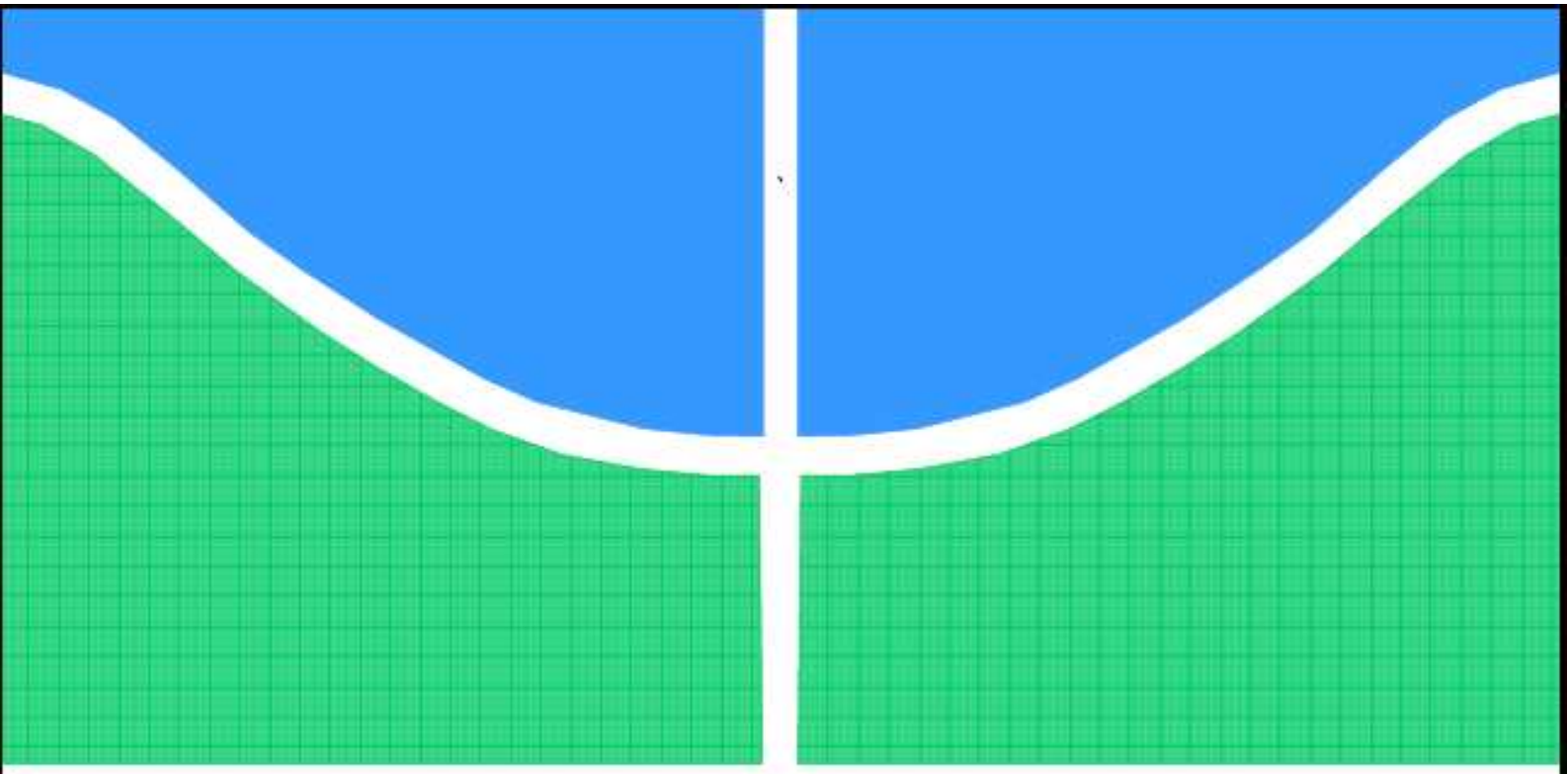

TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE DOIS ROBÔS MÓVEIS EM ARQUITETURA CLIENTE SERVIDOR PARA PESQUISA EM ROBÓTICA COOPERATIVA

Leandro Ribeiro Lustosa

Sinara Augusta Borges Campos

Brasília, julho de 2009

ENGENHARIA MECATRÔNICA

UNIVERSIDADE DE BRASÍLIA

UNIVERSIDADE DE BRASILIA
Faculdade de Tecnologia

TRABALHO DE GRADUAÇÃO

DESENVOLVIMENTO DE DOIS ROBÔS MÓVEIS EM ARQUITETURA CLIENTE SERVIDOR PARA PESQUISA EM ROBÓTICA COOPERATIVA

Leandro Ribeiro Lustosa

Sinara Augusta Borges Campos

Relatório submetido ao Departamento de Engenharia Mecatrônica como requisito parcial para obtenção do grau de Engenheiro de Controle e Automação

Banca Examinadora

Prof. Dr. Geovany Araújo Borges, ENE/UnB _____
Orientador

Prof. M. Eng. Lélio Ribeiro Soares, ENE/UnB _____
Examinador interno

Profa. Dra. Carla Maria Chagas e Calvacante Koike, ENE/UnB _____
Examinador interno

Dedicatórias

Aos meus pais e irmãos por todo o apoio, ao Leandro, grande companheiro de trabalho, e ao professor Geovany pelo incentivo e orientação.

Sinara Augusta Borges Campos

Aos meus pais que souberam me guiar pelos caminhos mais sábios. Ao don, meu irmão que trouxe equilíbrio ao meu stress com suas piadas e ironias geniais. À vó Beth e à vó Nazaré, por trazer harmônia a esta grande família na qual tenho orgulho de ter sido criado. À Sinara, grande companheira de trabalho que permitiu agregar à este trabalho a qualidade que o mesmo possui.

Leandro Ribeiro Lustosa

RESUMO

Esse trabalho consiste da construção de duas plataformas móveis, sendo uma diferencial e outra omnidirecional, para estudos e experimentos em robótica de cooperação. A contrução abrange desde a confecção da estrutura mecânica, até a composição eletrônica e engenharia de software. Ambas plataformas possuem diversos recursos sensoriais, tais como sensores infravermelho para estimação de distância, sensores de rotação, sensores ópticos e camêras. Os comandos das plataformas e os dados referentes ao sensores das mesmas estarão disponíveis em uma rede TCP/IP de forma a auxiliar experimentos em robótica de cooperação. Neste trabalho também são implementados sistemas de odometria e controle de velocidade em robótica móvel terrestre.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

Símbolos Latinos

Símbolos Gregos

Δ	Variação entre duas grandezas similares	
σ^2	Variância de uma variável aleatória	
ω_i	Velocidade angular da i-ésima roda	[rad/seg]

Grupos Adimensionais

K_p	Ganho Proporcional de um controlador PID
K_i	Ganho Integral de um controlador PID
K_d	Ganho Derivativo de um controlador PID

Subscritos

i	i-ésima

Sobrescritos

$\cdot$	Variação temporal
$-$	Valor médio

Siglas

A/D	Analógico/Digital
ABNT	Associação Brasileira de Normas Técnicas
AF_INET	Address Family Internet
API	Application Programming Interface
ARM	Advanced RISC Machine
CI	Circuito Integrado
GPS	Global Positioning System
GTK	Graphic Tool Kit
LCD	Liquid crystal display
LED	Light Emitting Diode
ms	Milissegundos
PCB	Printed Circuit Board
PI	Proporcional Integral
PID	Proporcional Integral Derivativo
PIT	Periodic Interrupt Timer
PWM	Pulse Width Modulation
RF	Rádio Frequência
RISC	Reduced Instruction Set Computer
RST	Reset
TCP/IP	Transmission Control Protocol/Internet Protocol
TTL	Transistor Transistor Logic
USART	Universal Synchronous Asynchronous Receiver Transmitter
USB	Universal Serial Bus

Capítulo 1

Introdução

Este capítulo realiza uma breve apresentação da área de robótica móvel, sistemas multi-robôs e desafio da cooperação entre robôs, sendo este último o grande motivador deste trabalho de graduação. Os objetivos são claramente apresentados para construir a expectativa do leitor acerca do trabalho realizado. Por fim, o manuscrito é apresentado.

1.1 Contextualização

A pesquisa em Robótica está presente em diversas vertentes no Laboratório de Robótica e Automação (LARA) do Departamento de Engenharia Elétrica da Universidade de Brasília. Na área de robótica móvel terrestre foi desenvolvida em 2005 uma plataforma móvel omnidirecional denominada ROHL [1] para fins de estudo de temas como odometria, mecânica, eletrônica, programação, modelamento cinemático, métodos de estimação de posição de robôs móveis e métodos de controle de velocidade. O robô construído possuía recursos escassos, tendo sido implementados apenas circuitos de alimentação, acionamento e controle de velocidade. A plataforma não possuía elementos que lhe permitisse obter uma melhor capacidade de localização como sensores acelerômetro ou giroscópio, e outros sensores que lhe oferecessem interação com o ambiente para desenvolvimento de um sistema de navegação, como sensores infravermelhos para detecção de distância ou obstáculos, ou ainda uma câmera. Segundo verificação dos autores do trabalho ROHL, as rodas holonômicas apresentavam problemas de aderência e os motores não se mostravam robustos o suficiente. De acordo com os fatores apresentados acima a possibilidade de continuidade de trabalho sob essa plataforma não mostrou-se convidativa. A plataforma desenvolvida é mostrada na figura 1.1.

Para que novos trabalhos de pesquisa em robótica móvel fossem desenvolvidos, partiu-se para o desenvolvimento de novas plataformas. A idéia era construir dois robôs, com capacidade de locomoção distintas, um diferencial e outro omnidirecional, que permitissem o estudo de dois tipos de modelagem cinemática e cálculo de odometria. Eles deveriam apresentar uma construção mecânica capaz de abrigar uma quantidade de *hardware* maior, como sensores e processadores. Sua estrutura seria leve e pequena para facilitar a locomoção em ambientes variados, além de tornar o transporte manual dos robôs simples. O *hardware* seria mais robusto, baseado em componentes

Figura 1.1: Robô móvel omnidirecional ROHL desenvolvido no LARA em 2005

eletrônicos modernos e sensores mais inteligentes. E como objetivo de trabalho, o projeto deveria apresentar capacidade de cooperação. Ou seja, ambas as plataformas deveriam ser capazes de trocar informações em tempo real.

1.2 Apresentação do tema

Apesar de a robótica estar cada vez mais presente nas mais diversas áreas de trabalho, sempre ouve-se o questionamento sobre o que é um "robô" e o que ele pode fazer.

Popularmente, associa-se o nome "robô" a uma composição mecânica com aparência humana, e foi realmente dessa idéia que o termo surgiu. Em 25 de Janeiro de 1921 na cidade de Praga, o nome "robô" foi usado pela primeira vez em um conto de ficção chamado *Rossum's Universal Robots*, sendo associado à pequenos bonecos mecânicos trabalhadores que seriam capazes de substituir um humano em qualquer trabalho. Desde então, o termo passou a estar ligado diretamente a uma criatura que realiza um trabalho humano. Com o tempo, a associação à capacidade de realizar um trabalho humano sobrepôs a relação com aparência humana para definição de um robô. Atualmente, a comunidade científica classifica como um robô inteligente uma criatura mecânica com capacidade de funcionar de maneira autônoma. Essa autonomia está ligada à capacidade do robô de se adaptar a mudanças no ambiente e continuar realizando sua missão.

O crescimento da população mundial aumentou muito a demanda por bens e serviços. Para aumentar a eficiência e capacidade de produção houve um crescimento da demanda por tecnologias autônomas como robôs que tivessem capacidade de realizar atividades humanas.

O funcionamento de um robô está associado a três princípios: Sentir, planejar e agir. Sentir está ligado a captar dados do ambiente através de sensores e gerar informações úteis ao robô. Essas informações dos sensores permitem que o robô elabore tarefas a serem realizadas, ou seja, realiza um planejamento de suas atividades. Diante dos dados captados, associados ou não a um planejamento, o robô pode tomar uma decisão e agir enviando comandos aos seus atuadores.

1.2.1 Robótica móvel

Uma das áreas impulsionadas por essa demanda foi a robótica móvel. Robôs móveis apresentam a capacidade de atuar em ambientes dinâmicos e não estruturados. Existem diversos obstáculos na implementação desta tecnologia que estão sempre associados à complexidade das tarefas motivadoras do uso de robôs móveis. O trabalho que será apresentado neste manuscrito apresenta algumas dessas dificuldades e mostra as soluções adotadas para realização da tarefa proposta.

Comercialmente, a motivação para a adoção de robôs móveis autônomos está na possibilidade de se ter estes elementos atuando de forma não-supervisionada em tarefas complexas que requerem interação com o meio físico. Os robôs móveis estão presentes em tarefas como a exploração espacial, a inspeção submarina de cabos e oleodutos, o transporte de cargas de alto risco e o rastreamento de minas terrestres, as quais tem, mais recentemente, contribuído para um crescimento considerável da pesquisa nessa área.

Sob o tripé sentir-planejar-agir, um robô móvel consegue estabelecer essa interação com o ambiente e atuar no sentido de cumprir a tarefa que lhe foi estabelecida.

1.2.2 Robótica de Cooperação

Motivados pela necessidade de autonomia e, portanto, capacidade de atuar de forma não supervisionada, sistemas multi-robô têm se apresentado como uma boa proposta para resolver tarefas de forma inteligente e eficiente para o ser humano. Um sistema multi-robô caracteriza-se por uma coleção de robôs móveis trabalhando juntos como um time. A figura 1.2 mostra um trabalho de pesquisa na área de robótica móvel cooperativa, em que os robôs possuem habilidade de localização, de coletar matérias no ambiente, e de retornar com o material até a base. Este sistema permite que os robôs sejam distribuídos em uma região que se deseja explorar cobrindo uma vasta área em muito menos tempo. Outra vantagem é que caso ocorra uma falha ou um dos robôs seja destruído, os outros robôs podem continuar e completar a tarefa sem que exista perda considerável de rapidez e eficiência.

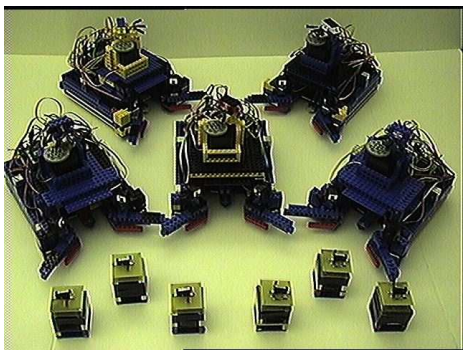

Figura 1.2: Robôs da Equipe *MinDART - The Minnesota Distributed Autonomous Robotic Team.*

Identificar a arquitetura de um sistema multi-robô é uma tarefa complicada, pois, por se tratar de um campo de pesquisa ainda muito recente, não existe consenso acerca de características e aspectos importantes de um sistema como esse. Em [2], alguns aspectos como *heterogeneidade*,

controle, *cooperação* e *objetivos*, mostram-se interessantes para descrição e caracterização de uma equipe.

Heterogeneidade refere-se à semelhanças entre um robô e o seu grupo. Semelhanças estão ligadas à igualdade da arquitetura de *hardware* e *software* entre os membros. Assim, os grupos de robôs podem ser *homogêneos* se os membros de equipe são idênticos, como os robôs desenvolvidos por *Wuhan Institute of Technology team*, participando da Oitava Competição Nacional de Futebol de Robôs em Wuhan apresentados na figura 1.3, ou ainda *heterogêneos* se o *hardware* ou *software* de dois membros forem diferentes, como os robôs *Flo* e *Jo* desenvolvidos com capacidades heterogêneas para pesquisa em robotica móvel e cooperação observados na figura 1.4.

Figura 1.3: Robôs desenvolvidos por *Wuhan Institute of Technology team*.

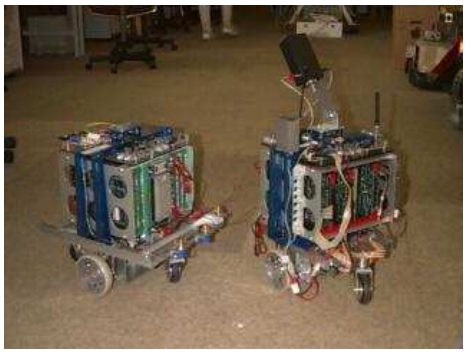

Figura 1.4: Robôs Flo e Jo desenvolvidos com capacidades heterogêneas.

Outro aspecto importante é a tarefa de controle. Ela pode ser realizada de forma *distribuída*, em que cada robô é responsável por tomar suas próprias decisões e agir, ou de forma *centralizada*, onde os dados colhidos por cada membro do grupo é passado à uma central de processamento, possivelmente um computador, e esta repassa a cada robô do grupo as ações a serem realizadas.

Uma das vertentes de sistemas multi-robô é dotar robôs autônomos com habilidades sociais que os permitam interagir com outros robôs autônomos e formar equipes de trabalho para a realização de tarefas cooperativas. Essa interação é chamada cooperação. Nesse contexto, a cooperação pode ser *ativa*, se os robôs trocarem dados (comunicação) ou objetos entre os demais membros. Será *inativa*, caso contrário.

Quanto ao escalonamento de tarefas, uma mesma tarefa pode ser compartilhada por todos os membros da equipe.

Como cada um dos aspectos levantados nesta seção pode se apresentar depende do tipo de

aplicação e tarefa a ser realizada.

1.3 Robôs Móveis Diferencial e Omnidirecional

Neste trabalho são desenvolvidas duas plataformas móveis, uma diferencial e outra omnidirecional. Um robô diferencial apresenta duas rodas tracionáveis e uma terceira roda livre para sustentação da estrutura. Sua movimentação é restrita à apenas uma direção (no caso, o robô pode apenas se mover incrementalmente para frente apenas, com alguma velocidade angular). A estrutura de um robô diferencial é mais comum por apresentar maior simplicidade de implementação.

Já um robô omnidirecional é um robô que possui capacidade de movimentação em qualquer direção no plano sem que exista necessidade de rotação do próprio corpo. A omnidirecionalidade de robôs pode ser obtida por diversas formas, entretanto, são mais utilizadas rodas holonômicas ou rodas orientáveis com eixo deslocado [1,6].

1.4 Definição do problema

O problema que este trabalho pretende solucionar é o desenvolvimento de um ambiente no qual se possa desenvolver estudos e experimentos em robótica de cooperação. Para tanto, construiu-se nesse projeto duas plataformas com capacidades de locomoção distintas, de forma a tornar o problema de cooperação mais interessante. Uma plataforma terá locomoção diferencial enquanto a outra possuirá locomoção omnidirecional. Devido a capacidade de locomoção de cada robô, eles receberam os seguintes nomes: **Jessi XX**, para a plataforma diferencial que se locomove em uma única direção (no caso a direção X_m ilustrada na figura 3.27), e **Kyle XY** para o robô omnidirecional que se locomove em qualquer direção em um plano XY. Ainda, as siglas XY e XX determinam um gênero (feminino ou masculino) para cada plataforma.

Por se tratar de estudos em robótica de cooperação, é desejável que cada plataforma tenha conhecimento do estado atual da configuração de todas as plataformas em estudo e do ambiente no qual essas estão envolvidas. Para tal, definiu-se neste projeto a necessidade de se disponibilizar uma interface na qual cada robô pode comandar ou requisitar dados de outros robôs através de uma rede TCP/IP.

Também definiu-se como necessidade deste projeto deixar todos os parâmetros de configuração de uma plataforma, tais como parâmetros do controlador de velocidade e parâmetros das curvas de calibração dos sensores, facilmente disponíveis para o pesquisador poder alterá-los, sem a necessidade de recompilação de códigos.

Deve-se também desenvolver a arquitetura de *software* e *hardware* para as duas plataformas de forma similar, apesar de possuírem capacidades de locomoção distintas, pois a similaridade facilita o desenvolvimento das plataformas e torna os estudos posteriores em cima destas mais simples. Também, o trabalho que for desenvolvido em uma plataforma, poderá ser facilmente adaptado à segunda plataforma.

Durante o desenvolvimento, julgou-se importante a modularização do projeto. Modularização implica em reusabilidade, rápida manutenção, depuração e disponibilização de uma documentação de fácil entendimento à pessoas externas ao projeto.

Portanto, esse projeto não visa o estudo de robótica de cooperação propriamente dito. Mas viabiliza um ambiente completo de trabalho para estudos no assunto.

1.5 Objetivos do projeto

São objetivos deste projeto:

- Construção da estrutura mecânica de duas plataformas robóticas;
- Desenvolvimento dos circuitos de acionamento, sensoriamento e processamento de dados;
- Desenvolvimento da interface de sensores proprioceptivos: *encoders* ópticos, girômetro;
- Desenvolvimento da interface de sensores exteroceptivos: webcam, sensores infravermelho;
- Disponibilização de uma API para controle da plataforma;
- Disponibilização de um ambiente para interação das plataformas em rede TCP/IP;

1.6 Apresentação do manuscrito

Esse manuscrito foi dividido em quatro capítulos. O primeiro capítulo é uma introdução ao trabalho realizado. Este capítulo realiza uma breve apresentação da área de robótica móvel, sistemas multi-robôs e desafio da cooperação entre robôs, sendo esse último o grande motivador deste trabalho de graduação. Os objetivos são claramente apresentados para construir a expectativa do leitor acerca do trabalho realizado. No capítulo dois serão consideradas as arquiteturas de *hardware* utilizadas no desenvolvimento de ambas as plataformas móveis deste trabalho. Após a leitura deste capítulo, o leitor terá um entendimento substancial das capacidades e limitações de cada plataforma. No capítulo três serão consideradas as arquiteturas de *software* utilizadas nas unidades de processamento do robô. Após ter lido esse capítulo, o leitor terá um entendimento completo sobre como o robô interpreta por *software* os sinais físicos do *hardware* do capítulo anterior e os utiliza para cumprir seus objetivos. Finalmente, no capítulo quatro serão encontradas as idéias dos autores quanto aos resultados obtidos por este trabalho de graduação.

Capítulo 2

Construção Mecânica e Eletrônica

Neste capítulo serão consideradas as arquiteturas de hardware utilizadas no desenvolvimento de ambas as plataformas móveis deste trabalho. Após lido este capítulo, o leitor terá um entendimento substancial das capacidades e limitações de cada plataforma.

2.1 Introdução

No capítulo anterior, foram apresentados alguns trabalhos de pesquisa em robótica de cooperação desenvolvidos pela comunidade científica ao redor do mundo. Foi apresentada também a proposta de estudo desenvolvida no LARA para a área. De acordo com a proposta, foram construídas duas pequenas plataformas móveis, uma diferencial e outra omnidirecional. Neste capítulo, serão apresentadas as etapas de construção e a composição de ambas as plataformas. Para tanto, será apresentado inicialmente o projeto mecânico e estrutural dos robôs. Depois, será mostrada a composição eletrônica de todos os módulos existentes na plataforma. Para cada circuito ou elemento exposto, serão apresentados seus recursos e também suas funcionalidades.

Foi necessário durante todo o desenvolvimento do projeto ter em mente que as plataformas seriam utilizadas para pesquisa em robótica de cooperação. Portanto, a similaridade física entre as mesmas é uma restrição de projeto extremamente importante durante o trabalho. Uma motivação da restrição dada é simples: quando se necessitar fazer alguma alteração na estrutura das plataformas, deseja-se que a repetibilidade da mudança na outra seja dado por um esforço mínimo. De fato, o leitor concluirá que a maioria das mudanças de projeto que poderiam ser realizadas sobre uma plataforma, poderão ser igualmente reproduzidas na outra. Outra motivação, bastante forte, é a facilidade de programação das plataformas na situação onde as arquiteturas são compatíveis. É muito mais eficiente programar um código apenas e grava-lo em todas as plataformas do conjunto de robôs em estudo.

As plataformas robóticas deverão se locomover em um ambiente *indoors*. Assim, inseriu-se na arquitetura física rodas e atuadores. A geometria e a disposição física das rodas definem as capacidades de locomoção de cada plataforma. Ainda, deseja-se que uma plataforma tenha a possibilidade de estimar sua localização sem depender de um sistema de GPS. Por essa razão, inseriu-se na ar-

quitetura também sensores ópticos de rotação dos motores, girômetro e um microcontrolador para processamento das medições. O robô também tem de possuir estratégias para conhecer o ambiente ao seu redor. Nas plataformas deste trabalho, inseriu-se sensores infravermelho para medições de distância e uma câmera embarcada. Para o processamento de dados da câmera, foi adicionado à arquitetura um *netbook*. O *netbook* é conveniente também para a comunicação em rede, necessária para a inter-cooperação entre robôs. A figura 2.1 ilustra os componentes da arquitetura física de ambas as plataformas. A diversidade relativamente alta de recursos físicos embutidos nestas plataformas as tornam atraentes para inúmeros experimentos e estudos em robôtica de cooperação, além de estudos em robótica *solo* em visão computacional e fusão sensorial.

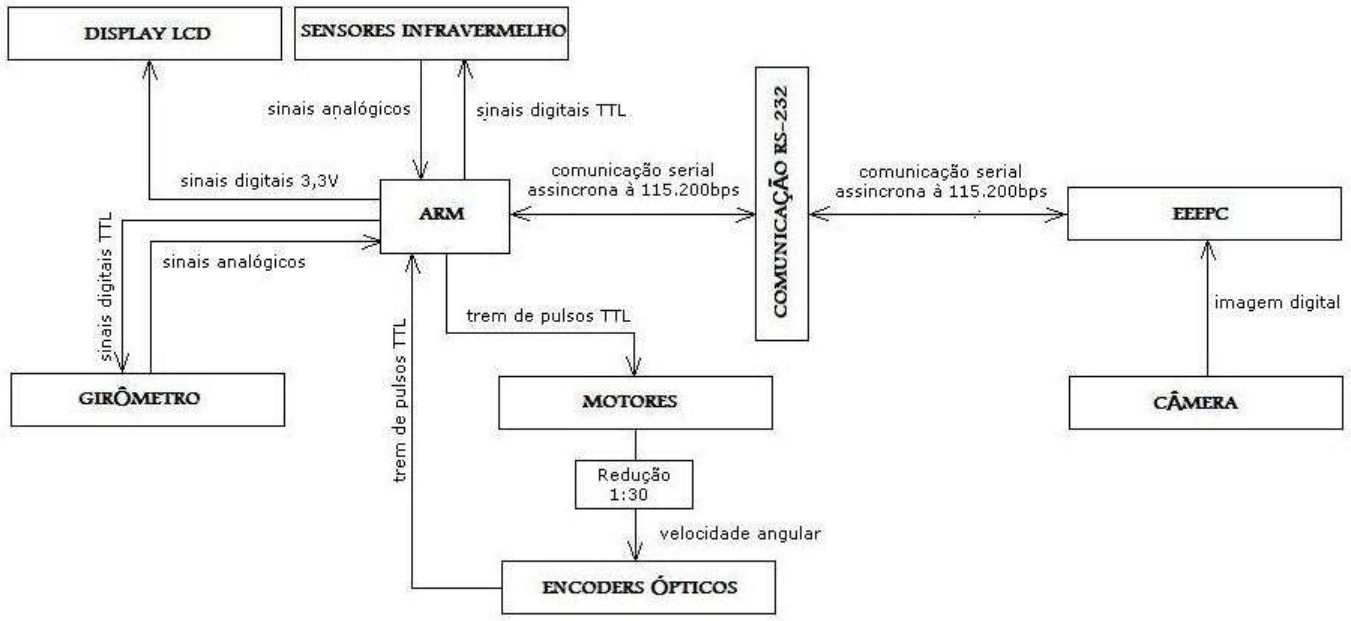

Figura 2.1: Arquitetura de *Hardware* de cada plataforma móvel.

Para abrigar o *hardware* apresentado acima, adotou-se uma composição estrutural prismática em níveis ou camadas, conforme ilustra a figura 2.2. Na figura pode-se observar três níveis ao todo. O primeiro nível (i.e., o mais baixo) acolhe os recursos de potência da plataforma, como circuitos de acionamento, bateria e elementos para distribuição de energia entre os diversos componentes de *hardware* presentes no robô. O segundo nível guarda os circuitos de processamento e caminho de dados, LCD, girômetro e comunicação, vide figuras 2.3 e 2.4. O terceiro e último nível abriga o *netbook* EEEPC e uma *webcam*.

Nas próximas sub-seções deste capítulo, cada módulo de *hardware* que compõe as plataformas será apresentado em detalhes. Serão mostradas funcionalidades e composição, além de experimentos e testes a fim de validar a escolha de cada componente.

2.2 Estrutura Mecânica

Formadas por três níveis, as bases das plataformas diferencial e omnidirecional possuem formato hexagonal regular pensando na distribuição de seis sensores infravermelho, um em cada lado do

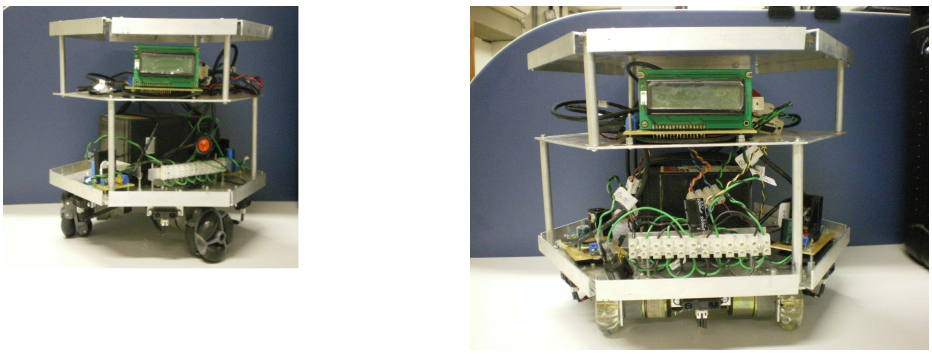

Figura 2.2: Vista lateral das plataformas, à esquerda a omnidirecional e à direita a diferencial.

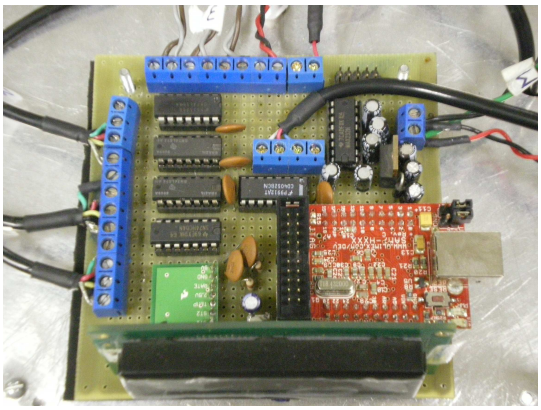

Figura 2.3: Placa principal do robô omnidirecional.

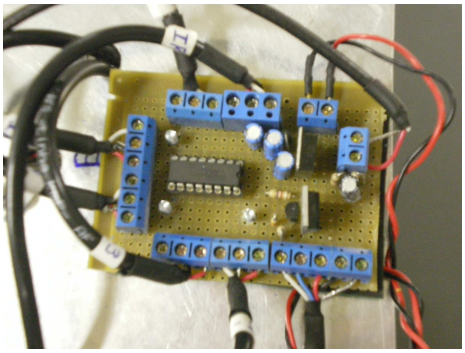

Figura 2.4: Circuito de administração dos sensores infravermelho.

hexágono, formando um cinturão. As três rodas da plataforma diferencial foram dispostas nos vértices de um triângulo isósceles, conforme ilustra a figura 2.5, à esquerda a diferencial e à direita a omnidirecional. As rodas tracionadas foram posicionadas paralelamente entre si e em dois vértices pertencentes à base do triângulo isósceles. A roda livre foi disposta no vértice restante do triângulo. Para a plataforma omnidireconal, as três rodas tracionadas foram dispostas nos vértices de um triângulo equilátero, conforme ilustra também a figura 2.5.

