UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

DIEGO CABERLON SANTINI

ARQUITETURA ABERTA PARA CONTROLE DE ROBÔS MANIPULADORES

Porto Alegre
2009

DIEGO CABERLON SANTINI

ARQUITETURA ABERTA PARA CONTROLE DE ROBÔS MANIPULADORES

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Controle e Automação

ORIENTADOR: Prof. Dr. Walter Fetter Lages

Porto Alegre
2009

DIEGO CABERLON SANTINI

ARQUITETURA ABERTA PARA CONTROLE DE ROBÔS MANIPULADORES

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Walter Fetter Lages, UFRGS
Doutor pelo Instituto Tecnológico de Aeronáutica – São José dos Campos, Brasil

Banca Examinadora:

Prof. Dr. Glauco Augusto de Paula Caurin, USP/SC
Doutor pela Eidgenössische Technische Hochschule – Zurique, Suiça

Prof. Dr. Luís Fernando Alves Pereira, UFRGS
Doutor pelo Instituto Tecnológico de Aeronáutica – São José dos Campos, Brasil

Prof. Dr. Renato Ventura Bayan Henriques, UFRGS
Doutor pela Universidade Federal de Minas Gerais – Belo Horizonte, Brasil

Coordenador do PPGEE: _____
Prof. Dr. Arturo Suman Bretas

Porto Alegre, outubro de 2009.

DEDICATÓRIA

Aos meus pais, Gilberto Santini e Maria Inês Caberlon Santini, e ao meu irmão Thiago Caberlon Santini.

AGRADECIMENTOS

Agradeço aos meus pais e irmão, pelo suporte ao longo de toda a minha vida.

Ao meu orientador, Prof. Dr. Walter Fetter Lages, pela sua atenção e ensinamentos ao longo da minha vida acadêmica.

Ao Programa de Pós-Graduação em Engenharia Elétrica, PPGEE, pela a oportunidade.

Aos bolsistas Guilherme Strack, Rodrigo Daniel Trevizan e Vinícuis Scheeren, por suas contribuições neste trabalho.

Aos meus amigos, em especial a Jorge Augusto Vasconcelos Alves, Diogo de Oliveira Fialho Pereira, Jonathan Henrique Efigenio de Oliveira, Marcos Rosendo Dalte e Renato Gonçalves Ferraz, pelas horas de diversão.

À minha namorada Camila Haas Ploia Manera, por sua compreensão.

RESUMO

Este documento trata da especificação de uma arquitetura aberta para controle de robôs manipuladores. A arquitetura é implementada utilizando o *framework* do projeto OROCOS, ambiente que já foi utilizado com sucesso em alguns sistemas de controle de robôs. Esta arquitetura é especificada para um robô manipulador genérico de N juntas, definindo componentes que abstraem o *hardware* dos robôs. A arquitetura é implementada com três tipos de controladores diferentes: PID independente por junta, controlador de torque calculado e controlador com *feedforward*. A sua validação é feita através da sua implementação em um robô real. Para isso é utilizada uma placa de acionamento, utilizando o barramento CAN devido ao seu determinismo e a sua taxa de comunicação. Também é necessário a utilização do modelo dinâmico do robô para as estratégias de controle de torque calculado e com *feedforward*. A obtenção de tal modelo é feita neste trabalho de forma analítica, e a seguir os parâmetros são identificados usando o sistema proposto.

Palavras-chave: Arquitetura aberta, OROCOS, Controle de robôs, Identificação parâmetrica, Barramento CAN.

ABSTRACT

This work deals with the specification of an open architecture for control of manipulator robots. The architecture is implemented by using the OROCOS framework. The architecture is specified for a generic manipulator robot with N joints, through definition of components which abstract the hardware of the robot. Three different controllers are implemented: an independent PID for each joint, a computed torque controller and a controller with feedforward. The validation is made through the implementation on the Janus robot. For this purpose, an actuator card is defined. This card uses the CAN bus due its determinism and bus rate. The dynamic model of Janus, used in computed torque and feedforward controllers, is obtained in an analytical way. After that, the parameters of this model are identified using the least squares method.

Keywords: Open architecture, OROCOS, Robot control, Parametric identification, CAN bus.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

API	*Application Programming Interface*
AIC	*Actuator Interface Card*
BFL	*The Orocos Bayesian Filtering Library*
CAN	*Controller Area Network*
CRC	*Cyclic Redundancy Check*
CTS	*Clear to Send*
DC	*Direct current*
DICAM	*Distributed Intelligent Control and Management*
FIFO	*First In First Out*
FIR	*Finite Impulse Response*
HAL	*Hardware Abstraction Layer*
ISO	*International Organization for Standardization*
I/O	*Input/Output*
JOP	*Japan FA Open Systems*
KDL	*The Orocos Kinematics and Dynamics Library*
OCL	*The Orocos Components Library*
OMAC	*Open Modular Architecture Controls*
OROCOS	*Open Robot Control Software*
OS	*Operating System*
OSACA	*Open System Architecture for Controls within Automation*
QEI	*Quadrature Encoder Interface*
PC	*Personal Computer*
PCI	*Peripheral Component Interconnect*
PLL	*Phase-Locked Loop*
PWM	*Pulse Width Modulation*
RCCL	*Robot Control C Library*

RCI	*Robot Control Interface*
RIPE	*Robot Independent Programming Environment*
RTAI	*RealTime Application Interface*
RTS	*Request to Send*
RTT	*The Orocos Real-Time Toolkit*
UART	*Universal Asynchronous Receiver Transmitter*
XML	*Extensible Markup Language*

LISTA DE SÍMBOLOS

β	Ângulo de *Pitch*
α	Ângulo de *Roll*
γ	Ângulo de *Yaw*
K_d	Ganho derivativo
K_i	Ganho integral
K_p	Ganho proporcional
$H(q, \dot{q})$	Matriz de forças centrífugas e de Coriolis
$D(q)$	Matriz de inércia
ϕ	Matrix de regressores
${}^{i}A_j$	Matriz de Transformação Homogênea
q	Posição angular
V	Tensão
τ	Torque
x	Vetor de estados
$G(q)$	Vetor de forças gravitacionais
θ	Vetor de parâmetros desconhecidos

SUMÁRIO

1 INTRODUÇÃO

Uma arquitetura de controle fornece uma forma estruturada de especificação, projeto, e se possível, implementação e validação de sistemas de controle complexos e seus subsistemas (PIRJANIAN et al., 2000). Assim, pode-se definir uma arquitetura de controle como sendo uma abstração de uma série de agentes como: o conjunto de componentes estruturais onde o acionamento e o sensoriamento ocorrem; a especificação de funcionalidade e de interface de cada componente, a topologia de interconexão entre os componentes, entre outros.

Sistemas robóticos são intrinsecamente complexos pois englobam teoria de controle, ciência de computação, programação e muita matemática. Além disto, eletrônica, mecânica e mecatrônica são conhecimentos necessários para construir e manter plataformas de *hardware*. Desta forma, quando um pesquisador deseja concentrar-se em um problema particular do robô, ele necessita aprender detalhes de todo o sistema. A facilidade para lidar com este problema está diretamente relacionada com o tipo de arquitetura do robô.

Os tipos de arquitetura de controladores de robôs podem ser classificados em três grandes categorias: sistemas fechados (também conhecidos como proprietários), abertos e híbridos (FORD, 1994).

1.1 Sistemas Proprietários

Um sistema proprietário tem como principal característica o fato do robô ser uma "caixa preta". Neste caso, a interface e as ações do robô são conhecidas, mas a implementação não. Atualmente, a grande maioria de controladores de robôs manipuladores que são vendidos no mercado possuem sistemas fechados. Isto até é desejável quando a aplicação do robô é específica e imutável, pois o custo do desenvolvimento do controlador é compensado pela venda de várias unidades.

Porém isto gera dois problemas: O primeiro é quando o robô deixa de ter o suporte do seu fabricante, ou ainda, quando algumas peças do robô não são mais vendidas. O segundo ocorre quando o sistema proprietário que satisfaz as necessidades do usuário não existe. Em ambos os casos, a modificação do sistema fechado é uma tarefa cara e árdua, quando não impossível sem uma total reformulação.

1.2 Sistemas Abertos

Em um sistema aberto, todos os detalhes do robô são amplamente documentados e a estrutura de *software* e *hardware* é feita de tal forma, que permite a adição de

novos sensores, controladores e interfaces, independente dos seus fabricantes. Dessa forma, todos os aspectos de projeto do robô podem ser modificados sem muita dificuldade.

Segundo (FORD, 1994), uma arquitetura aberta deve obedecer aos seguintes princípios:

1. Utilizar um sistema de desenvolvimento baseado em uma plataforma computacional não proprietária (SUN, PCs).

2. Utilizar um sistema operacional (UNIX,VxWorks) e uma linguagem de controle padrão (C ou C++).

3. Possuir um barramento de comunicação padrão que consiga interligar diversos periféricos.

Este tipo de estrutura, além de promover um sistema flexível que até agora não foi alcançado, devido a dificuldades de integração dos padrões dos diferentes fabricantes, estimula a concorrência entre os fabricantes e o reuso de componentes de sistemas já existentes.

