalho foram utilizados apenas os semáforos para manipulação de variáveis e semáforos, *ticks* para manipulação de tempo, e funções de criação de tarefas para manipulação destas.

Os semáforos são contadores que são usados para indicar se uma variável já está sendo utilizada por outra tarefa. Os semáforos, aliados a interrupções, são utilizados também para colocar uma tarefa no modo pronto para ser executada. Este mecanismo funciona quando uma tarefa verifica um semáforo que está bloqueado, assim, ela será paralisada até que este

seja liberado. Neste caso quem libera este semáforo é uma rotina de interrupção, que pode ser um timer, uma recepção de dados, outra tarefa e etc [51].

Como mencionado anteriormente, todos os acessos ao *kernel* são realizados por funções específicas. Logo, para acessar os recursos do µC/OS-II foram usadas as seguintes funções:

- *OSInit()*, função que inicializa o *kernel*;
- *OSTaskCreate()*, função que cria uma tarefa para que possa ser gerida pelo µC/OS-II ;
- *OSStart()*, função que inicia as multitarefas gerida pelo µC/OS-II;
- *OSTimeDly()*, função que coloca a tarefa que a chamou em modo de espera por um número definido de *ticks*.
- *OSSemAccept()*, função que retorna o valor do contador do semáforo. Se o valor for zero retorna zero; se o valor for diferente de zero retorna o valor e decrementa o contador do semáforo. Esta função não interrompe a tarefa que a chamou se o semaforo estiver sendo utilizado, ou seja, valor zero;
- *OSSemCreate()*, função que cria um semáforo gerido pelo *kernel*;
- *OSSemPend()*, função que verifica se um semaforo está sendo bloqueado por outra tarefa. Se sim, coloca a tarefa solicitante em modo de espera pela liberação; se não, bloqueia o semáforo, decrementando seu contador, e continua a execução da tarefa solicitante;
- *OSSemPost()*, função que desbloqueia um semáforo, ou seja, incrementa seu contador, e verifica se existe uma tarefa de prioridade mais alta aguardando a liberação deste;

Mais informações sobre as funções e recursos do µC/OS-II encontram-se em [51][52].

5.3 O Projeto de Tempo Real

Uma aplicação de tempo real é expressa na forma de um conjunto de tarefas e, para efeito de escalonamento, o somatório dos tempos de computação dessas tarefas na fila de "Pronto" determina a carga computacional (*task load*) que a aplicação constitui para os recursos computacionais em um determinado instante. Uma carga toma características de carga estática ou limitada quando todas as suas tarefas são bem conhecidas em tempo de projeto na forma de suas restrições temporais, ou seja, são conhecidas nas suas condições de chegada (*arrival times* das tarefas). O fato de conhecer a priori os tempos de chegada torna possível a determinação dos prazos a que uma carga está sujeita. As situações de pico (ou de pior caso) nestas cargas são também conhecidas em tempo de projeto. Cargas estáticas são modeladas através de tarefas periódicas e esporádicas [49].

Uma tarefa esporádica é descrita pela tripla (C_i, D_i, min_i), onde C_i é o tempo de computação, D_i é o *deadline* relativo medido a partir do instante da requisição do processamento aperiódico (chegada da tarefa esporádica) e min_i corresponde ao mínimo intervalo entre duas requisições consecutivas da tarefa esporádica. A descrição de uma tarefa aperiódica pura se limita apenas às restrições C_i e D_i. Na figura 2.2, a tarefa aperiódica esporádica (C_i, D_i, min_i) é apresentada com duas requisições. Tomando o tempo de chegada da requisição esporádica 2 como a_2, o *deadline* absoluto desta ativação assume o valor dado por: $d_2=a_2+D_i$ [49].

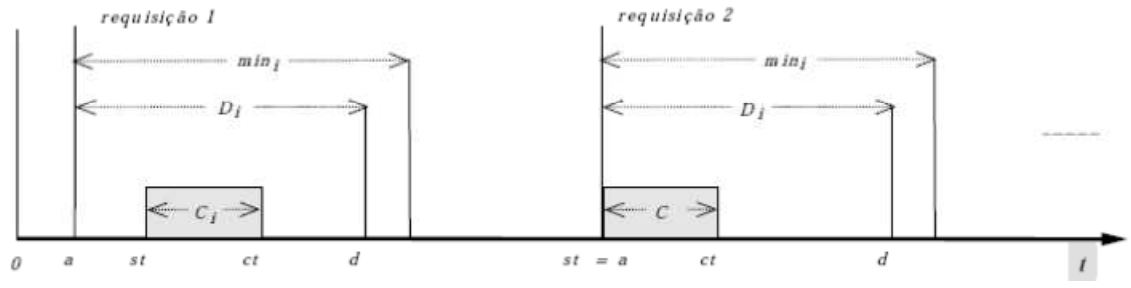


Figura 101: Representação de ativações de uma tarefa aperiódica.

O grupo garantia em tempo de projeto é formado por abordagens que tem como finalidade a previsibilidade determinista. A garantia em tempo de projeto é conseguida a partir de um conjunto de premissas:

- A carga computacional do sistema é conhecida em tempo de projeto (carga estática);
- No sistema existe uma reserva de recursos suficientes para a execução das tarefas, atendendo suas restrições temporais, na condição de pior caso.

Como mencionado anteriormente, o µC/OS-II é um *kernel* que usa prioridade fixa para as suas tarefas. Esta prioridade é dada por um número de 0 a 63, sendo que quanto menor este valor maior será a prioridade da tarefa. Este esquema de escalonamento é chamado de escalonamento Taxa Monotônica ("*Rate Monotonic*", RM).

O escalonamento Taxa Monotônica produz escalas em tempo de execução através de escalonadores preemptivos, dirigidos a prioridades. É um esquema de prioridade fixa; o que define então o RM como escalonamento estático e *on-line*. O RM é dito ótimo entre os escalonamentos de prioridade fixa na sua classe de problema, ou seja, nenhum outro algoritmo da mesma classe pode escalonar um conjunto de tarefas que não seja escalonável pelo RM [49].

As premissas do RM que facilitam as análises de escalonabilidade, definem um modelo de tarefas bastante simples:

a. As tarefas são periódicas e independentes.

b. O *"deadline"* de cada tarefa coincide com o seu período ($D_i=P_i$).

c. O tempo de computação (C_i) de cada tarefa é conhecido e constante ("*Worst Case Computation Time*").

d. O tempo de chaveamento entre tarefas é assumido como nulo.

As premissas *a* e *b* são muito restritivas para o uso desse modelo na prática, contudo, essas simplificações são importantes para que se tenha o entendimento sobre o escalonamento de tarefas periódicas [49].

A política que define a atribuição de prioridades usando o RM determina uma ordenação baseada nos valores de períodos das tarefas do conjunto: as prioridades decrescem em função do aumento dos períodos, ou seja, quanto mais frequente a tarefa maior a sua prioridade no conjunto. Como os períodos das tarefas não mudam, o RM define uma atribuição estática de prioridades (prioridade fixa).

A análise de escalonabilidade no RM, feita em tempo de projeto, é baseada no cálculo da utilização. Para que *n* tarefas tenham o atendimento de suas restrições temporais quando escalonadas pelo RM, deve ser satisfeito o teste abaixo que define uma condição *suficiente* [53]:

$$U = \sum_{i}^{n} \frac{C_i}{P_i} \leq n\left(2^{1/n} - 1\right)$$

Equação 16

Onde, U é a utilização total da CPU e $\frac{C_i}{P_i}$ é o custo de cada tarefa i.

As tarefas implementadas na cadeira de rodas podem ser consideradas independentes entre si, contudo não tem período definido. Nesta situação foi adotada como custo da tarefa a razão C_{max}/P_{min}, sendo C_{max} o tempo computacional mais elevado de uma tarefa e P_{min} o período mínimo de chamada desta mesma tarefa. Esta opção é por vezes criticada por alguns autores e projetistas, que argumentam que sobre dimensiona excessivamente o projeto, sendo que esta situação extrema tem, de modo geral, probabilidade baixa de ocorrência [49].

Contudo, por se tratar de um sistema no qual a integridade física do usuário é dependente de seu desempenho, para a cadeira de rodas foi adotado a hipótese extrema para cada tarefa.

5.3.1 Estudo de Escalonabilidade da Placa de Processamento

O estudo de escalonabilidade da cadeira de rodas foi dividido por placa, sendo então realizado um estudo de quantidade de tarefas, tempo computacional máximo de cada tarefa e

período mínimo de chamada de cada tarefa. Este estudo está ilustrado na Tabela 15 sendo as condições para as mensurações de cada tarefa as mais extremas de cada uma.

Tabela 15: Estudo de escalonabilidade da Placa de Processamento.

Tarefa	Prioridade	Duração máxima em Núm. de Ciclos de *Clock*	Tempo Máximo de Processamento (us)	Período Mínimo(ms)	Custo (%)
Controlador	18	2500	156,3	5	3,13%
Amostrador	40	10620	663,8	50	1,33%
Relogio	55	3000	187,5	150	0,13%
UART	50	72800	4.550,0	100	4,55%
SDControl	30	40500	2.531,3	50	5,06%
Driver MCP2515	8	12600	787,5	5	15,75%
Controlador de Variáveis CAN	15	16119	1.007,4	5	20,15%

As mensurações ilustradas na Tabela 15 referentes aos valores de ciclos de execução e período de execução de cada tarefa consideraram as seguintes hipóteses:

- O controlador tem seu maior ciclo quando executa o comando EXECUTAR. Seu período é estabelecido por uma interrupção de tempo fixada em 5 ms.
- O amostrador é uma rotina que possui *loops* dependentes da quantidade de algarismos decimais que cada dado amostrado possui. Assim, foi considerado como pior caso quando o amostrador realiza amostragem de seis valores de 5 algarismos decimais. Seu período é estabelecido pela função *OSTimeDly* de 50 ms;
- O Relógio é uma rotina periódica executada a cada 150 ms e que possui pior condição de execução quando executa a leitura do DS1307, portanto, esta é a situação extrema ilustrada na Tabela 15.
- A UART tem sua condição de maior demanda quando recebe o comando P da Tabela 6, sendo este comando enviado pela aplicação de alto nível / supervisório tipicamente a cada 100 ms.
- A tarefa SDControl tem sua condição de maior demanda quando executa a transferência para o DOSonCHIP no modo de escrita, sendo esta rotina também periódica com seu menor período igual a 50 ms.

- O *driver* MCP2515 necessita de maior demanda computacional quando executa o atendimento à RTS, sendo necessário o seu recebimento e posterior transmissão. Assumindo que foram implementadas nove mensagens CAN, a pior condição é referente à soma dos tempos de processamento de recepção de RTS seguido de envio das nove mensagens, sendo assumindo o período mínimo de 5 ms referente ao tempo de atualização da mensagem de menor período de atualização;
- Igualmente ao *driver* MCP2515, o controlador de variáveis tem sua condição de maior demanda quando executa atendimento à RST. Assumindo que foram implementadas nove mensagens CAN, a pior condição é referente à soma dos tempos de processamento de recepção de RTS seguido de envio das nove mensagens, sendo assumindo o período mínimo de 5 ms referente ao tempo de atualização da mensagem de menor período de atualização;

Portanto, a utilização total máxima do microcontrolador da Placa de Processamento é dada pela soma dos custos percentuais da Tabela 15, sendo igual a $U = 50{,}1$ %. Sabendo que a quantidade de tarefas implementadas é igual a 6, e usando a Equação 16, tem-se:

$$U = 0{,}501 \quad , \qquad Limite = 6\left(2^{\frac{1}{6}} - 1\right) = 0{,}7347 \quad \therefore U \leq limite$$

Equação 17

Assim, a condição suficiente para escalonabilidade [53] de tarefas independentes, periódicas, com prioridade determinada pelo método RM, é atendida para a Placa de Processamento.

5.3.2 Estudo de Escalonabilidade da Placa de Controle dos Periféricos

Similar ao estudo de escalonabilidade da Placa de Processamento, o estudo de escalonabilidade da Placa de Controle dos Periféricos assume as condições mais extremas de execução de suas tarefas. Assim, a Tabela 16 mostra tais situações para cada tarefa desta placa.

Tabela 16: Estudo de escalonabilidade da Placa de Controle dos Periféricos.

Tarefa	Prioridade	Duração máxima em Núm. de Ciclos de *Clock*	Tempo Máximo de Processamento (us)	Período Mínimo(ms)	Custo (%)
Hodômetro	25	3950	247	5	4,9%
Driver Motores	19	1000	63	5	1,2%
Display	40	20000	1250	100	1,25%
Driver MCP2515	8	12600	787,5	5	15,75%
Controlador de Variáveis CAN	15	16119	1.007,4	5	20,15%

As mensurações ilustradas na Tabela 16 referentes aos valores de ciclos de execução e período de execução de cada tarefa consideraram as seguintes hipóteses:

- O hodômetro tem sua condição de execução mais extrema quando executa o comando *Run*. Seu período é fixado em 5 ms;
- O *driver* motores tem sua condição mais extrema também quando executa o comando *Run*. Seu período também é fixado em 5 ms;
- O display tem sua condição de maior demanda de processamento quando mostra variáveis com cinco dígitos decimais. Seu período de execução é de 100 ms;
- O *driver* MCP2515 necessita de maior demanda computacional quando executa o atendimento à RTS, sendo necessário o seu recebimento e posterior transmissão. Assumindo que foram implementadas nove mensagens CAN a pior condição é referente à soma dos tempos de processamento de recepção de RTS seguido de envio das nove mensagens, sendo assumindo o período mínimo de 5 ms referente ao tempo de atualização da mensagem de menor período de atualização;
- Igualmente ao *driver* MCP2515, o controlador de variáveis tem sua condição de maior demanda quando executa atendimento à RST. Assumindo que foram implementadas nove mensagens CAN, a pior condição é referente à soma dos tempos de processamento de recepção de RTS seguido de envio das nove mensagens, sendo assumindo o período mínimo de 5 ms referente ao tempo de atualização da mensagem de menor período de atualização;

Portanto, a utilização total máxima do microcontrolador da Placa de Controle dos Periféricos é dada pela soma dos custos percentuais da Tabela 16, sendo igual a $U = 43{,}25\ \%$. Sabendo que a quantidade de tarefas implementadas é igual a 5, e usando a Equação 16, tem-se:

$$U = 0{,}4325 \quad , \qquad Limite = 5\left(2^{\frac{1}{5}} - 1\right) = 0{,}7434 \quad \therefore U \leq limite$$

Equação 18

Assim, a condição suficiente para escalonabilidade [53] de tarefas independentes, periódicas, com prioridade determinada pelo método RM, é atendida para a Placa de Controle dos Periféricos.

5.3.3 As Garantias de Tempo Real

Como mencionado na seção anterior, as condições suficientes para escalonabilidade em tempo real das tarefas de ambas as placas deste trabalho foram atendidas, visto que foram consideradas as situações mais extremas de demanda computacional de cada tarefa. Contudo, o modelo RM possui uma limitação referente ao fato de usar prioridade fixa. Esta limitação é evidenciada quando uma tarefa de prioridade menor usa um semáforo para restringir acesso de outras tarefas a um conjunto de variáveis. Assim, quando uma tarefa de maior prioridade tentar utilizar este semáforo entrará em estado de espera. Esta situação é conhecida como inversão de prioridades. Isso pode ser catastrófico, dependendo do que é controlado pela tarefa paralisada. Contudo, esta condição foi prevista neste trabalho, sendo tomadas medidas de detecção e tratamento no caso de alguma tarefa perder tempo real por este ou por qualquer outro motivo.

A primeira medida é eliminar a possibilidade de ocorrência de inversão de prioridades. Isso é possível usando-se variáveis intermediárias para cálculos e demais demandas computacionais e atualizar as variáveis globais, ou de rede, desabilitando-se as interrupções. Assim, é possível atualizar ou amostrar as variáveis globais, ou de rede, sem qualquer risco de corrompe-las ou promover de inversão de prioridades. Esta técnica é muito prática e segura, contudo demanda mais memória do microcontrolador, visto que duplica

informações, e ainda pode fazer o microcontrolador permanecer longos períodos de tempo com interrupções desabilitadas, assim, por estas razões deve ser usada com cautela.

A segunda medida é a determinação de quais tarefas necessitam ser executadas em tempo real, sendo eleitas as seguintes:

- Hodômetro;
- Controlador;
- *Driver* Motores;
- *Driver* MCP2515;
- Controlador de Variáveis;

No caso da Hodômetro, Controlador e *driver* Motores, foram utilizados testes dos semáforos de chamadas para detectar perda de tempo real. Assim, como estas tarefas são ativadas por semáforos liberados por interrupções de tempo periódicas, basta cada tarefa verificar se existe mais de uma liberação de seu semáforo ativa. Se isto ocorrer, significa que a interrupção de tempo ocorreu mais de uma vez no período de execução desta tarefa, caracterizando assim a perda de tempo real.

Ainda, no caso da Hodômetro, Controlador e *driver* Motores, esta perda de tempo real é muito crítica, visto que uma falha em qualquer uma destas tarefas pode colocar a integridade física do usuário em risco. Portanto, a ação tomada por cada tarefa é sinalizar a falha de perda de tempo real para as outras tarefas e entrar em estado de emergência, no qual suas funções são desligadas. As demais tarefas ao detectar a perda de tempo real de outras também assumem o estado de emergência.

No caso da tarefa *driver* MCP2515, a perda e tempo real é detectada por perdas de mensagens. Estas perdas são detectadas pela observação periódica dos *flags* de erro do MCP2515. Ao ler os *flags* de erros do MCP2515, o *driver* MCP2515 acende um LED de erro. Esta falha de perda de tempo real do *driver* MCP2515 é detectada pelas outras tarefas por indisponibilidade de alguma variável de barramento. Neste caso a ocorrência de seis falhas consecutivas deste tipo leva o controlador e o *driver* motores para o estado de espera de variável, fazendo-os desligarem suas funções.

A perda de tempo real no controlador de variáveis é também detectada pelas outras variáveis, como indisponibilidade de alguma variável de barramento. Novamente, a indisponibilidade de dados por seis vezes consecutivas leva o controlador e o *driver* motores para o estado de espera de variável, fazendo-os desligarem suas funções.

Foi realizado experimento no qual todo o sistema permaneceu ligado por três dias consecutivos, e neste período não foi detectada nenhuma ocorrência de perda de tempo real. Isso, não é garantia de que isto não ocorra, contudo, se vier a ocorrer, existem ações de detecção e segurança que tratam a ocorrência, que são basicamente *timers* que medem a o atendimento as restrições de tempo real e rotinas que quando tais restrições são violadas, param o sistema e sinalizam falha de tempo real para o usuário.

5.4 Conclusão

O sistema de tempo real é de suma importância para este projeto, visto que são necessárias várias tarefas para realizar as funcionalidades da Cadeira de Rodas Robótica. Sem o sistema de tempo real dificilmente poderia ser dada garantia de tempo real ao sistema, além disso, o sistema de tempo real disponibilizou recursos fundamentais para o projeto, que sem eles não seria possível adotar as rotinas de variáveis de barramento. Portanto, a garantia de tempo real e a disponibilização de recursos de gerenciamento de tarefas foram decisivos para implementar a arquitetura descrita neste projeto.

6 O CONTROLADOR DE BAIXO NÍVEL DA CADEIRA DE RODAS

A Cadeira de Rodas Robotizada é um sistema de controle automático, visto que suas variáveis velocidade angular e velocidade linear devem seguir valores estabelecidos pelo sistema de controle de alto nível. Objetivando prover controle preciso de tais variáveis, foi implementado um controlador do tipo proporcional integral com laço integral diferencial entre as rodas. Este Capítulo aborda sua implementação e os resultados dos testes práticos deste controlador, juntamente com a rotina executada na Placa de Processamento que o executa.

6.1 A implementação da Tarefa Controlador

A tarefa Controlador é a rotina que controla os sinais PWM dos motores da cadeira de roda. Por opção de projeto, esta rotina é executada na Placa de Processamento. Ela avalia os valores das variáveis de rede: velocidade da roda direita, velocidade da roda esquerda, distância medida dos sensores infravermelhos e comandos diretos enviados a ela e, assim executa os cálculos de PWMs para as rodas e os disponibiliza na rede para a PCP.

Antes de efetivamente realizar os cálculos para os valores de PWM dos motores, a tarefa controlador executa uma série de verificações de segurança, entre elas:

- Verifica disponibilidade das variáveis de rede;
- Verifica os valores das distâncias medidas pelos sensores infravermelhos;
- Verifica perda de tempo real, abordado no Capítulo 5;
- Verifica comando dado ao controlador;

A implementação da tarefa controlador usou a técnica de programação *State Chart* [43]. Assim, a máquina de estados criada para sua execução está representada na Figura 102 a qual é possível observar a existência dos seguintes estados;

- RESET CONTROLADOR, sendo este o estado inicial do controlador e onde o controlador atualiza os valores dos parâmetros com os valores armazenados na memória *Flash* do microcontrolador e faz os valores de *set-point*, os valores de PWMs e os integradores iguais a zero;
- COMANDO, sendo este estado o responsável pela interpretação dos comandos direcionados ao controlador pela UART, e por atualizar os valores dos sensores infravermelho, disponibilidade das variáveis de rede e atualizar a palavra de *status* do controlador com os erros de perda de tempo real;
- SALVA PARÂMETROS, sendo este estado o responsável por salvar os parâmetros oriundos da UART na memória *Flash.*
- ATUALIZA PARÂMETROS, sendo este estado o responsável por atualizar os valores dos parâmetros do controlador com os valores vindos da UART.
- REPARA FALHAS, sendo neste estado realizado o reset das falhas ativas do controlador e o envio do comando de limpa falhas para o *Driver* motores e hodômetro;
- VERIFICA FALHAS, sendo neste estado realizada a verificação das falhas ativas na palavra de *status* do controlador;
- EXECUÇAO, sendo neste estado realizada a execução dos cálculos do controlador;
- DESLIGA CONTROLADOR, sendo este o estado no qual os motores são desligados e os valores dos integradores e *set-points* igualados a zero;

Os comandos são enviados ao controlador por uma palavra de comandos pela UART. Assim, ao ser executado o estado COMANDO, a tarefa controlador verifica qual foi o comando enviado para ele e o executa. Esta palavra de comandos é composta por um *byte* que pode assumir os seguintes valores:

- EXECUTAR = 0x30;
- RESETAR_CONTROLADOR = 0x31;
- DESLIGAR_CONTROLADOR = 0x32;
- SALVAR_PARAMETROS = 0x33;
- ATUALIZAR_PARAMETROS = 0x34;

- LIMPA_FALHAS = 0x35;

As relações entre estes estados estão representadas na Figura 102, e o arquivo de compilação na linguagem C do controlador encontram-se no anexo 18.

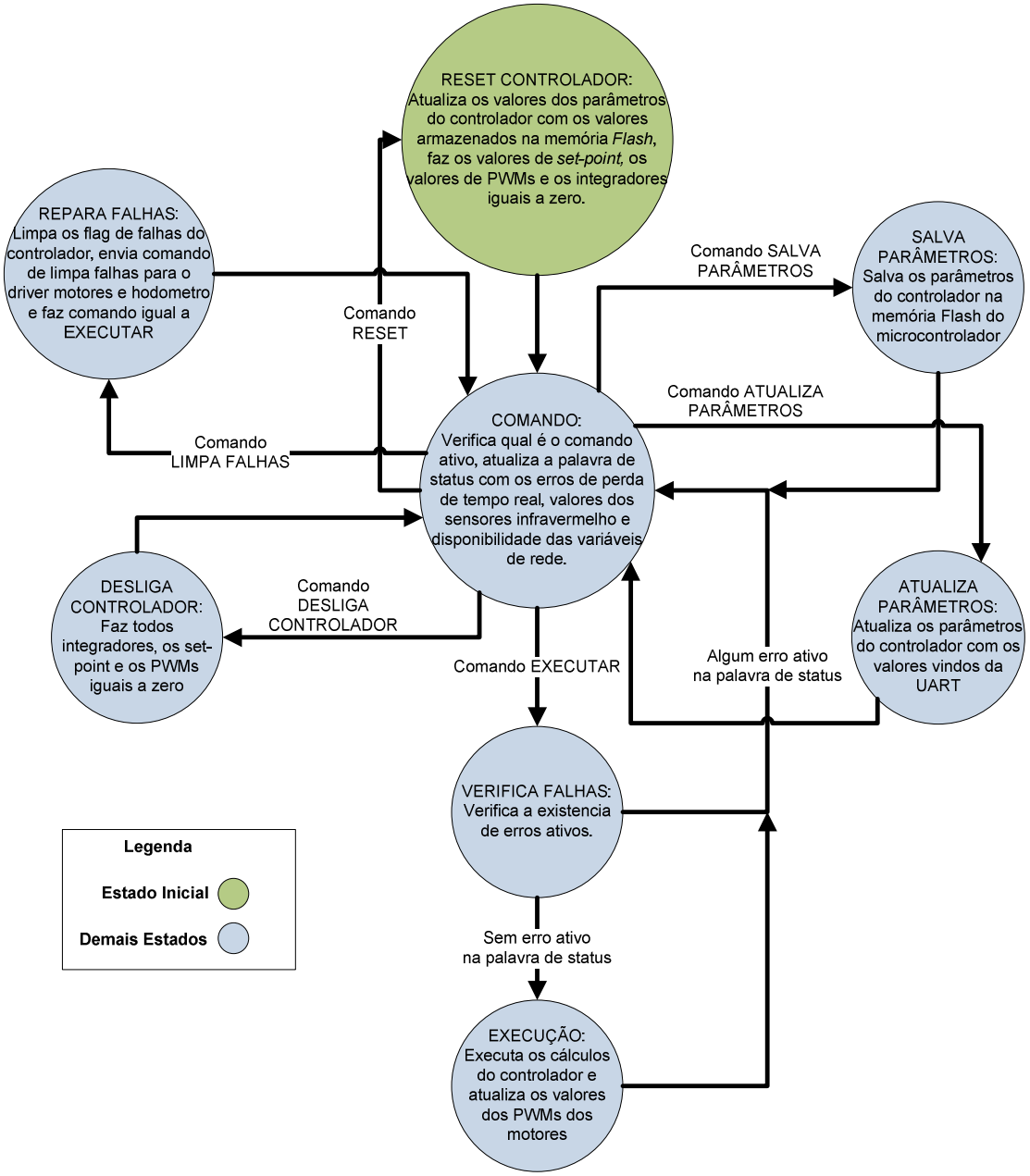


Figura 102: Representação da máquina de estados da tarefa Controlador.

6.2 O Modelo do Controlador de Baixo Nível da Cadeira de Rodas Robotizada

O controlador é uma função matemática que usa como entrada valores de referencia (*set-points*), valores dos seus estados e valores de medidos das variáveis controladas para fornecer um novo valor de controle dos atuadores das variáveis controladas.

A Cadeira de Rodas Robotizada é considerada um robô móvel com duas rodas tracionadas e duas rodas loucas (Figura 59). As rodas tracionadas possuem motores independentes. Pelo fato das rodas tracionadas serem independentes, as curvas são realizadas por diferença de velocidade entre estas duas rodas. Em [17] é proposto o seguinte controlador:

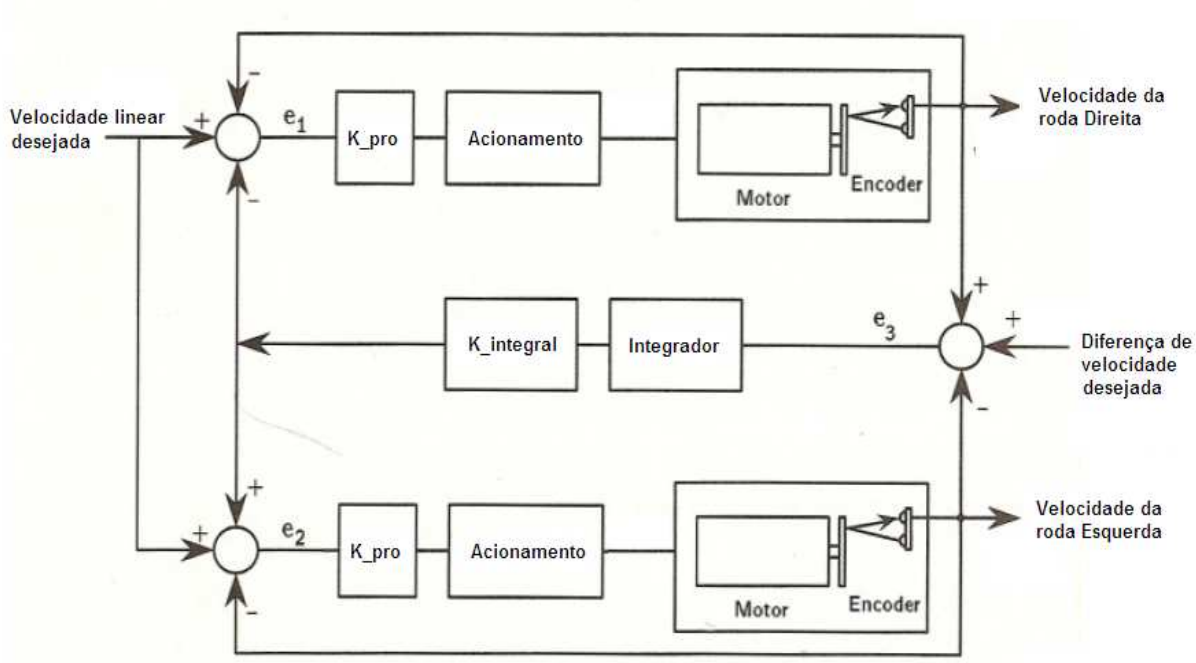


Figura 103: Controlador para robôs móveis do tipo proporcional usando laço integral diferencial entre as rodas.

No modelo do controlador ilustrado na Figura 103 nota-se a existência de um laço integral diferencial entre as rodas. Tal laço integral tem a função de controlar a diferença de velocidade entre as rodas. Nesta mesma Figura nota-se também a existência de um controlador proporcional individual para cada roda. Devido à existência deste controlador proporcional individual para cada roda o controlador apresenta erro em regime maior que zero, visto o comportamento típico de um controlador proporcional. Tal erro em regime do

referido controlador foi verificado em experimentos, cujos resultados estão ilustrados na Figura 104.

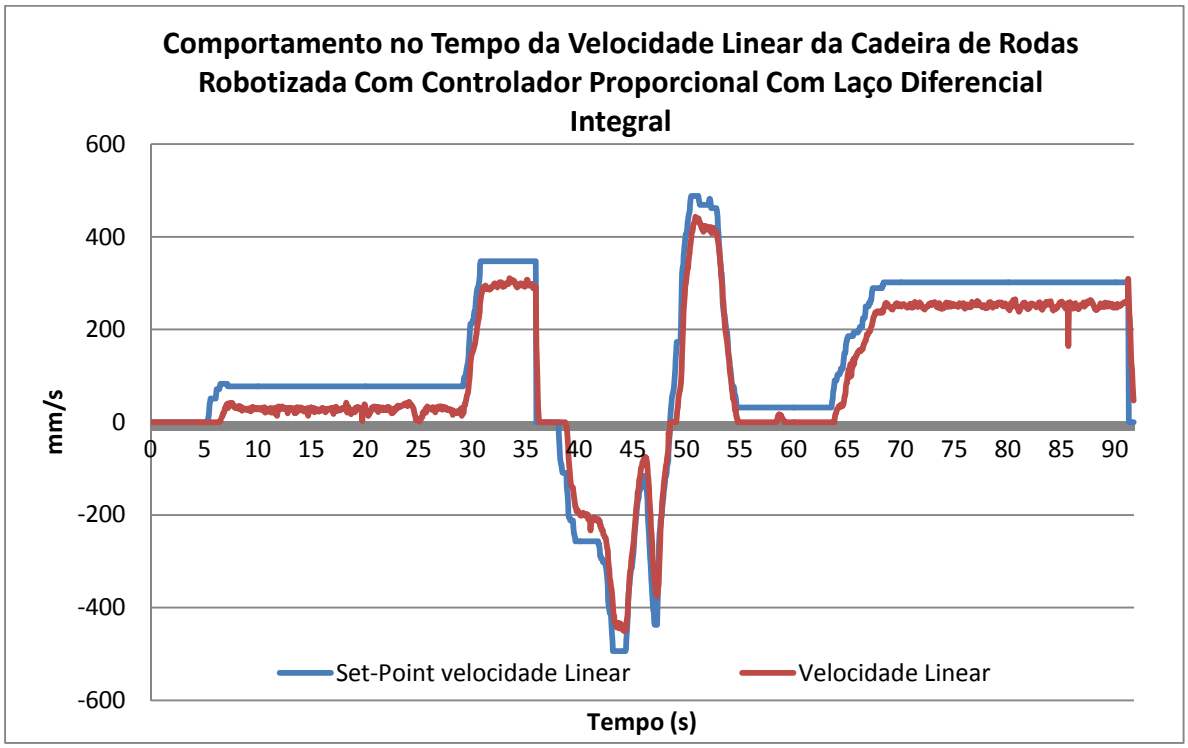


Figura 104: Gráfico de comportamento da velocidade linear no tempo da Cadeira de Rodas Robotizada com controlador proporcional com laço diferencial integral.

Os dados apresentados nos gráficos deste Capítulo são todos provenientes da coleta do *log* do cartão micro-SD com tempo de amostragem de 50 ms, sendo os testes realizados com a cadeira de rodas no chão e sem usuário. Estes testes foram realizados com o auxilio do supervisório abordado no Capítulo 7 e os parâmetros do controlador foram ajustados para tempo de resposta baixo, sendo o ganho do laço integral k_i' igual a 4, e os ganhos dos PIs individuais iguais a: k_p igual a 3 e k_i igual a 15.

Para solucionar o problema do erro em regime, foi implementado além do ganho proporcional um ganho integral individual para cada roda. Assim, o controlador implementado possui um laço diferencial integral entre as rodas, e um controlador PI individual para cada roda, sendo na Figura 105 ilustrada a topologia deste novo controlador proposto.

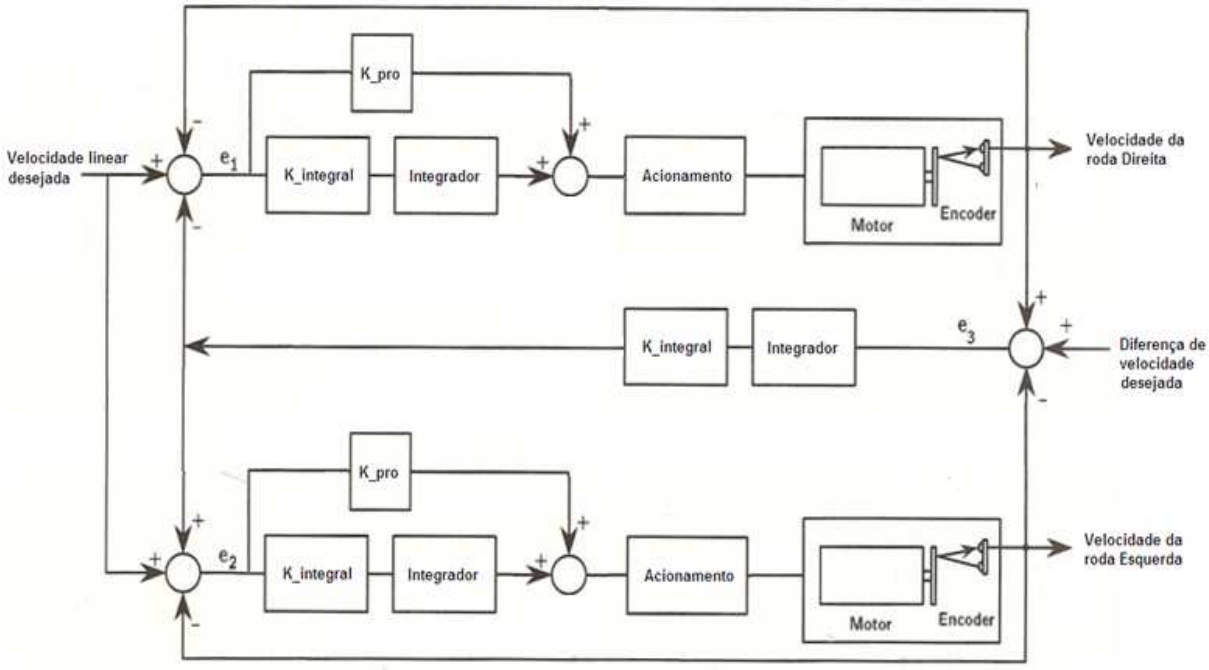


Figura 105: Controlador implementado na Cadeira de Rodas Robotizada.