Para separar cada um dos níveis, foram utilizadas três barras cilíndricas de alumínio de 100mm de comprimento entre o primeiro e o segundo nível e três barras de 80mm entre o segundo e o terceiro nível. Além disso, doze placas torcidas de alumínio em formato 'L' retangular foram presas a cada uma das laterais inferior e superior(níveis um e três) para servirem como guias para encaixe

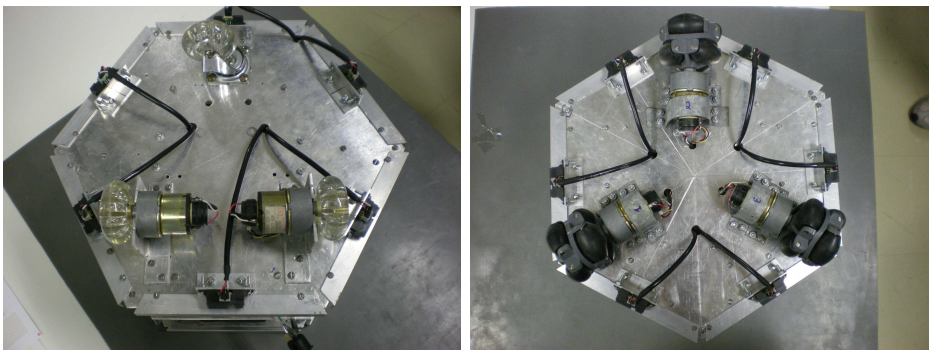

Figura 2.5: Vista inferior das plataformas.

de chapas de acrílico em torno do robô. Essas chapas que o envolvem oferecem uma interface amigável, além de proteger o *hardware* interno, e sua liberdade de encaixe e retirada permite fácil acesso ao *hardware* interno presente no primeiro e segundo níveis. Porém, existe a necessidade de acessar esse *hardware* para gravação de programas, aquisição e envio de dados para testes via serial RS-232, leitura de dados do LCD e um botão liga-desliga do robô. Para simplificar esse acesso, foram realizados furos nas placas de acrílico para fixar dos conectores DB25, DB9 e botão liga e desliga, além de furo que permite vizualizar o LCD, conforme se observa na figura 2.6, à esquerda a diferencial e à direita a omnidirecional.

Figura 2.6: Ilustração dos acessos através de furos nas placas de acrílico.

As rodas da plataforma diferencial, tanto tracionadas quanto livre, são de silicone e são comercializadas como rodas de gel para patins e *skateboards*. Possuem 50.60mm de diâmetro e apresentam boa resistência para suportar o peso do robô e o desgaste devido ao atrito com o solo. A roda livre possui um acoplamento em alumínio que dá a liberdade de rotação dessa roda. Ela já foi adquirida com essa composição e foi necessária apenas a fixação das rodas tracionadas a estruturas em zinco, as quais além de prender o conjunto motor-roda a base do robô, oferecem o nivelamento necessário para acompanhar a roda livre.

Na plataforma omnidirecional foram usadas rodas holonômicas cada uma com oito roletes que giram livremente perpendicularmente ao eixo de rotação da roda. São rodas de poliuretano fabricadas por *North American Roller Products*. Possuem diâmetro de $80mm$ de diâmetro e são

capazes de suportar até $34.02Kg$, sendo bastante usadas em aplicações de robótica móvel como a que este trabalho propõe. Para fixação das três rodas foram construídas três estruturas fixadoras feitas com formato "L" e dimensões semelhantes, todas em zinco.

Para tracionar todas as rodas de ambas as plataformas foram instalados motores de corrente contínua modelo HN-GH12-1634T-R do fabricante *JAMECO RELIAPRO*.

2.3 Fornecimento de energia

A alimentação de todo o *hardware* de acionamento, processamento, sensoriamento e comunicação é provida por uma bateria de 12V. Para distribuir a corrente utilizou-se um conector *sindal* de dez vias conforme mostrado na figura 2.7. Um capacitor de 2,2mF foi conectado diretamente aos terminais da bateria que chegam ao conector. O objetivo dele é diminuir os efeitos de oscilações de tensão causadas por carregamento da bateria. Para o ligamento e desligamento manual completo da distribuição de energia da bateria instalamos um interruptor, disponibilizado de forma a permitir fácil acesso ao mesmo.

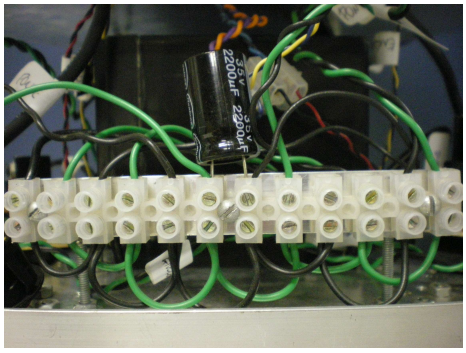

Figura 2.7: Distribuição de energia.

Porém, a alimentação de alguns elementos e dispositivos presentes no *hardware* da plataforma possui valor nominal 5V. Para converter 12V em 5V, utilizou-se um regulador de tensão na forma de circuito integrado, o 7805. Para que o CI7805 possa fornecer 5V em sua saída, a circuito da figura 2.8 foi implementado em ambas as plataformas. Foram utilizados capacitores conectados à entrada e à saída para eliminar ruídos RF, além de dar mais estabilidade à tensão de saída. Para que o CI7805 funcione corretamente, a tensão de saída V_{out} não pode ser maior que a entrada V_{in}.

Nota-se que existe um diodo 1N5819 na saída do circuito de alimentação. A função deste diodo é impedir que o circuito regulador de tensão seja danificado quando o microcontrolador é alimentado pela fonte de alimentação alternativa USB.

2.4 Comunicação RS-232

Deve haver um circuito de adaptação do canal de comunicação entre EEEPC e ARM. Ainda, durante o desenvolvimento do projeto, existiu a necessidade de depurar e acompanhar dados dos sensores *encoder*, girômetro e infravermelho. Esses dados muitas vezes não poderiam ser analisados

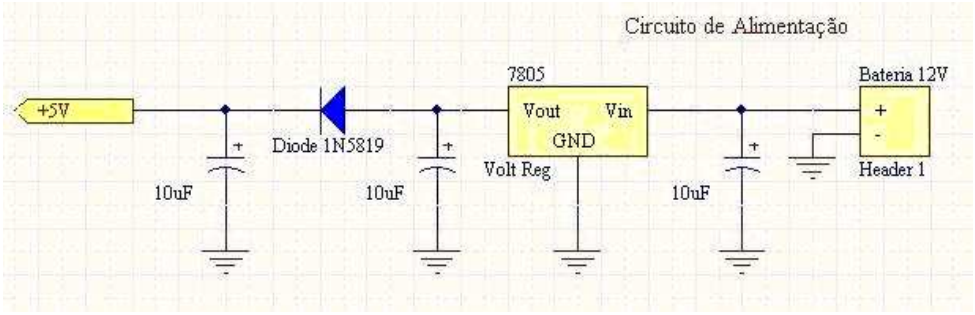

Figura 2.8: Circuito de alimentação.

através do LCD, dado sua escassez de rapidez e memória. Assim, para uma análise mais eficiente, os dados foram enviados para um *host* para armazenamento, análise, calibração e estudo das medições. A aquisição de dados também foi utilizada para armazenar a resposta dos motores no tempo, e assim, identificar o modelo dinâmico dos mesmos. Esse trabalho com gráficos, planilhas e identificação foi feito com o auxílio do *software* MATLAB.

Observa-se que a comunicação com o EEEPC é feita através de um canal conversor serial-USB. Assim, caso seja necessário transportar dados por relativas longas distâncias, aproveita-se o canal USB em detrimento ao de RS-232, i.e., utiliza-se um cabo RS-232 curto e um cabo USB mais longo. Porém, geralmente, dados irão trafegar uma distância pequena dentro do robô e, portanto, distâncias não causarão transtornos.

O padrão RS-232, é um protocolo de comunicação assíncrono, e teve seus parâmetros configurados da seguinte forma: 115200bps para velocidade de transmissão (*baud rate*), 8 bits de dados sem paridade, 1 bit de parada e ausência de *handshaking*.

Porém, essa interface trabalha com nível lógico "1" para tensões no intervalo -3V e -12V, e nível lógico "0" no intervalo 3V e 12V. O periférico do ARM responsável por essa comunicação serial, o USART 0 (*Universal Synchronous Asynchrinous Receiver Transmitter*), trabalha com tensão TTL. Assim foi necessário o uso de um circuito de conversão nível TTL/RS-232. Optou-se pelo circuito integrado MAX232, que trabalha com alimentação TTL e é bastante usado em aplicações deste tipo. O circuito dessa interface pode ser visto na figura 2.9.

Na figura 2.9 é importante observar que os pinos 11 e 12 do MAX232 são ligados respectivamente aos pinos 21 (Tx) e 22 (Rx) do ARM. O cabo serial ligado ao computador é composto de 3 fios (RX, TX *e* GND). Os pinos 2 e 3 do conector DB9 são conectados através do cabo serial respectivamente aos pinos 14 e 13 do MAX232. O pino 5 (GND) do conector é ligado à fonte de alimentação da placa controladora de acessos. Os capacitores eletrolíticos são utilizados para configurar o funcionamento correto do MAX232.

Essa mesma interface foi adotada para comunicação (através de um protocolo robusto e que será apresentado no capítulo 3) entre ARM e EEEPC. Como o EEEPC não possuía conector para troca de dados RS-232, usou-se um cabo conversor Serial/USB. O fato de o cabo possuir padrão USB, acabou validando o uso do padrão RS-232, pois, uma vez que o formato USB apresenta boa imunidade a ruídos, é dispensável o uso de um padrão como o RS-485, que permite maiores distâncias e maior imunidade a ruídos.

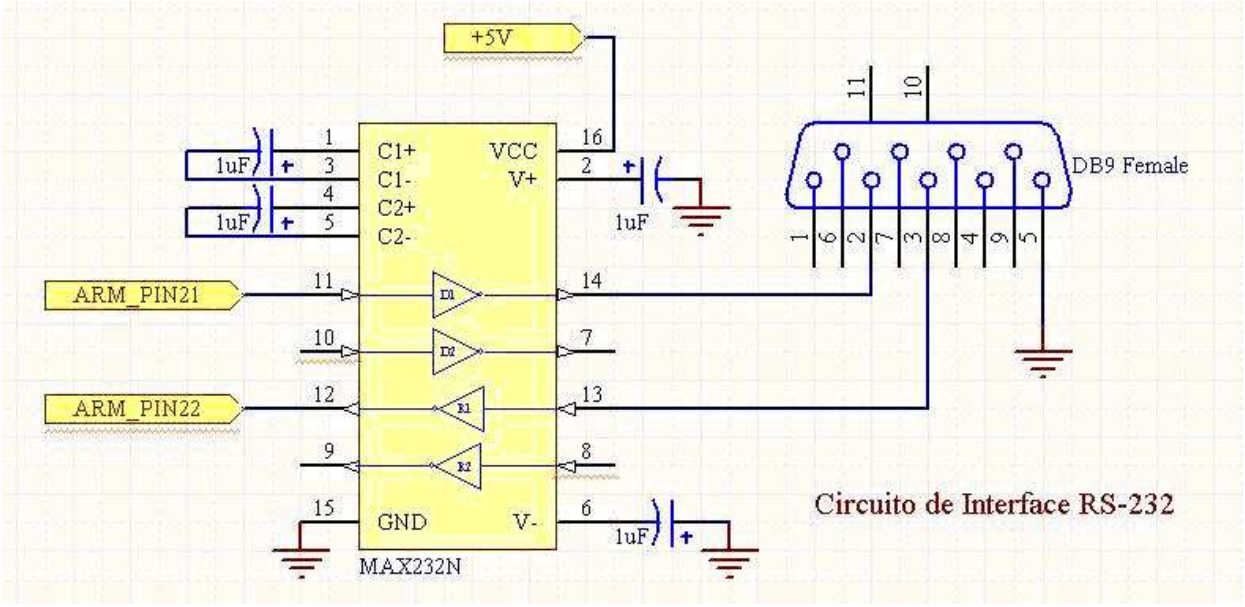

Figura 2.9: Circuito para comunicação serial RS-232.

2.5 Acionamento

Os motores de corrente contínua são controlados por circuitos com arquitetura ponte-H. Utilizou-se o CI dedicado LMD18200 da *National Semiconductor*, que já é formado por todos os elementos para a elaboração de um acionamento PWM. Este circuito integrado é fornecido num invólucro SIL de alta potência, de 11 pinos, montado num dissipador de calor, cujo diagrama em blocos apresentado na figura 2.10.

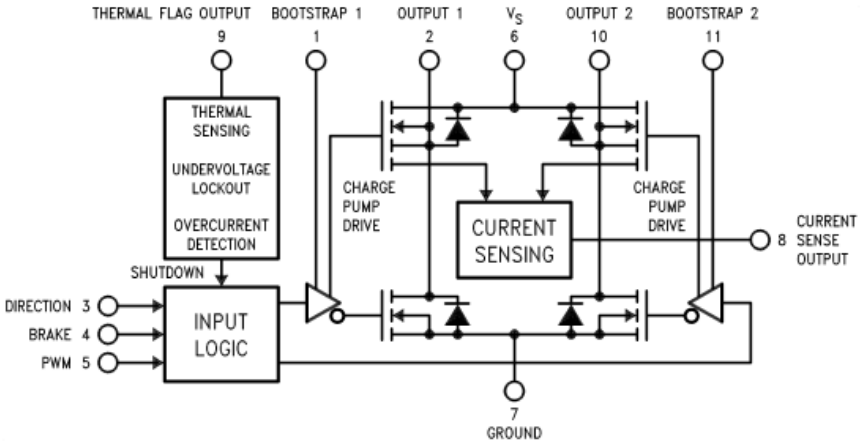

Figura 2.10: Diagrama de blocos da ponte-H LMD18200.

A ponte-H completa é capaz de controlar cargas de até 3A com tensão máxima de 55V. No caso dos robôs, a bateria utilizada fornece 12V. Entre os recursos disponibilizados pelo circuito notamos a saída *Vsense*(6). Esse terminal é usado para que se saiba quando o motor está carregado ou rodando sem carga (pela corrente circulante) para então compensar sua velocidade por circuitos externos ou corte da alimentação, caso ele tenha sido travado, evitando problemas. A tensão nesta

saída sería de 0,1V para cada ampere de corrente. A entrada *BRAKE* serve para parar o motor. Quando ela é acionada, a corrente no motor é cortada e seus enrolamentos são ligados em curto. Assim, a tensão que é gerada quando o motor continua girando, sendo colocada em curto, cria uma força de frenagem que paralisa rapidamente o movimento do motor. Cada sinal *BRAKE*, que decidimos tratar como sinal *ENABLE* (habilitador), foi gerado pelo ARM e disponilizado nos pinos PA0 e PA1 no caso da plataforma diferencial, e PA0, PA1 e PA2 para a plataforma omnidirecional. Observa-se aqui a compatibilidade de pinos entre as duas plataformas. Esse similaridade em *hardware* implicará em facilidades de programação das plataformas e compatibilidade de códigos.

A saída *THERMAL FLAG* fornece um sinal caso o circuito esteja excessivamente aquecido. Caso o sistema se encontre sobreaquecido, o nível da saída no pino 9 torna-se baixo. Para sinalizar esse evento, utilizou-se um *led* ligado á alimentação 12V. Dessa forma a queda de nível no pino 9 permite a passagem de corrente no *led* sinalizando o evento.

A entrada PWM controla a velocidade do motor através do processo *Pulse Width Modulation* (Modulação de largura de pulso). Utilizou-se a configuração PWM anti-fase, em que uma único sinal com ciclo de trabalho variável tem codificado tanto a direção quanto a amplitude de informação. Com um ciclo de 50% imposto ao sinal PWM o motor fica parado, uma vez que o valor líquido de tensão (integradas durante um período) entregue para o motor é nulo. Deste modo, o sinal de PWM é encaminhado à entrada de direção e a entrada PWM é fixada em alto. A escolha dessa configuração foi importante para economia de um pino do ARM que tornou-se disponível para outros recursos. O circuito resultante de acionamento pode ser conferido na figura 2.11.

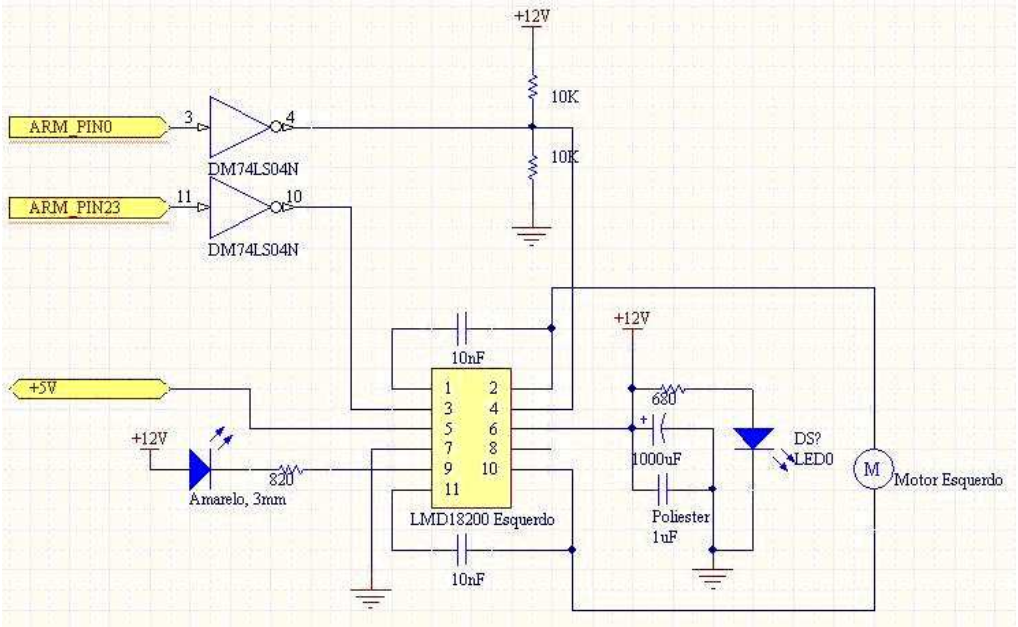

Figura 2.11: Circuito associado a ponte-H LMD18200.

Os sinais de PWM e *BREAK* (*ENABLE*) gerados pelo ARM foram transmitidos por *buffers* do tipo portas inversoras TTL 7404. O *buffer* realiza a conversão de nível entre circuitos ARM e lógica TTL da ponte-H.

2.6 Codificação Incremental Óptica

As rotinas de odometria e controle trabalham com informações da velocidade e direção de rotação de cada motor. O *encoder* óptico foi escolhido para fornecer ao controlador do robô essa informação. Dessa forma, ele funciona como uma unidade de realimentação (*feedback unit*) informando as posições atuais das rodas do robô, que comparadas com posições desejadas permite que os movimentos sejam planejados.

O modelo escolhido foi E4P-300-079-HT, que apresenta fácil encaixe ao motor utilizado e possui alimentação TTL que facilita o trabalho com o ARM. Fornece 300 pulsos/rotação em dois canais A e B. O ARM, por sua vez, receberá 9000 pulsos/rotação, devido a instalação de uma redução de 30 entre o *encoder* e o motor. O sinais apresentados em A e B são defasados de 90^o conforme mostrado na figura 2.12. Mas essa medida de defasagem pode apresentar um desvio nominal cujo cálculo é feito a partir dos parâmetros P e ϕ, em que P corresponde a meio período do sinal A ou B, e ϕ é a medida de defasagem entre os sinais A e B, de acordo com a equação 2.1.

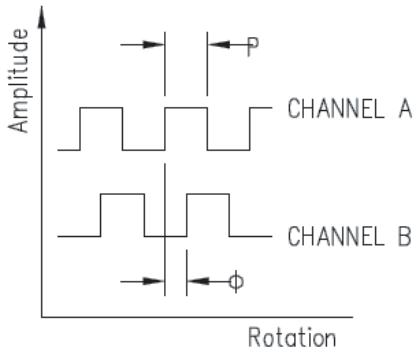

Figura 2.12: Sinais típicos das saída A e B do encoder óptico.

Para o cálculo da velocidade, o sinal do canal A (poderia ter sido escolhido o canal B), que é um sinal digital, é passado diretamente a um dos pinos do ARM que possui acesso a um dos periféricos de contagem (o ARM possui três periféricos contadores TC0, TC1 e TC2). A cada 10 milisegundos essa contagem é passada ao *software* que calcula a relação entre o número de pulsos nesse intervalo de tempo e passa para as rotinas de controle e odometria. O trabalho realizado pelo *software* é apresentado com mais detalhes no capítulo 3.

$$\Delta\phi = |90^o - \frac{\phi}{P} \cdot 180^o| \leq 45^o \tag{2.1}$$

Para determinar a direção de rotação do motor, utilizou-se a característica de defasagem dos sinais A e B. Para evitar que mais essa tarefa tomasse tempo de processamento do ARM, foi desenvolvido um *hardware* capaz de determinar a direção de rotação. Utilizou-se *flip-flops* tipo D disponíveis no circuito integrado SN74LS74AN, em um circuito cuja arquitetura é ilustrada na figura 2.13.

Um dos sinais do *encoder* é enviado como *clock* do *flip-flop*. Assim as bordas de subida do sinal do *clock* ativam a passagem do sinal de entrada D para a saída Q. Assim, se no momento de transição positiva (ou negativa) do sinal do relógio o sinal de entrada em D estiver em nível

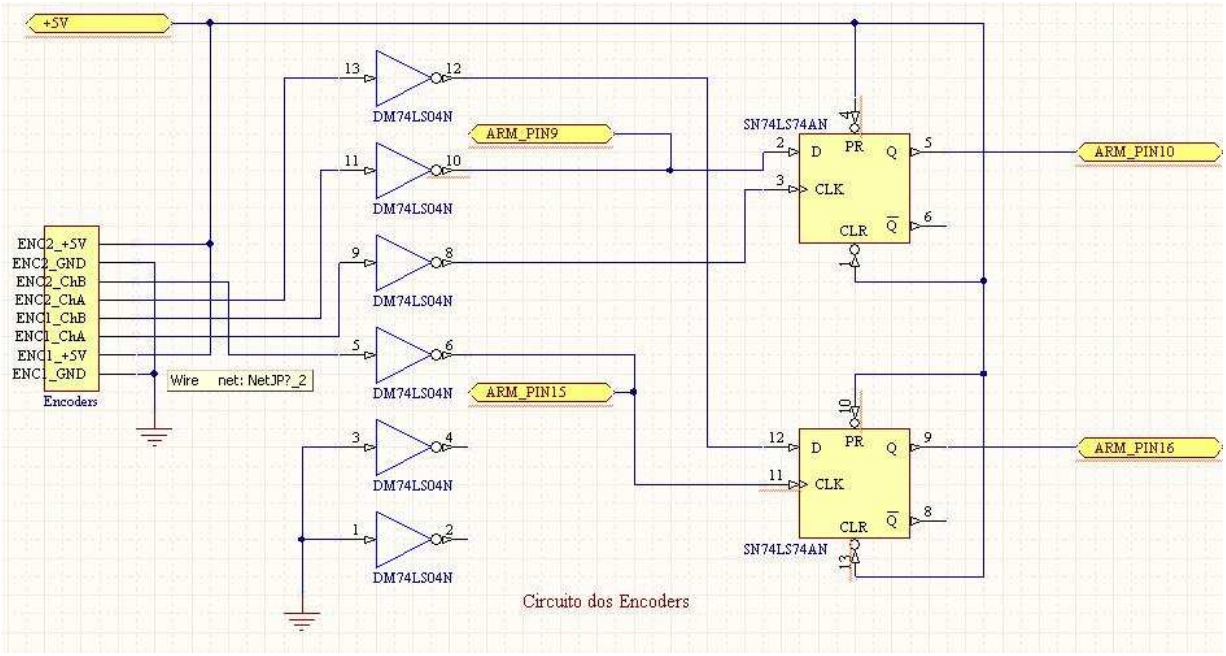

Figura 2.13: Circuito dos sensores ópticos para medição de velocidade dos motores.

alto, a saída Q estará em nível alto. Se no momento de ativação (*clock*) o sinal de entrada D estiver em nível baixo, a saída Q estará em nível baixo. Assim, os níveis alto e baixo da saída do *flip-flop* determinam direções de rotação diferentes. Então, os sinais de velocidade (trem de pulsos) e direção de rotação são passados por *buffers* do tipo portas inversoras TTL 7404 para que recebam um reforço, e então seguem à entradas do microcontrolador.

2.7 Sensor Girômetro

Para auxiliar no trabalho de localização dos robôs tornou-se interessante incorporar um sensor girômetro a cada plataforma, o qual abre a possibilidade de desenvolvimento de girodometria.

O sensor girômetro, modelo ADXRS300 (vide figura 2.14), utiliza o princípio de um giroscópio ressonante, e fornece a velocidade angular do robô em torno de um eixo, no caso o eixo perpendicular ao plano X-Y que sustenta a plataforma. Foi projetado para medir uma faixa de valores de $-300^o/s$ a $+300^o/s$. Essa medida é disponibilizada no terminal de saída nomeado $RATEOUT$ na forma de uma tensão incrementalmente proporcional à velocidade de rotação, que varia entre 0V e tensão de alimentação do sensor V_s (em torno de 4,7V gerada pela circuito de alimentação apresentado na seção 2.3). É importante ressaltar que a tensão $V_s/2$ na saída do sensor corresponde a tensão na situação de robô sem rotação. Na figura 2.15, podemos observar o eixo de medição da velocidade e um gráfico com a relação tensão de saída e velocidade.

O sensor apresentado disponibiliza recursos como a temperatura ambiente em um terminal nomeado $TEMP$ por meio de um sensor interno. A temperatura fornecida como uma tensão proporcional, pode ser usada para calibração. Para auxiliar no trabalho de calibração, uma tensão fixa de referência de 2,5V também é disponibilizada.

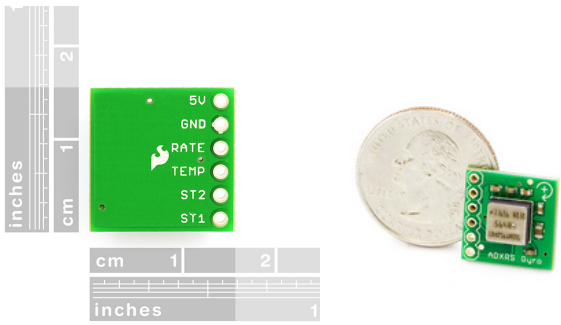

Figura 2.14: Sensor girômetro modelo ADXRS300.

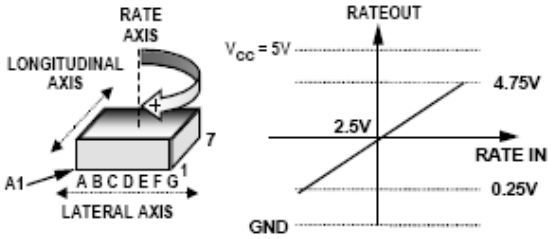

Figura 2.15: Curva característica do sensor de rotação da plataforma.

O *software* do robô possui uma rotina de girodometria. Essa rotina solicita periodicamente a medida de giro fornecida pela saída *RATEOUT*. Por ser um sinal analógico, a tensão *rate* é passada a uma entrada A/D com resolução 10 bits do ARM para que a rotina de girodometria possa utilizá-la. Neste ponto, surge uma limitação ao projeto: a entrada A/D do ARM suporta sinais até 3,3V e a tensão de saída do giroscópio RATEOUT pode alcançar 4,7V. Uma solução viável foi a construção de um divisor resistivo de proteção à entrada A/D do ARM, conforme apresentado na figura 2.16 que apresenta o *hardware* relacionado ao girômetro. Nesta mesma figura 2.16 é interessante notar a presença de um capacitor junto ao divisor resistivo de forma a criar um filtro RC com constante de tempo aproximado de 300ns. Esse filtro retira ruídos do sinal e também elimina uma fração de frequências acima do dobro da frequência de amostragem, no caso, a cada 10ms a girodometria solicitará esses dados. Aspectos teóricos e de implementação da girodometria serão apresentados na seção 3.2.5. Ainda, observa-se que o filtro se encontra na saída do multiplexador. Dado sua dinâmica, o microcontrolador ARM deve sempre esperar aproximadamente 5 contantes de tempo, i.e., 1,5 milissegundos, para realizar uma nova medição logo após um chaveamento do multiplexador. Conforme será visto no capítulo 3, o *software* do ARM sempre espera 2 milissegundos após um chaveamento.

Diante de limitações de recursos da plataforma ARM tornou-se interessante o uso da tarefa de multiplexação para que apenas uma porta A/D fosse usada para recepção do sinal 2,5V, o valor *RATEOUT* e a medida *TEMP*. Atualmente, o *software* não utiliza informação de temperatura. Porém o *hardware* já prevê esta possibilidade de aprimoração da medição. Essa multiplexação foi realizada pelo circuito integrado CD4052BCN, que possui capacidade de selecionar 4 canais analógicos a partir de dois sinais chamados *MuxGyroA* e *MuxGyroB*. Para gerar esses sinais de controle são usados os pinos PA7 e PA11 do ARM, já o canal A/D usado para recepção dos

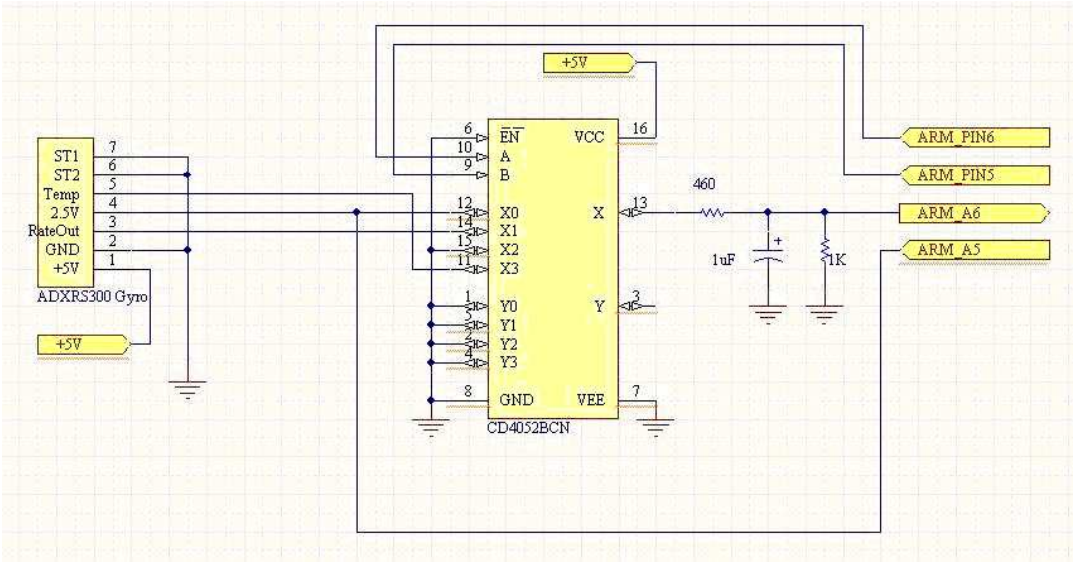

Figura 2.16: Circuito relacionado ao girômetro.

sinais provenientes do girômetro foi o AD5. Além disso, a saída de 2,5V só é necessária para o cálculo do ganho resistivo. Conciliando as informações anteriores mostrou-se interessante a idéia de multiplexação dos dados rate, 2,5V e temperatura. Utilizando um multiplexador CD4052BCN e dois sinais de controle do microcontrolador, limitou-se o uso de entradas A/D a duas: uma para a saída multiplexada e outra para a tensão de 2,5V do girômetro que funciona como referência para cálculo do ganho resistivo pelo microcontrolador. A idéia apresentada explica o projeto ilustrado pelo esquemático da figura 2.16.

Ao passar pela chave analógica, cada sinal percorre um caminho no CI de acordo com a posição da chave. Esse caminho oferece uma resistência ao sinal. Mas, como foi mencionado, a saída do CI multiplexador deve passar por um circuito com ganho resistivo e filtro RC. Nota-se, portanto, que a resistência interna da chave fica em série com o ganho resistivo e acaba por diminuí-lo, fazendo com que o sinal disponível ao ARM torne-se menor. Essa resistência também se associa ao filtro de saída, tornando a dinâmica do sistema mais lenta. Então, a presença dessas resistências internas de caminho percorrido pelo sinal de acordo com a posição da chave, influencia o projeto realizado para recepção do sinal na entrada A/D do ARM. Essa alteração no projeto, leva a uma mudança do ganho de calibração. Como a calibração é realizada, e como essa resistência interna influência nessa tarefa serão apresentadas na seção 3.2.4.

