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

RODRIGO ROSENFELD ROSAS

UMA ESTRUTURA DE PROGRAMAÇÃO PARA O DESENVOLVIMENTO DE APLICAÇÕES DE ROBÓTICA MÓVEL EM TEMPO-REAL

VITÓRIA
2006

RODRIGO ROSENFELD ROSAS

UMA ESTRUTURA DE PROGRAMAÇÃO PARA O DESENVOLVIMENTO DE APLICAÇÕES DE ROBÓTICA MÓVEL EM TEMPO-REAL

Dissertação apresentada ao Programa de Pós-Graduação em Engenharia Elétrica do Centro Tecnológico da Universidade Federal do Espírito Santo, como requisito parcial para obtenção do Grau de Mestre em Engenharia Elétrica, na área de concentração em Automação.
Orientador: Prof. Dr. Hans-Jörg Schneebeli
Co-orientador: Prof. Dr. Teodiano Freire Bastos Filho

VITÓRIA
2006

Dados Internacionais de Catalogação-na-publicação (CIP)
(Biblioteca Central da Universidade Federal do Espírito Santo, ES, Brasil)

Rosas, Rodrigo Rosenfeld, 1981-

R789e Uma estrutura de programação para o desenvolvimento de aplicações de robótica móvel em tempo-real / Rodrigo Rosenfeld Rosas. – 2006.

109 f. : il.

Orientador: Hans-Jörg Andreas Schneebeli.

Co-Orientador: Teodiano Freire Bastos Filho.

Dissertação (mestrado) - Universidade Federal do Espírito Santo, Centro Tecnológico.

1. Robôs móveis. 2. Framework (Programa de computador). 3. Programação em tempo-real. 4. Linux (Sistema operacional de computador). I. Schneebeli, Hans-Jörg Andreas. II. Bastos Filho, Teodiano Freire . III.Universidade Federal do Espírito Santo. Centro Tecnológico. IV. Título.

CDU: 621.3

RODRIGO ROSENFELD ROSAS

UMA ESTRUTURA DE PROGRAMAÇÃO PARA O DESENVOLVIMENTO DE APLICAÇÕES DE ROBÓTICA MÓVEL EM TEMPO-REAL

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

Aprovada em _________ de 2006.

COMISSÃO EXAMINADORA

____

Prof. Dr. Hans-Jörg Andreas Schneebeli
Universidade Federal do Espírito Santo
Orientador

____

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

____

Prof. Dr. Paulo Faria Santos Amaral
Universidade Federal do Espírito Santo

____

Prof. Dr. Marcelo Ricardo Stemmer
Universidade Federal de Santa Catarina

Agradecimentos

Ao orientador Hans que acreditou desde o início que eu conseguiria desenvolver, além do *framework*, o *driver* da câmera em tempo-real e o robô dentro (quase) do tempo previsto. Também por suas valiosas contribuições de idéias, além da ajuda na construção dos *encoders*. Ao co-orientador Teodiano pela ajuda prestada e pelo material que tornou possível a construção do robô, além do tempo de iniciação científica em que me orientou, o qual me ajudou a aprender sobre robôs móveis, além de aperfeiçoar meus conhecimentos sobre as linguagens de programação C e C++. Todos esses conhecimentos ajudaram muito para a conclusão deste trabalho perto do tempo previsto. Ao professor Paulo Amaral por ter levado o motor das rodas para sua empresa, a fim de adaptar o eixo do motor à roda. Sem essa contribuição, esse trabalho levaria ainda mais tempo para ser concluído.

Aos amigos Jan Kiszka, Philippe Gerum, Hannes Mayer e Gilles Chanterperdrix da lista de discussões do projeto Xenomai, além da ajuda de Paolo Mantegazza da lista do RTAI. Todos, especialmente os dois primeiros, me deram um suporte fantástico, além de conselhos e uma conversa agradável e bem-humorada, e a ajuda deles foi essencial para o desenvolvimento do *framework* e do *driver* da câmera.

Aos grandes amigos e companheiros do LAI (incluindo, claro, aqueles que não são especificamente do LAI mas pertencem ao mesmo grupo, como Alexandre Secchin, do Cisne, que me ajudou a fazer a placa do robô, além de Rodolfo e Serlon do Labtel)! Obrigado Vinícius, Wanderley, Leal, Flávio, Mariana, Christiano, Sandra, André (tanto um como o outro), Raquel, Luciene, Anselmo, Fabrício, Luismar, Marcos, Daniel, Felipe, Alexandre (os dois), Cristina, Lester, Róger e tantos outros que me acompanharam nesse período. O clima de companheirismo, ajuda, descontração e cooperação do pessoal do laboratório são essenciais para um bom desempenho no trabalho. Um abraço especial ao grande amigo Rodrigo Espíndula, que iniciou o mestrado comigo e esteve presente durante toda a graduação, iniciação científica e desenvolveu o Galvanostato-Potenciostato no projeto de graduação, em conjunto comigo. Casou-se no período final da minha dissertação e desejo-lhe muita felicidade.

Dedico um carinho e uma atenção especial à minha namorada Sabrina (e futura esposa, se Deus quiser!) pelo suporte emocional que me deu durante todo o tempo da dissertação. Dissipou boa parte das minhas preocupações e aconselhou-me em decisões que eu considerava complicadas de tomar (não decisões técnicas, claro, pois não é sua área).

Finalmente, à CAPES pela bolsa concedida, que tornou viável a realização deste trabalho.

Sumário

1 Introdução **1**

1.1 Motivação 1

1.2 Descrição da Área 2

1.3 Definição do Problema 6

1.4 Metodologia 7

1.5 Estrutura deste Documento 9

2 Revisão do Estado da Arte: Estruturas para Programação de Robôs Móveis **10**

2.1 Critérios para Avaliação de *Frameworks* Robóticos 10

2.2 Revisão da Literatura sobre *Frameworks* Existentes 13

2.2.1 O Projeto Player/Stage 14

2.2.2 O Projeto OROCOS 14

2.2.3 O Projeto MARIE 15

2.2.4 O Projeto MIRO 16

2.2.5 O Projeto CARMEN 16

2.2.6 DROS 16

2.2.7 MCA2 16

2.2.8 Pyro 17

2.2.9 Scalable Processing Box 17

2.3 Tabela de comparação 17
2.4 Conclusão 19

3 Sistema de Controle de Robôs 20

3.1 Exemplos de Robôs 20
3.2 Estrutura da Programação de Controle 21
3.3 Comunicação entre Tarefas 23
3.3.1 Classes de modelos de linguagens de programação concorrente . . . 23
3.3.2 Arquiteturas tipo Produtor-Consumidor 24
3.4 A Plataforma de Teste 28
3.5 Escolha do Sistema Operacional Base 30
3.5.1 Sistemas operacionais de tempo-real baseados em Linux 31
3.5.2 O problema da inversão de prioridades 33
3.6 Conclusão 34

4 O *Framework* Proposto 35

4.1 A Interface *Robot* 36
4.2 Tipos de Dados e Unidades 38
4.3 Abordagens sobre Programação Baseada em Ações 39
4.3.1 A Abordagem Usada no *Framework* 42
4.3.2 Uso de *Templates* 43
4.3.3 Definindo um Tipo para *ActionManager* 44
4.3.4 Uso de um Espaço de Nomes 44
4.3.5 Uso de uma Classe Abstrata Vazia 44
4.3.6 A Implementação Escolhida 45
4.4 Tarefas, *Xenomai* e C++ - A classe *Task* 45

4.5 O Gerenciador de Ações Padrão 47
4.6 O *Driver* de Captura de Imagens Desenvolvido 48
4.6.1 Abordagem de *drivers* de tempo-real no Xenomai 48
4.6.2 A Implementação do *Driver* 51
4.7 Conclusão 52

5 Experimentos 54
5.1 O Experimento 1 55
5.2 O Experimento 2 60
5.3 Conclusão 62

6 Conclusão e Trabalhos Futuros 63

A Aspectos Construtivos do Robô para Testes Desenvolvido 66

B rt_mmap.h 71

C Implementação da Seção Crítica do *Driver* da Placa de Captura de Imagens 72

D Código Relevante do Experimento 1 75

E Exemplo de documentação gerada pelo Doxygen para a classe Robot 81
E.1 Robot Interface Reference 81
E.1.1 Detailed Description 86
E.1.2 Constructor & Destructor Documentation 86
E.1.3 Member Function Documentation 86
E.1.4 Member Data Documentation 92

Lista de Tabelas

2.1 Comparação entre *frameworks* existentes. 18

5.1 Resultados obtidos para o segundo experimento. 60

Lista de Figuras

1.1 Um robô manipulador tipo SCARA. 2

1.2 Exemplos de robôs móveis. 3

1.3 Funcionamento de um robô móvel. 3

1.4 Camadas de abstração desde o *hardware* à aplicação de controle. 4

3.1 A estrutura de uma interface de requisição de objetos. 26

3.2 Plataforma de teste. 29

3.3 Projeto final do robô. 29

4.1 Herança de classes de robôs. 36

4.2 Relação entre classes para o robô construído (*SimpleRobot*). 38

4.3 Diagrama de funcionamento do *driver*. 52

5.1 Arranjo utilizado para a execução do experimento 1. 56

5.2 A interface gráfica durante a execução do experimento 1. 57

5.3 A trajetória percorrida pelo robô no experimento 1. 58

5.4 Representação do controlador PI utilizado em cada roda. 58

A.1 Representação da montagem do circuito 67

A.2 Disco utilizado no encoder e sua respectiva máscara 68

A.3 Sinais de saída para diferentes níveis de comparação 69

A.4 Foto do robô construído. 70

Resumo

Apesar de existirem vários robôs móveis, as soluções adotadas para sua programação são, normalmente, do tipo *ad hoc*. Alguns projetos foram criados com o objetivo de prover uma interface comum aos vários tipos de robôs móveis, mas nenhum parece ter alcançado os requerimentos de sistemas robóticos reais. A maioria não se importa com restrições de tempo, as quais são intrínsecas às aplicações robóticas. Suporte de tempo-real é um requisito muito importante para comportamento determinístico. Este trabalho compara as estruturas para programação de robôs móveis existentes e propõe uma nova estrutura (*framework*), através de critérios claros e objetivos. Esta estrutura é desenvolvida com base no sistema operacional Linux, utilizando uma extensão de tempo-real, para prover uma interface simples e flexível, adequada à programação de aplicações de robótica móvel, que possuem restrições de tempo severas. A extensão escolhida neste trabalho foi o projeto Xenomai, embora seja possível adaptar a solução para outras extensões como RTAI ou RTLinux. A estrutura proposta permite obter latências da ordem de micro-segundos, além de facilitar bastante o processo de desenvolvimento de aplicações com robôs móveis.

Abstract

While there are several mobile robots around, it is a common practice to program them using *ad hoc* solutions. Some projects were created aiming to provide a common interface to support several kinds of mobile robots but none of them seems to have met real world system requirements. Most of them do not care about time restrictions, which are intrinsic to robotic applications. Real-time support is a very important requirement for deterministic behavior. This project compares current robotic frameworks and proposes a new one, based on clear and objective criteria. This framework is designed in top of the Linux operating system, using a real-time extension, for providing a simple, yet flexible, interface, suitable to mobile robotic applications, which have hard real-time requirements. The choice for a real-time extension was Xenomai, although it would be possible to adapt the proposed framework to other extensions, like RTAI or RTLinux. Such framework provides microseconds latency, in addition of easing a lot the development of mobile robotic applications.

Capítulo 1

Introdução

Nas últimas décadas, vem crescendo o número de robôs móveis fabricados em todo o mundo, seja para fins de pesquisa ou para ajudar a resolver problemas práticos, como desarmamento de campos minados ou exploração de planetas. Surgiram também vários algoritmos para problemas comuns, como desviar de obstáculos, seguir paredes, locomover-se em um corredor e reconhecer objetos e faces. Para que seja possível reutilizar seus códigos em diferentes robôs, sem grande esforço de adaptação, é necessário criar-se uma camada de abstração para o programador.

1.1 Motivação

Diversos projetos, como Player/Stage (GERKEY; VAUGHAN; HOWARD, 2003), OROCOS (BRUYNINCKX, 2001) e MIRO (UTZ et al., 2002) surgiram para tentar prover essa camada mas, embora tenham conseguido grande avanço, todos falham em um ou mais aspectos. Por exemplo, a maioria desses projetos falha em não se atentar para as restrições de tempo existentes em muitas aplicações em robótica, principalmente em sistemas que necessitam de robustez. A solução para tais situações exige a utilização de um sistema operacional de tempo-real, o que não ocorre na maioria desses projetos. Como os projetos atuais não atendem os objetivos de aplicações robóticas genéricas, há muito campo de pesquisa nessa área.

1.2 Descrição da Área

Robôs são estruturas eletro-mecânicas, programados para realizarem certas atividades (ROMANO, 2002). Estas atividades podem ser repetitivas, como ocorre em ambientes de montagem e pintura de peças padronizadas, ou podem ser atividades onde o robô pode percorrer caminhos diferentes para atingir seu objetivo. Robôs industriais, ou manipuladores, costumam realizar atividades repetitivas, e o percurso percorrido pelo robô é sempre muito parecido. A figura 1.1 ilustra um robô manipulador típico em indústrias, do tipo SCARA. A qualidade destes é avaliada de acordo com sua precisão, repetitibilidade, facilidade de programação, manutenção e custo, entre outros critérios.

Figura 1.1: Um robô manipulador tipo SCARA. Figura obtida em http://www.scara-robots.com/scara.htm.

Robôs móveis são robôs autônomos que têm objetivos de interação com ambientes pouco conhecidos. Eles são utilizados para auxiliar o homem em tarefas repetitivas, como limpeza de ambientes genéricos, ou em tarefas que representam perigo ao ser humano, como desarmamento de campos minados e exploração submarina. Robôs móveis podem realizar certos trabalhos com precisão maior que as obtidas por seres humanos. A figura 1.2 ilustra alguns tipos de robôs móveis, como robôs movidos por rodas, patas e esteiras.

Robôs com rodas são fáceis de serem construídos e controlados mas são capazes apenas de locomoverem-se no plano. Robôs com patas são mais difíceis de serem controlados e locomovem-se normalmente a uma velocidade menor que os robôs com rodas. Porém, estes são capazes de subir e descer escadas entre outros movimentos não alcançados por robôs com rodas. Em terrenos irregulares, robôs com esteiras são mais adequados, pois são mais robustos e mais facilmente programados para locomoverem-se nesse tipo de terreno.

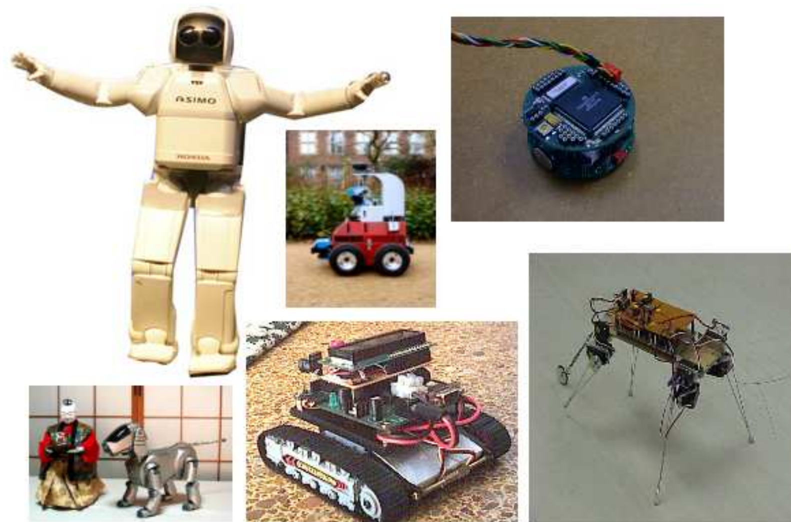


Figura 1.2: Exemplos de robôs móveis.

Para que os robôs sejam autônomos, eles necessitam interagir com o meio de alguma forma. Para isso, são utilizados sensores. Os sinais retornados pelos sensores são processados para obter características do meio, identificando obstáculos e outras marcas que possam ser interpretadas pelo robô. Sensores comuns em robôs móveis envolvem ultra-som, laser e câmeras de vídeo. A partir das informações interpretadas, o robô deve tomar alguma decisão de ação. Tal decisão é codificada em sinais elétricos para os atuadores, que transformam estes sinais em movimento mecânico. A figura 1.3 ilustra o funcionamento de um robô móvel.

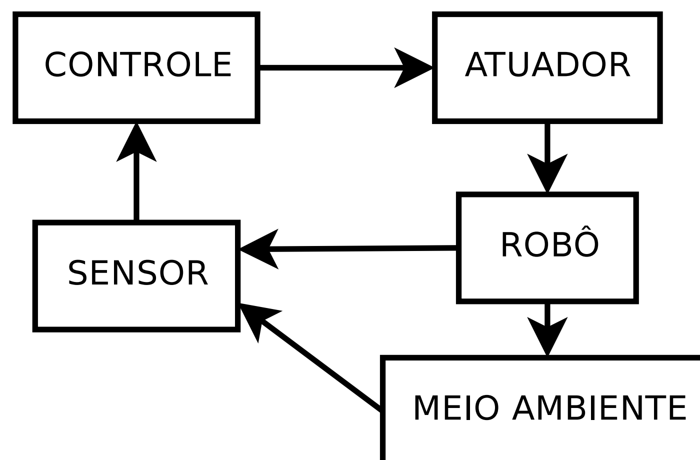


Figura 1.3: Funcionamento de um robô móvel.

Para que os robôs possam interagir com o ambiente, em harmonia, é necessário que as restrições naturais de tempo existentes sejam atendidas. Se um robô a rodas, montado sobre um automóvel adaptado, deve percorrer as ruas de uma cidade, as regras de trânsito deverão ser respeitadas. Quando o semáforo fecha, o robô deve capturar o comando, por

exemplo, utilizando uma câmera de vídeo, processar a imagem e enviar um comando de parada aos atuadores, dentro de um tempo específico. Se este tempo for ultrapassado, o robô ultrapassaria um semáforo vermelho, colocando em risco a vida de pessoas.

Aplicações que tenham restrições temporais são denominadas aplicações de tempo-real. Quando uma especificação de tempo não é atendida, diz-se que o robô violou uma restrição de tempo. Se esta violação resultar em um desastre, diz-se que o sistema é de tempo real crítico (*hard real-time*). Mas se ela simplesmente causar um comportamento incorreto, sem conseqüências graves, diz-se que o sistema é de tempo real brando (*soft real-time*).

Um sistema de programação de robôs móveis deve ser capaz de permitir programar um sistema de controle, em tempo real, de robôs, ou seja, deve permitir garantir que restrições de tempo sejam atendidas. Ele deve ser projetado de modo que seja possível integrar outras ferramentas e bibliotecas de funções já existentes, incentivando o reuso de *software*. É interessante que o sistema seja pequeno e simples o suficiente para ser utilizado em robôs de menor porte, além de ser extensível, para que possa suportar vários tipos de robôs e sensores.

Assim como um sistema operacional é uma abstração do *hardware*, um sistema de programação de robôs, comumente referido como *framework*, deve ser uma abstração entre o sistema de controle e os robôs e sensores. Por exemplo, um sensor de mapeamento retorna informações de distância e possui uma determinada resolução angular. Embora utilizem mecanismos diferentes, sensores laser e de ultra-som são capazes de prover tais informações e a aplicação deveria abstrair-se do tipo de sensor utilizado. Contanto que os sensores satisfaçam as restrições temporais da aplicação, qualquer um poderia ser utilizado, com um mínimo de alteração na aplicação. A figura 1.4 ilustra essa idéia.

Figura 1.4: Camadas de abstração desde o *hardware* à aplicação de controle.

O Sistema Operacional permite utilizar componentes do *hardware* por meio de uma HAL (*Hardware Abstraction Layer* – Camada de Abstração do Hardware) comum. Esta HAL provida pelo Sistema Operacional, por sua vez, comunica-se com o ambiente de controle que se pretende construir. Esse último provê uma API (*Application Programming Interface* – Interface de Programação de Aplicações) para acessar os componentes de um robô em um nível mais alto. Assim, as aplicações de controle podem abstrair-se da complexidade de

funcionamento de um sensor e do tipo de sensor, seja ele laser ou ultra-som, e concentrar-se no algoritmo de controle em um nível mais alto de abstração. Isso agiliza o desenvolvimento de algoritmos de controle, além de permitir um maior reuso de software.

A programação dos robôs móveis, normalmente é realizada com técnicas do tipo *ad hoc*. Ou seja, os programas são construídos para um robô específico para realizar tarefas específicas. A desvantagem dessa abordagem, é que ela torna trabalhosa a adaptação de algoritmos de um robô para outro, mesmo que eles operem de modo bem similar. A falta de uma estrutura padrão para programação de robôs móveis dificulta o desenvolvimento de novos projetos robóticos e manutenção de projetos existentes. Uma estrutura padrão, além de facilitar a programação de sistemas robóticos, permite que implementações de algoritmos sejam facilmente integradas reaproveitadas em diferentes projetos. Além disso, os sistemas tendem a ficar mais estáveis pois, uma vez que há uma grande quantidade de projetos utilizando um determinado módulo, todo erro de programação encontrado e corrigido fica disponível para todos os projetos que utilizem aquele módulo.

Algumas estruturas de programação foram propostas, com o objetivo de suprir essa necessidade, mas nenhuma teve uma aceitação generalizada a ponto de tornar-se um padrão. Um projeto que se destaca é o Player/Stage, que suporta alguns robôs comerciais, amplamente utilizados em ambientes educacionais para ensino de robótica, como a série de robôs *Pioneer* da *ActivMedia*. Eis um exemplo de programação, utilizando a linguagem C++:

```
#include <libplayerc++/playerc++.h>
using namespace PlayerCc;
int main() {
  PlayerClient    robot("localhost");
  SonarProxy      sp(&robot,0); // sensor de ultra-som
  Position2dProxy pp(&robot,0); // sistema de posicionamento
  for(;;) {
    double turnrate, speed;
    robot.Read(); // lê do sensor de ultra-som
    // algorimo simples para evitar colisões
    if((sp[0] + sp[1]) < (sp[6] + sp[7]))
      turnrate = dtor(-20); //girar 20 graus por segundo para esquerda
    else turnrate = dtor(20); // caso contrário, gira para a direita
// Pára o motor se há algum objeto muito próximo à frente do robô.
    if(sp[3] < 0.500) speed = 0;
    else speed = 0.100;
    pp.SetSpeed(speed, turnrate);
  }
}
```

O exemplo mostra como a programação de um robô pode ser bastante simplificada, através de uma estrutura de programação adequada. Infelizmente, o projeto não é capaz de garantir que restrições temporais sejam atendidas e não pode ser utilizado em sistemas reais, onde onde o robô não possa falhar em atender restrições de tempo. No capítulo 2, são abordados outros *frameworks*, assim como seus prós e contras, além de uma avaliação final.

1.3 Definição do Problema

Pretende-se desenvolver uma estrutura de programação (*framework*) para robôs móveis que apresente as seguintes características:

1. Desempenho em tempo real;
2. Eficiência de execução;
3. Facilidade de uso;
4. Qualidade de documentação;
5. Facilidade de manutenção;
6. Reuso de código;
7. Suporte a múltiplas arquiteturas de processadores.

Adicionalmente, além destes requisitos técnicos, o código fonte deverá ser aberto e gratuito para garantir a continuação deste projeto e para permitir que análises de tempo de execução crítico para cada tarefa sejam melhor analisadas por projetistas de sistemas robóticos.

A estrutura de programação a ser desenvolvida também deve permitir que outros critérios, citados na seção 2.1, sejam atendidos futuramente, quando não estiverem em conflito com os critérios considerados mais importantes. Por exemplo, o *framework* deve ser programado de modo que seja possível incluir um simulador de robôs futuramente. Não há, contudo, uma preocupação em suportar robôs manipuladores, uma vez que os modelos de programação destes diferem bastante daqueles utilizados em robótica móvel.

É muito importante que esta estrutura seja fácil de ser programada e que sua documentação seja de fácil leitura por parte dos desenvolvedores. Desenvolvedores de aplicações robóticas freqüentemente não são exímios programadores, embora tenham boa lógica sobre

estruturas de controle de robôs. Portanto, é preciso que o esforço de programação dos desenvolvedores esteja concentrado sobre as estruturas de controle, em vez de confrontá-los com detalhes sobre linguagens de programação. As aplicações finais devem ser fáceis de serem compreendidas por um desenvolvedor iniciante, o que também facilita o reuso de código que, por sua vez, é muito importante em aplicações robóticas. Robôs realizam diversas tarefas, usualmente independentes, mas sincronizadas, para atingirem seus objetivos. Por isso, é importante que tarefas implementadas por um desenvolvedor sejam aproveitadas por outro.

O *framework* proposto também deverá ser capaz de adquirir dados externos em tempo real. Para demonstrar essa capacidade, pretende-se construir um *driver* de captura de imagens com restrições temporais para a placa *framegrabber* PCI DT3153, da empresa *Data Translation*, uma vez que sua especificação foi disponibilizada aos autores deste trabalho.

Alguns tipos de robôs, como robôs celulares, utilizados em aplicações envolvendo cooperação entre robôs, são bem simples. Estes robôs também devem poder ser programados. Pretende-se, então, construir um robô de modo mais simples possível, para demonstrar que é possível programá-lo através do *framework* a ser desenvolvido. Robôs deste tipo possuem as restrições temporais mais críticas. Desta forma, basta mostrar que o *framework* funciona com esse tipo de robô para verificar que ele pode ser utilizado para programar praticamente qualquer outro tipo de robô. O robô a ser construído deve permitir que os sinais dos sensores, bem como os enviados para os atuadores, sejam processados pelo mesmo processador que realiza as tarefas de alto nível. Assim, mostra-se que um único processador é capaz de gerenciar todo um sistema robótico, não limitando o uso do *framework* em relação ao número mínimo de processadores a ser utilizado.