Desde o início dos anos 80, diversos projetos de controladores abertos foram desenvolvidos:

O projeto RIPE (MILLER; LENNOX, 1990), desenvolvido nos *Sandia National laboratories*, utiliza uma abordagem orientada a objetos para o desenvolvimento de arquiteturas de *software* para sistemas robóticos. Ele fornece uma linguagem de programação independente de robô, através da definição de uma hierarquia de classes bases genéricas e de suas subclasses, específicas a dispositivos. As subclasses herdam a interface das classes genéricas. Este tipo de abordagem permite que a subclasse seja substituída por outra, sem que o resto do sistema precise ser adaptado.

A arquitetura DICAM (HAYES-ROTH et al., 1992) consiste em uma arquitetura genérica de controle aplicável para múltiplas tarefas que precisam cooperar. Dentre as diversas *features* que esta arquitetura oferece, convém ressaltar uma série de módulos específicos que são combinados para produzir controladores inteligentes. Isso facilita o desenvolvimento de código reusável.

RCCL/RCI (LLOYD, 1992a,b) são duas bibliotecas, independentes, para controle de robôs manipuladores em linguagem C. A RCCL oferece uma série de funções e estruturas de dados específicas às aplicações robóticas, como matrizes de transformações e funções para determinar a trajetória de manipuladores. A RCI fornece a interface entre a RCCL e um servo-controlador, através da definição de uma interface de baixo nível, a qual o servo-controlador deve implementar.

Chimera (STEWART; VOLPE; KHOSLA, 1997) é uma metodologia que oferece o desenvolvimento de *software* reutilizável que pode ser reconfigurado dinamicamente utilizando uma estratégia baseada em portas. Nesse caso, uma porta é um objeto específico para a comunicação de outros objetos, onde a ligação entre as portas é feita através dos seus nomes. Isso melhora a comunicação de dados do sistema e permite uma configuração dinâmica.

Todos os projetos acima tentaram resolver o problema da realização de um controlador de arquitetura aberta, e diversos protótipos foram desenvolvidos. Eles não foram amplamente aceitos devidos as definições restritas e padrões especiais, porém apontaram que uma filosofia baseada em componentes deveria ser a solução para controladores de arquitetura aberta (LIANDONG; XINHAN, 2004)

Com esta filosofia, surgiram os controladores de alta flexibilidade: o americano OMAC (BAILO; ALDERSON; YEN, 1994), o europeu OSACA (OSACA HANDBOOK VERSION 2.0, 2001) e o japonês JOP (PRITSCHOW et al., 2001). Eles abordam o tema de maneira mais ampla, definindo padrões para o desenvolvimento de arquiteturas de controle de dispositivos eletro-mecânicos. Porém tais especificações são genéricas demais para serem consideradas uma solução para controladores de robôs manipuladores (GASPAR, 2003).

Seguindo a mesma filosofia, porém com uma arquitetura voltada especificamente para robôs, nasceu o projeto OROCOS, que é um acrônimo de *Open Robot Control Software*. O projeto desenvolve um *framework* modular para o controle de robôs e máquinas, seguindo o modelo *Open Source* que é amplamente utilizado em outros pacotes de *software* tais como: Linux, Firefox, LaTeX e outros. Ele já foi utilizado com sucesso em diversos trabalhos como em (LI; HENRIK; ANDERS, 2005), (TAVARES; AROCA; PAULA CAURIN, 2007) e (SWEVERS; VERDONCK; DE SCHUTTER, 2007).

O OROCOS foi desenvolvido a partir das seguintes premissas (BRUYNINCKX, 2001):

1. Possuir modularidade e flexibilidade de forma que os usuários construam seus próprios sistemas *à la carte* e que desenvolvedores possam contribuir para os módulos que eles estão interessados, sem precisar desenvolver código para todo o sistema.

2. Ser independente de fabricantes de robôs comerciais.

3. Servir para todos os tipos de dispositivos robóticos e rodar em diversas plataformas.

4. Oferecer componentes de *software* para modelos cinemáticos e dinâmicos de robôs, geração de trajetória, sensoriamento, controle, interface de *hardware*.

5. Despertar motivação nos pesquisadores e engenheiros para contribuírem e utilizarem o seu código.

É interessante notar que cuidados com documentação e roteiro não estão incluídos nessas premissas, de onde se percebe um dos pontos fracos da iniciativa e justifica-se uma revisão dos conceitos utilizados no decorrer deste trabalho.

1.3 Sistemas Híbridos

Os sistemas híbridos são compostos por partes proprietárias e partes abertas. Geralmente as funções de baixo nível são disponibilizadas ao usuário com as suas descrições. Por exemplo, o vendedor pode ser responsável pelo servo-motor de cada junta do robô, sendo que a referência para estas juntas são geradas por um *software* externo. Este *software* externo pode ser um programa desenvolvido pelo usuário, um programa desenvolvido por terceiros, ou ainda um programa do próprio vendedor.

Desta forma é possível ganhar alguns benefícios de sistemas fechados e mesmo assim conseguir adaptar o sistema conforme a necessidade do problema. Porém, quando é necessário modificar as partes proprietárias do sistema, tem-se os mesmos problemas dos sistemas fechados.

1.4 Objetivos

Este trabalho tem por objetivo desenvolver uma arquitetura aberta para controle de robôs manipuladores utilizando o *framework* OROCOS. A arquitetura deverá ser modular e genérica para suportar diversos robôs manipuladores.

Para alcançar este objetivo, serão definidos classes bases de forma a tornar a arquitetura independente da implementação, de modo similar ao que é feito em (MILLER; LENNOX, 1990). A seguir, uma implementação genérica para um robô manipulador de N juntas utilizando uma interface de *hardware* genérica será implementada. Esta interface de *hardware* genérica deverá definir as funções de baixo nível de forma similar a RCI (LLOYD, 1992b). A interface de comunicação entre os componentes da arquitetura utilizará uma primitiva similar ao conceito de portas de (STEWART; VOLPE; KHOSLA, 1997).

Para avaliar a arquitetura, ela será aplicada a um robô manipulador de 2 juntas, com três estratégias de controle diferentes. Para isso também será necessário desenvolver uma placa de acionamento para as juntas, bem como levantar o modelo dinâmico do robô.

1.5 Estrutura do documento

Além desta introdução, este documento contém outros 5 capítulos que são resumidos a seguir: o Capítulo 2 explica alguns conceitos básicos do *framework* OROCOS, necessários para a implementação da arquitetura. O Capítulo 3 descreve a arquitetura proposta, bem como a sua extensão para um manipulador de N juntas. No Capítulo 4 é desenvolvida a placa de acionamento, utilizando o barramento CAN. Por fim, o Capítulo 5 contém a implementação para o robô Janus e a identificação dos parâmetros do modelo dinâmico do mesmo. O Capítulo 6 conclui este trabalho.

2 OROCOS

OROCOS é um acrônimo de *Open Robot Control Software*. Ele é um *framework* para controle de robôs e máquinas. Esse projeto segue a filosofia de desenvolvimento *Open Source* que tem se mostrado útil em outros projetos como Linux, Apache, Perl ou LaTeX (BRUYNINCKX, 2001). O projeto OROCOS utiliza uma tática de "dividir para conquistar", de tal forma, que o seu código base está divido em quatro bibliotecas escritas em C++, conforme a Figura 1:

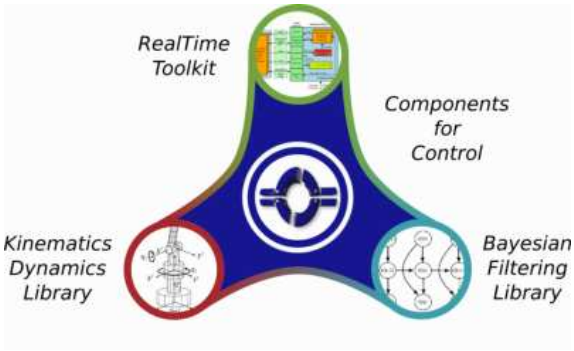


Figura 1: Projeto OROCOS(THE OROCOS PROJECT SMARTER CONTROL IN ROBOTICS AND AUTOMATION, 2009).

- *The Orocos Real-Time Toolkit* (RTT) (OROCOS REAL TIME TOOLKIT SMARTER REAL TIME, 2009): É um pacote que contém a infraestrutura básica do OROCOS, como abstração do sistema operacional e mecanismos de comunicação entre tarefas. Ele é voltado para o desenvolvimento de aplicações baseadas em componentes e de tempo real.

- *The Orocos Components Library* (OCL) (COMPONENTS FOR CONTROL THE OROCOS PROJECT, 2009): É uma biblioteca que contém diversos modelos de componentes prontos para serem utilizados. Estes modelos variam desde modelos para controle de robôs, ferramentas para depuração de sistemas ou *drivers* para acesso à *hardware*.