As medições do sensor de temperatura não são usadas para calibração por não estarmos interessados em tanta precisão atualmente.

2.8 Sensores Infravermelho

Os sensores infravermelhos foram os últimos sensores a serem incorporados às plataformas. Fazem parte da composição exteroceptiva dos robôs. Seu trabalho é realizar medições de distâncias na faixa de 20cm a 150cm segundo o fabricante. Nesta seção será mostrado o trabalho de fixação dos sensores às plataformas, a apresentação do modelo dos sensores e seu princípio de funcionamento,

com o circuito desenvolvido para chaveamento liga/desliga de suprimento de energia, e o processo de conexão entre os sensores e o ARM. Como as medidas geradas pelos sensores serão usadas, e ainda como acessá-las, é um assunto que será apresentado no capítulo 3.

Os sensores modelo GP2Y0A02YK fabricado pela SHARP geram uma tensão de saída analógica correspondente à distância. Possui em um mesmo invólucro um *led* emissor e um receptor, além de um circuito que converte o sinal recebido em uma tensão correspondente à distância do objeto. Para realizar as medições, o sensor utiliza-se da técnica de triangulação.

No método da triangulação, um feixe infravermelho é emitido e ilumina um objeto. Esse feixe atinge um objeto e é refletido por este. Depois, esse atinge o detector presente no sensor. A trajetória descrita pela radiação infravermelha forma um triângulo com vértices no emissor, no ponto de reflexão e no detector, daí o nome do método. Um cálculo então é feito baseado no ângulo incidente do raio de luz que determina a distância percorrida.

O dispositivo emite feixes de infravermelho através de pulsos que possuem comprimento de onda de $(850 \pm 70)nm$. E as leituras dos sinais recebidos que determinam a distância percorrida são atualizadas a uma taxa de 24Hz.

O detector é relativamente insensível a iluminação ambiente, bem como a refletividade do objeto a ser detectado. Dessa forma, é possível que o dispositivo detecte objetos relativamente escuros em plena luz solar.

A tensão de saída é uma função não-linear da distância entre o objeto e o receptor. A curva na figura 2.17 foi retirada da documentação do sensor SHARP, e foi tomada considerando um objeto branco com um índice de 90% de reflectividade.

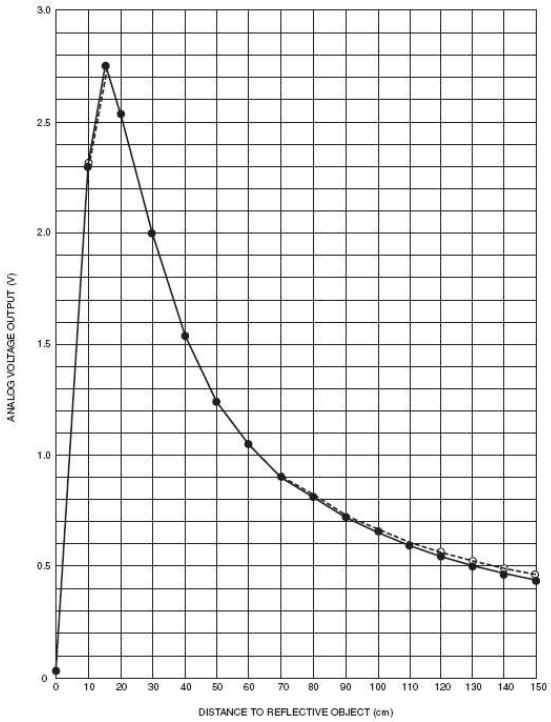

Figura 2.17: Curva característica da tensão de saída não linear dos sensores infravermelhos.

A figura 2.18 mostra a interface elétrica do sensor e a figura 2.19 mostra o circuito desenvolvido com os sensores para o projeto das plataformas. Como existem seis sensores infravermelhos, a placa de processamento já existente não tinha capacidade de prover alimentação e receber todos esses sinais. Optou-se pela construção de uma placa com um circuito de regulação de 12V para 5V.

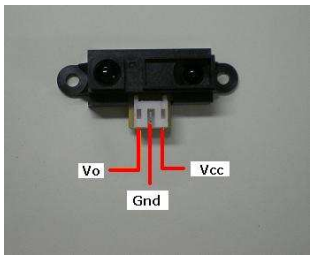

Figura 2.18: Interface elétrica do sensor SHARP GP2Y0A02YK.

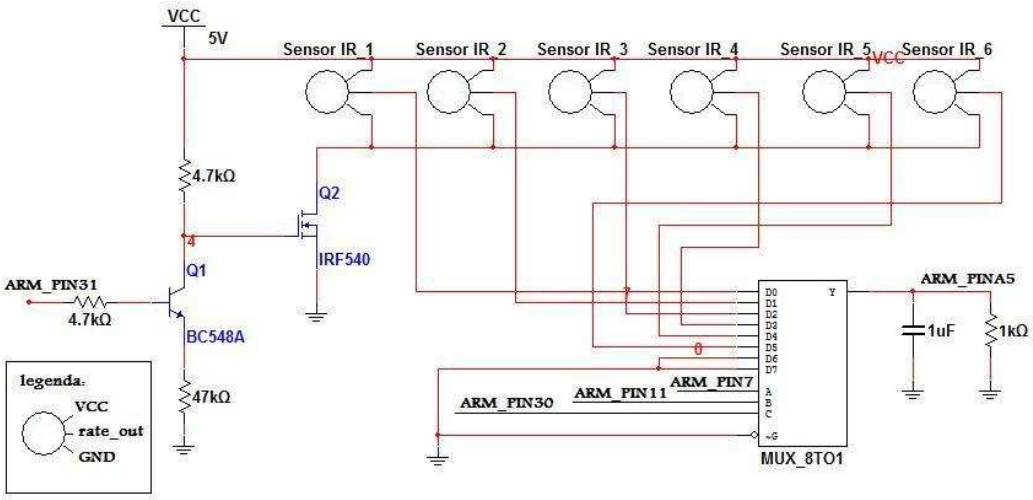

Figura 2.19: Circuito elétrico dos sensores SHARP GP2Y0A02YK.

Pelo gráfico da figura 2.17, nota-se que os objetos mais próximos do que cerca de 15 centímetros podem ser vistos como objetos em distâncias mais longas. Esta ambiguidade precisa ser considerada quando se estima obstáculos próximos ao robô.

Os seis sensores foram distribuídos em cada plataforma conforme apresentado na seção 2.2. Cada sensor foi fixado no meio de cada lado da base da plataforma, através de barras de alumínio com formato 'L'.

Por ser um sensor com saída analógica, o processamento dos sinais dos sensores só é possível a partir do uso de entradas A/D do microcontrolador. Como o ARM possui entradas A/D limitadas, e muitas delas já foram utilizadas para sinais de sensores, foi necessário que as medidas de cada sensor passasse por uma chave analógica do CI CD4051 de multiplexação para terem acesso à uma única entrada A/D do ARM.

Assim como no trabalho com o girômetro, as medidas fornecidas pelos sensores infravermelhos passam por um divisor resistivo e filtro RC, e apresentam os mesmos problemas de implementação relacionados a resistência interna de chaveamento .

2.9 ARM

Ambas as plataformas possuem uma unidade microcontroladora AT91SAM7S64, baseada na arquitetura ARM7TDMI. Essa unidade microcontroladora é utilizada como interface entre o EEEPC e recursos de hardware disponíveis, além de ser responsável pelas tarefas periódicas de odometria, girodometria e controle. Assim o EEEPC poderá solicitar ao ARM, via comunicação serial RS-232, dados de sensores e postura do robô. O EEEPC também fornece sinais de referência para o controle de velocidade das rodas. Para desempenhar as tarefas apresentadas, foram utilizados diversos periféricos disponíveis no ARM que serão apresentados nesta seção e nas seções seguintes deste capítulo, a medida que forem requisitados pelo *hardware* em questão.

O ARM utilizado está disponível em uma *header board* da Sparkfun Eletronics[1]. Essa *header board* apresenta alguns componentes e elementos que facilitam o acesso a alguns periféricos do ARM como:

- Conector USB;
- Circuito para RESET;
- Botão para RESET;
- Conector JTAG padrão com *layout* 2x10pinos para programação e *debugging* do ARM;
- *Led* indicador de *status*;
- *Jumper* TST para carregar o *software* SAM-BA;
- Circuito *OnBoard* regulador de tensão 3,3V com 800mA de corrente máxima;
- *Power Supply* através da USB ou pelo pino *header*;
- *Led* indicador alimentação;
- Filtro capacitivo para fonte de alimentação;
- *Socket* para cristal oscilador e cristal de 18,432MHz;
- Pinos *headers* extensores para todas as portas do microcontrolador, RST e fonte de alimentação.

A figura 2.20 mostra a *header board* e permite a visualização de alguns dos recursos citados.

Para programação e depuração do microcontrolador ARM, utilizou-se uma gravadora JTAG MACRAIGOR WIGGLER COMPATIBLE de 20 pinos. Foi adquirida uma PCB com o circuito desta, e a montagem foi realizada no laboratório LARA, vide figura 2.21.

O ARM possui inúmeros periféricos conforme apresentado no diagrama de blocos do AT91SAM7S64 disponível no apêndice H1. Porém, neste capítulo apenas serão apresentados aqueles que fazem interface com o *hardware* da plataforma. Essa apresentação somente mostra o recurso, com que

[1] http://www.sparkfun.com/

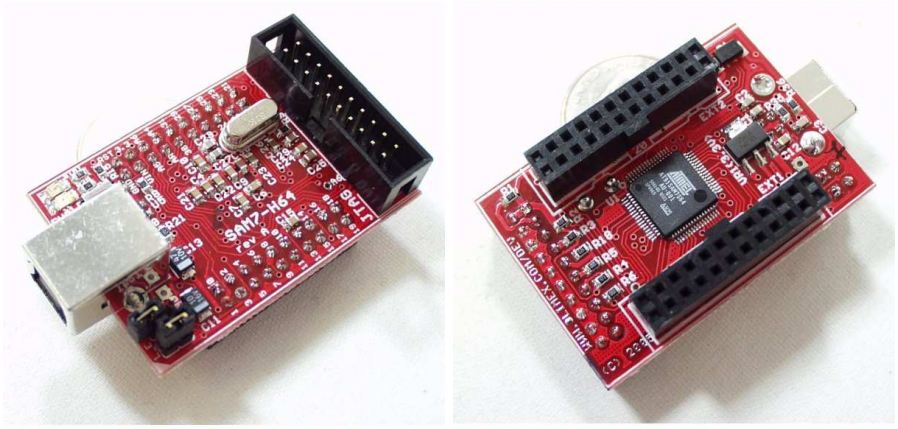

Figura 2.20: Vista superior e inferior da *header board* do ARM.

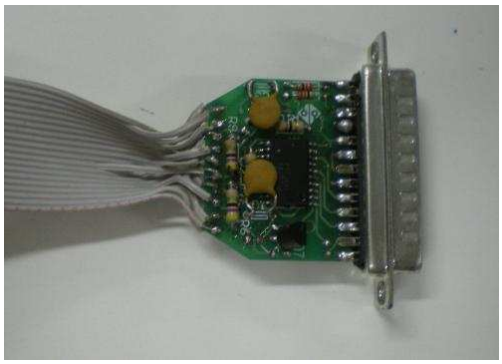

Figura 2.21: Gravadora JTAG MACRAIGOR WIGGLER COMPATIBLE.

elemento de *hardware* se relaciona e a tarefa que desempenha, sem apresentar como será usado via *software*, o que será feito apenas no capítulo 3.

Para tarefa de acionamento, o microcontrolador disponibiliza 4 canais de PWM de 16 bits gerados por uma unidade controladora chamada PWMC. A plataforma diferencial utiliza apenas dois destes canais para acionar seus dois motores, e a omnidirecional usa três. Dessa forma ainda resta uma disponível para tarefa de acionamento ou outra tarefa que utilize Modulação por Largura de Pulso.

A comunicação serial usada para depuração, para troca de dados entre o ARM e o EEEPC e para troca de dados entre ARM e MATLAB em determinados modos de operação (a serem vistos na seção 3.2.2) é garantida pela USART0 do microcontrolador que provê comunicação *full duplex*[2]. O ARM ainda disponibiliza outro periférico para comunicação, a USART1, que não foi utilizado no projeto.

Para realizar a contagem periódica do número de pulsos gerados por cada *encoder* óptico foram usados os módulos contadores TC0 e TC1 na plataforma diferencial e TC0, TC1 e TC2 na omnidirecional. O ARM disponibiliza apenas estes três contadores, exatamente o número de contadores que foi necessário ao desenvolvimento do projeto. O funcionamento operacional do *encoder* óptico utilizado neste projeto e seu circuito associado foram apresentados na seção 2.6.

[2]O modo *full-duplex* permite que cada nó da rede envie e receba dados simultaneamente.

Entre os sensores presentes no projeto, apenas os sensores infravermelho e girômetro possuem saída analógica e por isso utilizam conversores A/D do próprio microcontrolador para interfacear ao ARM. O girômetro possui dois sinais que são passados ao microcontrolador, a saída de uma chave analógica multiplexadora que pode ser a saída *RATEOUT*, que é tensão de saída proporcional a velocidade de giro do robô, a tensão gerada pelo sensor de temperatura interno *TEMP* que não é usada no projeto atual, mas que fica disponibilizada caso exista no futuro a necessidade, e a tensão de saída fixa em 2,5V usada para identificar a tensão de robô sem rotação. O segundo sinal passado é a mesma tensão fixa 2,5V neste caso usada para cálculo do ganho do divisor resistivo e para o trabalho de calibração do sensor, realizada toda vez que a plataforma é ligada (alimentação geral).

Os sensores infravermelhos também precisam passar sua tensão analógica para uma entrada A/D do ARM, para que este possa realizar o processamento. Porém, não existe a necessidade de passar simultaneamente todas as saídas ao microcontrolador. Uma chave multiplexadora também é usada para que, por *software*, os dados de cada sensor sejam passados, um a um, a apenas uma entrada A/D. A aquisição dos dados dos sensores infravermelhos não é uma tarefa de alta prioridade, e não está presente entre as tarefas periódicas. O usuário é responsável por solicitar esses dados e usá-los conforme desejar.

O ARM possui 8 entradas A/D e no estágio atual do projeto apenas três dessas entradas são usadas. Essas inúmeras entradas dão flexibilidade para o projeto crescer e incorporar mais elementos. A mesma flexibilidade também é garantida pela disponibilidade de mais uma USART e um canal PWM. O número de contadores suficientes aliado à flexibilidade de crescimento do projeto foram importantes para a escolha deste microcontrolador.

2.10 EEEPC

De acordo com o projeto apresentado para esse trabalho de pesquisa, as plataformas móveis devem ser capazes de navegar pelo ambiente via comandos passados por um computador remoto. Esses comandos normalmente são respostas a dados sensoriais do robô. Assim, há necessidade de troca de dados e comandos via rede sem fio, de forma a permitir a livre locomoção da plataforma com supervisão à distância. O microcontrolador ARM não possui interface para comunicação em rede, além de apresentar recursos de processamento e memória limitados. Foi então incorporado à cada plataforma um *NetBook* EEEPC da ASUS, visualizado na figura 2.22. Esse *NetBook* também permitiu a instalação de um sistema operacional, o que não seria possível no ARM devido às limitações citadas. Com o sistema operacional, foi possível a instalação de um *software* que fornece ferramentas para prover a comunicação em rede. Este *software* e o trabalho realizado para essa comunicação serão apresentados no capítulo 3.

A presença de uma *webcam* para captura de imagens do ambiente, trouxe a necessidade de um sistema que permitisse conexão USB para receber dados, e um processamento local desses dados. Esse processamento não poderia ser realizado no ARM por demandar bastante capacidade de processamento, além de memória.

Os recursos de *hardware* para processamento, comunicação e memória disponíveis no EEEPC,

Figura 2.22: *Netbook* EEEPC.

mostraram-se suficientes para desempenhar as tarefas apresentadas, além de serem pequenos e leves. São recursos disponíveis no EEEPC:

-Monitor: LCD 7" polegadas WVGA (800x480);

-Processador e *Chipset*: Intel Celeron-M ULV 353 a 900MHz, Intel 915GM series;

-Sistema Operacional: Linux (pré-instalado);

-Interfaces de Rede: *Wireless* LAN, 802.11 b/g;

-Memória: 512 MB (DDR2);

-Armazenamento: 4GB SSD (disco de estado sólido);

-Áudio: áudio de alta definição, com alto-falantes estéreo e microfone;

-Bateria: 4 células, 4400 mAh, com autonomia estimada em 2,8 horas;

-Peso: 0.92 kg

-3 entradas USB;

- *Webcam* integrada;

-Entre outros.

Conforme já apresentado, o EEEPC se comunica via serial RS-232 com o ARM para troca de dados e envio de comandos. Porém o EEEPC não possui uma entrada serial para comunicação. Foi adquirido um cabo conversão USB/Serial que permite o uso de uma das portas USB disponíveis para essa comunicação. A figura 2.23 apresenta o cabo conversor. Outra porta USB foi usada para conectar a *webcam*.

A disponibilidade de um monitor LCD mostrou-se importante para depuração de dados, além de permitir ajuste e inserção de parâmetros no *software*. Também serve como interface para trabalhar com o sistema operacional e *softwares* instalados.

A capacidade de autonomia de 2,8 horas é importante para que a plataforma tenha liberdade de navegação por longos períodos dando-lhe autonomia. Autonomia também possibilitada pelo

Figura 2.23: Cabo conversor USB/SERIAL.

fato de o computador ser bastante leve e pequeno. Por ser mais leve e menor, a carga sobre os motores é menor, o que permite que a corrente demandada da bateria seja menor, dando mais tempo de vida à esta bateria, contribuindo assim para a autonomia da plataforma.

O elementos apresentados e sua utilidade contribuem para a flexibilidade do projeto e foram muito importantes para a escolha deste computador como unidade processadora.

Capítulo 3

Software

Neste capítulo serão consideradas as arquiteturas de software utilizadas nas unidades de processamento do robô. Após ter lido esse capítulo, o leitor terá um entendimento completo sobre como o robô interpreta por software os sinais físicos do hardware do capítulo anterior e os utiliza para cumprir seus objetivos.

3.1 Aspectos Gerais

O escopo no qual este projeto se insere é o contexto da robótica de cooperação. As plataformas aqui construídas possuem o objetivo de auxiliar experimentos e estudos em robótica cooperativa. Portanto, postulou-se neste projeto a necessidade de disponibilizar os comandos de acionamento dos motores e os dados de cada robô em uma rede TCP/IP, de forma que o conhecimento da estrutura organizacional do conjunto de robôs e as capacidades de cada estejam disponíveis para permitir aos mesmos alterar *on-the-fly* suas estratégias individuais de forma a atingir um objetivo global de forma mais eficaz. Utilizou-se neste trabalho um software que fornece ferramentas para prover tal comunicação: o PLAYER[1].

De acordo com o website oficial do software, o PLAYER é um servidor de rede grátis e *open-source* sob a licença pública GNU para controle de robôs rodado em Linux. Rodando em um robô, o PLAYER provê uma interface padronizada limpa e simples para os sensores e atuadores do mesmo sobre uma rede TCP/IP. Um programa cliente conversa com o PLAYER sobre um socket TCP, lendo dados dos sensores e escrevendo comandos para os atuadores, além de configurar dispositivos *on-the-fly*. O PLAYER suporta atualmente uma grande variedade de hardware, inclusive a famosa família ActivMedia Pioneer[2] 2. A figura 3.1 ilustra duas plataformas compatíveis com o software PLAYER. À direita, uma das plataformas deste trabalho. Á esquerda, a plataforma Pioneer comercial. Para o PLAYER rodar em um robô, o servidor tem de possuir o driver para PLAYER do robô. No caso deste projeto, o driver teve de ser implementado pelos autores.

O software PLAYER estará instalado então no Asus EEEPC. O mesmo software poderia estar instalado no próprio microcontrolador ARM, desde que este possuísse o sistema Linux. Porém

[1]Software pode ser obtido em *http://playerstage.sourgeforge.net/*

[2]Mais informações sobre o Pioneer podem ser obtidas em *http://www.activrobots.com/*

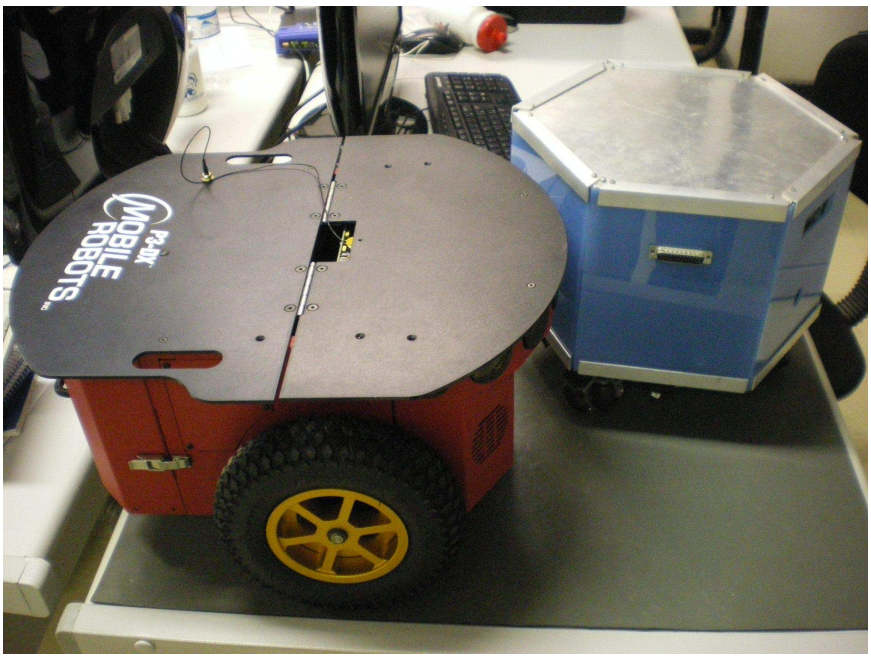

Figura 3.1: Exemplos de hardware compatível com o software PLAYER.

seria necessário um microcontrolador mais robusto com capacidade de memória e processamento maior, além de uma maior quantidade de periféricos (por exemplo, o AT91SAM7S64 não possui interface de rede). Ainda, caso o EEEPC não estivesse presente, o processamento local de imagens captadas pela webcam não seria viável devido ao limitado poder de processamento e memória. Mais informações sobre o PLAYER, e como os arquivos associados ao PLAYER são implementados para atingir os objetivos deste trabalho, vide seção 3.4.

O servidor PLAYER instalado no Asus EEEPC se comunica com o robô pela interface RS-232 do ARM. Essa comunicação é implementada no driver do PLAYER do robô com o auxílio de uma API em linguagem C desenvolvida pelos autores deste trabalho. As bibliotecas de desenvolvimento dessa API permitem a qualquer programador escrever um programa do EEEPC que leia os dados dos robôs e acione seus atuadores, com a necessidade apenas de um breve e superficial conhecimento do hardware do mesmo. Portanto, o PLAYER não é o único ambiente de desenvolvimento para estas plataformas. Caso algum dia seja necessário abandonar o PLAYER e desenvolver aplicativos com outra interface, o tempo de desenvolvimento será mínimo. Para mais informações a respeito desta API, e como a comunicação RS-232 é implementada em software, vide seção 3.3 e seção 3.2.3.

A API desenvolvida implementa então um enlace de comunicação entre o EEEPC e o ARM. O ARM utilizado no nosso projeto não foi contemplado com nenhum sistema operacional. Os autores julgaram que nessa aplicação não existe a necessidade de um gerenciador de recursos, ou de uma máquina virtual de qualquer tipo. Portanto, o software tem prioridade máxima sobre os recursos e periféricos do microcontrolador. Por se tratar de um protótipo, o software do ARM foi divido em módulos para facilitar sua modificação, para suportar a reusabilidade de código e para simplificar a leitura do mesmo por indivíduos externos ou novatos ao projeto. O ARM é o responsável por tarefas de mais baixo nível, tais como controle de velocidade das rodas, cálculo da odometria do robô, leitura bruta dos sensores de rotação e alcance, etc. Tarefas críticas que necessitam de um período

de tempo de acionamento constante para acontecer devem estar no ARM, afinal, o PLAYER não garante periodicidade de eventos com alta precisão. Mais informações acerca o software do ARM, e como a modularização é implementada para atingir os objetivos deste trabalho, vide seção 3.2.

Ainda, existem modos de operação do robô que podem prover dados que exibem o desempenho do mesmo, os quais podem ser analisados com o auxílio do software MATLAB. Para mais informações acerca dos modos de operação do robô e como esses dados podem ser coletados de forma a se calibrar os sensores, ou sintonizar os controladores digitais PI de velocidade das rodas, vide seção 3.5.

Reforçando as idéias vistas até então, os parágrafos anteriores são ilustrados pela figura 3.2, onde as bibliotecas de desenvolvimento associadas a cada nível são destacadas. O nível mais baixo corresponde ao microcontrolador ARM e seu código único *main.elf* escrito com base nos vários módulos explicitados: *motor.h*, *comm.h*, etc. programados pelos autores utilizando a biblioteca para desenvolvimento ARM em C[3]. O software *main.elf* cumpre os seguintes objetivos:

- Controle proporcional-integral da velocidade das rodas;
- Cálculos de Odometria;
- Coleta correta de dados dos sensores de rotação das rodas, de rotação do robô e de infravermelho.

Os dados coletados podem ser enviados para o EEEPC seguindo a filosofia mestre-escravo. O EEEPC no caso seria o mestre, e como tal, quando desejar um dado deve enviar uma mensagem de requisição pedindo tal dado. Da mesma forma, quando quiser passar uma referência a determinado acionamento, deve-se faze-lo utilizando mensagens de comando. Portanto, o EEEPC pode fazer controle de movimento de mais alto nível do que o ARM. A API *motores_sinale.h* desenvolvida neste projeto permite que tal conversa ocorra. Assim, com o apoio desta mesma API, a qual se encontra em um nível acima do ARM na figura 3.2, o PLAYER consegue se estabelecer como um intermediário para a comunicação entre um cliente (seja remoto ou local) e o robô, com seus sensores e atuadores. Observa-se então o cliente como o último nível da pilha da figura 3.2. Ainda, será demonstrado na seção 3.4 que o cliente remoto não necessita mais de nenhum arquivo desenvolvido neste trabalho para trocar informações com as plataformas, apenas das bibliotecas de desenvolvimento para clientes em PLAYER, o que incrementa a portabilidade deste projeto e facilidade de desenvolvimento para o mesmo, dado o apelo em modulação investido durante seu desenvolvimento.

Por portabilidade entende-se que exatamente o mesmo software cliente poderá tanto comandar robôs Pioneer como nossas plataformas, e várias outras desenvolvidas ao redor do globo que possuam compatibilidade com o PLAYER, auxiliando o desenvolvimento de trabalhos em um ambiente voltado para pesquisa e estudos em robótica de cooperação.

Por último, vale lembrar que neste trabalho foram projetadas duas plataformas robóticas com diferentes capacidades de locomoção e diferentes configurações de atuadores. Assim, é de se esperar que seus softwares sejam distintos. Porém, neste trabalho, esforço foi concentrado no objetivo de

[3]Bibliotecas de desenvolvimento para ARM em C podem ser obtidas em http://www.gnuarm.com

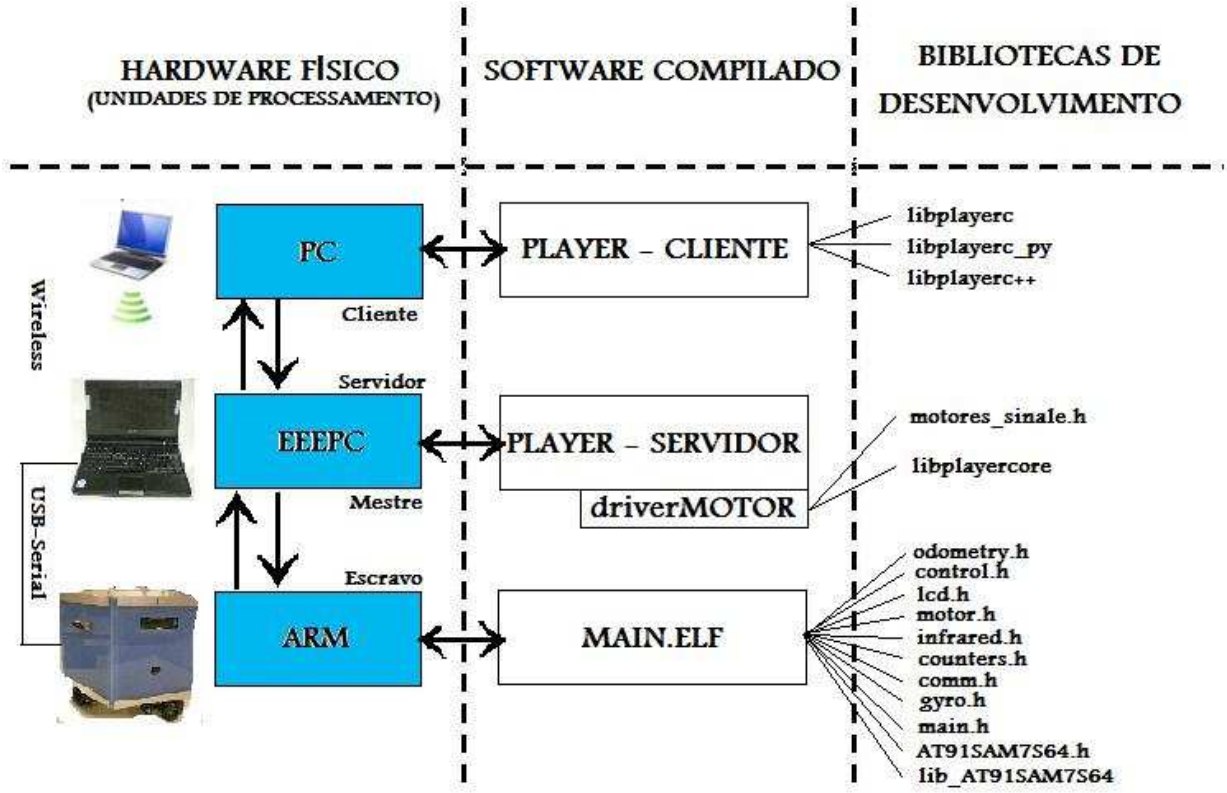

Figura 3.2: Arquitetura da comunicação entre as unidades de processamento envolvidas.

se manter os softwares idênticos para ambas as plataformas. De fato, o software desenvolvido é idêntico para ambos os robôs. O meio pelo qual a plataforma descobre seu tipo então é através de um arquivo de configuração do driver do robô no software PLAYER, conforme será discutido na seção 3.4.

3.2 Microcontrolador ARM AT91SAM7S

A tarefa principal dos microcontroladores AT91SAM7S embarcados nas plataformas é fazer a plataforma robótica associada atender mensagens de comando e mensagens de requisição que fluem pela interface RS-232 provenientes do netbook EEEPC. O sistema de controle digital e odometria incluídos no software do ARM servem de base para efetivar os comandos e atender às requisições. Os comandos e requisições existentes, acompanhados do protocolo da conversa entre o mestre (EEEPC) e o escravo (ARM), serão devidamente explicados nesta seção. O software do microcontrolador foi divido em módulos para facilitar sua modificação, para suportar a reusabilidade de código e para simplificar a leitura do mesmo por indivíduos externos ou novatos ao projeto. Os módulos existentes e suas interconexões serão detalhados nessa seção. Finalmente, no final desta seção, o leitor poderá entender como o algoritmo incluso no microcontrolador implementa os três modos de operação da plataforma e como o mesmo garante o atendimento das mensagens de comando e mensagens de requisição.