Assim, foi implementado o controlador da Figura 105 na Cadeira de Rodas Robotizada, sendo então realizados testes práticos. Os resultados de tais testes estão ilustrados na Figura 106 e na Figura 107, nas quais é possível observar, respectivamente, o comportamento da velocidade linear e velocidade angular juntamente com seus *set-points*. Nota-se no gráfico da Figura 107 um nível de ruído mais elevado que o ilustrado na Figura 106. Este fato é justificado pela ocorrência de deslizamentos e arrastos das duas rodas loucas presentes na dianteira da cadeira de rodas, sendo que estes deslizamentos e arrastos são mais significativos nos movimentos circulares que nos movimentos retilíneos.

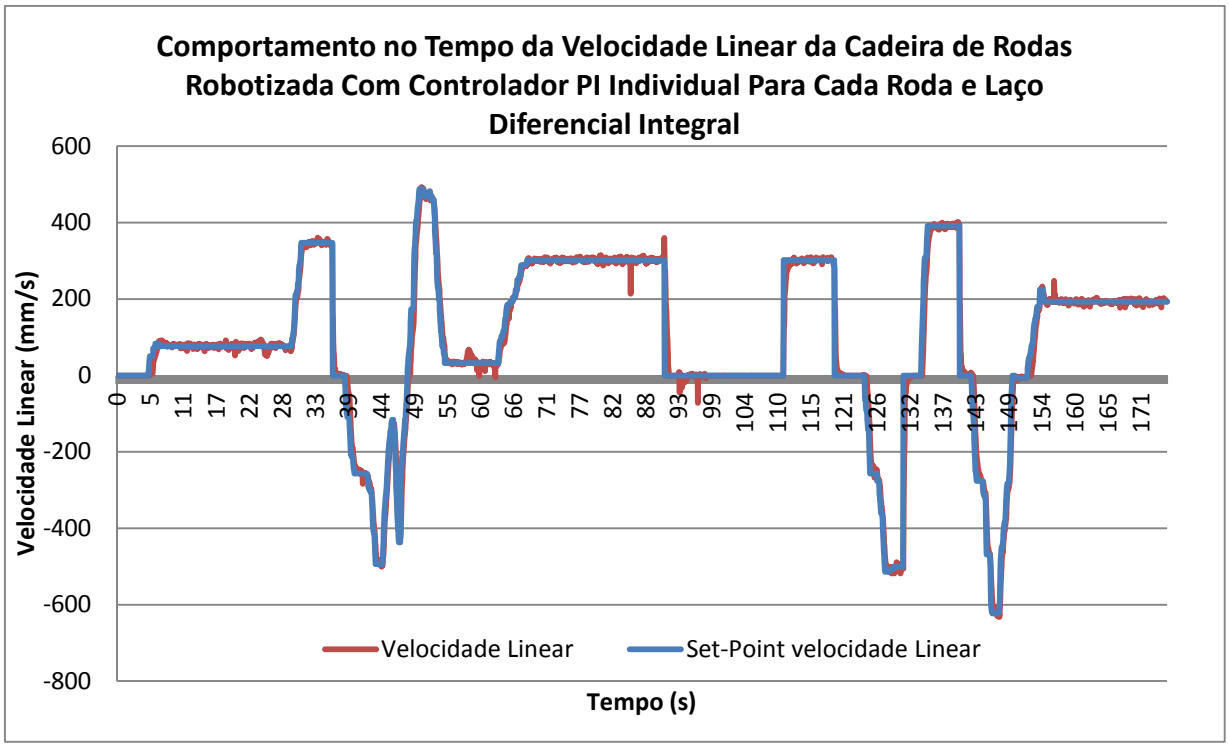


Figura 106: Gráfico do comportamento da velocidade linear no tempo da Cadeira de Rodas Robotizada com controlador PI individual para cada roda e laço diferencial integral.

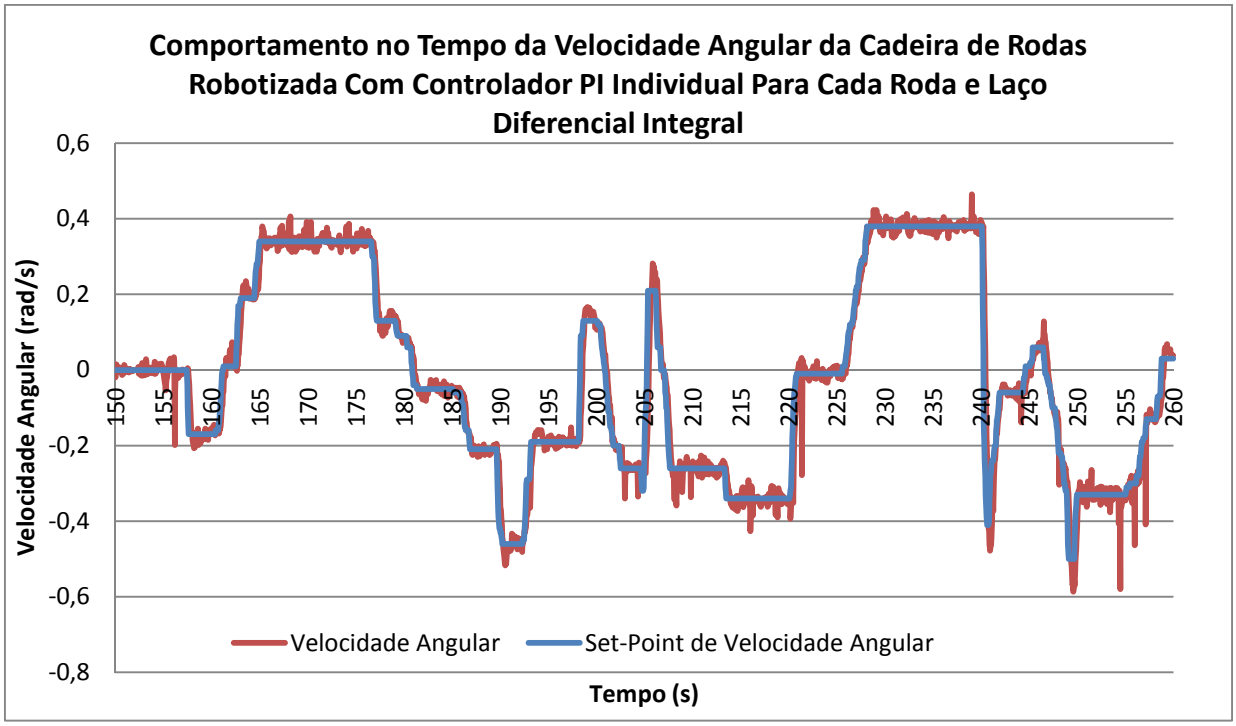


Figura 107: Gráfico do comportamento no tempo da velocidade angular da Cadeira de Rodas Robotizada com controlador PI individual para cada roda e laço diferencial integral.

Outra característica do controlador implementado é a independência entre as velocidades angular e linear e seus *set-point*, ou seja, as velocidades angular e linear não possuem qualquer dependência entre si. Em seguida será demonstrada tal característica.

Sendo k_i e k_p os ganhos dos controladores PIs individuas de cada roda e k_i' o ganho do laço integral diferencial entre as rodas; PWM_d e PWM_e as saídas do controlador; V_l' a velocidade linear medida; ω' a velocidade angular medida; V_e a velocidade medida da roda esquerda; V_d a velocidade medida da roda direita; d a distância entre as rodas; V_l o *set-point* de velocidade linear e ω o *set-point* de velocidade angular, as equações do controlador são:

$$PWM_d = \left(k_p + \frac{k_i}{s}\right)(V_l - V_d - C)$$

Equação 19

$$PWM_e = \left(k_p + \frac{k_i}{s}\right)(V_l - V_e + C)$$

Equação 20

$$C = \frac{dk_i'}{s}(\omega - \omega')$$

Equação 21

Assumindo que a velocidade da roda esquerda e a velocidade da roda direita dependem apenas dos valores de seus respectivos PWMs, estas são regidas pelas expressões:

$$G_e(s) = \frac{V_e}{PWM_e}$$

Equação 22

$$G_d(s) = \frac{V_d}{PWM_d}$$

Equação 23

Sendo, $G_e(s)$ e $G_d(s)$ as funções de transferência para as velocidades das rodas esquerda e direta, respectivamente, em relação a seus respectivos PWMs. Assumindo que a

cadeira de rodas é simétrica, e o peso, tanto do usuário quanto da própria cadeira, está divido igualmente em relação ao eixo que passa pelo ponto Q e é paralelo ao vetor V_l da Figura 59 é possível afirmar que:

$$G_e(s) = G_d(s) = G(s)$$

Equação 24

Fazendo a Equação 19 mais a Equação 20 e aplicando a Equação 4 se tem:

$$V_l' = \frac{V_l}{2} - \frac{PWM_d + PWM_e}{2\left(k_p + \frac{k_i}{s}\right)}$$

Equação 25

Fazendo a Equação 19 menos a Equação 20 e aplicando a Equação 4 se tem:

$$\omega' = \frac{PWM_e - PWM_d}{d\left(k_p + \frac{k_i}{s}\right)\left(1 - \frac{k_i'}{s}\right)} - \frac{k_i'}{s\left(1 - \frac{k_i'}{s}\right)}\omega$$

Equação 26

Assim, usando a Equação 22, Equação 23 e Equação 24, tem-se:

$$\begin{bmatrix} V_l' \\ \omega' \end{bmatrix} = \begin{bmatrix} \frac{G(s)\left(k_p s + k_i\right)}{s + (k_p s + k_i)G(s)} & 0 \\ 0 & \frac{2G(s)\left(k_p s + k_i\right)k_i'}{s^2 + sG(s)\left(k_p s + k_i\right) + 2\left(k_p s + k_i\right)k_i'} \end{bmatrix} \begin{bmatrix} V_l \\ \omega \end{bmatrix} \rightarrow \begin{bmatrix} V_l' \\ \omega' \end{bmatrix} = A \begin{bmatrix} V_l \\ \omega \end{bmatrix}$$

Equação 27

Sendo A a matriz da função de transferência da Cadeira de Rodas Robotizada.

Experimentalmente foi possível observar os efeitos de independência entre as velocidades angular e linear e seus *set-points* da Equação 27, sendo tais resultados experimentais ilustrados na Figura 108.

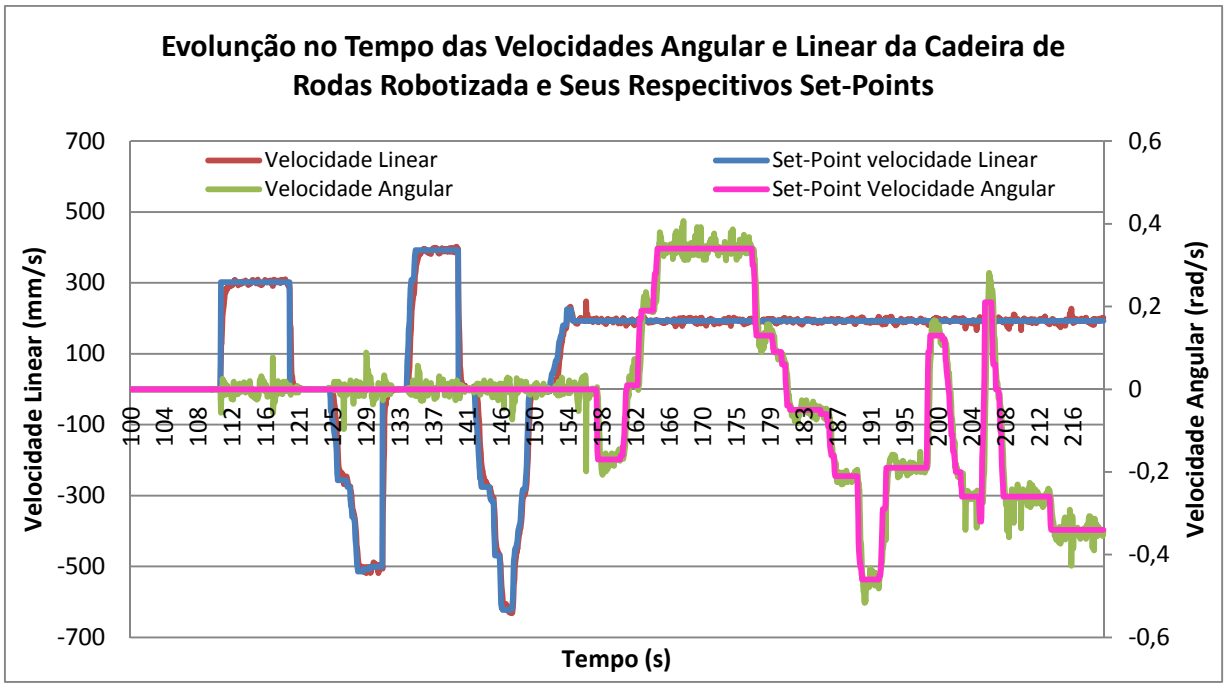


Figura 108: Evolução no tempo das velocidades angular e linear da Cadeira de Rodas Robotizada e seus respectivos *set-points*.

Na Equação 27 é possível observar que se todos os pólos de $G(s)$ estiverem no semi-plano negativo do plano complexo ou não, existem valores de k_p, k_i e k_i' que fazem o sistema estável, visto que é possível alocar os pólos de A no semi-plano negativo apenas ajustando-se tais parâmetros.

6.3 Conclusão

O controlador proposto e implementado neste trabalho mostrou-se eficiente, visto que leva os erros tanto de velocidade angular quanto de velocidade linear a zero. Esta característica pode ser observada na Figura 109 e Figura 110, nas quais é possível observar que embora exista um ruído típico o erro tende a zero. Permite também que sejam parametrizados tempo de subida e descida das velocidades linear e angular para cada usuário, apenas ajustando seus parâmetros, sendo esta característica importante para prover maior conforto para o usuário nas acelerações e desacelerações.

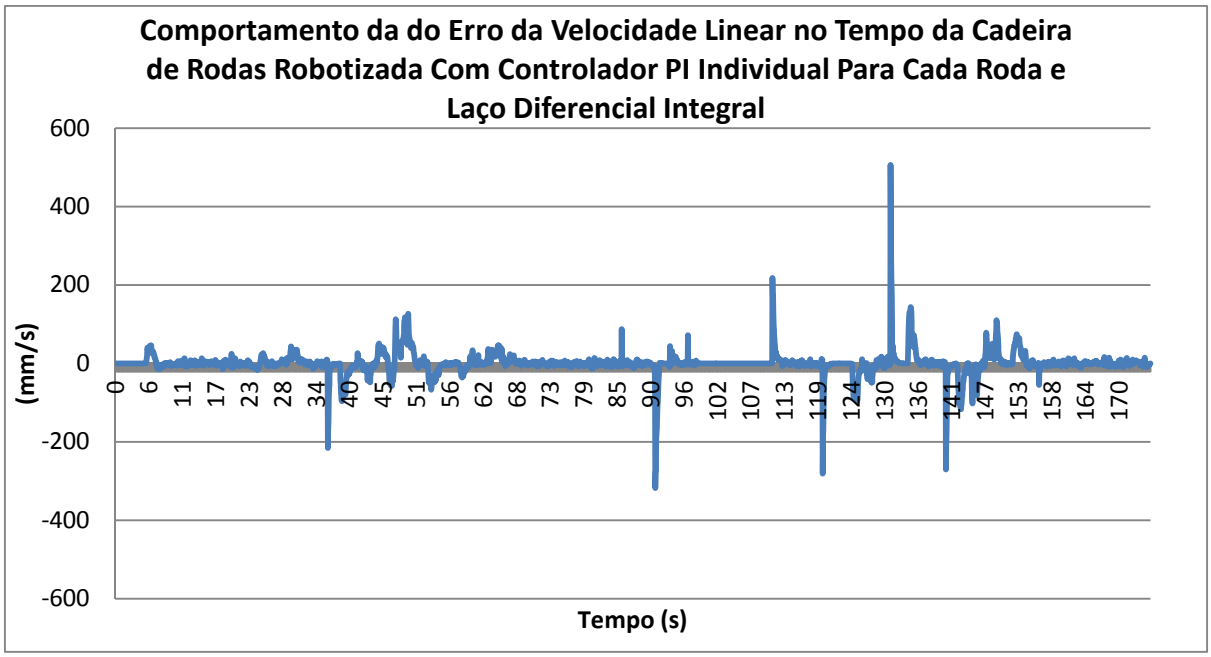


Figura 109: Comportamento da do Erro da Velocidade Linear no Tempo da Cadeira de Rodas Robotizada Com Controlador PI Individual Para Cada Roda e Laço Diferencial Integral.

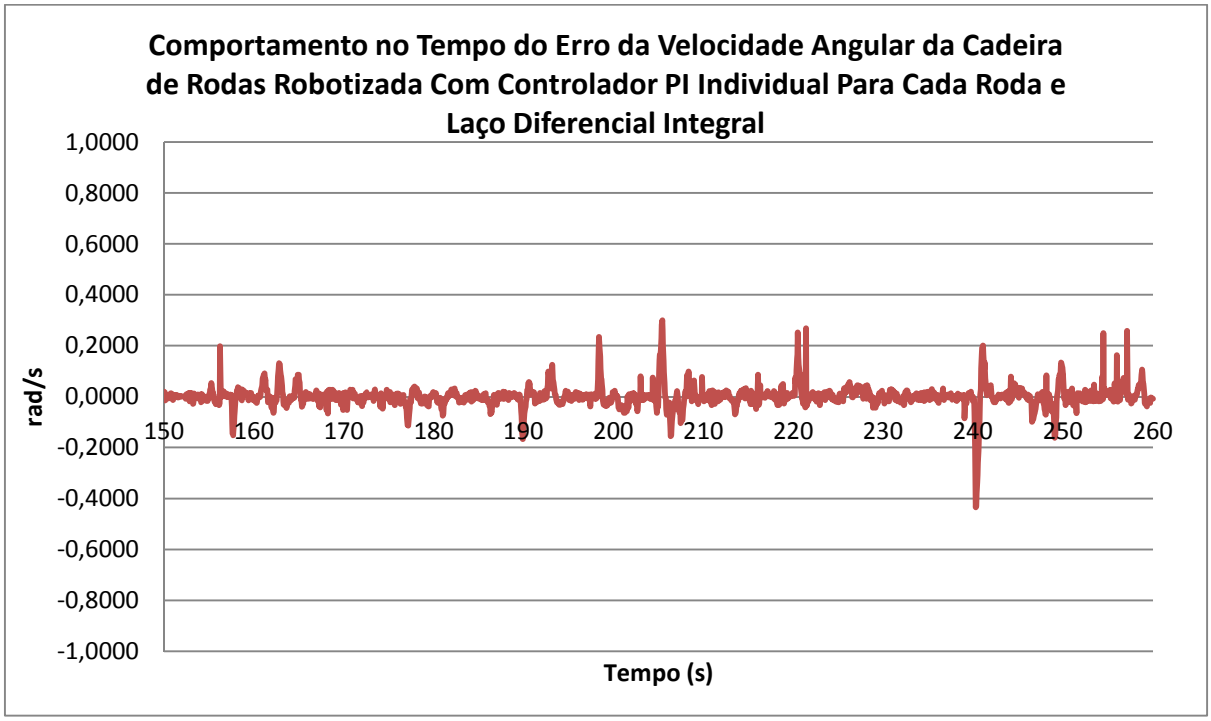


Figura 110: Comportamento no Tempo do Erro da Velocidade Angular da Cadeira de Rodas Robotizada Com Controlador PI Individual Para Cada Roda e Laço Diferencial Integral.

7 O SUPERVISÓRIO DE CONFIGURAÇÃO E MONITORAMENTO DA CADEIRA DE RODAS

Objetivando prover uma infraestrutura de configuração, testes e depuração de erros para a cadeira de rodas foi criado um supervisório executável em qualquer PC, proporcionando assim uma interface homem máquina amigável e eficiente dos usuários com a cadeira de rodas. Este Capítulo aborda o funcionamento e detalhes construtivos deste supervisórios.

7.1 O Supervisório: Considerações Gerais

Os sistemas supervisórios permitem que sejam monitoradas e rastreadas informações de um processo produtivo ou instalação física. Tais informações são coletadas através de equipamentos de aquisição de dados e, em seguida, manipulados, analisados, armazenados e, posteriormente, apresentados ao usuário. Estes sistemas também são chamados de SCADA (*Supervisory Control and Data Aquisition*). Os primeiros sistemas SCADA, basicamente telemétricos, permitiam informar periodicamente o estado corrente do processo industrial, monitorando sinais representativos de medidas e estados de dispositivos, através de um painel de lâmpadas e indicadores, sem que houvesse qualquer interface aplicacional com o operador [54].

Atualmente, os sistemas de automação industrial utilizam tecnologias de computação e comunicação para automatizar a monitoração e controle dos processos industriais, efetuando coleta de dados em ambientes complexos, eventualmente dispersos geograficamente, e a respectiva apresentação de modo amigável para o operador, com recursos gráficos elaborados (interfaces homem-máquina) e conteúdo multimídia. Para permitir isso, os sistemas SCADA identificam os *tags*, que são todas as variáveis numéricas ou alfanuméricas envolvidas na aplicação, podendo executar funções computacionais (operações matemáticas, lógicas, com vetores ou strings, etc) ou representar pontos de entrada/saída de dados do processo que está sendo controlado. Neste caso, correspondem às variáveis do processo real (ex: temperatura,

nível, vazão etc), se comportando como a ligação entre o controlador e o sistema. É com base nos valores das *tags* que os dados coletados são apresentados ao usuário. gravação de registros em Bancos de Dados, ativação de som, mensagem, mudança de cores, envio de mensagens por *pager*, *e-mail*, celular, etc [54].

7.2 A Aplicação de Supervisório na Cadeira de Rodas

A cadeira de rodas é um sistema de composto de vários componentes e *softwares*, sendo por diversas razões necessárias as configurações destes componentes. A cadeira de rodas robotizada tem por objetivo maior prover uma plataforma robótica para aplicações tecnológicas voltadas para pessoas com graves deficiências motoras. Cada possível aplicação para esta plataforma robótica pode necessitar de ajustes e configurações personalizados desta. O supervisório pretende evitar a necessidade de intervenções em nível de *firmware* das placas da cadeira de rodas, para tais ajustes e configurações.

Outra funcionalidade do supervisório está na depuração de erros e testes da cadeira de rodas, visto que para o desenvolvimento de outras aplicações é interessante dispor de uma interface operacional da plataforma robótica.

O supervisório é um programa desenvolvido em *Visual Basic* (VB) usando o ambiente de programação *Microsoft Visual Studio* 2008. O grande atrativo para o uso do VB é a facilidade de criação de interfaces gráficas e qualidade destas.

A comunicação entre supervisório e cadeira de rodas é realiza pela interface UART da Placa de Processamento desta. Na seção 3.8 foi abordado o protocolo de comunicação entre cadeira de rodas e aplicação de alto nível, sendo este protocolo usado pelo supervisório para comunicação com a cadeira de rodas.

7.3 As Telas do Supervisório

O supervisório é composto de uma instância principal que é responsável pela gestão da porta serial de comunicação com a cadeira de rodas, e por suportar as telas das aplicações específicas deste.

Esta tela permite também a conexão com a cadeira de rodas. Esta função pode ser executada no menu “Conectar → Conectar a CRI”. Da mesma maneira pode se desconectar a cadeira de rodas acessando-se o menu “Conectar → Desconectar a CRI”.

No canto esquerdo inferior da tela da instância principal é mostrado o *status* de conexão da cadeira de rodas, sendo mostrado “CRI Desconectada” para cadeira de rodas não conectada e “CRI Conectada” para cadeira de rodas conectada.

A instância principal ao executar a função de conectar a CRI abre porta COM1 do PC e envia um *ping* (Tabela 6) para a cadeira de rodas, se ao final de 500 *ms* a não houver resposta deste *ping* a aplicação fecha a porta COM1 e abre a COM2 realizando o mesmo procedimento. Esta seqüência é realizada até o teste da COM30 e se não houver resposta da cadeira de rodas a aplicação informa ao usuário “CRI não encontrada”. Se a CRI for encontrada em uma das portas testadas é informado ao usuário a mensagem “CRI encontrada na porta COMxx”.

A instância principal, após estabelecimento da comunicação com a CRI, realiza teste de conexão com a cadeira de rodas a cada 200 *ms* enviando o comando *ping* e aguardando retorno. A mensagem de *status* da conexão com a cadeira de rodas é atualizada com a resposta deste procedimento.

7.3.1 O Ajuste do RTC da Cadeira de Rodas

Como mencionado na seção 3.7 a Placa de Processamento dispõe de um RTC, sendo sua configuração realizada pelo supervisório usando a instância Relógio no menu *setup* da instância principal. A tela da instância relógio está ilustrada na Figura 111.

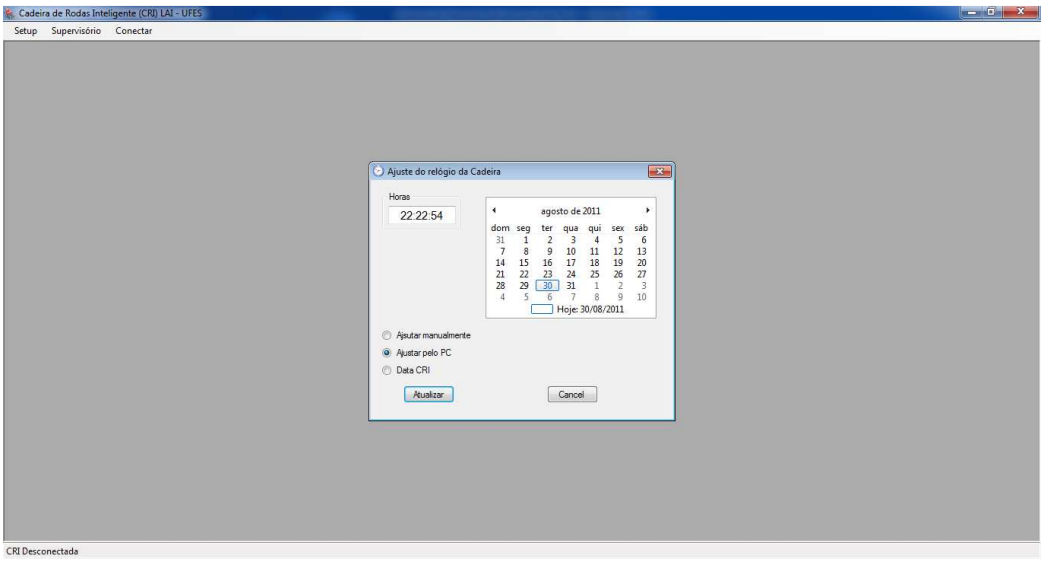


Figura 111: Tela de visualização da instância relógio.

Esta tela implementa três funções:

- Visualizar a data e hora atuais do RTC da Placa de Processamento. Nesta função o campo de hora e o calendário desta tela mostram a hora e a data atuais da cadeira de rodas respectivamente.
- Ajustar hora e data da cadeira de rodas manualmente. Nesta função o campo hora e o calendário desta tela fica habilitados para serem editados. Ao ajustar estes campos deve-se clicar no botão atualizar para enviar os dados para a cadeira de rodas.
- Ajustar hora e dada da cadeira de rodas com os valores de hora de data atuais do PC. Nesta função o campo hora e o calendário desta tela mostram a hora e a data do PC. Assim, deve-se clicar no botão atualizar para enviar os dados para a cadeira de rodas.

7.3.2 O Ajuste dos Parâmetros do Controlador de Baixo Nível da Cadeira de Rodas

Para configuração e sintonia do controlador de baixo nível da cadeira de rodas foi criada uma tela no supervisório. Nesta tela é possível ajustar os três parâmetros que compõe o controlador de baixo nível. A Figura 112 ilustra uma instância desta tela, sendo nela possível

observar os campos dos parâmetros do controlador, um campo de *status* do controlador, um campo de *set-point* de velocidade angular e linear e um campo de comandos para a cadeira de rodas.

Nos campos dos parâmetros do controlador de baixo nível são mostrados os valores atuais destes quando o comando do controlador é de visualização. Quando o comando é de “atualizar” ou “atualizar e salvar parâmetros” os campos dos parâmetros do controlador permitem sua edição, e para enviar estes valores para a cadeira de rodas basta pressionar o botão “*OK*” no campo de comandos.

Os campos de *set-points* desta tela são usados para enviar valores de velocidade linear e angular para a cadeira de rodas a fim de testá-la. Estes campos podem ser editados ou apenas lidos do controlador da cadeira de rodas, sendo que para editar basta selecionar a opção “*Write*” nesta tela, editar os campos para os valores desejados e então pressionar do botão “OK” do campo de *set-point*. Se a necessidade for de ler os valores atuais de *set-point* basta selecionar a opção de “*Read*” no mesmo campo.

O campo de comandos para o controlador além de prover as funções anteriormente citadas de edição dos parâmetros, ainda permite desligar o controlador, ligar o controlador, fazer o *reset* do controlador e ainda limpar as falhas do controlador. Para selecionar qualquer destas opções basta selecioná-la no campo de comandos do controlador e pressionar o botão de “*OK*” do mesmo campo.

Esta tela ainda prove uma visualização rápida do *status* da cadeira de rodas. Assim, é possível visualizar no campo de *status* do controlador todos os *bits* da palavra de *status* do controlador, sendo o *bit* em estado lógico 1 representado pela cor vermelho e o *bit* no estado lógico 0 representado pela cor verde.

Para acessar a tela do controlador da cadeira de rodas basta acessar o menu “Setup → Controlador”.

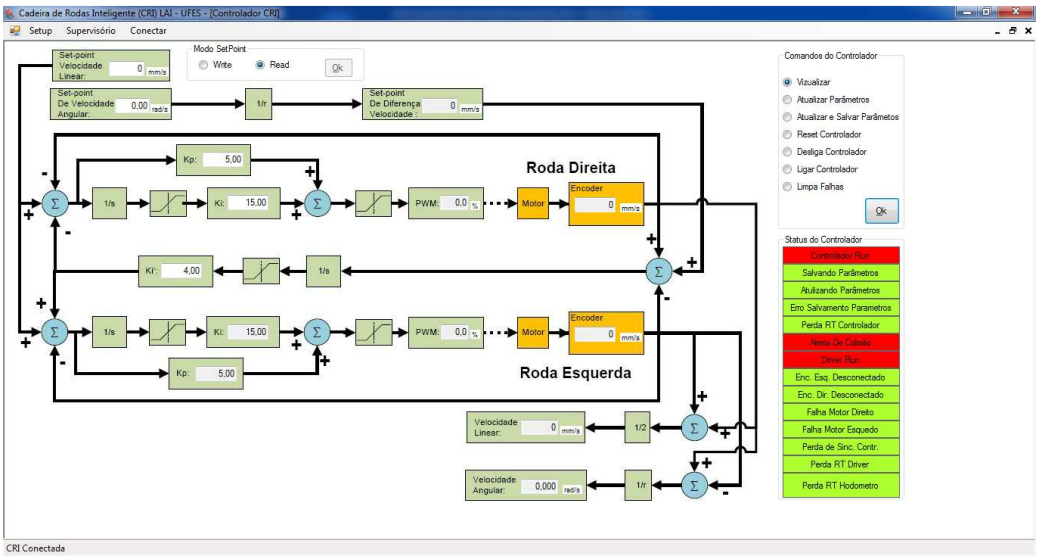


Figura 112: Tela do supervisório de configuração do controlador de baixo nível da cadeira de rodas.

7.3.3 Tela de Controle do Cartão Micro-SD

Objetivando prover uma interface de controle e visualização do *status* do cartão micro-SD de *log* da cadeira de rodas foi criada uma tela no supervisório específica para este fim. Nesta tela é possível acompanhar os valores atuais de cada um dos dois *buffers* de escrita e seus respectivos valores máximos atingidos, enviar comandos para o cartão micro-SD e observar o statu do mesmo.

Os comandos do cartão micro-SD incluem:

- Ativar *log*, sendo este comando utilizado para retomar o processo de escrita no cartão micro-SD;
- Parar *log*, sendo este comando utilizado para parar o processo de escrita no cartão micro-SD;
- Remover Cartão, sendo este comando utilizado para desmontar o cartão micro-SD e prover retirada segura do mesmo;
- Reset DOSonCHIP, sendo este comando utilizado para reiniciar todo o processo de escrita inclusive o DOSonCHIP;

No campo de *status* do cartão micro-SD é possível visualizar qual é seu *status*, e na ocorrência de alguma falha é possível visualizá-la neste campo e no campo de erro.

Para acessar a tela de controle do cartão micro-SD basta acessar o menu “Setup → SD Card”.

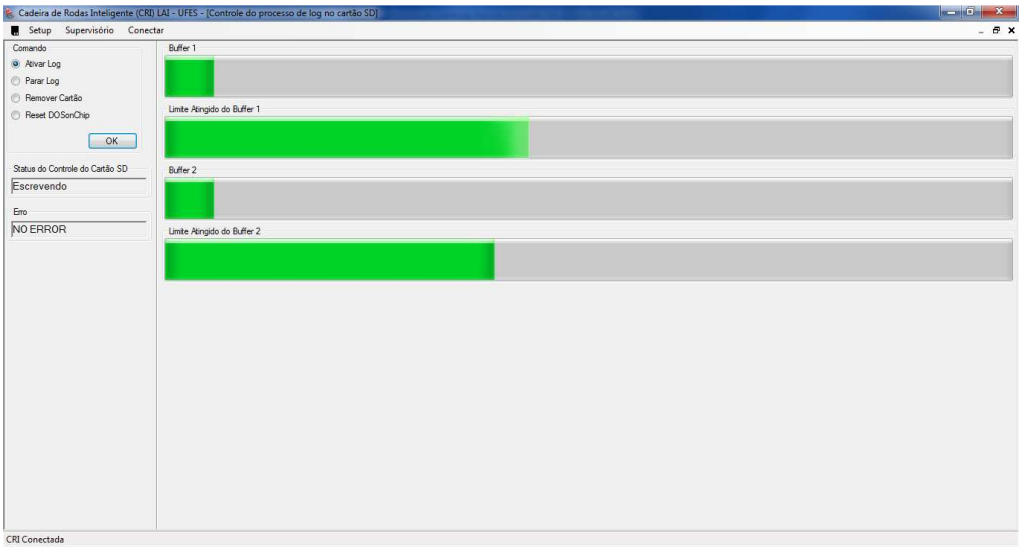


Figura 113: Tela do supervisório de controle do cartão micro-SD.

7.3.4 A Tela de Controle Principal do Supervisório

A tela principal do superveisório foi desenvolvida para ser a tela de teste da cadeira de rodas. Nela é possível enviar comando para o controlador, verificar sua palavra de *status*, visualizar as informações de valores de PWMs, visualizar os valores de velocidade das duas rodas, visualizar a velocidade angular, visualizar a velocidade linear, visualizar o *status* do cartão micro-SD, visualizar a tensão das baterias da cadeira de rodas, visualizar os *flags* de erros e registros de erros da rede CAN, visualizar os valores de distâncias medidas pelos sensores infravermelhos e enviar comandos de velocidades angular e linear para o controlador.

O campo de comandos para o controlador permite desligar o controlador, ligar o controlador, fazer o *reset* do controlador e ainda limpar as falhas do controlador. Para selecionar qualquer destas opções basta selecioná-la no campo de comandos do controlador e pressionar o botão de “*OK*” do mesmo campo.

A visualização do *status* da cadeira de rodas é possível através do campo de *status* do controlador, no qual todos os *bits* da palavra de *status* do controlador são mostrados, sendo o *bit* em estado lógico 1 representado pela cor vermelho e o *bit* no estado lógico 0 representado pela cor verde.