- *The Orocos Kinematics and Dynamics Library* (KDL) (OROCOS KINEMATICS AND DYNAMICS SMARTER IN MOTION, 2009): É uma biblioteca para cálculos de modelos cinemáticos e dinâmicos de robôs, em tempo real. Atualmente somente os modelos cinemáticos estão implementados e testados. Os modelos dinâmicos estão em versões de teste no momento da escrita deste trabalho.

- *The Orocos Bayesian Filtering Library* (BFL) (OROCOS BAYESIAN FILTERING LIBRARY THINK SMARTER, 2009): É um conjunto de funções independentes para processamento recursivo de informações e algoritmos de estimação baseados na regra de Bayes, como filtro de Kalman e filtro de partículas.

Essa modularização, permite que todas as bibliotecas sejam desenvolvidas de forma independente entre si, com exceção da OCL. Isto porque os modelos de componentes da OCL são feitos utilizando as funcionalidades das outras bibliotecas. Por exemplo, um gerador de trajetória pode utilizar a KDL para a interpolação dos pontos e o RTT para disponibilizar a trajetória para outros componentes.

O projeto OROCOS pode ser dividido em três partes básicas:

- *Framework*: É o conjunto de *software* que implementa a infraestrutura para a construção de aplicações. Essa infraestrutura tem por objetivo abstrair o sistema operacional da plataforma utilizada através da padronização das funções bases do sistema, como semáforos e *threads* dos sistemas operacionais, e também de fornecer funções de suporte para o desenvolvimento de aplicações do usuário, tais como a solução de modelos cinemáticos diretos e inversos no caso da KDL. As bibliotecas RTT, BFL e KDL são exemplos deste tipo de *software*.

- Modelos de Componentes: Os modelos de componentes descrevem uma interface de um serviço utilizando a infraestrutura do *framework*. Um bloco que calcula uma lei de controle a partir de uma referência e uma saída é um exemplo de componente. A OCL faz parte deste grupo de *software*.

- *Software* de Aplicação: Este é o *software* que utiliza os modelos de componentes e os interliga para fazer uma aplicação específica. Por exemplo, interligar um componente que implementa uma determinada lei de controle com um componente que simula uma determinada planta é uma aplicação. O projeto OROCOS como um todo é voltado para o desenvolvimento deste tipo de *software*.

Este capítulo tem por objetivo apresentar as funcionalidades dos subsistemas do OROCOS que são utilizadas neste trabalho, do ponto de vista de desenvolvedores de modelos de componentes e de aplicações. O RTT contém as primitivas necessárias para implementar os modelos de componentes adequados ao sistema, enquanto a OCL fornece alguns componentes para a integração do sistema. As versões do RTT 1.8.2 e OCL 1.8.2 foram utilizadas no desenvolvimento deste trabalho.

2.1 *Orocos Real-Time Toolkit*

O RTT é um *middleware* localizado entre a aplicação e o sistema operacional, que permite a construção de um sistema independente da plataforma computacional utilizada. Isto é possível pois o RTT faz a abstração das primitivas do sistema operacional para sua própria API. Ele também é responsável pela a comunicação e execução dos componentes no sistema.

Uma visão mais detalhada das camadas de uma aplicação utilizando o OROCOS é vista na Figura 2. Sobre o *hardware* é executado um sistema operacional (OS).

O OS fornece uma interface para o usuário acessar o *hardware*. O RTT abstrai o OS, permitindo que a aplicação execute sobre as seguintes plataformas: GNU/Linux (suportando também o *patch* do RTAI e o *patch* do Xenomai) e MAC/OS X. Utilizando esta abstração, o RTT fornece uma interface para a criação de modelos de componentes. Apesar disto, ainda é possível acessar o sistema operacional diretamente no componente, porém perde-se portabilidade ao fazer isto, uma vez que se cria uma dependência do componente em relação ao OS.

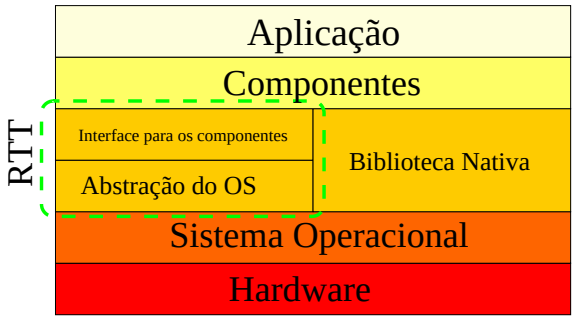


Figura 2: Camadas de uma aplicação utilizando o OROCOS.

Um componente é uma unidade básica que executa uma ou mais ações, com uma determinada atividade. O OROCOS permite que essas ações sejam uma função em linguagem C, ou um *script* com sua própria linguagem ou ainda uma máquina de estados hierárquica.

O OROCOS fornece quatro políticas para a forma de controle das atividades dos componentes: `NonPeriodicActivity`, `PeriodicActivity`, `SequentialActivity` e `SlaveActivity`. Como o próprio nome já diz, `PeriodicActivity` gera um componente com uma atividade periódica, fazendo com que ele execute a cada período. O esquema `NonPeriodicActivity` cria uma atividade não periódica que irá executar sempre que houver algo a ser feito e quando o processador estiver disponível. O esquema `SequentialActivity` cria uma tarefa sem atividade própria, ou seja, sempre que uma tarefa de um componente com esse esquema for requisitada, ela irá executar instantaneamente dentro da atividade do componente que o requisitou. Por fim, o esquema `SlaveActivity` permite que um outro componente comande a atividade do componente com este esquema.

No OROCOS, cada componente é derivado da classe `TaskContext`, que define uma interface pública para a interação entre componentes. Essa interface é descrita através de cinco primitivas do RTT: Evento, Método, Comando, Propriedade e Porta de Dados, como visto na Figura 3.

A primitiva evento é um objeto onde pode-se conectar funções. Quando o evento é disparado, as funções conectadas a ele são chamadas. O RTT permite que as funções sejam conectadas para serem funções síncronas ou assíncronas, e que elas possuam até quatro argumentos.

Quando um evento é emitido, todas as funções síncronas são executadas no mesmo instante, executando na mesma *thread* que emite o evento. As funções assíncronas são mapeadas para serem executadas em outra oportunidade de acordo com a atividade do componente. Note que um componente com uma atividade `SequentialActivity` irá mapear um evento assíncrono como se fosse síncrono.

Eventualmente, pode ocorrer que um evento seja redisparado, antes de que a função assíncrona associada a ele seja executada, neste caso, a função assíncrona

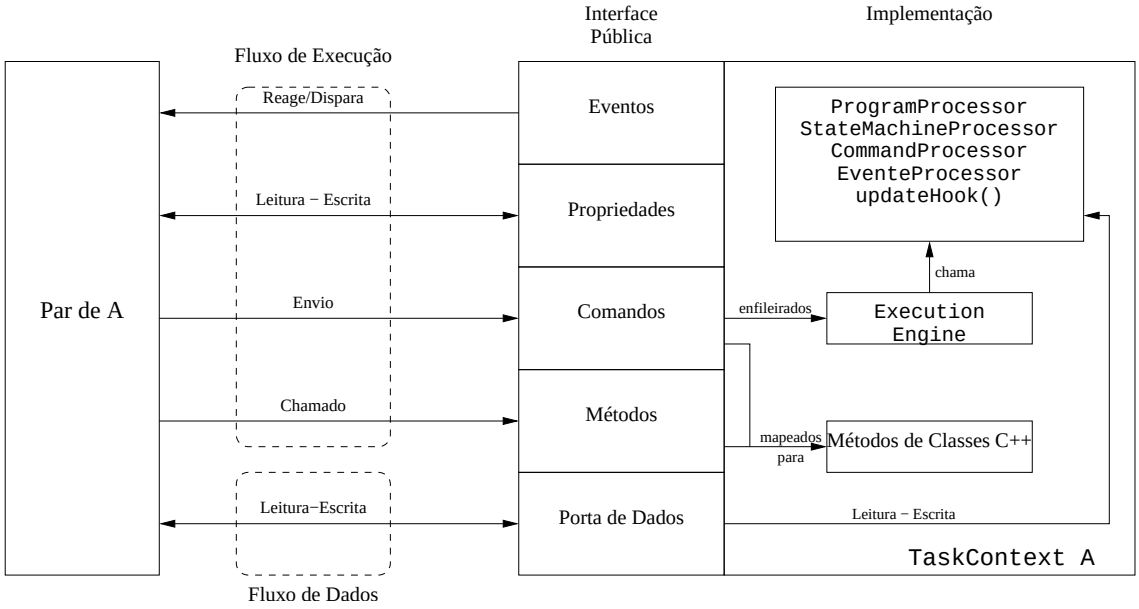


Figura 3: Interface de componentes no OROCOS.

será executada somente uma vez. Neste caso, também surge o problema de qual será o argumento passado para a função. O RTT fornece duas políticas para estes casos, que o argumento passado seja relativo ao primeiro evento gerado ou relativo ao último evento gerado.