Primeiramente, antes de adentrar nos detalhes da implementação do algoritmo, é conveniente expor o ambiente de desenvolvimento em arquitetura ARM7TDMI desse trabalho.

3.2.1 Ambiente de Desenvolvimento

Para a compilação do código-fonte, foi utilizado durante este projeto o toolchain de desenvolvimento GNU ARM[4] para Linux. Esse toolchain consiste dos aplicativos padrão *GNU Binary Utils*, o compilador GCC e o depurador *Insight*; depurador o qual não tivemos a oportunidade de utilizar durante o desenvolvimento deste projeto por dificuldades e complicações em sua compilação. Porém as outras ferramentas foram prontamente compiladas e instaladas sem maiores dificuldades[5].

Para a gravação do software na memória FLASH do microcontrolador, foram utilizados dois softwares de gravação distintos destinados ao uso de duas interfaces físicas distintas durante o desenvolvimento: openOCD[6] (JTAG - Porta Paralela do PC) e Sam_I_Am[7] (USB), a serem descritos ainda nesta sub-seção.

Os computadores *desktop* utilizados durante o projeto para a programação do ARM foram máquinas Intel(R) Core(TM)2 Duo CPU E8200 @ 2.66GHz, 3.3GiB RAM, com sistema operacional Ubuntu Release 8.10 (intrepid) Kernel Linux 2.6.27-7-generic GNOME 2.24.1. Recomenda-se o trabalho com o sistema Ubuntu dado as facilidades disponibilizadas durante os processos de instalação dos aplicativos dadas pelo software *apt-get* incluso na maioria das distribuições Ubuntu. Versões dos aplicativos utilizados neste projeto:

- arm-elf-gcc - 4.1.0
- GNU Binary Utils - 2.16.1
- Open On-Chip Debugger - 2006-06-25 23:00 CEST
- Sam_I_Am - 0.3

O processo de compilação e ligação dos códigos foi realizado pelo GCC conforme ilustra o arquivo *Makefile* contido no DVD em anexo. Trata-se de um *Makefile* escrito por Martin Thomas, baseado no *Makefile* do WinAVR[8] escrito por Eric B. Weddington. Não houve nesse projeto a necessidade de se modificar nenhuma linha do *Makefile* original, exceto a linha onde é definida a lista de arquivos fonte em C:

```
SRC = $(TARGET).c counters.c lcd.c motor.c comm.c infrared.c gyro.c control.c
                                    odometry.c
```

A razão pela qual essa linha é modificada é evidente. A filosofia de dividir o programa em módulos requer que vários módulos distintos sejam compilados, ao invés de compilar um código

[4] Bibliotecas de desenvolvimento em ARM podem ser obtidas em http://www.gnuarm.com

[5] Podem ocorrer problemas durante a compilação, porém existem inúmeros FAQs na internet para auxiliar a instalação

[6] openOCD pode ser obtido através do site http://openocd.berlios.de/

[7] Sam_I_Am pode ser obtido através do site http://claymore.engineer.gvsu.edu/šteriana/Software/Sam_I_Am/index.html

[8] Mais informações acerca WinAVR, vide http://winavr.sourceforge.net/

fonte único. Mais informações sobre os módulos citados nesta linha estão localizadas na sub-seção 3.2.4 e na sub-seção 3.2.5.

É importante comentar também sobre o formato de saída do arquivo-objeto gerado pelo processo de compilação, pois os softwares de gravação variam quanto aos formatos de arquivo-objeto compatíveis. O *Makefile* utilizado possui a seguinte linha:

```
FORMAT = binary
```

A linha de *script* acima define o formato de saída do arquivo-objeto como sendo binário. Esse formato é prontamente aceito pelo software de gravação openOCD, mas incompatível com o software Sam_I_Am. Porém, de modo a não ser necessário alterar o *Makefile*, utilizou-se no processo de gravação do Sam_I_Am outra técnica para alterar o formato do arquivo-objeto que será visto mais adiante nesta mesma sub-seção, conforme a figura 3.3 ilustra, quando explicarmos o processo de gravação com o software Sam_I_Am.

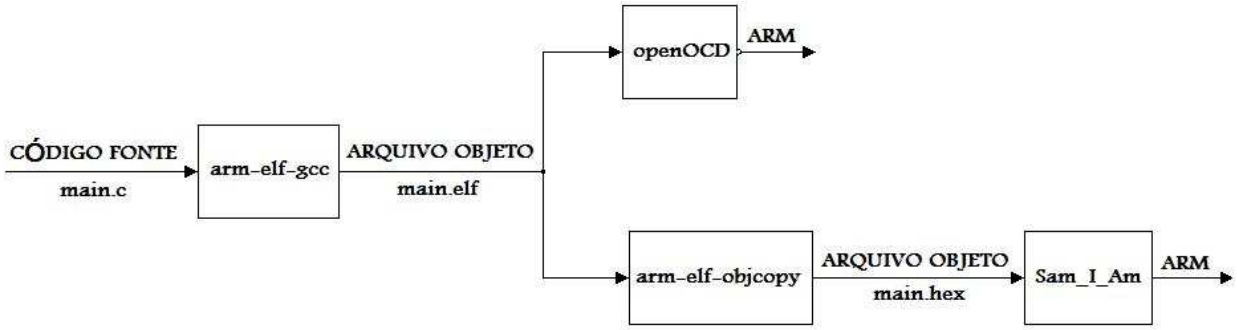

Figura 3.3: Estratégias de gravação no ambiente de desenvolvimento ARM.

Assim, na linha de comando, de forma a compilar o código-fonte e produzir o código-objeto, basta digitar:

`$ make clean` ; Limpa os arquivos previamente compilados.

`$ make all` ; Compila o software completo.

Por último, nesta sub-seção, será discutido o processo de gravação do arquivo-objeto no microcontrolador utilizando-se de dois softwares distintos.

O primeiro software a ser discutido é o openOCD. Esse é o software padrão no *Makefile* de Martin Thomas, portanto a seqüência de comandos a ser digitado no *Shell* do Linux para prosseguir a gravação está contida no arquivo *Makefile* e pode ser reproduzida pelo comando:

`# make program` ; Transfere o arquivo-objeto para o dispositivo microcontrolador.

Logicamente, de forma a se usar o comando acima corretamente, deve-se antes ter primeiramente conectado o microcontrolador ARM ao PC pela porta paralela através da gravadora *ARM-JTAG MACRAIGOR WIGGLER COMPATIBLE*. Deve-se também energizar o ARM pela conexão do mesmo ao computador pela interface USB. Note que a conexão no computador do ARM pela USB ocorre apenas pela fonte de alimentação, i.e., não ocorre troca de dados pela interface USB. Ainda, pelo fato do software openOCD precisar de acesso à porta paralela, o comando acima deve

ser executado com privilégios de super-usuário. O processo de gravação costuma durar quase um minuto.

O segundo software de gravação a ser discutido é o Sam_I_Am. De acordo com o website oficial do software, Sam_I_Am é um programa usado para se interagir com um microcontrolador Atmel AT91SAM7S rodando o software monitor SAM-BA. Foi especialmente designado para usuários Linux que desejassem se conectar com seus dispositivos pela porta USB. Não é compatível nem com Windows nem com a porta serial.

O assistente de inicialização SAM-BA é o programa Boot padrão que provê uma maneira simples para programar *in situ* a memória FLASH do microcontrolador. A maneira para fazer essa programação é explicitada nos passos a seguir:

- Entra-se no modo *SAM-BA Boot Recovery* mantendo os pinos TST, PA0, PA1 e PA2 em nível alto por pelo menos 10 segundos. Na placa de desenvolvimento OLIMEX utilizada neste projeto, esse feito é resumido à alocação do jumper TST por 10 segundos (vide figura 3.4). O *SAM-BA Boot Recovery* instalará então o SAM-BA boot nos dois primeiros setores da memória Flash *on-chip*.
- Desliga-se o microcontrolador ARM e retira-se o jumper TST.
- Carrega-se no computador responsável pelo software Sam_I_Am o driver de USB serial com os seguintes parâmetros: vendor=0x03EB e product=0x6124. No ambiente Linux, esse feito é resumido pela seguinte linha de comando com permissões de super-usuário:

  ```
  # modprobe usbserial vendor=0x03EB product=0x6124
  ```

- Converte-se o arquivo-objeto, no nosso caso *main.elf*, para o formato de arquivo-objeto Intel Hex. Esse feito é resumido pela seguinte linha de comando através do software incluso no pacote GNU BinUtils:

  ```
  $ arm-elf-objcopy -O ihex main.elf main.hex
  ```

- Grava-se este novo arquivo-objeto no microcontrolador através do programa Sam_I_Am, caso o microcontrolador esteja associado ao arquivo especial /dev/ttyUSB0:

  ```
  $ Sam_I_Am
  » open /dev/ttyUSB0
  » writew FFFFFF64 5A000004
  » writew FFFFFF64 5A002004
  » flash main.hex
  ```

A seqüência de passos anterior inclui dois passos de escrita em registradores especiais do ARM. Caso seja necessário programar na memória FLASH em uma região de memória travada, deve-se primeiro destravá-la. No caso do AT91SAM7S64, o controlador de memória FLASH embutido administra 16 *lock bits* para proteger 16 regiões da memória Flash contra comandos de programação ou limpeza inadvertidos. As duas primeiras regiões são automaticamente travadas pelo SAM-BA

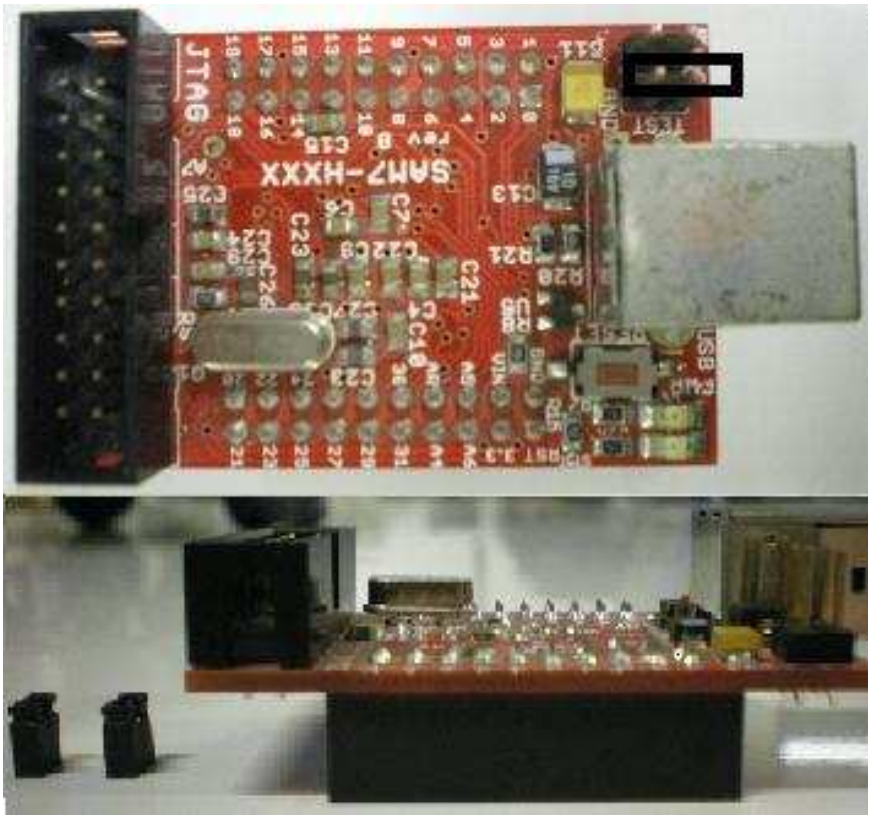

Figura 3.4: Localização do jumper TST na placa de desenvolvimento SAM7-H64 OLIMEX.

quando o mesmo se instala na memória FLASH. Portanto, os dois passos de escrita em registradores especiais do ARM são necessários para destravar tais regiões e permitir a gravação do software no ARM.

Caso o SAM-BA seja sobrescrito, o assistente de Boot não irá mais funcionar, portanto o Sam_I_Am também deixará de funcionar. Os cinco passos acima podem ser seguidos novamente para instalar o SAM-BA, e assim, o Sam_I_Am voltará a funcionar. O processo de gravação costuma durar apenas alguns segundos.

Existe um detalhe adicional na arquitetura do microcontrolador AT91SAM7S que pode trazer problemas ao passos ditos anteriormente para o Sam_I_Am. A memória Flash é mapeada no endereço 0x0010 0000. É também acessível pelo endereço 0x00 depois do *Reset* e antes do Comando de Remapeamento. Essa diferença de endereço deve ser levada em consideração nos arquivos AT91SAM7S64-RAM e AT91SAM7S64-ROM para correto funcionamento da gravação.

Nota-se facilmente que o segundo método de gravação é muito mais dispendioso que o método de gravação pelo openOCD, apesar de a gravação em si durar muito menos. Portanto, o método mais utilizado e altamente recomendado durante o desenvolvimento deste projeto foi a gravação pela porta paralela pelo software openOCD.

Por último, segue-se a recomendação de se manter o nível de otimização de compilação constante no *Makefile* e igual a 2. Esse foi o nível utilizado durante todo o desenvolvimento deste trabalho. Uma razão pela qual não se recomenda alterar este nível é ilustrada na sub-seção 3.2.4, quando se detalha a função *wait()*.

Tabela 3.1: Mensagens de solicitação e as respostas do ARM em termos de dados requisitados.

Código de Mensagem	**Dado Requisitado**	**Unidade**
OX	Coordenada X da Odometria	Metros
OU	Coordenada Y da Odometria	Metros
OT	Coordenada Thet da Odometria	Radianos
S1	Infravermelho 1 RAW	Não Definido
S2	Infravermelho 2 RAW	Não Definido
S3	Infravermelho 3 RAW	Não Definido
S4	Infravermelho 4 RAW	Não Definido
S5	Infravermelho 5 RAW	Não Definido
S6	Infravermelho 6 RAW	Não Definido
G1	Velocidade Angular do Robô	Radianos/seg

3.2.2 A interface de comandos e requisições e modos de operação

Os recursos físicos das plataformas, como os motores e sensores, são administrados pelo microcontrolador ARM embutido em cada uma. Porém, o microcontrolador não toma nenhuma decisão referente ao movimento e configuração da plataforma. Assim, as tomadas de decisão são tomadas pelo netbook EEEPC. Portanto, deve existir um meio pelo qual o EEEPC acesse os recursos da plataforma pelo ARM. Conforme já mencionado, esse meio é promovido por meio de uma arquitetura mestre-escravo na qual o EEEPC solicita informações por meio de *mensagens de solicitação* ou acessa atuadores por meio de *mensagens de comando* enviadas ao ARM. Respectivamente, as tabelas 3.1 e 3.2 listam as mensagens de solicitação e as mensagens de comando, além de explicitar as respostas do ARM em termos de dados requisitados ou comportamento da plataforma. Sobre como são exatamente definidas as mensagens em um enlace de comunicação RS-232, vide seção 3.2.3.

Segue a seguir alguns comentários sobre a interface definida. Observa-se que a interface projetada possui mensagens de comando para correção dos dados de posição do robô provenientes da odometria. Portanto, nossa implementação de robôs móveis possui uma possível extensão a possíveis sensores externos, i.e., sensores fisicamente desconectados do robô que meçam também a coordenada absoluta de posição da plataforma móvel. Ainda, existem também na previamente definida interface mensagens de requisição de odometria caso se deseje obter as coordenadas do robô calculadas pelos sensores proprioceptivos.

Para o correto cálculo no sistema de odometria, deve-se saber em qual tipo de plataforma o software se encontra. A plataforma sabe seu tipo através das mensagens de comando de códigos XX ou XY. Até que seja corretamente definido o tipo de plataforma, pode-se ter um valor errôneo retornado pelas mensagens de requisição de dados da odometria.

Na tabela 3.1, a unidade dos valores medidos pelos sensores infravermelho não é definida. Esse fato se deve à falta de monotonicidade crescente linear entre a magnitude do valor medido e alguma grandeza física de distância. Isso ocorre por diversos fatores que ocorrem em série. Entre eles estão

Tabela 3.2: Mensagens de comando e as respostas do ARM em termos de comportamento da plataforma.

Código Msg.	**Alvo**	**Comportamento**
M1	Motor 1	Seta a nova referência de velocidade do motor (m/s)
M2	Motor 2	Seta a nova referência de velocidade do motor (m/s)
M3	Motor 3	Seta a nova referência de velocidade do motor (m/s)
MD	Motores	Desliga todos motores
UM	Motores	Liga-se todos os motores
SU	Sensores IV	Liga-se todos sensores infravermelho
SD	Sensores IV	Desliga-se todos os sensores infravermelho
XX	Configuração	Seta o comportamento da plataforma como feminino
XY	Configuração	Seta o comportamento da plataforma como masculino
CA	Configuração	Entra no modo de operação Calibração
SI	Configuração	Entra no modo de operação Sintonização
CR	Configuração	Entra no modo de operação Cruzeiro
KP	Controle	Seta um novo parâmetro K_p para o controle de velocidade
KI	Controle	Seta um novo parâmetro K_i para o controle de velocidade
OR	Odometria	Reseta a posição para (x=0, y=0, thet=0)
SX	Odometria	Seta a posição x para um determinado valor (metros)
SY	Odometria	Seta a posição y para um determinado valor (metros)
ST	Odometria	Seta a posição thet para um determinado valor (radianos)

o formato peculiar da própria curva de calibração dada pelo fabricante (vide figura 2.17), o divisor resistivo encontrado no circuito de aquisição de dados (vide seção 2.8), a resistência de saída não-nula do multiplexador analógico e a transformação proveniente da conversão analógico-digital. Portanto, para o correto uso dos sensores infravermelho, deve-se fazer uma calibração dos mesmos. Essa calibração é feita em nível de software servidor PLAYER, portanto essa discussão será postergada até a seção 3.4, onde se é discutido como o PLAYER implementa a curva de calibração dos sensores, e mantêm os parâmetros da curva livres para serem alterados facilmente pelo usuário. A razão pela qual a implementação da calibração é feita pelo EEEPC é a desejada facilidade de desenvolvimento do algoritmo de processamento dos dados destes sensores no futuro. Deseja-se que o software do ARM seja modificado o menos possível. Da maneira na qual estão distribuídas as tarefas entre as unidades de processamento, para se modificar a estratégia de calibração e processamento dos sensores não é necessário modificar o software contido no ARM.

Por último, os sensores infravermelho podem ser ligados ou desligados pelo EEEPC. Essa função é crucial, dado que estas plataformas poderão trabalhar juntas para o alcance de determinado objetivo comum, e o sensor de uma não pode afetar as medições do sensor da outra.

Ao contrário dos sensores infravermelho que serão calibrados em nível de software PLAYER, o sensor de rotação da plataforma é calibrado no próprio software em ARM durante sua inicialização. Portanto, a mensagem de requisição de velocidade angular da plataforma retorna a magnitude *exata*

da grandeza para o usuário, sem a necessidade de nenhuma outra transformação em cima do valor retornado. A condição para correta calibração do girômetro é discutida na sub-seção 3.2.4.

Das diversas variáveis existentes em um sistema de controle, conforme ilustrado na figura 3.5, as mensagens de comando relativas às velocidades dos motores disponíveis na interface alteram apenas a referencia de velocidade no controle de velocidade. Os sinais de controle u(z) não estão disponíveis, assim como as medições de velocidade das rodas v(t). Portanto, existe um encapsulamento das informações relativas ao sistema de controle das velocidades das rodas e o usuário do EEEPC tem acesso apenas à referencia ref(z). Por outro lado, todos os parâmetros do controle proporcional-integral estão disponíveis ao usuário EEEPC. Pode-se modificar K_p e K_i pela interface definida. O método no qual se sintoniza o controle das rodas será devidamente apresentado na seção 3.5, com o auxílio do modo de operação sintonizador do robô, o qual será introduzido a seguir.

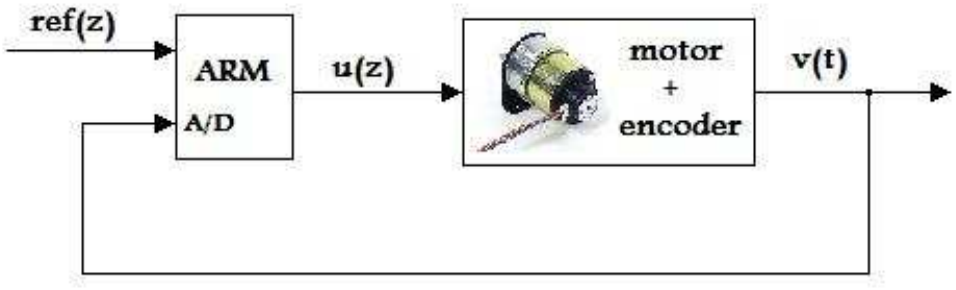

Figura 3.5: Esquema lógico de controle de velocidade simplificado.

As plataformas deste trabalho possuem três modos de operação distintos: *modo de operação cruzeiro*, *modo de operação sintonizador* e *modo de operação calibrador*. O modo de operação cruzeiro é o modo carregado como padrão ao se energizar o robô. Este modo provê a interface descrita nesta sub-seção e postula que a plataforma não deve tomar decisões por si mesma. Portanto, é nesse modo que a comunicação mestre-escravo com o EEEPC é implementada. Esse é o modo em que as plataformas devem operar em condições normais. Caso o controle digital PI dos motores esteja com o desempenho insatisfatório, o software embarcado no ARM permite a sintonização dos parâmetros do controlador pelo modo de operação sintonizador. Para entrar nesse modo, basta emitir a mensagem de comando correspondente encontrada na tabela 3.2. Após aproximadamente 5 segundos de recebida a mensagem de comando, o ARM excitará o controle digital de velocidade de seus motores conforme a referencia ilustrada na figura 3.6. A resposta em velocidade angular dos motores será enviada para o EEEPC, e a mesma pode ser analisada através do software MATLAB, conforme a seção 3.5 ilustrará. Após o fim da execução da função de excitação dos motores, o ARM volta automaticamente para o modo de operação cruzeiro. De maneira semelhante, caso as magnitudes dos valores retornados pelos sensores infravermelho não sejam satisfatórios, o software embarcado no ARM permite a modificação dos parâmetros da curva de calibração pelo modo de operação calibrador. Para entrar neste modo, basta emitir a mensagem de comando correspondente encontrada na tabela 3.2. Após aproximadamente 5 segundos de recebida a mensagem de comando, o ARM enviará em tempo real as medições dos sensores pelo meio de comunicação RS-232 para o EEEPC. Esses dados podem ser analisados pelo MATLAB,

conforme a seção 3.5 ilustrará. Após aproximadamente 1 minuto de execução neste modo, o ARM volta automaticamente para o modo de operação cruzeiro. Deve-se notar que as mensagens de comando e requisição são ignoradas durante os modos de calibração e sintonização.

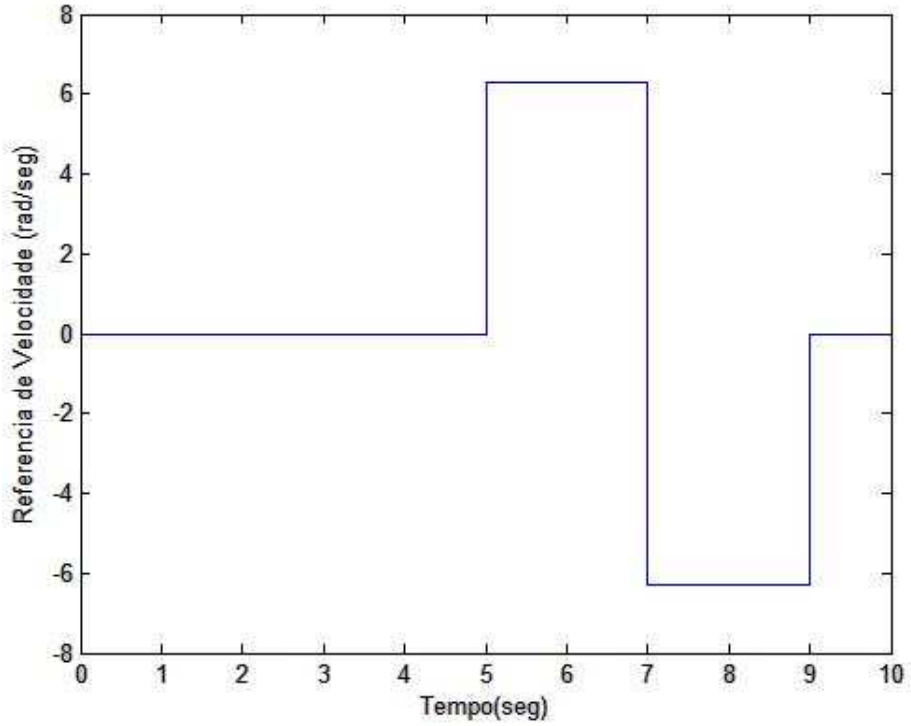

Figura 3.6: Referência de velocidade dos motores no tempo — modo de operação sintonização.

3.2.3 Semântica da comunicação mestre-escravo RS-232

Conhecida a interface de comandos e requisições, nesta sub-seção explanar-se-á como a mesma é implementada fisicamente em um enlace de comunicação RS-232 assíncrona *full duplex*. De forma a simplificar o projeto, utilizou-se neste trabalho o protocolo flexível de comunicação ilustrado pela figura 3.7. A configuração do canal serial RS-232 em ambas as pontas (EEEPC e ARM) é a seguinte:

- Velocidade de transmissão: 115200 bps;
- Bits de Paridade: 0;
- 8 bits de dados;
- Bits de Parada: 1;
- Sem handshaking.

Observa-se que existem sempre dois campos de informação em toda mensagem. Em cada mensagem, existe um caractere indicador de início '#', um campo de informação determinando o código da mensagem conforme tabela 3.2 ou tabela 3.1, um separador de campos '!', um campo

Figura 3.7: O protocolo flexível de comunicação para implementação da conversa EEEPC-ARM.

de informação numérica e um terminador de mensagem '$'. Os caracteres de início, separador de campos e terminador têm a função de fornecer à comunicação mais robustez, dado que os mesmos ajudam a processar uma detecção de perda de bytes e início de nova mensagem. O campo de código de mensagem é codificado como uma seqüência de dois bytes em ASCII. Ainda, o campo de informação numérica é codificado como um dado *float* de linguagem C de 32 bits, caso a mensagem necessite de uma informação numérica. Caso contrário, o campo de informação não é relevante para o processamento da informação e será descartado pelo dispositivo escravo (ARM), porém deve estar presente para manter o tamanho da mensagem fixo.

Nota-se que se uma mensagem é perdida, pelo protocolo visto até então, o dispositivo mestre (EEEPC) da comunicação não terá conhecimento da perda. De forma a tornar o protocolo mais robusto, existe o constante eco dos bytes recebidos pelo dispositivo escravo. Assim, se o eco recebido pelo dispositivo mestre não for idêntico à mensagem enviada, deduz-se que o dispositivo escravo pode não ter recebido a mensagem. Definiu-se neste projeto que o dispositivo mestre fará a escolha então de mandar novamente a mensagem, até que a comprovação de chegada correta da mesma chegue.

O protocolo é flexível no sentido em que permite que expansões na arquitetura das plataformas possam ocorrer. Assim, caso um novo sensor ou um novo atuador seja incluso no projeto, existe espaço na arquitetura da comunicação RS-232 para que esta acomode uma interação entre o EEEPC e o novo sensor/atuador sem que seja necessária uma mudança na semântica da comunicação mestre-escravo. No caso, basta criar um novo código de mensagem, i.e., criar uma nova entrada na tabela 3.1 ou na tabela 3.2. Porém, o protocolo não é flexível no que diz respeito ao tamanho da mensagem, i.e., a quantidade de bytes em uma mensagem é fixa. E assim, a mensagem sempre comportará um código de mensagem com seu propósito e apenas um dado do tipo float. Caso seja necessária a transmissão de mais de um dado do tipo float, essa poderá ser implementada em múltiplas mensagens com códigos de mensagens distintos, conforme exemplifica a figura 3.8.

Figura 3.8: Transmissão de mais de um dado do tipo *float* através de múltiplas mensagens.

Tabela 3.3: Associação entre módulos físicos e módulos em software ARM. Ainda, essa tabela discrimina as responsabilidades de cada módulo.

Módulo Lógico	**Módulo Físico**	**Funcionalidades**
counters.h	Encoders ópticos	Disponibilização da velocidade dos motores
motor.h	Pontes H	Acionar atuadores
infrared.h	Sensores Infravermelho	Leitura bruta dos sensores de Infravermelho
comm.h	Comunicação RS-232	Comunicação mestre-escravo com o EEEPC
lcd.h	Display LCD	Impressão de mensagens em tela LCD
gyro.h	Girômetro	Disponibilização da velocidade angular do robô
control.h	—	Controle de velocidade dos motores
odometry.h	—	Odometria
main.h	ARM	Definições de pinos e periféricos

3.2.4 Módulos do Software

Nesta sub-seção serão brevemente discutidas as bibliotecas em linguagem C desenvolvidas neste trabalho para auxiliar o desenvolvimento do algoritmo principal das plataformas. O software do ARM foi divido em módulos para facilitar sua modificação, para suportar a reusabilidade de código e para simplificar a leitura do mesmo por indivíduos externos ao projeto. Associou-se cada módulo do software à um módulo físico da plataforma que o ARM administra, facilitando o entendimento e o desenvolvimento de possíveis extensões futuras da plataforma. As relações entre cada módulo e suas funcionalidades são resumidas pela tabela 3.3.

Entende-se por módulo o conjunto de um arquivo cabeçalho ".h"contendo as assinaturas das funções e definições de macros, e um arquivo ".c"contendo o código das funções e variáveis globais privadas utilizadas. As variáveis globais possuem o modificador de tipo *private* para implementar o encapsulamento dos módulos. Os módulos existentes, assim como suas inter-relações, são

explicitados na figura 3.9.

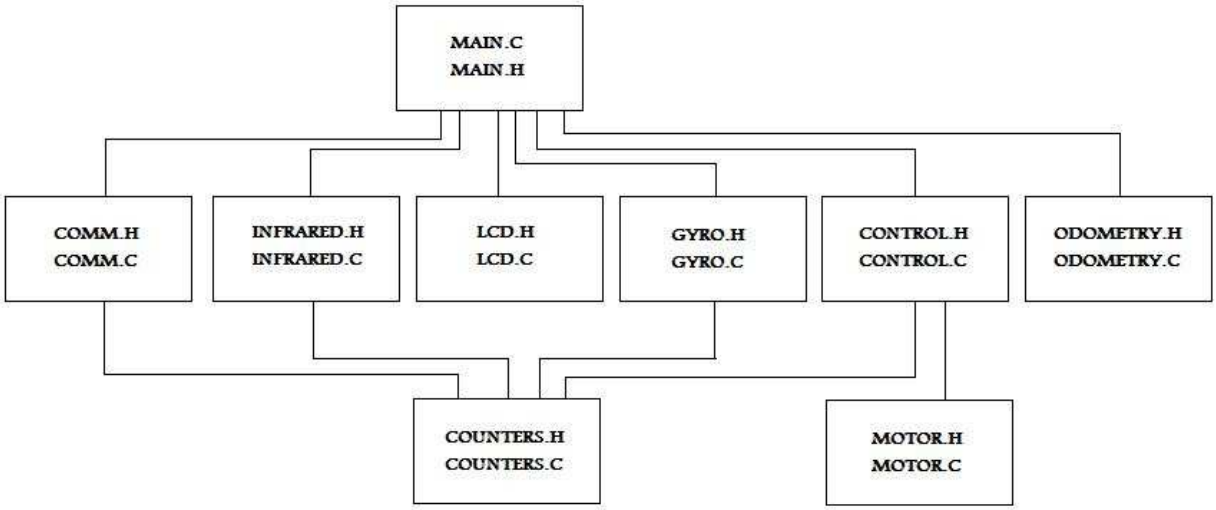

Figura 3.9: O software do ARM foi dividido nos módulos explícitados nesta figura, assim como suas inter-relações. Os módulos superiores utilizam as funções contidas nos inferiores em seus códigos.