Somente serão implementadas as classes relacionadas a este robô construído, mas o projeto será estruturado de modo a permitir que outras classes de robôs, bem como outros tipos de sensores, sejam adicionados, facilmente, à estrutura de programação proposta, sem a necessidade de remendos (*patches*) de código.

1.4 Metodologia

A primeira parte deste projeto é realizar uma pesquisa sobre as estruturas de programação para robôs já existentes, bem como buscar critérios claros e objetivos para avaliação e projeto de *frameworks* robóticos, cujo resultado está disponível no capítulo 2.

Para validar a estrutura de programação proposta neste trabalho, é necessária a existência de uma plataforma robótica onde serão realizados testes de validação. Para isso, é interessante construir um robô bem simples, com rodas e controlado diretamente por uma porta de

comunicação de um computador. Um único processador deve ser capaz de realizar tanto tarefas de alto nível como de baixo nível, as quais são responsáveis por controlar a velocidade das rodas, através de um sinal elétrico tipo PWM, além de ler dados dos sensores utilizados nos *encoders*. Deste modo, é possível mostrar que se o *framework* permite a programação de um robô sem qualquer poder de processamento a bordo, certamente também permite controlar robôs mais sofisticados. Esta etapa é importante para se conhecer as restrições de tempo tipicamente existentes em robôs móveis.

Por simplicidade, todos os testes são realizados em um PC normal, externo ao robô. Como o código deve ser portável, é perfeitamente possível adaptar o robô, em trabalhos futuros, para utilizar uma placa simples, para realizar o processamento à bordo do robô. Tais placas são freqüentemente referidas como SBC (*Single Board Computer*). Placas que poderiam ser utilizadas no robô, sem qualquer alteração de código, incluem as placas PCA-6188 e PCA-6189 do fabricante *Advantech Corporation*, baseadas em arquiteturas x86, e com suporte ao barramento PCI da placa de captura. Através de algumas poucas adaptações no *driver* de captura e no modo de acesso à porta paralela, é possível portar os experimentos a serem realizados, a fim de validar a estrutura, para outras placas baseadas em arquiteturas ARM, por exemplo. Mais informações sobre a montagem da estrutura estão detalhadas no capítulo 3.4.

Outra etapa, necessária para mostrar que aplicações complexas poderão ser construídas com base no *framework* proposto é a elaboração de um *driver* de captura de vídeo, utilizando o *framegrabber* DT3153. Inicialmente, propõe-se implementar a especificação *Video for Linux 2* (V4L2), para entender melhor o funcionamento da placa e concentrar-se no estudo de desenvolvimento de *drivers* para o Linux, aproveitando programas já existentes para exibição das imagens capturadas, como o XawTv, por exemplo.

A etapa seguinte é realizar um estudo sobre os sistemas operacionais de tempo-real (RTOS) de código aberto, baseados em Linux. Em seguida, deve-se escolher o RTOS base sobre o qual será projetado o *framework*. Em seguida, o *driver* da placa de captura é adaptado para este RTOS, adicionando-lhe garantias de tempo para a captura de imagens.

Finalmente, segue-se um estudo detalhado, com o fim de propor um novo modelo de *framework* que atenda os objetivos deste trabalho. Após a conclusão do *framework*, a última etapa é testá-lo, interagindo com o robô de teste e o *driver* de captura de imagens desenvolvido neste trabalho, através de alguns experimentos.

Os experimentos devem demonstrar que é possível atender às restrições de tempo necessárias para o funcionamento correto do robô e também para mostrar a facilidade de programação de sistemas robóticos, através da utilização da estrutura de programação proposta

neste trabalho. Adicionalmente, ao menos um experimento deve mostrar a utilização da câmera de vídeo para capturar e processar imagens, com garantias de tempo. Os experimentos devem mostrar que é possível desenvolver tanto aplicações robóticas simples, como aplicações complexas, obtendo latências da ordem de micro-segundos durante a execução das tarefas de controle. Uma vez que seja possível garantir o tempo máximo de execução de cada tarefa, todos os experimentos devem atingir seus objetivos, em todos os testes realizados, o que não costuma acontecer em sistemas de uso geral, os quais, normalmente não provêem garantias de tempo.

1.5 Estrutura deste Documento

Este capítulo introduziu o leitor à área de Robótica. O capítulo 2 apresenta e classifica os diversos projetos existentes para programação de robôs móveis. O capítulo 3 prepara o leitor para o entendimento do *framework*, a ser desenvolvido no capítulo 4. Ele informa sobre as estruturas de controle de robôs usuais, comenta sobre a necessidade de comunicação entre tarefas, apresenta as possíveis abordagens e as classifica, de acordo com sua eficiência. Os detalhes sobre a construção do robô, do *driver* da câmera, bem como a escolha de um sistema operacional de tempo-real base, sobre o qual será construído o *framework*, são objetos de estudo do capítulo 3.

O capítulo 4 apresenta, finalmente, a proposta de um novo *framework*, considerando os critérios adotados para este trabalho. Tal *framework* é validado no capítulo 5, onde são realizados experimentos com o robô construído, utilizando *encoders* baseados em reflexão de luz de espectro infra-vermelho, para realização do sistema de hodometria, bem como uma câmera analógica, acoplada a uma placa de captura (*framegrabber*), utilizando o *driver* desenvolvido.

Finalmente, o capítulo 6 conclui o trabalho, apresentando suas vantagens e sugerindo trabalhos futuros.

Capítulo 2

Revisão do Estado da Arte: Estruturas para Programação de Robôs Móveis

Este capítulo comenta sobre critérios, normalmente adotados, para avaliação de estruturas de programação para robôs para, então, avaliar as estruturas existentes. Os critérios são listados em 2.1 e o estado da arte é revisado na seção 2.2.

2.1 Critérios para Avaliação de *Frameworks* Robóticos

A fim de avaliar e comparar os *frameworks* já existentes ou propostos, é necessário utilizar alguns critérios claros e objetivos. Kuo utilizou alguns interessantes em seu artigo (KUO; MACDONALD, 2004): desempenho em tempo real, simplicidade, escalabilidade, fácil integração e independência de plataforma e linguagem de programação. Tolerância a falhas também foi discutido em (MELCHIOR; SMART, 2004). Outros critérios, embora abordados na literatura pesquisada, não foram utilizados como forma de classificação. São eles: eficiência, facilidade de manutenção, presença de simulador, disponibilidade de código fonte, independência de *hardware*, qualidade de documentação, ferramentas de depuração e processamento distribuído. Os critérios utilizados neste trabalho são os critérios utilizados por Kuo, mais os critérios citados. Alguns critérios são pouco compatíveis entre si e, por isso, a solução adotada deve ponderar as características mais importantes, de modo a considerar os compromissos de cada escolha. Eis a relação dos critérios propostos:

- **Independência de plataforma**: para que os usuários possam escolher o sistema operacional de tempo real de sua preferência.

- **Escalabilidade aprimorada**: as soluções devem ser modularizadas para permitir a construção de sistemas robóticos mais complexos e facilitar a identificação de falhas e permitir construções de unidades de testes dos módulos componentes.
- **Simplificação do processo de desenvolvimento**: o usuário deveria preocupar-se somente com o algoritmo e não com a complexidade envolvida na composição de um robô.
- **Facilidade de manutenção**: para simplificar a identificação e correção de erros, buscando uma garantia de funcionamento maior, bem como facilitar a adição de novos componentes e robôs.
- **Desempenho em tempo real**: as tarefas críticas de um robô devem ter restrições de tempo para garantir o funcionamento correto do sistema e assegurar sua robustez.
- **Integração com infra-estrutura existente**: para evitar "reinventar a roda", utilizando as tecnologias já existentes e bem testadas.
- **Promoção de reuso de *software***: as interfaces com o usuário devem ser desenvolvidas, de modo a permitir que componentes possam ser reutilizados em outras aplicações, tornando o processo de desenvolvimento mais rápido e menos propenso a falhas, uma vez que os componentes tendem a amadurecer.
- **Independência de linguagem de programação**: para que o usuário possa escolher a linguagem de programação que considerar mais adequada. Para efeitos de prototipagem, é possível utilizar uma linguagem interpretada pela facilidade de depuração e a não necessidade de recompilação, além de alta portabilidade. Tal técnica é abordada em ferramentas tipo *Matlab* ou *Scilab*.
- **Eficiência**: sistemas robóticos executam muitas vezes em hardware com pouco poder de processamento, principalmente em robôs celulares empregados em trabalhos de cooperação entre robôs. Eficiência é, portanto, uma questão crucial.
- **Disponibilidade de código fonte**: para que seja possível garantir o funcionamento correto e identificar falhas e correções.
- **Independência de *hardware***: para que se possa utilizar diferentes tipos de robôs e sensores.
- **Qualidade de documentação**: para facilitar o processo de desenvolvimento.
- **Presença de simulador**: para facilitar o processo de prototipagem e evitar danos aos robôs, além de agilizar o processo de desenvolvimento e permitir a utilização de um robô compartilhado entre diferentes pesquisadores.

- **Existência de ferramentas de depuração**: fundamental, durante o processo de desenvolvimento.

- **Processamento distribuído**: interessante, principalmente em trabalhos de robôs cooperativos ou de computação intensa. Por processamento distribuído, este trabalho está considerando os fracamente acoplados, ou seja, aqueles onde os processadores estão em computadores diferentes e a comunicação entre eles é realizada através de uma rede de computadores. Processamento fortemente acoplado está relacionado ao sistema operacional e não ao projeto de um *framework*.

- **Tolerância a falhas**: para sistemas que não podem falhar devido à inexistência de operadores próximos ao robô, como é o caso de exploração espacial.

Tais critérios serão utilizados como termos de comparação entre os projetos desenvolvidos ou propostos, e nortearão o presente trabalho na proposta de um sistema de programação de robôs móveis.

Uma característica crítica e que influencia a maior parte destes critérios é o modelo de comunicação entre processos, mais conhecido como IPC (*Inter Process Communication*), o qual é discutido com detalhes no capítulo 3.3. O modelo de comunicação interfere diretamente no desempenho de um sistema. Trocas de mensagens por meio de um espaço físico de memória compartilhado entre tarefas é a forma mais eficiente de comunicação, já que o processador tem acesso direto à informação.

Em processamento distribuído, contudo, esse tipo de comunicação não é possível. A tecnologia CORBA (OPEN MANAGEMENT GROUP, 1991) provê uma série de facilidades para realizar processamento distribuído, porém é um sistema complexo e, devido à sua generalidade, inadequado para muitas aplicações embutidas, principalmente com requerimentos de tempo-real.

Outros métodos utilizados para processamento remoto incluem as *chamadas a procedimentos remotos* que, assim como a tecnologia CORBA, provê transparência de localização. Isso significa que a rotina que executa esses procedimentos não precisa conhecer a localização do computador que os contém. Em sistemas embutidos, porém, como é o caso de Robótica, é importante para muitas aplicações ter total controle sobre as estruturas envolvidas em um sistema, principalmente em sistemas de tempo-real. Isso significa que transparência de localização, que pode ser uma característica excelente em sistemas de banco de dados, não é interessante em sistemas embutidos, pois a mudança de um serviço de um computador para outro pode afetar o desempenho de uma aplicação completamente, a ponto de ela funcionar incorretamente.

Por isso, a escolha dos modelos IPC deve ser realizada cuidadosamente em cada caso onde é necessário realizar comunicação entre diferentes tarefas e, sempre que apropriado, deveria ser possível permitir que o programador escolha o modelo que considerar mais adequado.

Como já foi mencionado, a escolha dos tópicos a serem considerados requer alguns compromissos (*trade-off*) de desempenho por flexibilidade. Por exemplo, ao utilizar uma tecnologia tipo CORBA para comunicação entre processos, é necessário abrir mão de desempenho. A integração com softwares existentes pode implicar, às vezes, em abrir mão da funcionalidade de tempo real. Portanto, é importante pesar todos os prós e contras de uma tecnologia, a fim de escolher um conjunto de características satisfatórias para a maioria das aplicações em robótica móvel, e, se possível, permitir ao usuário utilizar uma tecnologia específica para realizar determinada tarefa.

As estruturas de programação, ou *frameworks*, avaliadas nesta seção, são classificadas de acordo com os critérios citados e modelo de comunicação entre processos (IPC). Os compromissos existentes na escolha de cada tecnologia utilizada para a implementação do *framework* também serão mencionados. A terminologia utilizada neste trabalho é bem explicada em (Albus; Quintero; Lumia, 1994).

2.2 Revisão da Literatura sobre *Frameworks* Existentes

Dois projetos desenvolvidos para programação de robôs destacaram-se dos demais. O projeto Player/Stage (GERKEY; VAUGHAN; HOWARD, 2003) teve grande aceitação por desenvolvedores por contar com suporte para vários robôs móveis convencionais, incluindo o *Pioneer* da *ActivMedia*, e possuir um simulador para efetuar testes. O projeto OROCOS (BRUYNINCKX, 2001) é bastante citado e constitui-se de um conjunto de classes para programação de robôs, permitindo programar tarefas em tempo-real utilizando o RTAI (DIAPM, 1998) ou RTLinux (YODAIKEN; BARABANOV, 1996). Este último conta com uma lista bastante abrangente sobre projetos relacionados. Um enfoque especial será dado a esses dois projetos por sua grande popularidade e por utilizarem abordagens bem diferentes. O projeto MARIE (CÔTÉ et al., 2004) também será explicado com mais detalhes por utilizar uma abordagem também bem diferente dos outros projetos, focando na integração das ferramentas existentes.

2.2.1 O Projeto Player/Stage

O projeto Player/Stage (GERKEY; VAUGHAN; HOWARD, 2003) iniciou-se em 2000 como uma evolução dos projetos Ayllu/Arena e posteriormente Golem/Arena. Os projetos iniciais estavam mais amarrados aos robôs *Pioneer* da *ActivMedia*. Desde o início ele já contava com uma de suas melhores qualidades: a capacidade de simular um controle de robôs, utilizando múltiplos robôs.

Player/Stage avançou em relação aos projetos iniciais no sentido de permitir uma separação maior entre o *hardware* e a API para programação dos robôs, permitindo utilizar diferentes tipos de robôs, enquanto inicialmente apenas os robôs da *ActivMedia* eram suportados. Atualmente, o projeto suporta uma série de robôs e sensores, além de uma independência de linguagem de programação, e poder simular o movimento dos robôs e os dados coletados dos sensores a partir da utilização de um mapa (uma imagem binária). A partir do projeto *Gazebo*, também é possível realizar simulações em ambientes tridimensionais.

O projeto falha contudo em não ter sido planejado para utilização em ambientes que requerem uma programação em tempo real. Adaptá-lo para incluir essa funcionalidade suporia muitas modificações no código original, além de que a API deveria ser modificada, de forma a incluir as restrições de tempo. O fato da comunicação utilizar *socket* TCP, o qual não é determinístico, além de impossibilitar a programação em tempo-real, é pouco eficiente quando comparado a outros métodos IPC (ver capítulo 3.3).

2.2.2 O Projeto OROCOS

O projeto OROCOS (BRUYNINCKX, 2001) é um *framework* projetado para programar tarefas de tempo-real. É um projeto altamente estruturado e construído a partir de muitas camadas de classes de objetos, escrito em C++.

Como sua intenção é servir a um propósito genérico para programação de robôs, ele pode ser utilizado para programar tanto robôs industriais quanto robôs móveis, embora ainda não haja um invólucro (*wrapper*) para essa última classe de aplicação.

O projeto utiliza uma abordagem de orientação a objetos para realizar a comunicação entre tarefas e o sistema operacional de tempo real. O RTAI é uma API de tempo real suportada pelo projeto, para se comunicar com um sistema Linux modificado pelo projeto ADEOS – *Adaptive Domain Environment for Operating Systems* (YAGHMOUR, 2001). O método de comunicação utilizado para exercer as funções básicas é suficientemente eficiente para a quase totalidade das aplicações de tempo real.

Uma desvantagem é que possui uma curva de aprendizado elevada, mas isso pode ser contornado através da utilização de classes invólucros (*wrappers*) para facilitar a programação de tarefas mais simples. Contudo, às vezes, é necessário possuir um conhecimento mais aprofundado sobre as operações envolvidas, e, nesse momento, é muito difícil analisar o *framework*, tornando-o inapropriado para um desenvolvedor sem grandes habilidades computacionais, enquanto os mesmos são capazes de entender alternativas mais simples, até mesmo, para programadores experientes.

A utilização de vários níveis de classes de objetos também dificulta a garantia de funcionamento do sistema e o projeto é considerado muitas vezes como uma solução muito complexa para problemas simples. Além disso, devido a sua generalidade, o *footprint* envolvido é maior do que o necessário para aplicações em robótica móvel.

2.2.3 O Projeto MARIE

MARIE (CÔTÉ et al., 2004) se propõe a facilitar a integração entre diferentes projetos para a programação de robôs, como Player/Stage, CARMEN (MONTEMERLO; ROY; THRUN, 2002) e OROCOS, baseando-se em um padrão de projeto (*design pattern*) tipo Mediador, o qual é utilizado nos projetos RobotFlow/FlowDesigner (http://flowdesigner.sourceforge.net).

Esse padrão contém uma unidade intermediária, o Mediador, com uma API bem definida, capaz de se comunicar com cada um dos projetos suportados através dos Adaptadores de Aplicação, que são classes implementadas em C++ capazes de traduzir as informações específicas da aplicação para o Mediador e vice-versa. O projeto utiliza a rede de programação ACE (SCHMIDT, 1993) para realizar a comunicação. ACE suporta vários mecanismos de IPC, além de poder ser utilizado na implementação de uma tecnologia CORBA, o que acontece em TAO (SCHMIDT; DESHPANDE; O'RYAN, 2002).

MARIE possui muitas vantagens como integração com infra-estruturas já existentes, facilidade de manutenção, escalabilidade, simplificação do processo de desenvolvimento, independência de linguagem de programação (desde que os projetos utilizados permitam), disponibilidade de código fonte, possibilidade de uso de simulador através de integração com projetos existentes e possibilidade de processamento distribuído utilizando CORBA ou outra tecnologia. Porém, o projeto não se preocupa com requerimentos essenciais como desempenho em tempo real, eficiência e robustez. Uma vez que é possível integrar diferentes projetos, é complicado garantir o funcionamento correto, já que diferentes processos tentando acessar um mesmo recurso ao mesmo tempo podem provocar um resultado imprevisível.

2.2.4 O Projeto MIRO

O projeto MIRO (*Adaptive Middleware for a Mobile Internet Robot Laboratory*) (UTZ et al., 2002) acertou em desenvolver um *framework* com a preocupação de permitir a programação de tarefas com restrições temporais. Porém, ao utilizar a tecnologia CORBA (ver capítulo 3.3), o projeto faz uso de transparência de localização, o que não é desejado para certas aplicações robóticas, onde é importante conhecer a localização de objetos, uma vez que isso influencia diretamente no desempenho do controle de um sistema.

2.2.5 O Projeto CARMEN

O projeto CARMEN (MONTEMERLO; ROY; THRUN, 2002) – *Carnegie Mellon Robot Navigation Toolkit* – é outro *framework* para desenvolvimento de robôs móveis que se encontra, segundo o próprio *site*, em estado experimental (*alpha*). Esse projeto utiliza um pacote separado para comunicação entre processos denominado IPC, o qual acompanha a distribuição do *framework*. Esse projeto falha em não permitir a programação de tarefas em tempo-real.

2.2.6 DROS

O *framework* DROS (AUSTIN, 2002) – *Dave's Robotic Operating System* – foi desenvolvido por um único autor (David Austin) a partir de sua experiência de seis anos trabalhando em projetos como RobotADT, o sistema *Intelligent Service Robot* (ISR) e seu próprio projeto chamado *Robot Software System* (RSS). Esse projeto, que é uma continuação de seus trabalhos anteriores (ISR e RSS), foi lançado sob a licença GNU e portanto tem seu código aberto. O autor não pretende adicionar suporte a aplicações em tempo real em médio prazo. O projeto está ainda em uma etapa inicial de desenvolvimento e apenas sensores *SICK Laser* e robôs XR4000 são suportados no momento.

2.2.7 MCA2

O *framework* MCA2 (SCHOLL, 1998) – *Modular Controller Architecture Version 2* – utiliza o sistema RTLinux para realizar as atividades de tempo real. O projeto utiliza a linguagem C++ e é estruturado de modo altamente modular com pequenos módulos, de modo a incentivar o reuso de software. Uma desvantagem é o fato de usar o sistema RTLinux, em vez do RTAI ou Xenomai (GERUM, 2004), que além da questão polêmica das patentes sobre

o sistema, também se encontra mais atrasado que o RTAI e Xenomai, ao acompanhar as versões mais novas do *kernel* do Linux. É importante acompanhar as versões mais novas para permitir o uso de novos *hardwares* à medida que surgem suporte para eles no *kernel*. A seção 3.5 provê detalhes adicionais de comparação entre os três projetos e mostra outras desvantagens do RTLinux em relação aos outros.

2.2.8 Pyro

Pyro (BLANK et al., 2003) – *Python Robotics* – é um projeto interessante para pesquisa e prototipagem visto que Python é uma linguagem interpretada e oferece consideráveis vantagens para depuração, o que é muito importante nesse tipo de aplicação. O projeto contém uma série de algoritmos de visão, aprendizagem, evolução, inteligência artificial, entre outros, além de fornecer uma interface gráfica para o usuário. Porém, o projeto não preenche requisitos básicos de tempo real e eficiência.

2.2.9 Scalable Processing Box

O projeto *Scalable Processing Box* (SPB) (HOHMANN; GERECKE; WAGNER, 2003) tem o objetivo de fornecer um sistema de tempo real para programação de robôs com fins de aplicação em ensino de robótica e é um sub-projeto do MoRob (GERECKE; HOHMANN; WAGNER, 2004). O projeto utiliza uma distribuição Linux para sistemas embutidos, a fim de tornar o sistema operacional suficientemente pequeno para se adaptar a robôs menores. O projeto também desenvolve um kit robótico para integrar ao sistema desenvolvido. Existem interfaces simples para o *Matlab* já desenvolvidas, mas ainda não há um simulador para os robôs. O sistema operacional utilizado é o Linux modificado pelo RTAI.

2.3 Tabela de comparação

Outros projetos relacionados podem ser verificados na página do projeto OROCOS, em www.orocos.com/related.php.

A tabela 2.1 resume as vantagens e desvantagem dos projetos citados.

Framework	Desempenho em tempo real	Independência de plataforma	Escalabilidade aprimorada	Simplificação do processo de desenvolvimento	Facilidade de manutenção	Integração com infra-estrutura existente	Promoção de reuso de software	Independência de linguagem de programação	Eficiência	Disponibilidade de código fonte	Presença de simulador	Existência de ferramentas de depuração	Provê facilidades para processamento distribuído	Tolerância a falhas	Suporta múltiplos *hardwares*	Boa qualidade de documentação
Player/Stage			x	x	x	x	x	x		x	x	x	N		x	x
OROCOS	x		x	x*	x*		x		x	x		x	N	x	x	x
MARIE		x	x	x	x	x	x	P		x		x	N		x	x
MIRO	x	C	x	?	?	?	x	x		x		x	x	?	?	
CARMEN		x	x	x	?	x	x	x		x	x	x	N		x	
DROS			?	?	?		x		?	x	x	x	N			
MCA2	x		x	?	?	x	x		?	x		x				
Pyro		x	x	x	x	x	x			x	x	x	N		x	x
SPB	x															

C - Plataformas de tempo real que provejam uma implementação de Real-Time CORBA com TAO+ACE. Atualmente, há apenas um projeto não concluído de portar para o Linux+RTAI: o OCEAN.
? - Critérios não verificados devido à existência de falhas em outros critérios críticos.
* - a curva de aprendizagem é elevada, porém, quando superada, o processo de manutenção e de desenvolvimento são simplificados.
P - algumas partes precisam ser desenvolvidas em C++.
N - O critério está presente, mas não tem funcionalidade de tempo real.

Tabela 2.1: Comparação entre *frameworks* existentes.

2.4 Conclusão

A tabela 2.1 mostra que nenhum dos projetos atinge bem os objetivos considerados mais importantes neste trabalho: facilidade de uso, desempenho em tempo-real, eficiência de execução e flexibilidade. Mais do que isso, para que estes projetos possam atingir bem estes objetivos, o esforço de adaptação seria tão grande, que é mais simples começar um novo, aproveitando trechos de código específicos de arquiteturas de robôs, do que alterar o código e projeto (*design*) de algum dos projetos atuais.

Além disso, muitos dos projetos, devido à escolha da forma como foram projetados, inviabilizam que outros dos critérios mencionados na seção 2.1 possam ser alcançados. Todos esses aspectos motivaram a criação de uma nova abordagem para a construção de uma estrutura de programação de robôs que atendessem melhor os requisitos existentes em aplicações reais, e flexível o suficiente para permitir que outros critérios sejam alcançados futuramente.

Capítulo 3

Sistema de Controle de Robôs

O primeiro passo para a construção de um *framework* é entender o funcionamento dos robôs e as estruturas típicas de programação existentes. Este capítulo faz uma revisão rápida de Robótica, comenta sobre o robô móvel desenvolvido para testar o *framework* final, bem como sobre o *driver* da câmera desenvolvido, considerando restrições de tempo-real, com o propósito de exemplificar e validar o presente projeto.

3.1 Exemplos de Robôs

Existem, basicamente, dois tipos de robôs: robôs manipuladores ou industriais e robôs móveis. Os primeiros são aplicados a processos repetitivos como montagem, pintura, realização de furos e lavagens, entre outras aplicações. São muito utilizados em indústrias automobilísticas, automatizando quase toda a produção final. A outra categoria possui uma complexidade extra à realização de movimentos. Robôs móveis devem tomar decisões intermediárias para atingirem seus objetivos finais. As ações realizadas pelos robôs não são repetitivas, mas dependem da situação atual em que o robô se encontra, porém, o objetivo deve ser alcançado sempre que possível.