A primitiva método é bem similar às funções de C/C++ de um componente, porém com a vantagem de que elas podem ser chamadas a partir de um *script* ou até mesmo através de uma rede. Os métodos são executados de forma síncrona em relação aos componentes que os chamaram, assim, eles são executados como se fossem funções normais de C/C++. Eles podem receber até quatro argumentos, e o valor de retorno pode ser de qualquer tipo. Os métodos não são *thread-safe*, ou seja, deve-se tomar cuidado ao implementar um método que será acessado de forma simultânea por dois componentes distintos.

O conceito de *thread-safe* está relacionado com a segurança do código em um ambiente multi-tarefa. Códigos *thread-safe* são códigos reentrantes, ou seja, o código pode ser acessado de forma simultânea por várias *threads*, ou ainda, um trecho de código protegido contra acesso simultâneo por alguma forma de exclusão mútua (exemplo: semáforos, mutex, etc).

A primitiva comando é similar à primitiva método, com a diferença que comandos são enviados ao componente e são executados conforme a atividade do componente que recebe os comandos, ou seja, de forma assíncrona com o componente que o enviou. A Figura 4 ressalta esta diferença. Nela tem-se dois componentes periódicos de mesmo período. Quando o componente A chama um método do componente B, nota-se que imediatamente o método é executado. Quando o componente A envia um comando para o componente B, a execução só ocorre quando o componente B é novamente escalonado.

Como os métodos, os comandos podem receber até quatro argumentos, porém a variável de retorno deve ser obrigatoriamente uma variável booleana, que indica se o comando foi aceito ou não. Vários comandos podem ser enviados, neste caso, eles serão enfileirados com os seus respectivos argumentos.

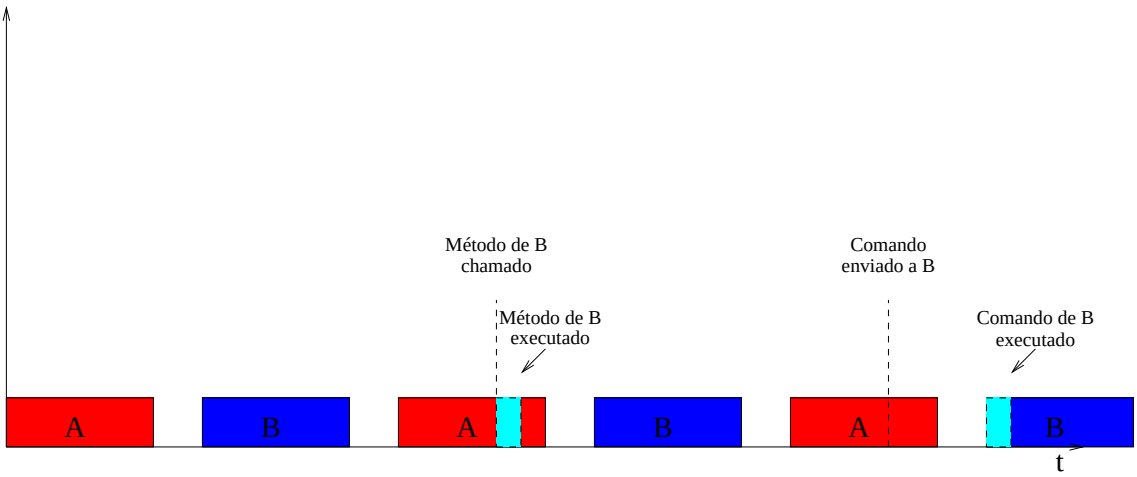


Figura 4: Comandos × métodos.

Associado a cada primitiva comando, pode se incluir uma condição de sucesso para ele. A condição de sucesso é uma função que pode ter os mesmos argumentos do comando associado a ela, ou somente o primeiro argumento ou nenhum argumento. Ela é executada no componente que recebeu o comando, logo após o mesmo ser executado. Ela deve retornar se ele foi realizado com sucesso ou não.

A primitiva propriedade é uma variável que pode ser lida de um arquivo de configuração no formato XML. Desta forma, ela serve para fazer configurações de parâmetros no componente ou, ainda, armazenar valores persistentes, como por exemplo, a posição final de um robô pode ser escrita em um arquivo XML ao finalizar o sistema e recuperada novamente ao inicializar o sistema.

Ler ou escrever em propriedades é uma tarefa de tempo real, porém não é *thread-safe*, e portanto deve ser utilizada com cautela.

Para que um componente possa utilizar as quatro primitivas citadas até agora de um outro componente, é necessário que este componente seja conectado como um par do outro componente. Ao ser conectado como um par, o componente ganha acesso à interface pública do outro componente, conforme a Figura 3. Isto pode ser feito através de métodos da classe `TaskContext` ou via outro componente específico que será explicado na subseção 2.2.1.

A primitiva porta de dados é uma forma de troca de dados entre componentes. Estas portas podem possuir ou não *buffers*. Uma porta com *buffer* funciona como uma FIFO, enquanto uma porta sem *buffer* funciona como uma variável em memória.

Estas portas podem ser de escrita, leitura ou escrita e leitura, e são acessadas em tempo real, de forma *thread-safe*. Assim, quando uma *thread* faz a leitura da porta, há garantia que nenhuma outra *thread* a estará modificando naquele momento. As portas de dados de leitura também podem ser configuradas para disparar a atividade do componente, ou ainda executar uma determinada função ao receber um dado.

As portas de dados devem ser conectadas entre si, e isso pode ser feito através de métodos da classe `TaskContext`, ou explicitamente através da própria porta, ou via outro componente específico, que será explicado na subseção 2.2.2.

O bloco `ExecutionEngine` da Figura 3 é executado de acordo com o esquema de atividade do componente e tem a sua função mostrada na Figura 5. Quando ativado, o `ExecutionEngine` irá chamar um trecho de código do usuário através da função `UpdateHook` e também os processadores mostrados na Figura 5.

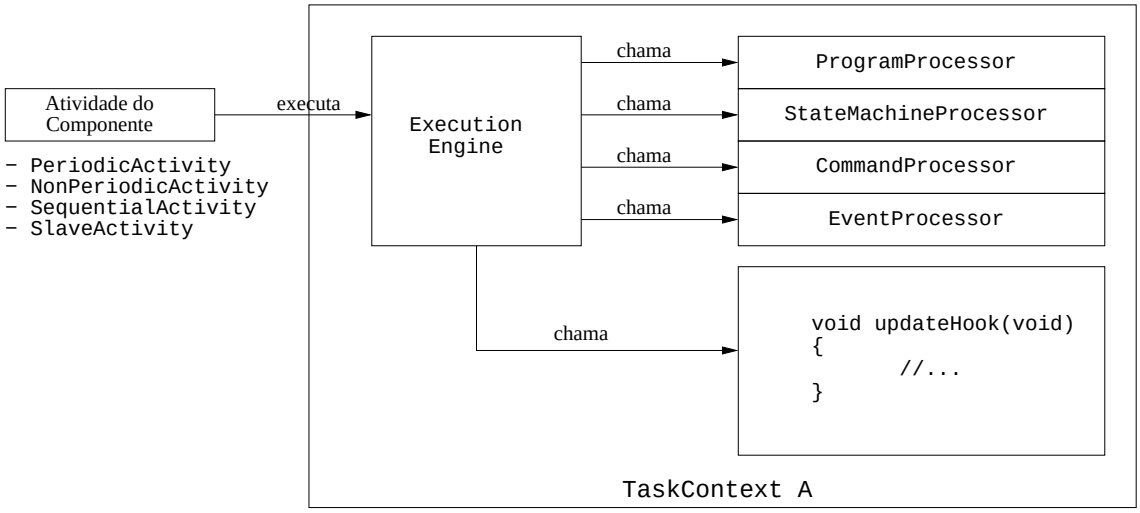


Figura 5: Função do `ExecutionEngine`.

O `CommandProcessor` irá enfileirar e executar um comando de cada vez, enquanto o `EventProcessor` irá chamar todos as *callbacks* assíncronas associadas aos eventos que ocorreram.

O `ProgramProcessor` serve para realizar tarefas através de *scripts*, cada vez que ele é chamado, ele irá executar todos os *scripts* que estão carregados. O *script* é executado até que ele precise esperar uma condição ser verdadeira. Maiores informações sobre a linguagem dos *scripts* são encontradas em (SOETENS, 2009a).

O `StateMachineProcessor` serve para realizar tarefas através de máquinas de estado. Cada vez que este processador é chamado, ele irá chamar um estado das máquinas de estado que estão carregadas nele. Maiores informações sobre as máquinas de estado são encontradas em (SOETENS, 2009a).