A seguir segue as funções pertinentes à cada módulo, acompanhadas de devidas explanações de funcionamento e projeto.

- **counters.h**: define funções relativas ao funcionamento dos contadores/temporizadores do microcontrolador. Os contadores são os periféricos responsáveis pela determinação da velocidade angular das rodas da plataforma. Também define uma função de atraso no fluxo de execução do código. Vide figura 3.10.

```
#define US                                  1               /* For use in Wait Function */
#define MS                                  1000            /* For use in Wait Function */
#define SEC                                 1000000         /* For use in Wait Function */

#define FOWARD                              1
#define BACKWARD                            20

void wait (unsigned int timeus);
int counter_init();
int counter_read(int encoder_escolhido);
int counter_read_signed(int encoder_escolhido);
int counter_clear(int encoder_escolhido);
int counter_direction(int encoder_escolhido);
```

Figura 3.10: Arquivo cabeçalho do módulo *counters*.

A função "*void wait (unsigned int timeus)*"impõe um atraso ao fluxo de execução do código de aproximadamente *timeus* microssegundos. Deve-se ressaltar aqui a falta de precisão desta função. Essa função não deve ser usada na resolução de problemas onde se deve ter um atraso que requisite uma duração de alta precisão. Observa-se na figura 3.11 essa característica. Os autores apontam duas razões para a falta de precisão observada. A primeira se trata da implementação da função de espera através de um algoritmo cujo resultado da implementação é dependente do grau de otimização do compilador, pois depende de como o mesmo transforma os laços de repetição do

código em linguagem de alto nível em código de máquina (vide código fonte em anexo). Neste momento então se esclarece o porquê da recomendação de manter o grau de otimização constante no desenvolvimento deste projeto (vide sub-seção 3.2.1). A segunda razão é o fato desta função não ser atômica. Portanto, a mesma pode ser interrompida por algum evento e pausar a contagem de tempo até o término da interrupção.

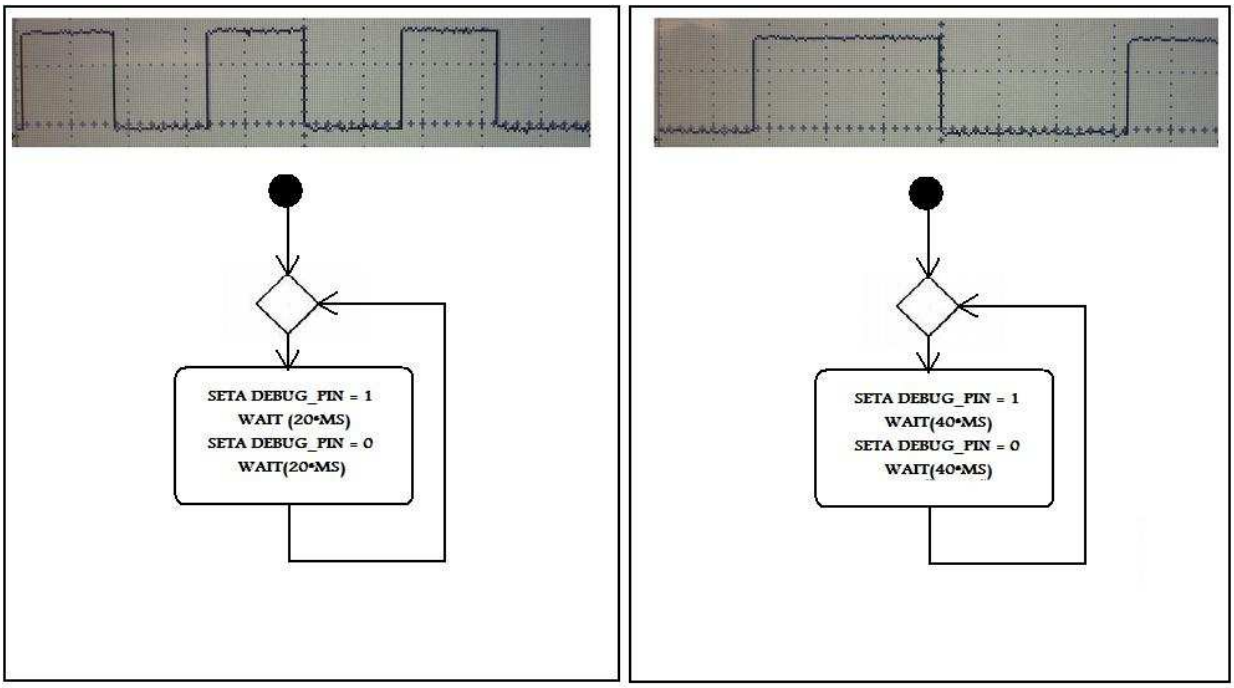

Figura 3.11: À esquerda, gráfico observado no osciloscópio quando sua ponta de prova é aplicada no pino debug do ARM executando o algoritmo à esquerda. À direita, equivalente.

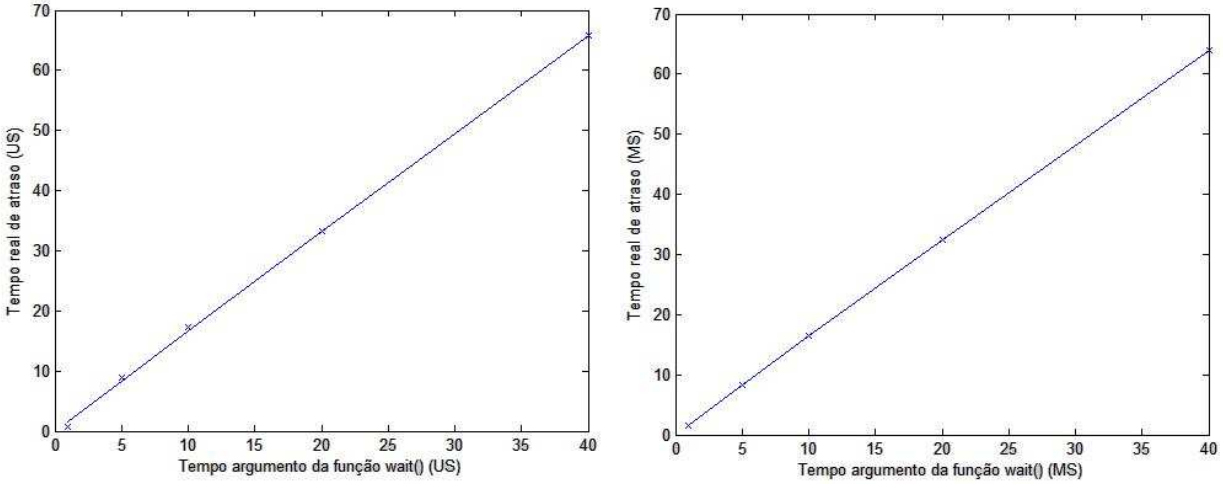

Figura 3.12: Curva característica da função *wait()*. O eixo horizontal corresponde ao tempo inserido como argumento da função. O eixo vertical ilustra o tempo real de atraso na execução estimado pelo algoritmo da função 3.11 aplicado a diversos valores de argumentos distintos.

A figura 3.12 demonstra a curva característica tomada experimentalmente da função *wait()*, a partir do algoritmo de teste localizado no anexo S1, compilado com grau de otimização 2 conforme

recomendado. Podem ocorrer desvios desta curva caso a função não seja executada atomicamente, ou caso o grau de otimização tenha sido modificado. O fato de não se respeitar sua atomicidade pode fazer com que a curva da figura 3.12 se desloque no sentido positivo do eixo vertical, com magnitude de deslocamento altamente dependente da duração da rotina de interrupção entrante. As conseqüências de não respeitar o grau de otimização padrão não é previsto neste relatório, pois no estudo deste problema deve-se recorrer ao estudo minucioso do compilador utilizado.

Para facilitar a leitura dos argumentos da função *wait()*, definiu-se neste arquivo cabeçalho as macros *US*, *MS*, *SEC*.

O restante das funções contidas neste arquivo cabeçalho dizem respeito ao tratamento dos sinais enviados pelos sensores de rotação dos motores pelos periféricos contadores/temporizadores do microcontrolador ARM. De forma a se entender o algoritmo de cálculo da velocidade angular de cada roda deve-se primeiramente ter conhecimento da estratégia utilizada para o seu cálculo. Conforme ilustrado na seção de hardware 2.7, o sensor de rotação provê o ARM com um trem de pulsos cujo período é inversamente proporcional à velocidade do motor. A cada 10ms é feita a contagem de quantos pulsos ocorreram neste intervalo de tempo (n pulsos), e assim, determina-se a velocidade média ω_i de cada roda (lembrando que cada volta completa da roda gera 9000 pulsos, devido à redução de 30 instalada entre o encoder óptico e o motor):

$$\omega_i = \frac{\Delta\phi_i}{\Delta t} = \frac{n_i . \frac{2\pi}{9000}}{0.01} rad/seg = \frac{n_i . \pi}{45} rad/seg$$

A contagem de pulsos é realizada por um periférico do microcontrolador, mais precisamente, pelo contador/temporizador. Essa decisão de projeto foi tomada para poupar o processador do microcontrolador de processamento excessivo de dados. Por exemplo, a contagem poderia ser feita através de uma porta de entrada digital que possuísse uma interrupção associada, e a cada sinal de interrupção, a rotina responsável deveria contar o tempo desde a última interrupção e calcular a velocidade pelo tempo decorrido. Porém, com uma densidade de 18000 interrupções por volta (não é possível obter 9000, pois o ARM utilizado não possui a configuração de interrupção por borda de subida apenas ou por borda de descida apenas, mas apenas a configuração de mudança de nível[9], a unidade de processamento do ARM poderia se sobrecarregar desnecessariamente.

Ainda, observa-se facilmente que no trem de pulsos gerado pelo sensor de rotação óptico não existe codificada a informação de sentido da velocidade. Por essa razão existe o circuito de direção com Flip-Flops D ilustrado na seção 2.7. Um pino do microcontrolador deve ser configurado como pino de entrada digital, de forma a se tornar possível a identificação do sentido da rotação. Todas os procedimentos necessários para iniciar e configurar os periféricos contadores do microcontrolador são resumidos pela função "*int counter_ init ( )*".

Para se ler o número de pulsos contados pode ser usada a função "*int counter_ read (int encoder_ escolhido)*". Essa função apenas lê o número de pulsos em um contador do conjunto de contadores importado do módulo *main.h*: {*ENC1_ PULSES*, *ENC2_ PULSES*, *ENC3_ PULSES*}. Caso se deseje resetar o valor de determinado contador, deve-se usar a função "*int counter_ clear (int*

[9]Cuidado deve ser tomado neste ponto. A biblioteca em C utilizada sugere uma falsa idéia de que é possível configurar a interrupção como borda de subida apenas.

encoder_ escolhido)". Ainda, observe que a função "*int counter_ read (int encoder_ escolhido)*"lê exatamente o número de pulsos contados até então. Portanto a informação acerca do sentido da rotação não é captada por esta função. Definiu-se neste projeto que o sentido será embutido no número de pulsos através de um sinal (negativo ou positivo). Esse sinal segue a padronização estipulada no capítulo de Introdução deste documento. A função que permite acesso a tal informação é a função "*int counter_ read_ signed (int encoder_ escolhido)*". Caso se deseje apenas o sentido, pode-se utilizar a função "*int counter_ direction (int encoder_ escolhido)*". No caso, essa função retornará um valor do conjunto $\{FOWARD, BACKWARD\}$.

Portanto, a velocidade pode ser calculada pelo algoritmo periódico de período 10ms ilustrado pela figura 3.13. A rotina de controle de velocidade das rodas utiliza esse algoritmo para o cálculo do erro de referência, conforme será visto na descrição do módulo *control.h*.

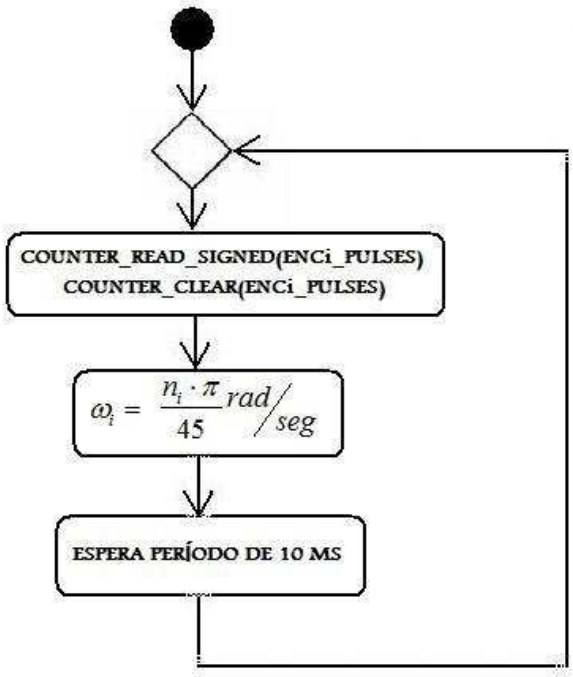

Figura 3.13: Tarefa periódica de 10ms que calcula a velocidade angular das rodas da plataforma.

O experimento para validar o funcionamento do algoritmo é extremamente simples. Trata-se apenas de girar todas as rodas uma volta completa e imprimir pelo LCD a quantidade de pulsos contados, conforme ilustra a figura 3.14. De fato, a estratégia ditada pelo algoritmo da figura 3.13 funciona. O programa de teste utilizado pode ser encontrado no anexo S2.

- **motor.h**: define funções relativas ao funcionamento e acionamento dos motores, sem controle de velocidade, o qual está incluso no módulo *control.h*. Vide figura 3.15

A função "*int set_ motor_ speed (int motor, int percentage)*"é responsável por acionar o pino de entrada *DIRECTION* da ponte H. Conforme explicado na seção 2.5, os motores são acionados por sinais PWM. Assim, a função requer que dois argumentos sejam passados: qual motor que se deseja modificar a velocidade, i.e., um valor do conjunto $\{MOTOR1, MOTOR2, MOTOR3\}$ definido no módulo, e uma porcentagem relativa ao ciclo de trabalho do sinal de acionamento correspondente.

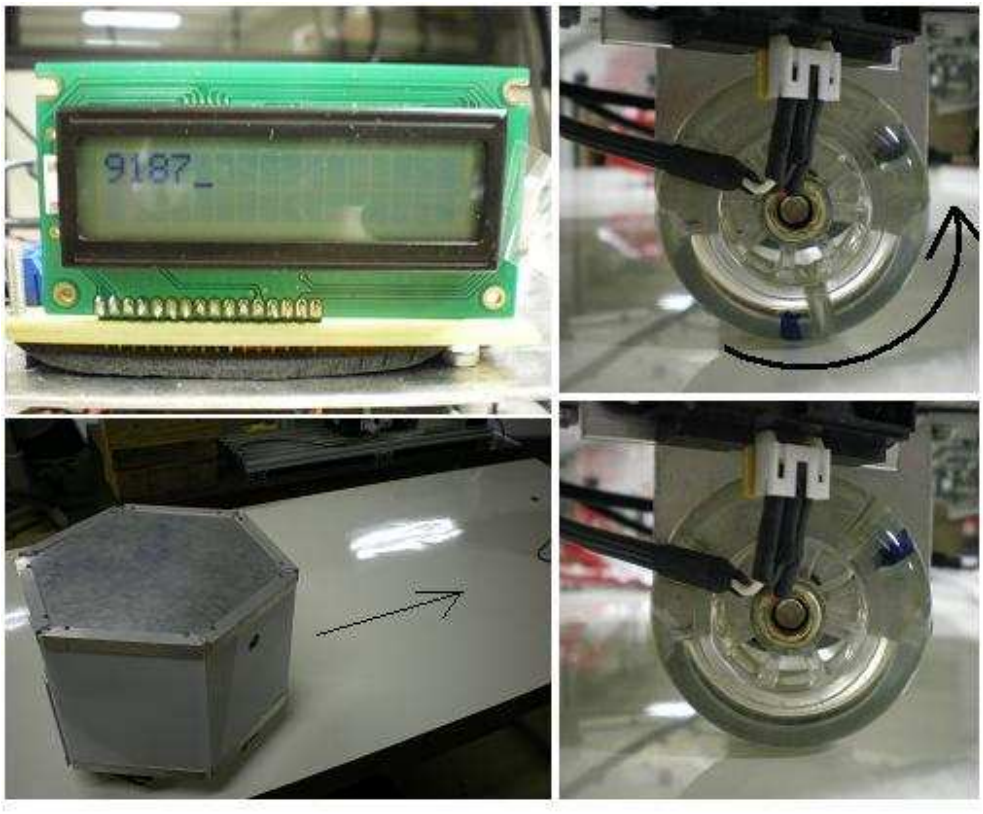

Figura 3.14: Experimento para validar o algoritmo de contagem de pulsos. A tarja azul facilita a localização da roda na situação em que a mesma efetua uma volta completa.

```
#define MOTOR1                                  (1 << 1)
#define MOTOR2                                  (1 << 2)
#define MOTOR3                                  (1 << 3)
#define ON                                      1
#define OFF                                     0

#define PWM_PERIOD                              0x400

/* PROTOTYPES */

int set_motor_speed(int motor, int percentage);
void turn_motor(int motor, int flag);
void motors_init(int perc1, int perc2, int perc3, int turn1, int turn2, int turn3);
```

Figura 3.15: Arquivo cabeçalho do módulo *motor*.

Existe uma relação bijetora entre o conjunto de valores que *percentage* pode assumir e o conjunto de valores de ciclo de trabalho do sinal PWM aplicado na entrada *DIRECTION* da ponte H conforme ilustrado pela figura 3.16. A razão de ser dessa bijeção é o desejo de se obter um acionamento numérico linear do motor pela função "*int set_ motor_ speed (int motor, int percentage)*". Portanto, propriedades de uma transformação linear T tais como

$$\begin{cases} T(0) = 0 \\ T(x) = \delta \rightarrow T(-x) = -\delta \end{cases}$$

são prontamente respeitadas pela função que relaciona valores de *percentage* às respostas de valor médio de voltagem de saída $\bar{v}$ da ponte H ao sinal de PWM correspondente. A afirmação anterior

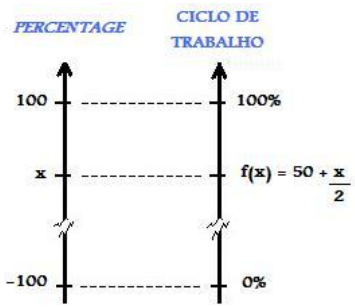

Figura 3.16: Ilustração da relação bijetora definida entre o conjunto de valores de argumento *percentage* para função *set_ motor_ speed()* e conjunto de valores de ciclo de trabalho do sinal PWM aplicado na entrada DIRECTION da ponte H.

pode ser demonstrada pelo desenvolvimento matemático a seguir, dado que v(t) é a tensão de saída da ponte H no tempo quando acionada por um sinal de PWM no pino *DIRECTION* durante um ciclo de trabalho de $\sigma\%$ (vide figura 3.17):

$$\bar{v} = \frac{1}{T} \cdot \int_0^T v(t)dt = \frac{1}{T}\left[12 \cdot \frac{T}{100} - \frac{12(100 - \sigma T)}{100}\right] = 24 \cdot \frac{\sigma}{100} - 12$$

Observa-se que $\sigma = \sigma(percentage)$ pela figura 3.16, e então:

$$\bar{v} = 24 \cdot \frac{\sigma(percentage)}{100} - 12 = 24 \cdot \frac{50 + \frac{percentage}{2}}{100} - 12 = 12 \cdot \frac{percentage}{100}$$

Conclui-se que $\bar{v}(\sigma)$ não é uma função linear em σ, e sim uma função afim, a qual não é desejada pelos projetistas pela necessidade de um acionamento linear para o correto funcionamento do controlador PI. Portanto, constrói-se a função $\bar{v}$(percentage), a qual é linear, e disponibiliza-se-a através da função "*int set_ motor_ speed (int motor, int percentage)*".

Outra função interessante deste módulo é a função "*void turn_ motor (int motor, int flag)*". Esta função possui a tarefa de desligar/ligar a tensão de saída da ponte H no motor. Por desligar, entende-se retirar a tração do motor. Por conveniência, definiu-se os valores do argumento motor como pertencente ao conjunto $\{ON, OFF\}$ definidos por macros no arquivo cabeçalho. A macro *PWM_ PERIOD* definida também no arquivo cabeçalho é utilizada apenas internamente ao módulo, e assim, não será explicada aqui. Caso o leitor realmente deseje saber aonde essa macro é utilizada, vide em anexo o código fonte do software para ARM das plataformas.

Para se utilizar as anteriores funções relativas ao acionamento de motores, deve-se primeiramente configurar os canais dos periféricos geradores de sinal PWM do ARM. A função "*void motors_ init (int perc1, int perc2, int perc3, int turn1, int turn2, int turn3)*"realiza tal inicialização. Os parâmetros são respectivamente as *percentages* inciais dos motores e se os mesmos devem inicializar em estado ligado ou desligado.

Por último, observa-se uma importante propriedade do software em ARM desenvolvido neste trabalho. Recapitula-se neste momento que exatamente o mesmo software é carregado na plataforma diferencial e na plataforma omnidirecional. Assim, a plataforma diferencial vai fazer referencia em seu código de três motores, apesar de possuir apenas dois. Qualquer chamada de função

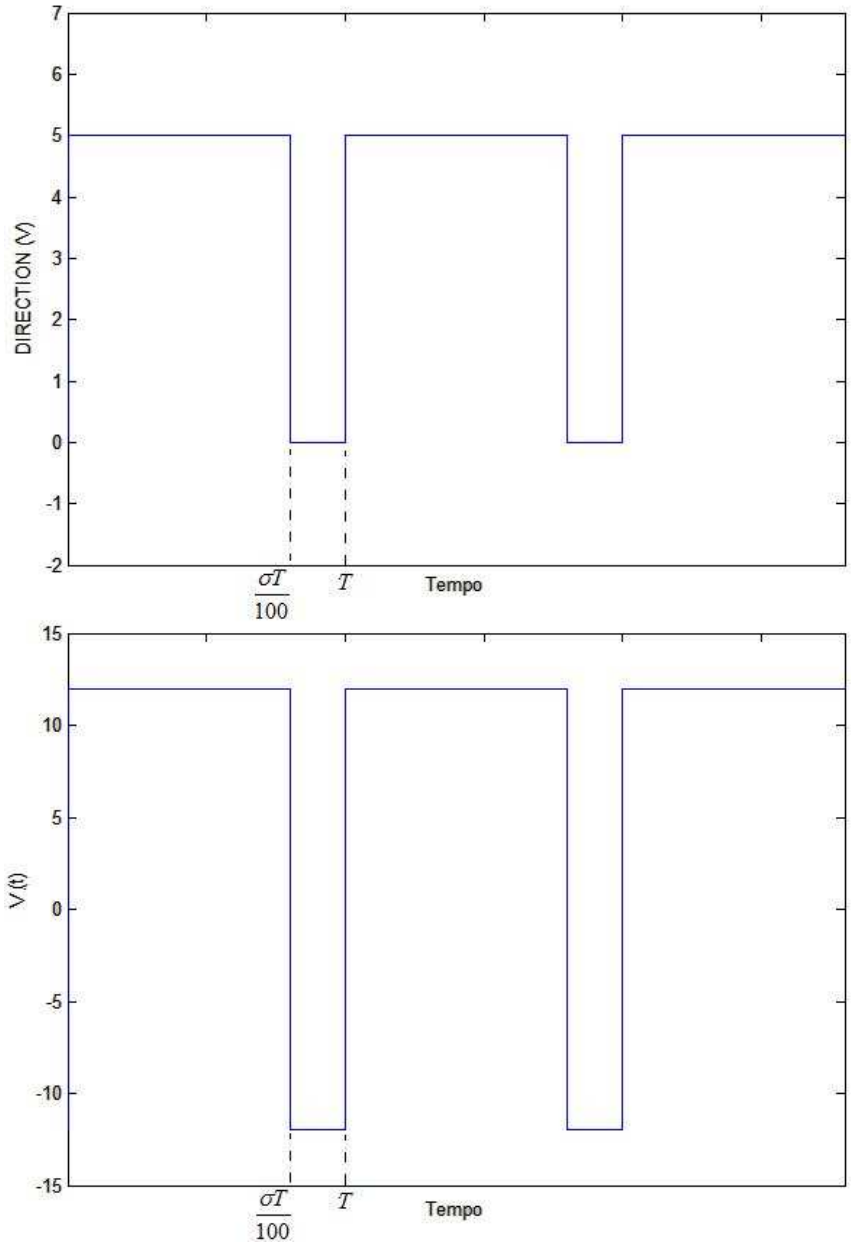

Figura 3.17: Tensão de saída da ponte H quando acionada por um sinal de PWM no pino DIRECTION durante um ciclo de trabalho σ.

referenciando o terceiro motor será de fato executada, porém, como as portas lógicas relacionadas ao terceiro motor estão em aberto (a arquitetura em hardware das plataformas é idêntica, exceto pelas entradas/saídas do ARM da plataforma diferencial associadas ao terceiro motor que estarão em aberto, vide capítulo 2), o resultado da execução não poderá ser visto fisicamente, exceto com auxílio de osciloscópios ou multímetros ligados às portas lógicas do ARM em aberto.

- **infrared.h**: define funções relativas ao funcionamento dos sensores infravermelho. Vide figura 3.18.

Não existe uma quantidade de conversores analógico-digitais no microcontrolador ARM suficiente o bastante para associar cada conversor com cada um dos seis sensores infravermelho.

```
/* DEFINITIONS */

#define IR_SENSOR_1                     1               /* For use in Function */
#define IR_SENSOR_2                     2               /* For use in Function */
#define IR_SENSOR_3                     3               /* For use in Function */
#define IR_SENSOR_4                     4               /* For use in Function */
#define IR_SENSOR_5                     5               /* For use in Function */
#define IR_SENSOR_6                     6               /* For use in Function */

/* PROTOTYPES */

int infrared_init();
int turn_on_IR_sensors();
int turn_off_IR_sensors();
unsigned int infrared_read_sensor(int which_sensor);
```

Figura 3.18: Arquivo cabeçalho do módulo *infrared*.

Portanto existe no projeto o multiplexador que administra o acesso dos sensores ao conversor analógico-digital. O módulo acima esconde os detalhes acerca a manipulação lógica do multiplexador, tópico explicado na seção 2.8. Parte desta manipulação envolve, por exemplo, a espera de um determinado tempo de estabilização da saída do multiplexador quando o mesmo acaba de sofrer comutação. Este tempo é da ordem de milissegundos e seu valor exato é estimado na seção 2.7. A forma na qual essas manipulações são implementadas pode ser conferida lendo-se o código fonte associado deste módulo em anexo.

Antes de se tentar obter leituras dos sensores através da função definida neste módulo, deve-se configurar os pinos associados ao multiplexador e inicializar os conversores analógico-digitais. Esse feito é resumido à chamada de função "*int infrared_ init ( )*". Esta função não calibra os sensores. A função de calibração é embutida ao PLAYER, dado que através do PLAYER os parâmetros da curva de calibração ficam livres para serem alterados facilmente pelo usuário. Assim, as funções deste módulo trabalham com o valor bruto adquirido pelos conversores de 10 bits do ARM. Portanto, os valores retornados dos sensores se localizarão no intervalo de valores inteiros iniciado em 0 e terminado em 1023. Esses valores podem ser obtidos através da função "*unsigned int infrared_ read_ sensor (int which_ sensor)*". O argumento *which_ sensor* deve pertencer ao conjunto {*IR_ SENSOR_ 1*, *IR_ SENSOR_ 2*, *IR_ SENSOR_ 3*, *IR_ SENSOR_ 4*, *IR_ SENSOR_ 5*, *IR_ SENSOR_ 6*} definido no arquivo cabeçalho.

Por último, os sensores infravermelho podem ser ligados ou desligados. Essa função é crucial, dado que as plataformas diferencial e omnidirecional poderão trabalhar juntas para o alcance de determinado objetivo comum, e o sensor de uma não pode afetar as medições do sensor da outra. Esses feitos são realizados pelas funções "*int turn_ on_ IR_ sensors ( )*"e "*int turn_ off_ IR_ sensors( )*".

- **comm.h**: define funções relativas à comunicação serial mestre-escravo sob o protocolo físico RS-232. Vide figura 3.19.

No momento em que a função "*int waitForMessage (char * commandR, float *dataR)*"é chamada, o fluxo de execução do programa é travado até o momento em que se chegar uma mensagem válida na interface RS-232, conforme o protocolo definido na sub-seção 3.2.3. Apesar de ser uma função que impede que a linha seguinte de código seja executada enquanto não chegar uma men-

```
/* DEFINITIONS */

#define BUFFSIZE                        9
#define BAUDRATE                        115200

/* PROTOTYPES */

int waitForMessage(char * commandR, float *dataR);
int sendMessage(const char * const commandS, const float * const dataS);
int comm_init();
```

Figura 3.19: Arquivo cabeçalho do módulo *comm*.

sagem válida, rotinas de interrupção não são travadas e serão executadas normalmente. E depois de executadas, o travamento do fluxo retoma o tempo de processador novamente. O fato de funções de interrupção poderem ser executadas não deprecia o desempenho da função, desde que a duração desta rotina de interrupção não seja demasiadamente longa. Ainda, existe dentro do microcontrolador um buffer de dados administrado por hardware de forma que não se é necessária nenhuma intervenção do processador para o armazenamento de dados não processados. Portanto, caso se chegue algum dado pela interface serial, mas não se ter ainda chamado a função "*int waitForMessage (char * commandR, float *dataR)*", não ocorrerá perda de dados.

A função "*int waitForMessage (char * commandR, float *dataR)*"filtra a mensagem e armazena em *commandR* o código da mensagem enquanto guarda em *dataR* a informação numérica associada. Essa função também implementa o eco da mensagem recebida, o que é uma característica desejável pelo protocolo de comunicação definido neste projeto.

Por sua vez, a função "*int sendMessage (const char * const commandS, const float * const dataS)*"envia uma mensagem pela interface serial, com o formato da semântica definida neste projeto.

Para o correto funcionamento destas funções, deve-se inicializar o periférico USART do microcontrolador. Todas as instruções necessárias para esta inicialização estão contidas na função "*int comm_ init()*".

Portanto, para a correta comunicação do ARM, pode-se seguir o simples algoritmo da figura 3.20. De fato, depois de se ter inicializado no código todos os periféricos utilizados no robô, o algoritmo principal das plataformas é apenas essa ilustração. Paralelamente à rotina principal, ocorrem as tarefas periódicas de controle e odometria, a serem explanadas na seção 3.2.5.

Observa-se neste momento que o software projetado não possui sua lógica de comunicação serial baseada em chamadas de interrupções de hardware do periférico USART. O ARM possui interrupções que sinalizam a chegada de um novo byte de informação. Porém, os projetistas decidiram por não usar essas funcionalidades. Isso se deve à vontade dos autores de simplificar a lógica do software. Nenhum módulo deverá ter acesso a interrupções. Apenas o módulo main.c poderá configurar interrupções e, conforme será visto na sub-seção 3.2.6, a única rotina de interrupção utilizada será a rotina de tratamento de *overflow* do periférico temporizador periódico (PIT - *Periodic Interrupt Timer*), necessária para a execução periódica precisa das tarefas de controle de velocidade e odometria.