Ambos os tipos de robôs possuem seus graus de complexidade mas os modelos de programação utilizados diferem bastante entre os dois tipos, de modo que pouco ou nada se aproveita, em termos de programação, atualmente. Desse modo, é comum existirem *frameworks* diferentes para programação de robôs móveis e manipuladores. Por exemplo, OROCOS, atualmente, permite a programação de robôs manipuladores, enquanto Player/Stage se propõe a atingir robôs móveis somente. O projeto OROCOS pretende ser expandido para lidar com robôs móveis também, mas até o presente momento, tal expansão não foi iniciada.

Vários são os motivos. Muitos robôs industriais possuem menos volume de informação para ser processado em um determinado tempo, quando comparados a robôs móveis que dependem de tratamento de imagens e tomadas de decisões. Dessa forma, um *framework* com bom suporte a processamento distribuído, sob o compromisso de ser menos eficiente, e que suportasse um modelo de programação típico em robôs industriais, atenderia bem a essa classe de robôs. Contudo, seria incapaz de suportar diversas aplicações com robôs móveis. Esses são os motivos mais prováveis pelo qual ainda não há um esforço para se criar uma camada que suporte robôs móveis no projeto OROCOS: pouca eficiência, muita dificuldade de programação e inflexibilidade na programação da estrutura de controle.

Entre os robôs móveis, existem outras classificações de robôs, como robôs móveis sobre rodas – que podem ser subdivididos em controle por tração diferencial, ou por deslocamento de um eixo comum a duas rodas – robôs sobre patas, robôs sobre esteiras, robôs submarinos, helicópteros robóticos, entre outros. Este trabalho concentra-se em robôs que se locomovem em um plano, que é o tipo mais comum, mas o *framework* desenvolvido pode ser expandido para os outros tipos de robôs móveis, uma vez que ele é bastante flexível, como será mostrado no capítulo 4.

3.2 Estrutura da Programação de Controle

O presente trabalho propõe um *framework* específico para aplicações de robótica móvel, sem ter em mente, como projetos futuros, suporte a robôs manipuladores, pois tal suposição envolveria algumas escolhas de projeto que não seriam adequadas à programação de robôs móveis, pois são muitos os compromissos existentes entre os critérios utilizados neste trabalho, como comentados no capítulo 2.1. Alguns critérios são mais adequados à programação de robôs manipuladores e outros são mais adequados a robôs móveis.

Um robô móvel freqüentemente necessita realizar diversas tarefas ao mesmo tempo. Para que isso fosse possível, seria necessário um processador para cada tarefa. Assim, se o robô precisasse realizar dez tarefas ao mesmo tempo, seriam necessários dez processadores. Tal prática tornar-se-ia inviável pelo custo e complexidade de ter tantos processadores intercomunicantes. A prática adotada na quase totalidade de sistemas robóticos, e que atende à maioria das aplicações, é dividir o uso de um ou mais processadores com várias tarefas. Cada tarefa utiliza o processador durante um pequeno intervalo de tempo e passa a vez. Como o intervalo de tempo é pequeno, há a impressão que as tarefas estão sendo executadas ao mesmo tempo. Quando isso acontece, diz-se que o sistema possui tarefas concorrentes e o sistema operacional é classificado como multi-tarefa (FARINES; FRAGA; OLIVEIRA, 2000).

Se o usuário tivesse que fazer o controle de chaveamento entre as tarefas, o código gerado seria pouco legível e muito propenso a erros. Uma vez que a maior parte dos sistemas de computação necessita executar várias tarefas concorrentemente, a quase totalidade dos sistemas operacionais atuais provê um meio de facilitar a programação concorrente, por meio de uma API para criar tarefas e definir suas prioridades. A diferença básica entre os sistemas operacionais está na presença ou não de determinismo em relação ao tempo de execução das tarefas. Sistemas operacionais direcionados para aplicações de servidores ou ambiente gráfico (*desktop*), como Windows e Linux não provêem determinismo. Outros sistemas, como QNX e VxWorks, são direcionados a aplicações embutidas, onde o determinismo é essencial. Tais sistemas são denominados Sistemas Operacionais de Tempo Real (FARINES; FRAGA; OLIVEIRA, 2000), ou RTOS, da sigla em inglês.

Sistemas robóticos móveis são freqüentemente embutidos e possuem restrições de tempo severas. Tais sistemas necessitam portanto de um sistema operacional de tempo-real. A escolha do sistema operacional base para validar o presente trabalho é discutida na seção 3.5.

A estrutura de controle dos sistemas robóticos consiste, basicamente, de dois tipos de tarefas: tarefas periódicas e tarefas ativadas por eventos. Tarefas como tomadas de decisões e processamento de imagens costumam ser implementadas como tarefas fixas, executadas em um certo intervalo fixo de tempo e, por isso, são denominadas tarefas períodicas. Outras tarefas, como leitura dos sensores, podem ser executadas sempre que um novo dado estiver disponível, através de interrupções de *hardware*. Portanto, tais tarefas são ativadas por eventos.

Para que um sistema robótico não falhe, deve existir uma ou mais metodologias para garantir que as restrições de tempo de cada tarefa sejam atendidas, mesmo sob as piores circunstâncias. Ou seja, quando todos os possiveis eventos ocorrerem ao mesmo tempo, é preciso garantir que, mesmo sob essa condição, o sistema seja capaz de operar corretamente. Uma vez que o processador é projetado para atender à situação de pior caso, o que raramente acontece, a maior parte do tempo o processador é sub-utilizado. Na prática, isso não costuma acontecer, pois utiliza-se o poder de processamento disponível para executar outras tarefas que não tenham restrições temporais, ou que a falha em atendê-las, apesar de provocar alguma degradação no desempenho, não implicaria em qualquer desastre. Tais tarefas são denominadas *soft real-time*, enquanto tarefas críticas são denominadas *hard real-time* (FARINES; FRAGA; OLIVEIRA, 2000).

As diversas tarefas que executam concorrentemente em um sistema robótico são comumente inter-dependentes. Por isso deve haver uma forma de comunicação entre elas para que estejam em sincronismo. Por exemplo, uma tarefa que processe imagens para decidir

qual direção o robô deve seguir, deve esperar que a tarefa de captura de imagens termine de armazenar alguma imagem, para, enfim, processar a imagem. A seção 3.3 trata dos detalhes de comunicação entre tarefas.

3.3 Comunicação entre Tarefas

Existem diversas formas de comunicação entre tarefas, e todas elas devem ser implementadas de modo a não infringir as restrições de tempo real. Tais estratégias de comunicação entre tarefas serão abordadas nesta seção. Seu entendimento é importante para analisar e classificar as possíveis soluções, de modo a tomar uma decisão sobre a implementação de um ambiente para programação de robôs móveis em tempo real, que é o objeto de estudo da pesquisa.

Sempre que há necessidade de comunicação entre processos, é necessário que haja o cuidado de não permitir o acesso concorrente a uma estrutura de dados. Se um processo está alterando os dados de uma estrutura enquanto outro processo está lendo os dados da mesma estrutura, pode haver uma fragmentação dos dados que provocará resultados errados no processo de leitura. Da mesma forma, um resultado imprevisível pode ocorrer se dois processos tentarem alterar uma estrutura concorrentemente. Por isso, existem regiões críticas que devem ser programadas de modo a não permitirem o acesso concorrente a um objeto compartilhado.

3.3.1 Classes de modelos de linguagens de programação concorrente

Andrews e Schneider apontaram em (ANDREWS; SCHNEIDER, 1983) três classes de modelos de linguagens de programação concorrente: orientadas a procedimento, orientadas a mensagens e orientadas à operação. Embora tais linguagens sejam pouco usadas atualmente, os conceitos das classes de comunicação entre processos (IPC - *Inter-Process Communication*) em programação concorrente permanecem. Tais classes são as baseadas em compartilhamento de memória, mensagem e operação. Essa pesquisa não avaliou tecnologias tipo Produtor-Consumidor (*Publisher-Subscriber*), como CORBA (OPEN MANAGEMENT GROUP, 1991) e COM (MICROSOFT, 1993), inexistentes no ano da publicação. Como existe pouca referência em artigos a essas tecnologias aplicadas em Engenharia e, principalmente, em aplicações em tempo real, essas arquiteturas terão um tópico especial neste trabalho, o que não significa que elas tenham uma importância maior.

Em comunicações baseadas em compartilhamento de memória, os processos utilizam variáveis compartilhadas para se comunicarem. Para isso, é necessário que eles tenham

acesso a uma mesma região física de memória. É a forma mais eficiente de comunicação, mas não pode ser utilizada para computação distribuída, por não haver uma memória em comum, para ser compartilhada.

As comunicações baseadas em mensagem e operação são ambas baseadas em passagem de mensagens, mas possuem abordagens diferentes. As orientadas a mensagem, provêem interações de comunicação do tipo *enviar* e *receber*. Não há memória compartilhada. Quando uma operação deve ser realizada, uma mensagem é enviada para o processo responsável que realiza a operação sobre o objeto e possivelmente retorna uma mensagem de término da operação ou seu resultado. Deste modo, objetos nunca estarão sujeitos a acesso concorrente, já que apenas um processo é responsável por cada objeto ou recurso. As mensagens podem ser (e normalmente são) enfileiradas de acordo com a prioridade de cada processo. Isto é, há uma fila de mensagens para cada nível de prioridade. Um processo só terá sua mensagem processada quando não houver mensagens de processos com nível de prioridade superior na fila de mensagens do processo responsável (*caretaker*).

Comunicações baseadas em operação provêem chamadas a procedimentos remotos, mais referidas como RPC (*Remote Procedure Call*), como o meio primário de interação entre processos. Como acontece em comunicações orientadas a mensagens, onde cada objeto possui um processo responsável, existem procedimentos responsáveis por cada objeto ou operação. A diferença é que durante a operação, o procedimento responsável sincroniza os dois processos, o que simplifica a programação de uma tarefa, já que esta executa um procedimento remoto como se fosse um procedimento normal, sem se preocupar com questões de sincronização e acesso concorrente a um objeto (regiões críticas). O programador de uma tarefa não se preocupa onde está o procedimento remoto (não necessariamente remoto, mas pode estar sendo executado na mesma CPU). Este acesso é realizado transparentemente. Esta classe de comunicação é, normalmente, mais aconselhada em aplicações de computação distribuída.

3.3.2 Arquiteturas tipo Produtor-Consumidor

Como já foi mencionado, CORBA (*Common Object Request Broker Architecture*), cuja primeira especificação surgiu em 1991, 8 anos após a pesquisa de Andrews e Schneider, pertence ao conjunto de arquiteturas tipo Publicação-Inscrição (*Publisher-Subscriber*), similar às linguagens RPC, mas com uma abordagem de orientação a objeto. Assim, em vez de chamadas remotas a procedimentos, o programador requisita uma interface para um objeto que contém todos os métodos necessários para operar um recurso. O usuário do objeto não precisa se preocupar com regiões críticas e acesso concorrente, sendo que estas questões são implementadas em cada método do objeto. É possível conhecer a interface de um objeto

em tempo de execução, sem a necessidade de recompilação de código, o que implica em uma perda de desempenho durante a fase de conhecimento da interface, uma vez que serão necessárias chamadas adicionais a métodos com esse fim. A especificação CORBA (OPEN MANAGEMENT GROUP, 1998) original não permite especificar restrições de tempo real em sua implementação. Para isto, foi adicionada uma extensão da linguagem, denominada Real-Time CORBA (OPEN MANAGEMENT GROUP, 2003), (SCHMIDT; KUHNS, 2000), (WOLFE et al., 2000a), (WOLFE et al., 2000b) com o objetivo de suprir essa falha.

O projeto OCEAN (PROJETO OCEAN, 2001) tem o objetivo de implementar a extensão Real-Time CORBA em um sistema operacional de tempo real de código aberto, como o Linux com RTAI. Ainda não há um projeto que alcance este objetivo com uma implementação completa.

Nas arquiteturas Publicação-Inscrição, um objeto responsável por uma série de tarefas se registra (publicação), informando as operações que ele é capaz de realizar, e outro objeto ou função requisita uma referência para o objeto (inscrição) e realiza as operações necessárias por meio de chamadas de métodos do objeto. A localização do objeto, tal como ocorre com os procedimentos nas arquiteturas RPC, é transparente, ou seja, o cliente não precisa nem tem como saber em que computador em uma rede o objeto requisitado está localizado. Exemplos desse tipo de arquitetura são CORBA (OPEN MANAGEMENT GROUP, 1991) e COM (MICROSOFT, 1993).

Tais tecnologias permitem programar sistemas distribuídos mais facilmente e permitem uma integração fácil entre diferentes linguagens de programação, uma vez que os objetos registrados podem ser construídos virtualmente em qualquer linguagem. A figura 3.1 mostra o modelo de implementação de uma tecnologia CORBA. As outras tecnologias apresentam um esquema semelhante. Existe um canal comum, denominado ORB (Object Request Broker), cuja interface é conhecida por todas as linguagens que o utilizam, que faz a comunicação entre os clientes e os objetos publicados. Há também uma linguagem de definição de interfaces, a IDL (Interface Description Language), que é utilizada para descrever as interfaces de um objeto. Tal linguagem é traduzida por um compilador IDL para gerar o código correspondente para a linguagem de programação em que será realizada a implementação de um objeto ou de um código que o utilize. Assim, a comunicação com a ORB é transparente e o programador executa os métodos como faria normalmente em uma linguagem orientada a objetos. Também é possível implementar e utilizar objetos em linguagens não orientadas a objetos através de chamadas de funções ou o recurso mais apropriado para a linguagem. Em qualquer caso, o código gerado pelo compilador IDL é responsável por comunicar-se com a ORB para transmitir e receber informações. Também é possível se conhecer a interface de um objeto dinamicamente sem a necessidade de um código IDL, mas a complexidade da programação aumenta consideravelmente.

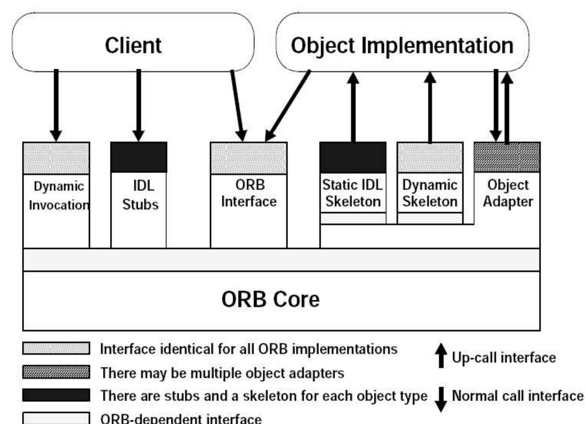


Figura 3.1: A estrutura de uma interface de requisição de objetos. Fonte: especificação CORBA.

Uma vez que não é possível saber a localização dos objetos, não é possível utilizar memória compartilhada como um método de IPC, já que não há garantia de que ambos os processos compartilhem uma memória física comum, havendo necessidade de cópias de dados mesmo quando os processos têm acesso a uma memória física compartilhada. Otimizações são possíveis para evitar-se essa sobrecarga. A ORB pode transmitir uma referência a posição de memória, porém o usuário não tem como saber se o método IPC sendo utilizado é de memória compartilhada ou cópia. Isso é um problema de eficiência crítico, pois certos processos podem ficar bem mais lentos se um objeto que se encontrava em um computador for transferido para outro. Diagnosticar o motivo de uma possível alteração de comportamento devido a essa transferência é muito complicado. Embora em muitas aplicações esse problema não seja tão crítico, nas aplicações em tempo real, onde o tempo de execução está associado ao funcionamento correto de um sistema, não é possível perder o controle sobre a localização dos objetos.

A extensão Real-Time CORBA (OPEN MANAGEMENT GROUP, 2003) não elimina o requisito de transparência de objetos, o que a torna inadequada para muitas aplicações em tempo real. Ainda não existe uma implementação de uma ORB que utilize Real-Time CORBA em um sistema de tempo real baseado em Linux, como o RTAI (DIAPM, 1998), (MANTEGAZZA; DOZIO; PAPACHARALAMBOUS, 2000) ou RTLinux (YODAIKEN; BARABANOV, 1996) e grande parte das publicações (LEVINE; FLORES-GAITAN; SCHMIDT, 1999), (SCHMIDT; DESHPANDE; O'RYAN, 2002), (HARRISON; LEVINE; SCHMIDT, 1997) relacionadas a esse tópico fizeram os testes em sistemas operacionais que não são em tempo real, como Windows e Linux tradicional, tornando os processos no maior nível de prioridade de tais sistemas; no mesmo nível dos *drivers*. O grande tempo de dedicação do projeto (ainda inacabado) OCEAN (PROJETO OCEAN, 2001) para integrar a especificação ACE/TAO a um RTOS baseado em Linux sugere que o sistema CORBA é muito complexo, o que acaba sendo

refletido no processo de depuração, quando algum *bug* é identificado no sistema. Portanto, é extremamente complicado garantir o funcionamento correto de um sistema de tempo-real que dependa de serviços CORBA para atender tarefas de tempo-real críticas.

Outra desvantagem é que a curva de aprendizado é muito elevada, exigindo muitos estudos para se desenvolver um simples projeto CORBA. O processo de instalação é complicado e o sistema é muito complexo, devido a sua generalidade. A complexidade do sistema, também implica em maior dificuldade de garantir um funcionamento correto do sistema devido ao número elevado de camadas. Desenvolver uma rede de comunicação em tempo real e integrá-la a ORB também não é uma tarefa simples. O *footprint*, que é a memória mínima necessária para fazer o sistema funcionar, também é muito grande devido à complexidade da solução. Como, em Robótica, é comum que haja escassez de recursos tanto de memória quanto de processamento, é interessante para uma plataforma de programação de robôs, que é o objetivo dessa pesquisa, que a solução seja eficiente e consuma o mínimo de memória. CORBA (e tecnologias semelhantes) foi desenvolvida para facilitar a programação de uma série de atividades, permitindo ao programador abstrair-se da comunicação entre objetos em uma rede, permitindo uma fácil programação de tarefas distribuídas. Porém, características como eficiência e previsibilidade, que são importantes nesse trabalho, foram substituídas por flexibilidade e facilidade nas arquiteturas Publicação-Inscrição. Portanto, apesar de facilitar a computação distribuída, CORBA, ou qualquer outra tecnologia tipo Publicação-Inscrição, não é adequada para a plataforma de programação de robôs móveis. Uma solução especialista é mais interessante para esse tipo de sistema.

3.4 A Plataforma de Teste

Com o objetivo de validar o *framework* proposto, foi construído um robô bem simples, sem qualquer capacidade local de processamento. Dessa forma, pode-se mostrar que é possível projetar um controle em tempo real para os robôs mais simples, no nível mais baixo, e atender as restrições de tempo, que são mais severas nesses casos. Assim, esse *framework* também deverá atender robôs com mais funcionalidades e, conseqüentemente, restrições de tempo menos severas.

Além disso, pretende-se mostrar que é possível reduzir custos ao se utilizar um único microprocessador, executando um sistema operacional de tempo real, de modo a executar as tarefas de controle dos sensores e atuadores concorrentemente com as tarefas finais a que se propõem os robôs. A plataforma de teste consiste em um robô simples com alimentação externa e controlado por um PC externo através de um cabo flat, como mostra a figura 3.2.

Futuramente, é possível implementar um robô com processamento a bordo, ou, simplesmente SBC – *Single Board Computer* – como é comumente conhecido, tal como o *Pioneer 3-DX* da *ActivMedia* (ACTIVMEDIA, 1995), porém com uma vantagem sobre este: apenas uma única unidade processada é necessária para se controlar o robô, como ilustra a figura 3.3. Isso permite menor custo e menos tempo de desenvolvimento. Mais detalhes sobre a construção do robô podem ser obtidas no anexo A.

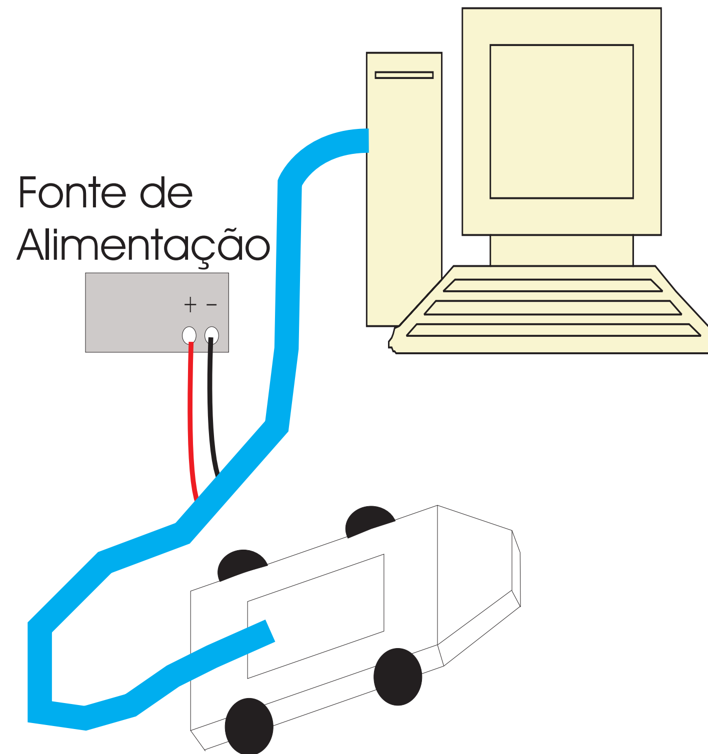


Figura 3.2: Plataforma de teste.

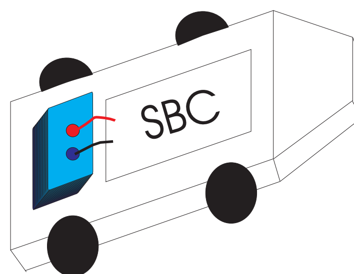


Figura 3.3: Projeto final do robô. É possível implementar um robô com bateria, placa mãe e processamento a bordo utilizando-se o *framework* desenvolvido.

3.5 Escolha do Sistema Operacional Base

Como já foi mencionado, o *framework* a ser desenvolvido deverá utilizar um sistema operacional de tempo real. Vários são os sistemas existentes, como o QNX, VxWorks, pSOS+, uITRON, VRTX, entre outros. Para escolhermos o sistema operacional a ser utilizado, escolhemos alguns critérios, compatíveis com os propostos no capítulo 1. São eles:

- **Facilidade de manutenção**: muitos sistemas operacionais de tempo real deixaram de ser mantidos. A escolha deve ser feita de modo que a probabilidade do projeto permanecer sendo mantido seja grande.
- **Integração com infra-estrutura existente**: é importante que o sistema operacional permita adaptar com facilidade códigos já existentes.
- **Promoção de reuso de *software***: para minimizar o trabalho de desenvolvimento e aumentar a robustez, uma vez que softwares existentes há muito tempo, normalmente, são bem testados.
- **Eficiência**: o sistema deve ser eficiente para atender as restrições temporais mais críticas.
- **Disponibilidade de código fonte**: para que seja possível garantir o funcionamento correto e identificar falhas e correções.
- **Independência de *hardware***: para que se possa utilizar diferentes tipos de robôs e sensores.
- **Qualidade de documentação**: para facilitar o processo de desenvolvimento.
- **Existência de ferramentas de depuração**: a existência de um depurador para o sistema operacional facilita o processo de desenvolvimento.
- **Custo envolvido**: uma vez que se pretende utilizar o framework também em projetos de pesquisa, é interessante que ele tenha um custo pequeno para viabilizar a maioria dos projetos.

A partir desses critérios, é fácil observar que um sistema operacional baseado no Linux é uma boa escolha, pois atenderia bem todos os critérios citados, exceto a qualidade de documentação, inferior a de outros sistemas operacionais. Porém, esta falta pode ser compensada pela diversidade de listas de discussões tratando sobre o sistema operacional.

3.5.1 Sistemas operacionais de tempo-real baseados em Linux

Existem algumas adaptações do Linux de modo a torná-lo um sistema operacional de tempo real. Estas adaptações envolvem alterações no código fonte do núcleo (*kernel*) do Linux, na forma de *patches*. Os projetos mais ativos são o RTLinux (YODAIKEN; BARABANOV, 1996), o RTAI (DIAPM, 1998) e o Xenomai (GERUM, 2004). Este último é um projeto derivado do *Fusion*, que foi uma transição do RTAI para o Xenomai.

Esses projetos provêem um ou mais escalonadores de tarefas em tempo-real, executando o escalonador do Linux como a tarefa de menor prioridade. A diferença entre eles está na abordagem. O projeto RTLinux foi o pioneiro e inclui muitas modificações no núcleo do Linux para incluir funcionalidades de tempo-real. Esse excesso de modificações torna difícil acompanhar as versões mais recentes do núcleo do Linux. Além dessa desvantagem, o fato de existirem duas versões - uma GPL e outra proprietária - sugere que a versão GPL possua recursos limitados em relação à proprietária. A empresa que suporta seu desenvolvimento, a RTLabs, tem interesses de comercialização de seu produto. Esse fator pode prejudicar o desenvolvimento da versão GPL e foi preferido evitar essa abordagem, já que existem outras sem fins lucrativos e de melhor qualidade, como será mostrado neste capítulo.

Ambos o RTAI e o Xenomai utilizam uma abordagem diferente. Eles utilizam o mesmo *patch* provido pelo projeto ADEOS, mantido por Philippe Gerum, o mesmo autor do projeto Xenomai. Esse *patch* é genérico e provê um maior controle sobre as funções do núcleo, como interrupções e gerenciamento de memória. Ele provê uma API sobre a qual pode ser implementado um ou mais núcleos de tempo real sob a forma de diferentes domínios. Desta forma, tanto o RTAI quanto o Xenomai criam um domínio diferente sobre o núcleo provido por ADEOS.

O núcleo do Linux possui dois espaços de endereços de memória: o espaço do núcleo e o espaço do usuário. Os programas escritos para o espaço do núcleo são os próprios programas ligados ao núcleo estaticamente ou os módulos do *kernel*. Esses programas possuem diversas limitações e não podem utilizar bibliotecas, nem mesmo a biblioteca *libc*, que provê as funções básicas definidas na linguagem C. Dessa forma, os recursos de programação são bem limitados, mas em algumas situações são a única forma de se comunicar com um determinado *hardware*. O espaço do usuário é o utilizado por programas criados pelos usuários do sistema operacional, e bibliotecas podem ser carregadas e utilizadas nesse espaço de endereços.