O código do usuário é implementado nas funções virtuais da classe `TaskContext` chamadas "Hooks":

- `configureHook()`: Função executada para configurar o componente;
- `startHook()`: Função executada para inicializar o componente;
- `updateHook()`: Função executada segundo a atividade do componente;
- `stopHook()`: Função executada para parar o componente;
- `cleanupHook()`: Função executada para finalizar o componente;
- `activeHook()`: Função executada para ativar o componente;
- `errorHook()`: Função que substitui `updateHook` em caso de erro não crítico;
- `resetHook()`: Função para recuperar um erro crítico;

Estas funções são executadas através de métodos públicos da classe `TaskContext`, quando o estado do componente é alterado. Um componente pode ter os estados:

- `PreOperational`: Um estado que indica que o componente precisa ser configurado, neste estado o `ExecutionEngine` não é ativado e o componente não aceita comandos e nem eventos;

- `Stopped`: Estado no qual o componente está parado, esperando para começar a rodar, neste estado o `ExecutionEngine` não é ativado e o componente não aceita comandos e nem eventos;
- `Active`: Estado no qual o `ExecutionEngine` executa o `CommandProcessor` e o `EventProcessor`;
- `Running`: Similar ao `Active`, porém a função `updateHook()` também é chamada;
- `RunTimeWarning`: Similar ao `Running`, e serve para indicar que um possível problema aconteceu;
- `RunTimeError`: Similar ao `Running`, porém a função `errorHook()` é chamada ao invés de `updateHook()`;
- `FatalError`: Este estado serve para indicar um erro fatal, neste estado o `ExecutionEngine` não é ativado e o componente não aceita comandos e nem eventos;

A relação entre os estados, as funções dos usuários, e os métodos de `TaskContext` pode ser vista na Figura 6. Note que para se recuperar de um erro fatal, a função do usuário `resetHook()` é chamada. Se ela retornar `true`, o componente não precisa ser configurado novamente e irá para o estado `Stopped`, caso contrário, irá para o estado `PreOperational`.

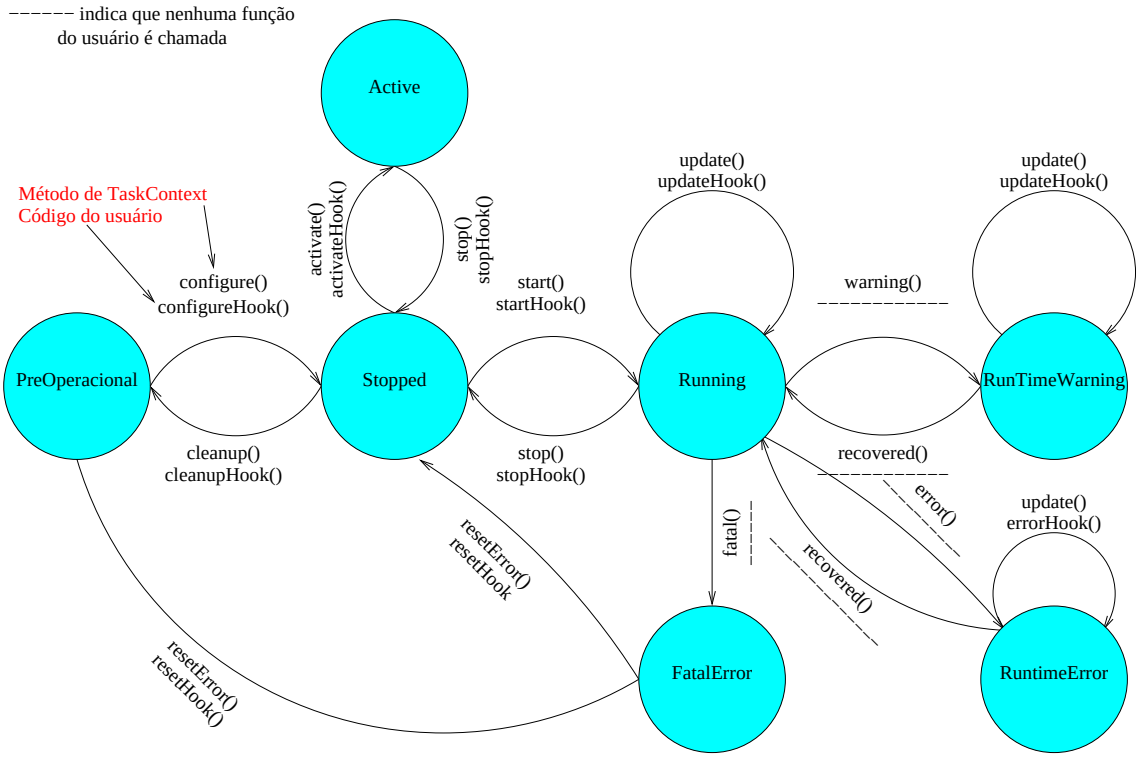


Figura 6: Relação entre os estados.

Maiores informações sobre o RTT podem ser encontradas em (SOETENS, 2009a).

2.2 *The Orocos Components Library*

A OCL é uma biblioteca de modelos de componentes do OROCOS. Os modelos utilizam o RTT como *framework* e algumas outras bibliotecas nas suas implementações. Os modelos de componentes da OCL oferecem uma variedade de funcionalidades como: acesso ao *hardware*, (utilizando a biblioteca do Comedi), ferramentas de *debug* e de *log*, controle de robôs (utilizando a biblioteca KDL), entre outras. Nesta seção são explicados os componentes utilizados neste projeto.

2.2.1 `TaskBrowser`

O `TaskBrowser` é um componente utilizado para interações com outros componentes (SOETENS, 2007). Cada componente no OROCOS tem uma interface padrão através da classe `TaskContext` e o `TaskBrowser` a utiliza para interagir com os componentes. Ele funciona como um console, ou seja, recebe comandos através de uma linha de comando e os executa.

O `TaskBrowser` pode se conectar dinamicamente como par de qualquer outro componente para obter acesso a sua interface. Paralelamente, ele também cria portas de dados similares ao do componente sendo visitado e as conecta. Deste modo, ele pode enviar dados e comandos, disparar eventos, executar métodos de qualquer outro componente.

O `TaskBrowser` também pode se conectar internamente a um componente. Desta forma, os comandos são interpretados como se fossem gerados pelo o próprio componente, enquanto no outro caso, eles são interpretados como comandos gerados por um componente par deste (SOETENS, 2007). A Tabela 1 lista alguns comandos ao `TaskBrowser`.

Tabela 1: Alguns comandos do `TaskBrowser`.

Comando	Argumento [opcional]	Descrição
help	-	Mostra os comandos disponíveis para o `TaskBrowser`.
ls	[par]	Mostra a interface e o *status* de um componente par.
cd	par	Visita um par do atual componente.
enter	-	Interpreta os comandos como se o componente os estivesse emitido.
leave	-	Interpreta os comandos como se eles fossem emitidos por um componente externo.
quit	-	Sai do `TaskBrowser`.

2.2.2 `DeploymentComponent`

O `DeploymentComponent` é um componente para carregar e configurar componentes a partir de arquivos XML (SOETENS, 2009b). Geralmente ele é utilizado em conjunto com o `TaskBrowser`, que passa os comandos para o `DeploymentComponent`.

A OCL possui uma aplicação para utilizar o `DeploymentComponent` chamada **deployer-<target>**, onde o **<target>** é o nome do alvo para o qual o OROCOS

foi compilado. Este programa ainda pode receber como argumento um arquivo XML que descreve: os componentes que devem ser carregados, as interconexões entre eles, e inicializa a aplicação. Após a aplicação ser inicializada, o `TaskBrowser` é chamado para interagir com o usuário.

Os passos que o `DeploymentComponent` utiliza para inicializar a aplicação estão representados na Figura 7. Inicialmente o comando `import` é utilizado para carregar os diretórios onde estão as bibliotecas dos componentes. A seguir os componentes utilizados são instanciados com o comando `loadComponent`.

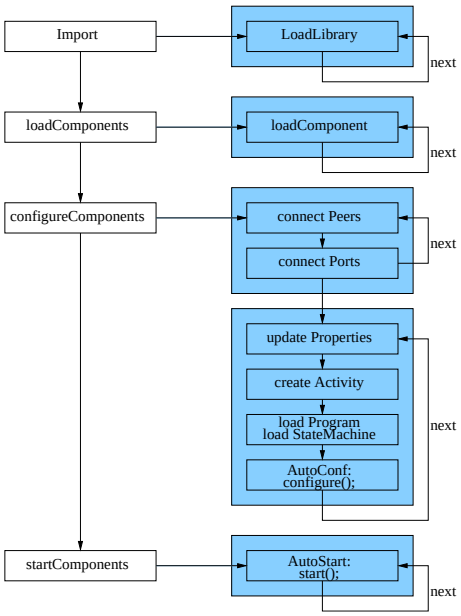


Figura 7: Inicialização do sistema utilizando o `DeploymentComponent`.

A configuração do sistema é feita inicialmente conectando os componentes com os seus respectivos pares e as suas portas de dados. Após a implementação de todas as conexões, as propriedades de cada componente são atualizadas de um arquivo XML. O esquema de atividade pode ser modificado nesta fases também, bem como novos *scripts* e máquinas de estados podem ser inseridas no `ExecutionEngine` de cada componente. Por fim, o método `configure()` pode ser chamado para executar a função do usuário `configureHook()`.

Na etapa de inicialização, a função `start()` pode ser chamada. Caso isto ocorra e a função retorne `true`, o estado do componente passa a ser `Running` e o componente começa a funcionar.