O campo de tensão das baterias mostra seu valor em *Volts* e o campo de *status* do cartão micro-SD mostra seu *status*. Foram criados dois campos para visualização dos valores de PWMs para os motores. Estes campos dispõem de uma caixa de texto e uma barra de rolagem para cada PWM que mostram seus valores em percentuais.

Os campos IR 0, IR 1, IR 2 e IR 3 mostram os valores em milímetros dos sensores infravermelho1, 2, 3 e 4 respectivamente.

Os campos “Vre”, “Vrd”, “Vel. Ang.” e “Vel. Lin” mostram os valores de velocidades da roda esquerda, velocidade da roda direita, velocidade angular e velocidade linear respectivamente, sendo estes valores mostrados em milímetros por segundo para o caso das velocidades das rodas e velocidade linear, e radianos por segundo no caso da velocidade angular. Estes campos dispõem de uma caixa de texto e uma barra de rolagem para cada roda.

Os campos de *set-point* de velocidade angular e linear foram implementados com campo de texto e barra de rolagem para facilitar os testes. Foram criados também botões “Zero” que fazem os valores de *set-point* iguais a zero de modo mais rápido, e um botão de “Set” no qual é possível inserir o valor desejado na caixa de texto e depois pressionar este botão para enviar o valor desejado para a cadeira de rodas.

O campo “CAN *Driver*” mostra os valores dos registros de erros da rede CAN, sendo “REC” o registro contador de erros de recepção e “TEC” o registro contador de erros de transmissão. Este campo ainda ilustra os *flags* de erros CAN, sendo estes:

- RX1OVR indica a ocorrência de sobre-escrita no *buffer* 1;
- RX0OVR indica a ocorrência de sobre-escrita no *buffer* 0;
- TXBO indica que o nó de rede está desconectado da mesma;
- TXEP indica que o TEC atingiu o valor de 128 erros. Neste estado o controlador CAN do nó entra em modo de *error-passive*, sendo então capaz

de enviar e receber dados, contudo, quando detectar erros ele não envia o *frame* de erros;

- RXEP indica que o REC atingiu o valor de 128 erros. Neste estado o controlador CAN do nó entra em modo de *error-passive*, sendo então capaz de enviar e receber dados, contudo, quando detectar erros ele não envia o *frame* de erros;
- TXWAR indica que o TEC atingiu o valor de 96 erros ;
- RXWAR indica que o REC atingiu o valor de 96 erros;
- EWARN indica que o TEC ou o REC atingiram o valor de 96 erros;

É possível visualizar o estado de cada *bit* da palavra de erros CAN também por um código de cores, sendo o *bit* em estado lógico 1 representado pela cor vermelho e o *bit* no estado lógico 0 representado pela cor verde.

Durante os testes com a cadeira de rodas é possível a ocorrência de situações de emergência, nas quais é necessário que os motores sejam desligados de forma rápida. Objetivando prover este nível de segurança foi disponibilizado um botão de “Emergência” que se pressionado desliga os motores da cadeira de rodas e para o controlador.

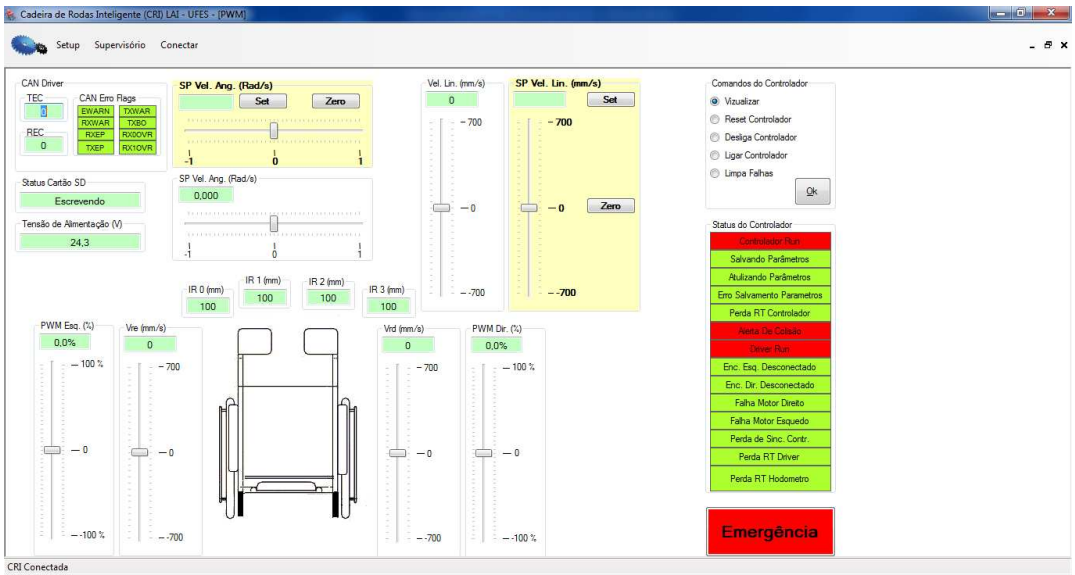


Figura 114: Tela de controle principal do supervisório.

7.4 Conclusão

Durante os testes práticos o supervisório se mostrou de extrema necessidade devido à facilidade de configuração e depuração de erros que ele propicia. Sua principal funcionalidade é permitir que sejam realizados ajustes personalizados par o usuário de forma rápida, fácil e confiável. Isto devido ao fato que nesta fase do projeto da Cadeira de Rodas Robotizada existe a necessidade de preparação de uma plataforma que possa ser comercializada. Assim, o projeto deste supervisório leva ao usuário final a possibilidade de configurar, testar e gerenciar sua Cadeira de Rodas Robotizada de maneira fácil e amigável.

CONCLUSÃO

Neste trabalho foi desenvolvida uma plataforma robótica para uma cadeira de rodas motorizada, sendo esta plataforma capaz de controlar de modo seguro e eficiente as velocidades angular e linear, armazenar dados gerais em cartão de memória, fazer interface com sensores de navegação, tais como sensores infravermelho [23] e sonar [27] e capaz de promover o deslocamento desta de forma segura para o usuário.

O usuário foi preponderante nas decisões deste trabalho, visto que sua condição de movimentos limitados ou nulos, exige da Cadeira de Rodas Robótica elevado grau de segurança. Assim, foram usadas varias técnicas de segurança, tais como: verificação de integridade dos *encoders*, limitação de velocidade angular em 0,6 rad/s e velocidade linear em 700 mm/s, verificação de execução em tempo real de tarefas criticas, verificação de *status* da rede CAN e dos dados que trafegam por ela e verificação da integridade dos motores.

Os acionamentos dos motores da cadeira de rodas utilizaram técnicas consagradas de chaveamento, sendo utilizadas pontes H para controlar a tensão média dos motores de corrente contínua de excitação independente que tracionam as rodas da cadeira. Este utiliza apenas transistores do tipo MOSFETs canal n, sendo este fato justificado pela menor impedância dos MOSFETs canal n em relação aos MOSFETs canal p. Esta escolha obrigou o uso de CIs *gate drivers* [16] que são capazes de prover chaveamento dos MOSFETs superiores das pontes H sem uso de fontes auxiliares. Esta topologia apresenta uma limitação: o PWM da ponte nunca pode assumir 100 % ou 0 % de ciclo, ou seja, deve-se prover de modo ininterrupto chaveamento dos transistores da ponte H, sob pena de queima dos transistores da parte superior das pontes H por dessaturação.

O projeto das placas de acionamento mostrou-se eficiente durante os testes, visto que observou-se baixo aquecimento dos MOSFETs e ainda foi capaz de prover para os motores alimentação controlada e regular. Contudo, notou-se grande irradiação de ruídos de 20 kHz. Isto foi possível verificando os sinais biológicos do usuário e os sinais dos *encoders* no osciloscópio. Após testes e avaliações constatou-se que boa parte da irradiação era oriunda dos dissipadores dos transistores das pontes H, visto que estes dissipadores não estavam

aterrados. A solução para o problema foi o aterramento destes dissipadores (Figura 115), sendo constatado após este procedimento uma redução significativa do nível de ruído gerado.

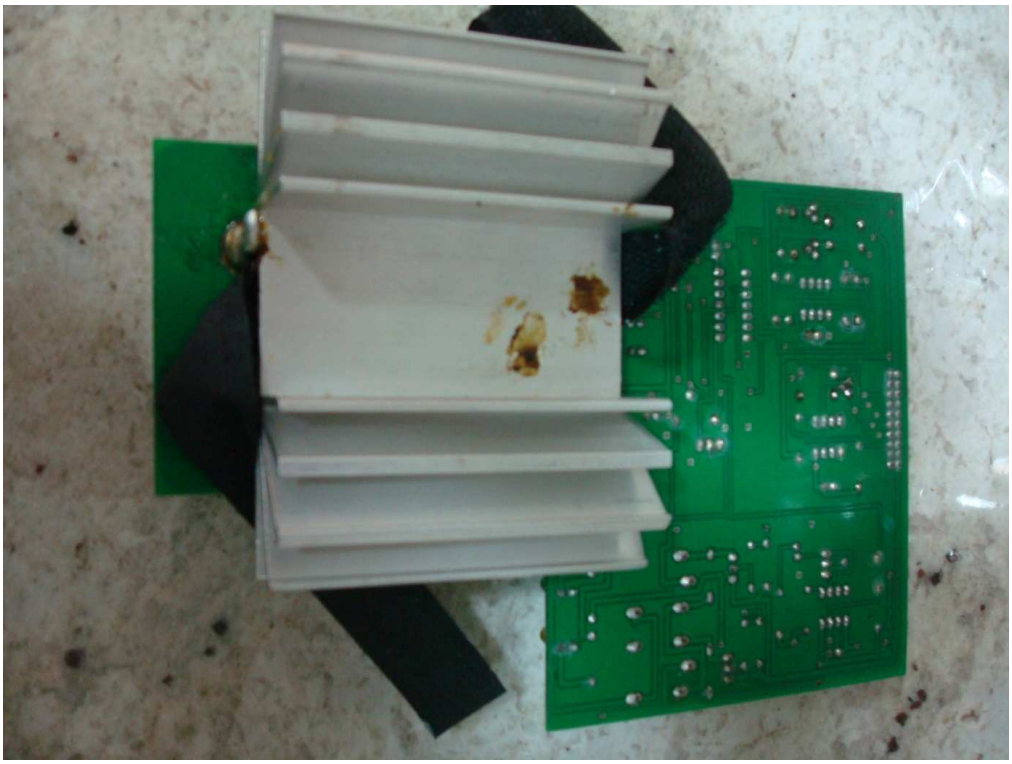


Figura 115: Fotografia do dissipador de uma das placas de acionamento dos motores da cadeira de rodas.

As placas de controle dos periféricos e de processamento apresentaram desempenho satisfatório visto que, com exceção da interface dos sensores infravermelho e da fonte chaveada que necessitam de ajustes técnicos, todo o *hardware* implementado nestas atenderam as expectativas.

Contudo, na observação da Tabela 15 e Tabela 16 nota-se que as rotinas de controle do barramento demandam custo alto para o microcontrolador MSP430F1611, visto que este opera a 16 MHz. Outro fato relevante é o tempo de chaveamento do *kernel* de tempo real, que necessita de 520 ciclos de *clock* para sua execução, e para este microcontrolador o tempo médio de chaveamento é de 33 µs. Considerando-se que o menor pacote de dados CAN é o RTS com 47 *bits* e que a taxa de sinalização CAN usada neste trabalho é 500 MHz, o tempo de envio deste pacote é de 94 µs. Assim, para esta situação, o tempo de chaveamento de tarefas corresponde a aproximadamente 35% do tempo de transito deste pacote CAN. Por este motivo, além do fato do alto custo das tarefas de controle CAN, é recomendado para

aplicações futuras o uso de microcontroladores com *clock* mais elevado. Para o uso da mesma arquitetura presente neste trabalho, a adição de novas tarefas é recomendada mediante a adição de novos nós de rede.

O controlador se mostrou adequado para a aplicação, visto que em todos os testes se mostrou estável, seguro e eficiente no que tange o controle das velocidades angular e linear da Cadeira de Rodas Robótica. Contudo, sua parametrização deve ser realizada de acordo com cada aplicação específica da cadeira de rodas, visto que para realização dos testes o ajuste dos seus parâmetros foi realizado para alto desempenho, ou seja, baixo tempo de resposta (Figura 106 e Figura 107).

Outro ponto de destaque deste trabalho foi a implementação do supervisório de controle. Esta aplicação se mostrou extremamente útil para a depuração de erros e ajuste do controlador.

O trabalho descrito em [9] se assemelha a este projeto, desde a arquitetura até os sensores. Outro fato semelhante e relevante é que ambos os trabalhos utilizam uma arquitetura de comunicação baseada em rede. Contudo, em [9] foi utilizada a rede *LonWorks*, e neste trabalho foi utilizada rede CAN. Quanto a este fato não existe significância, visto que ambas as redes são robustas e confiáveis. O principal fato que diferencia este trabalho de [9] e dos demais apresentados no estado da arte é que apenas neste pode ser vista a arquitetura de variáveis de barramento, que integra o *kernel* de tempo real e as funcionalidades da rede CAN para criar um ambiente no qual as tarefas podem ser executadas em qualquer nó de rede e suas variáveis são atualizadas em tempo real para todos os demais. Isso permite, entre outras funcionalidades, que existam redundâncias entre nós de rede, visto que quando um nó falhar outro pode assumir seu lugar. Isso porque tarefas interdependentes entre si podem estar alocadas em nós diferentes na rede, porem o acesso a suas variáveis é realizado por cada uma como se ambas estivessem no mesmo nó. Na Figura 116 é possível observar um exemplo do funcionamento das variáveis de barramento, no qual a tarefa controlador executada na PP envia dados para a tarefa *driver* motores e recebe dados da tarefa hodometro, ambas sendo executadas na PCP. Nesta Figura é possível observar ainda todos os dispositivos e tarefas que realizam esta conexão virtual entre tarefas.

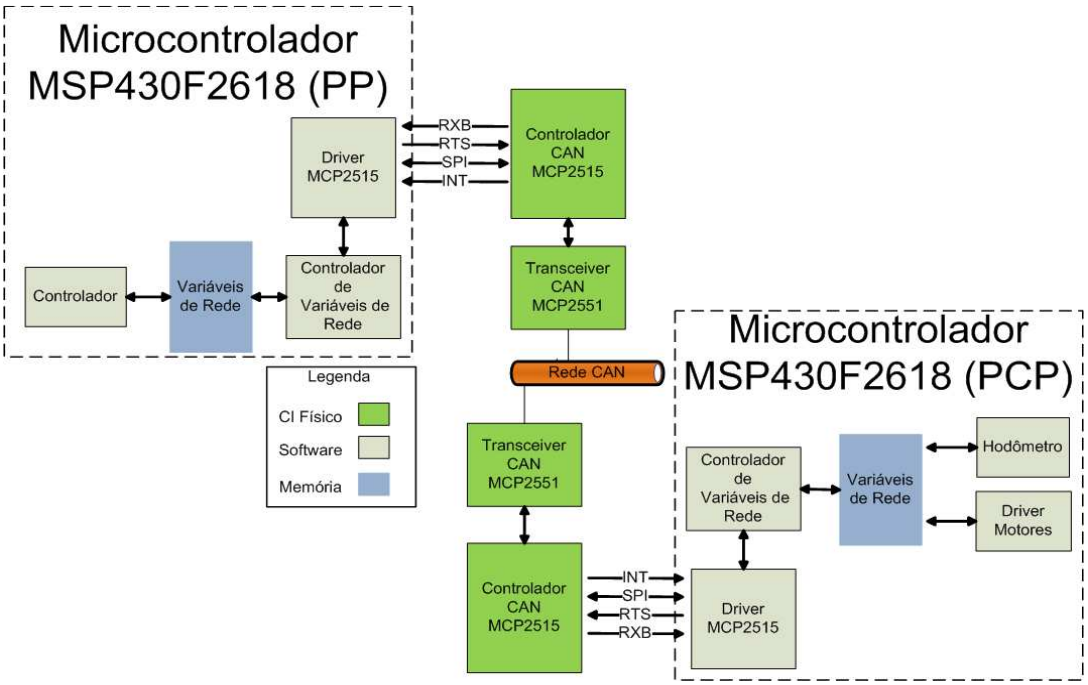


Figura 116: Ilustração do funcionamento da arquitetura de variáveis de barramento.

Ainda existe na arquitetura um sistema de detecção de falhas que durante todos os testes se mostrou robusto e confiável. Este funcionalidade não foi observada em nenhum outro trabalho estudado, assim como o uso de rede CAN para aplicações de robótica de acessibilidade.

Diante do exposto neste trabalho, pode se afirmar que a robotização da cadeira de rodas foi realizada em sua plenitude, visto que permite que qualquer outra aplicação de alto nível seja acoplada à cadeira de rodas apenas enviando comando simples de velocidade angular e linear, sendo então possível para esta aplicação a total abstração de técnicas de controle de robôs móveis, eletrônica de potência, redes de comunicação, sensoriamento e segurança do usuário.

Vale ressaltar que as seguintes publicações foram realizadas, frutos das pesquisas desta dissertação de mestrado: [55], [56], [57], e [58]

Além disso, existe o registro de patente referente a parte do trabalho demonstrado nesta dissertação de mestrado, de título: “Sistema Assistivo de Interface Homem Máquina”

Trabalhos Futuros

A plataforma robótica desenvolvida neste trabalho é de forma geral generalista, ou seja, pode ser adaptada para diversas aplicações. Estas adaptações se tratam principalmente do posicionamento dos sensores infravermelho e do sonar na cadeira de rodas. O posicionamento dos sensores infravermelho [23] e dos transdutores do sonar [27] devem obedecer a critérios específicos de cada aplicação, contudo, devido ao fato dos sensores infravermelho possuírem faixa de medição de distância de 10 a 90 cm, é recomendado que estes sejam posicionados voltados para o chão, visando a detecção de buracos ou degraus. Opostamente, os transdutores da série 6500 da Polaroid, utilizados em [27], possuem faixa de atuação de 30 cm a 12 m. Assim, é recomendado que estes sensores sejam dispostos de modo circular à cadeira de rodas, objetivando, portanto, a detecção de obstáculos durante a movimentação desta.

Além disso, a PCP possui entradas analógicas de expansão que podem conectar-se a módulos de medição de corrente do tipo efeito *Hall*. Estes módulos presentes em [59] possibilitam a medição de corrente para os motores da cadeira de rodas sem as perdas resistivas intrínsecas dos métodos baseados em resistores série.

BIBLIOGRAFIA

1. **Neto Frizera, Anselmo, et al.** Human-Machine Interface Based on Electro-Biological Signals for Mobile Vehicle Control. Montreal, Canadá : Industrial Electronics, IEEE International Symposium, 2006.

2. **Texas Instruments.** MSP430x2xx Family-User's Guide. Dallas : s.n., 2007. SLAU144D.

3. **IAR Systems.** IAR Embedded Workbench®. [Online] [Citado em: 01 de 09 de 2011.] http://www.iar.com/website1/1.0.1.0/50/1/.

4. **Röfer, Thomas e Lankenau, Axel.** *Architecture and Applications of the Bremen Autonomous Wheelchair.* 1998.

5. **POLAROID Corporation.** Ultrasonic ranging system / technical assistence. 199.

6. **Martín, P., Sanz, R. e Puente, E.A.** *Increasing intelligence in autonomous wheelchairs.* s.l. : Journal of Intelligent and Robotic Systems. Nº 22, pp: 211-232, 1998.

7. **Bourhis, G. e Agostini, Y.** *The Vahm Robotized Wheelchair: System Architecture and Human-Machine Interaction.* s.l. : Journal ofIntelligent and Robotic Systems. Nº 22,pp: 39-50, Kluwer Academic Publishers, 1998.

8. —. *Man-machine Cooperation for the Control of an Intelligent Powered Wheelchair.* s.l. : Journalof Intelligent and Robotic Systems. Nº 22, pp: 269-287, Kluwer Academic Publishers, 1998.

9. *An Integral System for Assisted Mobility.* **Mazo, Manuel;RESEARCH GROUP OF THE SIAMO PROJECT.** s.l. : IEEE Robotics & Automation Magazine, 2001. 1070-9932/01/$10.00©2001IEEE.

10. **Texas Instruments.** *Application Report: Introduction to the Controller Area Network (CAN).* 2002. SLOA101.

11. *Introduction to the LonWorks® System.* **ECHELON Corporation.** 1999. 078-0183-01A.

12. **Yanco, Holly A., et al.** Initial Report on Wheelesley: A Robotic Wheelchair System. Wellesley, MA : Department of Computer Science Wellesley College. 02181.

13. **Miller, David P. e Slack, Marc G.** Design and Testing of a Low-Cost Robotic Wheelchair Prototype. Boston : Kluwer Academic Publishers, 1995.

14. **Patsko, Luís Fernando.** Tutorial Montagem da Ponte H. *maxwellbohr*. [Online] 18 de 12 de 2006. [Citado em: 15 de 07 de 2011.] http://www.maxwellbohr.com.br/downloads/robotica/mec1000_kdr5000/tutorial_eletronica_-_montagem_de_uma_ponte_h.pdf.

15. **Spada, Adriano Luiz.** O Ouvido Humano. *Attack do Brasil.* [Online] 2011. [Citado em: 15 de 07 de 2011.] http://www.attack.com.br/artigos_tecnicos/ouvido_humano.pdf.

16. **International Rectifier.** Data Sheet: HALF-BRIDGE GATE DRIVER IC IR2114SSPbF/IR21141SSPbF/IR2214SSPbF/IR22141SSPbF. 18 de 5 de 2006. PD60213 revG.

17. **JONES, Joseph L, FLYNN, Anita M e SEIGER, Bruce A.** *Mobile Robots: inspiration to implementation.* Massachusetts : Wellesley, 2001.

18. **INTERNATIONAL RECTIFIER (IRF).** Data Sheet: IRF3205. 2001. PD-91279E.

19. **Philips Semiconductors.** *1N4148; 1N4448 High-speed diodes.* 2004.

20. **Fairchild Semiconductor Corporation.** *Single-Channel: 6N135, 6N136 , HCPL-2503, HCPL-4502 Dual-Channel: HCPL-2530, HCPL-2531 Rev. 1.0.5.* 2005.

21. **Texas Instruments.** *MSP430x241x, MSP430x261x MIXED SIGNAL MICROCONTROLLER.* s.l. : TEXAS INSTRUMENTS, 2007. SLAS541A.

22. **Microchip Technology Inc.** *SPI™ Overview and Use of the PICmicro Serial Peripheral Interface*. 2000.

23. **SHARP.** *GP2D02 Compact, High Sensitive Distance Measuring Sensor*. Data Sheet GP2D02.

24. **Computer Optical Products, Inc.** Data Sheet:CP-300 Series Housed Encoders. CP350.

25. **HITACHI.** *HD44780U (LCD-II) (Dot Matrix Liquid Crystal Display Controller/Driver).* Data Sheet HD44780U.

26. **Texas Instruments.** *Data Sheet 3V TO 5,5V MULTICHANNEL RS-232 LINE DRIVER/RECEIVER WITH 15kV ESD (HBM) PROTECTION*. 2009. SLLS350L.

27. **Freire, Eduardo Oliveira.** Desenvolvimento de um Sistema de Sensoriamento Ultra-Sônico para Robô Móvel. Vitória : UFES-Universidade Federal do Espírito Santo, 1997.

28. **Microchip Technology Inc.** *Data Sheet: MCP2551 High-Speed CAN Transceiver*. 2007. DS21667E.

29. —. *Data Sheet: MCP2515 Stand-Alone CAN Controller With SPI™ Interface.* 2003. DS21801B.

30. **Philips Semiconductors.** HCMOS family characteristics-FAMILY SPECIFICATIONS. 1988. Data Sheet HCMOS family characteristics.

31. **Kofuji, Prof. Dr. Sérgio Takeo.** *Fontes Chaveadas.* Campinas : s.n., 2004. CAPÍTULO 3.

32. **Texas Instruments.** *TPS5430/TPS5431 .3-A, WIDE INPUT RANGE, STEP-DOWN SWIFT™ CONVERTER*. 2006. SLVS632C.

33. —. *Data Sheet REG1117/REG1117A 800mA and 1A Low Dropout Positive Regulator 1.8V, 2.5V, 2.85, 3.3V, 5V, and Adjustable.* Dallas : s.n., 1992. SBVS001D.

34. **Philips Semiconductors.** DATA SHEET 74HC/HCT244 Octal buffer/line driver; 3-state. 1990.

35. **Texas Instruments.** *Application Report: Features of the MSP430 Bootstrap Loader*. 2006. SLAA089D.

36. —. *Application Report: Application of Bootstrap Loader in MSP430 With Flash Hardware and Software Proposal.* 2006. SLAA096D.

37. **Acroname Easier Robotics.** Sharp GP2D02 Interface to Basic Stamp II. *Acroname Examples.* [Online] 26 de 04 de 2005. [Citado em: 12 de 12 de 2007.] http://www.acroname.com/robotics/info/examples/GP2D02-4/GP2D02-4.html.

38. **DOSonCHIP™.** *Data Sheet CD17B10 Memory Card Interface with File System.* s.l. : WEARABLE INC., 2009.

39. **Fairchild Semiconductor International.** *Data Sheet KA78XX/KA78XXA 3-terminal 1A positive voltage regulator.* 2000.

40. **DALLAS Semiconductor.** *Data Sheet DS1307 64 x 8 Serial Real-Time Clock.*

41. **Matias, Juliano.** Encoders. [Online] Saber, 2002. [Citado em: 09 de julho de 2007.] http://www.mecatronicaatual.com.br/artigos/cnc/.

42. **Celeste, Wanderley Cardoso.** Um Sistema Autônomo Para Navegação de Cadeiras de Rodas Robóticas Orientadas a Pessoas Com Deficiência Motora Severa. Vitória : s.n., 2009.

43. **OMG.** UML Version 1.3. *OMG.* [Online] [Citado em: 2009 de 03 de 01.] www.ece.rutgers.edu/~parashar/Classes/00-01/.../2-StatechartDiagrams.pdf.

44. **Lava Computer MFG Inc.** RS-232: Serial Port. [Online] 2002. [Citado em: 04 de 04 de 2011.] http://www.lavalink.com/dev/fileadmin/dos/support/white_papers/rs_232_serial_ports.pdf.

45. **ISO - International Organization for Standardization .** Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling. *INTERNATIONAL STANDARD 11898-1*. 2003. ISO 11898-1:2003(E)/Cor.1:2006(E).

46. **Barros, Flávio Alencar do Rêgo.** *Protocolo CAN.* Niteroi - RJ : UFF - Universidade Federal Fluminence, 1998. CAA.

47. **Microchip Technology Inc.** *AN228: A CAN Physical Layer Discussion.* 2002. DS00228A.

48. —. *AN730: CRC Generating and Checking.* 2000. DS00730A.

49. **Farines, Jean-Marie, Fraga, Joni da Silva e de Oliveira, Rômulo Silva.** *Sistemas de Tempo Real.* Florianópolis : Departamento de Automação e Sistemas da Universidade Federal de Santa Catarina, 2000.

50. **Micrum.** µC/OS-II Kernel. *Micrum.* [Online] Micrum, 2000. http://www.micrium.com/page/products/rtos/os-ii.

51. **Micrium.** µC/OS-II Reference Manual. Chapter 16.

52. **Labrosse, Jean J.** *µC/OS-II: The Real Time Kernel.* s.l. : CMPbooks.

53. **LIU, C. L. e LAYLAND, JAMES W.** Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery.* 1973, Vol. 20.

54. **Silva, Ana Paula Gonçalves da e Salvador, Marcelo.** O que são sistemas supervisórios? 2005. RT 025.04.

55. **Bastos Filho, Teodiano Freire, et al.** Case Study: Cognitive Control of a Robotic. *Wearable Robots: Biomechatronic Exoskeletons.* Wiley : s.n., 2008. Vols. Chapter 9, Section 9.6.

56. **Guimarães, Edgard Batista, Filgueira, Patrick Noé dos Santos e Bastos Filho, Teodiano Freire.** Sensores de Indução e RFID Aplicados à Navegação Auto-Guiada

de Cadeiras de Rodas. Cartagena de Indias : s.n., 2008. V Congreso Iberoamericano sobre Tecnologías de Apoyo a la Discapacidad.

57. **Celeste, Wanderley Cardoso, et al.** Modelo Dinámico y Estructura de Control de una Silla de Ruedas Autónoma para Personas con Discapacidad Motora Severa. Río Gallegos : XII Reunión de Trabajo en Procesamiento y Control (RPIC 2007), 2007.

58. **Celeste, Wanderley Cardoso, et al.** MODELO DINÂMICO E CONTROLE DE UMA CADEIRA DE RODAS AUTÔNOMA PARA PESSOAS COM DEFICIÊNCIA MOTORA SEVERA. Florianópolis : VIII Simpósio Brasileiro de Automação Inteligente, 2007. Anais do VIII Simpósio Brasileiro de Automação Inteligente.

59. **SECON Sensores e instrumentos.** Grupo de Produtos / Sensores de Corrente AC/DC (Efeito HALL Realimentado). [Online] [Citado em: 01 de 09 de 2011.] http://www.secon.com.br/grupo.php?grupo_id=14.

ANEXO 1:ARQUIVO MCP2515DRIVER.H DO PROGRAMA DO DRIVER MCP2515 EXECUTADO NA PLACA DE PROCESSAMENTO

```
#include    <msp430x26x.h>
#include    "ucos_ii.h"

#define SPI_MODULO                  3           // 0 => UCA0, 1 => UCA1, 2 => UCB0, 3 => UCB1
#define TAXA_CAN                    2           // 0 => 1Mbps, 1 => 500 kbps, 2 => 250 kbps, 3 = > 125 kbps
#define TEMPO_LED_ERRO_ON           300
#define MCP2515DriverPriority       8          //Prioridade da tarefa
#define STKMCP2515Driver            300         // Número de Bytes da Pilha do Driver
#define CAN_RX_BUFFER_LEN           16          // 2, 4, 8, 16, 32 ...
#define CAN_TX_BUFFER_LEN           16          // 2, 4, 8, 16, 32 ...
#define MCP2515_STATUS_READ_TIMER       200


// INT Pin config
#define INT_BIT                             0X01
#define INT_PORT                            P2IN
#define INT_PORT_SEL                        P2SEL &= (~INT_BIT)
#define INT_PORT_DIR                        P2DIR &= (~INT_BIT)
// RXB0 Pin config
#define RXB0_BIT                            0X40
#define RXB0_PORT                           P2IN
#define RXB0_PORT_SEL                       P2SEL &= (~RXB0_BIT)
#define RXB0_PORT_DIR                       P2DIR &= (~RXB0_BIT)
// RXB1 Pin config
#define RXB1_BIT                            0X80
#define RXB1_PORT                           P2IN
#define RXB1_PORT_SEL                       P2SEL &= (~RXB1_BIT)
#define RXB1_PORT_DIR                       P2DIR &= (~RXB1_BIT)
// TXB0 Pin config
#define TXB0_BIT                            0X08
#define TXB0_PORT                           P2OUT
#define TXB0_PORT_SEL                       P2SEL &= (~TXB0_BIT)
#define TXB0_PORT_DIR                       P2DIR |= TXB0_BIT
// TXB1 Pin config
#define TXB1_BIT                            0X10
#define TXB1_PORT                           P2OUT
#define TXB1_PORT_SEL                       P2SEL &= (~TXB1_BIT)
#define TXB1_PORT_DIR                       P2DIR |= TXB1_BIT
// TXB2 Pin config
#define TXB2_BIT                            0X20
#define TXB2_PORT                           P2OUT
#define TXB2_PORT_SEL                       P2SEL &= (~TXB2_BIT)
#define TXB2_PORT_DIR                       P2DIR |= TXB2_BIT
// LED FAULT Pin config
#define LED_CAN_FAULT_BIT                   0X02
#define LED_CAN_FAULT_PORT                  P4OUT
#define LED_CAN_FAULT_PORT_SEL              P4SEL &= (~LED_CAN_FAULT_BIT)
#define LED_CAN_FAULT_PORT_DIR              P4DIR |= LED_CAN_FAULT_BIT
// LED CAN ACTIVE Pin config
#define LED_CAN_ACTIVE_BIT                  0X04
#define LED_CAN_ACTIVE_PORT                 P4OUT
#define LED_CAN_ACTIVE_PORT_SEL             P4SEL &= (~LED_CAN_ACTIVE_BIT)
#define LED_CAN_ACTIVE_PORT_DIR             P4DIR |= LED_CAN_ACTIVE_BIT

#if SPI_MODULO == 2
```

```
        #define FLAG_SPI_INT_TX         UCB0TXIFG
        #define FLAG_SPI_INT_RX         UCB0RXIFG
        #define SPI_RX_BUF              UCB0RXBUF
        #define SPI_TX_BUF              UCB0TXBUF
        #define SPISTAT                 UCB0STAT
        #define SPIBUSY                 UCBUSY
        #define REG_SPI_INT_FLAG        IFG2
        #define CSHIGH                  P3OUT |= 0x01
        #define CSLOW                   P3OUT &= (~0x01)
#endif

#if SPI_MODULO == 3
        #define FLAG_SPI_INT_TX         UCB1TXIFG
        #define FLAG_SPI_INT_RX         UCB1RXIFG
        #define SPI_RX_BUF              UCB1RXBUF
        #define SPI_TX_BUF              UCB1TXBUF
        #define SPISTAT                 UCB1STAT
        #define SPIBUSY                 UCBUSY
        #define REG_SPI_INT_FLAG        UC1IFG
        #define CSHIGH                  P5OUT |= 0x01
        #define CSLOW                   P5OUT &= (~0x01)
#endif

// register CAN VALUE

#if TAXA_CAN == 0
  #define CNF1_VALUE            0x00 // 0b00 000000 SJW = 1, BRP = 1
  #define CNF2_VALUE            0xD0 // 0b11 010 000 PS1 = 3, Tpr = 1
  #define CNF3_VALUE            0x02 // 0b00 XXX 010 PS2 = 3
#endif

#if TAXA_CAN == 1
  #define CNF1_VALUE            0xC0 // 0b11 000000 SJW = 2, BRP = 1
  #define CNF2_VALUE            0xEA // 0b11 101 010 PS1 = 6, Tpr = 3
  #define CNF3_VALUE            0x05 // 0b00 XXX 101  PS2 = 6
#endif

#if TAXA_CAN == 2
  #define CNF1_VALUE            0xC1 // 0b11 000001 SJW = 2, BRP = 2
  #define CNF2_VALUE            0xEA // 0b11 101 010 PS1 = 6, Tpr = 3
  #define CNF3_VALUE            0x05 // 0b00 XXX 101  PS2 = 6
#endif

#if TAXA_CAN == 3
  #define CNF1_VALUE            0xC3 // 0b11 000011 SJW = 2, BRP = 4
  #define CNF2_VALUE            0xEA // 0b11 101 010 PS1 = 6, Tpr = 3
  #define CNF3_VALUE            0x05 // 0b00 XXX 101  PS2 = 6
#endif


#define CANCTRL_CONFIG_VALUE    0x90
#define CANCTRL_NORMAL_VALUE    0x04
#define BFPCTRL_VALUE           0x0F
#define CANINTE_VALUE           0x1F
#define CANINTF_VALUE           0x00
#define TXRTSCTRL_VALUE         0x07
#define RXB0CTRL_VALUE          0x64 // 0b01100100
#define RXB1CTRL_VALUE          0x60
#define EFLG_VALUE              0x00



// Macros de Comados
////////////////////////////////////////////////////////////////////////////////
// LED FAULT
```

```
#define LED_CAN_FAULT_ON            LED_CAN_FAULT_PORT |= LED_CAN_FAULT_BIT
#define LED_CAN_FAULT_OFF           LED_CAN_FAULT_PORT &= (~LED_CAN_FAULT_BIT)
#define LED_CAN_FAULT_TEST          LED_CAN_FAULT_PORT & LED_CAN_FAULT_BIT


//LED Active
#define LED_CAN_ACTIVE_ON           LED_CAN_ACTIVE_PORT |= LED_CAN_ACTIVE_BIT
#define LED_CAN_ACTIVE_OFF          LED_CAN_ACTIVE_PORT &= (~LED_CAN_ACTIVE_BIT)
#define LED_CAN_ACTIVE_TEST         LED_CAN_ACTIVE_PORT & LED_CAN_ACTIVE_BIT

// CAN Comandos
#define TEST_INT_PIN                INT_PORT & INT_BIT
#define RXB0_PIN_TEST               RXB0_PORT & RXB0_BIT
#define RXB1_PIN_TEST               RXB1_PORT & RXB1_BIT
#define TX0_RTS_UP                  TXB0_PORT |= TXB0_BIT
#define TX0_RTS_DOWN                TXB0_PORT &= (~TXB0_BIT)
#define TX1_RTS_UP                  TXB1_PORT |= TXB1_BIT
#define TX1_RTS_DOWN                TXB1_PORT &= (~TXB1_BIT)
#define TX2_RTS_UP                  TXB2_PORT |= TXB2_BIT
#define TX2_RTS_DOWN                TXB2_PORT &= (~TXB2_BIT)
#define TX0_RTS_TEST                TXB0_PORT & TXB0_BIT
#define TX1_RTS_TEST                TXB1_PORT & TXB1_BIT
#define TX2_RTS_TEST                TXB2_PORT & TXB2_BIT


///
// Defines//

#define CAN_RX_BUFFER_MASKCAN_RX_BUFFER_LEN - 1
#define CAN_TX_BUFFER_MASK          CAN_TX_BUFFER_LEN - 1

#define RESET                       0xC0
#define WRITE                       0x02
#define READ                        0x03
#define BIT_MODIFY                  0x05
#define READ_RX_BUFFER              0x90
#define READ_STATUS                 0xA0

#define BUFFER_0_ALL        0
#define BUFFER_1_ALL        2
#define BUFFER_2_ALL        4
#define LOAD_TX_BUFFER              0x40
// register CAN ADDRESS
#define CANCTRL_ADDRESS             0x0F
#define CNF1_ADDRESS                0x2A
#define CNF2_ADDRESS                0x29
#define CNF3_ADDRESS                0x28
#define BFPCTRL_ADDRESS             0x0C
#define EFLG_ADDRESS                0x2D
#define CANINTE_ADDRESS             0x2B
#define CANINTF_ADDRESS             0x2C
#define CANSTAT_ADDRESS             0x0E
#define TXRTSCTRL_ADDRESS           0x0D
#define RXB0CTRL_ADDRESS            0x60
#define RXB1CTRL_ADDRESS            0x70
#define TEC_ADDRESS                 0x1C
#define REC_ADDRESS                 0x1D


#define TRUE                        1
#define FALSE                       0
#define ERRO                        1
#define NOERRO                      0
```

```
////////////////////////////////////////////////////////////////////////////
// Tipo de buffer para de recepção CAN
typedef struct Estrutura BufferRXCAN {
        char  Nada0;
        union{
                //unsigned int All;
                struct{
                        char Up;
                        char Down;
                }Byte;
                struct{
                        char  IdUp;             //Identificação da msg
                        char  BitDownId: 3;
                        char  IDE: 1;
                        char  RTS: 1;
                }Bit;
        }Id;
        char  ExtendUp;
        char  ExtendDown;
        char  Len;
        char  Data[8];
}Estrutura BufferRXCAN;

typedef struct Estrutura BufferTXCAN {
        char Comando;
        union{
                struct{
                        char Up;
                        char Down;
                }Byte;
                struct{
                        char  IdUp;             //Identificação da msg
                        char  BitDownId: 3;
                        char  IDE: 1;
                }Bit;
        }Id;
        char  ExtendUp;
        char  ExtendDown;
        struct{
                char  Len: 4;
                char  UnUsed: 2;
                char  RTS: 1;
        }Inf;
        char  Data[8];
}Estrutura BufferTXCAN;


struct{
        char  TEC;          // Indica A quantidade de  erros de transmissão CAN
        char  REC;          // Indica A quantidade de  erros de recepção CAN
        unsigned int        TimerStatusReadCounter;     // Contador de tempo CAN

        union{
                char Byte;
                struct{
                        char RX0IF      : 1;    //Receive Buffer 0 Full Interrupt Flag
                        char RX1IF      : 1;    //Receive Buffer 1 Full Interrupt Flag
                        char TX0IF      : 1;    //Transmit Buffer 0 Empty Interrupt Flag
                        char TX1IF      : 1;    //Transmit Buffer 1 Empty Interrupt Flag
                        char TX2IF      : 1;    //Transmit Buffer 2 Empty Interrupt Flag
                        char ERRIF      : 1;    //Error Interrupt Flag (multiple sources in EFLG
register)
                        char WAKIF      : 1;    //Wakeup Interrupt Flag
                        char MERRF      : 1;    //Message Error Interrupt Flag
                }Bit;
```

```
        }IntReg;
        union {
                char Byte;
                struct{
                        char  EWARN :1;         //Error Warning Flag
                        char  RXWAR :1;         //Receive Error Warning Flag
                        char  TXWAR :1;         //Transmit Error Warning Flag
                        char  RXEP :1;          //Receive Error-Passive Flag
                        char  TXEP :1;          //Transmit Error-Passive Flag
                        char  TXBO :1;          //Bus-Off Error Flag
                        char  RX0OVR :1; //Receive Buffer 0 Overflow Flag
                        char  RX1OVR :1; //Receive Buffer 1 Overflow Flag
                }Bit;
        }Erro;
        union{
                char Byte;
                struct{
                  char      Iniciando :1;
                  char      Ativo     :1;
                }Bit;
        }Status;
        union{
                char Byte;
                struct{
                  char      Reset :1;
                  char      Ativo     :1;
                }Bit;
        }Comando;
}MCP2515;

// Variáveis
extern                          INT8U err;

OS_STK                                  MCP2515DriverStk[STKMCP2515Driver];
//OS_EVENT                              *SemPrintSPI;
OS_EVENT                        *SemIntMCP2515;
OS_EVENT                        *SemTXCAN;
OS_EVENT                        *SemCANControllerWakeUp;
EstruturaBufferTXCAN TXBufferCAN[CAN_TX_BUFFER_LEN];
EstruturaBufferRXCAN RXBufferCAN[CAN_RX_BUFFER_LEN];
unsigned int RXCANBufferPointer;
unsigned int RXCANBufferPointerRead;
unsigned int TXCANBufferPointer;
unsigned int TXCANBufferPointerWritten;

#include    "MCP2515Driver.c"
```