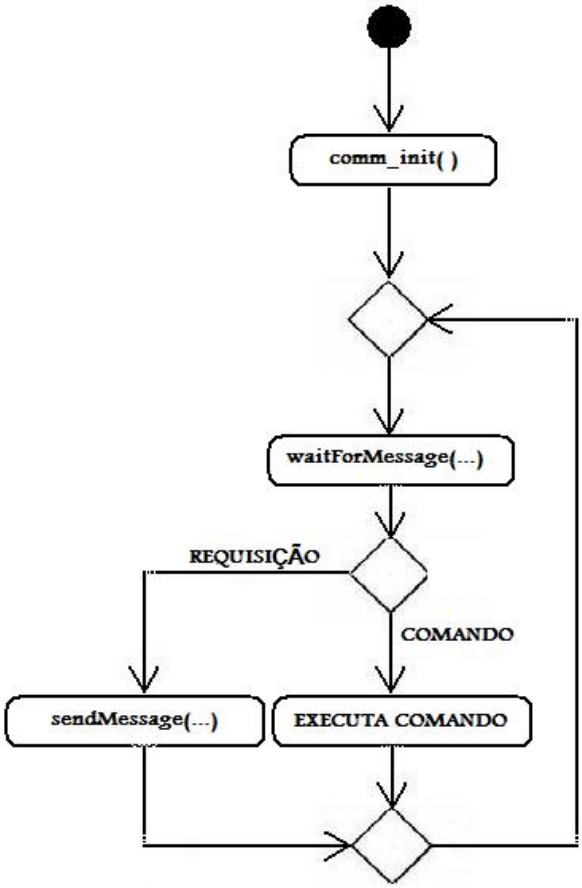

Figura 3.20: Implementação em ARM da comunicação mestre-escravo utilizando a biblioteca desenvolvida neste trabalho.

- **lcd.h**: define funções relativas ao funcionamento e operação do LDC. Vide figura 3.21.

Este módulo não é acessado pelo código final do software deste projeto, mas foi extensamente utilizado como um módulo de depuração durante o desenvolvimento do trabalho, inclusive durante a validação dos algoritmos de contagem de pulsos do sensor de rotação dos motores, conforme visto previamente nesta sub-seção. Ainda, através deste módulo, no futuro pode-se continuar utilizando o LCD no desenvolvimento com grande facilidade.

As macros definidas neste módulo não necessitam de nenhuma modificação, exceto caso as mesmas sejam utilizadas em outro projeto, onde exista outro modelo de LCD com diferentes parâmetros físicos (por exemplo, diferente número de caracteres por linha ou diferente número de colunas).

As funções deste módulo não são complexas o suficiente para merecer maiores detalhamentos. O próprio nome de cada função demonstra sua utilidade. Exalta-se aqui apenas a necessidade de se chamar a função "*int lcd_ init(void)*"antes de se usar qualquer outra deste módulo.

- **gyro.h**: define funções relativas ao funcionamento e operação do sensor de rotação da plataforma. Vide figura 3.22.

```
/* DEFINITIONS */

#define LINES                                   2
#define COLUMNS                                       16

#define CLEAR_DISPLAY                           0x01
#define CURSOR_HOME                             0x02

#define SET_INTERFACE_LENGTH                    1 << 5
#define _8_BIT_MODE                             1 << 4
#define _2_LINE_DISPLAY                               1 << 3
#define _5x10_DISPLAY                           1 << 2

#define ENABLE_DISPLAY_CURSOR                   1 << 3
#define DISPLAY_ON                             1 << 2
#define CURSOR_ON                               1 << 1
#define BLINKING_ON                             1 << 0

#define SET_ENTRY_MODE                          1 << 2
#define INCREMENT_CURSOR                        1 << 1
#define SHIFT_DISPLAY                           1 << 0

#define ENABLE_PULSE_WIDTH                      600


/* PROTOTYPES */

void pulse_lcd_e(void);
void send_byte(int byte, int RS);
void command_lcd(int instruction);
void place_cursor(int line, int column);
void write_lcd(int character);
void write_at(int byte, int line, int column);
void putc ( char c );
void write_string ( char *s );
int pow ( int b , int e );
void write_integer (int i);
void assign_lcd_pins(int rs, int e, int d4, int d5, int d6, int d7);
void lcd_init ( void );
```

Figura 3.21: Arquivo cabeçalho do módulo *lcd*.

```
/* DEFINITIONS */

#define NUM_MEDIDAS_CALIBRACAO          100

/* PROTOTYPES */

void calibra_gyro(void);
float calcula_gyro(void);
int gyro_init(void);
```

Figura 3.22: Arquivo cabeçalho do módulo *gyro*.

Antes de se poder utilizar as funções de acesso aos dados do girômetro, deve-se inicializar os periféricos associados ao sensor. As medições são realizadas, conforme visto na seção 2.7, com o auxílio dos conversores *ARM_AD4* e *ARM_AD6*. Ainda, *ARM_PIN5* e *ARM_PIN6* são utilizados como saídas digitais para administração dos sinais de controle do circuito multiplexador associado ao girômetro. A inicialização dos periféricos utilizados pelo módulo então é resumida pela função "*int gyro_init(void)*".

Ao se inicializar os periféricos pela função "*int gyro_init(void)*", é realizada automaticamente a calibração do sensor. De fato, no final da própria rotina de inicialização é chamada a função

"*int calibra_gyro(void)*", como se pode ver nos códigos-fonte em anexo. Esse procedimento de calibração pode ser realizado novamente em um momento posterior chamando novamente a rotina "*int calibra_gyro(void)*". Essa função é responsável pelo cálculo de dois parâmetros de suma importância para o correto cálculo do girômetro: *float V0gyro* e *float ganhoResistivo*. Conforme mencionado na seção 2.7, o valor nulo de voltagem não implica em robô sem rotação. Portanto deve existir um valor de *bias VOgyro* associado às medidas, o qual será estimado pela rotina de calibração. Ainda, a constante de sensibilidade do sensor dada pelo fabricante não é diretamente válida na medição feita pelo conversor analógico-digital por conseqüência da existência do ganho do circuito divisor de tensão. Portanto, de forma a se obter maior exatidão nos valores medidos, deve-se calcular o valor exato do divisor resistivo. É importante observar que os valores nominais dos resistores estão disponíveis no projeto, porém, os mesmos foram relevantes apenas para o cálculo aproximado do valor de divisão resistiva desejado. Na rotina de calibração, calcula-se um valor com maior exatidão para o ganho resistivo, valor o qual independerá de incertezas associadas à fabricação dos resistores e temperatura de operação.

O algoritmo de calibração presente na figura 3.23 ilustra como os parâmetros *float V0gyro* e *float ganhoResistivo* são calculados pela função "*int gyro_init(void)*"no microcontrolador ARM. Conforme pode ser visto nos circuitos das plataformas em anexo e na seção 2.7, a porta 0 do multiplexador corresponde à referencia de 2.5V disponibilizada pelo girômetro e a porta 1 corresponde à saída analógica da variável medida pelo sensor. Observa-se que o algoritmo calculará o valor de *bias* do sensor, portanto, a plataforma deve necessariamente estar sem rotação neste momento. Recomenda-se que a plataforma esteja em completo repouso, i.e., velocidade de translação nula e velocidade de rotação nula.

Explica-se neste momento a lógica de funcionamento da rotina de calibração. Chamadas de *wait()* são executadas logo após cada chaveamento de porta no multiplexador. Essa chamada é absolutamente necessária, pois deve-se esperar a saída do multiplexador se estabilizar antes de coletar a informação. Deve-se notar que a dinâmica predominante é a dinâmica proveniente do filtro passa-baixa incluso no circuito, explicado na seção 2.7. Conforme demonstrado, deve-se esperar aproximadamente 2 milissegundos para se obter o valor correto.

Primeiramente, seleciona-se a porta 0 do multiplexador, a qual é responsável pela referencia de 2.5V disponibilizada pelo girômetro. A partir desta referencia, sabe-se através de *ARM_A4* a magnitude do sinal de entrada do divisor resistivo, e através de ARM_A6 a magnitude do sinal de saída do divisor. Assim, o microcontrolador pode calcular o ganho através de uma divisão dos valores obtidos. Depois de obtido o *ganhoResistivo*, pode-se estimar o valor de *bias* do sensor. Para isso, seleciona-se a porta 1 do multiplexador, a qual é responsável pela saída analógica da variável medida pelo sensor, e faz-se a média aritmética de NUM_MEDIDAS_CALIBRACAO medidas do valor na porta *ARM_A6* com a plataforma parada. Essa média, após aplicado o ganho inverso do divisor resistivo, é o valor de tensão disponibilizado pelo sensor quando a plataforma está sem rotação.

Finalmente, após calibrado, o sensor pode disponibilizar o valor de rotação instantânea da plataforma (em rad/seg) através da função "*float calcula_gyro(void)*". Essa função simplesmente

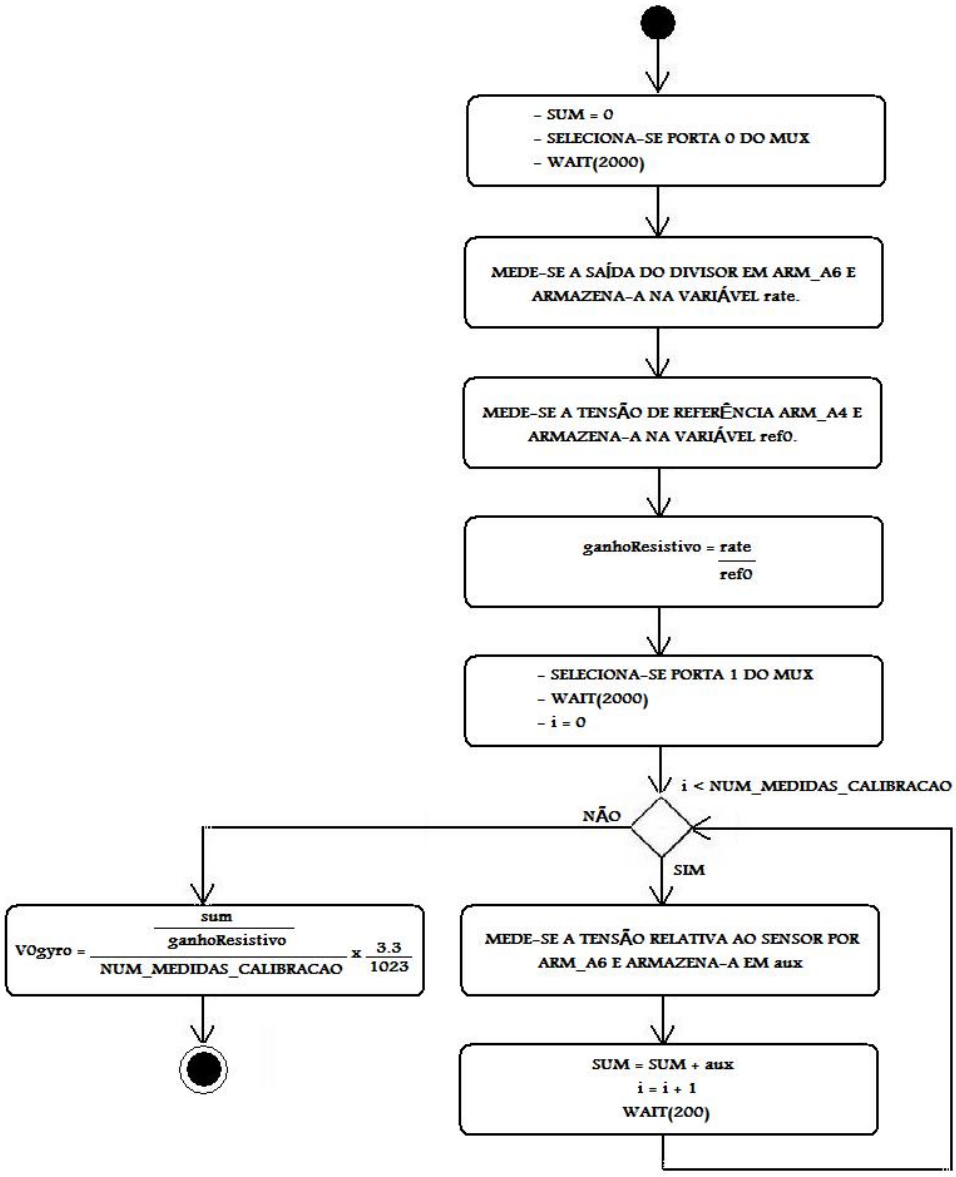

Figura 3.23: Algoritmo de calibração do sensor de rotação da plataforma.

aplica a seguinte fórmula no valor *rate* obtido pelo conversor analógico digital ARM_A6:

$$\omega = \frac{\frac{\frac{rate}{1023} \cdot 3.3}{ganhoResistivo} - V_{0gyro}}{0.005} \cdot \frac{\pi}{180}$$

As constantes que aparecem na fórmula acima foram obtidas teoricamente. *rate* é dividido por 1023 para se obter qual fração da tensão de alimentação do conversor (i.e., 3.3V) corresponde a tensão física na saída do divisor resistivo, dado que se trata de um conversor analógico digital de 10 bits. O valor resultante é divido pelo *ganhoResistivo* de forma a se obter a tensão de saída do sensor de rotação. Subtrai-se o valor de *bias* para linearizar a resposta do sensor. Com a resposta linearizada, divide-se o valor obtido por 0.005, o qual corresponde à sensibilidade do sensor em V/(graus/seg) de acordo com o fabricante. Por último, converte-se a unidade do valor de velocidade para rad/seg através do fator $\pi/180$.

3.2.5 Módulos de odometria e controle de velocidade

Os módulos de odometria e controle serão explanados nesta sub-seção separadamente dado que ambos possuem uma falta de associação direta com um módulo físico. Além dessa característica, existe outra propriedade que os diferem dos demais módulos: o módulo de controle e o módulo de odometria possuem funções que devem ser executadas periodicamente com alta precisão de período. De fato, o ARM executa apenas duas tarefas periódicas, uma atualizando ações de controle e outra atualizando dados de odometria.

Neste momento, explica-se o porquê do tempo de amostragem de 10ms ser um tempo razoável. Após implementadas as tarefas períodicas de controle e odometria, mediu-se o tempo que as mesmas demoram para ser executadas. Essa medição foi realizada através de um pino do ARM que foi definido como saída lógica DEBUG, conforme pode-se ver no código *main.c* em anexo. No momento em que se inicia as tarefas periódicas, seta-se DEBUG como "1"lógico. No momento em que as tarefas terminam sua execução, seta-se DEBUG como "0"lógico. Assim, através de um osciloscópio, pode-se fazer a medição do tempo que as tarefas periódicas gastam ao serem processadas. Esse tempo foi medido pelos autores como 2.5 milissegundos. Assim, 10 milissegundos é tempo o bastante para executar as tarefas periódicas, e ainda sobra 7.5 milissegundos para atividades restantes da rotina principal.

- **control.h**: define funções relativas à tarefa de controle de velocidade das rodas. Vide figura 3.24.

```
#define EMPTY_SPEED          1000000
#define EMPTY                  900000

/* PROTOTYPES */

void controlAction(float * angularSpeedWheelRad1, float * angularSpeedWheelRad2, float *
angularSpeedWheelRad3);
int setParamProporcional(float Kp);
int setReferenceWheelSpeedRad(float angularSpeedWheelRad1, float angularSpeedWheelRad2, float
angularSpeedWheelRad3);
int setParamIntegral(float Ki);
```

Figura 3.24: Arquivo cabeçalho do módulo *control*.

Neste projeto, definiu-se o tempo de estabilização das rodas, i.e., tempo após o qual a velocidade angular das mesmas permanece dentro de uma pequena porcentagem de seu valor final, como sendo 100ms. Caso o tempo de estabilização fosse muito maior que 100ms, a dinâmica da plataforma teria seu desempenho insatisfatório para os fins de robótica móvel, com uma dinâmica muito lenta. Por outro lado, caso o esse tempo fosse muito menor que 100ms, o controle seria inviável fisicamente e relativamente instável. Definiu-se também o período das tarefas como 10ms. Esse valor é apropriado por razões que se tornarão evidentes no final desta sub-seção.

Conforme pode ser visto, esse módulo não possui nenhuma função de inicialização. Porém, para o correto funcionamento do sistema de controle, o atual módulo requer que os motores e os contadores de pulsos do sensor óptico estejam inicializados e ligados. Além disso, o controle só funcionará caso a função "*void controlAction(float * angularSpeedWheelRad1, float * angularSpeedWheelRad2,*

*float * angularSpeedWheelRad3)*"seja chamada a cada 10ms. Neste projeto, utiliza-se o periférico do ARM chamado PIT (*Periodic Interval Timer*) para auxiliar a execução de tarefas periódicas. O PIT é projetado para oferecer máxima acurácia na tarefa de medição do tempo dentro do microcontrolador ARM. A inicialização do PIT, ou inicialização de qualquer outra estratégia que ofereça a solução do agendamento da rotina de controle a cada 10ms, é dever do código que chama a função de controle. A função retorna os valores de velocidade angular de cada roda no instante em que a função é chamada. Essa velocidade angular é estimada através do módulo *counters.h*, já explicado anteriormente.

Para se modificar a referência que a velocidade de alguma roda deve seguir através do sistema de controle deve-se usar a função "*int setReferenceWheelSpeedRad (float angularSpeedWheelRad1, float angularSpeedWheelRad2, float angularSpeedWheelRad3)*". Note que a função sempre recebe três argumentos. Porém, pode existir a ocasião onde se deseje modificar apenas a referencia de uma roda, mantendo as outras constantes. A mesma função anterior implementa esse caso, e na ocasião, insere-se a macro EMPTY_SPEED nas velocidades as quais não se deseja modificar a referencia.

A seguir, segue uma descrição da implementação da estratégia de controle de velocidade das rodas.

Primeiramente, deve-se estimar a velocidade angular instantânea da roda que se deseja controlar. Esse feito é cumprido através do módulo *counters.h*. A cada 10ms, conta-se o número de pulsos emitidos pelo sensor óptico e estima-se a velocidade conforme já explicado na sub-seção 3.2.4. Para obter-se maior precisão nas estimações, *reseta-se* o contador associado logo após a leitura ter sido feita.

Em segundo lugar, calcula-se o erro de referência e gera-se o sinal de controle apropriado segundo uma estratégia de controle digital proporcional-integral com anti-windup por integração condicional, conforme ilustra o diagrama lógico da figura 3.25.

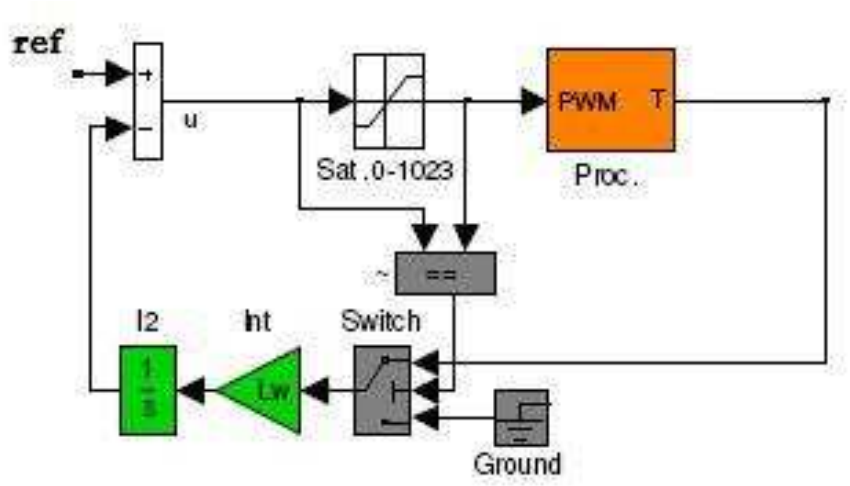

Figura 3.25: Diagrama lógico da lógica de controle integral com anti-windup.

A arquitetura de controle possui dois parâmetros numéricos importantes para a dinâmica final das rodas da plataforma: K_i e K_p. Esse dois parâmetros podem ser modificados pelo robô *on-*

the-fly utilizando a mensagem de comando correspondente encontrada na tabela 3.2 no caso de o controle digital PI estar com o desempenho insatisfatório. As funções que modificam K_i e K_p em nível de ARM são as seguintes: "*int setParamProporcional(float Kp)*"e "*int setParamIntegral(float Ki)*". A resposta em velocidade angular dos motores pode ser analisada no modo de sintonização e do software MATLAB, conforme a seção 3.5 ilustrará.

Em posse do modelo matemático do motor encontrado na seção 2.5 e o modo de operação de sintonização da plataforma, sintoniza-se K_i e K_p tais que a condição de tempo de estabilização igual a 100ms seja respeitado teoricamente. Também deseja-se que o controle de velocidade das rodas a mantenham com comportamentos e respostas semelhantes. Os valores recomendados pelos autores para todos os motores são:

- $K_p = 0.300$
- $K_i = 0.600$

Com os valores de K_p e K_i recomendados, obteve-se as respostas dinâmicas ilustradas pela figura 3.34.

- **odometry.h**: define funções relativas à tarefa de administração do sistema de odometria. Vide figura 3.26.

```
/* DEFINITIONS */

#define DIFF_WHEEL_RADIUS                       0.253
#define DIFF_AXIS_DIST                          1.650
#define OMNI_WHEEL_RADIUS                       0.253
#define OMNI_AXIS_DIST                          1.650
#define SAMPLE_TIME                             0.010

/* PROTOTYPES */

int task_odometry_calcu(int plataform, float angularSpeedWheelRad1, float
angularSpeedWheelRad2, float angularSpeedWheelRad3, float theta_dot);
int reset_odometry();
float ret_x_odometry();
float ret_y_odometry();
float ret_theta_odometry();
int set_x_odometry(float x_odom);
int set_y_odometry(float y_odom);
int set_theta_odometry(float theta_odom);
```

Figura 3.26: Arquivo cabeçalho do módulo *odometry*.

Esse módulo implementa as funções relativas à tarefa de odometria. As funções acima permitem ao usuário resetar a posição odométrica do robô, além de setar esta posição para qualquer valor desejado. Setar a posição é importante em situações ou estudos onde sensores externos ao robô estarão presentes e poderão auxiliar o mesmo em sua navegação.

A odometria de um robô depende de seu modelo cinemático. A cinemática de um robô móvel descreve o movimento deste com relação a um sistema de referência. Admite-se que o robô seja um corpo rígido, e que sua posição e orientação sejam tomados a partir de um referencial inercial $\{X_r, Y_r\}$, com origem em O, e o corpo esteja em um sistema de coordenadas $\{X_m, Y_m\}$ com origem em $P = (x, y)$, de acordo com a figura 3.27.

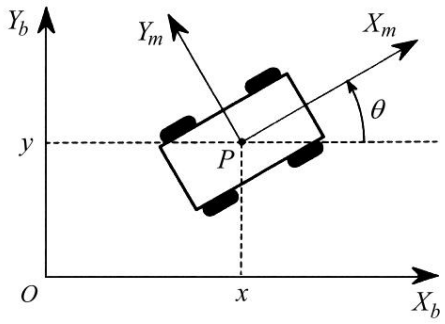

Figura 3.27: Sistema de coordenadas de um robô móvel. Adaptado de [Campion, Bastin e DŠAndréa-Novel 1996].

O modelamento cinemático de um robô diferencial baseia-se nas seguintes equações de movimento, vide [6]:

$$\begin{cases} \dot{x} = v\cos\theta = \frac{(\dot{q_d}+\dot{q_e})}{2} r\cos\theta \\ \dot{y} = v\sin\theta = \frac{(\dot{q_d}+\dot{q_e})}{2} r\sin\theta \\ \dot{\theta} = \frac{\dot{q_d} r}{b} - \frac{\dot{q_e} r}{b} = (\dot{q_d} - \dot{q_e})\frac{r}{b} \end{cases}$$

Em que:

- x, y, θ - representam a posição do sistema de referência do robô em relação ao sistema de referência externo;
- r - raio da roda diferencial;
- b - distância entre as rodas tracionadas;
- $\dot{q_d}$ - velocidade de rotação da roda direita;
- $\dot{q_e}$ - velocidade de rotação da roda esquerda;
- v - velocidade de translação.

A representação matricial das equações acima será:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = \begin{pmatrix} r\cos\theta & r\cos\theta \\ r\sin\theta & r\sin\theta \\ \frac{r}{b} & -\frac{r}{b} \end{pmatrix} . \begin{pmatrix} \dot{q_d} \\ \dot{q_e} \end{pmatrix}$$

Por sua vez, o modelo cinemático de um robô omnidirecional [1,6] como o deste trabalho, que considera possuir o ponto de contato das três rodas exatamente nos vértices de um triângulo equilátero, é dado por:

$$\begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{pmatrix} = -r \begin{pmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} . \begin{pmatrix} \dot{\phi_1} \\ \dot{\phi_2} \\ \dot{\phi_3} \end{pmatrix}$$

Em que:

- r - raio das rodas omnidirecionais;
- b - distância de cada roda ao centro do robô(considerado a mesma para todas);
- θ - rotação do sistema de referência do robô em relação ao sistema principal;
- ϕ_1, ϕ_2, ϕ_3 - são as posições angulares das rodas.

Considera-se como sentido positivo de rotação das rodas, quando todas as rodas colocadas para girar no mesmo sentido fazem a rotação resultante do robô ser no sentido antihorário. A identificação de cada roda foi realizada no sentido antihorário de modo numérico crescente, a partir da frente do robô omnidirecional, que é considerada como o lado da plataforma que apresenta o display LCD.

Assim, apresentados os modelos cinemáticos contínuos das duas plataformas utilizadas neste trabalho, contrói-se computacionalmente a função "*void task_ odometry_ calcu(float *angularSpeedWheelRad1, float *angularSpeedWheelRad2, float *angularSpeedWheelRad3, float theta_ dot)*", a qual é responsável por integrar as equações cinemáticas discretamente pela *fórmula de Euler* [3] e obter as poses atuais dos robôs. As variáveis necessárias para o cálculo ($\dot{\phi}_i$ e θ) são obtidas pelos sensores internos ao robô. No caso, $\dot{\phi}_i$ são obtidas através dos encoders ópticos e θ pode ser obtida através da integração das próprias equações cinemáticas ou através da integração dos valores do sensor girômetro incluso na plataforma. Ambos métodos de estimação de θ irão ser postos à prova ainda nesta seção.

O sistema de odometria só funcionará caso a função "'*void task_ odometry_ calcu(float *angularSpeedWheelRad1, float *angularSpeedWheelRad2, float *angularSpeedWheelRad3, float theta_ dot)*"' seja chamada a cada 10ms. Neste projeto, utiliza-se o periférico do ARM chamado PIT (*Periodic Interval Timer*) para auxiliar a execução de tarefas periódicas, conforme já mencionado. A inicialização do PIT, ou inicialização de qualquer outra estratégia que ofereça a solução do agendamento da rotina de controle a cada 10ms, é dever do código que chama a função de cálculos de odometria. Os parâmetros dessa função podem ser obtidos através da rotina de controle já mencionada. Portanto, no algoritmo periódico associado ao PIT, chama-se primeiramente a função de controle. Através da função de controle obtem-se todos os parâmetros da função de odometria, exceto o parâmetro *theta_ dot*. Esse parâmetro pode ser obtido através das equações mencionadas no capítulo de Introdução ou através do sensor girômetro (ou às vezes atráves dos dois simultaneamente, com algorirmos de fusão sensorial que não fazem parte do escopo deste trabalho).

Utilizou-se um experimento simples para escolher entre a estimação de *theta_ dot* por equações cinemáticas ou por medições do sensor girômetro. A plataforma foi conduzida através do laboratório até retornar ao ponto inicial de onde ela partiu. Idealmente, deve-se ter a posição final retornada pela odometria como sendo a posição (x=0, y=0, θ=0). Os resultados foram os seguintes (em metros e radianos):

- Odometria por girômetro: (x=1,2 ; y=1,0 ; θ=0,2)
- Odometria por equações cinemáticas: (x=0,05 ; y=0,05 ; θ=0,0)

Assim, observou-se que o cálculo de odometria pelo método por equações cinemáticas realizou o experimento com maior precisão. Portanto, a plataforma atualmente utiliza este método como método de cálculo de odometria. No futuro, pode-se realizar um estudo sobre fusão sensorial e talvez melhorar ainda mais a resposta da odometria.

3.2.6 Algoritmo Principal - main.c

Conhecidos todos os módulos em software, pode-se neste momento dissertar acerca da estrutura do algoritmo principal rodado no microcontrolador ARM. Relembra-se aqui os objetivos deste algoritmo, os quais se remetem aos deveres da plataforma perante o EEEPC:

- Pronto atendimento das mensagens de requisição;
- Pronto atendimento das mensagens de comando;
- Controle digital de velocidade dos motores;
- Cálculos odométricos.

Observa-se que se deve ter um algoritmo que de alguma forma implemente uma escuta na interface RS-232 à espera de mensagens de requisição ou comando enquanto ocorrem tarefas periódicas de controle digital de velocidade e odometria. O algoritmo descrito pela figura 3.28 ilustra como foi implementado em ARM o código que satisfaz os requisitos listados.

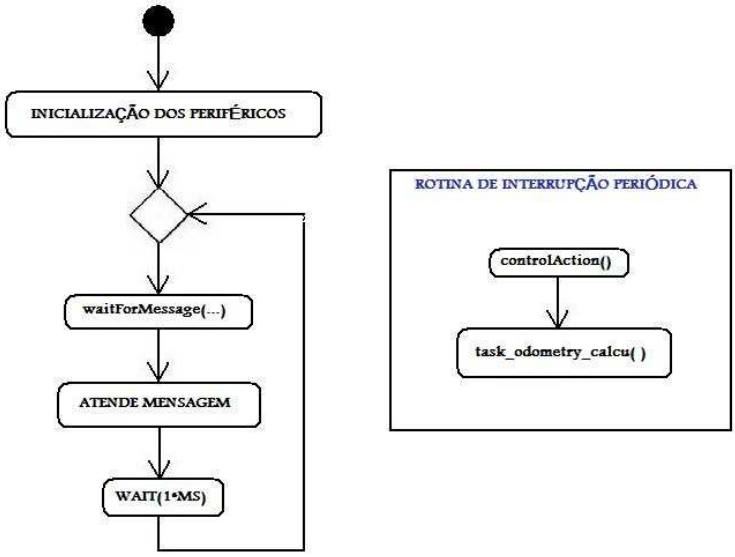

Figura 3.28: Algoritmo principal rodado no microcontrolador ARM.

A inicialização dos periféricos se resume à utilização das funções de inicialização de cada módulo e à configuração e habilitação do PIT com 10ms de período. Após a inicialização dos periféricos, entra-se num laço de repetição onde a primeira ação é travar o fluxo de processamento do código até que uma mensagem válida seja recebida pela interface RS-232. Após o recebimento da mensagem, a mesma é tratada. Caso seja uma mensagem de requisição, o dado requisitado é enviado para

o mestre EEEPC pela interface RS-232. Por outro lado, se for uma mensagem de comando, a plataforma imediatamente mudará seu estado e sua configuração para atender ao comando. Após processada a mensagem, espera-se aproximadamente 1 milissegundo e volta-se a travar o fluxo de processamento do código até que uma nova mensagem válida seja recebida.

O algoritmo citado no parágrafo anterior pode ser interrompido a qualquer momento por interrupções. No caso, a rotina de interrupção será executada com o endereço da instrução atual da rotina principal armazenado em uma pilha, de forma que, após o termino da rotina de interrupção, o fluxo de processamento possa retornar de onde parou, i.e., o endereço armazenado na pilha. Neste projeto, habilita-se apenas uma interrupção no microcontrolador: a interrupção periódica PIT com o valor de 10ms configurado. Essa interrupção garante a periodicidade das tarefas de controle e odometria.

3.3 API em C para EEEPC

A seção 3.2 discorreu acerca o software instalado no microcontrolador. Deixou-se em evidência a interface de comunicação entre o ARM e o EEEPC. Para que o EEEPC controle as plataformas, o mesmo deve enviar mensagens de comando e mensagens de requisição para o microcontrolador. Com o intuito de facilitar o desenvolvimento e pesquisa em cima das plataformas, criou-se neste projeto uma API para acesso aos recursos das plataformas. Esta API tem o dever de abstrair a camada de comunicação. Assim, qualquer programador tem o poder de facilmente programar aplicativos que controlem as plataformas sem a necessidade de um conhecimento profundo sobre a arquitetura física das mesmas.

O estudo do ambiente necessário ao desenvolvimento será apresentado na sub-seção 3.3.1. As facilidades que esta API provê serão discutidas na sub-seção 3.3.2.