O projeto RTAI foi o pioneiro a utilizar uma abordagem que permitia a utilização de chamadas a serviços de tempo real a partir do espaço de memória do usuário, recurso esse denominado LXRT. Esta característica facilitava a programação em tempo-real com um pe-

queno aumento de latência, aceitável para a maioria dos projetos de tempo-real. O Xenomai, herdou essa mesma característica do projeto *Fusion*/RTAI. Atualmente, o projeto RTLinux também permite essa abordagem, em sua versão profissional.

O projeto Xenomai foi a extensão de tempo real escolhida, por alguns motivos:

- Provê as mesmas funcionalidades do RTAI.
- Seu autor e mantenedor é o mesmo do projeto ADEOS, o que implica que ele possui maior conhecimento sobre o núcleo, permitindo fazer um uso mais eficiente da adaptação ADEOS. Isso não signfica, contudo, que o projeto dependa exclusivamente deste autor, uma vez que há outros desenvolvedores ajudando a melhorá-lo e o projeto é bem estruturado. Porém, enquanto o autor estiver envolvido, haverá uma velocidade maior no aperfeiçoamento do Xenomai. Além disso, o autor já desenvolve sistemas de tempo-real há muitos anos e tudo indica que continuará por muito mais tempo, já que trabalha como consultor da OpenWide, uma empresa francesa que presta serviços de integração com programas de código aberto.
- Utiliza uma abordagem coerente nas transições para o domínio do Linux quando estas ocorrem, preocupando-se com questões como inversão de prioridades, elevando a prioridade do Linux quando uma tarefa depende de algum serviço deste. Obviamente, tais transições não devem ocorrer em situações críticas, mas são úteis em tarefas de menor prioridade para interagir com o usuário. Xenomai também provê outros serviços para comunicação com processos normais do Linux que podem ser utilizadas em tarefas de maior prioridade sem que essas transitem para o domínio secundário. Esta é a expressão usada por Philippe Gerum para se referir a uma tarefa que é transferida temporariamente para o escalonador do núcleo do Linux enquanto depender de serviços providos por este. Ver seção 3.5.2.
- Permite a utilização de diferentes *skins*, provendo diversas APIs para programadores de tempo real: uma API nativa, RTAI, VxWorks, Posix, pSOS+, uITRON, UVM e VRTX. Outras APIs podem ser implementadas utilizando os serviços básicos oferecidos pelo projeto *Xenomai*. Além disso, há o *skin* Real-Time Driver Model – RTDM, que provê uma API para facilitar a programação de drivers em tempo-real. Esta API permite a utilização de chamadas do sistema do tipo *open/close*, *read/write* e *ioctl*. Infelizmente, ainda não foram implementadas chamadas tipo *mmap*, *select* e *poll*. Chamadas *mmap* facilitam o mapeamento de uma faixa de memória física utilizada pelo driver para que possa ser utilizada diretamente pelo usuário que utilize o driver. Isso é útil, por exemplo, em aplicações de vídeo onde a memória pode ser compartilhada com o usuário, sem a necessidade de se copiar o conteúdo da imagem para a memória do usuário, como ocorre nas chamadas tipo *read*, o que reduz a eficiência. Nas

chamadas *mmap*, a memória é mapeada quando o usuário faz a requisição e depois é utilizada diretamente pelo usuário, sem a necessidade de cópias de um bloco grande de memória. Essa deficiência pode ser contornada, mapeando-se, na inicialização do programa, a memória no domínio secundário. Uma vez que a memória foi mapeada, o programa volta para o contexto de tempo-real. Isso é aceitável para a maioria dos programas de tempo-real, onde a inicialização pode ocorrer sem restrições temporais.

3.5.2 O problema da inversão de prioridades

Quando uma tarefa de maior prioridade depende de um recurso sendo utilizado por uma de menor prioridade, é dito ocorrer uma situação de inversão de prioridades. Se esta situação não for detectada pelo escalonador de tarefas, um comportamento ruim ocorrerá. Supondo um programa formado por três tarefas, onde P(T) designa a prioridade da tarefa T. Consideremos

$$P(T1) > P(T2) > P(T3).$$

Supondo R um recurso que só pode ser utilizado por uma tarefa de cada vez, imaginemos a seguinte situação. R está livre e T3 o ocupa. Enquanto T3 o ocupa, T1 aguarda a liberação de R para que possa utilizá-lo. Porém, T2, por possuir uma prioridade maior que T3, terá preferência no uso do processador em relação a T3. Como T1 depende de R, que está sendo utilizado por T3, na prática, T2 passa a ter uma prioridade maior que T1 nessa situação. Diz-se que houve uma inversão de prioridades, onde P(T2) passou a ser maior que P(T1).

Para evitar essa situação indesejada, o escalonador deve elevar a prioridade de T3 para P(T1) enquanto T3 utilizar R. Assim que T3 liberar esse recurso, sua prioridade volta a P(T3). Assim, o problema de inversão de prioridades é contornado. Diz-se que tal escalonador implementa prioridades dinâmicas.

A mesma situação acontece no Xenomai, quando uma tarefa depende de um recurso do Linux. A tarefa do escalonador de tarefas do Linux, que normalmente possuiria a menor prioridade no escalonador de tarefas do Xenomai, eleva sua prioridade quando uma tarefa Xenomai de maior prioridade depende de um recurso usado pelo Linux, até que o recurso seja liberado. Esse comportamento é perigoso e não deve ser usado em tarefas com restrições temporais críticas, pois uma vez que o Linux não provê garantias temporais, as tarefas com prioridade menor que a atribuída ao Linux nesse instante poderão ultrapassar *deadlines*.

É possível realizar a comunicação com processos normais Linux, utilizando *pipelines*, que são filas de comunicação. Obviamente, tal comunicação não ocorrerá em tempo-real, mas também não elevará a prioridade do Linux. Caso seja necessário que a utilização de um

driver do Linux ocorra com restrições temporais, um novo driver deverá ser desenvolvido com essas preocupações, adaptando o código original do driver para atender às restrições temporais.

3.6 Conclusão

Este capítulo mostrou a importância da escolha do método IPC a ser suportado pelo framework e classificou as alternativas existentes. Ele mostrou que arquiteturas tipo Publicação-Inscrição, como CORBA, não são adequadas a sistemas de robótica móvel, devido à baixa eficiência, complexidade desnecessária e dificuldade em garantir o comportamento correto, atendendo todas as restrições temporais, em qualquer condição.

Foi adotado o Xenomai, como um sistema operacional de tempo-real, baseado em Linux, utilizando critérios coerentes, para servir de sistema base para o desenvolvimento do *framework*. Também foi apresentada a estrutura sobre a qual foi construída o robô de teste.

Capítulo 4

O *Framework* Proposto

Para definir a interface do *framework*, que é a peça fundamental deste trabalho, é preciso conhecer os modos como robôs móveis são programados. Basicamente, há dois modos de programação de robôs móveis: baseados em movimentos e baseados em ações ou comportamentos.

Programação baseada em movimento são as mais simples e a interface é a mesma para todos os tipos de robôs móveis. Ela oferece, basicamente, métodos para ajustar as velocidades lineares e angulares do robô. O controle de velocidade é realizado de modo diferente para os diversos tipos de robô, mas a interface para o usuário final é sempre a mesma. Todo o controle do robô pode ser baseado nesses dois parâmetros de velocidade.

Aplicações mais complexas costumam utilizar o conceito de programação baseada em ações ou comportamentos. O robô precisa realizar várias tarefas independentes harmonicamente. Existem várias formas de implementar controle baseado em ações na literatura (ver seção 4.3), cada uma com seus prós e contras. O *framework* deve permitir o uso de qualquer um dos métodos disponíveis para ser flexível o suficiente para atender uma grande classe de aplicações. A seção 4.3 faz uma revisão dos métodos existentes e as decisões tomadas para a definição final da interface de programação baseada em comportamentos.

Aplicações muito simples utilizam somente a abordagem baseada em movimentos e não devem sofrer de *overhead* causado pelo código que lida com ações. Uma vez que a interface baseada em movimentos é bem simples, ela será definida primeiramente. Os robôs serão implementados em cima de uma classe base *Robô*. Uma vez que se quer oferecer reuso de código, é interessante que outras pessoas possam contribuir para o aperfeiçoamento do *framework*. Portanto, todo o código final será desenvolvido no idioma inglês, por ser o idioma mais aceito no mundo na área de Engenharia. Portanto, a classe base chamar-se-á *Robot*.

4.1 A Interface *Robot*

Em linguagens orientadas a objeto, o conceito de interface é uma definição de métodos a serem chamados pelo usuário final. Nenhuma implementação é definida. Em C++, interfaces podem ser implementadas por meio de uma classe abstrata. Tal classe não pode ser instanciada, mas serve de base para outras classes que implementam a interface. Como existem diferentes tipos de robô e todos eles possuem características comuns, a classe base deve prover a interface básica utilizada por todos os tipos de robôs. Esta classe receberá o nome de *Robot*.

Robot servirá de base para outras classes, como *Pioneer2DX*, *SimpleRobot*, etc. Alguns robôs podem ser enquadrados em categorias mais específicas, como a de robôs com tração diferencial. Desse modo, as classes Pioneer2DX e *SimpleRobot* seriam derivadas da classe *DiffDriveRobot*, que por sua vez seria derivada de *Robot*, como ilustra a figura 4.1.

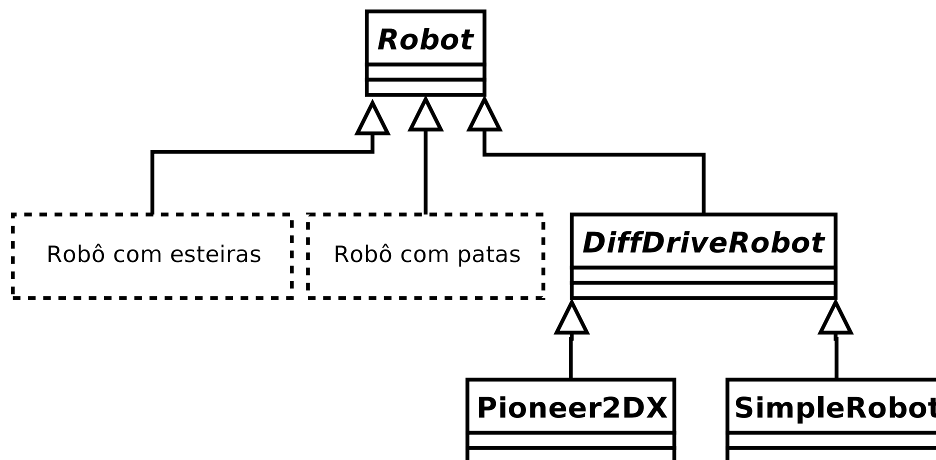


Figura 4.1: Herança de classes de robôs.

Essa abordagem explora o reuso de código, que além de facilitar a manutenção do código do *framework*, também reduz o tamanho final do arquivo binário, uma vez que evita a repetição de código. Ela também permite que algoritmos válidos para uma classe de robôs possam ser aproveitados para todos os robôs daquela classe. Por exemplo, um algoritmo desenvolvido para robôs com tração diferencial funcionarão sem modificação, tanto para o *Pioneer* como para o robô construído, detalhado no anexo A.

A interface *Robot* conterá os seguintes métodos: *SetVel(linear, angular)*, *GetLinearVel()* e *GetRotVel()*. Os tipos de entrada e saída serão definidos mais adiante, na seção 4.2. A interface *DiffDriveRobot*, derivada de *Robot*, conterá os seguintes métodos adicionais: *SetLeftSpeed(speed)*, *SetRightSpeed(speed)*, *GetLeftSpeed()* e *GetRightSpeed()*.

Em sistemas multi-robôs, cada robô possui uma identificação única. Por isso, existe um

método virtual *GetId()*, na classe *Robot*, para suportar aplicações de cooperação entre robôs. Para suportar aplicações com interface gráfica com o usuário ou simuladores, a interface *Robot* também possui o método *GetName()*, retornando um nome para cada robô.

Existem, basicamente, dois tipos de robôs. Aqueles que, como o *Pioneer*, são controlados por um microcontrolador dedicado e se comunicam com a placa mãe através de um protocolo e um canal de comunicação, e aqueles cujo sinal de controle é gerado pela mesma unidade que processa as tarefas do *framework*. Como o primeiro tipo envia comandos de velocidade para o micro-controlador, não faz sentido converter as velocidades linear e angular em velocidades das rodas direita e esquerda. Por isso, foi criada uma classe a mais para facilitar o desenvolvimento de robôs do segundo tipo: *DiffDriveRobotWithControl*. Portanto, na realidade, *SimpleRobot* será implementada com base nessa classe, já que pertence à segunda categoria de robôs, ou seja, uma única unidade de processamento controlará todo o robô.

Robôs controlados por um único processador possuem alguma forma de acesso aos atuadores (ou servos). Os servos podem ser controlados por um sinal PWM, um conversor D/A ou algum protocolo. Contudo, todos possuem a mesma finalidade e, por isso, foi criada uma interface para representar um atuador, que foi implementada na classe abstrata *ServoDriver*.

Seus métodos são *Init()*, *Start()*, *Stop()*, *SetSpeed(DistanceType speed)*, *GetSpeed()* e *GetDesiredSpeed()*, todos auto-explicativos. Em servos onde o controle é realizado por *hardware* e não há realimentação externa sobre a velocidade real, *GetSpeed()* pode retornar o mesmo que *GetDesiredSpeed()*. Esse é o comportamento padrão, quando o método *GetSpeed()* não é implementado, na classe filha.

A classe *DiffDriveRobotWithControl* contém dois ponteiros para instâncias de *ServoDriver*, um para cada roda. Um robô real, que implemente a interface *DiffDriveRobotWithControl*, deverá atribuir os ponteiros para objetos gerados a partir de classes que implementam *ServoDriver*.

Para o caso do robô desenvolvido, o servo é controlado por um sinal PWM. Por isso, foi desenvolvida uma interface *PWMDriver*, derivada de *ServoDriver*. Essa interface é válida para os diversos tipos de atuadores baseados em PWM.

Uma vez que outros tipos de circuitos são controlados por sinal PWM, foi desenvolvida uma interface para gerar um sinal PWM, a *PWMGenerator*. *PWMDriver*, por sua vez, contém uma instância de *PWMGenerator*. Desse modo, é incentivado o reuso de software, facilitando a manutenção do *framework* e reduzindo o tamanho do código binário. A implementação usada em *SimpleRobot* para gerar um sinal de controle PWM pela porta paralela foi desenvolvida na classe *SimpleRobotPWMGenerator*, derivada de *PWMGenerator*, implementando seus métodos puramente virtuais *DutyOn()* e *DutyOff()*.

A figura 4.2 ilustra a relação entre classes para o robô construído, implementado na classe *SimpleRobot*, e resume a explicação acima. *SimpleRobot* atribui aos ponteiros para *ServoDriver* das rodas direita e esquerda para duas instâncias de *PWMDriver*. Esses ponteiros são herdados de *DiffDriveRobotWithControl*. Ao construtor de *PWMDriver*, é passado qual será a classe geradora de pulsos PWM, nesse caso, *SimpleRobotPWMGenerator*. Ao construtor de *SimpleRobotPWMGenerator*, é passado um tipo enumerado, contendo os valores *WheelLeft* e *WheelRight*, que indicam para qual roda os pulsos serão gerados.

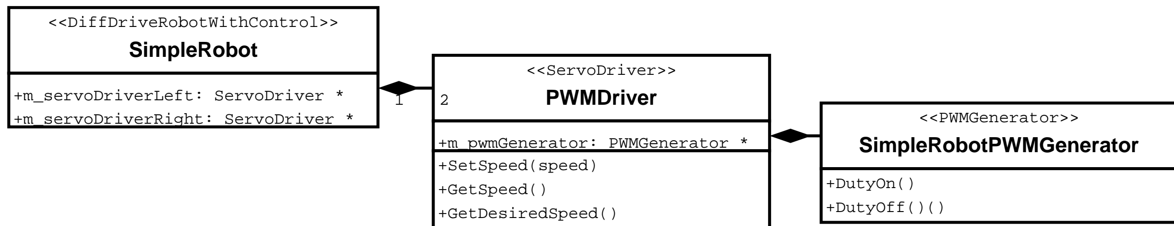


Figura 4.2: Relação entre classes para o robô construído (*SimpleRobot*).

4.2 Tipos de Dados e Unidades

O tipo de dados a ser utilizado no *framework* pode afetar diretamente o desempenho da aplicação final. Algumas arquiteturas não possuem suporte em *hardware*, por exemplo, para realizar operações com ponto-flutuante. O uso de um tipo ponto-flutuante em tais arquiteturas reduziria bastante a eficiência em comparação com tipos inteiros, enquanto os últimos atenderiam bem a uma série de situações.

Entretanto, o uso de inteiros pode prejudicar o desempenho em arquiteturas com suporte em hardware a operações de ponto-flutuante. Operações de ponto-flutuante, embora levem mais ciclos que as operações inteiras, podem ser mais eficientes em alguns casos, se usadas corretamente. Elas podem ser executadas em paralelo em seqüências de operações e podem tirar maior proveito de memória *cache* (como L1 e L2). Além disso, uma instrução de ponto-flutuante pode começar, em alguns casos, antes da anterior terminar (*pipeline*). Conversões entre inteiro e ponto-flutuante também consomem ciclos de processamento. Assim, um programa que precise usar tipo ponto-flutuante, para algum dado, pode preferir usar o mesmo tipo para dados inteiros em certas circunstâncias.

Por essas e outras razões, o *framework* não deve definir previamente o tipo a ser utilizados para os dados internos, pois poderia impedir que certas otimizações sejam realizadas. A forma comumente adotada em C++ para resolver esse tipo de situação é a definição de um tipo *AngleType*, através da palavra reservada **typedef**: "**typedef float** *AngleType*;". Dessa

forma, os tipos de parâmetros das funções e métodos poderão ser alterados, de uma vez, com uma única diretiva de compilação.

Usando a mesma idéia dos tipos de dados, o *framework* deve ser flexível também para lidar com unidades de comprimento, ângulo e tempo. Se definirmos previamente as unidades, as unidades escolhidas podem não ser convenientes para o usuário final. Algumas ainda podem impedir uma melhora na eficiência de certas aplicações. Por exemplo, ao se trabalhar com imagens, nem sempre é necessário exprimir a distância em milímetros, centímetros ou polegadas. Muitas vezes é mais interessante trabalhar diretamente com o número de pixels, tornando o código mais rápido já que nenhuma conversão precisaria ser realizada.

Outros algoritmos, contudo, dependem de alguma unidade do mundo real. Por exemplo, um robô que precisa se deslocar um metro pra frente, deve saber qual a unidade que é utilizada pelo robô para fazer as conversões necessárias. O número de pixels por si só não é capaz de dar essa informação.

Uma solução que provê essa flexibilidade e é utilizada nesse trabalho é a seguinte: são definidos tipos para distância, ângulo e tempo. As unidades são definidas pelo robô e são referenciadas por *au* (*angle unit*), *du* (*distance unit*) e *tu* (*time unit*). A classe *Robot* conterá métodos que retornam os fatores de conversão para as unidades milímetro, radianos e segundos. Os fatores de conversão são do tipo ponto flutuante, mas isso não gera qualquer ineficiência, já que os valores são estáticos e podem ser convertidos e armazenados em outro tipo, durante a inicialização, se for conveniente.

4.3 Abordagens sobre Programação Baseada em Ações

Uma vez definidos os métodos básicos de movimento, é necessário definir como será a interface para lidar com ações (ou comportamentos). Para definir a interface, é importante conhecer o estado da arte da programação baseada em ações. Após apresentar um resumo sobre as técnicas existentes, será possível apresentar uma solução que permita o uso de qualquer das técnicas, para manter a flexibilidade do *framework*.

Em (ROSENBLATT, 1995), Rosenblatt faz uma revisão sobre as abordagens clássicas e introduz outra. Um resumo sobre essas abordagens são apresentados nas próximas seções.

A idéia de usar ações independentes para controlar um robô é fundamental para que um robô realize tarefas mais complexas. Cada ação é implementada de modo independente, sem se preocupar com as outras, o que facilita o desenvolvimento e compreensão da aplicação final. O robô, contudo, precisa realizar um movimento final, tipo seguir em frente, virar para

a esquerda ou esperar parado. O processo de decisão sobre qual será o movimento final do robô baseado nas ações ativas é complicado e é objeto de estudo de vários trabalhos. Cada método existente na literatura tem seus prós e contras e vão se encaixar melhor em algumas aplicações do que em outras. Por isso, o *framework* não pode escolher um que julgar melhor, pois esse conceito não existe.

Existem compromissos nas tomadas de decisões. O fator principal na escolha do método é o tempo. Algoritmos levam tempo para serem processados. O robô tem um tempo limite para tomar decisões ou a decisão não se aplicará mais. Portanto, se uma cadeira de rodas inteligente demorar a tomar a decisão de parar ou recuar ao perceber uma escada, quando a decisão estiver pronta, o paciente já terá caído pela escada.

Existem, então, duas categorias básicas de controle: **reativo** e **planejado** (ROSENBLATT, 1995). Ambos precisam ser combinados para que um supra as necessidades do outro. Controle reativo realiza uma ação imediata sem planejamento baseada em alguma informação sensorial, como um obstáculo encontrado por exemplo. Ele é importante para a proteção do robô em si e seu ambiente em volta, evitando o problema da cadeira de rodas e a escada, por exemplo.

Porém, o controle reativo não é capaz de alcançar objetivos maiores, como atravessar uma cidade por exemplo. Para isso existe o controle planejado. Quanto maior o planejamento, maior o custo computacional, e se este for excessivo, a decisão final poderá não mais ser válida. O que as arquiteturas de controle se propõem a fazer é estudar como devem ser as relações entre o controle reativo e o planejado.

As arquiteturas de controle podem ser classificadas em **centralizadas** e **distribuídas** (ROSENBLATT, 1995). As arquiteturas centralizadas possuem a vantagem de poder tomar uma decisão coerente com os desejos das ações ativas, para atingir múltiplos objetivos. As arquiteturas distribuídas são capazes de responder mais rapidamente a estímulos sensoriais, provendo reatividade, flexibilidade e robustez.

Fusão Sensorial vs. Arbitragem de Comandos

Arquiteturas também podem ser caracterizadas pelo modo que combinam as informações sensoriais e objetivos (ROSENBLATT, 1995). As que realizam fusão sensorial, buscam criar um modelo do mundo baseado na informação obtida por todos os sensores disponíveis, fazendo fusão de informações, possivelmente redundantes. Isso aumenta a confiabilidade das informações e permite o uso do modelo criado para tomar decisões. Porém, é criado um gargalo computacional com custo muito grande, não satisfazendo os requisitos de tempo em muitos sistemas robóticos.

Arquiteturas baseadas em comportamentos não criam um modelo central do mundo. Elas agem localmente de acordo com os sensores que cada ação julgar necessários. Isso as permite tomar decisões mais rapidamente. Por isso, diz-se que um comportamento é um módulo auto-contido. Cada comportamento indica um comando ao módulo de *Arbitragem de Comandos*. Este módulo, por sua vez, decide que ação o robô realizará baseado nos comandos recebidos por cada comportamento.

Controle de Baixo pra Cima vs. Cima pra Baixo

Os sistemas de controle são classificados como sendo do tipo de Baixo pra Cima (*Bottom-Up*) e de Cima pra Baixo (*Top-Down*) (ROSENBLATT, 1995). Em arquiteturas hierárquicas, o controle é realizado normalmente de cima pra baixo. As tarefas de nível mais alto comandam as tarefas de nível mais baixo. Adicionalmente, apenas um módulo em um dado nível é ativado pelo módulo um nível acima.

Nas *Arquiteturas de Subsunção* (*Subsumption*) (ROSENBLATT, 1995), os comportamentos estão ativos o tempo todo, no sentido que sempre processam dados quando disponíveis. A estrutura de controle é de baixo pra cima porque cada comportamento decide por si só se a informação é relevante para aquele comportamento, baseado nos dados que ele recebe. Ligações de inibição são usadas para definir a saída do comportamento que será usada para controlar o robô.

As abordagens hierárquicas e baseadas em comportamento são combinadas na *abordagem hierárquica com evento superveniente* (*supervenience*) (ROSENBLATT, 1995), nas quais comportamentos recebem inibição e ativação de outros comportamentos. Comportamentos de níveis mais altos só trocam informações com comportamentos de nível mais baixo.

Outra classe de arquitetura de baixo pra cima é a arquitetura *emergente* (ROSENBLATT, 1995), na qual os comportamentos recebem ativação e inibição de outros comportamentos, todos eles competindo pelo controle do robô. Ambas as estruturas hierárquicas e prioridades pré-definidas são evitadas para se obter maior reatividade. Em algumas dessas arquiteturas, os comportamentos ou ações recebem ativações baseadas em motivações, refletindo se os seus alvos estão sendo alcançados ou não. Tais arquiteturas são mais flexíveis mas possuem o inconveniente de serem menos preditíveis.

Uma outra abordagem baseada em comportamentos e não hierárquica é utilizar um planejador de alto nível, participando no controle compartilhado do robô. Em vez de planos, o planejador provê conselhos ao sistema.

Decisões Baseadas em Campo de Força

Uma idéia para decidir a ação final a ser executada pelo robô é realizar a soma vetorial das velocidades desejadas por cada comportamento (ROSENBLATT, 1995). É uma idéia intuitiva e que é muito usada em algoritmos para mover o robô até um determinado alvo, desviando de obstáculos. O obstáculo seria representado por um campo de força que repele o robô para fora dele. O alvo é representado por um campo de força que atrai o robô. O caminho seguido pelo robô será determinado pela força vetorial da interação entre os campos. Embora a idéia funcione bem para esse caso, em outras aplicações, o movimento final pode não satisfazer a qualquer dos comportamentos.