ANEXO 2: ARQUIVO MCP2515DRIVER.C DO PROGRAMA DO DRIVER MCP2515 EXECUTADO NA PLACA DE PROCESSAMENTO E NA PLACA DE CONTROLE DOS PERIFÉRICOS

```
void ReceivedCanMsgEvent(OS_EVENT *Sem){
	SemCANControllerWakeUp = Sem;
}

// Interrupção do MCP2515
void InterrupcaoMCP2515(void){
	OSSemPost(SemIntMCP2515);
}

void StatusReadTimerMCP2515(void){
	if (MCP2515.TimerStatusReadCounter){
		MCP2515.TimerStatusReadCounter --;
	}
	else{
		OSSemPost(SemIntMCP2515);
	}
}

////////////////////////////////////////////////////////////////////////////////
// Função de impressão na SPI

void PrintSPI(char *EnderecoPacoteTXSPI,char *EnderecoPacoteRXSPI,char TamanhoPacote){

	char AuxTX;

	//OSSemPend(SemPrintSPI, 0, &err);
	_DINT();
	CSLOW;
	SPI_TX_BUF = *(EnderecoPacoteTXSPI);

	while ((--TamanhoPacote)){
		AuxTX = *(++EnderecoPacoteTXSPI);
		while (SPISTAT & SPIBUSY);
		SPI_TX_BUF = AuxTX;
		*(EnderecoPacoteRXSPI++) = SPI_RX_BUF;
	}

	while (SPISTAT & SPIBUSY);
	*(EnderecoPacoteRXSPI) = SPI_RX_BUF;

	CSHIGH;
	_EINT();
	//OSSemPost(SemPrintSPI);
}
////////////////////////////////////////////////////////////////////////////////
char TXCAN(char IdUp,char IdDown,char Len,char RTS,char *Data){
	char i;
	OSSemPend(SemTXCAN, 0, &err);

	TXBufferCAN[TXCANBufferPointer].Id.Byte.Up = IdUp;
	TXBufferCAN[TXCANBufferPointer].Id.Byte.Down = IdDown;
	TXBufferCAN[TXCANBufferPointer].Inf.RTS = RTS;
```

```
		TXBufferCAN[TXCANBufferPointer].Inf.Len = Len;

		if (RTS == 0){
			for (i=0;i < Len;i++)
				TXBufferCAN[TXCANBufferPointer].Data[i] = *(Data + i);
		}

		if (TX0_RTS_TEST){
			TXBufferCAN[TXCANBufferPointer].Comando = LOAD_TX_BUFFER + BUFFER_0_ALL;

	PrintSPI(&TXBufferCAN[TXCANBufferPointer].Comando,&TXBufferCAN[TXCANBufferPointer].Comando,6 +
Len);
			TX0_RTS_DOWN;
		}
		else{
			if (TX1_RTS_TEST){
				TXBufferCAN[TXCANBufferPointer].Comando = LOAD_TX_BUFFER +
BUFFER_1_ALL;

	PrintSPI(&TXBufferCAN[TXCANBufferPointer].Comando,&TXBufferCAN[TXCANBufferPointer].Comando,6 +
Len);
				TX1_RTS_DOWN;
			}
			else{
				if (TX2_RTS_TEST){
					TXBufferCAN[TXCANBufferPointer].Comando = LOAD_TX_BUFFER +
BUFFER_2_ALL;

	PrintSPI(&TXBufferCAN[TXCANBufferPointer].Comando,&TXBufferCAN[TXCANBufferPointer].Comando,6 +
Len);
					TX2_RTS_DOWN;
				}
				else{
					TXCANBufferPointer++;
					TXCANBufferPointer &= CAN_TX_BUFFER_MASK;
				}
			}
		}
		OSSemPost(SemTXCAN);
		return(0);
	}

	////////////////////////////////////////////////////////////////////////////////

	void MCP2515Driver(void *pdata){

		enum MAQUINA_DE_ESTADO{
			SCHEDULER = 0,
			ISR_STATUS_READ,
			ISR_TX0 = 6,
			ISR_TX1 = 8,
			ISR_TX2 = 10,
			ISR_RX0 = 12,
			ISR_RX1 = 14,
			RESET_MCP2515,
			START_MCP2515,
			SET_MASKS,
			READ_STATUS_MCP2515,
		}Estado;



		char BuffersPI[17];
```

```
#if SPI_MODULO == 2
        UCB0CTL1 = UCSWRST;
        UCB0CTL0 = UCCKPL + UCMSB + UCMST + UCSYNC;
        UCB0CTL1 |= UCSSEL_2;
        UCB0BR0 = 0x01;
        UCB0BR1 = 0x00;
        P3DIR |= 0x0B; //xxxx1011
        P3SEL |= 0x0E; //xxxx1110
        P2DIR &= (~0x01); //xxxxxxx0
        P2SEL &= (~0x01); //xxxxxxx0
        P2IES |= 0x01; //xxxxxxx1
        P2IE |= 0x01; //xxxxxxx1
        UCB0CTL1 &= (~UCSWRST);
        ////IE2 |= UCB0TXIE;
        //IE2 |= UCB0RXIE;
        CSHIGH;
#endif
#if SPI_MODULO == 3
        UCB1CTL1 = UCSWRST;
        UCB1CTL0 = UCCKPL + UCMSB + UCMST + UCSYNC;
        UCB1CTL1 |= UCSSEL_2;
        UCB1BR0 = 0x01;
        UCB1BR1 = 0x00;
        P5DIR |= 0x0B; //xxxx1011
        P5SEL |= 0x0E; //xxxx1110
        P2DIR &= (~0x01); //xxxxxxx0
        P2SEL &= (~0x01); //xxxxxxx0
        P2IES |= 0x01; //xxxxxxx1
        P2IE |= 0x01; //xxxxxxx1
        UCB1CTL1 &= (~UCSWRST);
        //UC1IE |= UCB1TXIE;
        //UC1IE |= UCB1RXIE;
        CSHIGH;
#endif

INT_PORT_SEL;
INT_PORT_DIR;
RXB0_PORT_SEL;
RXB0_PORT_DIR;
RXB1_PORT_SEL;
RXB1_PORT_DIR;
TXB0_PORT_SEL;
TXB0_PORT_DIR;
TXB1_PORT_SEL;
TXB1_PORT_DIR;
TXB2_PORT_SEL;
TXB2_PORT_DIR;
LED_CAN_FAULT_PORT_SEL;
LED_CAN_FAULT_PORT_DIR;
LED_CAN_ACTIVE_PORT_SEL;
LED_CAN_ACTIVE_PORT_DIR;

TX0_RTS_UP;
TX1_RTS_UP;
TX2_RTS_UP;

LED_CAN_FAULT_OFF;
LED_CAN_ACTIVE_OFF;

TXCAN*Buffer*PointerWritten = 0;
TXCAN*Buffer*Pointer = 0;
RXCAN*Buffer*Pointer = 0;
RXCAN*Buffer*PointerRead = 0;

Estado = RESET_MCP2515;
```

```
SemIntMCP2515 =  OSSemCreate(0);

MCP2515.TimerStatusReadCounter = MCP2515_STATUS_READ_TIMER;
MCP2515.Comando.Bit.Ativo = 1;



while(1){
        switch(Estado){
        case SCHEDULER:
                if (TEST_INT_PIN){
                        if (MCP2515.TimerStatusReadCounter){
                                LED_CAN_ACTIVE_OFF;
                                OSSemPend(SemIntMCP2515, 0, &err);
                        }
                        else{
                                MCP2515.TimerStatusReadCounter                  =
MCP2515_STATUS_READ_TIMER;
                                Estado = ISR_STATUS_READ;;
                        }
                }
                else{
                        LED_CAN_ACTIVE_ON;
                        if (RXB0_PIN_TEST){
                                if (RXB1_PIN_TEST){
                                        BuffersPI[0] = READ;
                                        BuffersPI[1] = CANINTF_ADDRESS;
                                        PrintSPI(BuffersPI,BuffersPI,3);
                                        MCP2515.IntReg.Byte =  BuffersPI[2];

                                        if (MCP2515.IntReg.Bit.TX0IF){
                                                Estado = ISR_TX0;
                                        }
                                        else{
                                                if (MCP2515.IntReg.Bit.TX1IF){
                                                        Estado = ISR_TX1;
                                                }
                                                else{
                                                        if (MCP2515.IntReg.Bit.TX2IF){
                                                                Estado = ISR_TX2;
                                                        }
                                                        else{
                                                                Estado                  =
ISR_STATUS_READ;

                                                        }
                                                }
                                        }
                                }
                                else{
                                        Estado = ISR_RX1;
                                }
                        }
                        else{
                                Estado = ISR_RX0;
                        }
                }
                break;
                // Rotina de Tratamento de Erros
        case ISR_STATUS_READ:
                BuffersPI[0] = READ;
                BuffersPI[1] = EFLG_ADDRESS;
                PrintSPI(BuffersPI,BuffersPI,3);
                MCP2515.Erro.Byte = BuffersPI[2];
```

```
                                        BuffersPI[0] = READ;
                                        BuffersPI[1] = TEC_ADDRESS;
                                        PrintSPI(BuffersPI,BuffersPI,3);
                                        MCP2515.TEC = BuffersPI[2];

                                        BuffersPI[0] = READ;
                                        BuffersPI[1] = REC_ADDRESS;
                                        PrintSPI(BuffersPI,BuffersPI,3);
                                        MCP2515.REC = BuffersPI[2];

                                        if (MCP2515.Erro.Byte){
                                                LED_CAN_FAULT_ON;
                                                BuffersPI[0] = BIT_MODIFY;
                                                BuffersPI[1] = EFLG_ADDRESS;
                                                BuffersPI[2] = 0xFF;
                                                BuffersPI[3] = 0x00;
                                                PrintSPI(BuffersPI,BuffersPI,4);

                                                BuffersPI[0] = BIT_MODIFY;
                                                BuffersPI[1] = CANINTF_ADDRESS;
                                                BuffersPI[2] = 0xE0;
                                                BuffersPI[3] = 0x00;
                                                PrintSPI(BuffersPI,BuffersPI,4);
                                        }
                                        else{
                                                LED_CAN_FAULT_OFF;
                                        }

                                        if ((MCP2515.Comando.Bit.Ativo == 0) && (MCP2515.Status.Bit.Ativo == 1) ){
                                                *BuffersPI = RESET;
                                                PrintSPI(BuffersPI,BuffersPI,1);
                                                MCP2515.Status.Bit.Ativo = 0;
                                                OSTimeDly(10);
                                        }
                                        else{
                                                if (((MCP2515.Comando.Bit.Ativo == 1) && (MCP2515.Status.Bit.Ativo
== 0))  || (MCP2515.Comando.Bit.Reset == 1)){
                                                        Estado = RESET_MCP2515;
                                                        MCP2515.Comando.Bit.Reset = 0;
                                                }
                                                else{
                                                        Estado = SCHEDULER;
                                                }
                                        }
                                        break;
                                case ISR_TX0:
                                        TX0_RTS_UP;
                                        BuffersPI[0] = BIT_MODIFY;
                                        BuffersPI[1] = CANINTF_ADDRESS;
                                        BuffersPI[2] = 0x04;
                                        BuffersPI[3] = 0;
                                        PrintSPI(BuffersPI,BuffersPI,4);

                                        if (TXCANBufferPointerWritten != TXCANBufferPointer){
                                                (TXBufferCAN + TXCANBufferPointerWritten)->Comando =
LOAD_TX_BUFFER + BUFFER_0_ALL;
                                                PrintSPI(&(TXBufferCAN + TXCANBufferPointerWritten)-
>Comando,&(TXBufferCAN + TXCANBufferPointerWritten)->Comando,14);
                                                TX0_RTS_DOWN;
                                                TXCANBufferPointerWritten ++;
                                                TXCANBufferPointerWritten &= CAN_TX_BUFFER_MASK;
                                        }
                                        Estado = SCHEDULER;
                                        break;
                                case ISR_TX1:
```

```
                                        TX1_RTS_UP;
                                        BuffersPI[0] = BIT_MODIFY;
                                        BuffersPI[1] = CANINTF_ADDRESS;
                                        BuffersPI[2] = 0x08;
                                        BuffersPI[3] = 0;
                                        PrintSPI(BuffersPI,BuffersPI,4);
                                        if (TXCANBufferPointerWritten != TXCANBufferPointer){
                                                (TXBufferCAN + TXCANBufferPointerWritten)->Comando = LOAD_TX_BUFFER + BUFFER_1_ALL;
                                                PrintSPI(&(TXBufferCAN + TXCANBufferPointerWritten)->Comando,&(TXBufferCAN + TXCANBufferPointerWritten)->Comando,14);
                                                TX1_RTS_DOWN;
                                                TXCANBufferPointerWritten ++;
                                                TXCANBufferPointerWritten &= CAN_TX_BUFFER_MASK;
                                        }
                                        Estado = SCHEDULER;
                                        break;
                                case ISR_TX2:
                                        TX2_RTS_UP;
                                        BuffersPI[0] = BIT_MODIFY;
                                        BuffersPI[1] = CANINTF_ADDRESS;
                                        BuffersPI[2] = 0x10;
                                        BuffersPI[3] = 0;
                                        PrintSPI(BuffersPI,BuffersPI,4);

                                        if (TXCANBufferPointerWritten != TXCANBufferPointer){
                                                (TXBufferCAN + TXCANBufferPointerWritten)->Comando = LOAD_TX_BUFFER + BUFFER_2_ALL;
                                                PrintSPI(&(TXBufferCAN + TXCANBufferPointerWritten)->Comando,&(TXBufferCAN + TXCANBufferPointerWritten)->Comando,14);
                                                TX2_RTS_DOWN;
                                                TXCANBufferPointerWritten ++;
                                                TXCANBufferPointerWritten &= CAN_TX_BUFFER_MASK;
                                        }
                                        Estado = SCHEDULER;
                                        break;
                                case ISR_RX0:
                                        BuffersPI[0] = READ_RX_BUFFER;
                                        PrintSPI(BuffersPI, &(RXBufferCAN[RXCANBufferPointer].Nada0), 14);
                                        RXCANBufferPointer++;
                                        RXCANBufferPointer &= CAN_RX_BUFFER_MASK;
                                        Estado = SCHEDULER;
                                        OSSemPost(SemCANControllerWakeUp);
                                        break;
                                case ISR_RX1:
                                        BuffersPI[0] = READ_RX_BUFFER + 4;
                                        PrintSPI(BuffersPI, &(RXBufferCAN[RXCANBufferPointer].Nada0), 14);
                                        RXCANBufferPointer++;
                                        RXCANBufferPointer &= CAN_RX_BUFFER_MASK;
                                        Estado = SCHEDULER;
                                        OSSemPost(SemCANControllerWakeUp);
                                        break;

                                case RESET_MCP2515:
                                        MCP2515.Status.Bit.Iniciando = 1;
                                        MCP2515.Status.Bit.Ativo = 0;

                                        *BuffersPI = RESET;
                                        PrintSPI(BuffersPI,BuffersPI,1);
                                        OSTimeDly(10);
                                        Estado = START_MCP2515;
                                        break;
                                case START_MCP2515:
                                        /* Configuração inicial do MCP2515
                                                Taxa CAN = 1 Mbps
```

Pinos de RXB0 e RXB1 configurados para interrupção
Pinos TXB0, TXB1 e TXB2 configurados para gatilho de transmissão
Interrupções de erro, recepção, transmissão e multiplas interrupções habilitadas

```
                */
                BuffersPI[0] = WRITE;
                BuffersPI[1] = CANCTRL_ADDRESS;
                BuffersPI[2] = CANCTRL_CONFIG_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                BuffersPI[0] = WRITE;
                BuffersPI[1] = CNF1_ADDRESS;
                BuffersPI[2] = CNF1_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                BuffersPI[0] = WRITE;
                BuffersPI[1] = CNF2_ADDRESS;
                BuffersPI[2] = CNF2_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                BuffersPI[0] = WRITE;
                BuffersPI[1] = CNF3_ADDRESS;
                BuffersPI[2] = CNF3_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                BuffersPI[0] = WRITE;
                BuffersPI[1] = BFPCTRL_ADDRESS;
                BuffersPI[2] = BFPCTRL_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                BuffersPI[0] = WRITE;
                BuffersPI[1] = CANINTE_ADDRESS;
                BuffersPI[2] = CANINTE_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                BuffersPI[0] = WRITE;
                BuffersPI[1] = CANINTF_ADDRESS;
                BuffersPI[2] = CANINTF_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                BuffersPI[0] = WRITE;
                BuffersPI[1] = TXRTSCTRL_ADDRESS;
                BuffersPI[2] = TXRTSCTRL_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                BuffersPI[0] = WRITE;
                BuffersPI[1] = EFLG_ADDRESS;
                BuffersPI[2] = EFLG_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                BuffersPI[0] = WRITE;
                BuffersPI[1] = CANCTRL_ADDRESS;
                BuffersPI[2] = CANCTRL_NORMAL_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                Estado = SET_MASKS;
                break;
        case SET_MASKS:
                BuffersPI[0] = WRITE;
                BuffersPI[1] = RXB0CTRL_ADDRESS;
                BuffersPI[2] = RXB0CTRL_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                BuffersPI[0] = WRITE;
                BuffersPI[1] = RXB1CTRL_ADDRESS;
                BuffersPI[2] = RXB1CTRL_VALUE;
                PrintSPI(BuffersPI,BuffersPI,3);
                Estado = SCHEDULER;
                OSTimeDly(50);
                MCP2515.Status.Bit.Iniciando = 0;
                MCP2515.Status.Bit.Ativo = 1;
                OSSemPost(SemCANControllerWakeUp);
                break;
        default:
                Estado = SCHEDULER;
                break;
```

```
                }
            }
        }

        void StartMCP2515Driver(void){
            //SemPrintSPI = OSSemCreate(1);
            SemTXCAN = OSSemCreate(1);
            SemCANControllerWakeUp =  OSSemCreate(0);
            SemIntMCP2515 = OSSemCreate(0);
            OSTaskCreate(MCP2515Driver, (void *)0, &MCP2515DriverStk[STKMCP2515Driver - 1],
MCP2515DriverPriority);
        }
```

ANEXO 3:ARQUIVO MCP2515DRIVER.H DO PROGRAMA DO DRIVER MCP2515 EXECUTADO NA PLACA DE CONTROLE DOS PERIFÉRICOS

```
#include    <msp430x26x.h>
#include    "ucos_ii.h"

#define SPI_MODULO                  3           // 0 => UCA0, 1 => UCA1, 2 => UCB0, 3 => UCB1
#define TAXA_CAN                    2           // 0 => 1Mbps, 1 => 500 kbps, 2 => 250 kbps, 3 = > 125 kbps
#define TEMPO_LED_ERRO_ON           300
#define MCP2515DriverPriority       8          //Prioridade da tarefa
#define STKMCP2515Driver            300         // Número de Bytes da Pilha do Driver
#define CAN_RX_BUFFER_LEN           16          // 2, 4, 8, 16, 32 ...
#define CAN_TX_BUFFER_LEN           16          // 2, 4, 8, 16, 32 ...
#define MCP2515_STATUS_READ_TIMER       200


// INT Pin config
#define INT_BIT                             0X01
#define INT_PORT                            P2IN
#define INT_PORT_SEL                        P2SEL &= (~INT_BIT)
#define INT_PORT_DIR                        P2DIR &= (~INT_BIT)
// RXB0 Pin config
#define RXB0_BIT                            0X40
#define RXB0_PORT                           P2IN
#define RXB0_PORT_SEL                       P2SEL &= (~RXB0_BIT)
#define RXB0_PORT_DIR                       P2DIR &= (~RXB0_BIT)
// RXB1 Pin config
#define RXB1_BIT                            0X80
#define RXB1_PORT                           P2IN
#define RXB1_PORT_SEL                       P2SEL &= (~RXB1_BIT)
#define RXB1_PORT_DIR                       P2DIR &= (~RXB1_BIT)
// TXB0 Pin config
#define TXB0_BIT                            0X08
#define TXB0_PORT                           P2OUT
#define TXB0_PORT_SEL                       P2SEL &= (~TXB0_BIT)
#define TXB0_PORT_DIR                       P2DIR |= TXB0_BIT
// TXB1 Pin config
#define TXB1_BIT                            0X10
#define TXB1_PORT                           P2OUT
#define TXB1_PORT_SEL                       P2SEL &= (~TXB1_BIT)
#define TXB1_PORT_DIR                       P2DIR |= TXB1_BIT
// TXB2 Pin config
#define TXB2_BIT                            0X20
#define TXB2_PORT                           P2OUT
#define TXB2_PORT_SEL                       P2SEL &= (~TXB2_BIT)
#define TXB2_PORT_DIR                       P2DIR |= TXB2_BIT
// LED FAULT Pin config
#define LED_CAN_FAULT_BIT                   0X02
#define LED_CAN_FAULT_PORT                  P2OUT
#define LED_CAN_FAULT_PORT_SEL              P2SEL &= (~LED_CAN_FAULT_BIT)
#define LED_CAN_FAULT_PORT_DIR              P2DIR |= LED_CAN_FAULT_BIT
// LED CAN ACTIVE Pin config
#define LED_CAN_ACTIVE_BIT                  0X01
#define LED_CAN_ACTIVE_PORT                 P1OUT
#define LED_CAN_ACTIVE_PORT_SEL             P1SEL &= (~LED_CAN_ACTIVE_BIT)
#define LED_CAN_ACTIVE_PORT_DIR             P1DIR |= LED_CAN_ACTIVE_BIT

#if SPI_MODULO == 2
```

```
        #define FLAG_SPI_INT_TX         UCB0TXIFG
        #define FLAG_SPI_INT_RX         UCB0RXIFG
        #define SPI_RX_BUF              UCB0RXBUF
        #define SPI_TX_BUF              UCB0TXBUF
        #define SPISTAT                 UCB0STAT
        #define SPIBUSY                 UCBUSY
        #define REG_SPI_INT_FLAG        IFG2
        #define CSHIGH                  P3OUT |= 0x01
        #define CSLOW                   P3OUT &= (~0x01)
#endif

#if SPI_MODULO == 3
        #define FLAG_SPI_INT_TX         UCB1TXIFG
        #define FLAG_SPI_INT_RX         UCB1RXIFG
        #define SPI_RX_BUF              UCB1RXBUF
        #define SPI_TX_BUF              UCB1TXBUF
        #define SPISTAT                 UCB1STAT
        #define SPIBUSY                 UCBUSY
        #define REG_SPI_INT_FLAG        UC1IFG
        #define CSHIGH                  P5OUT |= 0x01
        #define CSLOW                   P5OUT &= (~0x01)
#endif

// register CAN VALUE

#if TAXA_CAN == 0
  #define CNF1_VALUE            0x00 // 0b00 000000 SJW = 1, BRP = 1
  #define CNF2_VALUE            0xD0 // 0b11 010 000 PS1 = 3, Tpr = 1
  #define CNF3_VALUE            0x02 // 0b00 XXX 010 PS2 = 3
#endif

#if TAXA_CAN == 1
  #define CNF1_VALUE            0xC0 // 0b11 000000 SJW = 2, BRP = 1
  #define CNF2_VALUE            0xEA // 0b11 101 010 PS1 = 6, Tpr = 3
  #define CNF3_VALUE            0x05 // 0b00 XXX 101  PS2 = 6
#endif

#if TAXA_CAN == 2
  #define CNF1_VALUE            0xC1 // 0b11 000001 SJW = 2, BRP = 2
  #define CNF2_VALUE            0xEA // 0b11 101 010 PS1 = 6, Tpr = 3
  #define CNF3_VALUE            0x05 // 0b00 XXX 101  PS2 = 6
#endif

#if TAXA_CAN == 3
  #define CNF1_VALUE            0xC3 // 0b11 000011 SJW = 2, BRP = 4
  #define CNF2_VALUE            0xEA // 0b11 101 010 PS1 = 6, Tpr = 3
  #define CNF3_VALUE            0x05 // 0b00 XXX 101  PS2 = 6
#endif


#define CANCTRL_CONFIG_VALUE    0x90
#define CANCTRL_NORMAL_VALUE    0x04
#define BFPCTRL_VALUE           0x0F
#define CANINTE_VALUE           0x1F
#define CANINTF_VALUE           0x00
#define TXRTSCTRL_VALUE         0x07
#define RXB0CTRL_VALUE          0x64 // 0b01100100
#define RXB1CTRL_VALUE          0x60
#define EFLG_VALUE              0x00



// Macros de Comados
////////////////////////////////////////////////////////////////////////////////
// LED FAULT
```

```
#define LED_CAN_FAULT_ON            LED_CAN_FAULT_PORT |= LED_CAN_FAULT_BIT
#define LED_CAN_FAULT_OFF           LED_CAN_FAULT_PORT &= (~LED_CAN_FAULT_BIT)
#define LED_CAN_FAULT_TEST          LED_CAN_FAULT_PORT & LED_CAN_FAULT_BIT


//LED Active
#define LED_CAN_ACTIVE_ON           LED_CAN_ACTIVE_PORT |= LED_CAN_ACTIVE_BIT
#define LED_CAN_ACTIVE_OFF          LED_CAN_ACTIVE_PORT &= (~LED_CAN_ACTIVE_BIT)
#define LED_CAN_ACTIVE_TEST         LED_CAN_ACTIVE_PORT & LED_CAN_ACTIVE_BIT

// CAN Comandos
#define TEST_INT_PIN                INT_PORT & INT_BIT
#define RXB0_PIN_TEST               RXB0_PORT & RXB0_BIT
#define RXB1_PIN_TEST               RXB1_PORT & RXB1_BIT
#define TX0_RTS_UP                  TXB0_PORT |= TXB0_BIT
#define TX0_RTS_DOWN                TXB0_PORT &= (~TXB0_BIT)
#define TX1_RTS_UP                  TXB1_PORT |= TXB1_BIT
#define TX1_RTS_DOWN                TXB1_PORT &= (~TXB1_BIT)
#define TX2_RTS_UP                  TXB2_PORT |= TXB2_BIT
#define TX2_RTS_DOWN                TXB2_PORT &= (~TXB2_BIT)
#define TX0_RTS_TEST                TXB0_PORT & TXB0_BIT
#define TX1_RTS_TEST                TXB1_PORT & TXB1_BIT
#define TX2_RTS_TEST                TXB2_PORT & TXB2_BIT


///
// Defines//

#define CAN_RX_BUFFER_MASKCAN_RX_BUFFER_LEN - 1
#define CAN_TX_BUFFER_MASK          CAN_TX_BUFFER_LEN - 1

#define RESET                       0xC0
#define WRITE                       0x02
#define READ                        0x03
#define BIT_MODIFY                  0x05
#define READ_RX_BUFFER              0x90
#define READ_STATUS                 0xA0

#define BUFFER_0_ALL        0
#define BUFFER_1_ALL        2
#define BUFFER_2_ALL        4
#define LOAD_TX_BUFFER              0x40
// register CAN ADDRESS
#define CANCTRL_ADDRESS             0x0F
#define CNF1_ADDRESS                0x2A
#define CNF2_ADDRESS                0x29
#define CNF3_ADDRESS                0x28
#define BFPCTRL_ADDRESS             0x0C
#define EFLG_ADDRESS                0x2D
#define CANINTE_ADDRESS             0x2B
#define CANINTF_ADDRESS             0x2C
#define CANSTAT_ADDRESS             0x0E
#define TXRTSCTRL_ADDRESS           0x0D
#define RXB0CTRL_ADDRESS            0x60
#define RXB1CTRL_ADDRESS            0x70
#define TEC_ADDRESS                 0x1C
#define REC_ADDRESS                 0x1D


#define TRUE                        1
#define FALSE                       0
#define ERRO                        1
#define NOERRO                      0
```

```
////////////////////////////////////////////////////////////////////////////
// Tipo de buffer para de recepção CAN
typedef struct Estrutura BufferRXCAN {
        char  Nada0;
        union{
                //unsigned int All;
                struct{
                        char Up;
                        char Down;
                }Byte;
                struct{
                        char  IdUp;             //Identificação da msg
                        char  BitDownId: 3;
                        char  IDE: 1;
                        char  RTS: 1;
                }Bit;
        }Id;
        char  ExtendUp;
        char  ExtendDown;
        char  Len;
        char  Data[8];
}Estrutura BufferRXCAN;

typedef struct Estrutura BufferTXCAN {
        char Comando;
        union{
                struct{
                        char Up;
                        char Down;
                }Byte;
                struct{
                        char  IdUp;             //Identificação da msg
                        char  BitDownId: 3;
                        char  IDE: 1;
                }Bit;
        }Id;
        char  ExtendUp;
        char  ExtendDown;
        struct{
                char  Len: 4;
                char  UnUsed: 2;
                char  RTS: 1;
        }Inf;
        char  Data[8];
}Estrutura BufferTXCAN;


struct{
        char  TEC;          // Indica A quantidade de  erros de transmissão CAN
        char  REC;          // Indica A quantidade de  erros de recepção CAN
        unsigned int        TimerStatusReadCounter;     // Contador de tempo CAN

        union{
                char Byte;
                struct{
                        char RX0IF      : 1;    //Receive Buffer 0 Full Interrupt Flag
                        char RX1IF      : 1;    //Receive Buffer 1 Full Interrupt Flag
                        char TX0IF      : 1;    //Transmit Buffer 0 Empty Interrupt Flag
                        char TX1IF      : 1;    //Transmit Buffer 1 Empty Interrupt Flag
                        char TX2IF      : 1;    //Transmit Buffer 2 Empty Interrupt Flag
                        char ERRIF      : 1;    //Error Interrupt Flag (multiple sources in EFLG
register)
                        char WAKIF      : 1;    //Wakeup Interrupt Flag
                        char MERRF      : 1;    //Message Error Interrupt Flag
                }Bit;
```

```
        }IntReg;
        union {
                char Byte;
                struct{
                        char  EWARN :1;         //Error Warning Flag
                        char  RXWAR :1;         //Receive Error Warning Flag
                        char  TXWAR :1;         //Transmit Error Warning Flag
                        char  RXEP :1;          //Receive Error-Passive Flag
                        char  TXEP :1;          //Transmit Error-Passive Flag
                        char  TXBO :1;          //Bus-Off Error Flag
                        char  RX0OVR :1; //Receive Buffer 0 Overflow Flag
                        char  RX1OVR :1; //Receive Buffer 1 Overflow Flag
                }Bit;
        }Erro;
        union{
                char Byte;
                struct{
                  char      Iniciando :1;
                  char      Ativo     :1;
                }Bit;
        }Status;
        union{
                char Byte;
                struct{
                  char      Reset :1;
                  char      Ativo     :1;
                }Bit;
        }Comando;
}MCP2515;

// Variáveis
extern                          INT8U err;

OS_STK                                  MCP2515DriverStk[STKMCP2515Driver];
//OS_EVENT                              *SemPrintSPI;
OS_EVENT                        *SemIntMCP2515;
OS_EVENT                        *SemTXCAN;
OS_EVENT                        *SemCANControllerWakeUp;
EstruturaBufferTXCAN TXBufferCAN[CAN_TX_BUFFER_LEN];
EstruturaBufferRXCAN RXBufferCAN[CAN_RX_BUFFER_LEN];
unsigned int RXCANBufferPointer;
unsigned int RXCANBufferPointerRead;
unsigned int TXCANBufferPointer;
unsigned int TXCANBufferPointerWritten;

#include    "MCP2515Driver.c"
```