3.3.1 Ambiente de Desenvolvimento em EEEPC

O EEEPC vem de fábrica com um sistema operacional chamado *Xandros* em uma versão otimizada para rodar no EEEPC. Essa otimização visa usuários que desejem um notebook extremamente portátil para rodar aplicações como editores de texto, navegadores de Internet, programas multimedia e *chats*. Essa otimização não visa os usuários que desejem desenvolver aplicativos. Pelo contrário, o Xandros foi considerado pelos autores deste trabalho como um ambiente relativamente hostil para programadores. Neste trabalho necessitou-se instalar as bibliotecas de visão computacional *openCV*[10] e as bibliotecas do PLAYER. Essas últimas serão abordadas na seção 3.4.

OpenCV (*Open Source Computer Vision*) é uma biblioteca de funções destinadas principalmente ao processamento de imagens em tempo real na área de visão computacional. Aplicações típicas desta biblioteca são identificações de objetos, segmentação e reconhecimento, reconhecimento de faces, reconhecimento de gestos, rastreamento de movimentos, robótica móvel, e muitas outras. No presente trabalho, a biblioteca foi necessária para se fazer a obtenção de imagens da

[10]openCV pode ser obtido em http://opencvlibrary.sourceforge.net/

câmera e enviá-las pela rede. Mais informações acerca esse assunto, vide sub-seção 3.4.3. Para instalação completa e correta desta biblioteca deve-se ter na máquina instalados os seguintes pacotes:

- GCC 4.x ou mais atual;
- CMake 2.6 ou mais atual;
- Cliente Subversion (SVN)
- GTK+ 2.x ou mais atual, incluindo *headers* (por exemplo, libgtk2.0-dev);
- pkgconfig;
- libpng, zlib, libjpeg, libtiff, libjasper com arquivos de desenvolvimento (por exemplo, libjpeg-dev);
- Python 2.3 ou mais atual com pacotes de desenvolvimento (por exemplo, python-dev);
- SWIG 1.3.30 ou mais atual;
- libavcodec pelo programa ffmpeg 0.4.9-pre1 ou mais atual com arquivos de desenvolvimento;
- libdc1394 2.x com arquivos de desenvolvimento para captura de imagens de cameras *firewire*.

Da lista de requisitos acima, o pacote GTK+ 2.0 ao ser instalado leva o sistema Xandros à instabilidade. Os autores julgaram mais interessante ao projeto instalar outro sistema operacional no EEEPC. Ao se pesquisar na rede, os autores descobriram o sistema *eeeBuntu*. O sistema eeeBuntu[11] é baseado no sistema *Ubuntu 9.04, Jaunty Jackalope*. O sistema Ubuntu possui uma ampla documentação na Internet, e por isso, o mesmo foi aderido de imediato ao projeto.

Na época da redação deste trabalho, o sistema eeeBuntu era distribuído em três versões: *Standart*, *NBR*, *Base*. A versão *Base* foi a versão escolhida pelos autores por ser a versão mais enxuta do sistema operacional. Com posse de uma versão limpa, pode-se adaptá-la com mais liberdade e mais espaço em disco para os fins de robótica móvel. O processo de instalação do sistema pode ser auxiliado por ferramentas do Ubuntu. Os autores recomendam instalar usando-se das ferramentas por se tratar de uma estratégia mais simples e pelo fato de que o EEEPC não possui CD-ROM, fator esse que pode tornar a instalação por dispositivos removiveis frustante.

Para instalar, fez-se o *download* da ferramenta *UNetbootin*[12] e da imagem do eeeBuntu Base. Pode-se baixar a ferramenta para sistema Linux ou para sistema Windows. Neste trabalho, trabalhou-se com a versão Windows. Depois de obtida a ferramenta, instalou-se a imagem do sistema em um pen drive através da interface amigável do UNetBootin. Finalmente, com o sistema no pen drive, instala-se o sistema no EEEPC rodando o pen drive como *boot*.

Para compilação do código-fonte, foi utilizado durante este projeto o compilador *GCC 4.3.2*. Por se tratar de uma API de pequeno porte, não foram desenvolvidos nem arquivos Makefile nem um ambiente para controle de versão. De fato, a API se resume a apenas um arquivo cabeçalho *motores_sinale.h* e outro arquivo contendo o códigos das funções *motores_sinale.c*.

[11] eeeBuntu pode ser obtido em http://www.eeebuntu.org/
[12] UNetbootin pode ser obtido em unetbootin.sourceforge.net/

3.3.2 Funções Disponíveis

As funções disponibilizadas pela API desenvolvida neste trabalho são listadas no arquivo cabeçalho *motores_sinale.h* ilustrado na figura 3.29. Elas podem facilmente ser correlacionadas diretamente com a interface de comandos e requisições definida na seção 3.2.2.

```
#define BUFSIZE                  9

#define OMNI                     1
#define DIFF                     2

#define IR1                      1
#define IR2                      2
#define IR3                      3
#define IR4                      4
#define IR5                      5
#define IR6                      6
#define GYRO                     7

#define DIFF_WHEEL_RADIUS        0.0253
#define DIFF_AXIS_DIST           0.1650
#define OMNI_WHEEL_RADIUS        0.0253
#define OMNI_AXIS_DIST           0.1650
#define SAMPLE_TIME              0.0100s

int robot_init(const char* linux_addr); /* return file descriptor witch contains the Motores
Address */
int robot_shutdown(); /* closes file descriptor witch continains Motores Address */

int set_plataform(int plataform); /* sets wich plataform this code is used for */

int cmd_vel_roda_1(float * rad_p_seg); /* sends a velocity reference to the motor 1 */
int cmd_vel_roda_2(float * rad_p_seg); /* sends a velocity reference to the motor 2 */
int cmd_vel_roda_3(float * rad_p_seg); /* sends a velocity reference to the motor 3 */
int turn_off_motors(); /* turn motors energy off */
int turn_on_motors(); /* turn motors energy on */
int set_Kp(float Kp); /* configures parameters of control rotine */
int set_Ki(float Ki); /* configures parameters of control rotine */

float req_value_sensor(int sensor); /* requests value from a IR sensor: IRi, where i belongs
to [1,6] */
int turn_on_sensors(); /* works only with IR sensors */
int turn_off_sensors(); /* works only with IR sensors */

/* odometry fuctions, depends on wich robot is used, MUST set plataform before using these
ones.. */
float req_value_x_meters();
float req_value_y_meters();
float req_value_theta_rad();
int reset_odometry();
int set_value_x_meters(float value);
int set_value_y_meters(float value);
int set_value_theta_rad(float value);
int set_vel(float x_dot, float y_dot, float theta_dot);
```

Figura 3.29: Arquivo cabeçalho da API de desenvolvimento no EEEPC para as plataformas.

3.4 PLAYER

Serão abordados nesta seção vários aspectos relativos ao desenvolvimento de aplicativos para a plataforma PLAYER. Dependendo do objetivo do usuário e seu conjunto de plataformas robóticas (sejam físicas ou virtuais), o plano de desenvolvimento poderá variar. Aqui serão cobertos assuntos como quais objetivos o PLAYER possui e quais problemas o mesmo pretende solucionar. Além

desses, será discutido como se carrega corretamente o driver das plataformas móveis desenvolvidas pelos autores deste trabalho, e quais parâmetros podem ser modificados no arquivo de configuração do driver. Será visto também como imagens da câmera embarcada em uma plataforma podem ser acessadas pela rede TCP/IP na qual a mesma se encontra.

Neste projeto se executou um plano de desenvolvimento relativamente extenso, cobrindo todos os aspectos de desenvolvimento na plataforma PLAYER, exceto aspectos relacionados à simulação. Para trabalhar em um ambiente de simulação é necessário o software *Stage* para simulações 2D ou *Gazebo* para simulações 3D. Trabalhos futuros poderão adicionar essa capacidade de simulação ao ambiente de desenvolvimento das plataformas, mas atualmente tal capacidade não é existente.

3.4.1 Introdução ao PLAYER

De acordo com o website oficial do projeto, o PLAYER é um servidor de rede grátis e open-source sob a licença pública GNU para controle de robôs rodado em Linux. Rodando em um robô, o PLAYER provê uma interface padronizada limpa e simples para os sensores e atuadores do mesmo sobre uma rede TCP/IP ou UDP/IP. Um programa cliente conversa com o PLAYER sobre um socket de família de endereçamento *AF_INET* [4], lendo dados dos sensores e escrevendo comandos para os atuadores, além de configurar dispositivos *on-the-fly*.

O software PLAYER é independente de plataforma ou linguagem de programação. Inclusive, existem versões não oficiais das bibliotecas de desenvolvimento de clientes com suporte em linguagem JAVA. Assim, pode-se até acessar um determinado conjunto de robôs através de clientes rodados em plataformas computacionais móveis que possuam a máquina virtual JAVA, tais como *Palm*, *Pocket PC*, celulares, etc. Ainda, um mesmo programa cliente compilado em JAVA pode ser executado tanto no Palm quanto em um telefone móvel *Nokia*, devido à portabilidade inerente à linguagem JAVA. Um cliente pode rodar então em qualquer máquina que possua uma conexão por rede para um robô, e ainda, um cliente pode ser escrito em qualquer linguagem de programação que possua compatibilidade com sockets da família de endereçamento AF_INET. Atualmente existem bibliotecas de desenvolvimento para clientes PLAYER em uma ampla gama de linguagens de programação, entre elas C++, Tcl, Java, e Python. Ainda, o PLAYER não faz nenhuma restrição à arquitetura da estrutura do código do cliente. Neste sentido, pode-se ter códigos com inúmeras tarefas rodando concorrentemente uma às outras ou pode-se ter códigos aonde a lógica de controle é ditada apenas por um loop interminável. Outra opção seria um cliente que disponibilizasse controle interativo através de um cliente Tcl.

Existem três conceitos extremamente importantes quando se trabalha com o ambiente de desenvolvimento PLAYER:

- *Interface*: Uma especificação de como interagir com uma determinada classe de sensor, atuador ou algoritmo. A interface define então a sintaxe e a semântica de todas as mensagens que podem ser trocadas com entidades da mesma classe. A lista de interfaces disponíveis é dada pela tabela 3.4.
- *Driver*: Software normalmente escrito em C++ que conversa com os dispositivos robóticos (sensores, atuadores, algoritmos, etc) e traduz suas entradas e/ou saídas em parâmetros per-

Tabela 3.4: Interfaces definidas na versão PLAYER 2.1.0

localize	speech	speech_recognition	joystick	pointcloud3d
ranger	camera	player	position3d	blackboard
position1d	graphics2d	mcom	wifi	graphics3d
planner	vectormap	audio	gripper	aio
laser	blobfinder	dio	actarray	ir
blinkenlight	fiducial	map	wsn	health
ptz	sonar	simulation	log	limb
gps	power	position2d	bumper	rfid
opaque	imu	—	—	—

tencentes a determinadas interfaces existentes no robô. O objetivo do driver é então esconder os detalhes específicos de cada hardware e unificar o comportamento dos vários componentes de uma mesma classe. Por exemplo, espera-se que um sensor medidor de distância retorne valores reais positivos, independentemente do fabricante. Cada robô comercial terá um driver, mas todos os *drivers* implementarão a mesma interface *ranger*.

- *Device*: Um robô pode ter mais de um dispositivo associado à uma mesma interface. Por exemplo, cada plataforma deste trabalho possui seis sensores infravermelho para medições de distâncias. Cada um desses sensores será associado à uma interface ranger, e assim, torna-se necessário um índice para identificar cada dispositivo de uma mesma interface. Esse dígito é o Device.

O PLAYER, portanto, permite que múltiplos dispositivos ou robôs possuam a mesma interface. Por exemplo, o Pioneer 2 e as plataformas deste projeto se utilizam da interface *position2d*, a qual permite controle de um robô em um plano horizontal. Assim, exatamente o mesmo código cliente pode acessar os atuadores de um robô como os do outro. O conceito demonstrado é equivalente ao de uma máquina virtual de um sistema operacional. Quando se programa em C++/Linux um processo que armazena em disco determinadas informações, não é necessário saber qual modelo ou fabricante do hardware está sendo utilizado. Uma série de funções do sistema operacional, nomeadas *chamadas de sistema*, abstraem o hardware utilizado de forma que o usuário não necessite saber informações específicas de cada dispositivo, por exemplo, a localização do cabeçote de leitura e gravação, quantas trilhas, setores, cilindros existem, etc.

A discussão do parágrafo anterior pode induzir o leitor a acreditar que o PLAYER é um sistema operacional para robôs. Essa proposição pode ser verdadeira ou falsa dependendo da definição de sistema operacional que o leitor aceite. Os autores deste projeto seguem o paradigma definido em [5], o qual postula que um sistema operacional deve ser uma entidade que provê uma máquina virtual para diversos tipos de *hardware* (conforme comentado anteriormente), e que seja um escalonador de processos, de tal forma que os recursos físicos disputados possam ser usados por múltiplos processos concorrentemente no tempo. No caso, o PLAYER não escalona as requisições dos clientes (i.e., processos) quando múltiplos clientes estão conectados em um robô apenas. Portanto, os autores acreditam que o PLAYER atualmente não é um sistema operacional

para robôs.

O PLAYER é projetado também para suportar virtualmente qualquer número de clientes. Os robôs podem a qualquer momento conhecer a estrutura organizacional do conjunto de robôs no qual eles estão inseridos e as capacidades de cada um dinamicamente, permitindo que os mesmos possam alterar on-the-fly suas estratégias individuais de forma a atingir uma tarefa global de forma mais eficaz e portável. Ainda, o PLAYER é um software livre, publicado sobre a licença pública GNU. Este software, portanto, é o candidato ideal para ser usado como plataforma de desenvolvimento de software em robótica de cooperação, o qual é o contexto no qual este trabalho está inserido.

Neste momento, explanar-se-á como se pode inicializar o servidor PLAYER corretamente. Assim como ocorre com inúmeros outros aplicativos servidores, o PLAYER requer um arquivo de configuração como entrada que especifica como o mesmo deve se comportar. O arquivo de configuração é altamente dependente da plataforma utilizada, e assim, do driver implementado. Por exemplo, os drivers carregados pelo PLAYER são listados pelo arquivo de configuração. Além da lista de drivers à serem carregados, o arquivo *.cfg* também expõe informações acerca de como estes serão carregados e quais interfaces os mesmos deverão disponibilizar e de quais interfaces necessitam para o correto funcionamento. O arquivo de configuração exibido na figura 3.30 ilustrará os conceitos visto até então.

```
Driver
(
  name "p2os"
  provides ["odometry:::position2d:0"
            "compass:::position2d:1"
            "gyro:::position2d:2"]
)
driver
(
  name "camerauvc"
  provides ["camera:0"]
  port "/dev/video0"
  size [640 480]
)
```

Figura 3.30: Arquivo de configuração exemplo para a plataforma Pioneer 2.

O arquivo acima carregará dois drivers concorrentemente. Cada driver é uma *Thread* dentro do PLAYER. No caso acima, serão carregados os drivers *p2os* e *camerauvc*. O driver p2so é o driver da plataforma Pioneer 2, o qual provê na rede TCP/IP o sistema de odometria do robô (através da interface *position2d*), o sistema de compasso (através de outra interface *position2d*) e os dados do girômetro (através de mais uma interface *position2d*). Observe a semântica de cada Device no arquivo de configuração:

provides ["KEY:HOST:ROBOT:INTERFACE:INDEX"]

- *KEY*: o propósito deste campo é permitir que um driver que suporte várias interfaces do

mesmo tipo mapeie essas interfaces em diferentes dispositivos através de apelidos, por exemplo, *odometry* e *compass* no arquivo de configuração acima.

- *HOST*: é o IP da máquina onde algum servidor PLAYER está sendo executado. Caso este campo esteja vazio, como no arquivo de configuração acima, o IP padrão é *localhost*, i.e., 127.0.0.1, o endereço de *loopback*.
- *ROBOT*: Porta TCP do nó host indicado no campo HOST na qual o software PLAYER está escutando. Quando se está em vazio, como no arquivo de configuração acima, a porta padrão será utilizada, no caso a porta 6665, destinada para o PLAYER normalmente.
- *INTERFACE*: Algum valor pertencente à tabela 3.4. Este campo não pode estar vazio.
- *INDEX*: *Device* alvo. Esse campo não pode estar vazio.

Portanto, caso o robô Pioneer tenha seu servidor instalado no endereço IP 192.168.0.56, um cliente remoto de endereço 192.168.0.1 pode acessar a velocidade angular deste através de uma mensagem de requisição de dados da interface position2d:2 para o endereço "192.168.0.56:6665".

Além de carregar o driver do Pioneer, este arquivo de configuração também carrega o driver *camerauvc*. Observe que o driver provê a interface câmera na rede. Além da palavra chave *provides*, a qual determina quais interfaces estarão disponíveis na rede pelo driver, existem também as palavras chave *port* e *size* definindo o comportamento deste driver. Estas palavras chave não apareciam no driver anterior. Um driver pode ter palavras chave específicas, as quais não podem ser encontradas em outros drivers. Portanto, sempre é interessante ler o manual de cada driver para se obter ótima performance. No caso, a palavra *port* indica em qual arquivo especial do Linux está o dispositivo físico e a palavra *size* define o tamanho esperado do frame obtido pelo dispositivo em pixels.

Apesar de o PLAYER possuir várias interfaces para disponibilização na rede das imagens da câmera embutida na plataforma, as mesmas não serão utilizadas. Vários FAQs na Wide World Web indicam que o PLAYER não transfere as imagens de forma eficaz na rede, gerando atrasos e congestionamentos. De fato, os autores implementaram a transferência de imagens pelo PLAYER e obtiveram resultados insatisfatórios, além do fato de alguns drivers de câmera nem chegarem a obter imagens. Portanto, neste projeto utilizou-se um software externo produzido pelos autores para rápida transferência de imagens, o qual será mais bem explanado na sub-seção 3.4.3.

Por último nesta seção, ilustrar-se-á os conceitos vistos até então com o auxílio do software cliente *PLAYERV*. PLAYERV é um cliente GUI que dispõe ao usuário uma visualização dos dados dos sensores obtidos do servidor remoto em tempo real. O PLAYERV necessita das bibliotecas de desenvolvimento GTK+-2.0. Essas bibliotecas não funcionam corretamente sobre o sistema operacional nativo de fábrica Xandros do EEEPC. Os autores enfrentaram inúmeros obstáculos ao tentarem instalar as bibliotecas relativas ao GTK+-2.0 no Xandros. Após inúmeras tentativas e incontáveis frustrações, os autores decidiram por instalar no EEEPC outro sistema operacional. A escolha do sistema operacional, assim como sua instalação, pode ser visualizada na sub-seção 3.3.1.

Tabela 3.5: Interfaces de dados suportadas pelo PLAYERV na época de redação deste relatório.

blobfinder	bumper	fiducial	gripper	ir
laser	localize	map	position2d	power
ptz	sonar	wifi	—	—

Tabela 3.6: Interfaces de comando e teleoperação suportadas pelo PLAYERV na época de redação deste relatório.

position2d	ptz	—	—	—

PLAYERV pode atualmente, i.e., no momento da redação deste relatório, visualizar dados provenientes das interfaces contidas na tabela 3.5.

PLAYERV pode atualmente, i.e., no momento de redação deste relatório, teleoperar as interfaces contidas na tabela 3.6.

De forma a se poder utilizar o PLAYERV, carregou-se o servidor PLAYER em uma das plataformas robóticas deste trabalho. O endereço IP deste servidor foi designado pelo DHCP local e é dado por 192.168.0.86. A porta TCP utilizada foi a porta TCP padrão do programa, i.e. 6665. O cliente foi rodado em um computador Desktop remoto com IP dado por 192.168.0.39. O PLAYERV foi inicializado com a seguinte linha de comando.

 # playerv -h 192.168.0.86

De imediato, o cliente gera uma conexão com o servidor e o robô está prontamente acessível ao usuário do Desktop. A interface do cliente PLAYERV é ilustrada pela figura 3.31. Trata-se de uma ferramenta visual extremamente intuitiva e amigável ao usuário.

De início, a tela inicial do programa é vazia. Porém, ao se inscrever à um dispositivo seja um sensor ou atuador, informações ou comandos inerentes ao mesmo aparecerão na tela. A inscrição é feita através da aba Devices. Dentro desta mesma aba estão disponíveis todas as funções de cada interface do robô. Por exemplo, pode-se ter uma imagem visual da configuração dos sensores, conforme indica a figura 3.31. Ainda, pode-se facilmente movimentar a plataforma móvel atrás do mouse se o driver do robô possuir uma interface *position2d* completamente implementada.

O PLAYERV não é o único cliente disponível oficialmente pelo projeto PLAYER. Vários outros podem ser encontrados ao se visitar o website do projeto PLAYER. Neste presente trabalho foram desenvolvidos apenas pequenos clientes para teste e depuração da plataforma, os quais não serão descritos neste texto por serem irrelevantes em comparação ao PLAYERV.

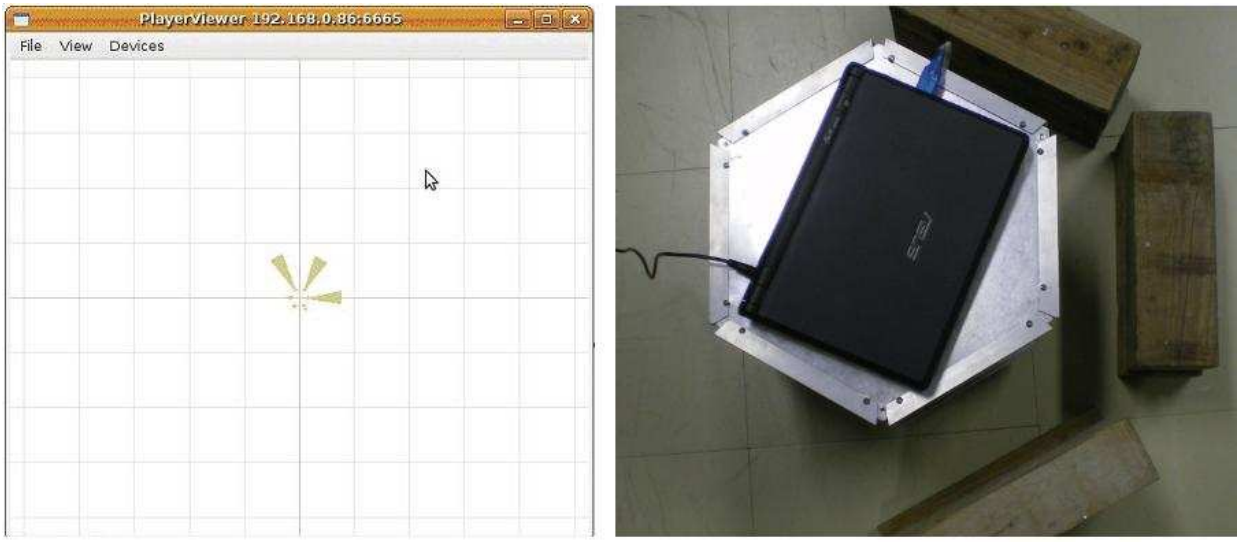

Figura 3.31: Á esquerda, aspecto gráfico do software cliente PLAYERV. À direita, configuração do robô física associada.

3.4.2 Arquivo de configuração e driver do robô

Ao contrário dos drivers *p2os* e *camerauvc* citados na sub-seção 3.4.1 anteriormente, o driver das plataformas robóticas deste trabalho não estão inclusos no projeto oficial do PLAYER. De fato, o driver desenvolvido neste trabalho não possui nem a pretensão de se tornar parte integrante do projeto PLAYER, pois o driver *driverMOTOR* desenvolvido é designado especialmente para as plataformas robóticas desenvolvidas aqui, as quais não serão produzidas em massa. As plataformas aqui desenvolvidas são protótipos para estudos e experimentos em robótica de cooperação no laboratório LARA da Universidade de Brasília. Nesta sub-seção se dará a descrição completa dos parâmetros que podem ser alterados e configurados ao se carregar o driver *driverMOTOR* no ambiente PLAYER.

Um exemplo típico de arquivo de configuração associado às plataformas móveis Jessi XX e Kyle XY é dado pela figura 3.32, o qual pode também ser encontrado nos códigos fonte em anexo ou no DVD deste trabalho.

```
Driver
(
  name "driverMOTOR"
  plugin "driverMOTOR.so"
  provides ["position2d:0" "sonar:0"]
  plataform "DIFF"
  ir_m 0.000147341
  ir_b 0.000442022
  ir_k 4
  alwayson 1
)
```

Figura 3.32: Arquivo de configuração exemplo para a plataforma Jessi XX.

Observam-se palavras chaves comuns encontradas em outros drivers, tais quais *name* e *provides*. Existe uma diferença entre o parâmetro *name* e o parâmetro *plugin*. O parâmetro *name* diz ao PLAYER o nome do driver utilizado. O parâmetro *plugin* diz ao PLAYER qual biblioteca dinâmica carregar para se buscar as instruções referentes ao funcionamento do driver.

O parâmetro *provides* define as interfaces disponíveis. A interface *position2d:0* é responsável pelos comandos de velocidade das rodas e o sistema de odometria. A interface *sonar:0* é responsável pelo cinturão de infravermelho do robô. Observa-se que se escolheu a interface *sonar:0* para se representar os sensores infravermelho, ao invés da escolha lógica de interface *ir:0*, pois a interface *ir:0* possui campos e parâmetros desnecessários para este projeto. Por exemplo, a interface *ir:0* possui campos para as voltagens brutas lida nos sensores, além dos campos de distâncias lidas. A interface *sonar:0* por sua vez, possui apenas os campos de distâncias lidas, os quais são os campos realmente importantes quando se deseja projetar um sistema portável. Voltagens são inerentes ao tipo de sensor utilizado e devem ser omitidas na rede para se disponibilizar apenas informação que todos os programas clientes possam usar. Como conseqüência deste fato também, a interface *sonar:0* é mais enxuta, fator este que aprimora a velocidade das comunicações e troca de dados entre cliente e servidor.

O parâmetro *alwayson* determina o período de tempo no qual o driver estará ligado. Caso o parâmetro seja setado como 1, o driver estará funcionando desde a inicialização do servidor até o momento de sua terminação. Por outro lado, caso o parâmetro seja setado 0, o driver estará funcionando apenas quando um cliente tiver aberto uma conexão com o servidor. Este parâmetro é utilizado em todos os arquivos de configuração de todos os drivers.

O parâmetro *plataform* determina com qual plataforma o servidor irá se comunicar através do driver. Comenta-se aqui a extrema e completa portabilidade de códigos entre as duas plataformas Kyle XY e Jessi XX. Os códigos em ARM para a plataforma Kyle XY e para a plataforma Jessi XX, conforme já foi comentado no capítulo 2, são exatamente idênticas (a estratégia de projeto dessa igualdade de códigos não é trivial, pode-se, por exemplo, citar as funções odométricas que necessitam de ser diferentes por conta das geometrias físicas distintas de cada robô). Portanto, em algum momento no software se deve fazer a distinção entre plataformas. Esse momento é no arquivo de configuração do driver. O arquivo de configuração do driver não precisa ser re-compilado e pode facilmente ser modificado, o que o torna tão atraente para possuir a responsabilidade de definir o tipo de plataforma na qual o servidor está conectado.

Conforme explicado na seção 3.2.2, a calibração dos sensores infravermelho é responsabilidade do administrador do servidor PLAYER. Os parâmetros ir_m, ir_b e ir_k são os parâmetros da curva de calibração do sensor conforme a interface disponibilizada pelo MATLAB explicitada na seção 3.5.2 deste relatório.

3.4.3 Uso da câmera remota

Conforme já comentado , o software PLAYER realizou a transferência de imagens pela rede TCP/IP com um desempenho lento. Havia um grande atraso no recebimento de imagens pelo driver *camerauvc* através do cliente PLAYERV. Portanto, os autores deste trabalho desenvolveram

dois aplivativos para envio de imagens pela rede, os quais fucionaram com melhor desempenho, e portanto, serão utilizado nos estudos posteriores envolvendo as plataformas aqui desenvolvidas. Trata-se de um servidor e um cliente trocando dados através de um socket. Os aplicativos *client.cpp* e *server.cpp* associados podem ser conferidos pois se encontram em anexo.

3.5 MATLAB

Conforme apresentado na sub-seção 3.2.2, o robô em operação normal possui uma interface de comandos e requisições pela qual o EEEPC pode comandar o mesmo. Porém, poderão existir ocasiões em que se deseja avaliar o comportamento e desempenho do controlador de velocidade. Essa característica não é disponível pelo modo de operação normal (definido neste projeto como *modo de operação cruzeiro*). Ainda, foi definido na seção 3.2.4 que o responsável pela calibração dos sensores infravermelho seria o PLAYER. Assim, é conveniente se criar uma interface onde o administrador do servidor PLAYER possa facilmente obter as medições dos sensores. Portanto, dois modos de operação foram desenvolvidos. Nesses modos, basicamente, o robô apenas envia dados para um *host* durante um certo período de tempo. Após o envio dos dados, a plataforma volta para o modo de operação cruzeiro e pode ser comandada pela interface de comando e requisições. Para visualização dos dados, recomenda-se o software MATLAB. Neste software, foram desenvolvidos modelos em simulink para processamento dos dados. Os modelos desenvolvidos e a forma pelo qual os dados são transmitidos é discutido nas sub-seções a seguir.

3.5.1 Obtenção de dados do modo de operação *calibração*

Quando se entra neste modo, as plataformas irão executar o movimento ditado na seção 3.2.2. Paralelamente ao movimento, a plataforma enviará dados periodicamente (10ms) ao MATLAB informando a velocidade de seus motores (inteiro *int* 16 bits na unidade graus/seg) e a referência de velocidade (inteiro *int* 16 bits também em graus/seg) que o mesmo está tentando seguir. O modelo que permite ao simulink receber esses dados é ilustrado na figura 3.33. Neste modelo também está ilustrado o protocolo de comunicação de dados utilizado. Por ser um modo de depuração, trata-se de um protocolo muito pouco robusto. Este possui apenas um caracter iniciador e terminador (no caso '#', ou 35 em ASCII), o que garante a integridade da quantidade de bytes recebidos, mas não garante a integridade dos dados em si. Neste protocolo não existe retransmissão, sequenciamento e não é orientado a conexão (o robô enviará dados mesmo se o canal físico de comunicação estiver defeituoso ou não existente).

Após o processo de sintonização, os autores consideraram os seguintes valores de K_p e K_i como parâmetros satisfatórios para o controle de velocidade.

- $K_p = 0.300$
- $K_i = 0.600$

Com estes parâmetros, a resposta dos motores é ilustrada pela figura 3.34. Durante o processo de sintonização, desejou-se um comportamento uniforme dos motores e resposta rápida. Con-

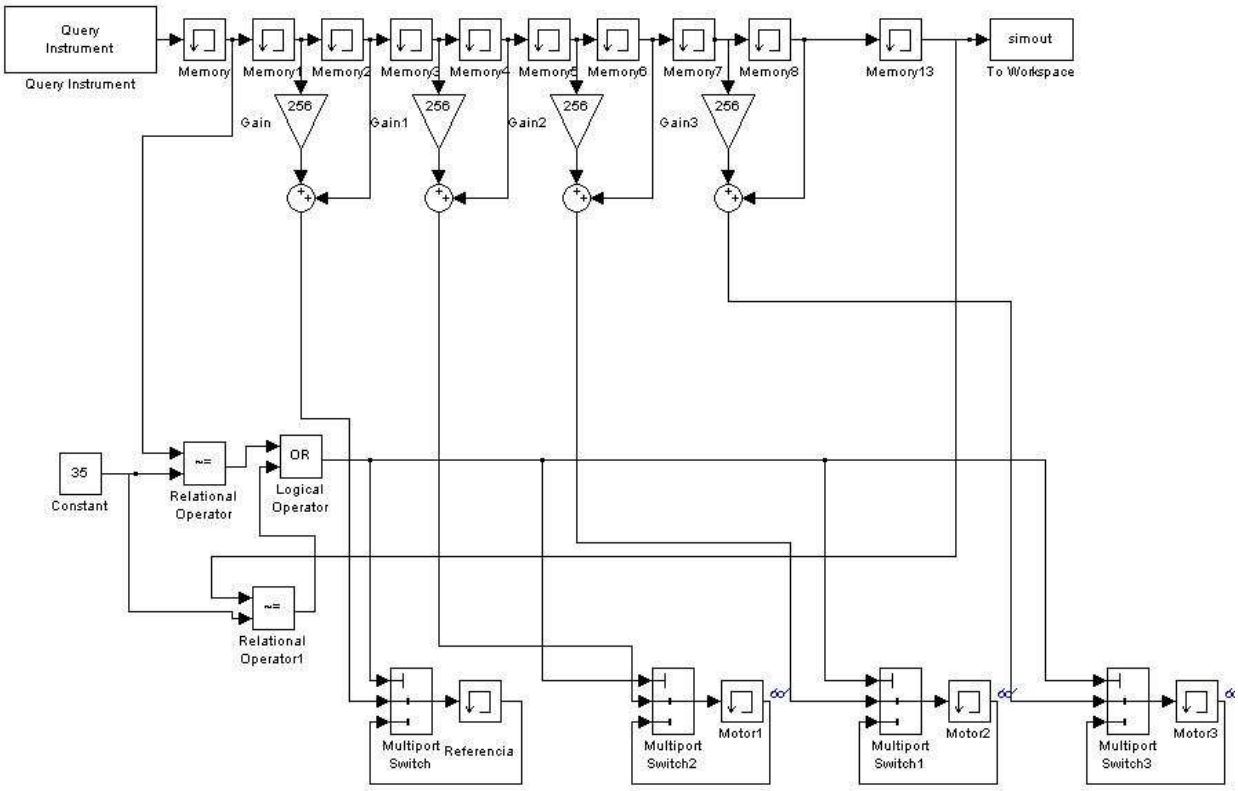

Figura 3.33: Modelo simulink para aquisição de dados dos motores.

forme ilustrado pela figura, os parâmetros acima configuram o controlador de forma que o mesmo respeite os requisitos de funcionamento definidos. Estes valores são carregados no controlador automaticamente ao se ligar as plataformas.