Arquitetura Distribuída para Navegação Móvel - DAMN

Rosenblatt apresentou outra solução, a qual batizou de DAMN (Distributed Architecture for Mobile Navigation) (ROSENBLATT, 1995). A idéia é ter um planejador que recolhe votos dos comportamentos sobre um conjunto de ações possíveis de serem executadas pelo robô. Basicamente, a ação que receber mais votos será eleita para guiar o robô. Essa abordagem evita o problema das decisões baseadas em campo de força, mas exige que um número de possíveis ações seja selecionado e avaliado por cada comportamento, tornando o processo relativamente custoso e menos reativo.

4.3.1 A Abordagem Usada no *Framework*

Uma vez que existem várias formas de se controlar robôs, através de sistemas baseados em comportamentos, cada uma será mais indicada para um determinado contexto. A única característica comum entre os autores das diversas abordagens é que aplicações que exigem um nível mais alto de objetivos a serem cumpridos precisa usar uma abordagem baseada em comportamentos ou ações.

Por isso, a interface do *framework* deve considerar a possibilidade de se programar comportamentos no robô. É impossível implementar uma única interface que possibilite as diversas abordagens citadas, já que elas utilizam diferentes dados de entrada e saída. A idéia, então, é permitir ao usuário escolher como gerenciar os comportamentos. Desta forma, foi criada uma classe *ActionManager*, que gerenciará as ações ou comportamentos. Cada robô conterá um ponteiro para um gerenciador de ações. Dessa forma, é possível reduzir o *overhead* pra robôs utilizando a abordagem puramente baseada em movimentos, atribuindo o valor 0 (zero) a esse ponteiro. Além disso, vários robôs podem usar o mesmo gerenciador, alternando-se somente as ações a serem gerenciadas.

Falta definir, contudo, como será a classe *ActionManager*. Uma vez que as abordagens existentes são bem diferentes, a classe *ActionManager* deve se comportar de modo diferente para as diferentes abordagens. Algumas soluções são possíveis, como o uso de templates, a definição de um tipo ou de uma classe abstrata e o uso de espaço de nomes.

4.3.2 Uso de *Templates*

É possível utilizar *templates* para as classes Robot e suas derivadas, passando o tipo de *ActionManager* desejado. Desse modo, as classes de robô seriam definidas como *Robot* $<$ *ActionManagerType* $>$, *DiffDriveRobot* $<$ *ActionManagerType* $>$ e *SimpleRobot* $<$ *ActionManagerType* $>$. Externamente ao *framework*, o usuário só declarará as classes de robôs reais, como *SimpleRobot* $<$ *ActionManagerVotes* $>$ e *Pioneer2DX* $<$ *ActionManagerVotes* $>$ para utilizar uma abordagem baseada em votos, como apresentada na seção 4.3, por exemplo.

O uso de templates permite uma chamada consistente dos métodos do gerenciador de ações. Contudo, mensagens de erro utilizando *templates* são mais complicadas de serem entendidas. Muitos programadores de robôs não têm formação em computação e possuem conhecimento limitado sobre linguagens de programação. Muitos estão habituados à programação em C e conhecem um mínimo de C++. *Templates* é uma ferramenta da linguagem C++ que muitos engenheiros desconhecem e seu uso poderia dificultar a aceitação de um *framework* que eles não dominam.

Mesmo para bons programadores, o uso de *templates* apresenta outra desvantagem. Se for desejado utilizar múltiplos robôs, cada um com uma abordagem diferente para lidar com comportamentos, muito código será duplicado se *templates* forem utilizados. Esse problema é mais relevante em arquiteturas com pouca memória, visto que as classes de robôs não ocupam tanto espaço. Porém, quando se programa um robô com paradigma de programação baseada em comportamentos, é usual assumir que o sistema é mais complexo e possui mais memória e poder de processamento.

Existe ainda uma inconveniência quando se trabalha com *templates*. Cada método estático seria chamado através de SimpleRobot<ActionManagerVotes>::GetMmPerDu() em vez de simplesmente SimpleRobot::GetMmPerDu(), para obter a conversão de unidades de distância para milímetros, por exemplo. Outro problema é que seria preciso uma série de diretivas condicionais para retirar os templates em aplicações que utilizam uma abordagem puramente baseada em movimento, o que dificultaria muito a leitura e manutenção do código.

4.3.3 Definindo um Tipo para *ActionManager*

Uma alternativa para o uso de *templates*, é a definição de um tipo *ActionManager*, através da diretiva ***typedef***, que pode ser alterado em tempo de compilação. Desse modo, obter-se-ia um resultado parecido com o uso de *templates* na consistência ao chamar os métodos da classe *ActionManager*, que são diferentes para cada abordagem. Isso eliminaria boa parte dos problemas apresentados pela técnica do uso de *templates*.

Contudo, essa técnica impede o uso de abordagens de comportamento diferentes para cada robô, pois apenas uma abordagem seria definida em tempo de compilação.

4.3.4 Uso de um Espaço de Nomes

Uma solução similar à definição de um tipo para *ActionManager*, é o uso de um espaço de nomes, ou ***namespace***, como é definido em C++. Assim, em vez de usar diretivas de compilação para escolher qual gerenciador de comportamentos usar, a definição seria dada, por exemplo, do seguinte modo, para utilizar a abordagem por votos:

using namespace *ActionManagerVotes*;

4.3.5 Uso de uma Classe Abstrata Vazia

Outra idéia, é implementar uma classe *ActionManager* totalmente vazia, que servirá somente de base para que gerenciadores reais de ações sejam implementados. Na verdade, haveria alguns métodos comuns na classe base: *GetType()*, *Init()*, *Start()* e *Stop()*. O primeiro retorna o tipo de abordagem sendo usada. Desse modo, um algoritmo genérico poderia fazer a conversão para o tipo correto em tempo de execução. Obviamente, tal algoritmo faria com que a aplicação final ficasse maior porque conteria as implementações de todas as classes de gerenciadores de comportamentos suportadas pelo algoritmo.

Diferentemente das outras duas abordagens, não será possível utilizar o *ActionManager* diretamente nessa abordagem. Será sempre necessário fazer uma conversão para o tipo correto no programa do usuário. Isso é um incômodo para o usuário, mas provavelmente é um incômodo menor que o uso de *templates* para a maioria deles.

4.3.6 A Implementação Escolhida

As implementações citadas não apresentam qualquer diferença significativa no desempenho final das aplicações. Por isso, o método de escolha será baseado em facilidade de uso pelo usuário final e de manutenção do código interno ao *framework*, além da flexibilidade oferecida. O uso de templates introduz certa complicação, ambos para o usuário final e para a manutenção do código fonte do *framework*, e por isso não será usado.

O uso das opções de definição de tipos ou espaço de nomes provê mais facilidade de uso por parte do usuário final, mas menos flexibilidade, em comparação com o uso de uma classe vazia. Portanto, escolheu-se utilizar a abordagem da classe vazia, obtendo maior flexibilidade, uma vez que é possível contornar o problema das conversões de tipo necessárias. É possível definir um ponteiro global, por exemplo, para um tipo específico de gerenciador de ações, no início da aplicação. Desse modo, somente uma conversão de tipo seria necessária.

4.4 Tarefas, *Xenomai* e C++ - A classe *Task*

Sistemas de tempo-real são formados por tarefas que executam concorrentemente. A cada tarefa é atribuída uma prioridade. Sistemas operacionais de tempo real provêem uma API para a criação de tarefas a serem executadas em contexto de tempo-real. Para reduzir a dependência do *framework* em relação ao sistema operacional escolhido no capítulo 3.5 e também para prover uma interface mais simples para o usuário final, diminuindo o risco de erros de programação na criação de tarefas, criou-se uma interface *Task*. Dessa forma, métodos comuns a tarefas de tempo-real em qualquer RTOS são executados na aplicação do usuário, enquanto a implementação real está interna ao *framework*, permitindo que muitas aplicações possam ser portadas para outro RTOS, através de mudanças na implementação da interface *Task*.

Tal interface esconde os detalhes de implementação do usuário, provendo métodos simples, como *Init()*, *Destroy()*, *Resume()*, *Suspend()* e *Entry()*. O primeiro cria a tarefa, enquanto que *Destroy()* a exclui. *Resume()* inicia (ou continua) a execução da tarefa, enquanto que *Suspend()* a suspende. *Entry()* é um método puramente virtual que contém o código principal da tarefa. Uma tarefa real deve ser derivada de *Task* e implementar *Entry()*. Opcionalmente é possível sobrescrever os outros métodos citados, com o cuidado de chamar os métodos da classe base, adicionando-se o código que for específico da tarefa.

A principal dificuldade em se criar essa classe, e que vale a pena notar, está no encapsulamento das funções do *skin native* da extensão *Xenomai*. Todas as APIs do *Xenomai* e da

maioria dos RTOS são implementadas na linguagem C, sem suporte a orientação a objetos. A API *native* separa a criação da tarefa de sua execução através das funções *rt_task_create()* e *rt_task_start()*. A última exige um ponteiro para uma função que é o ponto de entrada da tarefa. Essa é sua definição:

```
rt_task_start (RT_TASK *task, void(*entry)(void *cookie), void *cookie);
```

Não é possível converter um método comum de uma classe em C++ para um ponteiro para uma função porque o método está associado a uma instância da classe, implicitamente, através do ponteiro ***this***. Uma opção seria solicitar ao usuário um ponteiro da função de entrada, através de um parâmetro de entrada de algum método da classe *Task*. Porém tal abordagem seria tanto inconveniente para o usuário, quanto abriria mão da vantagem de se utilizar o encapsulamento em linguagens orientadas a objeto. Um problema mais sério é que a função deveria ser copiada para cada instância de uma determinada tarefa, ou então deveriam chamar uma função comum, alterando os parâmetros de entrada.

O método adotado provê uma solução bem mais elegante. O terceiro parâmetro, *cookie*, é um ponteiro para alguma estrutura definida pelo usuário e é o único parâmetro para a função que é o ponto de entrada. O ponteiro ***this*** é passado nesse parâmetro, de modo que apenas uma função, na verdade, um método, é executado no ponto de entrada da tarefa, como mostra o seguinte código:

```
void funcao(void *cookie) { ((Task *) cookie)->Entry(); }
```

Uma vez que *Entry()* é um método virtual, ele é diferente para cada instância de uma tarefa. Além disso, não é necessário passar qualquer parâmetro para *Entry()*, uma vez que qualquer dado desejado pode ser incluído na classe da tarefa (derivada de *Task*) em si. O método *Entry()* está associado a um ponteiro ***this***, único para cada instância. O uso de uma função de entrada, contudo, ainda não provê uma solução elegante, pois ela estaria declarada fora da classe. Em vez dela, é utilizado um método estático, que funciona exatamente como uma função para esse propósito, com a vantagem de não poluir o espaço de nomes com o nome de uma função específica.

Desse modo, é possível portar tal classe para qualquer outro sistema operacional que permita obter um ponteiro para uma estrutura definida pelo usuário. Em outros sistemas, em que o processo corrente é transformado em uma tarefa de tempo-real, como o RTAI, também é possível implementar a mesma interface *Task*, através da sincronização dos métodos *Init()*, *Resume()* e *Suspend()* com o método *Entry()*, a ser transformado em uma tarefa de tempo-real.

Para facilitar a programação de tarefas periódicas, foi criada outra interface, a ***PeriodicTask***, derivada de *Task*. *PeriodicTask* implementa o método puramente virtual *Entry()*,

executando um novo método puramente virtual, o *DoCycle()*, a ser executado a cada período definido por *SetPeriod()*. O período pode ser obtido por *GetPeriod()*. A ocorrência de um *deadline* provoca a execução de outro método puramente virtual, o *OnOverruns()*, que recebe o número de *deadlines* perdidos e retorna verdadeiro ou falso, dependendo se a tarefa deve continuar ou parar, respectivamente. Outros tipos de erro são tratados em outro método puramente virtual, o *OnError()*, também retornando verdadeiro ou falso para indicar se a tarefa deve ou não continuar.

4.5 O Gerenciador de Ações Padrão

Para demonstrar como implementar um gerenciador de ações a ser utilizado pelo robô, foi criado um gerenciador padrão para o *framework*, que é utilizado quando não se explicita qual gerenciador usar. Esse gerenciador foi implementado em uma classe denominada *AMBasic*, dentro de um espaço de nomes com o mesmo nome e utilizando um modelo bem simples, baseado na idéia empregada no *framework* ARIA (ACTIVMEDIA ROBOTICS, INC, 2004).

Um espaço de nomes foi utilizado para que classes comuns em diferentes arquiteturas de gerenciamento de ações possam utilizar o mesmo nome, como *Action*, por exemplo. Caso contrário, um prefixo deveria ser adicionado a cada classe para evitar conflito de nomes entre diferentes arquiteturas, o que dificultaria a leitura do código.

A idéia empregada em ARIA consiste em separar o movimento do robô em dois movimentos básicos: translação e rotação. Cada ação especifica, basicamente, qual movimento deseja que o robô execute e a intensidade do desejo, ou seja, o quão forte o comportamento deseja realizar aquela ação. A cada intervalo de tempo fixo definido, é calculado o movimento final do robô com base nos desejos de cada ação. Em outras palavras, esse modelo é uma variante da técnica de decisão do movimento do robô baseado em campos de força, conforme explicado na seção 4.3. Existe uma intensidade de desejo para cada canal especificado. Simplificadamente, os canais são: velocidade linear, velocidade angular, posição e ângulo desejados.

Cada ação ou comportamento possui uma prioridade. O gerenciador de ações possui um resolvedor, que é incumbido de movimentar o robô de acordo com os desejos de cada ação e suas prioridades. Um resolvedor padrão é adotado caso não se especifique qual resolvedor utilizar. O resolvedor padrão funciona do seguinte modo. Ações de mesma prioridade são agrupadas e uma média é calculada para cada canal. As intensidades de desejo das ações, que variam de 0 a 1, são somadas até que o valor 1 seja atingido. A partir daí, as outras ações param de ser processadas. Desse modo, somente as ações de maior prioridade são processadas, até que a soma das intensidades de desejo atinja o valor 1.

A implementação adotada no gerenciador de ações padrão desse *framework* difere da implementação empregada por ARIA em um aspecto. No *framework* ARIA, a intensidade total, a ser comparada com o valor 1, é calculada através da média das intensidades de tarefas agrupadas com mesma prioridade. Porém, um cálculo mais coerente é somar sempre as intensidades em vez de avaliar a média para tarefas de mesma prioridade. Se duas tarefas de proteção do robô possuem a prioridade máxima e apenas uma deseja executar fortemente uma ação, de acordo com a técnica empregada por ARIA, a tarefa que deseja fracamente executar uma ação irá enfraquecer o desejo do agrupamento das duas tarefas, o que pode gerar um acidente.

4.6 O *Driver* de Captura de Imagens Desenvolvido

O *framework* desenvolvido deve ser capaz de realizar tarefas robóticas complexas. Tais tarefas, freqüentemente se utilizam de câmeras de vídeo para processar imagens. Um *framework* robótico, portanto deve ter a possibilidade de se programar e utilizar *drivers* que provejam garantias de tempo-real. Como prova de conceito, foi desenvolvido um *driver* de captura de imagem, que implementa restrições temporais. O laboratório dispõe de um *frame-grabber* (placa de captura de imagem) da Data Translation (DT-3153), utilizando um barramento PCI e sua respectiva especificação.

Como não havia um *driver* já desenvolvido para o Linux, inicialmente foi interessante desenvolver um, utilizando a interface padrão *Video For Linux 2* (V4L2), a fim de testá-lo no Linux com aplicações já existentes. O Xawtv, por exemplo, é capaz de exibir imagens capturadas de *drivers* utilizando a API V4L2, assim como mudar propriedades como brilho, saturação e tamanho da imagem.

Uma vez desenvolvido esse *driver*, foi necessário adaptá-lo para que atendesse a restrições temporais. Tal *driver* necessitaria de chamadas do sistema tipo *ioctl*, entre outras. Tais chamadas (*ioctl*) são gerenciadas pelo *Linux* através de interrupções de *software*, mais precisamente, *int 80*. Portanto, elas não são executadas em contexto de tempo real uma vez que dependem da execução do *Linux*, que é a tarefa de menor prioridade e não garante o atendimento de restrições temporais.

4.6.1 Abordagem de *drivers* de tempo-real no Xenomai

Para endereçar esse problema, o projeto *Xenomai* oferece um *skin* (RTDM) que provê uma API semelhante para chamadas tipo *ioctl* de um modo determinístico. A diferença na

API está no tipo de retorno. Nos sistemas POSIX o valor de retorno é zero quando não há erros. Caso contrário o valor de retorno será -1 e o código de erro será informado através da variável *errno*. No RTDM o valor de retorno será positivo quando não houver erro, e negativo quando houver erro. Nesse caso, o código de erro será o negativo do valor de retorno. Ver o exemplo abaixo.

Sistemas POSIX: {... if (erro) {errno=EAGAIN; return -1;} ...}

RTDM: {... if (erro) return -EAGAIN; ...}

É interessante que a API disponibilizada pelo *driver* seja a mais próxima possível da V4L2. Isso facilitaria a reutilização de códigos já existentes diminuindo os esforços necessários para adaptá-los a um sistema de tempo real.

Como mencionado na seção 3.5, a interface provida por RTDM não inclui chamadas tipo *mmap*, *select* e *poll*. Desta forma, não é possível implementar uma API idêntica a V4L2. Algumas soluções são possíveis:

1. Implementar tais chamadas no RTDM e manter a compatibilidade com V4L2;
2. Realizar o mapeamento de memória e as funções oferecidas por *select* e *poll* através de chamadas tipo *ioctl* e adaptar a API de acordo.
3. Utilizar descritores de arquivos padrões do Linux e realizar o procedimento de mapeamento de memória durante a inicialização da aplicação. Tal procedimento não se dará em tempo real, mas tampouco isso é necessário na maioria das aplicações. A API também será ligeiramente modificada.

As soluções 1 e 2 exigem um conhecimento profundo sobre a estrutura do Linux e sobre como ele lida com memória virtual. Tal conhecimento requer muito tempo de estudo e está fora do escopo desse trabalho. A solução 1 seria a mais elegante, mas também exige mais conhecimento para sua implementação.

O caminho 3 é mais rápido e serve como um atalho, enquanto as opções 1 e 2 ainda não forem desenvolvidas e, por isso, será utilizado nesse trabalho. Assim, poder-se-ia utilizar o descritor de arquivo */dev/mem* providos pelo Linux para mapear a região de memória utilizada pelo *driver*. Contudo, tal abordagem provê pouco controle sobre as regiões de memória mapeadas por parte do *driver*, o que é requerido para atender às especificações V4L2. Uma abordagem melhor é prover um descritor de arquivo no mesmo *driver*, de modo a ter-se um controle maior sobre os mapeamentos de memória.

Para evitar que o usuário necessite abrir dois descritores de arquivos (um para as chamadas de tempo real e outro para fazer o mapeamento), pode-se abrir um descritor de arquivos para o mapeamento no próprio *driver*, tornando isto transparente para o usuário. Este, por sua vez, requisitará o mapeamento por meio de uma chamada tipo *ioctl*, que ocorrerá, excepcionalmente, no domínio secundário, isto é, sem restrições de tempo real. Por isso, tal chamada deve ser realizada durante a inicialização do programa. Embora esse mapeamento não execute em um contexto de tempo-real, a utilização dessa área de memória pelo programa do usuário ocorrerá em tempo real.

Uma pequena modificação será necessária à API V4L2, no modo de mapear a memória através de uma chamada *ioctl* em vez de uma chamada *mmap*. Funções como *select* e *poll* seguem a mesma abordagem. Assim, serão definidas chamadas *ioctl* padrão para essas funções. São elas RTDM_MMAP, RTDM_MUNMAP, RTDM_SELECT e RTDM_POLL. Ver detalhes da definição no cabeçalho apresentado no Apêndice B.

A idéia geral pode ser resumida da seguinte forma:

1. Na inicialização, o usuário solicita um número de *buffers* de memória, os quais conterão as imagens, através da chamada VIDIOC_REQBUFS;
2. O *driver* aloca memória ou reserva [1] memória pré-alocada para o usuário;
3. O usuário solicita o mapeamento de tais memórias para o espaço do usuário, uma de cada vez, através de chamadas RT_MMAP;
4. O usuário coloca os *buffers* alocados na fila através de chamadas VIDIOC_QBUF e então chama VIDIOC_STREAMON para começar a capturar;
5. O *driver* preenche os *buffers* na fila com dados obtidos da câmera e os coloca na fila de saída;
6. O usuário finalmente retira os *buffers* da fila de saída, através de VIDIOC_DQBUF, usa-os e os coloca novamente na fila de entrada. Este processo se repete indefinidamente até que a aplicação resolva parar de processar imagens. Nesse momento o usuário chama VIDIOC_STREAMOFF e libera a memória através de chamadas RT_MUNMAP para cada buffer mapeado e finalmente realizando outra chamada VIDIOC_REQBUFS, passando o parâmetro 0 (zero) como número de *buffers*.

[1] Uma vez que memórias DMA em dispositivos PCI devem ser contínuas em arquiteturas tipo x86, o usuário pode passar o parâmetro “mem=100M” para o *kernel* do Linux para reservar 28 MB em um sistema com 128 MB de RAM. O *driver* pode, então, utilizar essa memória para realizar transferências DMA.

4.6.2 A Implementação do *Driver*

A interface definida para programação de câmeras no Linux em tempo-real, com base no Xenomai, foi definida como RT_V4L2. A próxima etapa foi implementar o *driver* da câmera.

O *driver* implementado utiliza uma fila de *buffers* circular com tamanho estático, que pode ser mudado em tempo de compilação. Decidiu-se utilizar um tamanho estático uma vez que a inicialização dos *buffers* de imagens são normalmente definidos na inicialização, onde já se sabem quantos *buffers* são necessários. Implementar a fila estaticamente é mais simples e atende bem ao caso de uso que se escolheu para validar o framework, como se pode ver no capítulo 5. Outros *drivers*, contudo, são livres para implementar a fila dinamicamente, já que o skin *RTDM* suporta alocação dinâmica de memória em contexto de tempo-real.

A idéia de funcionamento do *driver* é ilustrada na figura 4.3. Na inicialização do Linux, é passado o parâmetro "mem=510M" para reservarmos 2 MB para uso pelo *driver*. O usuário deve informar qual o endereço de memória pode ser utilizado pelo *driver* e qual o tamanho que pode ser utilizado pelo *driver*. Se tais parâmetros forem omitidos, será utilizada a última posição de memória usada pelo Linux (*high_mem*) e um tamanho de 2 MB. Estes parâmetros permitem que múltiplos *drivers* utilizando a técnica de reservar memória na inicialização possam operar em harmonia.

Quando o usuário realiza uma chamada VIDIOC_REQBUFS, a memória reservada é mapeada pelo *driver*, retornando o número real de *buffers* alocados. Este número nunca passará do limite de memória informado durante o carregamento do *driver*. O usuário então enfileira os *buffers* através de VIDIOC_QBUF. O procedimento então verifica se o processo de aquisição já foi iniciado com VIDIOC_STREAMON. Caso ainda não tenha sido iniciado, as aquisições só ocorrerão na chamada VIDIOC_STREAMON, caso a fila de entrada não esteja vazia. Caso contrário, verifica-se se há uma aquisição em andamento. Se houver, VIDIOC_QBUF não toma qualquer atitude, pois outra rotina se encarregará de realizar a nova aquisição quando a corrente acabar. Caso nenhuma aquisição esteja em andamento, uma nova aquisição é requisitada ao *frame-grabber*.

Quando a aquisição é concluída, uma interrupção ocorre. A rotina de tratamento de interrupção transfere o buffer preenchido para a fila de saída e requisita uma nova aquisição se a fila de entrada não estiver vazia. Para retirar o *buffer* da fila de saída, existem dois modos de operação, de acordo com a API V4L2. No modo bloqueador, a chamada VIDIOC_DQBUF é bloqueada se a fila de saída estiver vazia, até que algum *buffer* seja disponibilizado. No outro modo, a rotina retorna EAGAIN imediatamente, informando que a fila está vazia. O modo é especificado na chamada *open()*. No modo bloqueador, a chamada espera por um

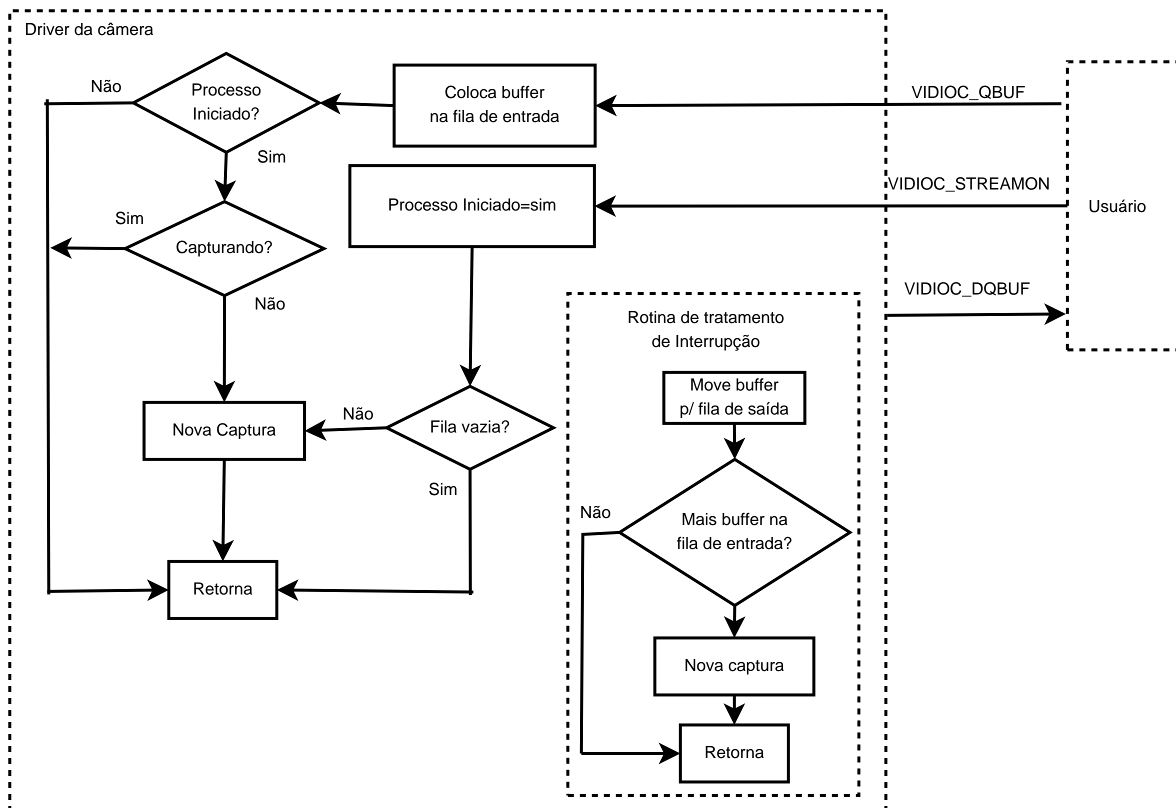


Figura 4.3: Diagrama de funcionamento do *driver*.

evento enviado pela rotina de interrupção sempre que uma aquisição termina.