ANEXO 4:ARQUIVO ISR.C. ROTINA DE TRATAMENTO DE INRRUPÇÕES DA PLACA DE PROCESSAMENTO

```
extern unsigned int CAN_Timer;
extern unsigned int TempLED1;
extern unsigned int TensaoAlimentacao;
void ISR_TX_SD (void);
void ISR_RX_SD (void);


// Interrupt of Time B
#pragma vector = TIMERB0_VECTOR
__save_reg20 __interrupt void trata_timer_B0(void){

 TBCCR0 += 8000;

 Status_ReadTimer_MCP2515();
 Int_Timer_CANMSGControlTask();
}


////////////////////////////////////////////////////////////////////////////////////////////

// Recepção da UART e CAN
#pragma vector = USCIAB1RX_VECTOR
__save_reg20 __interrupt void recepcaoAB1(void)
{
  //SPI_ISR_RX();

  // UART Recepção
  if (UC1IFG & UCA1RXIFG){
    Buffer_RXUART[PonteiroRXUART] = UCA1RXBUF;
    PonteiroRXUART ++;
    PonteiroRXUART &= 0x7F;
    OSSemPost(SemRXUART);
  }
}
////////////////////////////////////////////////////////////////////////////////////////////
// UART Transmissão e CAN Transmissão
#pragma vector = USCIAB1TX_VECTOR
 __save_reg20 __interrupt void TransmissaoAB1(void)
{
  // SPI Transmiter
  //SPI_ISR_TX();
  // UART Transmiter
  if (UC1IFG & UCA1TXIFG){
    if (PonteiroTXUART != PonteiroSetPointTXUART){
      UCA1TXBUF = Buffer_TXUART[PonteiroTXUART];
      PonteiroTXUART ++;
      PonteiroTXUART &= 0x7F;
    }
    else{
      PonteiroTXUART = 0xFF;
      UC1IFG &= (~UCA1TXIFG);
    }
  }
}
// Interrupção de tratamento do barramento I2C
#pragma vector = USCIAB0TX_VECTOR
__save_reg20 __interrupt void TransmissaoAB0(void)
```

```
{

 if (IFG2 & UCB0TXIFG){
   if ((PonteiroI2C < PonteiroI2C_Buffer_)){
     UCB0TXBUF = _Buffer_I2C[PonteiroI2C];
     PonteiroI2C++;
   }
   else{
     IFG2 &= (~UCB0TXIFG);
     OSSemPost(SemI2C);
   }
 }
 if (IFG2 & UCB0RXIFG){
   _Buffer_I2C[PonteiroI2C] = UCB0RXBUF;
   PonteiroI2C++;
   if (PonteiroI2C == PonteiroI2C_Buffer_){
     UCB0CTL1 |= UCTXSTP;
   }
   if (PonteiroI2C > PonteiroI2C_Buffer_){
     OSSemPost(SemI2C);
   }

 }

 ISR_TX_SD();
}

#pragma vector = USCIAB0RX_VECTOR
__interrupt void recepcaoAB0(void)
{
        //if (IFG2 & UCB0RXIFG)
                //IFG2 &= (~UCB0RXIFG);
        ISR_RX_SD();
}



#pragma vector = PORT2_VECTOR
__save_reg20 __interrupt void MCP2515InterruptPin(void){
 P2IFG = 0x00;
 InterrupcaoMCP2515();
}

#pragma vector = ADC12_VECTOR
__save_reg20 __interrupt void ADC12(void){
 ADC12IFG = 0x00;
 TensaoAlimentacao = ADC12MEM0;
}
```

ANEXO 5:ARQUIVO ISR.C. ROTINA DE TRATAMENTO DE INRRUPÇÕES DA PLACA DE CONTROLE DOS PERIFÉRICOS

```
// Interrupt of Time B
#pragma vector = TIMERB1_VECTOR
__interrupt void trata_timer_B1(void){


        switch (TBIV){
                case 0x02 : // Encoder esquerdo canal 1
                        if (TBCCTL1 & 0x0002){
                                TBCCTL1 &= (~0x0002);
                        }
                        else{
                                TBCCTL1 &= (~0x0001);
                                EncEsq.Anterior = EncEsq.Atual;
                                EncEsq.Atual = TBCCR1;
                                EncEsq.OverFlowCap = EncEsq.OverFlow;
                                EncEsq.OverFlow = 0;
                                EncEsq.Sentido = 0;
                                if (TBCCTL2 & CCI)
                                        EncEsq.Sentido = 1;
                                EncEsq.UpDate = 1;
                        }
                        break;
                case 0x06 : // Encoder direito canal 1
                        if (TBCCTL3 & 0x0002){
                                TBCCTL3 &= (~0x0002);
                        }
                        else{
                                TBCCTL3 &= (~0x0001);
                                EncDir.Anterior = EncDir.Atual;
                                EncDir.Atual = TBCCR3;
                                EncDir.OverFlowCap = EncDir.OverFlow;
                                EncDir.OverFlow = 0;
                                EncDir.Sentido = 1;
                                if (TBCCTL4 & CCI)
                                        EncDir.Sentido = 0;
                                EncDir.UpDate = 1;
                        }
                        break;
                case 0x0A:

                        switch(IR.BitCounter){
                        case 0:
                                IR.BitCounter = 25;
                                IR_CLK_LOW;
                                TBCCR5 += 35712;
                                IR.ShiftRegister0 = 0;
                                IR.ShiftRegister1 = 0;
                                IR.ShiftRegister2 = 0;
                                IR.ShiftRegister3 = 0;
                                break;
                        case 1:
                                IR.BitCounter --;
                                IR_CLK_XOR;
                                TBCCR5 += 16000;
                                IR.ShiftRegister0 <<= 1;
                                IR.ShiftRegister1 <<= 1;
                                IR.ShiftRegister2 <<= 1;
```

```
                    IR.ShiftRegister3 <<= 1;
                    if(P7IN & 0x01)
                            IR.ShiftRegister0 += 1;
                    if(P7IN & 0x02)
                            IR.ShiftRegister1 += 1;
                    if(P7IN & 0x04)
                            IR.ShiftRegister2 += 1;
                    if(P7IN & 0x08)
                            IR.ShiftRegister3 += 1;

                    IR.Data0 = IR.ShiftRegister0;
                    IR.Data1 = IR.ShiftRegister1;
                    IR.Data2 = IR.ShiftRegister2;
                    IR.Data3 = IR.ShiftRegister3;
                    IR.NewData = 1;
                    break;
            case 2:
            case 3:
            case 4:
            case 5:
            case 6:
            case 7:
            case 8:
            case 9:
            case 10:
            case 11:
            case 12:
            case 13:
            case 14:
            case 15:
                    if (IR.BitCounter & 0x01){
                            IR.ShiftRegister0 <<= 1;
                            IR.ShiftRegister1 <<= 1;
                            IR.ShiftRegister2 <<= 1;
                            IR.ShiftRegister3 <<= 1;
                            if(P7IN & 0x01)
                                    IR.ShiftRegister0 += 1;
                            if(P7IN & 0x02)
                                    IR.ShiftRegister1 += 1;
                            if(P7IN & 0x04)
                                    IR.ShiftRegister2 += 1;
                            if(P7IN & 0x08)
                                    IR.ShiftRegister3 += 1;
                    }
            case 16:
            case 17:

                    IR.BitCounter --;
                    IR_CLK_XOR;
                    TBCCR5 += 800;

                    break;
            default:
                    IR.BitCounter --;
                    break;
            }
            break;
case 0x0e : // estouro do Timer TBR
            if(EncDir.OverFlow < 0X7FFF0000)
                    EncDir.OverFlow += 0x00010000;
            if(EncEsq.OverFlow < 0X7FFF0000)
                    EncEsq.OverFlow += 0x00010000;
            break;
default :// nada
            break;
```

```
        }
}



// Interrupt of Time B
#pragma vector = TIMERB0_VECTOR
__interrupt void trata_timer_B0(void){

 TBCCR0 += 8000;
 StatusReadTimerMCP2515();
 IntTimerCANMSGControlTask();
}


////////////////////////////////////////////////////////////////////////////////////////////////

// Recepção da UART
/*
#pragma vector = USCIAB0RX_VECTOR
__save_reg20 __interrupt void recepcaoAB1(void)
{
  // UART Recepção
  if (IFG2 & UCA0RXIFG){
    BufferRXUART[PonteiroRXUART] = UCA0RXBUF;
    PonteiroRXUART ++;
    PonteiroRXUART &= 0x1F;
    OSSemPost(SemRXUART);
   }
}

// Interrupção de tratamento do barramento I2C
#pragma vector = USCIAB0TX_VECTOR
__interrupt void TransmissaoAB0(void)
{
  // SPI Transmiter
  //SPI_ISR_TX();
  // UART Transmiter
  if (IFG2 & UCA0TXIFG){
    if (PonteiroTXUART != PonteiroSetPointTXUART){
      UCA0TXBUF = BufferTXUART[PonteiroTXUART];
      PonteiroTXUART ++;
      PonteiroTXUART &= 0x1F;
    }
    else{
      PonteiroTXUART = 0xFF;
      IFG2 &= (~UCA0TXIFG);
    }
  }

}
*/
#pragma vector = PORT2_VECTOR
__save_reg20 __interrupt void MCP2515InterruptPin(void){
  P2IFG = 0x00;
  InterrupcaoMCP2515();
}
```

ANEXO 6:ARQUIVO CAN_CONTROL_TASK.C. PROGRAMA DO CONTROLADOR DE VARIÁVEIS

```
void IntTimerCANMSGControlTask(void){
	if (ControlCanMsgController.Timer){
		ControlCanMsgController.Timer --;
	}
	else{
		OSSemPost(SemCANControllerWakeUp);
	}
}
////////////////////////////////////////////////////////////////////////////////////////////
void CANMSGSemPend(HANDLE_CAN_MSG HandleMSG){
	OSSemPend(BufferMsgControl[HandleMSG].Sem, 0, &err);
}

////////////////////////////////////////////////////////////////////////////////////////////
void CANMSGSemPost(HANDLE_CAN_MSG HandleMSG){
	OSSemPost(BufferMsgControl[HandleMSG].Sem);
}
////////////////////////////////////////////////////////////////////////////////////////////
char CANDataAvaliable(HANDLE_CAN_MSG HandleMSG){
	if (BufferMsgControl[HandleMSG].NoData)
		return 0;
	return 1;
}

////////////////////////////////////////////////////////////////////////////////////////////
HANDLE_CAN_MSG InsertOnBus (unsigned int Id,unsigned int TimerTrigger,void *DataPointer,char DataLength, char Envia){

	HANDLE_CAN_MSG Aux = 0;

	while((BufferMsgControl + Aux)->Id.All != 0xFFFF)
		Aux++;

	(BufferMsgControl + Aux)->Id.Byte.Up = (char)(Id >> 3);
	(BufferMsgControl + Aux)->Id.Byte.Down = (char)(Id << 5);
	(BufferMsgControl + Aux)->TimerTrigger = TimerTrigger;
	(BufferMsgControl + Aux)->Timer = TimerTrigger;
	(BufferMsgControl + Aux)->DataPointer = DataPointer;
	(BufferMsgControl + Aux)->DataLength = DataLength;
	(BufferMsgControl + Aux)->WaitForRec = 0;
	(BufferMsgControl + Aux)->NoData = 0;
	//if (Envia != 0)
		//TXCAN(Aux,0,1,0x00);
	return(Aux);
}
////////////////////////////////////////////////////////////////////////////////////////////
char IndexMsgOnBuffer(char ByteUp,char ByteDown){
	EstruturaID ID;
	HANDLE_CAN_MSG i = 0;

	ID.Byte.Up = ByteUp;
	ID.Byte.Down = (ByteDown & 0xE0);

	while ((BufferMsgControl + i)->Id.All != 0xFFFF){
		if ((BufferMsgControl + i)->Id.All == ID.All){
			return(i);
```

```
                }
                i++;
        }
        return(0xFF);
}



//char ScanCAN(char Comando,char NBytes);
/*
*********************************************************************************************************
*
*                                   CAN CONTROLLER TASK
*********************************************************************************************************
*/
void CANController(void *pdata){
        HANDLE_CAN_MSG HandleMSG;
        char i = 0;
        enum MAQUINA_DE_ESTADO{
                ESPERA_POR_DRIVER_ATIVO,
                IDLE,
                READ_MSG,
                TICK,
        }Estado;


        pdata  = pdata;
        Estado = ESPERA_POR_DRIVER_ATIVO;

        while(1){

                switch(Estado){
                case ESPERA_POR_DRIVER_ATIVO:
                        OSSemPend(SemCanMsgControllerAtive, 0, &err);
                        if (MCP2515.Status.Bit.Ativo)
                                Estado = IDLE;
                        break;
                case IDLE:
                        OSSemPend(SemCanMsgControllerAtive, 0, &err);
                        if (RXCANBufferPointer != RXCANBufferPointerRead){
                                Estado = READ_MSG;
                        }
                        else{
                                if (ControlCanMsgController.Timer == 0){
                                        ControlCanMsgController.Timer = BASE_TIMER_CAN_MSG_CONTROLLER;
                                        Estado = TICK;
                                }
                        }
                        break;
                case READ_MSG:
                        HandleMSG = IndexMsgOnBuffer(RXBufferCAN[RXCANBufferPointerRead].Id.Byte.Up,RXBufferCAN[RXCANBufferPointerRead].Id.Byte.Down);
                        if (HandleMSG != 0xFF){
                                if (RXBufferCAN[RXCANBufferPointerRead].Id.Bit.RTS){
                                        OSSemPend(BufferMsgControl[HandleMSG].Sem, 0, &err);

        TXCAN(BufferMsgControl[HandleMSG].Id.Byte.Up,BufferMsgControl[HandleMSG].Id.Byte.Down,BufferMsgControl[HandleMSG].DataLength,0,BufferMsgControl[HandleMSG].DataPointer);
                                        OSSemPost(BufferMsgControl[HandleMSG].Sem);
                                }
                                else{
                                OSSemPend(BufferMsgControl[HandleMSG].Sem, 0, &err);
```

```
                                                for (i = 0; i < Buffer*MsgControl[HandleMSG].DataLength;i++){
                                                        *(Buffer*MsgControl[HandleMSG].DataPointer + i) =
*(RXBuffer*CAN[RXCANBuffer*PointerRead].Data + i);
                                                }
                                        OSSemPost(Buffer*MsgControl[HandleMSG].Sem);
                                                Buffer*MsgControl[HandleMSG].WaitForRec = 0;
                                                Buffer*MsgControl[HandleMSG].NoData = 0;
                                        }
                                }
                                RXCANBuffer*PointerRead++;
                                RXCANBuffer*PointerRead &= CAN_RX_BUFFER_MASK;

                                if (RXCANBuffer*Pointer == RXCANBuffer*PointerRead){
                                        Estado = IDLE;
                                }

                                break;
                        case TICK:
                                HandleMSG = 0;
                                while(Buffer*MsgControl[HandleMSG].Id.All != 0xFFFF){
                                        if (Buffer*MsgControl[HandleMSG].Timer*Trigger != 0){
                                                if ((--Buffer*MsgControl[HandleMSG].Timer*) == 0){
                                                        Buffer*MsgControl[HandleMSG].Timer*                =
Buffer*MsgControl[HandleMSG].Timer*Trigger;
                                                        if (Buffer*MsgControl[HandleMSG].WaitForRec == 1) {
                                                                Buffer*MsgControl[HandleMSG].NoData = 1;

                                                        }

        TXCAN(Buffer*MsgControl[HandleMSG].Id.Byte*.Up,Buffer*MsgControl[HandleMSG].Id.Byte*.Down,Buffer*MsgCo
ntrol[HandleMSG].DataLength,1,Buffer*MsgControl[HandleMSG].DataPointer);
                                                        Buffer*MsgControl[HandleMSG].WaitForRec = 1;
                                                }
                                        }
                                        HandleMSG ++;
                                }
                                Estado = IDLE;
                                break;
                        default:
                                Estado = IDLE;
                                break;
                        }
                }
        }


        ////////////////////////////////////////////////////////////////////////////////////////////////
        void StartCANControlTask(void){
                char Aux;

                for (Aux = 0; Aux < TAMANHO_BUFFER_CONTROL_MSG;Aux++){
                        (Buffer*MsgControl + Aux)->Id.Byte*.Up = 0xFF;
                        (Buffer*MsgControl + Aux)->Id.Byte*.Down = 0xFF;
                }

                SemCanMsgControllerAtive = OSSemCreate(0);
                ReceivedCanMsgEvent(SemCanMsgControllerAtive);
                OSTaskCreate(CANController, (void *)0, &CANControllerStk[CAN_CONTROL_TASK_STACK_LEN
- 1], CAN_CONTROL_TASK_PRIORITY);
        }
```

ANEXO 7:ARQUIVO CAN_CONTROL_TASK.H.DO PROGRAMA DO CONTROLADOR DE VARIÁVEIS

```
#define TAMANHO_BUFFER_CONTROL_MSG                    16
#define BASE_TIMER_CAN_MSG_CONTROLLER   5
#define CAN_CONTROL_TASK_PRIORITY                15
#define CAN_CONTROL_TASK_STACK_LEN               300
typedef union EstruturaID{
                unsigned int All;
                struct{
                        char Up;
                        char Down;
                }Byte;
}EstruturaID;


typedef struct EstruturaMsg {
        EstruturaID     Id;
        OS_EVENT        *Sem;                   //Semafaro quando necessário
        unsigned int    TimerTrigger;           //periodo de scan da Msg
        unsigned int    Timer;                  //contador da msg
        char    *DataPointer;           //Ponteiro da variável que deve ser atualizada ou enviada
        char            DataLength;             //Tamanho do pacote de dados
        char            WaitForRec: 1;          //Bit que indicada se está esperando resposta de um RTS
        char            NoData: 1;              //tempo de recepção estourou e não chegou nova msg
}EstruturaMsg;

struct ControlCanMsgController{
        unsigned int Timer;
}ControlCanMsgController;

typedef unsigned char HANDLE_CAN_MSG;


//////////////////////////////////////////////////////////////////

OS_STK          CANControllerStk[CAN_CONTROL_TASK_STACK_LEN];
OS_EVENT                *SemCanMsgControllerAtive;
EstruturaMsg    BufferMsgControl [TAMANHO_BUFFER_CONTROL_MSG];


#include "CAN_Control_Task.c"
```

ANEXO 8:ARQUIVO HODÔMETRO.H. DA TAREFA DE HODOMETRIA DA CADEIRA DE RODAS

```
#define		CONSTANTE_VELOCIDADE			3978482//3840*8192
#define		DISTANCIA_ENTRE_RODAS			600 // mm
#define	PERIODO_ENCODER_CONTROL_TASK		2
#define		ENCODER_DIR_CH_A			0x02
#define		ENCODER_DIR_CH_B			0x04
#define		ENCODER_ESQ_CH_A			0x08
#define		ENCODER_ESQ_CH_B			0x10

#define		ENCODER_DIR_PORT_DIR			P4DIR
#define		ENCODER_DIR_PORT_SEL			P4SEL
#define		ENCODER_ESQ_PORT_DIR			P4DIR
#define		ENCODER_ESQ_PORT_SEL			P4SEL

#define CONFIG_ENCODER_DIR_PORT_DIR	ENCODER_DIR_PORT_DIR		&=
~(ENCODER_DIR_CH_A + ENCODER_DIR_CH_B)
#define CONFIG_ENCODER_DIR_PORT_SEL	ENCODER_DIR_PORT_SEL		|=
(ENCODER_DIR_CH_A + ENCODER_DIR_CH_B)
#define CONFIG_ENCODER_ESQ_PORT_DIR	ENCODER_ESQ_PORT_DIR		&=
~(ENCODER_ESQ_CH_A + ENCODER_ESQ_CH_B)
#define CONFIG_ENCODER_ESQ_PORT_SEL	ENCODER_ESQ_PORT_SEL		|=
(ENCODER_ESQ_CH_A + ENCODER_ESQ_CH_B)

#define		IR_CLK_HIGH				P4OUT |= 0x80
#define		IR_CLK_LOW				P4OUT &= (~0x80)
#define		IR_CLK_XOR				P4OUT ^= (0x80)
#define		IR_PIN_TEST				P4IN	& 0x80


HANDLE_CAN_MSG InsertOnBus (unsigned int,unsigned int,void  *,char, char);

typedef struct ENCODER{
	unsigned int Atual;
	unsigned int Anterior;
	unsigned long int OverFlow;
	unsigned long int OverFlowCap;
	char	Sentido: 1;
	char	UpDate: 1;
}ENCODER;

ENCODER EncDir;
ENCODER EncEsq;
long int Vre = 0;
long int Vrd = 0;

typedef struct POSICAO{
	long int X;
	long int Y;
}POSICAO;

typedef struct IR_Estrutura{
	int				BitCounter;
	unsigned int	Data0;
	unsigned int	Data1;
	unsigned int	Data2;
	unsigned int	Data3;
	unsigned int	Distancia0;
```

```
        unsigned int        Distancia1;
        unsigned int        Distancia2;
        unsigned int        Distancia3;
        unsigned int        ShiftRegister0;
        unsigned int        ShiftRegister1;
        unsigned int        ShiftRegister2;
        unsigned int        ShiftRegister3;
        char                        NewData : 1;
}IR_Estrutura;

IR_Estrutura    IR;
extern unsigned int         IR3Display;

#include          "Hodômetro.c"
```

ANEXO 9:ARQUIVO HODÔMETRO.C DA TAREFA DE HODOMETRIA DA CADEIRA DE RODAS

```
float CalculaVelocidade(ENCODER *Enc){
	float AuxReturn;
	ENCODER Dado;
	unsigned long int Periodo;
	unsigned long int AuxPeriodo;

	_DINT();
	Dado.Atual = Enc->Atual;
	Dado.Anterior = Enc->Anterior;
	Dado.OverFlow = Enc->OverFlow;
	Dado.OverFlowCap = Enc->OverFlowCap;
	Dado.Sentido = Enc->Sentido;
	_EINT();


	Periodo = (Dado.OverFlowCap + (unsigned long int)(Dado.Atual)) - (unsigned long int)(Dado.Anterior);

	if (Dado.OverFlow > 0){
		AuxPeriodo = Dado.OverFlow - (unsigned long int)(Dado.Atual);
	}
	else{
		AuxPeriodo = 0x00010000 - (unsigned long int)(Dado.Atual);
	}

	if (((Enc->UpDate) == 0) && (AuxPeriodo > Periodo))
		Periodo = AuxPeriodo;

	Enc->UpDate = 0;

	AuxReturn = CONSTANTE_VELOCIDADE / (Periodo);
	if (AuxReturn > 3000)
		AuxReturn = 3000;

	if (Dado.Sentido){
		return AuxReturn;
	}
	else{
		return (-1)*AuxReturn;
	}

}

void TaskControleDos*Encoders* (void *pdata){
	HANDLE_CAN_MSG		MSGSpeedData;
	HANDLE_CAN_MSG		MSGPosicaoData;
	HANDLE_CAN_MSG		MSGTetaData;
	HANDLE_CAN_MSG	MSG_IRData;
	POSICAO		Posicao;
	POSICAO		PosicaoAux;
	float		SomatorioPosicao_X;
    float		SomatorioPosicao_Y;
	float		Vl = 0;
	float		Vw = 0;
    float		Vl_Ant = 0;
```

```
    float               Vl_Medio = 0;

float                           SomatorioTeta;
    long int            Teta = 0;
    float                       TetaAux = 0;
float                           TetaAux_Ant = 0;
float           Teta_Medio = 0;

    float                   VreAux;
    float                   VrdAux;
float           PeriodoAmostragem;
    unsigned long int   IR0Aux;
    unsigned long int   IR1Aux;
    unsigned long int   IR2Aux;
    unsigned long int   IR3Aux;


    pdata = pdata;

    TBCTL = TBCLGRP_0 + CNTL_0 + TBSSEL_2 + ID_0 + MC_2 + TBIE;
    TBCCTL0 = CCIE;
    TBCCTL1 = CM_1 + SCS + CAP + CCIE + CCIS_1;
    TBCCTL2 = CM_1 + SCS + CAP + CCIS_1;
    TBCCTL3 = CM_1 + SCS + CAP + CCIE + CCIS_1;
    TBCCTL4 = CM_1 + SCS + CAP + CCIS_1;
    TBCCR0 = 0;

    CONFIG_ENCODER_DIR_PORT_DIR;
    CONFIG_ENCODER_DIR_PORT_SEL;
    CONFIG_ENCODER_ESQ_PORT_DIR;
    CONFIG_ENCODER_ESQ_PORT_SEL;

    TBCCTL5 = CCIE;
    P4DIR |= 0x80;
    P4SEL &= (~0x80);
    P7DIR &= (~0x0F);
    P7SEL &= (~0x0F);
    IR_CLK_HIGH;

    SomatorioTeta = 0;
    SomatorioPosicao_X = 0;
    SomatorioPosicao_Y = 0;
Vl = 0;
Vl_Ant = 0;
TetaAux = 0;
TetaAux_Ant = 0;
Vre = 0;
Vrd = 0;
VreAux = 0;
VrdAux = 0;

PeriodoAmostragem = PERIODO_ENCODER_CONTROL_TASK;
PeriodoAmostragem = PeriodoAmostragem / 1000;

    MSGSpeedData    = InsertOnBus (0x0111,0,(void *)&Vre,8,0);
    MSG_IRData      = InsertOnBus (0x0113,0,(void *)&IR.Distancia0,8,0);
    MSGPosicaoData  = InsertOnBus (0x0123,0,(void *)&Posicao,8,0);
    MSGTetaData     = InsertOnBus (0x0124,0,(void *)&Teta,4,0);



    while(1){
            VrdAux = CalculaVelocidade(&EncDir);
            VreAux = CalculaVelocidade(&EncEsq);
```

```
                CANMSGSemPend(MSGSpeedData);
                Vre = (long int)(VreAux);
                Vrd = (long int)(VrdAux);
                CANMSGSemPost(MSGSpeedData);

        Vl_Ant = Vl;
                Vl = (VrdAux + VreAux) / 2;
                Vw = (VrdAux - VreAux) / DISTANCIA_ENTRE_RODAS;

        TetaAux_Ant = TetaAux;
        SomatorioTeta += Vw;
        TetaAux = SomatorioTeta * PeriodoAmostragem;

        CANMSGSemPend(MSGTetaData);
                Teta = (long int)(TetaAux * 1000);
                CANMSGSemPost(MSGTetaData);

                Vl_Medio = (Vl + Vl_Ant) / 2;
        Teta_Medio = (TetaAux + TetaAux_Ant) / 2;

        SomatorioPosicao_X += Vl_Medio*cos(Teta_Medio);
                SomatorioPosicao_Y += Vl_Medio*sin(Teta_Medio);

                PosicaoAux.X = (long int)(SomatorioPosicao_X * PeriodoAmostragem);
                PosicaoAux.Y = (long int)(SomatorioPosicao_Y * PeriodoAmostragem);

                CANMSGSemPend(MSGPosicaoData);
                Posicao.X = PosicaoAux.X;
                Posicao.Y = PosicaoAux.Y;
                CANMSGSemPost(MSGPosicaoData);

                if (IR.NewData){
                        IR.NewData = 0;
                        IR0Aux= IR.Data0;
                        IR0Aux = (156000/((10*IR0Aux) - 694)) - 5;

                        IR1Aux= IR.Data1;
                        IR1Aux = (156000/((10*IR1Aux) - 694)) - 5;

                        IR2Aux= IR.Data2;
                        IR2Aux = (156000/((10*IR2Aux) - 694)) - 5;

                        IR3Aux= IR.Data3;
                        IR3Aux = (156000/((10*IR3Aux) - 694)) - 5;

                        CANMSGSemPend(MSG_IRData);
                        IR.Distancia0 = (unsigned int)(IR0Aux);
                        IR.Distancia1 = (unsigned int)(IR1Aux);
                        IR.Distancia2 = (unsigned int)(IR2Aux);
                        IR.Distancia3 = (unsigned int)(IR3Aux);
                        CANMSGSemPost(MSG_IRData);
                        IR3Display = IR.Distancia3;
                }

                OSTimeDly(PERIODO_ENCODER_CONTROL_TASK);
        }
}
```

ANEXO 10:ARQUIVO DOSPONCHIP.H. ARQUIVO CABEÇALHO DAS FUNÇÕES DE COMANDO DO CD17B10

```
#define DOS_FIRMWARE_MAJOR              0x02 /* expected firmware (major) version */
#define DOS_FIRMWARE_MINOR              0x02 /* expected firmware (minor) version */
#define DOS_MAX_HANDLES                 4    /* Firmware 2.x limit */

/* PACKET CONSTANTS */
#define DOS_PACKET_LENGTH_HEADER        1   /* single header byte */
#define DOS_PACKET_LENGTH_PAYLOAD_MAX   512 /* default & maximum DATA_BLOCK payload length */
#define DOS_PACKET_LENGTH_PAYLOAD_NAME  12  /* (8+1+3) default & maximum DATA_NAME packet payload length */
#define DOS_PACKET_LENGTH_PAYLOAD_CMDRES 4  /* command & response payload length */
#define DOS_PACKET_LENGTH_PAYLOAD_HANDSHAKE 0  /* handshake payload length */
#define DOS_PACKET_LENGTH_FOOTER        0   /* no footer at this time */

/* PACKET HEADER DEFINITIONS */
#define DOS_HANDSHAKE_OFF               0xFF
#define DOS_HANDSHAKE_WAIT              0x00
#define DOS_HANDSHAKE_POLL              DOS_HANDSHAKE_WAIT
#define DOS_HANDSHAKE_GO                0xEA
#define DOS_HANDSHAKE_PAK_NEXT          0xE0
#define DOS_HANDSHAKE_PAK_ABORT         0xE1

#define DOS_DATA_NAME                   0x7B
#define DOS_DATA_BLOCK                  0x7C
#define DOS_DATA_BLOCK_END              0x7D

#define DOS_COMMAND_CARD_ACCESS_MIN     0x5B /* NON-COMMAND: minimum command header byte value for a command requiring card access */

#define DOS_CMD_GET_ID                  0x40 /* "@" : get the DOSonCHIP silicon version */
#define DOS_CMD_UPDATE                  0x42 /* "B" : update firmware via bootloader */
#define DOS_CMD_GET_NAME                0x47 /* "G" : get current file/directory name */
#define DOS_CMD_SET_HANDLE              0x48 /* "H" : set the current handle number */
#define DOS_CMD_SET_BLOCK_LEN           0x4C /* "L" : set the byte length for transferring data in a single packet transfer */
#define DOS_CMD_SET_NAME_LEN            0x4D /* "M" : set the byte length for the file/directory name */
#define DOS_CMD_SET_NAME                0x4E /* "N" : set the current file/directory name */
#define DOS_CMD_SET_TIME_ON_OFF         0x52 /* "R" : enable/disable the real-time clock to commence incrementing */
#define DOS_CMD_SET_TIME                0x53 /* "S" : set the real-time clock date & time */
#define DOS_CMD_GET_TIME                0x54 /* "T" : get the real-time clock date & time */
#define DOS_CMD_GET_VERSION             0x56 /* "V" : get the DOSonCHIP firmware versions (bootloader & filesystem) */
#define DOS_CMD_SHUTDOWN                0x5A /* "Z" : place the DOSonCHIP into minimal power shut down */
#define DOS_CMD_DIR                     0x5C /* "\" : retrieve directory entry */
#define DOS_CMD_MOUNT                   0x5F /* "_" : initialize the card file system */
#define DOS_CMD_WRITE_PREALLOCATE       0x61 /* "a" : add bytes to current file open for writing */
#define DOS_CMD_CLOSE                   0x63 /* "c" : flush any pending writes to the file, update the modification time stamp, and clear the handle number */
#define DOS_CMD_DELETE                  0x64 /* "d" : delete the current directory entry */
#define DOS_CMD_DIR_GET_PROPERTY        0x69 /* "i" : get the current directory entry property (used with DOS_CMD_DIR) */
#define DOS_CMD_SET_DIR                 0x6C /* "l" : set the current directory */
#define DOS_CMD_MAKE_DIR                0x6D /* "m" : make a new directory within the current directory */
```

```
#define DOS_CMD_OPEN_WRITE              0x6F /* "o" : open a file for writing */
#define DOS_CMD_OPEN_READ               0x70 /* "p" : open a file for reading */
#define DOS_CMD_READ_DATA               0x72 /* "r" : read data from a file */
#define DOS_CMD_SEEK                    0x73 /* "s" : change the current position pointer in a file for subsequent reading or writing */
#define DOS_CMD_GET_FREE_SECTORS        0x75 /* "u" : get the amount of available but unused sectors for the current file system */
#define DOS_CMD_WRITE_DATA              0x77 /* "w" : write data to a file */

/* COMMAND PARAMETER DEFINITIONS */
#define DOS_OFF                 0x00 /* Disable feature */
#define DOS_ON                  0x01 /* Enable feature */
#define DOS_FIRST               0x00 /* Point to first valid directory entry */
#define DOS_NEXT                0x01 /* Point to next directory entry */

/* FILE ATTRIBUTES (not mutually exclusive) */
#define DOS_ATTR_READ_ONLY      0x01 /* if set, directory entry is read only */
#define DOS_ATTR_HIDDEN         0x02 /* if set, directory entry is hidden */
#define DOS_ATTR_SYSTEM         0x04 /* directory entry system tag bit */
#define DOS_ATTR_DIRECTORY      0x10 /* if set, directory entry is directory (not a file) */
#define DOS_ATTR_ARCHIVE        0x20 /* directory entry archive tag bit */

/* DIRECTORY ENTRY PROPERTIES (for DOS_CMD_DIR_GET_PROPERTY) */
#define DOS_PROPERTY_SIZE               0x00 /* return the current entry's size of file */
#define DOS_PROPERTY_TIME_CREATED       0x01 /* return the current entry's creation date/time stamp */
#define DOS_PROPERTY_TIME_MODIFIED      0x02 /* return the current entry's last modified date/time stamp */

/* RESPONSE HEADER DEFINITIONS */
#define DOS_RES_NOERROR         0x80 /* no error */

#define DOS_RES_CARD_ERROR              0x81 /* card error (card error in payload) */
#define DOS_RES_CARD_NOT_DETECTED       0x82 /* card not detected */
#define DOS_RES_CARD_INIT_FAILURE       0x83 /* card could not be initialized */
#define DOS_RES_CARD_BLOCK_LENGTH_FAILURE   0x84 /* either card block length is improper or card could not have block length set */
#define DOS_RES_CARD_VOLTAGE_OUT_OF_RANGE   0x85 /* card required voltage not within allowable range */
#define DOS_RES_CARD_NOT_MOUNTED        0x86 /* card volume is not mounted */

#define DOS_RES_INVALID_COMMAND         0x90 /* invalid command */
#define DOS_RES_INVALID_PACKET_TYPE     0x91 /* improper packet type than expected */
#define DOS_RES_INVALID_PARAMETER       0x92 /* parameter out of range */
#define DOS_RES_INVALID_OPERATION       0x93 /* cannot perform this operation--not allowed */

#define DOS_RES_DISK_FORMAT_INCOMPATIBLE    0x98 /* card file system is not supported (please reformat card) */
#define DOS_RES_DISK_FULL               0x99 /* disk full error */
#define DOS_RES_DISK_ROOT_DIR_FULL      0x9A /* root directory full error (FAT16) */

#define DOS_RES_NAME_ERROR              0xA0 /* invalid directory name/not a directory/filename error */
#define DOS_RES_NAME_NOT_FOUND          0xA1 /* file/directory not found; entry does not exist in specified dir */
#define DOS_RES_NAME_DUPLICATE          0xA2 /* duplicate name error; file/directory already exists */

#define DOS_RES_HANDLE_INVALID          0xA8 /* invalid handle/handle out of range */
#define DOS_RES_HANDLE_PREVIOUSLY_ASSIGNED  0xA9 /* handle previously assigned */

#define DOS_RES_FILE_END                0xB0 /* end-of-file */
#define DOS_RES_FILE_READ_ONLY          0xB1 /* read-only error/write access is denied */
#define DOS_RES_FILE_OPEN_PREVIOUS      0xB2 /* file already open */
#define DOS_RES_FILE_NOT_OPEN_FOR_READ  0xB3 /* not opened for read operation error ;should this be generic 'cannot apply this operation' ??? */
#define DOS_RES_FILE_NOT_OPEN_FOR_WRITE 0xB4 /* not opened for write operation error */
```

```
#define DOS_RES_FILE_CANNOT_DELETE          0xB5 /* open file cannot be deleted error */
#define DOS_RES_FILE_TOO_LARGE              0xB6 /* file would be too large */

#define DOS_RES_DIR_END                     0xC0 /* directory iterator at end */
#define DOS_RES_DIR_NOT_INITIALIZED         0xC1 /* directory iterator is not initialized */

/* Highest allowable error code is 0xDF */
```