2.2.3 `ReportingComponent`

O `ReportingComponent` é um componente para monitorar e capturar a troca de dados entre componentes do OROCOS (SOETENS, 2009c). Ele funciona se conectando como par dos componentes que serão monitorados, e ligando-se as portas de dados especificadas.

O `ReportingComponent` não pode ser usado diretamente, pois ele é um componente virtual. Um componente virtual define uma interface e geralmente não contém nenhuma implementação. O `ReportingComponent` é estendido através de dois componentes: `FileReporting` para gravar os dados monitorados em um arquivo; e o `ConsoleReporting` para imprimir os dados na tela.

Este componente possui uma série de opções que controlam o seu funcionamento, essas opções são propriedades do componente e podem ser carregadas de um arquivo XML, entre elas pode-se destacar as seguintes:

- *AutoTrigger:* Permite que o `ReportingComponent` funcione de forma periódica ou aperiódica, na forma aperiódica ele só irá gravar os dados quando o usuário chamar a função `snapshot()`;
- *WriteHeader:* Escreve um cabeçalho no início do arquivo de monitoramento;
- *Decompose:* Permite que o `ReportingComponent` grave arquivos no formato NetCDF (Network Common Data Format).
- *ReportFile:* Nome do arquivo a ser gravado no disco, no caso do `FileReporting`.
- *Synchronize:* Grava o tempo no qual a amostragem ocorreu.

2.2.4 `nAxisGeneratorPos`

Este modelo de componente é um gerador de trajetórias no espaço das juntas de um robô. Ele cria uma trajetória entre as posições dos eixos atuais do robô e as novas posições desejadas. Ele possui três propriedades para a sua configuração: o número de eixos do robô, a velocidade máxima de cada junta e a aceleração máxima de cada junta. Ele utiliza a KDL para a interpolação de pontos. A interpolação utiliza um perfil de velocidade trapezoidal, levando em conta a máxima aceleração e a máxima velocidade de cada junta.

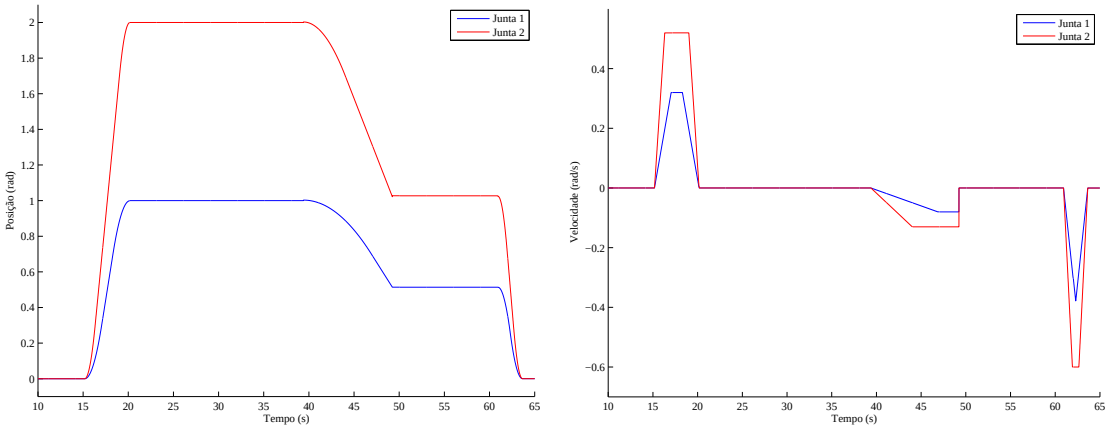
A trajetória de todos os eixos são normalizadas de forma que o movimento de todos os eixos irá levar o mesmo tempo. Ele utiliza três portas de dados para a sua comunicação: uma porta de leitura para receber um vetor com a posição atual, e duas portas de escrita para escrever a posição e a velocidade desejadas.

O `nAxisGeneratorPos` possui o comando `moveTo(pos,time)`. Este comando irá gerar um novo perfil de posição e velocidade para as juntas. A trajetória iniciará na posição atual das juntas do robô e terminará na posição especificada no argumento `pos`, demorando o tempo especificado no argumento `time`.

Ele também possui o método `resetPosition` para implementar uma parada de emergência. Diferente de um comando, o método sempre será executado de forma síncrona com quem o chamou. Esse método reseta a posição desejada para a posição atual do robô e atualizará a velocidade desejada para zero.

As Figuras 8(a) e 8(b) mostram os perfis de posição e velocidade, respectivamente, gerados para um robô de duas juntas, para a seguinte sequência de comandos: um comando `moveTo` é enviado em 15 s, outro comando `moveTo` em 40 s, o método `resetPosition` é chamado em 50 s e por fim, outro comando `moveTo` é enviado em 60 s. As velocidades máximas são limitadas em 0.6 rad/s.

O primeiro comando `moveTo` é enviado para solicitar que a junta 1 vá para a posição 1 rad e a junta 2 para 2 rad, em um tempo de 5 s. A seguir, o segundo comando `moveTo` é novamente enviado para que as juntas retornem a posição inicial em 20 s. Durante a execução deste comando, o método `resetPosition` é chamado, parando assim o robô. Por fim, o último comando `moveTo` é enviado para que as juntas retornem a posição inicial em 10 ms. Este tempo faz com que a velocidade da junta 2 sature no seu máximo, demorando mais tempo para ser executado. Note que a velocidade da junta 1 não satura pois ela é normalizada com o tempo da junta mais lenta, no caso a junta dois.



(a) Posição Gerada. (b) Velocidade Gerada.

Figura 8: Trajetória gerada pelo `nAxisGeneratorPos`

2.3 Conclusão

Com base nesta introdução ao OROCOS, nota-se que ele atende todas as necessidades para a implementação de um sistema de controle para robôs. A implementação de um *framework* com uma filosofia baseada em componentes o torna uma boa opção para a implementação de uma arquitetura aberta, facilitando futuras extensões e modificações no sistema. As primitivas do RTT, citadas na Seção 2.1, possibilitam diversas formas de interações entre os componentes e oferecem muitos graus de liberdade na implementação do sistema, isto possibilita que o usuário implemente o seu sistema de acordo com a sua necessidade. As suas implementações *thread-safe* e de tempo real oferecem a robustez necessária para sistemas de controle de robôs manipuladores.

Os componentes já implementados na OCL, citados na Seção 2.2, facilitam a implementação do sistema. O `DeploymentComponent` permite que o sistema seja configurado de forma dinâmica, sem a necessidade de uma recompilação do mesmo. O `TaskBrowser`, além de ser uma poderosa ferramenta de *debug*, oferece uma interface com o usuário, permitindo que uma interação *online* entre o sistema e o usuário final. O `ReportingComponent` se mostra a forma adequada para a geração de `log`, fato necessário na maioria dos sistemas de controle.

Por fim, o OROCOS possibilita implementações de sistemas complexos de forma conveniente, pois seria muito árduo caso fosse preciso implementar as funcionalidades necessárias, justificando assim o seu uso.

3 ARQUITETURA PROPOSTA

Uma topologia típica de controle é vista na Figura 9. Nela tem-se a planta a qual deseja-se controlar e um controlador que gera os valores para a entrada da planta. A realimentação é utilizada, uma vez que ela dá uma maior robustez ao sistema, diminuindo a sensibilidade do sistema em relação à variação dos parâmetros dele e em relação a perturbações externas, além de permitir a estabilização de um sistema originalmente instável.

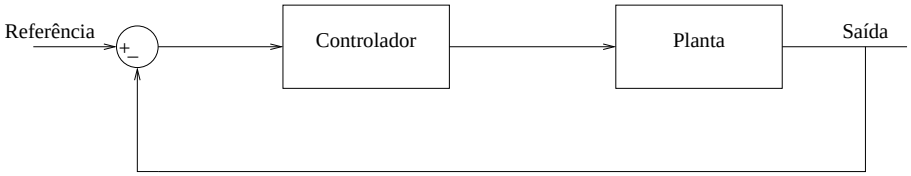


Figura 9: Diagrama de blocos típicos de um sistema de controle.

Do ponto de vista do controlador, este sistema pode ser generalizado realizando a soma internamente no controlador e assumindo que o controlador recebe o sinal da planta através de sensores e que ele atua na planta através de um atuador.

Para realizar o controle em um computador, é necessário a colocação de mais dois blocos no sistema. Estes blocos têm a função de realizar uma ponte entre o sistema dinâmico contínuo que é a planta, e o sistema dinâmico discreto que é o controlador. O amostrador irá amostrar o sinal do sensor, enquanto o segurador não permite que a entrada do atuador varie até a próxima amostragem. A Figura 10 representa este sistema. Esta topologia servirá como base para o desenvolvimento da arquitetura.

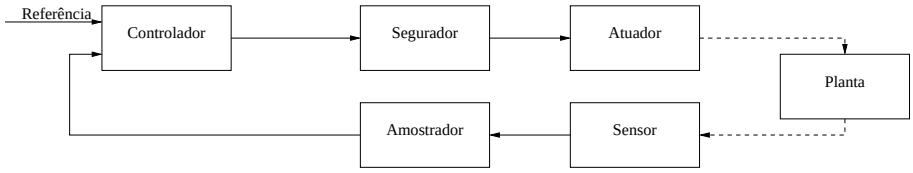


Figura 10: Diagrama de blocos típicos de um sistema de controle amostrado.