As operações sobre a fila envolvem a atualizações de alguns ponteiros e índices. Tais operações devem ser realizadas atomicamente para não permitir que dados incompletos sejam obtidos por alguma rotina. Tais procedimentos foram, então, protegidos por meio de regiões exclusivas ou *mutex*. Os três códigos que utilizam as filas de entrada e saída são os relacionados às chamadas tipo *ioctl* VIDIOC_QBUF, VIDIOC_DQBUF e a rotina de interrupção. Os três códigos relevantes estão disponíveis no Apêndice C.

4.7 Conclusão

Este capítulo propôs, de modo sistemático, um novo modelo de *framework* para programação de robôs móveis, bastante flexível e eficiente, utilizando a linguagem C++ e a ferramenta de documentação de código *Doxygen* para gerar sua API (um exemplo de documentação está disponível no anexo E). O sistema pode ser configurado para ter um *footprint* pequeno, e pode ser expandido para outros tipos de robôs, sem complicações.

Com exceção da classe *SimpleRobot*, que realiza a comunicação com a porta paralela de

um PC, de um modo não portável, todo o código do *framework* é portável e funciona com qualquer outra arquitetura de processadores. A comunicação com a porta paralela também pode ser portada, definindo um conjunto de chamadas *IOCTL*, como foi feito para o *driver* de captura. Portanto, embora não implementado nesse trabalho, basta seguir a metodologia, mostrada neste capítulo, utilizada na implementação do *driver* de captura de imagens, para implementar uma API portável para comunicação com a porta paralela e alterar as seções correspondentes na implementação de *SimpleRobot*, para que esta classe possa ser utilizada com outras arquiteturas de computadores.

A API desenvolvida neste capítulo é muito simples e intuitiva, além de permitir uma implementação eficiente das classes. A curva de aprendizagem, portanto, é bem rápida, quando comparada a projetos similares. E o código final possui poucas dependências, o que facilita sua distribuição. Além disso, é suficientemente eficiente para atender os requisitos de aplicações de robótica móvel.

Capítulo 5

Experimentos

No *framework* proposto, há dois modos de programação: baseado em ajustes diretos de velocidade do robô, e baseado em ações ou comportamentos. Dois experimentos foram, então, desenvolvidos para explorar cada um dos modos de programação. Para demonstrar o uso do driver da câmera, um dos experimentos realiza um controle do robô baseado em processamento de imagens. No experimento 1, foi utilizada a abordagem de controle direto de velocidade, utilizando o método *Robot::SetVel()*. No experimento 2, utilizou-se o gerenciador de ações padrão do *framework* para demonstrar sua utilização. O controle do primeiro experimento foi realizado usando realimentação visual, por meio do *driver* desenvolvido, enquanto que o segundo experimento possui somente a realimentação dos *encoders* acoplados à roda, demonstrando o funcionamento do sistema de hodometria. Ambos os experimentos respeitaram as restrições de tempo impostas pelo sistema e nenhum *deadline* foi ultrapassado, em todos os testes realizados, para as tarefas de tempo-real, independentemente da carga de processamento no Linux, como já era esperado.

Para verificar se algum *deadline* foi ultrapassado, é possível utilizar duas abordagens. Uma é sobrescrever o método virtual *OnOverruns()*, da classe *Task*. Outra alternativa é verificar os registros gravados por *Xenomai* no sistema de arquivos *procfs*, do Linux padrão, os quais podem ser consultados a qualquer instante, indicando várias informações úteis durante a fase de desenvolvimento de sistemas de tempo real. Tais registros ficam localizados, normalmente, no endereço "/proc/xenomai".

Em todos os experimentos, por simplicidade, foi utilizado um computador externo ao robô, equipado com um processador Pentium 4 1.7 GHz e 512 MB de RAM. Contudo, o mesmo processamento seria possível, com menos memória, além disso muitas opções de placas tipo SBC com esses recursos estão disponíveis no mercado e poderiam ser utilizadas com o robô desenvolvido. Porém, tal adaptação foge ao escopo desse trabalho. A comunicação com o robô se deu através de um cabo *flat* conectando a porta paralela do computador

à placa de controle dos atuadores no robô. O computador gerava os sinais PWM de controle para o robô e lia as informações dos *encoders*, a baixo nível, diretamente, através da porta paralela. Nenhum processamento foi utilizado a bordo do robô.

5.1 O Experimento 1

O primeiro experimento tem como objetivo demonstrar:

- o funcionamento do driver;
- o controle do robô pelo método mais direto, ajustando as velocidades linear e angular do robô;
- a simples integração com o sistema Linux;
- o funcionamento do robô;
- o uso do *framework*;
- e que as restrições de tempo impostas foram atendidas.

O experimento, ilustrado na figura 5.1, é bem simples. A câmera, posicionada no teto do laboratório, deve identificar a posição e orientação do robô, relativa à posição do alvo. Então, foi utilizado um controle puramente proporcional ao erro na orientação relativa do robô ao alvo para que o robô atinja seu objetivo. A velocidade linear foi mantida constante, enquanto a velocidade angular era proporcional ao erro de orientação, e no sentido de corrigir o erro, utilizando técnicas convencionais de Controle.

Para a identificação do robô e do alvo, o processamento utilizado foi bem simples, utilizando métodos comuns de identificação de cores. Cada *pixel* da imagem capturada, no formato BGR0 (azul-verde-vermelho-nulo), foi convertido para o equivalente de matiz de cor (*hue*), no plano HSV, de modo que variações no brilho e saturação não impedem a identificação correta da cor. Em cima do robô foram colocados dois rótulos lado a lado, um amarelo e um vermelho. Convencionou-se que o amarelo indicaria a frente do robô. A posição do robô foi calculada como a média entre os centros de massa do rótulo amarelo e do rótulo vermelho. O chão era azul e o alvo era um retângulo verde.

O método empregado para o reconhecimento de cada uma das cores foi o mesmo. Primeiramente, fez-se uma binarização da imagem, baseado na faixa de matiz que cada papel

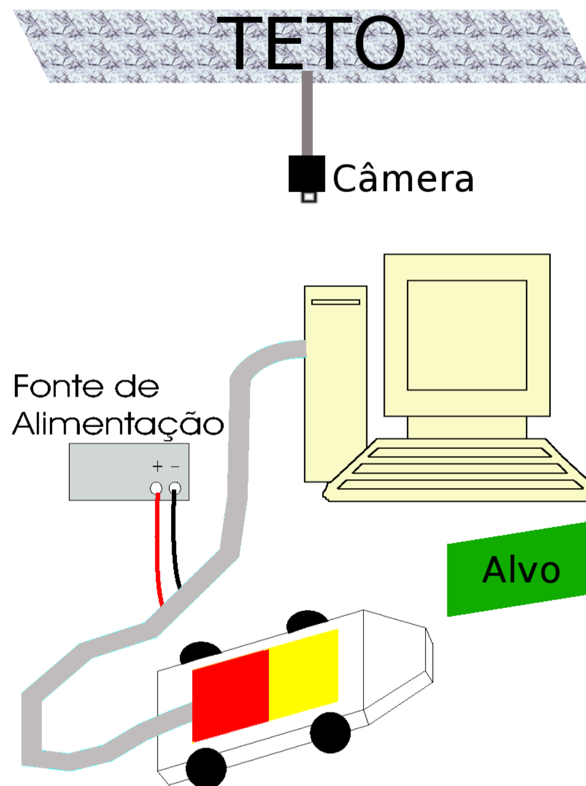


Figura 5.1: Arranjo utilizado para a execução do experimento 1.

ocupava. Um processo de erosão binária foi realizado para o alvo verde, para retirar ruído, já que seu matiz se confunde com algumas partes do chão azul.

Para a captura de imagens, foram utilizados dois *buffers* de armazenamento da imagem. Dessa forma, um *buffer* pode ser preenchido, enquanto o outro está sendo processado. Isso não era necessário nessa aplicação. Poderia ter sido usado um *buffer* simplesmente, uma vez que o processamento se dá em menos de 20 ms, o menor laço utilizado foi de 100 ms, e o tempo de captura de um quadro é de aproximadamente 17 ms. Apenas um dos dois quadros que compõem o *frame* foi utilizado, gerando uma imagem de 640x240 *pixels* não quadrados. Isto é, a imagem processada é deformada, mas é suficiente para seu processamento. Portanto, se essa fosse a única tarefa de tempo-real em execução, seria possível processar todo quadro par (ou ímpar) na freqüência de 60 Hz da câmera NTSC. Então, a fim de ilustração, a implementação utiliza dois buffers para demonstrar a idéia. A requisição de captura é praticamente instantânea e deveria ser realizada antes de algum processamento para que se possa processar enquanto o *frame-grabber* captura a próxima imagem a ser processada e tirar vantagem desse *design*. Não é o que acontece nessa implementação por não haver necessidade, mas a idéia está clara.

O laço de controle foi executado a uma taxa de 3 quadros por segundo, guiando o robô até seu alvo perfeitamente, sobrando uma parcela razoável de poder de processamento para

que outras tarefas pudessem ser executadas concorrentemente. Contudo, a fim de demonstrar a interação com o Linux, uma outra tarefa de tempo-real, com a prioridade mais baixa, foi criada para gerar uma seqüência de arquivos com as imagens processadas, para que se pudesse gerar um vídeo, posteriormente, com a trajetória real do robô. Para uma boa visualização do vídeo, o período do laço de controle foi reduzido para 100 ms, gerando um vídeo com 10 quadros por segundo (10 FPS), que possui uma visualização melhor do que apenas 3 quadros por segundo. Cada quadro do vídeo é uma imagem real que foi processada pelo laço de controle.

Outra forma de demonstrar a interação com o Linux foi o desenvolvimento de uma interface gráfica para exibir a posição corrente do robô e do alvo durante os experimentos. Evidentemente, essa interface gráfica não possuía quaisquer garantias de tempo e poderia não ser atualizada na taxa desejada se a carga do sistema estiver muito grande. Porém, em situações normais, as posições do robô e do alvo eram atualizados a uma taxa fixa. É importante frisar que mesmo sob condições de carga pesada de processamento do sistema, embora a interface gráfica pudesse parar de responder por um tempo, o controle do robô não era afetado de modo algum. A figura 5.2 mostra a interface gráfica em um dado instante do experimento, enquanto a figura 5.3 exibe a trajetória obtida no experimento.

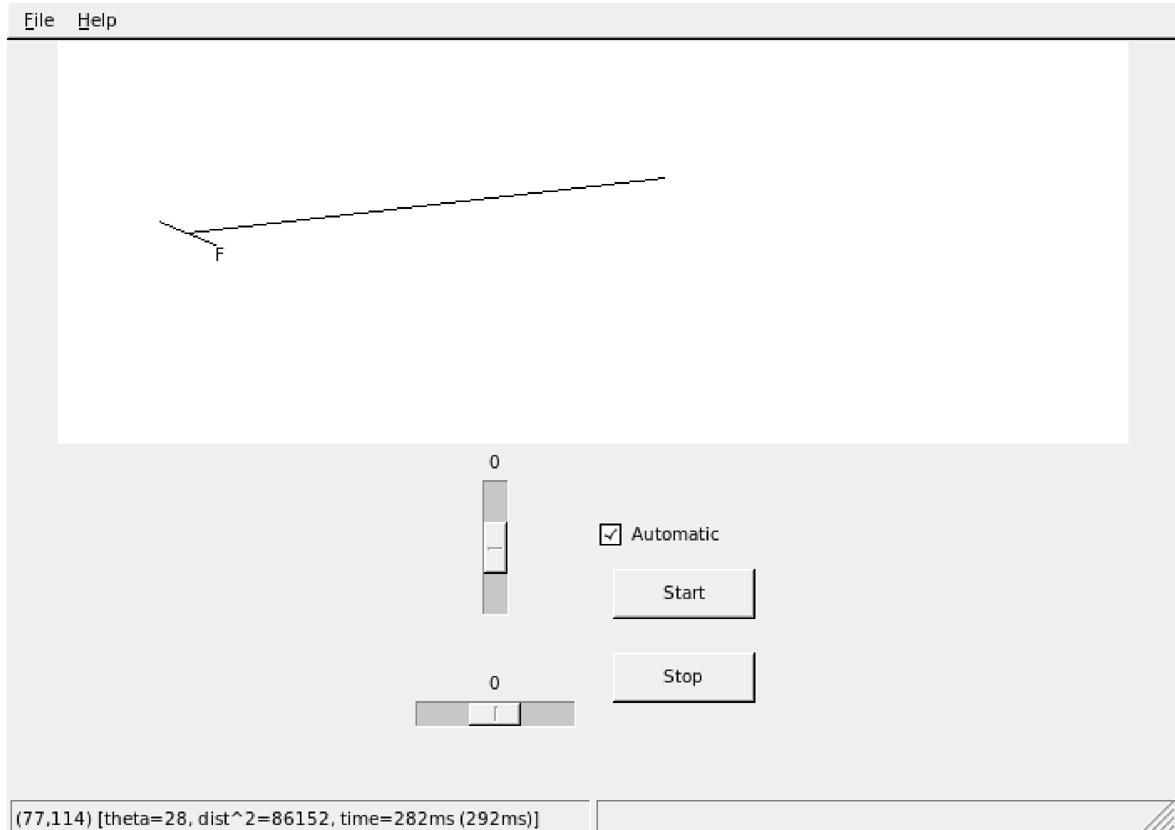


Figura 5.2: A interface gráfica durante a execução do experimento 1.

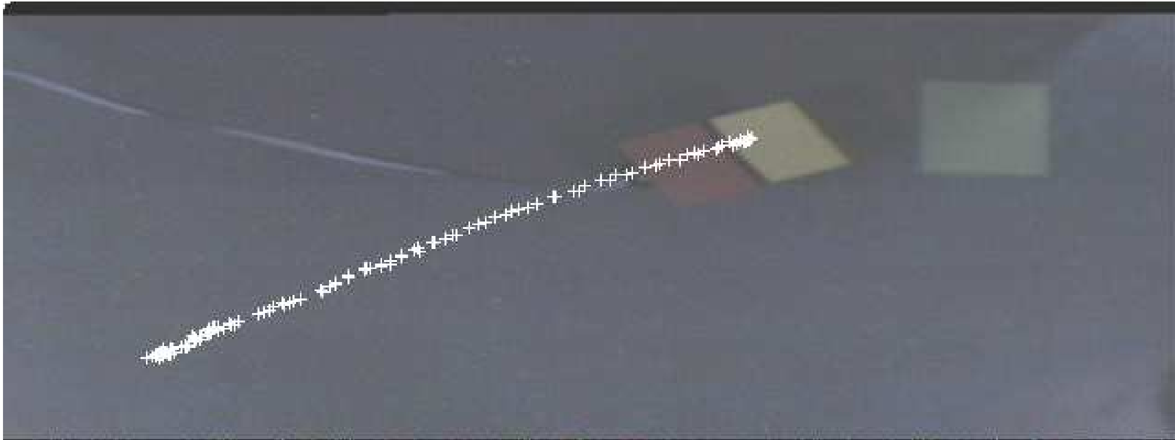


Figura 5.3: A trajetória percorrida pelo robô no experimento 1.

Esse é um sistema de tempo-real típico, possuindo tarefas com restrições críticas de tempo (*hard real-time*) e tarefas com restrições menos severas (*soft real-time*) para a geração dos arquivos de imagem e interface gráfica. O controle em si do robô é garantido de funcionar e ter suas restrições de tempo atendidas. As restrições de tempo para a geração dos arquivos contendo imagens, contudo, são importantes mas não precisam ser atendidas. Nos experimentos realizados todas as restrições de tempo foram atendidas, incluindo tarefas *soft real-time*, contudo, não se pode garantir que nenhum frame será perdido na geração dos arquivos em todos os testes no futuro e que a tela da interface gráfica será constantemente atualizada. Mas ocorrências desse tipo não prejudicam o controle do sistema. Simplesmente o vídeo final poderia ter um ou mais frames perdidos ou o usuário poderia perceber algum incômodo visual ao observar a interface gráfica.

As informações dos *encoders* foram processadas com uma taxa de amostragem de 100 micro-segundos. O período do sinal PWM utilizado para controlar os motores foi 10 ms, por se mostrar suficiente para um bom controle, como percebido experimentalmente. Para o controle de velocidade das rodas, foi utilizado um controlador proporcional-integrador (PI), para cada roda, baseado nas informações dos *encoders*, como ilustrado na figura 5.4.

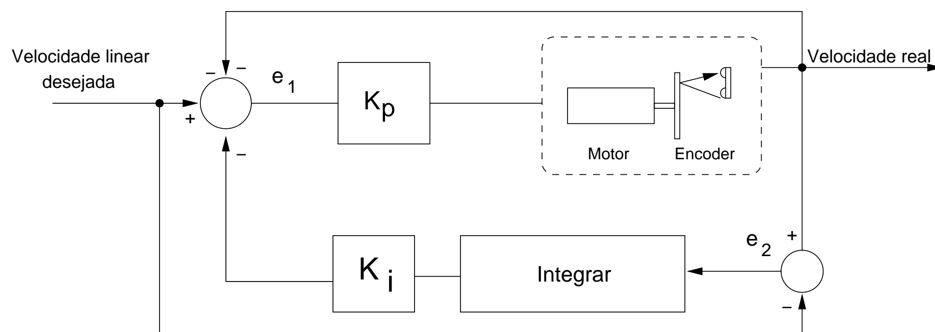


Figura 5.4: Representação do controlador PI utilizado em cada roda.

Foram atendidas as restrições de tempo, em todos os testes, para todas as tarefas de tempo real, como geração dos sinais PWM para cada motor, a leitura dos 4 *encoders*, e as tarefas de controle de velocidade, além da tarefa principal de controle.

O código no Apêndice D ilustra as partes relevantes do programa de controle. Foram omitidos os códigos relacionados à construção da interface gráfica, cabeçalhos e tratamento de erros por simplicidade. Inicialmente foram declaradas as variáveis, estaticamente, para o processamento de imagens. Toda a memória necessária para o processamento é alocada durante o carregamento da aplicação, antes da chamada de *mlockall()*, que impede que o programa seja copiado para o disco, através do processo de *cache* de memória virtual.

Uma tarefa periódica foi criada para realizar o processamento de imagens e ajustar as velocidades linear e angular do robô a cada iteração. A tarefa foi construída de modo bem simples, como a instância de uma nova classe, derivada de *PeriodicTask*, onde a prioridade e período foram passados ao construtor de *PeriodicTask*. O método *Init()* foi sobrescrito para abrir um descritor de arquivo do *skin* RTDM para se iniciar a comunicação com o driver da câmera. Simetricamente, o método *Destroy()* foi sobrescrito para encerrar a comunicação e liberar a câmera para outra aplicação.

O método puramente virtual *DoCycle()* foi implementado para realizar a tarefa periódica em si, calculando a cada iteração a posição e orientação do robô e do alvo e ajustando a velocidade angular do robô utilizando um controlador Proporcional simples, mantendo a velocidade linear fixa. Vale ressaltar que existem técnicas melhores de controle mas elas fogem ao escopo deste trabalho.

Uma instância de *SimpleRobot* é declarada estaticamente. Na inicialização da aplicação gráfica, a função rt_main() é chamada, a qual inicializa a tarefa de tempo real criada, além de executar o método *Start()* de *SimpleRobot*, criando as tarefas necessárias para o funcionamento do robô. A interface gráfica exibe, periodicamente, um desenho ilustrando a situação atual, ou seja, as posições do robô e do alvo, além da orientação do robô. A comunicação entre as tarefas de tempo-real e a interface gráfica deu-se por meio de memória compartilhada, através de algumas variáveis globais.

Enfim, todo o esforço da aplicação concentrou-se no desenvolvimento da interface gráfica e da tarefa de controle em si. Todo o controle interno de velocidade das rodas do robô, além do modelo de equações para robôs tipo par diferencial foram omitidos do programa principal e foram implementados internamente na biblioteca oferecida pelo *framework* desenvolvido, permitindo um alto nível de abstração na execução da tarefa. O modelo de programação para captura de imagens também mostrou-se bem versátil e poderoso, tornando possível garantir a correta operação do controle do robô, independente da demanda de recursos pelo Linux.

5.2 O Experimento 2

O segundo experimento tem como objetivo demonstrar, adicionalmente, o uso do gerenciador de ações e do sistema de hodometria, que provê uma estimativa da posição do robô, ao longo do tempo, utilizando somente as informações obtidas dos sensores de infravermelho que compõem os *encoders*.

O objetivo do robô foi locomover-se até uma posição específica, em coordenadas (x,y) do plano do robô (o chão). Para tanto, foram utilizadas duas ações, já citadas no capítulo 4: *AcGotoXY*, e *AcSetLimits*. A primeira se encarrega de levar o robô à posição desejada, enquanto que a segunda limita a velocidade angular do robô, a fim de diminuir erros de hodometria, devido a deslizamento das rodas.

Novamente, todos os prazos das tarefas foram respeitados, como previsto, e o robô atingiu seu objetivo em todos os testes. É difícil estimar exatamente o erro da hodometria porque as medições externas da posição e orientação do robô são difíceis. A tabela 5.1 exibe o erro obtido em cada experimento. Há uma precisão pequena na estimativa do ângulo inicial, podendo variar até 5°, o que equivale a erros de posição tanto maiores quanto maior for a distância que o robô percorre.

Apesar da grande imprecisão nas medidas, são informados alguns dados aproximados sobre os erros obtidos em alguns experimentos realizados. Considerando que o robô tenha andado em linha reta (o que parece ocorrer, visualmente), a tabela mostra qual seria o erro nessa condição. A última coluna é exibida na forma *A*/*B*. *A* é o erro, em módulo, considerando que a medição da orientação inicial é válida. *B* é o erro, considerando que o robô andou em linha reta. Todos os valores na tabela estão em milímetros. Os erros são medidos entre a posição alcançada e a estimada para enfocar no sistema de hodometria.

Exp.	Pos. desejada	Pos. alcançada	Pos. estimada*	Erros
1	(1200, 0)	(1160, 60)	(1227,1)	89/65
2	(1500, 0)	(1500, 80)	(1521,3)	80/19
3	(900, 300)	(910, 280)	(922,305)	28/19
4	(1200, -600)	(1150, -690)	(1211,-603)	106/12
4	(300, 300)	(310, 290)	(327,325)	39/37

* Pelo sistema de hodometria do robô.

Tabela 5.1: Resultados obtidos para o segundo experimento.

O código a seguir ilustra a facilidade de programação de tal sistema, obtendo proveito de reuso de código (o *framework* em si).

```
using namespace AM_Basic;

INIT_FRAMEWORK()

SimpleRobot robot;

/* (x,y) = 1000 [mm] ou 1 m para frente e 100 [mm] ou 10 cm
  para a esquerda
vel = 60 mm/s ou 6cm/s
Erro máximo permitido: 3cm (30mm) */
AcGotoXY action(&robot, 1000, 100, 60, 30);

/* Limita a velocidade angular em 5 graus por segundo em modulo.
Os primeiros parâmetros da ação são limitações de velocidade
linear e angular */
AcSetLimits ac_limits(&robot, 0, 0, 5*M_PI/180, -5*M_PI/180);

// Utilizamos o gerenciador de ações padrão.
AMBasic am(&robot, 0.1); /* ciclo de tomada de decisões: 100 ms */

int main() {
        am.AddAction(&action);
        am.AddAction(&ac_limits);
        robot.Init();
        robot.Start();
        // Espera uma tecla ser pressionada para sair do programa
        getchar();
        robot.Stop();
        /* informa ao usuário a posição corrente do robô,
        baseado na hodometria */
        DistanceType x, y;
        robot.GetPos(x,y);
        cout << "(x,y): ("<< x << "," << y << ") Theta = " <<
          robot.GetTheta() << endl;
        return 0;
}
```

Esse é o código completo da aplicação. Ele mostra como a programação de um sistema robótico completo pode ser simplificada através de uma estrutura de programação de alto nível. O *framework* proposto permite executar tarefas com certa complexidade com pou-

cas linhas de código, permitindo ao programador concentrar-se nas tarefas de alto nível. A grande modularidade e flexibilidade do *framework* permite que a programação em baixo e alto nível sejam realizadas de modo independente. Assim, cada programador pode se concentrar em uma parte diferente da programação de um sistema robótico móvel, de modo cooperativo. Alguns aperfeiçoarão a estrutura interna do *framework*, enquanto outros desenvolverão aplicações finais, utilizando a estrutura provida pelo *framework*.

5.3 Conclusão

O primeiro experimento validou a implementação do *driver* de captura de imagens, além de demonstrar a facilidade de programação do *framework*, através de uma API simples. Também mostrou que todas as restrições temporais para gerar os sinais PWM, bem como capturar os sinais dos sensores, foram atendidas, sem exceção, em todos os testes realizados. Além disso, exemplificou o uso do modelo de programação através do controle direto de velocidade do robô.