ANEXO 10:ARQUIVO DOSPONCHIP.C. ARQUIVO DAS FUNÇÕES DE COMANDO DO CD17B10

```
unsigned int PonteiroSetPointSDTX = 0;
unsigned int PonteiroSDTX = 0;
char *BufferSDTX;
char BufferSDRX[64];
OS_EVENT   *SemRXSD;

extern OS_EVENT   *SemRXSD;
extern OS_EVENT   *SemTXSD;
void PrintSD (char *DataTX,char BytesToTX,char HS);
char ScanSD (unsigned int);


char HandShakDosToHost(void){

        while ((PonteiroSetPointSDTX != PonteiroSDTX) || (UCA0STAT & UCBUSY)){
                OSTimeDly(1);
        }
        _DINT();
        PonteiroSDTX = 1;
        PonteiroSetPointSDTX = 1;
        _EINT();
        UCA0TXBUF = DOS_HANDSHAKE_POLL;
        while(ScanSD(10) == DOS_HANDSHAKE_WAIT){
                _DINT();
                PonteiroSDTX = 1;
                PonteiroSetPointSDTX = 1;
                _EINT();
                UCA0TXBUF = DOS_HANDSHAKE_POLL;
        }
        return(0);
}

char ScanSD (unsigned int TimeOut){
static char Ponteiro;
        P4OUT |= (0x01);
        OSSemPend(SemRXSD, TimeOut, &err);
        P4OUT &= (~0x01);
        if ( err != OS_ERR_TIMEOUT){
                Ponteiro++;
                Ponteiro &= 0x3F;
                return(*(BufferSDRX + Ponteiro));
        }
        else{
                return(0xFF);
        }
}


void PrintSD (char *DataTX,char BytesToTX,char HS){

        P4OUT |= (0x01);
        if (HS)
                HandShakDosToHost();

        while ((PonteiroSetPointSDTX != PonteiroSDTX) || (UCA0STAT & UCBUSY)){
```

```
                OSTimeDly(1);
        }
        _DINT();
        BufferSDTX = DataTX;
        PonteiroSDTX = 1;
        PonteiroSetPointSDTX = BytesToTX;
        _EINT();
        UCA0TXBUF = *DataTX;
        P4OUT &= (~0x01);
}


void ISR_TX_SD (void){
        if (IFG2 & UCA0TXIFG){
                if (PonteiroSDTX != PonteiroSetPointSDTX){
                        UCA0TXBUF = *(BufferSDTX + PonteiroSDTX);
                        PonteiroSDTX ++;
                }
                else{
                        IFG2 &= (~UCA0TXIFG);
                }
        }
}

void ISR_RX_SD (void){
static char Ponteiro;
        if (IFG2 & UCA0RXIFG){
                Ponteiro ++;
                Ponteiro &= 0x3F;
                *(BufferSDRX + Ponteiro) = UCA0RXBUF;
                OSSemPost(SemRXSD);
        }
}

////////////////////////////////////////////////////////
char InitDos(void){
char DataTX[2];
char DataRX;

        /**(DataTX) = 0x0D;
        *(DataTX + 1) = 0x0D;
        PrintSD(DataTX,2,0);
*/
        *(DataTX) = 0x0D;
        PrintSD(DataTX,1,0);
        *(DataTX) = 0x0D;
        PrintSD(DataTX,1,0);


        do{
                DataRX = ScanSD(1000);
                switch(DataRX){
                case DOS_HANDSHAKE_WAIT:
                        break;
                default:
                        return(DataRX);
                }
        }while (1);
}



////////////////////////////////////////////////////////
```

```
char ClockDosControl(char Data){

char DataTX[6];
char DataRX;

	*(DataTX + 0) = DOS_CMD_SET_TIME_ON_OFF;
	*(DataTX + 1) = 0;
	*(DataTX + 2) = 0;
	*(DataTX + 3) = 0;
	*(DataTX + 4) = Data;
	PrintSD(DataTX,5,1);

	do{
		DataRX = ScanSD(100);
		switch(DataRX){
		case DOS_HANDSHAKE_WAIT:
			break;
		default:
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			return(DataRX);
		}
	}while (1);
}

char DosMountSD(void){
char DataTX[6];
char DataRX;
	*(DataTX + 0) = DOS_CMD_MOUNT;
	*(DataTX + 1) = 0xFF;
	*(DataTX + 2) = 0xFF;
	*(DataTX + 3) = 0xFF;
	*(DataTX + 4) = 0xFF;
	PrintSD(DataTX,5,1);
	P4OUT |= 0x01;

	do{
		DataRX = ScanSD(65500);
		switch(DataRX){
		case DOS_HANDSHAKE_GO:
		case DOS_HANDSHAKE_WAIT:
			break;
		case 0xFF:
			return(DataRX);
		default:
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			return(DataRX);
		}
	}while (1);
}

char SetNameBufferDos (char *Data){
char DataTX[26];
char DataRX;
char Len;

	Len = strlen(Data);
	*(DataTX + 0) = DOS_CMD_SET_NAME_LEN;
	*(DataTX + 1) = 0;
	*(DataTX + 2) = 0;
```

```
	*(DataTX + 3) = 0;
	*(DataTX + 4) = Len;
	PrintSD(DataTX,5,1);
	do{
		DataRX = ScanSD(100);
		switch(DataRX){
		case DOS_RES_NOERROR:
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			break;
		case DOS_HANDSHAKE_WAIT:
			break;
		default:
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			return(DataRX);
		}
	}while(DataRX == DOS_HANDSHAKE_WAIT);

	*(DataTX + 0) = DOS_CMD_SET_NAME;
	PrintSD(DataTX,5,0);
	*(DataTX + 0) = DOS_DATA_NAME;
	PrintSD(DataTX,1,0);
	strcpy(DataTX,Data);
	PrintSD(DataTX,Len,0);
	do{
		DataRX = ScanSD(100);
		switch(DataRX){
		case DOS_RES_NOERROR:
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			return(DataRX);

		case DOS_HANDSHAKE_WAIT:
			break;
		default:
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			return(DataRX);
		}
	}while(1);

}

char MakeDirDos(char *Data){
char DataTX[6];
char DataRX;

	SetName*Buffer*Dos(Data);
	*(DataTX + 0) = DOS_CMD_MAKE_DIR;
	*(DataTX + 1) = 0xFF;
	*(DataTX + 2) = 0xFF;
	*(DataTX + 3) = 0xFF;
	*(DataTX + 4) = 0xFF;
	PrintSD(DataTX,5,1);

	do{
```

```
                DataRX = ScanSD(1000);
                switch(DataRX){
                case DOS_HANDSHAKE_WAIT:
                case DOS_HANDSHAKE_GO:
                        break;
                default:
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        return(DataRX);
                }
        }while (1);
}

char SetDirDos(char *Data){
char DataTX[6];
char DataRX;

        SetName*Buffer*Dos(Data);
        *(DataTX + 0) = DOS_CMD_SET_DIR;
        *(DataTX + 1) = 0xFF;
        *(DataTX + 2) = 0xFF;
        *(DataTX + 3) = 0xFF;
        *(DataTX + 4) = 0xFF;
        PrintSD(DataTX,5,1);
        do{
                DataRX = ScanSD(100);
                switch(DataRX){
                case DOS_HANDSHAKE_WAIT:
                case DOS_HANDSHAKE_GO:
                        break;
                default:
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        return(DataRX);
                }
        }while (1);
}



char SetFileHandleDos(char Data){
char DataTX[6];
char DataRX;

        *(DataTX + 0) = DOS_CMD_SET_HANDLE;
        *(DataTX + 1) = 0;
        *(DataTX + 2) = 0;
        *(DataTX + 3) = 0;
        *(DataTX + 4) = Data;
        PrintSD(DataTX,5,1);

        do{
                DataRX = ScanSD(100);
                switch(DataRX){
                case DOS_HANDSHAKE_WAIT:
                case DOS_HANDSHAKE_GO:
                        break;
                default:
                        ScanSD(100);
                        ScanSD(100);
```

```
                        ScanSD(100);
                        ScanSD(100);
                        return(DataRX);
                }
        }while (1);
}

unsigned long int OpenFileWriteModeDos(char *Data){
char DataTX[6];
char DataRX;
unsigned long int Aux;

        SetName*Buffer*Dos(Data);
        //SetFileHandleDos(Handle);
        *(DataTX + 0) = DOS_CMD_OPEN_WRITE;
        *(DataTX + 1) = 0;
        *(DataTX + 2) = 0;
        *(DataTX + 3) = 0;
        *(DataTX + 4) = 0;
        PrintSD(DataTX,5,1);


        do{
                DataRX = ScanSD(100);
                switch(DataRX){
                case DOS_RES_NOERROR:
                        Aux = ((unsigned long int)(ScanSD(100))) << 24;
                        Aux += ((unsigned long int)(ScanSD(100))) << 16;
                        Aux += ((unsigned long int)(ScanSD(100))) << 8;
                        Aux += ((unsigned long int)(ScanSD(100)));
                        Aux |= 0x80000000;
                        return(Aux);
                case DOS_HANDSHAKE_WAIT:
                case DOS_HANDSHAKE_GO:
                        break;
                default:
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        return(DataRX);
                }
        }while(1);

}

char CloseFileDos(char Handle){
char DataTX[6];
char DataRX;
        SetFileHandleDos(Handle);
        *(DataTX + 0) = DOS_CMD_CLOSE;
        *(DataTX + 1) = 0;
        *(DataTX + 2) = 0;
        *(DataTX + 3) = 0;
        *(DataTX + 4) = 0;
        PrintSD(DataTX,5,1);

        do{
                DataRX = ScanSD(100);
                switch(DataRX){
                case DOS_HANDSHAKE_WAIT:
                case DOS_HANDSHAKE_GO:
                        break;
                default:
                        ScanSD(100);
```

```
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        return(DataRX);
                }
        }while (1);
}

char WriteDataDos(char *Data, unsigned int Len){
char DataRX;
char DataTX[6];
static unsigned int BlockLen;
unsigned int BlockAcc = 0;

        if (BlockLen == 0)
                BlockLen = Len;



        *(DataTX + 0) = DOS_CMD_WRITE_DATA;
        *(DataTX + 1) = 0;
        *(DataTX + 2) = 0;
        *(DataTX + 3) = (char)(Len >> 8);
        *(DataTX + 4) = (char)(Len & 0x00FF);
        PrintSD(DataTX,5,1);
        while (1){
                DataRX = ScanSD(65000);

                switch(DataRX){
                case DOS_HANDSHAKE_WAIT:
                        *(DataTX + 0) = DOS_HANDSHAKE_POLL;
                        PrintSD(DataTX,1,0);
                        break;
                case DOS_DATA_BLOCK_END:
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        return(DataRX);
                case DOS_HANDSHAKE_GO:
                        *DataTX = DOS_DATA_BLOCK;
                        PrintSD(DataTX,1,0);
                        PrintSD((Data + BlockAcc),BlockLen,0);
                        BlockAcc += BlockLen;
                        *(DataTX + 0) = DOS_HANDSHAKE_POLL;
                        PrintSD(DataTX,1,0);
                        break;
                case DOS_CMD_SET_BLOCK_LEN:
                        BlockLen = ((unsigned long int)(ScanSD(100))) << 24;
                        BlockLen += ((unsigned long int)(ScanSD(100))) << 16;
                        BlockLen += ((unsigned long int)(ScanSD(100))) << 8;
                        BlockLen += ((unsigned long int)(ScanSD(100)));
                        *(DataTX + 0) = DOS_HANDSHAKE_POLL;
                        PrintSD(DataTX,1,0);
                        break;
                default:
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        return(DataRX);
                }
        }
}
```

```
char PreallocateWriteDataDos(unsigned long int NumBytes,char Handle){
char DataTX[6];
char DataRX;

	SetFileHandleDos(Handle);
	*(DataTX + 0) = DOS_CMD_WRITE_PREALLOCATE;
	*(DataTX + 1) = (char)(NumBytes >> 24);
	*(DataTX + 2) = (char)(NumBytes >> 16);
	*(DataTX + 3) = (char)(NumBytes >> 8);
	*(DataTX + 4) = (char)(NumBytes);
	PrintSD(DataTX,5,1);

	do{
		DataRX = ScanSD(65500);
		switch(DataRX){
		case DOS_HANDSHAKE_WAIT:
		case DOS_HANDSHAKE_GO:
			break;
		default:
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			ScanSD(100);
			return(DataRX);
		}
	}while (1);
}

char SetClockDos(char Ano,char Mes, char DiaMes,char Horas,char Minutos,char Segundos){
char DataTX[6];
char DataRX;
unsigned long int Dias;

Dias = (Ano + 32)/4;
Dias = Dias + 365*(Ano + 30);

	switch (Mes){
	case 1:
		Dias += 0;
		break;
	case 2:
		Dias += 31;
		break;
	case 3:
		Dias += 59;
		break;
	case 4:
		Dias += 90;
		break;
	case 5:
		Dias += 120;
		break;
	case 6:
		Dias += 151;
		break;
	case 7:
		Dias += 181;
		break;
	case 8:
		Dias += 212;
		break;
	case 9:
		Dias += 243;
		break;
	case 10:
```

```
                Dias += 273;
                break;
        case 11:
                Dias += 304;
                break;
        case 12:
                Dias += 334;
                break;
        }

        Dias += DiaMes;
        Dias *= 24;
        Dias += Horas;
        Dias *= 60;
        Dias += Minutos;
        Dias *= 60;
        Dias += Segundos;

        *(DataTX + 0) = DOS_CMD_SET_TIME;
        *(DataTX + 1) = (char)(Dias >> 24);
        *(DataTX + 2) = (char)(Dias >> 16);
        *(DataTX + 3) = (char)(Dias >> 8);
        *(DataTX + 4) = (char)(Dias);
        PrintSD(DataTX,5,1);

        do{
                DataRX = ScanSD(100);
                switch(DataRX){
                case DOS_HANDSHAKE_WAIT:
                        break;
                default:
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        return(DataRX);
                }
        }while (1);

}

char Get*Clock*Dos(void){
char DataTX[6];
char DataRX;

        *(DataTX + 0) = DOS_CMD_GET_TIME;
        *(DataTX + 1) = 0xFF;
        *(DataTX + 2) = 0xFF;
        *(DataTX + 3) = 0xFF;
        *(DataTX + 4) = 0xFF;
        PrintSD(DataTX,5,1);

        do{
                DataRX = ScanSD(100);
                switch(DataRX){
                case DOS_HANDSHAKE_WAIT:
                        break;
                default:
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        ScanSD(100);
                        return(DataRX);
                }
        }while (1);
```

}

ANEXO 11:ARQUIVO SD_CONTROL_TASK.H. ARQUIVO HEADER DO CÓDIGO C DAS TAREFAS SDCONTROL E AMOSTRADOR.

```
#define LED_SDCARD_BIT              0x01
#define LED_SDCARD_PORT             P4OUT
#define SD2MSP                      P3IN & 0x08
#define SD2MSP_DIR                  P3DIR &= (~0x08)
#define SD2MSP_SEL                  P3SEL &= (~0x08)

#define MSP2SD_LOW                  P3OUT &= (~0x01)
#define MSP2SD_HIGH                 P3OUT |= (0x01)
#define SD_DIR_PORT                 P3DIR |= (0x01)
#define SD_SEL_PORT                 P3SEL &= (~0x09)

#define LED_SD_DIR_PORT             P4DIR |= (0x01)
#define LED_SD_SEL_PORT             P4SEL &= (~0x01)
#define TENSAO_MINIMA               1160
#define COUNT_ERRO_LIMIT            100
#define BUFFERSIZE                  200

////////////////////////////////////////////////////////////////
//Configuraçao de endereço na EEPROM

#define ADDRESS_SAMPLE_TIME         1000


//Comandos LED
#define LED_SDCARD_OFF              LED_SDCARD_PORT &= (~LED_SDCARD_BIT)
#define LED_SDCARD_ON               LED_SDCARD_PORT |= (LED_SDCARD_BIT)
#define LED_SDCARD_TEST             LED_SDCARD_PORT & LED_SDCARD_BIT


char                    Buffer1[BUFFERSIZE];
char                    Buffer2[BUFFERSIZE];
char                    StatusSDControlTask;
char                    *BufferRecData;
OS_EVENT                *SemSampleTimer;

enum SDCardStatus{
        RESETANDO = 0,
        BAIXA_TENSAO,
        ESCREVENDO,
        INICIANDO,
        DESMONTADO,
        MONTANDO,
        LENDO,
        ERRO_FATAL,
        PARADO,
        ABRINDO_DIRETORIO,
        SEM_SD_CARD,
        ABRINDO_HANDLE,
        ABRINDO_ARQUIVO,
        DESMONTANDO,
        CARTAO_NAO_DETECTADO,
}SDCardStatus;
enum SDCardComando{
        ESCREVER = '0',
        PARAR,
```

```
        REMOVER,
        MONTAR,
        RESETAR,
        LER,
}SDCardComando;


unsigned int    Max*Buffer*1;
unsigned int    Max*Buffer*2;
unsigned int    LastLen*Buffer*1;
unsigned int    LastLen*Buffer*2;
char                    SDResponse;

char            **Buffer*RecData;
unsigned int    PonteiroRecData;
OS_EVENT        *SemRecData;

extern char     Segundos;
extern char     Minutos;
extern char     Horas;
extern char     DiaSemana;
extern char     DiaMes;
extern char     Mes;
extern char     Ano;
extern long int SPVL;
extern long int SPDV;


#include    "SD_Control_Task.c"
```

ANEXO 12:ARQUIVO SD_CONTROL_TASK.C. ARQUIVO DO CÓDIGO C DAS TAREFAS SDCONTROL E AMOSTRADOR.

```
#include    <msp430x26x.h>

//#include    "string.h"                    /* for string operations */
#include              "DosOnChipApp.h"
#include              "DosOnChipApp.c"

extern long int     VreCAN;
extern long int     VrdCAN;
extern int          UeCAN;
extern int          UdCAN;

char PrintText(char *Dest, long int dado);


void SDControl (void *pdata){

        enum MAQUINA_DE_ESTADO{
                RESET_SD,
                BAIXA_TENSAO_ESTADO,
                START_SD,
                SET_TIME,
                MOUNT,
                MAKE_DIR,
                SET_DIR,
                SET_HANDLE_FILE,
                MAKE_FILE_WRITE_MODE,
                SCAN_RECORD,
                SCAN_COMMAND,
                IDLE,
                DESMONTA_CARTAO,
                SD_REMOVED,
                ERRO_IRREPARAVEL,
                ERRO_IRREPARAVEL_1,
                NO_DOSONCHIP,
        }Estado;

        unsigned int Pointer;
        unsigned long int Aux2;
        char NBuffer;
        //char Estado;
        char FileName[25];
        char Handle;
        //char Aux;


                UCA0CTL1 = UCSWRST;
                UCA0CTL0 = 0;
                UCA0CTL1 |= UCSSEL_2;
                UCA0BR0 = 26;
                UCA0BR1 = 0;
                UCA0MCTL = 0x11;

                P3DIR |= 0x10;
                P3SEL |= 0x30;
                UCA0CTL1 &= (~UCSWRST);
```

```
_DINT();
IE2 |= UCA0TXIE;
IE2 |= UCA0RXIE;
IFG2 &= (~UCA0TXIFG);
IFG2 &= (~UCA0RXIFG);
_EINT();

LED_SD_DIR_PORT;
LED_SD_SEL_PORT;

SD_DIR_PORT;
SD_SEL_PORT;
MSP2SD_LOW;

SD2MSP_DIR;
SD2MSP_SEL;

P4REN |= 0x08;
P4DIR |= 0x08;
P4SEL &= (~0x08);

pdata = pdata;
Estado = RESET_SD;
SDCard*Status* = RESETANDO;
Handle = 0;
SemRXSD = OSSemCreate(0);
SemRecData = OSSemCreate(0);


while (1){
        switch (Estado){
        case NO_DOSONCHIP:
                OSTimeDly(200);
                if (TensaoAlimentacao > TENSAO_MINIMA){
                        SDResponse = InitDos();
                        switch (SDResponse){
                        case 0xFF:
                                Estado = NO_DOSONCHIP;
                                SDCard*Status* = SEM_SD_CARD;
                                break;
                        default:
                                Estado = RESET_SD;
                                SDCard*Status* = RESETANDO;
                                break;
                        }
                }
                else{
                        Estado = BAIXA_TENSAO_ESTADO;
                }
                break;
        case BAIXA_TENSAO_ESTADO:
                OSTimeDly(500);
                SDCard*Status* = BAIXA_TENSAO;
                if (TensaoAlimentacao > TENSAO_MINIMA){
                        Estado = RESET_SD;
                }
                break;
        case RESET_SD:
                OSTimeDly(1000);
                if (TensaoAlimentacao > TENSAO_MINIMA){
                        P4OUT &= (~0x08);
                        OSTimeDly(200);
                        P4OUT |= (0x08);
                        OSTimeDly(1000);
                        Estado = START_SD;
```

```
                                        SDCard Status = INICIANDO;
                                        Max Buffer 1 = 0;
                                        Max Buffer 2 = 0;
                                }
                                else{
                                        Estado = BAIXA_TENSAO_ESTADO;
                                }
                                break;
        ////////////////////////////////////////////////////////////////////////////
                        case START_SD:
                                if (TensaoAlimentacao > TENSAO_MINIMA){
                                        SDResponse = InitDos();
                                        switch (SDResponse){
                                        case 0xFF:
                                                Estado = NO_DOSONCHIP;
                                                SDCard Status = SEM_SD_CARD;
                                                break;
                                        default:
                                                Estado = SET_TIME;
                                                break;
                                        }
                                }
                                else{
                                        Estado = BAIXA_TENSAO_ESTADO;
                                        SDCard Status = BAIXA_TENSAO;
                                }
                                OSTimeDly(50);
                                break;
        ////////////////////////////////////////////////////////////////////////////
                        case SET_TIME:
                                if (TensaoAlimentacao > TENSAO_MINIMA){
                                        SDResponse                                              =
Set Clock Dos(Ano,Mes,DiaMes,Horas,Minutos,Segundos);
                                        SDResponse = Clock DosControl(DOS_ON);
                                        switch (SDResponse){
                                        case 0xFF:
                                                Estado = NO_DOSONCHIP;
                                                SDCard Status = SEM_SD_CARD;
                                                break;
                                        default:
                                                Estado = MOUNT;
                                                SDCard Status = MONTANDO;
                                                break;
                                        }
                                }
                                else{
                                        Estado = BAIXA_TENSAO_ESTADO;
                                        SDCard Status = BAIXA_TENSAO;
                                }
                                break;
        ////////////////////////////////////////////////////////////////////////////
                        case MOUNT:
                                if (TensaoAlimentacao > TENSAO_MINIMA){
                                        SDResponse = DosMountSD();
                                        switch (SDResponse){
                                        case DOS_RES_NOERROR:
                                                Estado = MAKE_DIR;
                                                SDCard Status = ABRINDO_DIRETORIO;
                                                break;
                                        case DOS_RES_CARD_NOT_MOUNTED:
                                                OSTimeDly(500);
                                                break;
                                        case DOS_RES_CARD_NOT_DETECTED:
                                                SDCard Status = CARTAO_NAO_DETECTADO;
                                                OSTimeDly(1000);
```

```
                                        break;
                                case DOS_RES_CARD_INIT_FAILURE:
                                        Estado = RESET_SD;
                                        SDCardStatus = RESETANDO;
                                        break;
                                case 0xFF:
                                        Estado = NO_DOSONCHIP;
                                        SDCardStatus = SEM_SD_CARD;
                                        break;
                                default:
                                        Estado = ERRO_IRREPARAVEL;
                                        SDCardStatus = ERRO_FATAL;
                                        break;
                                }
                                P4OUT &= (~0x01);
                        }
                        else{
                                Estado = BAIXA_TENSAO_ESTADO;
                                SDCardStatus = BAIXA_TENSAO;
                        }
                        break;
////////////////////////////////////////////////////////////////////////////
                case MAKE_DIR:
                        if (TensaoAlimentacao > TENSAO_MINIMA){
                                SDResponse = MakeDirDos("LOG");
                                switch (SDResponse){
                                case DOS_RES_NOERROR:
                                        Estado = SET_DIR;
                                        break;
                                case DOS_RES_CARD_NOT_MOUNTED:
                                        Estado = MOUNT;
                                        SDCardStatus = MONTANDO;
                                        break;
                                case DOS_RES_CARD_NOT_DETECTED:
                                        Estado = MOUNT;
                                        SDCardStatus = MONTANDO;
                                        break;
                                case DOS_RES_CARD_INIT_FAILURE:
                                        Estado = RESET_SD;
                                        SDCardStatus = RESETANDO;
                                        break;
                                case DOS_RES_NAME_DUPLICATE:
                                        Estado = SET_DIR;
                                        break;
                                case DOS_RES_DIR_NOT_INITIALIZED:
                                        OSTimeDly(500);
                                        break;
                                case 0xFF:
                                        Estado = NO_DOSONCHIP;
                                        SDCardStatus = SEM_SD_CARD;
                                        break;
                                default:
                                        Estado = ERRO_IRREPARAVEL;
                                        SDCardStatus = ERRO_FATAL;
                                        break;
                                }
                                P4OUT &= (~0x01);
                        }
                        else{
                                Estado = BAIXA_TENSAO_ESTADO;
                                SDCardStatus = BAIXA_TENSAO;
                        }
                        break;
////////////////////////////////////////////////////////////////////////////
                case SET_DIR:
```

```
                                        if (TensaoAlimentacao > TENSAO_MINIMA){
                                                SDResponse = SetDirDos("LOG");
                                                switch (SDResponse){
                                                case DOS_RES_NOERROR:
                                                        Estado = SET_HANDLE_FILE;
                                                        SDCard Status = ABRINDO_HANDLE;
                                                        break;
                                                case DOS_RES_CARD_NOT_MOUNTED:
                                                        Estado = MOUNT;
                                                        SDCard Status = MONTANDO;
                                                        break;
                                                case DOS_RES_CARD_NOT_DETECTED:
                                                        Estado = MOUNT;
                                                        SDCard Status = MONTANDO;
                                                        break;
                                                case DOS_RES_CARD_INIT_FAILURE:
                                                        OSTimeDly(1000);
                                                        Estado = RESET_SD;
                                                        SDCard Status = RESETANDO;
                                                        break;
                                                case DOS_RES_NAME_DUPLICATE:
                                                        Estado = SET_HANDLE_FILE;
                                                        SDCard Status = ABRINDO_HANDLE;
                                                        break;
                                                case 0xFF:
                                                        Estado = NO_DOSONCHIP;
                                                        SDCard Status = SEM_SD_CARD;
                                                        break;
                                                default:
                                                        Estado = ERRO_IRREPARAVEL;
                                                        SDCard Status = ERRO_FATAL;
                                                        break;
                                                }
                                                P4OUT &= (~0x01);
                                        }
                                        else{
                                                Estado = BAIXA_TENSAO_ESTADO;
                                                SDCard Status = BAIXA_TENSAO;
                                        }
                                        break;
////////////////////////////////////////////////////////////////////////////////
                                case SET_HANDLE_FILE:
                                        if (TensaoAlimentacao > TENSAO_MINIMA){
                                                SDResponse = SetFileHandleDos(Handle);
                                                switch (SDResponse){
                                                case DOS_RES_NOERROR:
                                                        Estado = MAKE_FILE_WRITE_MODE;
                                                        SDCard Status = ABRINDO_ARQUIVO;
                                                        break;
                                                case DOS_RES_CARD_NOT_MOUNTED:
                                                        Estado = MOUNT;
                                                        SDCard Status = MONTANDO;
                                                        break;
                                                case DOS_RES_CARD_NOT_DETECTED:
                                                        Estado = MOUNT;
                                                        SDCard Status = MONTANDO;
                                                        break;
                                                case DOS_RES_CARD_INIT_FAILURE:
                                                        Estado = RESET_SD;
                                                        SDCard Status = RESETANDO;
                                                        break;
                                                case DOS_RES_HANDLE_INVALID:
                                                        Handle = 0;
                                                        break;
                                                case DOS_RES_HANDLE_PREVIOUSLY_ASSIGNED:
```

```
                                        Handle++;
                                        break;
                                case 0xFF:
                                        Estado = NO_DOSONCHIP;
                                        SDCard*Status* = SEM_SD_CARD;
                                        break;
                                default:
                                        Estado = ERRO_IRREPARAVEL;
                                        SDCard*Status* = ERRO_FATAL;
                                        break;
                                }
                                P4OUT &= (~0x01);
                        }
                        else{
                                Estado = BAIXA_TENSAO_ESTADO;
                                SDCard*Status* = BAIXA_TENSAO;
                        }
                        break;
////////////////////////////////////////////////////////////////////////////////////////////////////
                case MAKE_FILE_WRITE_MODE:
                        if (TensaoAlimentacao > TENSAO_MINIMA){
                                FileName[0] = DiaMes / 10 + 0x30;
                                FileName[1] = DiaMes % 10 + 0x30;
                                FileName[2] = '_';
                                FileName[3] = Horas / 10 + 0x30;
                                FileName[4] = Horas % 10 + 0x30;
                                FileName[5] = '_';
                                FileName[6] = Minutos / 10 + 0x30;
                                FileName[7] = Minutos % 10 + 0x30;
                                FileName[8] = '.';
                                FileName[9] = 'T';
                                FileName[10] = 'X';
                                FileName[11] = 'T';
                                FileName[12] = 0;

                                //strcpy(FileName,"");
                                //strcat(FileName,
                                SDResponse = CloseFileDos(Handle);
                                Aux2 = OpenFileWriteModeDos(FileName);
                                SDResponse = Aux2;
                                switch (Aux2){
                                case DOS_RES_NOERROR:
                                        PonteiroRecData = 0;
                                        N*Buffer* = 0;
                                        *Buffer*RecData = *Buffer*1;
                                        Estado = SCAN_COMMAND;
                                        SDCardComando = ESCREVER;
                                        SDCard*Status* = ESCREVENDO;
                                        break;
                                case DOS_RES_CARD_NOT_MOUNTED:
                                        Estado = MOUNT;
                                        SDCard*Status* = MONTANDO;
                                        break;
                                case DOS_RES_CARD_NOT_DETECTED:
                                        Estado = MOUNT;
                                        SDCard*Status* = MONTANDO;
                                        break;
                                case DOS_RES_CARD_INIT_FAILURE:
                                        Estado = RESET_SD;
                                        SDCard*Status* = RESETANDO;
                                        break;
                                case DOS_RES_NAME_DUPLICATE:
                                        PonteiroRecData = 0;
                                        N*Buffer* = 0;
                                        *Buffer*RecData = *Buffer*1;
```

```
                                        Estado = SCAN_COMMAND;
                                        SDCardComando = ESCREVER;
                                        SDCard Status = ESCREVENDO;
                                        break;
                                case DOS_RES_HANDLE_INVALID:
                                        Estado = SET_HANDLE_FILE;
                                        SDCard Status = ABRINDO_HANDLE;
                                        break;
                                case DOS_RES_HANDLE_PREVIOUSLY_ASSIGNED:
                                        Estado = SET_HANDLE_FILE;
                                        SDCard Status = ABRINDO_HANDLE;
                                        break;
                                case 0xFF:
                                        Estado = NO_DOSONCHIP;
                                        SDCard Status = SEM_SD_CARD;
                                        break;
                                default:
                                        if (Aux2 >= 0x80000000){
                                                PonteiroRecData = 0;
                                                N Buffer = 0;
                                                Buffer RecData = Buffer 1;
                                                Estado = SCAN_COMMAND;
                                                SDCardComando = ESCREVER;
                                                SDCard Status = ESCREVENDO;
                                        }
                                        else{
                                                Estado = ERRO_IRREPARAVEL;
                                                SDCard Status = ERRO_FATAL;
                                        }
                                        break;
                                }
                        }
                        else{
                                Estado = DESMONTA_CARTAO;
                                SDCard Status = DESMONTANDO;
                        }
                        break;
////////////////////////////////////////////////////////////////////////////////////////////////
                case SCAN_COMMAND:
                        if ((TensaoAlimentacao > TENSAO_MINIMA)){
                                switch(SDCardComando){
                                        case ESCREVER:
                                                Estado = SCAN_RECORD;
                                                SDCard Status = ESCREVENDO;
                                                break;
                                        case PARAR:
                                                SDCard Status = PARADO;
                                                OSTimeDly(500);
                                                break;
                                        case REMOVER:
                                                if (SDCard Status != DESMONTADO){
                                                        CloseFileDos(Handle);
                                                        SDCard Status = DESMONTADO;
                                                }
                                                else{
                                                        OSTimeDly(500);
                                                }
                                                break;
                                        case MONTAR:
                                                Estado = MOUNT;
                                                SDCard Status = MONTANDO;
                                                break;
                                        case RESETAR:
                                                Estado = RESET_SD;
                                                SDCard Status = RESETANDO;
```

```
                                        break;
                                default:
                                        Estado = ERRO_IRREPARAVEL;
                                        SDCard Status = ERRO_FATAL;
                                        break;
                        }
                }
                else{
                        Estado = DESMONTA_CARTAO;
                        SDCard Status = DESMONTANDO;
                }
                break;
        case SCAN_RECORD:
                if ((TensaoAlimentacao > TENSAO_MINIMA)){

                        if (PonteiroRecData > 0){
                                switch(N Buffer){
                                case 0:
                                        _DINT();
                                        if (PonteiroRecData > Max Buffer 1)
                                                Max Buffer 1 = PonteiroRecData;
                                        LastLen Buffer 1 = PonteiroRecData;
                                        Pointer = PonteiroRecData;
                                        PonteiroRecData = 0;
                                        Buffer RecData = Buffer 2;
                                        _EINT();
                                        N Buffer = 1;
                                        SDResponse                              =
WriteDataDos(Buffer 1,Pointer);
                                        break;
                                case 1:
                                        _DINT();
                                        if (PonteiroRecData > Max Buffer 2)
                                                Max Buffer 2 = PonteiroRecData;
                                        LastLen Buffer 2 = PonteiroRecData;
                                        Pointer = PonteiroRecData;
                                        PonteiroRecData = 0;
                                        Buffer RecData = Buffer 1;
                                        _EINT();
                                        SDResponse                              =
WriteDataDos(Buffer 2,Pointer);
                                        N Buffer = 0;
                                        break;
                                default:
                                        break;
                                }
                                switch (SDResponse){
                                case DOS_RES_NOERROR:
                                        Estado = SCAN_COMMAND;
                                        break;
                                case DOS_DATA_BLOCK_END:
                                        Estado = SCAN_COMMAND;
                                        break;
                                case DOS_RES_CARD_NOT_MOUNTED:
                                        Estado = MOUNT;
                                        SDCard Status = MONTANDO;
                                        break;
                                case DOS_RES_CARD_NOT_DETECTED:
                                        Estado = MOUNT;
                                        SDCard Status = MONTANDO;
                                        break;
                                case DOS_RES_CARD_INIT_FAILURE:
                                        Estado = RESET_SD;
                                        SDCard Status = RESETANDO;
                                        break;
```

```
                                        case 0xFF:
                                                Estado = NO_DOSONCHIP;
                                                SDCardStatus = SEM_SD_CARD;
                                                break;
                                        case DOS_RES_CARD_ERROR:
                                        case
DOS_RES_CARD_BLOCK_LENGTH_FAILURE:
                                        case DOS_RES_INVALID_COMMAND:
                                        case DOS_RES_INVALID_PACKET_TYPE:
                                        case DOS_RES_INVALID_PARAMETER:
                                        case DOS_RES_INVALID_OPERATION:
                                        case DOS_RES_DISK_FORMAT_INCOMPATIBLE:
                                        case DOS_RES_DISK_FULL:
                                        case DOS_RES_DISK_ROOT_DIR_FULL:
                                        case DOS_RES_FILE_END:
                                        case DOS_RES_FILE_TOO_LARGE:
                                        case DOS_RES_DIR_END:
                                        case DOS_RES_DIR_NOT_INITIALIZED:
                                                Estado = ERRO_IRREPARAVEL;
                                                SDCardStatus = ERRO_FATAL;
                                                break;
                                        default:
                                                Estado = ERRO_IRREPARAVEL;
                                                Estado = SCAN_COMMAND;
                                                break;
                                        }
                                        P4OUT &= (~0x01);
                                        OSTimeDly(4);
                                }
                                else{
                                        OSTimeDly(4);
                                }
                        }
                        else{
                                Estado = DESMONTA_CARTAO;
                                SDCardStatus = DESMONTANDO;
                        }

                        break;
                case ERRO_IRREPARAVEL_1:
                        P4OUT &= (~0x01);
                        OSTimeDly(2000);
                        P4OUT |= (0x01);
                        OSTimeDly(50);
                        P4OUT &= (~0x01);
                        OSTimeDly(50);
                        P4OUT |= (0x01);
                        OSTimeDly(50);
                        P4OUT &= (~0x01);

                        switch(SDCardComando){
                        case MONTAR:
                                Estado = MOUNT;
                                SDCardStatus = MONTANDO;
                                break;
                        case RESETAR:
                                Estado = RESET_SD;
                                SDCardStatus = RESETANDO;
                                break;
                        default:
                                Estado = ERRO_IRREPARAVEL;
                                SDCardStatus = ERRO_FATAL;
                                break;
                        }
```

```
                                        break;
                                case ERRO_IRREPARAVEL:
                                        CloseFileDos(Handle);
                                        Estado = ERRO_IRREPARAVEL_1;
                                        break;
                                case DESMONTA_CARTAO:
                                        CloseFileDos(0);
                                        Estado = BAIXA_TENSAO_ESTADO;
                                        SDCard*Status* = BAIXA_TENSAO;
                                        break;
                                default:
                                        break;
                                }

                        }
}

char PrintText(char *Dest, long int dado){
char *Buffer*[20];
int i = 0;
int j = 0;

        if (dado < 0){
                *(Dest + j) = '-';
                dado *= -1;
                j++;
        }

        while (dado >= 10){
                *Buffer*[i] = (dado % 10) + 0x30;
                dado /= (10);
                i++;
        }
        *Buffer*[i] = (dado % 10) + 0x30;

        for (i = i;i >= 0;i--){
                *(Dest + j) = *Buffer*[i];
                j++;
        }
        *(Dest + j) = 0;
        return(j);
}


void CaptureSingle(char Dado){

        while (((BUFFERSIZE - PonteiroRecData) < 1) || (SDCard*Status* != ESCREVENDO))
                OSTimeDly(1);

        _DINT();
        *(*Buffer*RecData + PonteiroRecData ) = Dado;
         PonteiroRecData++;
        _EINT();
}


void Capture(char *Dado,int Len){

unsigned int i;

        while (((BUFFERSIZE - PonteiroRecData - Len) < 1) || (SDCard*Status* != ESCREVENDO))
                OSTimeDly(10);
        _DINT();
        for (i = 0; i < Len;i++){
                *(*Buffer*RecData + PonteiroRecData ) = *(Dado + i);
```

```
                PonteiroRecData++;
        }
        _EINT();
}


void CaptureLongInt(int dado){

        char Buffer[11];
        int i = 0;

        if (dado < 0){
                CaptureSingle('-');
                dado *= -1;
        }

        while (dado >= 10){
                Buffer[i] = (dado % 10) + 0x30;
                dado /= (10);
                i++;
        }
        Buffer[i] = (dado % 10) + 0x30;

        for (i = i;i >= 0;i--){
                CaptureSingle(Buffer[i]);
        }
}


void Amostrador (void *pdata){
        pdata = pdata;


        while (1){
                OSTimeDly(50);
                //CaptureSingle((char)(DiaMes / 10) + 0x30);
                //CaptureSingle((char)(DiaMes % 10) + 0x30);
                //CaptureSingle('/');

                //CaptureSingle((char)(Mes / 10) + 0x30);
                //CaptureSingle((char)(Mes % 10) + 0x30);
                //CaptureSingle('/');

                //CaptureSingle((char)(Ano / 10) + 0x30);
                //CaptureSingle((char)(Ano % 10) + 0x30);
                //CaptureSingle(' ');

                /*CaptureSingle((char)(Horas / 10) + 0x30);
                CaptureSingle((char)(Horas % 10) + 0x30);
                CaptureSingle(':');

                CaptureSingle((char)(Minutos / 10) + 0x30);
                CaptureSingle((char)(Minutos % 10) + 0x30);
                CaptureSingle(':');

                CaptureSingle((char)(Segundos / 10) + 0x30);
                CaptureSingle((char)(Segundos % 10) + 0x30);
                CaptureSingle(';');*/

                CaptureLongInt(UdCAN);
                CaptureSingle(';');
                CaptureLongInt(UeCAN);
                CaptureSingle(';');
                CaptureLongInt((int)(VrdCAN));
                CaptureSingle(';');
```

```
                    CaptureLongInt((int)(VreCAN));
                    CaptureSingle(';');
                    CaptureLongInt((int)(SPVL));
                    CaptureSingle(';');
                    CaptureLongInt((int)(SPDV));


                    CaptureSingle(0x0D);
                    CaptureSingle(0x0A);

            }
}
```