O modelo dinâmico de um robô manipulador pode ser representado por (FU; GONZALES; LEE, 1987):

$$\tau = D(q)\ddot{q} + H(q, \dot{q}) + G(q) \tag{1}$$

onde $D(q)$ é a matriz de inércia generalizada, $H(q, \dot{q})$ é o vetor de forças centrífugas e de Coriolis, $G(q)$ é o vetor de forças gravitacionais, q é o vetor com as posições angulares das juntas e τ é o vetor com os torques aplicados nas juntas.

Em geral, o torque é o sinal de acionamento da junta, enquanto a posição e a velocidade são sinais medidos ou estimados das juntas. A Equação (1) pode ainda ser reescrita na forma:

$$\ddot{q}_i = f_i(q_1, \ldots, q_N, \dot{q}_1, \ldots, \dot{q}_N, \tau_i)$$

ou

$$\dot{x} = f(x, \tau)$$

com

$$x = \begin{bmatrix} q & \dot{q} \end{bmatrix}^T$$

Neste trabalho é proposta uma arquitetura de controle de robôs manipuladores com $2N$ sensores e N atuadores, onde N representa o número de juntas do robô. A implementação dessa arquitetura é feita utilizando o *framework* OROCOS, de forma modular e genérica, e é explicada neste capítulo.

3.1 Implementação no OROCOS

Para a implementação da arquitetura é conveniente a criação de modelos de componentes para os blocos da Figura 10, com exceção do segurador, que é considerado como parte do atuador do robô. A Figura 11 demonstra os modelos criados. A criação destes modelos de componentes torna o sistema modular, onde cada modelo tem a sua função específica e independe da implementação dos demais componentes do sistema.

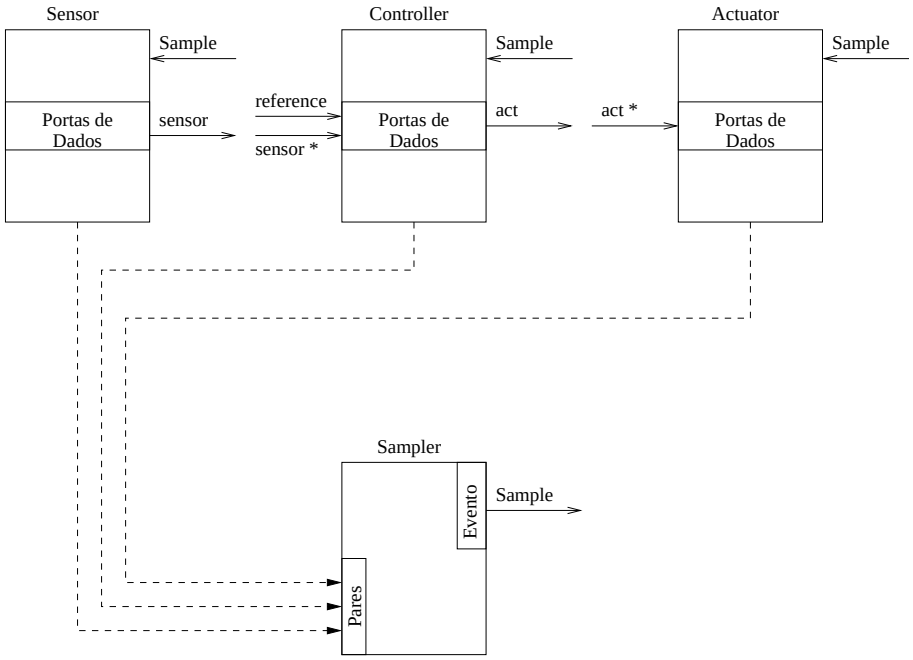


Figura 11: Modelos de componentes.

O modelo de componente `Sensor`, bem como o `Controller` e o `Actuator`, são modelos de componentes base, similar ao `ReportingComponent` descrito na subse-

ção 2.2.3. Deste modo, eles apenas definem a interface entre eles e não contêm nenhuma implementação. Assim, implementações específicas devem ser feitas utilizando estes componentes como classes bases do componente a ser implementado.

O `Sampler` é o amostrador do sistema. Ele tem como função gerar o evento `Sample` para sincronizar os demais componentes. O evento é gerado na sua função `updateHook()`, de forma que, esta funcionará de acordo com a atividade do componente. De modo geral, esta atividade será periódica, porém nada a impede de ser aperiódica. Todos os demais componentes são registrados como pares do `Sampler` para obter acesso à sua interface de eventos.

O modelo `Sensor` tem a função de abstrair os sensores do sistema. Ele contém uma porta de dados onde ele só deve escrever o valor dos sensores. Essa porta é representada por um vetor e o seu tamanho dependerá do número de juntas do sistema, que é definido através do construtor desta classe. Também é definido que os dados que trafegam na porta sensor são ordenados de acordo com (2), onde as posições angulares estão nas primeiras posições do vetor, seguido depois pelas velocidades angulares.

$$sensor = \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_N \\ \dot{q}_1 \\ \dot{q}_2 \\ \vdots \\ \dot{q}_N \end{bmatrix} \tag{2}$$

O modelo `Sensor` cadastra uma *callback* para ser executado de forma síncrona com o evento `Sample`. A *callback* é implementada na função virtual `sample_now`, de forma que o usuário possa reimplementá-la novamente caso necessário. Por definição, a função `sample_now` apenas atribui à variável `sample_ocurred` o valor `true`. Está variável é utilizada para indicar ao componente que o evento `Sample` foi emitido, permitindo uma eventual sincronização entre os componentes do sistema.

O modelo `Actuator` tem a função de abstrair o atuador do sistema. Na sua implementação base, ele também conecta-se ao evento `Sample` de forma similar ao modelo `Sensor`, instalando uma função virtual com o nome `sample_now`. Ele possui uma porta de dados, no qual ele somente pode ler o valor que deverá ser aplicado no sistema. Esta porta também é especificada como um vetor cujo o tamanho será igual ao número de juntas do sistema. Este argumento é passado na construção da classe. O `*` na porta `act*`, indica que essa porta irá emitir um evento quando houver uma escrita dos dados de atuação nela. Uma função virtual com o nome `act_now` é automaticamente associada a este evento de forma assíncrona. Está função serve para fazer com que uma escrita na porta `act*` reflita em uma atuação efetiva no sistema.

O modelo `Controller` tem a função de abstrair o controlador do sistema. Para isso possui três portas de dados, similar aos dados que entram no controlador da Figura 10. As portas `reference` e `sensor*` são para a leitura do vetor da referência e dos sensores do sistema, sendo que quando um componente escrever na porta `sensor*` a função virtual `controller_now` é chamada. Essa função tem o objetivo de

calcular o valor da atuação no sistema. A implementação derivada deste componente deve implementar essa função de acordo com a sua necessidade. A porta `act` serve para a escrita do sinal de controle por parte do controlador. De forma similar aos outros modelos, este componente também possui uma função virtual conectada no evento `Sample`, para oferecer um método de sincronismo ao controlador.

As funções de configuração dos componentes `configureHook()` servem para inicializar as portas de dados de escrita `sensor` no `Sensor` e `act` no `Controller` com os seus respectivos tamanhos. As funções de inicialização `startHook()` verificam se todas as portas estão conectadas, e conectam as *callbacks* no evento `Sample`.

3.2 Modelo de Componente `Joint`

A criação do modelo de componente `Joint` tem por objetivo a manipulação de uma junta de um robô. Esse modelo de componente deverá estar associado à um modelo que represente o *hardware* de cada junta. É interessante que o acesso ao *hardware* não seja feito dentro do componente `Joint` para não atrelar esse componente a um determinado *hardware*. Fazendo-se as duas implementações separadamente, permite-se que qualquer atuador seja utilizado, desde que ele respeite a interface do componente `Joint`. A Figura 12 demonstra a interface desse componente.

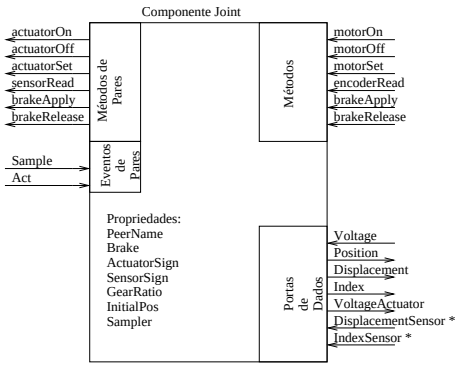


Figura 12: Modelo de componente `Joint`.

Quando o modelo `Joint` é criado, ele deve ser conectado como par do componente que representa o *hardware* do robô. O componente que representa o *hardware* deve ser capaz de fornecer os seguintes métodos para o componente `Joint`:

- `actuatorOn`: método que liga o acionamento na junta do robô.
- `actuatorOff`: método que desliga o acionamento na junta do robô.
- `actuatorSet`: método que aplica uma determinada entrada na junta do robô.
- `sensorRead`: método para solicitar o valor do sensor na junta do robô.
- `brakeApply`: método que aciona o freio eletromecânico na junta do robô.
- `brakeRelease`: método que libera o freio eletromecânico na junta do robô.