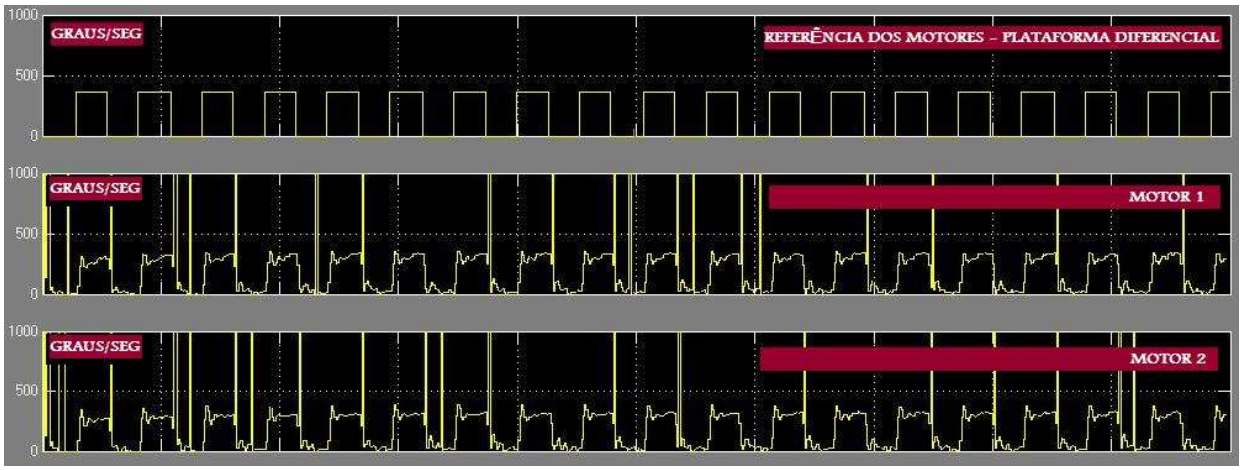

Figura 3.34: Resposta no tempo dos motores da plataforma diferencial.

3.5.2 Obtenção de dados do modo de operação *sintonização*

Quando se entra neste modo, as plataformas irão executar as ações ditadas na seção 3.2.2. Portanto, a plataforma enviará dados periodicamente (10ms) ao MATLAB informando as medições

brutas de seus sensores (inteiro *int* 16 bits) em tempo real. O modelo que permite ao simulink receber esses dados é ilustrado na figura 3.35. Neste modelo também está ilustrado o protocolo de comunicação de dados utilizado. Por ser um modo de depuração, trata-se de um protocolo muito pouco robusto. Este possui apenas um caracter iniciador e terminador (no caso '#', ou 35 em ASCII), o que garante a integridade da quantidade de bytes recebidos, mas não garante a integridade dos dados em si. Neste protocolo não existe retransmissão, sequenciamento e não é orientado a conexão (o robô enviará dados mesmo se o canal físico de comunicação estiver defeituoso ou não existente).

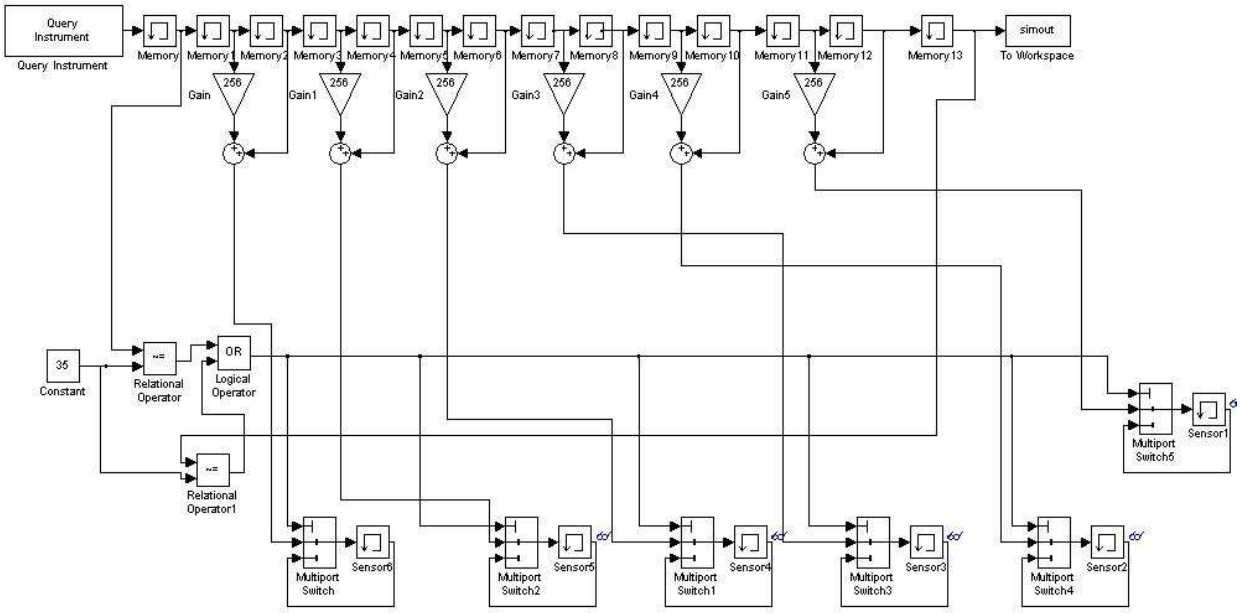

Figura 3.35: Modelo simulink para aquisição de dados do Infravermelho.

Através deste modelo, as respostas dos sensores são visualizadas em tempo real, e assim, podem ser obtidas para se obter uma curva de calibração dos sensores.

A curva de calibração pode ser obtida pelo processo ilustrado a seguir. Observa-se que a curva característica dos sensores infravermelho se assemelha a uma função quociente de dois polinômios de primeiro grau. Portanto, deseja-se aproximar a curva de calibração para uma função do tipo 3.1.

$$F(x) = \frac{mx + b}{x + k} \tag{3.1}$$

O método utilizado para proceder com tal aproximação é o método de *Trust-Region*, implementado no *toolbox* de *Curve Fitting* do software MATLAB. Para uma primeira calibração dos sensores, os autores coletaram 15 conjuntos de dados, cada qual associado à uma distância diferente de um obstáculo plano vetical e branco em um ambiente mal iluminado. Esse conjunto de dados foi obtido pelo modo de operação de calibração e é resumido na tabela 3.7, onde σ^2 é a variância do conjunto de valores medidos e $\overline{x}$ é a média aritmética dos mesmos. Lembra-se que os valores obtidos de x não possuem unidade, conforme discutido na seção 3.2.2.

Observa-se que os valores médios $\overline{x}$ retornados pelo conversor A/D do microcontrolador correspondem ao comportamento esperado conforme folha de dados do fabricante. Porém, existem alguns conjuntos de dados com valores de variância σ^2 absurdos, i.e., extremamente grandes. Um

Tabela 3.7: Dados coletados pelo modo de operação de calibração.

Distância Física F(x) (mm)	$\overline{x}$ **(valor retornado pelo A/D)**	σ^2 (mm^2)	Número de medições
738	86,8737	267,8959	50000
688	91,3911	39,3189	50000
638	95,1269	42,1202	50000
588	99,8972	37,7134	50000
538	105,9082	44,6205	50000
488	112,9588	369,7333	50000
438	122,8494	1902,8	50000
388	132,8226	43,85	50000
338	157,4936	99808	50000
288	178,5673	215140	50000
238	200,3971	134040	50000
188	234,4876	39,0829	50000
138	307,8276	386850	50000
88	415,5717	39,1631	50000
38	478,7021	39,0332	50000

sensor com tais valores possuiria quase nenhuma aplicação prática. Não é o caso do sensor infravermelho deste projeto. Existe uma explicação para o grande valor de variância encontrado e essa é deduzida a partir da figura 3.36. Na figura, observa-se que existem alguns picos no conjunto de valores para medição da distância de 488mm. Esses picos correspondem à valores corrompidos pelo meio de comunicação RS-232, por conta do protocolo utilizado em ambiente MATLAB não ser confiável. Quando se troca aleatoriamente um bit de um valor, esse pode modificar sua magnitude completamente, o que pode alterar o valor da variância do conjunto. Tais corrupções não ocorrerão com o robô em modo de cruzeiro pois o mesmo possui nesse modo um protocolo de comunicação mais robusto à falhas.

Figura 3.36: Conjuntos de medições do microcontrolador para distâncias de 688mm e 488mm.

Em posse desses dados, processa-os através da ferramenta do toolbox de *fitting* e obtêm-se os parâmetros para a função dada em 3.1:

- m = -21,42

- b = 4327
- k = -30,66

Com esses parâmetros, obtêm-se finalmente a curva de calibração ilustrada pela figura 3.37, a qual pode prontamente ser inserida no PLAYER através do driver desenvolvido neste trabalho.

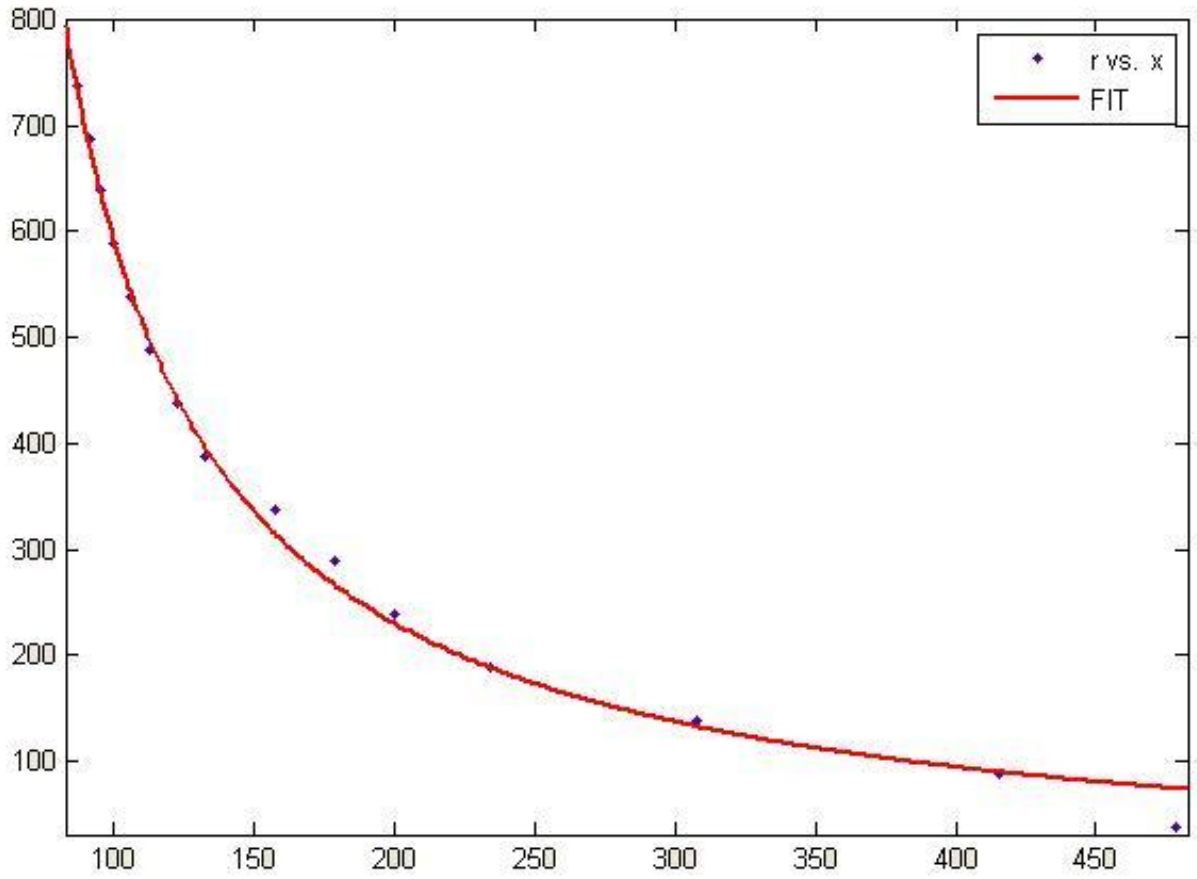

Figura 3.37: Curva de calibração para os sensores obtida experimentalmente.

Os parâmetros da curva (m, b e k) são então configurados pelo arquivo de configuração do driver do PLAYER.

Capítulo 4

Conclusões

Neste capítulo serão encontradas as idéias dos autores quanto aos resultados obtidos por este trabalho de graduação.

4.1 Conclusões

Os autores consideraram este trabalho bastante construtivo para a carreira dos mesmos. Apesar de ter sido um projeto de cunho altamente técnico, vários aspectos teóricos assimilados durante o curso de Engenharia de Controle e Automação puderam ter sido postos à prova. Além disso, grande satisfação temos de ter conseguido desenvolver as plataformas para um estado no qual as duas já podem prontamente ser utilizadas para estudos em robótica de cooperação. Através do grande número de recursos envolvidos nas plataformas, pode-se realizar vários estudos que vão além da robótica de cooperação, tais como visão computacional aplicada à robótica móvel e fusão sensorial.

Atualmente, as plataformas podem ser comandadas através de um cliente posicionado fisicamente em qualquer lugar da mesma rede TCP/IP em que as plataformas se encontram. O cliente tem acesso aos atuadores dos robôs e aos diversos sensores instalados. Foi com sucesso implementado um controle de velocidade das rodas. Inclusive, implementou-se todo um ambiente amigável para futuras sintonizações dos controladores, caso necessário. Da mesma forma, pode-se calibrar *on-the-fly* os sensores de infravermelho com grande facilidade e não existe necessidade de se recompilar nenhum arquivo aqui desenvolvido para tal.

Apesar de atualmente estarem funcionalmente preparadas para serem utilizadas, a eficiência das mesmas pode ser aumentada no futuro através de algum desenvolvimento nas mesmas. Por exemplo, o circuito de alimentação do robô se utiliza de um regulador de tensão de 12V para 5V arquitetura 7085. Essa arquitetura faz com que grande quantia de energia seja dissipada como calor no circuito. Esse fato leva à redução da autonomia da bateria no robô de um fator maior que 2. Os autores indicam um estudo para um circuito de alimentação chaveado para o robô, no qual a energia seria utilizada com maior eficiência.

Outro trabalho futuro pode ser a tentativa de implementação da interface de câmera no PLAYER. Foi desenvolvido um aplicativo para troca de imagens na rede, externo ao PLAYER. Talvez exista uma maneira de aumentar a eficiência da comunicação através do próprio PLAYER.

Por exemplo, pode-se no futuro ver a viabilidade de se estruturar um socket no PLAYER baseado em UDP que, apesar de ser menos confiável, é muito mais rápido.

Os estudos no qual estas plataformas irão participar não estão definidos. Mas os autores imaginam um grande leque de possibilidades para experimentos em ambientes *in-doors*. A ampla quantidade de recursos tornam essas plataformas atrantes para um amplo leque de soluções em robótica de cooperação.

REFERÊNCIAS BIBLIOGRÁFICAS

ANEXOS

I. DIAGRAMAS ESQUEMÁTICOS

Este anexo apresenta os digramas esquemáticos dos circuitos construídos para cada um dos robôs. São eles: circuito de acionamento, de alimentação, dos encoders, do girômetro, dos sensores infravermelho, de comunicação e do ARM, sendo que o último encontra-se disponível em uma headerboard do fabricante Sparkfun.

A placa de potência é responsável pelo acionamento dos motores através da ponte H. A alimentação é responsável pelo fornecimento de energia a todos os elementos da composição eletrônica do robô. O circuito dos encorders é responsável por transformar os pulsos de saída dos encoders em dados de direção e velocidade de rotação. O circuito do girômetro mede a velocidade de rotação do plano da plataforma em torno de um eixo, e fornece interface de disponibilização destes dados ao microcontrolador. O circuito dos sensores infravermelhos fornecem alimentação TTL a cada sensor, chaveamento e disponibilização dos dados de saída de cada sensor, além de um sistema liga/desliga. O circuito de comunicação é baseado no padrão RS-232 para troca de dados via serial entre o EEEPC(responsável por tarefas de alto nível) e o microcontrolador(responsável por tarefas de baixo nível).

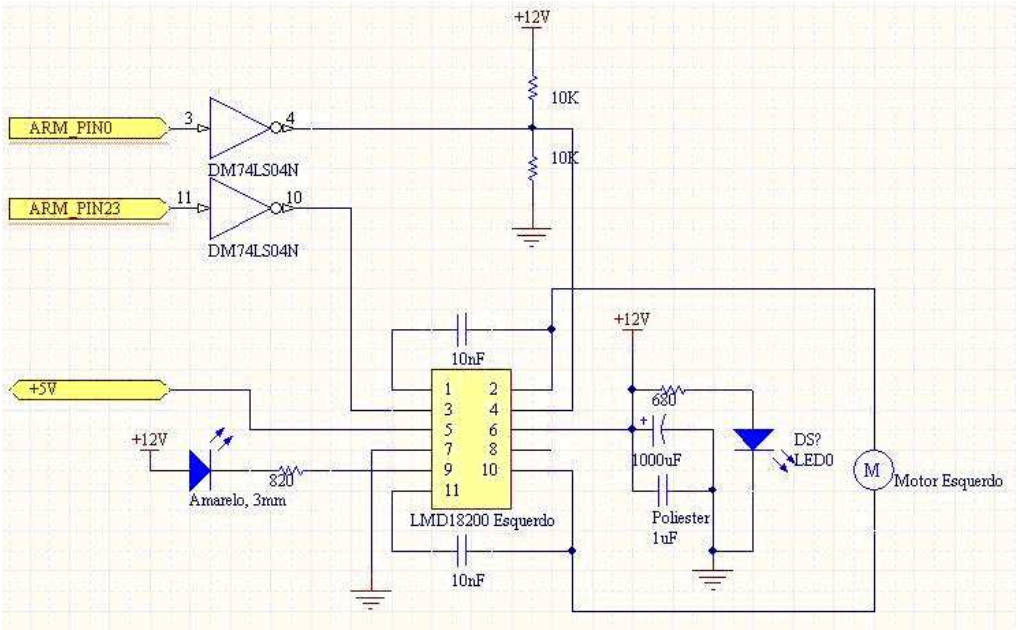

Figura I.1: Circuito associado a ponte-H LMD18200.

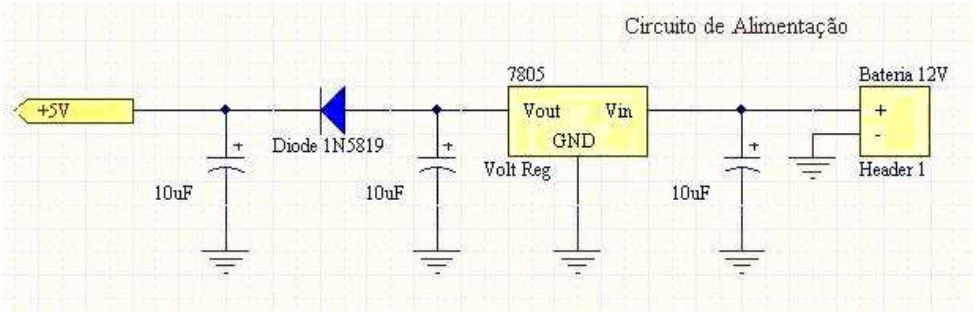

Figura I.2: Circuito de alimentação.

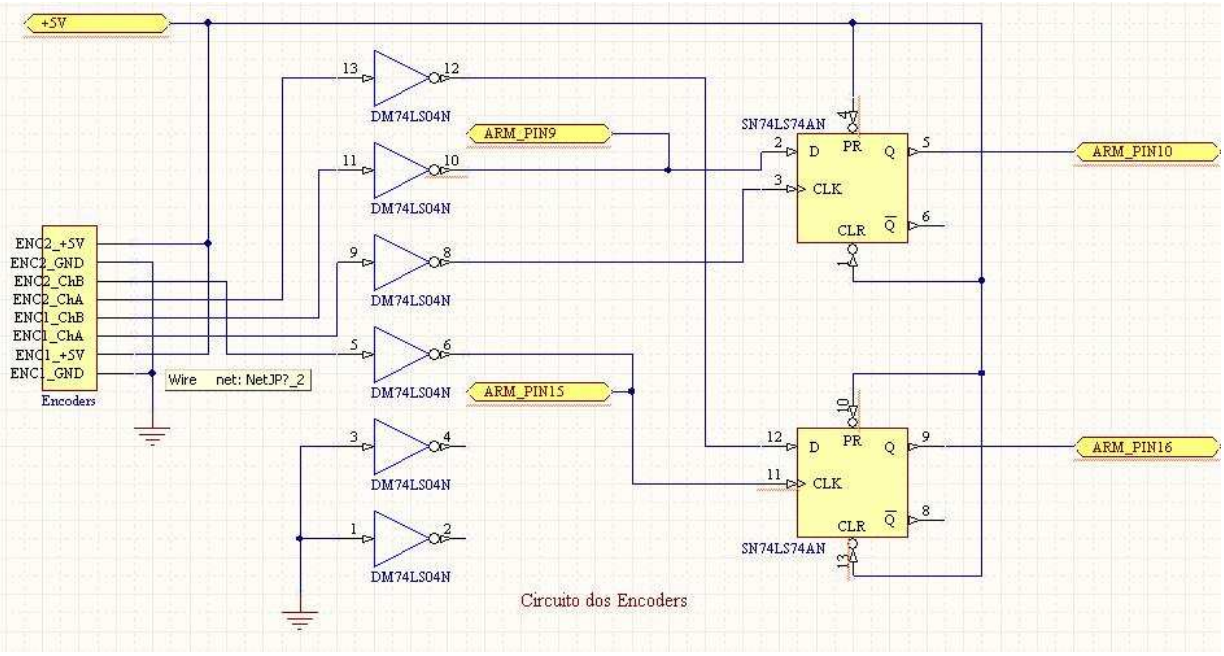

Figura I.3: Circuito dos sensores ópticos para medição de velocidade dos motores.

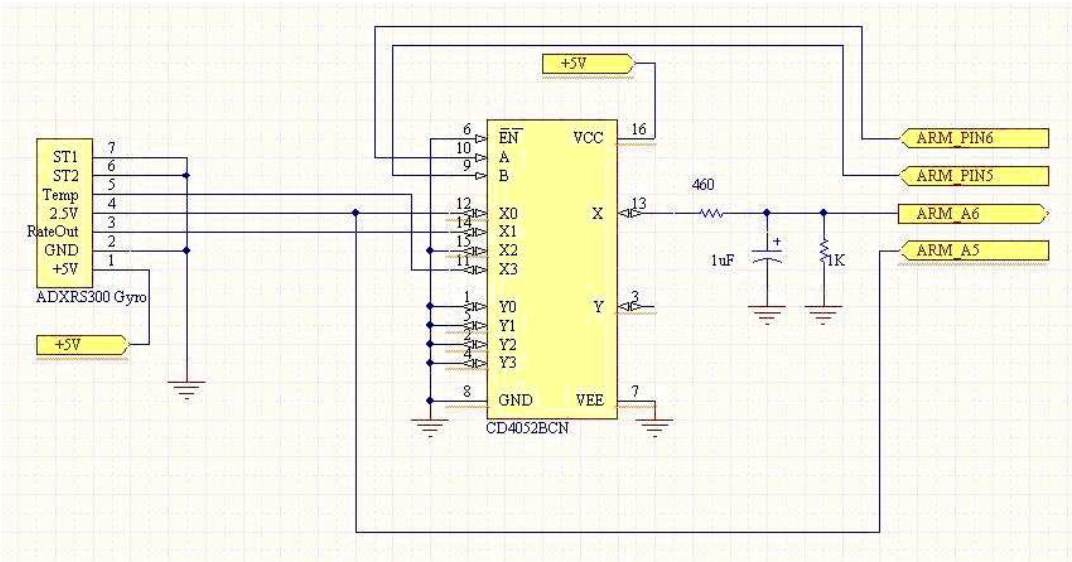

Figura I.4: Circuito relacionado ao girômetro.

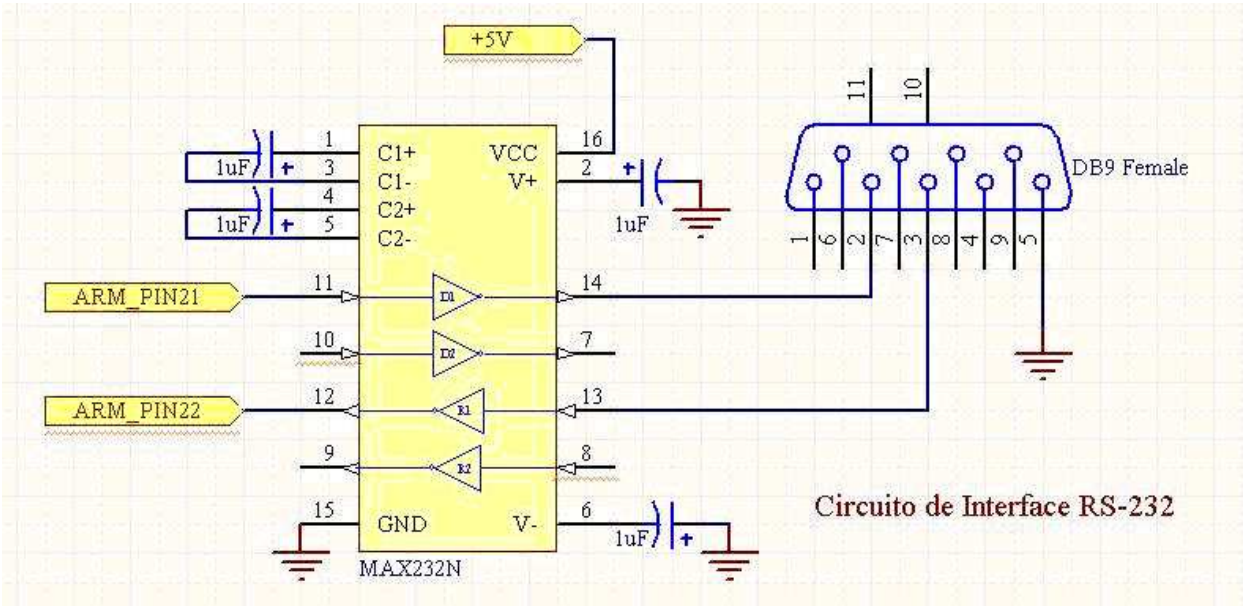

Figura I.5: Circuito para comunicação serial RS-232.

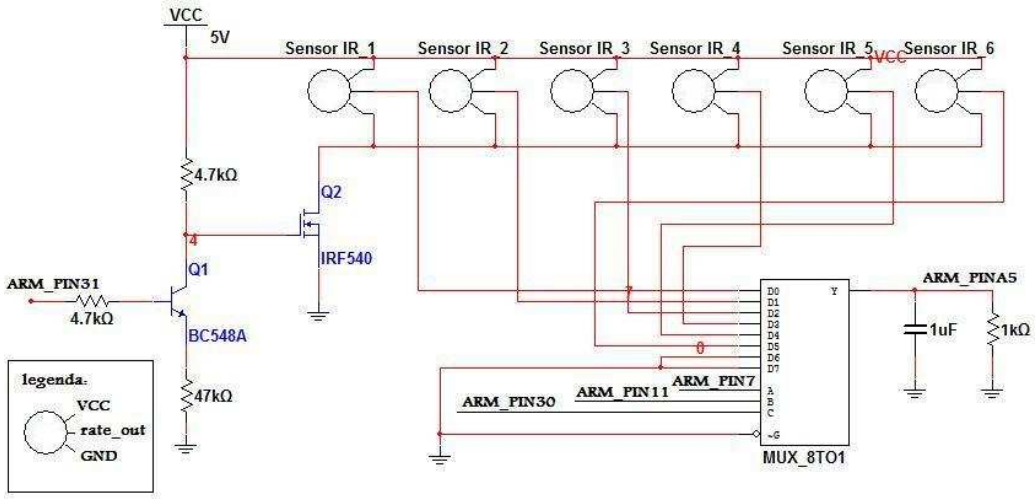

Figura I.6: Circuito elétrico para sensores infravermelho.

II. BIBLIOGRAFIA

[1] LIMA, L.; INUZUKA, H. *Desenvolvimento de um robô móvel omnidirecional: O robô ROHL.* Trabalho de graduação - Universidade de Brasília, 2005.

[2] MURPHY, R. ROBIN. *Introduction to A.I. Robotics.* [S.l.]: MIT, 2000.

[3] FRANKLIN, G. F.; POWELL, J.D.; WORKMAN, M.L. *Digital Control of Dynamic Systems.* [S.l.]: Addison-Wesley, 1997.

[4] ALVES, M. MELO. *Sockets Linux.* [S.l.]: Brasport, 2008.

[5] TANENBAUM, A.; WOODHULL, A. *Sistemas Operacionais, Projeto e Implementação.* [S.l.]: Prentice Hall, 2008.

[6] CAMPION G., D'ANDREA N. BRIGITTE, *Modeling and control of mobile robots with more than two independent steering wheels not satisfying ideal geometrical and kinematic constraints*, In: Proc. IFAC Workshop on Motion Control, Grenoble, France, September, 1998, pp.87-92. (Publié).

[7] OGATA, K. Engenharia de Controle Moderno. [S.l.]: Prentice Hall Brasil, 2003.

[8] SLOSS, A.; SYMES, D.; WRIGHT, C. ARM Systems Developer's Guide: Designing and Optimizing System Sotware. [S.l.]: Elsevier, 2004.

[8] FURBER, S. ARM System-on-Chip Architecture. [S.l.]: Pearson Education, 2000.

[9] SIEGWART, R.; NOURBAKHSH I. Introduction to Autonomous Mobile Robots. [S.l.]: MIT, 2004.

[10] BRÄUNL, T. Embedded Robotics: Mobile Robot Design and Applications with Embedded Systems. [S.l.]: Springer, 1998.

III. CONTEÚDO DO CD EM ANEXO

A seguir segue uma listagem completa dos documentos e arquivos inclusos no CD em anexo.

- Arquivo de slides da defesa: */Apresentacao/defesa-TG*;

- Arquivos do PLAYER, incluindo *plugins* e arquivos configuração: */PLAYER*;

- Arquivos da API para desenvolvimento em EEEPC: */API-EEEPC*;

- Arquivos dos módulos em ARM dos robôs: */ARM*;