ANEXO 13:ARQUIVO DISPLAY.C. ARQUIVO DO CÓDIGO C DA TAREFA DE CONTROLE E ATUALIZAÇÃO DO DISPLAY

```
#define E_LOW               P6OUT &= (~0x40)
#define E_HIGH              P6OUT |= (0x40)
#define RS_LOW                      P6OUT &= (~0x80)
#define RS_HIGH                     P6OUT |= (0x80)

#define E_DIR               P6DIR |= (0x40)
#define E_SEL               P6SEL &= (~0x40)

#define RS_DIR              P6DIR |= (0x80)
#define RS_SEL              P6SEL &= (~0x80)
#define DISP_PORT_DIR       P5DIR |= (0xFF)
#define DISP_PORT_SEL       P5SEL &= (~0xFF)
#define DISP_PORT                   P5OUT

extern long int             Vrd;
extern long int             Vre;

typedef union STATUS_CONTROLADOR{
        struct {
                char Run                                        :1;//0
                char SalvandoParametros                 :1;//1
                char AtulizandoParametros   :1;//2
                char ErroSalvamentoParametros       :1;//3
                char PerdaRTControlador                 :1;//4
                char AlertaDeColisao                    :1;//5
                char Driver*Run                                 :1;//6
                char EncEsqDes                          :1;//7
                char EncDirDes                          :1;//8
                char FaultMD                                    :1;//9
                char FaultME                                    :1;//10
                char PerdaSinc                                  :1;//11
                char PerdaRTDriverMotores           :1;//12
                char PerdaRTHodômetro                   :1;//13
        }Bit;
        unsigned int Byte;
}STATUS_CONTROLADOR;

STATUS_CONTROLADOR              Status*Controlador;
unsigned int                TensaoAlimentacao;

char                                ControleRelogio = 0;
char                                Segundos = 0;
char                                Minutos=0;
char                                Horas =0;
char                                DiaSemana = 0;
char                                DiaMes = 0;
char                                Mes = 0;
char                                Ano = 0;
char                                SDCard*Status*;


void *Clock*LCD(char dado){
        E_HIGH;
        OSTimeDly(dado);
        E_LOW;
        OSTimeDly(dado);
```

```
}


void LCD(char BufferLCD){

	DISP_PORT = BufferLCD;
	RS_HIGH;
	ClockLCD(1);

}


void SetDisplay(void){

	DISP_PORT = 0x3F; // 0b0011 1111;
	RS_LOW;
	ClockLCD(2);
}

void TunrOnDisplay(void){
	DISP_PORT = 0x0C; //0b0000 1100;
	RS_LOW;
	ClockLCD(2);
}
void ClearDisplay(void){
	DISP_PORT = 0x01; //0b00000001;
	RS_LOW;
	ClockLCD(2);
}
void EntryDisplay(void){
	DISP_PORT = 0x06; //0b00000110;
	RS_LOW;
	ClockLCD(2);
}

void HomeDisplay(void){
	DISP_PORT = 0x02; //0b00000010;
	RS_LOW;
	ClockLCD(200);
}



void GotoLCD(char linha, char coluna){

	switch (linha){
		case 1:
			DISP_PORT = coluna + 0x80;
			break;
		case 2:
			DISP_PORT = coluna + 0xC0;
			break;
		case 3:
			DISP_PORT = coluna + 0x94;
			break;
		case 4:
			DISP_PORT = coluna + 0xD4;
			break;
		default:
			break;
	}
	RS_LOW;
	ClockLCD(3);
```

```
}


char PrintfStrLCD (char *p){
char i = 0;

        while (*(p + i) != 0){
                LCD(*(p + i));
                i++;
        }
return(i);
}

char PrintfLongIntLCD(long int dado){
char Buffer[20];
int i = 0;
int Count = 0;
char Sinal = 0;

        if (dado < 0){
        Sinal = '-';
        dado *= -1;
        }

        while (dado >= 10){
                Buffer[i] = (dado % 10) + 0x30;
                dado /= (10);
                i++;
        }
        Buffer[i] = (dado % 10) + 0x30;
        dado = dado - (Buffer[i] - 0x30);

        if (Sinal == '-'){
                LCD(Sinal);
                Count++;
        }

        for (i = i;i >= 0;i--){
                LCD(Buffer[i]);
                Count++;
        }

        /*if (CasasDecimais != 0){
                LCD(',');
                Count++;
        }

        for (CasasDecimais = CasasDecimais;CasasDecimais != 0;CasasDecimais--){
                dado *= 10;
                Buffer[0] = (char)((unsigned long int)dado % 10);
                LCD(Buffer[0] + 0x30);
                dado -= Buffer[0];
                Count++;
        }
        */
        return(Count);
}



char PrintfFloatLCD(float dadofloat, int Casas){
long int dado;
char Buffer[20];
int i = 0;
int Count = 0;
```

```
char Sinal = 0;


	for (Count = Casas; Count > 0; Count --)
		dadofloat *= 10;

	dado = (long int)(dadofloat);

	if (dado < 0){
		Sinal = '-';
		dado *= -1;
	}

	while (dado >= 10){
		Buffer[i] = (dado % 10) + 0x30;
		dado /= (10);
		i++;
	}
	Buffer[i] = (dado % 10) + 0x30;
	dado = dado - (Buffer[i] - 0x30);

	if (Sinal == '-'){
		LCD(Sinal);
		Count++;
	}

	for (i = i;i >= 0;i--){
		LCD(Buffer[i]);
		Count++;
		if (Casas == i){
			LCD(',');
			Count++;
		}
	}

	/*
	for (CasasDecimais = CasasDecimais;CasasDecimais != 0;CasasDecimais--){
		dado *= 10;
		Buffer[0] = (char)((unsigned long int)dado % 10);
		LCD(Buffer[0] + 0x30);
		dado -= Buffer[0];
		Count++;
	}
	*/
	return(Count);
}


void Display(void *pdata){
	char	aux = 0;
	HANDLE_CAN_MSG		MSGTensaoAlimentacao;
	HANDLE_CAN_MSG			MSGSegundos;
	HANDLE_CAN_MSG			MSGStatusControlador;


	pdata = pdata;

	E_DIR;
	E_SEL;
	RS_DIR;
	RS_SEL;
	GotoLCD(1,0);
	DISP_PORT_DIR;
	DISP_PORT_SEL;
```

```
MSGTensaoAlimentacao      = InsertOnBus (0x0200,80,(void *)&TensaoAlimentacao,2,0);
MSGSegundos                         = InsertOnBus (0x02AA,100,(void *)&Segundos,8,0);
MSGStatusControlador      = InsertOnBus (0x0450,100,(void *)&StatusControlador.Byte,2,0);


TensaoAlimentacao = TensaoAlimentacao;
Segundos = Segundos;
Minutos = Minutos;
Horas = Horas;
DiaSemana = DiaSemana;
DiaMes = DiaMes;
Mes = Mes;
Ano = Ano;
SDCardStatus = SDCardStatus;

SetDisplay();
TunrOnDisplay();
ClearDisplay();
EntryDisplay();
HomeDisplay();

GotoLCD(1,0);
PrintfStrLCD("  LAI 2 UFES PPGEE  ");
GotoLCD(2,0);
PrintfStrLCD("  Cadeira de rodas  ");
GotoLCD(3,0);
PrintfStrLCD("Controlada p/ sinais");
GotoLCD(4,0);
PrintfStrLCD("      Cerebrais     ");


//printf(LCD, "     e sonhos      ");
OSTimeDly(4000);

ClearDisplay();
EntryDisplay();

GotoLCD(1,7);
while (1){

        GotoLCD(1,0);
        aux = PrintfStrLCD("Vrd:");
        aux += PrintfLongIntLCD(Vrd);
        aux += PrintfStrLCD("mm/s");
        for(aux=aux;aux < 14;aux++)
                LCD(' ');

        if (CANDataAvaliable(MSGSegundos)){
                if (Horas < 10)
                        PrintfStrLCD("0");
                PrintfLongIntLCD(Horas);
                PrintfStrLCD(":");
                if (Minutos < 10)
                        PrintfStrLCD("0");
                PrintfLongIntLCD(Minutos);
        }
        else{
                PrintfStrLCD("XXXXX");
        }
```

```
		GotoLCD(2,0);
		aux = PrintfStrLCD("Vre:");
		aux += PrintfLongIntLCD(Vre);
		aux += PrintfStrLCD("mm/s");
		for(aux=aux;aux < 14;aux++)
			LCD(' ');

		if (CANDataAvaliable(MSGSegundos)){
			aux += PrintfStrLCD("SD:");
			switch (SDCard*Status*){
			case 0:
				aux += PrintfStrLCD("Res");
				break;
			case 1:
				aux += PrintfStrLCD("BT");
				break;
case 2:
				aux += PrintfStrLCD("Esc");
				break;
case 3:
				aux += PrintfStrLCD("Ini");
				break;
			case 4:
				aux += PrintfStrLCD("Des");
				break;
			case 5:
				aux += PrintfStrLCD("Mon");
				break;
case 6:
				aux += PrintfStrLCD("Len");
				break;
			case 7:
				aux += PrintfStrLCD("ErF");
				break;
case 8:
				aux += PrintfStrLCD("Par");
				break;
case 9:
				aux += PrintfStrLCD("AbD");
				break;
case 10:
				aux += PrintfStrLCD("SDO");
				break;
case 11:
				aux += PrintfStrLCD("AbH");
				break;
case 12:
				aux += PrintfStrLCD("AbA");
				break;
case 13:
				aux += PrintfStrLCD("Dem");
				break;
			case 14:
				aux += PrintfStrLCD("NSD");
				break;
default:
				aux += PrintfStrLCD("XXX");
				break;
			}
		}
		else{
			aux += PrintfStrLCD("SD:");
			aux += PrintfStrLCD("XXX");
		}
```

```
                for(aux=aux;aux < 20;aux++)
                        LCD(' ');

                GotoLCD(3,0);
                aux = PrintfStrLCD("Tmd:");
                aux += PrintfFloatLCD(((float)(PWM.d))/10,1);;
                aux += PrintfStrLCD("%");
                for(aux=aux;aux < 11;aux++)
                        LCD(' ');

                aux += PrintfStrLCD("Sta:");

                if (CANDataAvaliable(MSGStatusControlador)){
                        switch(StatusControlador.Byte){
                        case 0x0041:
                                aux += PrintfStrLCD("On");
                                break;
                        case 0:
                                aux += PrintfStrLCD("Off");
                                break;
                        default:
                                aux += PrintfLongIntLCD(StatusControlador.Byte);
                                break;
                        }
                }
                else{
                        aux += PrintfStrLCD("Desc.");
                }
                for(aux=aux;aux < 20;aux++)
                        LCD(' ');

                GotoLCD(4,0);
                aux = PrintfStrLCD("Tme:");
                aux += PrintfFloatLCD(((float)(PWM.e))/10,1);
                aux += PrintfStrLCD("%");

                for(aux=aux;aux < 11;aux++)
                        LCD(' ');

                if (CANDataAvaliable(MSGTensaoAlimentacao)){
                        aux += PrintfStrLCD("Bat:");
                        aux += PrintfFloatLCD(((float)(TensaoAlimentacao))*0.0063,1);
                        aux += PrintfStrLCD("V");
                }
                else{
                        aux += PrintfStrLCD("Bat:");
                        aux += PrintfStrLCD("Desc.");
                }
                for(aux=aux;aux < 20;aux++)
                        LCD(' ');

                OSTimeDly(100);
        }

}
```

ANEXO 14:ARQUIVO DRIVERMOTORES.C. ARQUIVO DO CÓDIGO C DA TAREFA DRIVER MOTORES

```
#define PERIDO_DRIVER_MOTORES        5
typedef union MOTORES_COMANDO{
        struct{
                char LimpaFalha              :1;
                char LigaMotores   :1;
        }Bit;
        char Byte;
}MOTORES_COMANDO;

typedef union MOTORES_STATUS{
        struct {
                char Run             : 1;
                char EncEsqDes  : 1;
                char EncDirDes  : 1;
                char FaultMD         : 1;
                char FaultME         : 1;
                char PerdaSinc       : 1;
                char PerdaRT         : 1;
        }Bit;
        char Byte;
}MOTORES_STATUS;

MOTORES_COMANDO              MotoresComando;
MOTORES_STATUS               MotoresStatus;
OS_EVENT                  *SemDriverMotores;

struct PWM{
        int         e;
        int         d;
}PWM;

void IntDriverMotores(void){
static int          Timer = 0;

        if ((--Timer) <= 0){
                 Timer = PERIDO_DRIVER_MOTORES;
                 OSSemPost(SemDriverMotores);
        }
}


char EsperaTick (void){
        if ((OSSemAccept(SemDriverMotores)) >= 1){
                return 0;
        }
        else{
                OSSemPend(SemDriverMotores, 0, &err);
                return 1;
        }
}


void DriverMotores(void *pdata){
        enum Estado{
                COMANDO = 0,
                START,
                STOP,
```

```
		TESTE_ENCODERS,
		TESTE_FALHA_DOS_MOTORES,
		TESTE_VARIAVEIS_REDE,
		ESPERA_VARIAVEIS_DE_REDE,
		RUN,
		FALHA_IRREPARAVEL,
		LIMPA_FALHA_IRREPARAVEL,
}Estado;

int TimerEncEsqDes = 0;
int TimerEncDirDes = 0;
int CountErros = 0;

struct PWMAux{
		int		e;
		int		d;
}PWMAux;

HANDLE_CAN_MSG			MSGPWMData;
HANDLE_CAN_MSG			MSGComandoMotores;

pdata = pdata;

TACTL = TASSEL_2 + ID_0 + MC_1;
TACCTL0 =0X00;
TACCTL1 = OUTMOD_3;
TACCTL2 = OUTMOD_7;
TACCR0 = 400;
TACCR1 = 200;
TACCR2 = 200;

P1DIR = 0xCD;
P1SEL = 0x0C;

P8DIR = 0x3F;
P8OUT = 0;



MSGPWMData				= InsertOnBus (0x0110,4,(void *)&PWM,4,0);
MSGComandoMotores	= InsertOnBus (0x0109,13,(void *)&MotoresComando.Byte,1,0);
InsertOnBus (0x0115,0,(void *)&MotoresStatus.Byte,1,0);



OSTimeDly(2000);

SemDriverMotores =  OSSemCreate(0);

TimerEncDirDes = 0;
TimerEncEsqDes = 0;
MotoresStatus.Bit.EncDirDes = 0;
MotoresStatus.Bit.EncEsqDes = 0;
MotoresStatus.Bit.PerdaSinc = 0;
MotoresStatus.Bit.FaultME = 0;
MotoresStatus.Bit.FaultMD = 0;
CountErros = 0;
Estado = COMANDO;

while(1){
		switch(Estado){
		case START:
			TACCR1 = 200;
			TACCR2 = 200;
			P8OUT = 0x12;
```

```
		P1OUT |= 0xC0;
		OSTimeDly(400);
		P8OUT = 0x36;
		OSTimeDly(400);
		P1OUT &= (~0xC0);
		OSTimeDly(10);
		P8OUT = 0x3F;
		OSTimeDly(50);
		P8OUT = 0x36;
		OSSemSet(SemDriverMotores, 0, &err);
		Espera*Tick*();
		OSSemSet(SemDriverMotores, 0, &err);
		Motores*Status*.Bit.Run = 1;

		if (P1IN & 0x10){
			Motores*Status*.Bit.FaultMD = 0;
		}
		else{
			Motores*Status*.Bit.FaultMD = 1;
		}

		if (P1IN & 0x20){
			Motores*Status*.Bit.FaultME = 0;
		}
		else{
			Motores*Status*.Bit.FaultME = 1;
		}

		if ((Motores*Status*.Bit.FaultME) || (Motores*Status*.Bit.FaultMD)){
			P8OUT = 0;
			Motores*Status*.Bit.Run = 0;
			OSTimeDly(100);
			Estado = FALHA_IRREPARAVEL;
		}
		else{
			Estado = COMANDO;
		}
		break;
case COMANDO:
		if (Espera*Tick*()){
			if (MotoresComando.Bit.LigaMotores){
				if (Motores*Status*.Bit.Run){
					Estado = TESTE_ENCODERS;
				}
				else{
					Estado = START;
				}
			}
			else{
				Estado = STOP;
			}
		}
		else{
			Estado = FALHA_IRREPARAVEL;
			Motores*Status*.Bit.Run = 0;
			Motores*Status*.Bit.PerdaRT = 1;
		}
		break;
case TESTE_ENCODERS:
		if ((Vre == 0) && ((PWM.e >= 300)|| (PWM.e <= -300))){
			*Timer*EncEsqDes ++;
		}
		else{
			*Timer*EncEsqDes = 0;
		}
```

```
                if ((Vrd == 0) && ((PWM.d >= 300)|| (PWM.d <= -300))){
                        TimerEncDirDes ++;
                }
                else{
                        TimerEncDirDes = 0;
                }
                if (TimerEncEsqDes >= 200)
                        MotoresStatus.Bit.EncEsqDes = 1;

                if (TimerEncDirDes >= 200)
                        MotoresStatus.Bit.EncDirDes = 1;

                if ((MotoresStatus.Bit.EncDirDes) || (MotoresStatus.Bit.EncDirDes)){
                        Estado = FALHA_IRREPARAVEL;
                }
                else{
                        Estado =  TESTE_FALHA_DOS_MOTORES;
                }

                break;
        case TESTE_FALHA_DOS_MOTORES:
                if (P1IN & 0x10){
                        MotoresStatus.Bit.FaultMD = 0;
                }
                else{
                        MotoresStatus.Bit.FaultMD = 1;
                }

                if (P1IN & 0x20){
                        MotoresStatus.Bit.FaultME = 0;
                }
                else{
                        MotoresStatus.Bit.FaultME = 1;
                }

                if ((MotoresStatus.Bit.FaultME) || (MotoresStatus.Bit.FaultMD)){
                        P8OUT = 0;
                        MotoresStatus.Bit.Run = 0;
                        OSTimeDly(2000);
                        Estado = START;
                }
                else{
                        Estado = TESTE_VARIAVEIS_REDE;
                }

                break;

        case TESTE_VARIAVEIS_REDE:
                if                      ((CANDataAvaliable(MSGPWMData))                 &&
(CANDataAvaliable(MSGComandoMotores))){
                        Estado = RUN;
                        CountErros = 0;
                }
                else{
                        CountErros ++;
                        if (CountErros >= 6){
                                P8OUT = 0;
                                TACCR1 = 200;
                                TACCR2 = 200;
                                MotoresStatus.Bit.Run = 0;
                                MotoresStatus.Bit.PerdaSinc = 1;
                                Estado = ESPERA_VARIAVEIS_DE_REDE;
                        }
                        else{
```

```
                                        Estado = RUN;
                                }
                        }
                        break;

                case RUN:

                        CANMSGSemPend(MSGPWMData);
                        PWMAux.d = PWM.d;
                        PWMAux.e = PWM.e;
                        CANMSGSemPost(MSGPWMData);
                        TACCR1 = (PWMAux.d  / 5) + 200;
                        TACCR2 = (PWMAux.e  / 5) + 200;
                        Estado = COMANDO;
                        break;
                case STOP:

                        P8OUT = 0;
                        TACCR1 = 200;
                        TACCR2 = 200;
                        Motores Status.Bit.Run = 0;
                        Estado = COMANDO;
                        break;
                case ESPERA_VARIAVEIS_DE_REDE:
                        if                      ((CANDataAvaliable(MSGPWMData))                 &&
(CANDataAvaliable(MSGComandoMotores))){
                                Estado = COMANDO;
                                Motores Status.Bit.PerdaSinc = 0;
                        }
                        else{
                                if (Espera Tick() == 0){
                                        Estado = FALHA_IRREPARAVEL;
                                        Motores Status.Bit.Run = 0;
                                        Motores Status.Bit.PerdaRT = 1;
                                }
                        }
                        break;
                case FALHA_IRREPARAVEL:
                        P8OUT = 0;
                        TACCR1 = 200;
                        TACCR2 = 200;
                        Motores Status.Bit.Run = 0;
                        Espera Tick();
                        if (MotoresComando.Bit.LimpaFalha){
                                Estado = LIMPA_FALHA_IRREPARAVEL;
                        }
                        break;
                case LIMPA_FALHA_IRREPARAVEL:
                        Espera Tick();
                        if (MotoresComando.Bit.LimpaFalha == 0){
                                Motores Status.Byte = 0;
                                Estado = START;
                        }

                        break;

                default:
                        Estado = FALHA_IRREPARAVEL;
                        break;
                }
        }

}
```

ANEXO 15:ARQUIVO RELOGIO_CONTROL_TASK.C. ARQUIVO DO CÓDIGO C DA TAREFA RELÓGIO

```
//////////////////////////////////////////////// relogio Control Task
//////////////////////////////////////////////////////////////////////////////////////////
#define RESET_RELOGIO ControleRelogio |= 0x01
#define TESTE_RESET_RELOGIO ControleRelogio & 0x01



void ReadI2C(char Address, char Num*Bytes*);
void TransmiteI2C (char Dado,char Controle);
char Int2BCD(char Dado);
OS_EVENT      *SemI2C;
OS_EVENT      *SemRelogio;
char          MsgRelogio;
char          ControleRelogio;



// Variáveis Globais I2C
char          *Buffer*I2C[10];
char          PonteiroI2C*Buffer* = 0;
char          PonteiroI2C=0;

/*
*********************************************************************************************************
**************
                                    Relogio Task
*********************************************************************************************************
**************
*/
void Relogio(void *pdata){
enum MAQUINA_DE_ESTADO{
   INICIALIZAR_I2C_PORT,
   CONFIGURAR_RTC,
   WAIT_FOR_1HZ_CLK,
   READ_RTC,
} Estado;

Estado = INICIALIZAR_I2C_PORT;
char P11Anterior = 0;
char Aux= 0 ;

pdata  = pdata;

SemI2C = OSSemCreate(0);
SemRelogio = OSSemCreate(1);

 while(1){
   OSTimeDly(100);
   switch(Estado){
   case INICIALIZAR_I2C_PORT:
     //UCA0CTL1 = UCSWRST;
     UCB0CTL1 = UCSWRST;
     UCB0CTL0 = UCMST + UCMODE_3 + UCSYNC;
     UCB0CTL1 = UCB0CTL1 + UCSSEL_2;
     UCB0BR0 = 0x50;
     UCB0BR1 = 0x00;
     UCB0I2COA = 100;
```

```
    P3DIR |= 0x06;
    P3SEL |= 0x06;
    P1DIR &= 0xFE;
    P1SEL &= 0xFE;
    UCB0CTL1 &= (~UCSWRST);
    IE2 +=  (UCB0TXIE + UCB0RXIE);
    Estado = READ_RTC;
    break;
   case CONFIGURAR_RTC:
    OSSemPend(SemRelogio, 0, &err);
    TransmiteI2C(0x00,'R');
    TransmiteI2C(0x00,'B');              // address
    TransmiteI2C(Int2BCD(Segundos),'B');  // secunds
    TransmiteI2C(Int2BCD(Minutos),'B');   //minutes
    TransmiteI2C(Int2BCD(Horas),'B');     //hours
    TransmiteI2C(Int2BCD(DiaSemana),'B'); //week day
    TransmiteI2C(Int2BCD(DiaMes),'B');    //month day
    TransmiteI2C(Int2BCD(Mes),'B');       //month
    TransmiteI2C(Int2BCD(Ano),'B');       //year
    OSSemPost(SemRelogio);
    TransmiteI2C(0x90,'B');              //config
    TransmiteI2C(0x00,'T');
    Estado = WAIT_FOR_1HZ_CLK;
    break;
   case WAIT_FOR_1HZ_CLK:
    if ((TESTE_RESET_RELOGIO) != 0){
     Estado=CONFIGURAR_RTC;
     ControleRelogio &= 0xFE;
    }
    else{
     Aux = P1IN;
     if ((P11Anterior == 0) && ((Aux & 0x01)!=0)){
      P11Anterior = Aux & 0x01;
      if(Segundos >= 59){
       Estado=READ_RTC;
      }
      else{
       Segundos++;
      }
     }
     else{
      P11Anterior = Aux & 0x01;
     }
    }
        OSTimeDly(100);
    break;
   case READ_RTC:
    ReadI2C(0x00,8);
    OSSemPend(SemRelogio, 0, &err);
    Segundos =  (((BufferI2C[0] & 0xF0)>>4) * 10) + (BufferI2C[0] & 0x0F);
    Minutos =   (((BufferI2C[1] & 0xF0)>>4) * 10) + (BufferI2C[1] & 0x0F);
    Horas =     (((BufferI2C[2] & 0xF0)>>4) * 10) + (BufferI2C[2] & 0x0F);
    DiaSemana = (((BufferI2C[3] & 0xF0)>>4) * 10) + (BufferI2C[3] & 0x0F);
    DiaMes =    (((BufferI2C[4] & 0xF0)>>4) * 10) + (BufferI2C[4] & 0x0F);
    Mes =       (((BufferI2C[5] & 0xF0)>>4) * 10) + (BufferI2C[5] & 0x0F);
    Ano =       (((BufferI2C[6] & 0xF0)>>4) * 10) + (BufferI2C[6] & 0x0F);
    OSSemPost(SemRelogio);
    Estado = WAIT_FOR_1HZ_CLK;
    break;
   default:
    break;
   }
  }
}
void ReadI2C(char Address, char NumBytes){
```

```
  TransmiteI2C(0x00,'R');
  TransmiteI2C(Address,'B');
  TransmiteI2C(0x00,'T');
  PonteiroI2C_Buffer = Num_Bytes;
  PonteiroI2C = 0;
  UCB0I2CSA = 0x68;
  UCB0CTL1 &= (~UCTR);
  UCB0CTL1 |= UCTXSTT;
  OSSemPend(SemI2C, 0, &err);
}

void TransmiteI2C (char Dado,char Controle){
  UCB0I2CSA = 0x68;
  switch(Controle){
  case 'B':
    _BufferI2C[PonteiroI2C_Buffer] = Dado;
    PonteiroI2C_Buffer++;
    break;
  case 'R':
    PonteiroI2C_Buffer = 0;
    PonteiroI2C = 0;
    break;
  case 'T':
    UCB0CTL1 |= UCTR;
    UCB0CTL1 |= UCTXSTT;
    OSSemPend(SemI2C, 0, &err);
    UCB0CTL1 |= UCTR;
    UCB0CTL1 |= UCTXSTP;
    OSTimeDly(1);
    PonteiroI2C_Buffer = 0;
    PonteiroI2C = 0;
  default:
    break;
  }
}

char Int2BCD(char Dado){
char aux1=0;
char aux2=0;
  aux1 =(char)Dado/10;
  aux1 <<= 4;
  aux2 =(char)Dado%10;
  return(aux1 | aux2);
}
```