Adicionalmente, o segundo experimento mostrou que, com poucas linhas de código, bastante intuitivas, foi possível controlar o robô para percorrer uma determinada distância, utilizando a abordagem baseada em comportamento ou ações, de modo satisfatório, em todos os testes realizados.

Esses dois experimentos são suficientes para ilustrar que é possível implementar sistemas de controle mais sofisticados, mostrando que é possível controlar o robô tanto em alto nível quanto em baixo nível. Além disso, esses experimentos mostram que um *driver* pode ser desenvolvido para suportar placas mais complexas, como *framegrabber*, placas de rede ou qualquer outra, garantindo que restrições temporais sejam atendidas para praticamente qualquer operação com requisitos de tempo-real, uma vez que o *hardware* seja suficiente para atender às restrições temporais.

Utilizando o *framework* proposto, sistemas robóticos podem ser mais facilmente elaborados, uma vez que as funções básicas já estão implementadas e o desenvolvedor pode se concentrar em uma programação de mais alto nível.

Capítulo 6

Conclusão e Trabalhos Futuros

Neste trabalho, foi desenvolvido um *framework*, que atinge as características fundamentais necessárias para a elaboração de aplicações de robótica móvel, conforme proposto no capítulo 1. Além destas, ainda foram alcançadas mais algumas características desejáveis, dentre as citadas na seção 2.1. As características alcançadas neste trabalho são:

- desempenho em tempo real;
- flexibilidade;
- reuso de código;
- facilidade de uso;
- eficiência;
- fácil manutenção;
- boa documentação;
- código aberto;
- suporte a múltiplas arquiteturas de processadores (aquelas suportados pelo projeto Xenomai).

Outras características podem ser alcançadas em trabalhos futuros. Algumas sugestões estão listadas no final deste capítulo.

Vale ressaltar a grande facilidade de programação permitida pelo *framework*, como se pôde observar nos experimentos mostrados no capítulo 5. Aplicações com certa complexidade foram desenvolvidas com bem menos esforço do que seria necessário na ausência do

framework. Em relação a projetos semelhantes já existentes, pode-se perceber uma simplicidade muito maior para a programação de sistemas robóticos, além de melhor versatilidade e boa eficiência no código binário final. Houve também um cuidado maior para documentar todas as classes implementadas, além de um código limpo, bem documentado e de fácil entendimento. Isto facilita a programação de aplicações que utilizem o *framework*, bem como a continuidade do mesmo, permitindo que novas classes de robôs sejam facilmente acrescentadas, além de facilitar o aprimoramento das classes já existentes.

Vantagens em relação a outros projetos incluem grande flexibilidade em vários aspectos:

1. tipos de dados e unidades escolhidos pelo usuário;
2. a arquitetura de controle não é fixa, e é definida pelo usuário;
3. suporta programação por ações e não limita o uso de um único tipo de gerenciador de ações, mas oferece suporte para que qualquer gerenciador de ações possa ser acrescentado ao *framework*, embora forneça um gerenciador padrão para facilitar o projeto de aplicações pouco dependentes do tipo de gerenciador de ações.

Além da grande flexibilidade, este projeto provê outras vantagens:

1. interface mais intuitiva e simples;
2. código mais eficiente, permitindo que sistemas de tempo-real rápidos possam ser implementadas mais facilmente, enquanto vários dos outros projetos sequer suportam programação em tempo-real;
3. único utilizando o Xenomai, que se mostrou um sistema operacional mais flexível, fácil de ser mantido e que provê melhor suporte a construções de *drivers* em tempo-real;
4. provê uma interface para captura de vídeo em tempo-real;
5. documentação mais clara e suscinta.

Para permitir a continuidade desse trabalho, o *framework*, que recebeu o nome *Real-time Mobile Robotic Framework* – RTMRF, é disponibilizado em `http://www.ele.ufes.br/~rosenfeld/`.

Outras características desejáveis, que tornariam o *framework* desenvolvido ainda mais interessante, podem ser alcançadas em trabalhos futuros, como:

- a construção de um simulador;
- suporte a robôs populares, como o *Pioneer* da *ActivMedia* e o *Rug Warrior*;
- uma biblioteca para processamento de imagens de modo determinístico;
- outros drivers de câmera utilizando a API sugerida no capítulo 4.6;
- suporte a outros periféricos, como ultra-som, *laser* e GPS;
- outros tipos de gerenciadores de ações ou de resolvedores a serem utilizados pelo gerenciador padrão *AMBasic*;
- embutir robustez ao *framework*, tratando falhas que possam vir a acontecer;
- implementar um sistema de comunicação em tempo real para atividades de comunicação explícita entre robôs cooperativos.

Enfim, *frameworks* destinados a robôs móveis possuem um longo campo de desenvolvimento a ser explorado. Esse trabalho dá um passo a mais, através de uma comparação entre os *frameworks* existentes, a identificação de suas falhas, a apresentação de critérios de avaliação para *frameworks* robóticos, e o desenvolvimento sistemático, a partir desses critérios, para a construção dessa nova proposta, a qual difere, em muitos aspectos, do estado da arte atual, atendendo a vários dos critérios mencionados. E não menos importante: não impedindo que novas contribuições supram as características desejadas que ainda não foram implementadas.

Apêndice A

Aspectos Construtivos do Robô para Testes Desenvolvido

O robô projetado consiste de uma estrutura de 20x25 cm construída com acrílico de 1 cm de espessura, capaz de suportar bem os impactos sofridos durante a fase de teste. Ele possui dois motores CC com redução em configuração tipo par diferencial. A alimentação dos motores é de 12 V e, com as rodas de 8 cm de diâmetro, podem atingir uma velocidade linear máxima de cerca de 10 cm/s. A alimentação elétrica, bem como a comunicação com o computador utilizam um cabo flat com 14 fios como meio físico, para reduzir a complexidade de se instalar e recarregar uma bateria, assim como do circuito necessário para comunicação sem fio e dos programas necessários para realizar tal comunicação em tempo real.

A interface utilizada para comunicação com o PC é a porta paralela, por possuir diversos sinais paralelos e que, por isso, podem ser tratados sem a necessidade de se utilizar qualquer protocolo de comunicação e, principalmente, a facilidade de se garantir seu funcionamento em tempo real. Em condições normais, com um cabo de até 2 metros, a porta paralela deve funcionar até 2 Mbps, ou equivalentemente, 262,1 kHz por pino. No laboratório, experimentalmente, utilizando *buffers* da família 74XX244 apenas no terminal conectado ao robô, foi observado o mesmo comportamento com um cabo de cerca de 10 metros. Também foi verificado que cada fio do cabo flat utilizado é capaz de suportar uma corrente de pelo menos 3 A. Todo o circuito do motor, mais os dois motores em velocidade máxima não consumiram juntos mais do que 1 A, a não ser por curtíssimo tempo durante a partida. Utilizou-se, portanto, apenas um par de fios para a alimentação do motor e uma referência (terra) extra para a comunicação com a porta paralela devido à queda de tensão sobre o cabo.

Utilizou-se um regulador de tensão de 12 V na placa do robô para compensar as quedas de tensão sobre o cabo, que variam com a carga dos motores. O regulador de 12 V, por sua

vez, alimenta os motores e outro regulador de 5 V, o qual alimenta o circuito de lógica do robô. Como a corrente máxima requerida pelo robô não supera 1A, utilizou-se os bastante conhecidos 7812 (com dissipador) e 7805, cuja limitação de corrente é de 1 A.

O circuito consiste basicamente de duas pontes H com controle por PWM (L298) para alimentação dos motores, 4 pares emissor/receptor de infravermelho para os dois *encoders* óticos em quadratura, um comparador com histerese (LM339) para cada par ótico e dois *buffers* (74HCT244) para realizar a interface com a porta paralela, como mostra o diagrama em blocos da figura A.1.

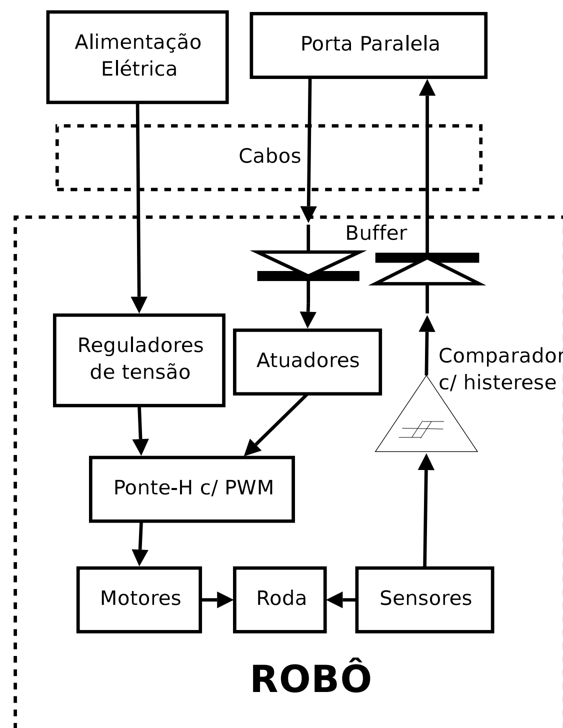
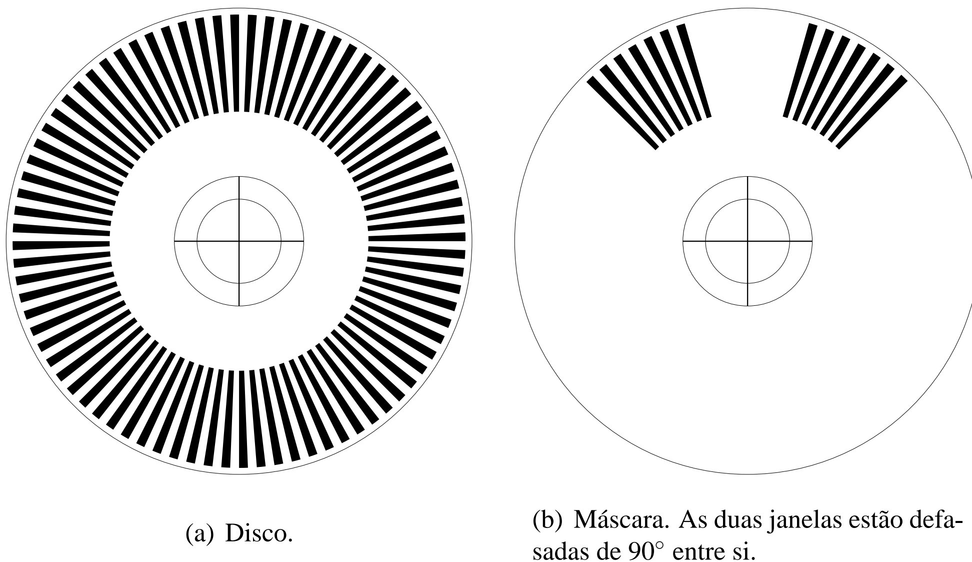


Figura A.1: Representação da montagem do circuito

Para realizar o controle de velocidade das rodas e programar o sistema de hodometria, foram utilizados *encoders* óticos baseados em reflexão, produzidos com pares de emissor/receptor de infravermelho utilizados em controles remotos de eletrodomésticos. Uma máscara circular furada, como mostrada na figura A.2(a), está fixa no robô. Uma abordagem mais completa sobre sistemas de hodometria e sensores é mostrada em (BORENSTEIN; EVERETT; FENG, 1996). As janelas estão posicionadas de modo que os sinais capturados

pelos dois sensores estejam em quadratura, ou seja, defasados de 90° entre si. Isto permite conhecer o sentido do rotação da roda. Fixada em cada uma das rodas está um material (foi utilizado papel neste projeto) conforme a figura A.2(b), de modo que a parte preta reflita pouco o raio infravermelho emitido e a parte branca reflita bem este mesmo raio. O disco e a máscara foram desenvolvidos diretamente na linguagem Postscript, garantindo uma excelente resolução quando impresso em uma impressora Postscript. Além disso, é fácil mudar o número de ranhuras, o raio do disco entre outras características, mudando-se as variáveis definidas no documento Postscript desenvolvido.



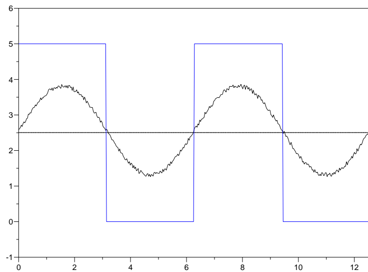
(a) Disco.

(b) Máscara. As duas janelas estão defasadas de 90° entre si.

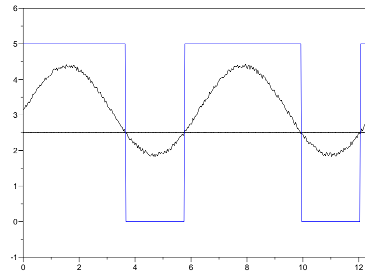
Figura A.2: Disco utilizado no encoder e sua respectiva máscara

Os discos possuem 160 ranhuras no total (80 brancas e 80 pretas). Como é muito complicado obter uma onda quadrada para uma velocidade constante, por motivos que serão explicados mais adiante, são utilizadas apenas transições de subida (ou descida), o que equivale a 80 transições por revolução da roda, ou equivalentemente, cerca de 3 mm de deslocamento linear por transição. Se a onda fosse quadrada a uma velocidade constante, poder-se-ia conseguir o dobro da resolução. Para que isto aconteça, é necessário que o valor do nível de tensão da comparação seja igual ao valor médio da forma de onda na entrada do comparador. As figuras A.3(a) e A.3(b) ilustram as situações em que o nível de comparação equivale ao nível médio do sinal e quando diferem entre si. Para se obter essa condição, é necessário um arranjo mecânico preciso para regular a distância entre os sensores e a roda, ou um circuito eletrônico para deslocar a forma de onda de modo que seu valor médio coincida com o nível de comparação. Como o robô montado não dispõe de qualquer destes recursos e já que a resolução obtida já é suficiente para realizar os testes de validação, estão sendo utilizados apenas as transições de subida (ou descida) para a estimativa da velocidade, o que resulta em um pulso a cada 4,5° de revolução. É pouco vantajoso investir em uma resolução

muito maior que esta, pois o erro ocasionado por deslizamento das rodas iria mascarar a alta precisão obtida pelo encoder, e a tarefa de medição de velocidade necessitaria de mais processamento, pois o período de *polling* seria menor. Não é possível realizar a medida por interrupção porque apenas um dos pinos da porta paralela do PC gera interrupções.



(a) Nível de comparação equivale ao nível médio do sinal do encoder



(b) Nível de comparação difere do nível médio do sinal do encoder

Figura A.3: Sinais de saída para diferentes níveis de comparação

A montagem final do robô pode ser vista na figura A.4.

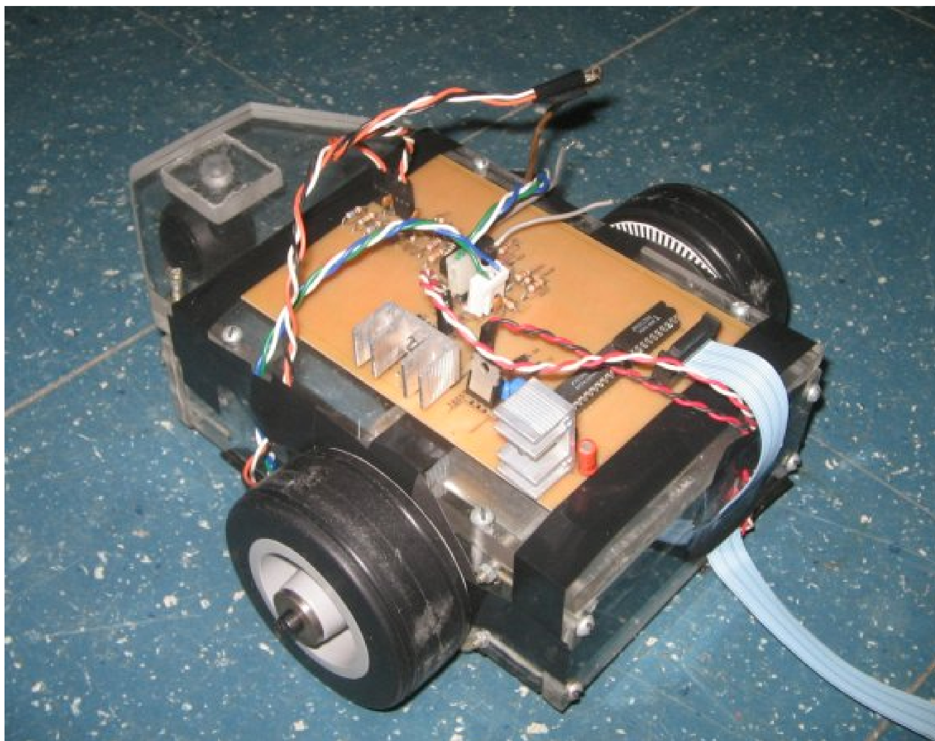


Figura A.4: Foto do robô construído.

Apêndice B

rt_mmap.h

```
#ifndef RT_MMAP_H
#define RT_MMAP_H

#include <linux/types.h>

struct rtdm_mmap_ioctl {
        void *start;
        size_t length;
        int prot;
        int flags;
/* It makes no sense to use fd here since we already know it. */
        off_t offset;
/* return value */
        void *mapped_area;
};

struct rtdm_munmap_ioctl {
        void *start;
        size_t length;
/* return value */
        int errcode;
};
#define RTDM_MMAP      _IOWR ('m', 1, struct rtdm_mmap_ioctl)
#define RTDM_MUNMAP    _IOWR ('m', 2, struct rtdm_munmap_ioctl)
#endif /* RT_MMAP_H */
```

Apêndice C

Implementação da Seção Crítica do *Driver* da Placa de Captura de Imagens

Dentro da chamada IOCTL:

```
case VIDIOC_QBUF:
{
    struct v4l2_buffer *b = arg;
    int i;
    if ((b->type != ctx->rb.type) || (b->index >= ctx->rb.count) ||
        (b->memory != ctx->rb.memory))
        return -EINVAL;

    if (ctx->inbuffer_count == MAX_BUFFERS_PER_QUEUE)
        return -EINVAL;


    /* Mutual exclusion with interrupt handler */
    RTDM_EXECUTE_ATOMICALLY
    (

/* Pushes the buffer to the queue */
        i = ctx->inbuffer_count++ + ctx->inbuffer_pos;
        i %= MAX_BUFFERS_PER_QUEUE;
        ctx->inbuffer[i] = *b;

/* Begin capture if VICIOC_STREAMON was called
```

```
  and we are not yet capturing */
            if (ctx->streaming && (! dt->capturing))
            {
                  dt->capturing = 1;
                  /* Begin capture */
                  rt_dt3153_acquire(dt, b);
            }
      )

      return 0;
}

case VIDIOC_DQBUF:
{
      struct v4l2_buffer *b = arg;
      if ((b->type != ctx->rb.type) || (b->index >= ctx->rb.count) ||
            (b->memory != ctx->rb.memory))
            return -EINVAL;
      if (! ctx->outbuffer_count)
      {
            if (ctx->nonblocking)
                  return -EAGAIN;
            /* 100 ms should suffice*/
            else if (rtdm_event_timedwait(&(dt->evt_captured),
                  100*1000*1000 /*ns*/, NULL))
                  return -EIO;
      }

      *b = ctx->outbuffer[ctx->outbuffer_pos];

      /* Mutual exclusion with interrupt handler */
      RTDM_EXECUTE_ATOMICALLY
      (
            ctx->outbuffer_count--;
            ctx->outbuffer_pos++;
            ctx->outbuffer_pos %= MAX_BUFFERS_PER_QUEUE;
      )
      return 0;
```

}

Esse é o código da rotina de interrupção:

```
static int rt_dt3153_irq(rtdm_irq_t *irq_handle)
{
    struct rt_dt3153 *dt = rtdm_irq_get_arg(irq_handle,
            struct rt_dt3153);
    struct rt_dt3153_context *ctx = dt->ctx;
    static int ret;

    // código que verifica se a interrupção é desse dispositivo,
    // e ajusta os registros internos da placa, caso afirmativo.

    ...

    ret = (ctx->outbuffer_pos + ctx->outbuffer_count) %
            MAX_BUFFERS_PER_QUEUE;
    ctx->outbuffer[ret] = ctx->inbuffer[ctx->inbuffer_pos++];
    ctx->inbuffer_pos %= MAX_BUFFERS_PER_QUEUE;
    ctx->inbuffer_count--;
    ctx->outbuffer_count++;

    ctx->outbuffer[ret].timestamp = *((struct timeval *) &timestamp);

    rtdm_event_clear(&(dt->evt_captured));
    rtdm_event_signal(&(dt->evt_captured));

    if (ctx->inbuffer_count)
            //Acquire again...
            rt_dt3153_acquire(dt, &(ctx->inbuffer[ctx->inbuffer_pos]));
    else
            dt->capturing = 0;
    return RTDM_IRQ_HANDLED; /* re-enable interrupt line on return */
}
```

Apêndice D

Código Relevante do Experimento 1

```
const int Width=640, Height=240;
unsigned char hue[Width*Height];
unsigned char binary_front[Width*Height], binary_back[Width*Height],
        binary_target[Width*Height];
unsigned char eroded_target[Width*Height];

// memoria compartilhada com a interface grafica, estado do robo.
struct robot_state {
// indica a interface grafica, quando as informacoes estao coerentes
    int incomplete_result;
    int x, y, target_x, target_y;
// variacao desejada no angulo
    FloatType theta;
} rs = {0};

FloatType target_angle;

// Quadrado da distancia entre robo e alvo, dada em pixels.
int dist_2;

// essas variáveis são globais para serem compartilhadas
// pelo código da interface gráfica
int x_1, y_1, x_2, y_2;
bool Chegou=false;
```

```
// Inicializa o framework. Atualmente, apenas executa mlockall()
// para impedir que o programa seja escrito para o disco, com o
// uso de memoria virtual
INIT_FRAMEWORK()

SimpleRobot robot;

// Tarefa principal
class TaskFollowColor : public PeriodicTask {
public:
/* prioridade: 10, periodo: 300 ms */
    TaskFollowColor() : PeriodicTask(10, 300*1000*1000),
      rtvideo_handler(-1), i(0) {
        SetName("TaskFollowColor");
    }
    ~TaskFollowColor() {Destroy();}
    int Init() {
        int i;
        PeriodicTask::Init(); // Inicializa a classe base primeiramente.
        rtvideo_handler=rt_dev_open("rt_video0",O_RDWR);
/* preenche estrutura da API do Video4Linux utilizamos dois buffers nessa
implementacao. Enquanto um e preenchido, o outro e processado. Estes dois buffers
serao preenchidos nessa tarefa, um a cada iteracao do laco principal DoCycle(). */
        memset(&b, 0, sizeof(b)); // Inicializa estrutura com 0s binarios
        b[0].index      = 0; // Desnecessario. Para deixar clara a intencao
        b[0].type       = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        b[1].index      = 1;
        b[1].type       = V4L2_BUF_TYPE_VIDEO_CAPTURE;
// Estrutura do V4L2 para requisicao de buffers
        memset(&rb, 0, sizeof(rb));
        rb.type         = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        rb.memory       = V4L2_MEMORY_MMAP;
        rb.count        = 2; // requisitamos 2 buffers
        rt_dev_ioctl(rtvideo_handler, VIDIOC_REQBUFS, &rb);
        for (i=0;i<2;i++)
        {
            rt_dev_ioctl(rtvideo_handler, VIDIOC_QUERYBUF, &(b[i]));
/* A estrutura seguinte nao existe na API V4L2 e foi introduzida para substituir a
```

```
chamada mmap, que nao tem equivalente no RTDM, para uma chamada tipo IOCTL */
            mmap_ioctl.start=NULL;
            mmap_ioctl.length=b[i].length;
            mmap_ioctl.offset=b[i].m.offset;
            rt_dev_ioctl(rtvideo_handler, RTDM_MMAP, &mmap_ioctl);
            marea[i] = (unsigned char*) mmap_ioctl.mapped_area;
        }
// Enfileira os dois buffers
        for (i=0;i<2;i++)
            rt_dev_ioctl(rtvideo_handler, VIDIOC_QBUF, &(b[i]));
// Inicia processo de captura
        rt_dev_ioctl(rtvideo_handler, VIDIOC_STREAMON, &(b[0].type));

        return 0;
    }

    int Destroy(){
        Suspend(); // Suspende a execucao da tarefa
// Para o processo de captura
        rt_dev_ioctl(rtvideo_handler, VIDIOC_STREAMOFF, &(b[0].type));
/* Desfaz o mapeamento de memoria.
Somente a titulo de ilustracao. Nao e necessario pois tal desmapeamento
e realizado automaticamente quando o descritor de arquivos e fechado */
        struct rtdm_munmap_ioctl munmap_ioctl;
        for (i=0;i<2;i++) {
            munmap_ioctl.start = marea[i];
            munmap_ioctl.length = b[i].length;
            rt_dev_ioctl(rtvideo_handler, RTDM_MUNMAP, &munmap_ioctl);
        }
        rt_dev_close(rtvideo_handler); // fecha o descritor de arquivo
// Chama Destroy da classe base para excluir a tarefa
        return PeriodicTask::Destroy();
    }

// Metodo principal. Laco de controle da tarefa.
    void DoCycle(){
        i = i++ & 1; // Um modo de alternar entre os buffers a cada iteracao
        rt_dev_ioctl(rtvideo_handler, VIDIOC_DQBUF, &(b[i]));
```

```
/* As funcoes de processamento de imagem estao implementadas em outro arquivo
 Esta converte a imagem do formato BGR0 para uma matriz de saturacao:
 o H do espaco de cores HSV */
        bgr0tohue(marea[i], hue, Width*Height);
        for (p=hue, bf=binary_front,bb=binary_back, bt=binary_target;
            p<hue + sizeof(hue); p++,bf++,bb++,bt++) {
                if (*p>=20 && *p<=69) *bf = 1;
                else *bf = 0; /* amarelo: frente do robo */
                if (*p>210 || *p<=15) *bb = 1;
                else *bb = 0; /* vermelho: traseira do robo */
                if (*p>95 && *p<138) *bt = 1;
                else *bt = 0; /* verde: alvo */
        }
// Calcula os centroides de cada cor
        centromassa_margem(Width, Height, &x_1, &y_1, binary_front);
        centromassa_margem(Width, Height, &x_2, &y_2, binary_back);
/* Realiza uma erosao no alvo para retirar ruidos onde alguns pixels da faixa
 considerada como verde aparecem no chao azul. */
        image_erode(binary_target, Width, Height, eroded_target);
        centromassa_margem(Width, Height, &(rs.target_x),
          &(rs.target_y), eroded_target, 10, 5);
        rs.incomplete_result=1; // inicia calculo do estado do robo
        rs.x = (x_1+x_2)/2;
        rs.y = (y_1+y_2)/2;
        rs.theta = atan2((y_1-y_2),(x_1-x_2));
        dist_2 = (rs.target_y-rs.y)*(rs.target_y-rs.y)*4 +
                (rs.target_x-rs.x)*(rs.target_x-rs.x);
        if (dist_2<10000) {
                Chegou = true;
                robot.SetVel(0,0);
        }
        else
        {
                target_angle = atan2(rs.target_y-rs.y, rs.target_x-rs.x);
// Variacao desejada do angulo
                rs.theta = target_angle - rs.theta;
// Normaliza theta entre -pi e pi
                if (rs.theta>M_PI) rs.theta -= 2*M_PI;
```

```
            else if (rs.theta<-M_PI) rs.theta += 2*M_PI;
/* Invertido porque o angulo cresce em direcoes diferentes no sistema
   cartesiano do robo e no sistema da camera */
            rs.theta = -rs.theta;
            w = rs.theta*KP;
// Limita a velocidade angular a 40 graus por segundo
            if (w>40.0*M_PI/180.0)
                w = 40.0*M_PI/180.0;
            else if (w<-40.0*M_PI/180.0)
                w = -40.0*M_PI/180.0;
            robot.SetVel((DistanceType)(robot.GetMaxVel()*0.7), w);
        }
        rs.incomplete_result=0; // calculo do estado do robo concluido
// Enfileira outro buffer 33 ms antes do proximo ciclo
        rt_task_inquire(NULL, &task_info);
        rt_task_sleep(task_info.relpoint-33*1000*1000);
        rt_dev_ioctl(rtvideo_handler, VIDIOC_QBUF, &(b[i]));
    }
    bool OnOverruns (unsigned long overruns) {
        cout << "Overruns have occurred: " << overruns << endl;
        return true;
    }
    bool OnError (int errcode) {
        cout << "An error has occurred. Error code: " << errcode << endl;
        return true;
    }
private:
    int rtvideo_handler, i;
    unsigned char* marea[2];
    struct v4l2_buffer b[2]; // Inititialize all data with binary 0
    struct v4l2_requestbuffers rb;
    struct rtdm_mmap_ioctl mmap_ioctl;
    struct rtdm_munmap_ioctl munmap_ioctl;
    unsigned char *p, *bf, *bb, *bt;
    FloatType theta, w;
    static const FloatType KP = 1.0;
    RT_TASK_INFO task_info;
} taskFollowColor;
```

```
// Executada na inicializacao da interface grafica.
int rt_main()
{
    robot.Init();
    robot.Start();
    taskFollowColor.Init();
    taskFollowColor.Resume();
    return 0;
}
/* Metodo executado quando a janela e fechada.
Os outros metodos da interface grafica foram omitidos por simplicidade */
void MyFrame::OnClose(wxCloseEvent &event)
{
    robot.Stop();
    taskFollowColor.Destroy();
// Chama destrutor da janela grafica
    Destroy();
}
```