ANEXO 16:ARQUIVO UART_CONTROL_TASK.C. ARQUIVO DO CÓDIGO C DA TAREFA UART

```
char ScanUART(void);
void PrintUART(char);
void PrintLongInt(long int);

#define LEN_BUFFER_TX            128
#define LEN_BUFFER_RX            128
#define MASCARA_LEN_BUFFER_TX LEN_BUFFER_TX - 1
#define MASCARA_LEN_BUFFER_RX LEN_BUFFER_RX - 1

char                    PonteiroSetPointTXUART = 0;
char                    PonteiroTXUART = 0;
char                    PonteiroRXUART = 0;
char                    BufferTXUART[LEN_BUFFER_TX];
char                    BufferRXUART[LEN_BUFFER_RX];
OS_EVENT        *SemRXUART;

////////////////////////////////////////////////////////////////////////////////
//Variaveis Globais do Controlador

void UART(void *pdata){
        const enum MAQUINA_DE_ESTADO{
                                IDLE,
        };


        char Estado = 'h';
        char aux = 0;
        long int Aux2 = 0;
        char Sinal;


        pdata  = pdata;


        /* CONFIGURAÇÃO DA UART
        57600 bps
        SEM PARIDADE
        1 STOP BIT
        1 START BIT
        BIT MENOS SGNIFICATIVO PRIMEIRO
        */

        UCA1CTL1 = UCSWRST;
        UCA1CTL0 = 0;
        UCA1CTL1 |= UCSSEL_2;
        // 57600 bps
        UCA1BR0 = 0x08;
        UCA1BR1 = 0x00;
        UCA1MCTL = 0xB1;

        //// 19200 kbps
        //UCA1BR0 = 26;
        //UCA1BR1 = 0x00;
        //UCA1MCTL = 0x11;

        P3DIR |= 0x40;
```

```
P3SEL |= 0xC0;
UCA1CTL1 &= (~UCSWRST);
UC1IE |= UCA1TXIE;
UC1IE |= UCA1RXIE;



////////////////////////////////////////////////////////////////////////////


InsertOnBus (0x0200,0,(void *)&TensaoAlimentacao,2,0);
MsgRelogio          = InsertOnBus (0x02AA,0,(void *)&Segundos,8,0);

////////////////////////////////////////////////////////////////////////////

SemRXUART = OSSemCreate(0);

while(1){
        switch(Estado){
        case IDLE:
                Estado = ScanUART();
                break;
        case 'D':
                switch(ScanUART()){
                case 'a':
                        aux = ScanUART();
                        PrintUART('D');
                        PrintUART('a');
                        PrintLongInt(StatusControlador.Byte);
                        PrintUART(':');
                        PrintLongInt(Vre);
                        PrintUART(':');
                        PrintLongInt(Vrd);
                        PrintUART(':');
                        PrintLongInt(Ue);
                        PrintUART(':');
                        PrintLongInt(Ud);
                        PrintUART('@');
                        break;
                case 'b':
                        aux = ScanUART();
                        Aux2 = 0;
                        Sinal = '+';
                        while (aux != '@'){
                                if (aux == '-'){
                                        Sinal = '-';
                                }
                                else{
                                        Aux2 = Aux2 * 10 + (aux - 0x30);
                                }
                                aux = ScanUART();
                        }
                        if (Sinal == '-')
                                Aux2 *= -1;
                        SPVL = Aux2;
                        break;
                case 'c':
                        aux = ScanUART();
                        Aux2 = 0;
                        Sinal = '+';
                        while (aux != '@'){
                                if (aux == '-'){
                                        Sinal = '-';
                                }
                                else{
```

```
                        Aux2 = Aux2 * 10 + (aux - 0x30);
                }
                aux = ScanUART();
        }
        if (Sinal == '-')
                Aux2 *= -1;
        SPDV = Aux2;
        break;
case 'd':
        aux = ScanUART();
        Aux2 = 0;
        Sinal = '+';
        while (aux != '@'){
                if (aux == '-'){
                        Sinal = '-';
                }
                else{
                        Aux2 = Aux2 * 10 + (aux - 0x30);
                }
                aux = ScanUART();
        }
        if (Sinal == '-')
                Aux2 *= -1;
        Kp = Aux2;
        break;
case 'e':
        aux = ScanUART();
        Aux2 = 0;
        Sinal = '+';
        while (aux != '@'){
                if (aux == '-'){
                        Sinal = '-';
                }
                else{
                        Aux2 = Aux2 * 10 + (aux - 0x30);
                }
                aux = ScanUART();
        }
        if (Sinal == '-')
                Aux2 *= -1;
        Kid = Aux2;
        break;
case 'f':
        aux = ScanUART();
        Aux2 = 0;
        Sinal = '+';
        while (aux != '@'){
                if (aux == '-'){
                        Sinal = '-';
                }
                else{
                        Aux2 = Aux2 * 10 + (aux - 0x30);
                }
                aux = ScanUART();
        }
        if (Sinal == '-')
                Aux2 *= -1;
        Ki = Aux2;
        break;
case 'g':
        aux = ScanUART();
        PrintUART('D');
        PrintUART('g');
        PrintLongInt(SPVL);
        PrintUART(':');
```

```
                PrintLongInt(SPDV);
                PrintUART(':');
                PrintLongInt(Kp);
                PrintUART(':');
                PrintLongInt(Kid);
                PrintUART(':');
                PrintLongInt(Ki);
                PrintUART('@');
                break;
        case 'h':
                aux = ScanUART();
                while (aux != '@'){
                        ComandoControlador = aux;
                        aux = ScanUART();
                }
                break;
        case 'i':
                aux = ScanUART();
                PrintUART('D');
                PrintUART('i');
                PrintLongInt(IR.Distancia0);
                PrintUART(':');
                PrintLongInt(IR.Distancia1);
                PrintUART(':');
                PrintLongInt(IR.Distancia2);
                PrintUART(':');
                PrintLongInt(IR.Distancia3);
                PrintUART('@');
                break;
        default:
                break;
        }
        Estado = IDLE;
        break;
case 'C':
        switch(ScanUART()){
        case '1':
                SDCardComando = ScanUART();
                aux = ScanUART();
                break;
        case '2':
                PrintUART('C');
                PrintLongInt(SDCard Status);
                PrintUART(':');
                PrintLongInt((Max Buffer 1 * 100) /(BUFFERSIZE));
                PrintUART(':');
                PrintLongInt((Max Buffer 2 * 100) /(BUFFERSIZE));
                PrintUART(':');
                PrintLongInt((LastLen Buffer 1 * 100) /(BUFFERSIZE));
                PrintUART(':');
                PrintLongInt((LastLen Buffer 2 * 100) /(BUFFERSIZE));
                PrintUART(':');
                PrintLongInt(SDResponse);
                PrintUART('@');
                aux = ScanUART();
                break;
        default:
                aux = ScanUART();
                break;
        }
        Estado = IDLE;
        break;
case 'h':
// ENVIA A DATA
```

```
                OSSemPend(SemRelogio, 0, &err);
                PrintUART('h');
                PrintUART((Horas/10) + 0x30);
                PrintUART((Horas%10) + 0x30);
                PrintUART(':');
                PrintUART((Minutos/10) + 0x30);
                PrintUART((Minutos%10) + 0x30);
                PrintUART(':');
                PrintUART((Segundos/10) + 0x30);
                PrintUART((Segundos%10) + 0x30);
                PrintUART(' ');
                PrintUART((DiaSemana/10) + 0x30);
                PrintUART((DiaSemana%10) + 0x30);
                PrintUART(' ');
                PrintUART((DiaMes/10) + 0x30);
                PrintUART((DiaMes%10) + 0x30);
                PrintUART('/');
                PrintUART((Mes/10) + 0x30);
                PrintUART((Mes%10) + 0x30);
                PrintUART('/');
                PrintUART((Ano/10) + 0x30);
                PrintUART((Ano%10) + 0x30);
                PrintUART('@');
                OSSemPost(SemRelogio);
                Estado = IDLE;

                break;
case 'H':
                //CONFIGURA A DATA
                OSSemPend(SemRelogio, 0, &err);
                aux = ScanUART() - 0x30;
                Horas = aux * 10 + (ScanUART() - 0x30);
                aux = ScanUART() - 0x30;
                aux = ScanUART() - 0x30;
                Minutos = aux * 10 + (ScanUART() - 0x30);
                aux = ScanUART() - 0x30;
                aux = ScanUART() - 0x30;
                Segundos = aux * 10 + (ScanUART() - 0x30);
                aux = ScanUART() - 0x30;
                DiaSemana = (ScanUART() - 0x30);
                aux = ScanUART() - 0x30;
                aux = ScanUART() - 0x30;
                DiaMes = aux * 10 + (ScanUART() - 0x30);
                aux = ScanUART() - 0x30;
                aux = ScanUART() - 0x30;
                Mes = aux * 10 + (ScanUART() - 0x30);
                aux = ScanUART() - 0x30;
                aux = ScanUART() - 0x30;
                Ano = aux * 10 + (ScanUART() - 0x30);
                RESET_RELOGIO;
                OSSemPost(SemRelogio);
                Estado = IDLE;
                aux = ScanUART();
                break;
case 'P':
                PrintUART('P');
                PrintLongInt(MCP2515.TEC);
                PrintUART(':');
                PrintLongInt(MCP2515.REC);
                PrintUART(':');
                PrintLongInt(MCP2515.Erro.Byte);
                PrintUART(':');
                PrintLongInt(IR.Distancia0);
                PrintUART(':');
                PrintLongInt(IR.Distancia1);
```

```
                                PrintUART(':');
                                PrintLongInt(IR.Distancia2);
                                PrintUART(':');
                                PrintLongInt(IR.Distancia3);
                                PrintUART(':');
                                PrintLongInt(StatusControlador.Byte);
                                PrintUART(':');
                                PrintLongInt(VreCAN);
                                PrintUART(':');
                                PrintLongInt(VrdCAN);
                                PrintUART(':');
                                PrintLongInt(UeCAN);
                                PrintUART(':');
                                PrintLongInt(UdCAN);
                                PrintUART(':');
                                PrintLongInt(SDCardStatus);
                                PrintUART(':');
                                PrintLongInt(TensaoAlimentacao);
                                PrintUART('@');
                                aux = ScanUART();
                                Estado = IDLE;
                                break;
                        case 'T':
                                PrintUART('T');
                                PrintUART('@');
                                Estado = IDLE;
                                break;
                        default:
                                Estado = IDLE;
                                break;
                        }
                }
}




void PrintUART (char Dado){
  _DINT();
  if (PonteiroTXUART != 0xFF){
    BufferTXUART[PonteiroSetPointTXUART] = Dado;
    PonteiroSetPointTXUART ++;
    PonteiroSetPointTXUART &= MASCARA_LEN_BUFFER_TX;
  }
  else{
    PonteiroTXUART = PonteiroSetPointTXUART;
    UCA1TXBUF = Dado;
  }
  _EINT();
}

char ScanUART(void){
  static char Ponteiro = 0;
  char aux;
  OSSemPend(SemRXUART, 0, &err);
  _DINT();
  aux = BufferRXUART[Ponteiro];
  Ponteiro ++;
  Ponteiro &= MASCARA_LEN_BUFFER_RX;
  _EINT();
  return(aux);
}

void PrintLongInt(long int dado){
```

```
	char Buffer[20];
	int i = 0;
	int j = 0;

	if (dado < 0){
		PrintUART('-');
		dado *= -1;
		j++;
	}

	while (dado >= 10){
		Buffer[i] = (dado % 10) + 0x30;
		dado /= (10);
		i++;
	}
	Buffer[i] = (dado % 10) + 0x30;

	for (i = i;i >= 0;i--){
		PrintUART(Buffer[i]);
		j++;
	}
}
```

ANEXO 17:ARQUIVO OS_CPU_A.S43. ARQUIVO DO PORT ASSEMBLY PARA O MSP430 FAMILIA 2

```
;*********************************************************************************************************
;                                              uC/OS-II
;                                        The Real-Time Kernel
;
;                    (c) Copyright 2008, Micrium, Inc., Weston, FL
;                                 All Rights Reserved
;
;                                          TI MSP430X
;                                          MSP430x5xx
;
;
; File      : OS_CPU_A.S43
; By        : Jian Chen (yenger@hotmail.com)
;             Jean J. Labrosse
;*********************************************************************************************************

#include   <msp430x26x.h>

;*********************************************************************************************************
;                                       PUBLIC AND EXTERNAL DECLARATIONS
;*********************************************************************************************************



        EXTERN  OSIntExit
        EXTERN  OSIntNesting

        EXTERN  OSISRStkPtr

        EXTERN  OSPrioCur
        EXTERN  OSPrioHighRdy

        EXTERN  OSRunning

        EXTERN  OSTCBCur
        EXTERN  OSTCBHighRdy

        EXTERN  OSTaskSwHook
        EXTERN  OSTimeTick

        PUBLIC  OSCtxSw
        PUBLIC  OSCPURestoreSR
        PUBLIC  OSCPUSaveSR
        PUBLIC  OSIntCtxSw
        PUBLIC  OSStartHighRdy
        PUBLIC  WDT_ISR

;*********************************************************************************************************
;                                       START HIGHEST PRIORITY READY TASK
;
; Description: This function is called by OSStart() to start the highest priority task that is ready to run.
;
```

```
            ; Note      : OSStartHighRdy() MUST:
            ;              a) Call OSTaskSwHook() then,
            ;              b) Set OSRunning to TRUE,
            ;              c) Switch to the highest priority task.
            ;*********************************************************************************************
**************

              RSEG      CODE                ; Program code

            OSStartHighRdy
              CALLA     #OSTaskSwHook

              MOV.B     #1, &OSRunning      ; kernel running

              MOVX.A    SP, &OSISRStkPtr    ; save interrupt stack

              MOVX.A    &OSTCBHighRdy, R13  ; load highest ready task stack
              MOVX.A    @R13, SP

              POPM.A    #12, R15

              RETI                          ; emulate return from interrupt


            ;*********************************************************************************************
**************
            ;                               TASK LEVEL CONTEXT SWITCH
            ;
            ; Description: This function is called by OS_Sched() to perform a task level context switch.
            ;
            ; Note      : OSCtxSw() MUST:
            ;              a) Save the current task's registers onto the current task stack
            ;              b) Save the SP into the current task's OS_TCB
            ;              c) Call OSTaskSwHook()
            ;              d) Copy OSPrioHighRdy to OSPrioCur
            ;              e) Copy OSTCBHighRdy to OSTCBCur
            ;              f) Load the SP with OSTCBHighRdy->OSTCBStkPtr
            ;              g) Restore all the registers from the high priority task stack
            ;              h) Perform a return from interrupt
            ;*********************************************************************************************
**************

            OSCtxSw
              POP.W     R12               ; Pop lower 16 bits of PC.
              POP.W     R13               ; Pop upper  4 bits of PC.

              PUSH.W    R12               ; Save lower 16 bits of PC.

              RLAM.A    #4, R13           ; Save SR + upper 4 bits of PC.
              RLAM.A    #4, R13
              RLAM.A    #4, R13
              MOVX.W    SR,  R12
              ADDX.A    R13, R12
              PUSH.W    R12

              PUSHM.A   #12, R15          ; Save R4-R15.

              MOVX.A    &OSTCBCur, R13    ; OSTCBCur->OSTCBStkPtr = SP
              MOVX.A    SP, 0(R13)

              CALLA     #OSTaskSwHook

              MOV.B     &OSPrioHighRdy, R13   ; OSPrioCur = OSPrioHighRdy
              MOV.B     R13, &OSPrioCur
```

```
    MOVX.A    &OSTCBHighRdy, R13    ; OSTCBCur  = OSTCBHighRdy
    MOVX.A    R13, &OSTCBCur

    MOVX.A    @R13, SP              ; SP       = OSTCBHighRdy->OSTCBStkPtr

    POPM.A    #12, R15              ; Restore R4-R15.

    RETI                            ; Return from interrupt.


;*********************************************************************************************************
;                                    ISR LEVEL CONTEXT SWITCH
;
; Description: This function is called by OSIntExit() to perform an ISR level context switch.
;
; Note      : OSIntCtxSw() MUST:
;             a) Call OSTaskSwHook()
;             b) Copy OSPrioHighRdy to OSPrioCur
;             c) Copy OSTCBHighRdy to OSTCBCur
;             d) Load the SP with OSTCBHighRdy->OSTCBStkPtr
;             e) Restore all the registers from the high priority task stack
;             f) Perform a return from interrupt
;*********************************************************************************************************

OSIntCtxSw
    CALLA     #OSTaskSwHook

    MOV.B     &OSPrioHighRdy, R13   ; OSPrioCur = OSPrioHighRdy
    MOV.B     R13, &OSPrioCur

    MOVX.A    &OSTCBHighRdy, R13    ; OSTCBCur  = OSTCBHighRdy
    MOVX.A    R13, &OSTCBCur

    MOVX.A    @R13, SP              ; SP       = OSTCBHighRdy->OSTCBStkPtr

    POPM.A    #12, R15

    RETI                            ; return from interrup


;*********************************************************************************************************
;                                        TICK ISR
;
; Description: This ISR handles tick interrupts.  This ISR uses the Watchdog timer as the tick source.
;
; Notes     : 1) The following C pseudo-code describes the operations being performed in the code below.
;
;             Save all the CPU registers
;             if (OSIntNesting == 0) {
;                OSTCBCur->OSTCBStkPtr = SP;
;                SP                    = OSISRStkPtr;  /* Use the ISR stack from now on          */
;             }
;             OSIntNesting++;
;             Enable interrupt nesting;                 /* Allow nesting of interrupts (if needed) */
;             Clear the interrupt source;
;             OSTimeTick();                             /* Call uC/OS-II's tick handler           */
;             DISABLE general interrupts;               /* Must DI before calling OSIntExit()     */
;             OSIntExit();
;             if (OSIntNesting == 0) {
;                SP = OSTCBHighRdy->OSTCBStkPtr;        /* Restore the current task's stack       */
;             }
;             Restore the CPU registers
```

```
;              Return from interrupt.
;
;           2) ALL ISRs should be written like this!
;
;           3) You MUST disable general interrupts BEFORE you call OSIntExit() because an interrupt
;              COULD occur just as OSIntExit() returns and thus, the new ISR would save the SP of
;              the ISR stack and NOT the SP of the task stack.  This of course will most likely cause
;              the code to crash.  By disabling interrupts BEFORE OSIntExit(), interrupts would be
;              disabled when OSIntExit() would return.  This assumes that you are using OS_CRITICAL_METHOD
;              #3 (which is the prefered method).
;
;           4) If you DON'T use a separate ISR stack then you don't need to disable general interrupts
;              just before calling OSIntExit().  The pseudo-code for an ISR would thus look like this:
;
;              Save all the CPU registers
;              if (OSIntNesting == 0) {
;                 OSTCBCur->OSTCBStkPtr = SP;
;              }
;              OSIntNesting++;
;              Enable interrupt nesting;              /* Allow nesting of interrupts (if needed) */
;              Clear the interrupt source;
;              OSTime*Tick*();                        /* Call uC/OS-II's *tick* handler          */
;              OSIntExit();
;              Restore the CPU registers
;              Return from interrupt.
;*********************************************************************************************
**************

WDT_ISR                         ; wd timer ISR
  PUSHM.A    #12, R15

  BIC.B      #0x01, IE1         ; disable wd timer interrupt

  CMP.B      #0, &OSIntNesting      ; if (OSIntNesting == 0)
  JNE        WDT_ISR_1

  MOVX.A     &OSTCBCur, R13         ;    save task stack
  MOVX.A     SP, 0(R13)

  MOVX.A     &OSISRStkPtr, SP       ;    *load* interrupt stack

WDT_ISR_1
  INC.B      &OSIntNesting          ; increase OSIntNesting

  BIS.B      #0x01, IE1         ; enable wd timer interrupt

  EINT                          ; enable general interrupt to allow for interrupt nesting

  CALLA      #OSTime*Tick*          ; call *ticks* routine

  DINT                          ; IMPORTANT: disable general interrupt BEFORE calling OSIntExit()

  CALLA      #OSIntExit

  CMP.B      #0, &OSIntNesting      ; if (OSIntNesting == 0)
  JNE        WDT_ISR_2

  MOVX.A     &OSTCBHighRdy, R13     ;    restore task stack SP
  MOVX.A     @R13, SP

WDT_ISR_2
  POPM.A     #12, R15

  RETI                          ; return from interrupt
```

```
            ;*********************************************************************************************
***************
            ;                       SAVE AND RESTORE THE CPU'S STATUS REGISTER
            ;
            ; Description: These functions are used to implement OS_CRITICAL_METHOD #3 by saving the status register
            ;            in a local variable of the calling function and then, disables interrupts.
            ;
            ; Notes      : R12 is assumed to hold the argument passed to OSCPUSaveSR() and also, the value returned
            ;            by OSCPURestoreSR().
            ;*********************************************************************************************
***************

            OSCPUSaveSR
              MOV.W      SR, R12
              DINT
              RETA


            OSCPURestoreSR
              MOV.W      R12, SR
              RETA

            ;*********************************************************************************************
***************
            ;                           WD TIMER INTERRUPT VECTOR ENTRY
            ;
            ; Interrupt vectors
            ;*********************************************************************************************
***************
              COMMON     INTVEC

              ORG        WDT_VECTOR
            WDT_VEC   DW    WDT_ISR              ; interrupt vector. Watchdog/Timer, Timer mode

              END
```

ANEXO 18:ARQUIVO CONTROLADOR.C. ARQUIVO C DA TAREFA CONTROLADOR

```
#define LIMITEINTEGRAL                          6000000
#define LIMITE_SAT_PWM                  950
#define PERIODO_CONTROLADOR             5
#define LIMITE_SPVL                             700
#define LIMITE_SPDV                             600

#define Kp_ADDRESS                              0x01000
#define Kid_ADDRESS                             0x01004
#define Ki_ADDRESS                              0x01008

unsigned char InsertOnBus (unsigned int,unsigned int,void *,char, char );
HANDLE_CAN_MSG InsertOnBus (unsigned int,unsigned int,void  *,char, char);
void CANMSGSemPend(HANDLE_CAN_MSG);
void CANMSGSemPost(HANDLE_CAN_MSG);
char CANDataAvaliable(HANDLE_CAN_MSG);

enum ComandoHodometro{
        RUN_HODOMETRO = '0',
        LIMPA_FALHAS_HODOMETRO,
}ComandoHodometro;

////////////////////////////////////////////////////////////////////////////////
// Estruturas de comando Controlador


typedef union STATUS_CONTROLADOR{
        struct {
                char Run                                        :1;//0
                char SalvandoParametros                 :1;//1
                char AtulizandoParametros   :1;//2
                char ErroSalvamentoParametros   :1;//3
                char PerdaRTControlador                 :1;//4
                char AlertaDeColisao                    :1;//5
                char DriverRun                                  :1;//6
                char EncEsqDes                          :1;//7
                char EncDirDes                          :1;//8
                char FaultMD                                    :1;//9
                char FaultME                                    :1;//10
                char PerdaSinc                                  :1;//11
                char PerdaRTDriverMotores       :1;//12
                char PerdaRTHodometro                   :1;//13
        }Bit;
        unsigned int Byte;
}STATUS_CONTROLADOR;

STATUS_CONTROLADOR StatusControlador;


typedef enum COMANDO_CONTROLADOR{
        EXECUTAR = '0',
        RESETAR_CONTROLADOR,
        DESLIGAR_CONTROLADOR,
        SALVAR_PARAMETROS,
        ATUALIZAR_PARAMETROS,
        LIMPA_FALHAS,
}COMANDO_CONTROLADOR;
```

```
COMANDO_CONTROLADOR ComandoControlador;


////////////////////////////////////////////////////////////////////////////////
//Dados Hodometro

typedef union STATUS_HODOMETRO{
	struct {
		char PerdaRT :1;
	}Bit;
	char Byte;
}STATUS_HODOMETRO;

STATUS_HODOMETRO StatusHodometro;

////////////////////////////////////////////////////////////////////////////////
//Dados do DriverMotores
typedef union MOTORES_COMANDO{
	struct{
		char LimpaFalha		:1;
		char LigaMotores	:1;
	}Bit;
	char Byte;
}MOTORES_COMANDO;

MOTORES_COMANDO MotoresComando;

typedef union MOTORES_STATUS{
	struct {
		char Run		: 1;
		char EncEsqDes	: 1;
		char EncDirDes	: 1;
		char FaultMD	: 1;
		char FaultME	: 1;
		char PerdaSinc	: 1;
		char PerdaRT	: 1;
	}Bit;
	char Byte;
}MOTORES_STATUS;

MOTORES_STATUS MotoresStatus;

typedef struct ESTRUTURA_IR{
	unsigned int	Distancia0;
	unsigned int	Distancia1;
	unsigned int	Distancia2;
	unsigned int	Distancia3;
}ESTRUTURA_IR;

ESTRUTURA_IR IR;

////////////////////////////////////////////////////////////////////////////
// Variáveis globais do controlador


long int	SPVL = 0;
long int	SPDV = 0;
long int		Kp = 0;
long int		Kid = 0;
long int		Ki = 0;
long int	Vre = 0;
long int	Vrd= 0;
```

```
long int        Ue = 0;
long int        Ud = 0;
long int        VreCAN = 0;
long int        VrdCAN = 0;
int                             UeCAN = 0;
int                             UdCAN = 0;

OS_EVENT                *SemControlador;

////////////////////////////////////////////////////////////////////////////////
// rotina de interrupção

void IntControlador(void){
static int      Timer = 0;

        if ((--Timer) <= 0){
                Timer = PERIODO_CONTROLADOR;
                OSSemPost(SemControlador);
        }
}

char EsperaTick (void){
        if ((OSSemAccept(SemControlador)) >= 1){
                return 0;
        }
        else{
                OSSemPend(SemControlador, 0, &err);
                return 1;
        }
}



////////////////////////////////////////////////////////////////////////////////
/* Rotinas de manipulaçao da memoria flash */

void ReadFlashMemory(char *Dest,char *Address,unsigned int Len){
        char *Origem;
        unsigned int i = 0;

        Origem = Address;
        for (i = 0;i < Len;i++){
                *(Dest + i) = *(Origem + i);
        }
}


unsigned int EraseFlashMemory(char *Address){
        char *Pointer;
        _DINT();
        while (FCTL3 & BUSY);
        Pointer = Address;
        FCTL2 = FWKEY + FSSEL_1 + 39;
        FCTL1 = FWKEY + ERASE;
        FCTL3 = FWKEY;
        *Pointer = 0;
        while (FCTL3 & BUSY);
        FCTL3 = FWKEY + LOCK;
        _EINT();
        return (FCTL3 & FAIL);
}

unsigned int WriteFlashMemory(char *Origem,char *Address,unsigned int Len){
        char *Dest;
        unsigned int i = 0;
```

```
	Dest = Address;

	_DINT();


	for (i = 0;i < Len;i++){
		while (FCTL3 & BUSY);
		FCTL3 = FWKEY;
		FCTL2 = FWKEY + FSSEL_1 + 45;
		FCTL1 = FWKEY + WRT;
		*(Dest + i) = *(Origem + i);
		while ((FCTL3 & WAIT) == 0);
	}

	FCTL1 = FWKEY;
	while (FCTL3 & BUSY);
	FCTL3 = FWKEY + LOCK;
	_EINT();
	return (FCTL3 & FAIL);
}



/*
*********************************************************************************************************
***************
*                                   CONTROLADOR TASK
*********************************************************************************************************
***************
*/
void Controlador(void *pdata){
	HANDLE_CAN_MSG		MSGSpeedData;
	HANDLE_CAN_MSG		MSGPWMData;
	HANDLE_CAN_MSG		MSGStatusMotores;
	HANDLE_CAN_MSG	MSG_IRData;
	HANDLE_CAN_MSG	MSG_StatusHodometro;

	enum Estado{
		COMANDO = 0,
		VERIFICA_FALHAS,
		EXECUCAO,
		RESET_CONTROLADOR,
		SALVA_PARAMETROS,
		ATUALIZA_PARAMETROS,
		DESLIGA_CONTROLADOR,
		REPARA_FALHAS,
	}Estado;

	long int  KpLinha = 0;
	long int  KidT = 0;
	long int  KiT = 0;
	long int   Sc = 0;
	long int   Se = 0;
	long int   Sd = 0;
	long int   Cc = 0;
	long int   Cd = 0;
	long int   Ce = 0;
	long int   Ed = 0;
	long int   Ee = 0;
	long int   Ec = 0;
	int		CountErros = 0;

	MSGSpeedData		= InsertOnBus (0x0111,5,(void *)&VreCAN,8,0);
	MSGStatusMotores	= InsertOnBus (0x0115,5,(void *)&MotoresStatus,1,0);	//10
```

```
MSG_StatusHodometro = InsertOnBus (0x0415,5,(void *)&StatusHodometro.Byte,1,0);    //10
MSG_IRData                          = InsertOnBus (0x0113,5,(void *)&IR.Distancia0,8,0);          //10

InsertOnBus (0x0450,0,(void *)&StatusControlador.Byte,2,0);
InsertOnBus (0x0110,0,(void *)&UeCAN,4,0);
InsertOnBus (0x0109,0,(void *)&MotoresComando,1,0);



Estado = RESET_CONTROLADOR;
ComandoControlador = EXECUTAR;

pdata  = pdata;

OSTimeDly(1500);

SemControlador =  OSSemCreate(0);

while(1){
        switch (Estado){

                case COMANDO:
                        switch (ComandoControlador){
                        case EXECUTAR:
                                StatusControlador.Bit.Run = 1;
                                StatusControlador.Bit.SalvandoParametros = 0;
                                StatusControlador.Bit.AtulizandoParametros = 0;
                                Estado = VERIFICA_FALHAS;
                                break;
                        case RESETAR_CONTROLADOR:
                                StatusControlador.Bit.Run = 0;
                                StatusControlador.Bit.SalvandoParametros = 0;
                                StatusControlador.Bit.AtulizandoParametros = 0;
                                Estado = RESET_CONTROLADOR;
                                break;
                        case DESLIGAR_CONTROLADOR:
                                StatusControlador.Bit.Run = 0;
                                StatusControlador.Bit.SalvandoParametros = 0;
                                StatusControlador.Bit.AtulizandoParametros = 0;
                                Estado = DESLIGA_CONTROLADOR;
                                break;
                        case SALVAR_PARAMETROS:
                                StatusControlador.Bit.Run = 0;
                                StatusControlador.Bit.SalvandoParametros = 1;
                                StatusControlador.Bit.AtulizandoParametros = 0;
                                Estado = SALVA_PARAMETROS;
                                break;
                        case ATUALIZAR_PARAMETROS:
                                StatusControlador.Bit.Run = 0;
                                StatusControlador.Bit.SalvandoParametros = 0;
                                StatusControlador.Bit.AtulizandoParametros = 1;
                                Estado = ATUALIZA_PARAMETROS;
                                break;
                        case LIMPA_FALHAS:
                                StatusControlador.Bit.Run = 0;
                                StatusControlador.Bit.SalvandoParametros = 0;
                                StatusControlador.Bit.AtulizandoParametros = 0;
                                Estado = REPARA_FALHAS;
                                break;
                        default:
                                break;
                        }

                        if (EsperaTick() == 0)
                                StatusControlador.Bit.PerdaRTControlador = 1;
```

```
                                        if((IR.Distancia3 < 300)){
                                                *Status*Controlador.Bit.AlertaDeColisao = 1;
                                        }
                                        else{
                                                *Status*Controlador.Bit.AlertaDeColisao = 0;
                                        }

                                        *Status*Controlador.*Byte* &=  0x003F;
                                        *Status*Controlador.*Byte* += (((unsigned int)(Motores*Status*.*Byte*)) & 0x007F)
<< 6;
                                        *Status*Controlador.Bit.PerdaRTHodometro = *Status*Hodometro.Bit.PerdaRT;

                                        if ((*Status*Controlador.Bit.PerdaSinc == 0) &&
(CANDataAvaliable(MSGSpeedData)) && (CANDataAvaliable(MSG*Status*Motores)) &&
(CANDataAvaliable(MSG_IRData)) && CANDataAvaliable(MSG_*Status*Hodometro)){
                                                *Status*Controlador.Bit.PerdaSinc = 0;
                                                CountErros = 0;
                                        }
                                        else{
                                                CountErros ++;
                                                if (CountErros >= 6){
                                                        *Status*Controlador.Bit.PerdaSinc = 1;
                                                        CountErros = 6;
                                                }
                                        }

                                        break;

                                case VERIFICA_FALHAS:

                                        if (*Status*Controlador.*Byte* & 0X3F98){
                                                SPDV = 0;
                                                SPVL = 0;
                                                UdCAN = 0;
                                                UeCAN = 0;
                                                Sc = 0;
                                                Se = 0;
                                                Sd = 0;
                                                MotoresComando.Bit.LigaMotores = 0;

                                                OSTimeDly(1500);
                                                OSSemSet(SemControlador, 0, &err);
                                                Estado = COMANDO;
                                        }
                                        else{
                                                MotoresComando.Bit.LigaMotores = 1;
                                                Estado = EXECUCAO;
                                        }
                                        break;

                                case EXECUCAO:
                                        // Parada de emergencia
                                        if (*Status*Controlador.Bit.AlertaDeColisao == 1){
                                                SPDV = 0;
                                                SPVL = 0;
                                                UdCAN = 0;
                                                UeCAN = 0;
                                                Sc = 0;
                                                Se = 0;
                                                Sd = 0;
                                        }

                                        if (SPDV >= LIMITE_SPDV)
```

```
                SPDV = LIMITE_SPDV;
            if (SPDV <= -LIMITE_SPDV)
                SPDV = -LIMITE_SPDV;

            if (SPVL >= LIMITE_SPVL)
                SPVL = LIMITE_SPVL;
            if (SPVL <= -LIMITE_SPVL)
                SPVL = -LIMITE_SPVL;


            //CANMSGSemPend(MSGSpeedData);
            _DINT();
            Vrd = VrdCAN;
            Vre = VreCAN;
            _EINT();
            //CANMSGSemPost(MSGSpeedData);

            Ec = (Vre + (SPDV - Vrd));
            if (!(((Sc >= LIMITEINTEGRAL) && (Ec > 0)) || ((Sc <= -
LIMITEINTEGRAL) && (Ec < 0))))
                Sc = Sc + Ec;
            Cc = (KidT >> 7) * (Sc >> 10); //Valor Multiplicado por 1000

            Ed = (SPVL - Vrd) + Cc;//Valor Multiplicado por 1000
            if (!(((Sd >= LIMITEINTEGRAL) && (Ed > 0)) || ((Sd <= -
LIMITEINTEGRAL) && (Ed < 0))))
                Sd = Sd + Ed;
            Cd = (KiT >> 7) * (Sd >> 10) ;//Valor Multiplicado por 1000
            Ud = (((KpLinha*Ed) >> 7) + Cd) / 6;
            if (Ud >= LIMITE_SAT_PWM)
                Ud = LIMITE_SAT_PWM;
            if (Ud <= -LIMITE_SAT_PWM)
                Ud = -LIMITE_SAT_PWM;

            Ee = (SPVL - Vre) - Cc;//Valor Multiplicado por 1000
            if (!(((Se >= LIMITEINTEGRAL) && (Ee > 0)) || ((Se <= -
LIMITEINTEGRAL) && (Ee < 0))))
                Se = Se + Ee;
            Ce = (KiT >> 7) * (Se >> 10);//Valor Multiplicado por 1000
            Ue = (((KpLinha*Ee) >> 7) + Ce) / 6;
            if (Ue >= LIMITE_SAT_PWM)
                Ue = LIMITE_SAT_PWM;
            if (Ue <= -LIMITE_SAT_PWM)
                Ue = -LIMITE_SAT_PWM;

            //CANMSGSemPend(MSGPWMData);
            _DINT();
            UdCAN = (int)(Ud);
            UeCAN = (int)(Ue);
            _EINT();
            //CANMSGSemPost(MSGPWMData);

            Estado = COMANDO;
            break;
        case RESET_CONTROLADOR:

            Read_FlashMemory((void *)(&Kp),(void *)(Kp_ADDRESS),12);
            KpLinha = Kp;
            KpLinha = KpLinha << 7;
            KpLinha /= 100;
            KiT = (Ki * PERIODO_CONTROLADOR);
            KiT = KiT << 7;
            KiT /= 100;
            KidT = (Kid * PERIODO_CONTROLADOR);
            KidT = KidT << 7;
```

```
                KidT /= 100;

                SPDV = 0;
                SPVL = 0;
                UdCAN = 0;
                UeCAN = 0;
                Sc = 0;
                Se = 0;
                Sd = 0;
                ComandoControlador = EXECUTAR;
                Estado = COMANDO;
                break;
        case SALVA_PARAMETROS:

                if (Erase FlashMemory((void *)(Kp_ADDRESS))) {
                        StatusControlador.Bit.ErroSalvamentoParametros = 1;
                        Estado = COMANDO;
                }
                else{
                        if (WriteFlashMemory((void *)(&Kp),(void *)(Kp_ADDRESS),12)){
                                StatusControlador.Bit.ErroSalvamentoParametros = 1;
                                Estado = COMANDO;
                        }
                        else{
                                StatusControlador.Bit.ErroSalvamentoParametros = 0;
                                Estado = RESET_CONTROLADOR;
                        }
                }
                break;
        case ATUALIZA_PARAMETROS:
                KpLinha = Kp;
                KpLinha = KpLinha << 7;
                KpLinha /= 100;
                KiT = (Ki * PERIODO_CONTROLADOR);
                KiT = KiT << 7;
                KiT /= 100;
                KidT = (Kid * PERIODO_CONTROLADOR);
                KidT = KidT << 7;
                KidT /= 100;

                ComandoControlador = EXECUTAR;
                Estado = COMANDO;
                break;
        case DESLIGA_CONTROLADOR:
                SPDV = 0;
                SPVL = 0;
                UdCAN = 0;
                UeCAN = 0;
                Sc = 0;
                Se = 0;
                Sd = 0;
                MotoresComando.Bit.LigaMotores = 0;
                Estado = COMANDO;
                break;
        case REPARA_FALHAS:

                        MotoresComando.Bit.LigaMotores = 0;
                        MotoresComando.Bit.LimpaFalha = 1;
                        OSTimeDly(100);
                        MotoresComando.Bit.LimpaFalha = 0;
                        OSTimeDly(100);

                        OSSemSet(SemControlador, 0, &err);
```

```
                                                ComandoHodometro = LIMPA_FALHAS_HODOMETRO;
                                                TXCAN((char)(0x0420 >> 3),(char)(0x0420 <<
5),1,0,&ComandoHodometro);

                                                StatusControlador.Bit.PerdaRTControlador = 0;
                                                ComandoControlador = EXECUTAR;
                                                Estado = COMANDO;

                                        break;
                                default:
                                        Estado = RESET_CONTROLADOR;
                                        OSTimeDly(PERIODO_CONTROLADOR);
                                        break;
                        }
                }
        }
        ////////////////////////////////////////////////////////////////////////////////////////////////////////////
```