Apêndice E

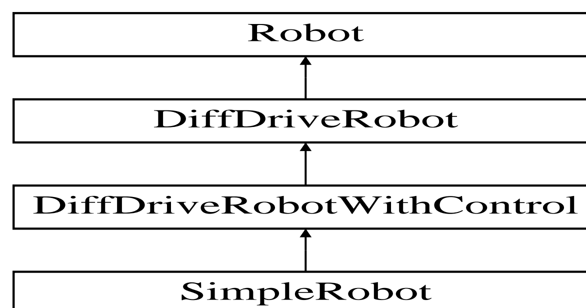
Exemplo de documentação gerada pelo Doxygen para a classe Robot

E.1 Robot Interface Reference

The **Robot**(p. 81) interface.

```
#include "rtmrf/robot.h"
```

Inheritance diagram for Robot::



Public Member Functions

- **Robot** (const char ∗name, float radPerAu, float nsPerTu, float mmPerDu=0.0)
- const char ∗ **GetName** ()

  Get robot name in a NULL terminated string.

- virtual int **GetId** ()

  Get robot identification.

- virtual enum **RobotType GetType** ()=0

  Get robot type.

- virtual int **Init** ()=0

  Initialize robot.

- virtual int **Start** ()=0

  Start-up the robot tasks.

- virtual int **Stop** ()=0

  Stop robot tasks.

- **FloatType GetRadPerAu** ()

  Used to convert between angle units (au) and radians (rad).

- **FloatType GetAuPerRad** ()

  Used to convert between radians (rad) and angle units (au).

- **FloatType GetNsPerTu** ()

  Used to convert time units (tu) to nanoseconds (ns), as used by Xenomai.

- **FloatType GetTuPerNs** ()

  Used to convert nanoseconds (ns), as used by Xenomai, to time units (tu).

- **FloatType GetMmPerDu** ()

  Used to convert distance units (du) to milimeters (mm) [Optional].

- **FloatType GetDuPerMm** ()

  Used to convert milimeters (mm) to distance units (du) [Optional].

- virtual void **DisableMotors** ()=0

Disable motor driving.

- virtual void **EnableMotors** ()=0

  Enable motor driving.

- virtual void **SetVel** (**DistanceType** vel, **AngleType** w)=0

  Set both linear and rotational speed.

- virtual **DistanceType** **GetVel** ()=0

  Get current linear speed.

- **DistanceType** **GetDesiredVel** ()

  Get previously set linear speed.

- virtual **AngleType** **GetRotVel** ()=0

  Get current rotational speed.

- **AngleType** **GetDesiredRotVel** ()

  Get previously set rotational speed.

- virtual void **SetPos** (**DistanceType** x, **DistanceType** y)=0

  Recalibrate robot position.

- virtual void **SetTheta** (**AngleType** theta)=0

  Recalibrate robot orientation.

- virtual void **GetPos** (**DistanceType** &x, **DistanceType** &y)=0

  Get current robot position.

- virtual **AngleType** **GetTheta** ()=0

  Get current robot orientation.

- virtual **DistanceType** **GetMaxVel** ()=0

  Get maximum robot linear speed.

- virtual **DistanceType** **GetMinVel** ()=0

Get minimum robot linear speed.

- virtual **AngleType GetMaxRotVel** ()=0

  Get maximum robot rotational speed.

- virtual **AngleType GetMinRotVel** ()=0

  Get minimum robot rotational speed.

- void **SetActionManager** (**ActionManager** ∗am)

  Set the action manager to be used in the **Robot**(p. 81).

- **ActionManager** ∗ **GetActionManager** ()

  Get the action manager used by the **Robot**(p. 81).

- void **SetSensorManager** (**SensorManager** ∗sm)

  Set the sensor manager to be used in the **Robot**(p. 81).

- **SensorManager** ∗ **GetSensorManager** ()

  Get the sensor manager used by the **Robot**(p. 81).

Protected Attributes

- int **m_id**

  Robot(p. 81) *identification.*

- **FloatType m_duPerMm**

  Used to convert milimeters (mm) to distance units (du).

- **FloatType m_radPerAu**

  Used to convert angle units (au) to radians (rad).

- **FloatType m_auPerRad**

  Used to convert radians (rad) to angle units (au).

- **FloatType m_nsPerTu**

  Used to convert time units (tu) to nanoseconds (ns), as used by Xenomai.

- **FloatType m_tuPerNs**

  Used to convert nanoseconds (ns), as used by Xenomai, to time units (tu).

- **FloatType m_mmPerDu**

  Used to convert distance units (du) to milimeters (mm).

- bool **m_motorsEnabled**

  True if motors are enabled and false otherwise.

- **DistanceType m_desiredVel**

  The linear speed set by SetVel should be stored here.

- **AngleType m_desiredRotVel**

  The rotational speed set by SetRotVel should be stored here.

- **DistanceType m_pos_x**

  Current x position of robot.

- **DistanceType m_pos_y**

  Current y position of robot.

- **AngleType m_theta**

  Current orientation of robot.

- char **m_name** [NAME_SIZE]

  Name of robot.

- **ActionManager** ∗ **m_actionManager**

  Action manager.

- **SensorManager** ∗ **m_sensorManager**

  Sensor(p. XXX) *manager.*

Static Protected Attributes

- static int **m_lastUsedId** = 0

  The last used robot identification.

E.1.1 Detailed Description

The **Robot**(p. 81) interface.

All actual robots must implement the **Robot**(p. 81) interface.

E.1.2 Constructor & Destructor Documentation

Robot::Robot (const char ∗ *name*, float *radPerAu*, float *nsPerTu*, float *mmPerDu* = `0.0`**)** `[inline]`

Constructor.

Parameters:

name **Robot**(p. 81) name. If GRAPHICAL_SUPPORT is not defined, this parameter does not exist.

radPerAu Radians per Angle Unit.

nsPerTu Nanoseconds per Time Unit.

mmPerDu Milimeters per Distance Unit [optional].

E.1.3 Member Function Documentation

virtual int Robot::GetId () `[inline, virtual]`

Get robot identification.

Each robot identification is unique. They are provided to be used by cooperative robots.

virtual enum RobotType Robot::GetType () `[pure virtual]`

Get robot type.

See RobotType enumeration type. This allows code such as

```
if (robot->GetType() == DIFFERENTIAL_DRIVE)
       diffRobot = (DiffDriveRobot *) robot;
diffRobot->SetLeftSpeed(10);...
```

Returns:
The robot type.

Implemented in **DiffDriveRobot** (p. XXX.

virtual int Robot::Init () `[pure virtual]`

Initialize robot.

Real-time tasks should be created here. They should, then, be started with **Start()**(p. 87) and stopped with **Stop()**(p. 88).

Returns:
0 (zero) upon success and a negative value otherwise. The error code is the negative of the return value. Error codes are defined by each real robot.

Implemented in **SimpleRobot** (p. XXX).

virtual int Robot::Start () `[pure virtual]`

Start-up the robot tasks.

Returns:
0 (zero) upon success and a negative value otherwise. The error code is the negative of the return value. Error codes are defined by each real robot.

Implemented in **SimpleRobot** (p. XXX).

virtual int Robot::Stop () `[pure virtual]`

Stop robot tasks.

Returns:

0 (zero) upon success and a negative value otherwise. The error code is the negative of the return value. Error codes are defined by each real robot.

Implemented in **SimpleRobot** (p. XXX).

FloatType Robot::GetRadPerAu () `[inline]`

Used to convert between angle units (au) and radians (rad).

Returns:

Angle[rad] = **GetRadPerAu()**(p. 88)∗Angle[au]

FloatType Robot::GetAuPerRad () `[inline]`

Used to convert between radians (rad) and angle units (au).

Returns:

Angle[au] = **GetAuPerRad()**(p. 88)∗Angle[rad]

FloatType Robot::GetNsPerTu () `[inline]`

Used to convert time units (tu) to nanoseconds (ns), as used by Xenomai.

Returns:

Time[ns] = **GetNsPerTu()**(p. 88)∗Time[tu]

FloatType Robot::GetTuPerNs () `[inline]`

Used to convert nanoseconds (ns), as used by Xenomai, to time units (tu).

Returns:

Time[tu] = **GetTuPerNs()**(p. 88)∗Time[ns]

FloatType Robot::GetMmPerDu () `[inline]`

Used to convert distance units (du) to milimeters (mm) [Optional].

This allows a generic program to work with any robot, using a common unit. However, its implementation is optional, returning 0 (zero) if not implemented.

Returns:

Distance[mm] = **GetMmPerDu()**(p. 89)∗Distance[mm]

FloatType Robot::GetDuPerMm () `[inline]`

Used to convert milimeters (mm) to distance units (du) [Optional].

This allows a generic program to work with any robot, using a common unit. However, its implementation is optional, returning 0 (zero) if not implemented.

Returns:

Distance[du] = **GetDuPerMm()**(p. 89)∗Distance[mm]

virtual void Robot::DisableMotors () `[pure virtual]`

Disable motor driving.

Robot(p. 81) should stop. This is the default state.

Implemented in **DiffDriveRobotWithControl** (p. XXX), and **SimpleRobot** (p. XXX).

virtual void Robot::SetVel (DistanceType *vel*, AngleType *w*) `[pure virtual]`

Set both linear and rotational speed.

Parameters:

vel the linear speed in du/tu. This value is returned by **GetDesiredVel()**(p. 90) method.

w the rotational speed in rad/s. This value is returned by **GetDesiredRotVel()**(p. 90) method. Positive if robot is turning to left and negative otherwise.

virtual DistanceType Robot::GetVel () `[pure virtual]`

Get current linear speed.

Returns:

the current linear speed in du/tu.

Implemented in **DiffDriveRobotWithControl** (p. XXX).

DistanceType Robot::GetDesiredVel () `[inline]`

Get previously set linear speed.

Returns:

the desired linear speed in du/tu, as set by **SetVel()**(p. 89) method.

virtual AngleType Robot::GetRotVel () `[pure virtual]`

Get current rotational speed.

Returns:

the current rotational speed in rad/s. Positive if robot is turning to left and negative otherwise.

Implemented in **DiffDriveRobotWithControl** (p. XXX).

AngleType Robot::GetDesiredRotVel () `[inline]`

Get previously set rotational speed.

Returns:

the desired rotational speed in rad/s, as set by SetRotVel() method. Positive if robot is turning to left and negative otherwise.

virtual void Robot::SetPos (DistanceType *x*, DistanceType *y*) `[pure virtual]`

Recalibrate robot position.

Parameters:

x the x axis is the robot direction. Forward values are positive and backward are negative.

y left values are positive and right ones are negative.

Implemented in **DiffDriveRobotWithControl** (p. XXX).

virtual void Robot::SetTheta (AngleType *theta*) `[pure virtual]`

Recalibrate robot orientation.

Parameters:

theta counter-clockwise angles are positive.

Implemented in **DiffDriveRobotWithControl** (p. XXX).

virtual void Robot::GetPos (DistanceType & *x*, DistanceType & *y*) `[pure virtual]`

Get current robot position.

Parameters:

x the x axis is the robot direction. Forward values are positive and backward are negative.

y left values are positive and right ones are negative.

Implemented in **DiffDriveRobotWithControl** (p. XXX).

virtual AngleType Robot::GetTheta () `[pure virtual]`

Get current robot orientation.

Returns:

Robot(p. 81) orientation in rad from $-\pi$ to π. Counter-clockwise angles are positive.

Implemented in **DiffDriveRobotWithControl** (p. XXX).

void Robot::SetActionManager (ActionManager ∗ *am*) [inline]

Set the action manager to be used in the **Robot**(p. 81).

Parameters:

am the action manager. Use 0 (zero) if not used.

ActionManager∗ Robot::GetActionManager () [inline]

Get the action manager used by the **Robot**(p. 81).

Returns:

the action manager. 0 (zero) is returned if no action manager is used.

void Robot::SetSensorManager (SensorManager ∗ *sm*) [inline]

Set the sensor manager to be used in the **Robot**(p. 81).

Parameters:

sm the sensor manager. Use 0 (zero) if there is not used.

SensorManager∗ Robot::GetSensorManager () [inline]

Get the sensor manager used by the **Robot**(p. 81).

Returns:

the sensor manager. 0 (zero) is returned if no sensor manager is used.

E.1.4 Member Data Documentation

int Robot::m_lastUsedId = 0 [static, protected]

The last used robot identification.

A new class should use "m_id=++m_lastUsedId;" in Constructor or **Init()**(p. 87). Since **Robot**(p. 81) constructor already does that, it is possible to implement a new class like that:

```
class TheRobot: public Robot {
 public:
       TheRobot():Robot(){...}
 ...
 };
```

int Robot::m_id `[protected]`

Robot(p. 81) identification.

Derived classes should define an ID based on m_lastUsedId value.

FloatType Robot::m_duPerMm `[protected]`

Used to convert milimeters (mm) to distance units (du).

Distance[du] = m_duPerMm∗Distance[mm]

FloatType Robot::m_radPerAu `[protected]`

Used to convert angle units (au) to radians (rad).

Angle[rad] = m_radPerAu∗Angle[au]

FloatType Robot::m_auPerRad `[protected]`

Used to convert radians (rad) to angle units (au).

Angle[au] = m_auPerRas∗Angle[rad]

FloatType Robot::m_nsPerTu `[protected]`

Used to convert time units (tu) to nanoseconds (ns), as used by Xenomai.

Time[ns] = m_nsPerTu∗Time[tu]

FloatType Robot::m_tuPerNs `[protected]`

Used to convert nanoseconds (ns), as used by Xenomai, to time units (tu).

Time[tu] = m_tuPerNs∗Time[ns]

FloatType Robot::m_mmPerDu `[protected]`

Used to convert distance units (du) to milimeters (mm).

Distance[mm] = m_mmPerDu∗Distance[du]

DistanceType Robot::m_pos_x `[protected]`

Current x position of robot.

Should be initialized with zero. The x axis is the robot direction. Positive values are forward, related to original robot's position.

DistanceType Robot::m_pos_y `[protected]`

Current y position of robot.

Should be initialized with zero. The y axis is positive to the left of the original robot's position.

AngleType Robot::m_theta `[protected]`

Current orientation of robot.

Should be initialized with zero. Positive values are counter-clockwise.

char Robot::m_name[NAME_SIZE] `[protected]`

Name of robot.

NAME_SIZE is defined per default as 15 in **robot_types.h**(p. XXX).

ActionManager∗ Robot::m_actionManager `[protected]`

Action manager.

Should be 0 if not used or point to an **ActionManager**(p. XXX) object.

SensorManager∗ Robot::m_sensorManager `[protected]`

Sensor(p. XXX) manager.

Should be 0 if not used or point to a **SensorManager**(p. XXX) object.

The documentation for this interface was generated from the following files:

- include/rtmrf/**robot.h**
- robot.cpp

Referências Bibliográficas

ACTIVMEDIA. *Mobile Robots & Robotics Software*. 1995. Disponível em: <http://www.activmedia.com>. Acesso em: 05 jan. 2006.

ACTIVMEDIA ROBOTICS, INC. *ARIA Overview 2.1.1*. [S.l.], 2004.

Albus, J. S.; Quintero, R.; Lumia, R. Overview of NASREM: The NASA/NBS standard reference model for telerobot control system architecture. *NASA STI/Recon Technical Report N*, v. 95, p. 12854–+, abr. 1994.

ANDREWS, G. R.; SCHNEIDER, F. B. Concepts and notations for concurrent programming. *Computing Surveys*, v. 15, n. 1, p. 3–43, mar. 1983.

AUSTIN, D. *Dave's Robotics Operating System*. 2002. Disponível em: <http://dros.org/>. Acesso em: 02 mai. 2005.

BLANK, D. et al. Pyro: A python-based versatile programming environment for teaching robotics. *J. Educ. Resour. Comput.*, ACM Press, New York, NY, USA, v. 3, n. 4, p. 1–15, 2003. ISSN 1531-4278. Disponível em: <http://pyro.sourceforge.net/>. Acesso em: 02 mai. 2005.

BORENSTEIN, J.; EVERETT, H.; FENG, L. *Where am I? Sensors and Methods for Mobile Robot Positioning*. 1996. Disponível em: <http://www-personal.umich.edu/j̃ohannb/-shared/pos96rep.pdf>.

BRUYNINCKX, H. Open robot control software: the OROCOS project. In: *ICRA*. Seoul, Korea: IEEE, 2001. p. 2523–2528. ISBN 0-7803-6578-X.

CÔTÉ, C. et al. Code reusability tools for programming mobile robots. In: *Proceedings of IROS 2004*. [S.l.: s.n.], 2004.

DIAPM. *RTAI: Real-Time Application Interface*. 1998. Disponível em: <http://www.rtai.org>. Acesso em: 03 mai. 2005.

FARINES, J. M.; FRAGA, J. da S.; OLIVEIRA, R. S. de. *Sistemas de Tempo Real*. [s.n.], 2000. Disponível em: <http://www.lcmi.ufsc.br/gtr/livro/principal.htm>.

GERECKE, U.; HOHMANN, P.; WAGNER, B. Concepts and components for robots in higher education. In: *Proceedings of the World Automation Congress (WAC'2004)*. Sevilla, Spain: [s.n.], 2004.

GERKEY, B. P.; VAUGHAN, R. T.; HOWARD, A. The Player/Stage project: Tools for multi-robot and distributed sensor systems. In: *Proceedings of the International Conference on Advanced Robotics*. Coimbra, Portugal: [s.n.], 2003. p. 317–323. Disponível em: <http://robotics.stanford.edu/ gerkey/research/final_papers/icar03-player.pdf>. Acesso em: 02 mai. 2005.

GERUM, P. *Xenomai - Implementing a RTOS emulation framework on GNU/Linux*. [S.l.], 2004. Disponível em: <http://www.xenomai.org>. Acesso em: 07 fev. 2006.

HARRISON, T. H.; LEVINE, D. L.; SCHMIDT, D. C. The design and performance of a real-time CORBA event service. In: . [s.n.], 1997. p. 184–200. Disponível em: <citeseer.ist.psu.edu/article/harrison97design.html>.

HOHMANN, P.; GERECKE, U.; WAGNER, B. A scalable processing box for systems engineering teaching with robotics. In: *Proceedings of the International Conference on Systems Engineering (ICSE'2003)*. Coventry, UK: [s.n.], 2003.

KUO, Y. hsin; MACDONALD, B. A. Designing a distributed real-time software framework for robotics. *Proceedings of the Australasian Conference on Robotics And Automation 2004*, dez. 2004.

LEVINE, D.; FLORES-GAITAN, S.; SCHMIDT, D. *Measuring OS Support for Real-time CORBA ORBs*. 1999. Disponível em: <citeseer.ist.psu.edu/article/levine99measuring.html>.

MANTEGAZZA, P.; DOZIO, E. L.; PAPACHARALAMBOUS, S. Rtai: Real time application interface. *Linux Journal*, Specialized Systems Consultants, Inc., Seattle, WA, USA, v. 2000, n. 72es, p. 10, 2000. ISSN 1075-3583. Disponível em: <http://www2.linuxjournal.com/article/3838>. Acesso em: 03 mai. 2005.

MELCHIOR, N.; SMART, W. D. A framework for robust mobile robot systems. In: GAGE, D. W. (Ed.). *Proceedings of SPIE: Mobile Robots XVII*. [S.l.: s.n.], 2004. v. 5609.

MICROSOFT. *COM: Component Object Model Technologies*. 1993. Disponível em: <http://www.microsoft.com/com/>. Acesso em: 02 mai. 2005.

MONTEMERLO, M.; ROY, N.; THRUN, S. *CARMEN: Carnegie Mellon Robot Navigation Toolkit*. 2002. Disponível em: <http://www-2.cs.cmu.edu/ carmen/>. Acesso em: 02 mai. 2005.

OPEN MANAGEMENT GROUP. *CORBA: Common Object Request Broker Architecture*. 1991. Disponível em: <http://www.omg.org/>. Acesso em: 02 mai. 2005.

OPEN MANAGEMENT GROUP. *The Common Object Request Broker: Architecture and Specification*: Version 2.2. [S.l.], 1998.

OPEN MANAGEMENT GROUP. *RealTime - CORBA Specification*: Version 2.0. [S.l.], 2003.

PROJETO OCEAN. *OCEAN: Open Controller Enabled by an Advanced Real-Time Network*. 2001. Disponível em: <http://www.fidia.it/english/research_ocean_fr.htm>. Acesso em: 02 mai. 2005.

ROMANO, V. F. (Ed.). *Robótica Industrial: Aplicação na Indústria de Manufatura e de Processos*. [S.l.]: Editora Edgard Blücher LTDA, 2002.

ROSENBLATT, J. K. DAMN: A distributed architecture for mobile navigation. In: *Proc. of the AAAI Spring Symp. on Lessons Learned from Implememted Software Architectures for Physical Agents*. Stanford, CA: [s.n.], 1995. Disponível em: <citeseer.ist.psu.edu/rosenblatt95damn.html>.

SCHMIDT, D. C. The ADAPTIVE communication environment: An object-oriented network programming toolkit for developing communication software. In: *Proceedings of the Sun User Group Conference*. San Jose, California: [s.n.], 1993. Disponível em: <http://www.cs.wustl.edu/ schmidt/ACE-papers.html>.

SCHMIDT, D. C.; DESHPANDE, M.; O'RYAN, C. Operating system performance in support of real-time middleware. *Proceedings of the Seventh IEEE Workshop on Object-oriented Real-time Dependable Systems*, San Diego, CA, EUA, p. 199–206, jan. 2002.

SCHMIDT, D. C.; KUHNS, F. An overview of the real-time corba specification. *IEEE Computer*, v. 33, n. 6, p. 56–63, 2000.

SCHOLL, K.-U. *Modular Controller Architecture 2*. 1998. Disponível em: <http://mca2.sourceforge.net/>. Acesso em: 02 mai. 2005.

UTZ, H. et al. Miro – middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, v. 18, n. 4, p. 493–497, ago. 2002.

WOLFE, V. F. et al. Real-time CORBA. *IEEE Transactions on Parallel and Distributed Systems*, v. 11, n. 10, p. 1073–1089, 2000. Disponível em: <citeseer.ist.psu.edu/article/wolfe97realtime.html>.

WOLFE, V. F. et al. Real-time corba. *IEEE Transactions Parallel Distributed Systems*, v. 11, n. 10, p. 1073–1089, 2000.

YAGHMOUR, K. Adaptive domain environment for operating systems. fev. 2001.

YODAIKEN, V.; BARABANOV, M. *A Real-Time Linux*. 1996. Disponível em: <http://www.fsmlabs.com>. Acesso em: 03 mai. 2005.