Para configurar o modelo `Joint`, de acordo com as especificações de cada junta do robô, existem as seguintes propriedades:

- `PeerName`: Nome do componente que irá fornecer os métodos de acesso ao *hardware* para este componente.

- `Brake`: Esta propriedade serve para indicar se uma determinada junta possui freio eletromecânico.

- `ActuatorSign`: Dada uma determinada construção de *hardware*, o motor girará sempre no mesmo sentido quando uma determinada tensão for aplicada. É de interesse que, dado um eixo de giro com um determinado sentido para uma junta, o motor se mova no sentido de giro positivo (definido pela a regra da mão direita) quando uma tensão positiva for aplicada ao motor. Este parâmetro serve para inverter ou não o sinal da tensão aplicado no motor, permitindo assim que o sentido de giro possa ser definido pelo o usuário.

- `SensorSign`: Similar ao `ActuatorSign`. Dada uma determinada construção de *hardware*, o *encoder* medirá com um determinado sinal quando o motor girar com um determinado sentido. Este parâmetro serve para inverter ou não o sinal do *encoder* lido no motor. Note que este parâmetro não é redundante com o `ActuatorSign` uma vez que o motor pode girar no sentido positivo do eixo e o *encoder* pode medir um valor negativo, pois o seu sinal depende da ordem dos canais A e B.

- `GearRatio`: Relação de engrenagens na junta do robô. Serve para transformar o deslocamento no eixo do motor (lido no *encoder*), em deslocamento e posição realizados na junta.

- `InitialPos`: Valor que contém a posição inicial da junta. Este valor serve como referência para o *encoder* incremental.

- `Sampler`: Esta propriedade faz com que o componente `Joint` tente se conectar aos eventos `Sample` e `Act`.

O modelo `Joint` também exporta para outros componentes métodos similares ao que ele recebe do componente `PeerName` que acessa o *hardware* do robô. Desta forma, é possível comandar a junta sem precisar comandar diretamente o *hardware* do robô. Os métodos `motorOn` e `motorOff` são diretamente mapeados nos métodos `actuatorOn` e `actuatorOff` importados. O mesmo ocorre com os métodos do freio, porém eles só serão exportados caso a propriedade `Brake` confirme a presença do freio na junta. Isto é útil, uma vez que caso a junta não possua freio, não tem sentido que o seu componente possua métodos para liberá-lo e acioná-lo. O método `motorSet` exportado é mapeado no método `actuatorSet` importado do componente de *hardware* de acordo com o valor da propriedade `ActuatorSign`, caso ela seja `true`, o argumento deste método é invertido. Por fim, o método `sensorRead` irá retornar o valor do deslocamento na junta do robô, utilizando as propriedades `SensorSign` e `GearRatio`, transformando o valor do deslocamento do eixo do motor, no valor do deslocamento na junta.

Em termos de porta de dados, o componente `Joint` possui três portas de dados para se conectar com o componente `PeerName` e mais quatro portas de dados para informar o *status* da junta. A porta `VoltageActuator` é para aplicar uma tensão no componente `PeerName`, isto é feito escrevendo um valor na mesma. A porta de dados `Voltage` serve para que a junta receba o valor de tensão que ela deverá aplicar

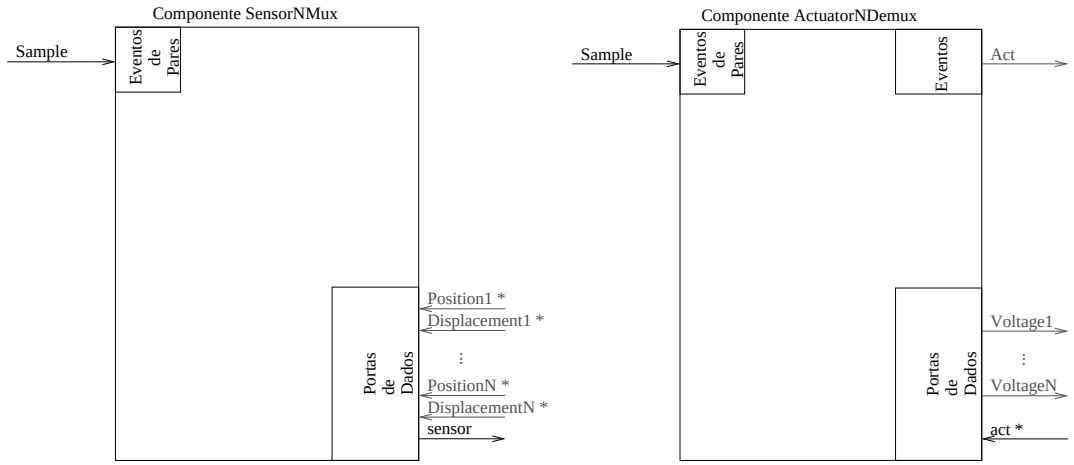
no seu motor. É esperado que esta operação seja mapeada no método `actuatorSet` pelo o componente que gerencia o *hardware* do robô.

A porta `DisplacementSensor*` serve para que o componente `PeerName` informe o deslocamento do motor, desde a última leitura. Essa porta está associada a uma *callback* que utilizará este valor, em conjunto com as propriedades `GearRatio` e `SensorSign` para atualizar o valor da porta `Displacement`, que contém o deslocamento da junta. Ela também integra o valor do deslocamento e utiliza a propriedade `InitialPos` para atualizar o valor da porta `Position`, que contém a posição da junta. De forma similar, a porta `IndexSensor` atualiza o valor da porta `Index`. É esperado que o método `sensorRead` importado, atualize as portas `IndexSensor` e `DisplacementSensor*`.

O evento `Sample` serve para gerar uma chamada do método `encoderRead`, fazendo com que as portas de dados `Displacement`, `Position` e `Index` sejam atualizadas. O evento `Act`, irá realizar uma leitura na porta `Voltage`, para depois realizar uma escrita na porta `VoltageActuator`. Esses eventos permitem a leitura e o acionamento da junta de forma sincronizada e paralela.

3.3 Extensão dos Componentes `Sensor` e `Actuator` com o Componente `Joint`

Para integrar os componentes `Joint`s representando as juntas, na arquitetura proposta na Figura 11, é necessário implementar modelos de componentes derivados de `Sensor` e de `Actuator` de forma a criar o vetor `sensor` com os sensores do robô e desagrupar o vetor `act*` com os valores dos atuadores. Isso é necessário porque os componentes `Sensor` e `Actuator` são componentes que não contém implementação específica. As suas interfaces são mostrada na Figura 13.



(a) `SensorNMux`. (b) `ActuatorNDemux`.

Figura 13: Extensões dos componentes `Sensor` e `Actuator`.

O componente `SensorNMux`, Figura 13(a), tem por finalidade agrupar as $2N$ portas de dados, de acordo com o vetor (2). O valor N é definido em tempo de compilação, e pode ser qualquer número positivo. Assim, o componente `SensorNMux` irá criar N portas de posição (`Position1*`,...,`PositionN*`) e N portas de deslocamento (`Displacement1*`,...,`DisplacementN*`). O `*` nas portas indica que uma escrita nelas

irá gerar um evento que disparará uma *callback*. Esta servirá para armazenar o valor escrito na porta. Quando todas as $2N$ portas tiverem sido escritas e após o evento `Sample` ter ocorrido, a porta `sensor` é atualizada. Caso uma das $2N$ portas não for escrita, ou ainda, caso o evento `Sample` não tenha ocorrido após uma escrita na porta `sensor`, a porta `sensor` não é atualizada.

O componente `ActuatorNDemux`, Figura 13(b), tem por finalidade desagrupar o vetor `act` nas N portas `Voltage`. A função virtual `act_now()` é reimplementada no `ActuatorNDemux` de forma a quebrar o vetor recebido pela porta `act` nas suas N portas de tensão (`Voltage1`,...,`VoltageN`). A seguir, caso o evento `Sample` tenha acontecido, desde a última geração do evento `Act`, o evento `Act` é gerado. Este evento serve para que todos os atuadores realizem a atuação no mesmo instante de tempo.

Os componentes `SensorNMux` e `ActuatorNDemux` são feitos de forma genérica para agrupar/desagrupar os dados dos sensores/atuadores respectivamente. Neste trabalho, eles serão utilizados em conjunto com N componentes `Joint`s, de acordo com a Figura 14. O componente `CardN` é um componente que implementa os métodos importados pelo componente `Joint`. Os componentes `Joint1`, ..., `JointN`, `Actuator2Demux` e `Sensor2Mux` conectados como pares do componente `Sampler` para reagir ao evento `Sample`, bem como os componentes `JointN` são conectados como par dos respectivos componentes `CardN` para garantir o acesso ao *hardware*. As linhas tracejadas indicam estas conexões. As linhas cheias mostram os fluxos nas portas de dados.

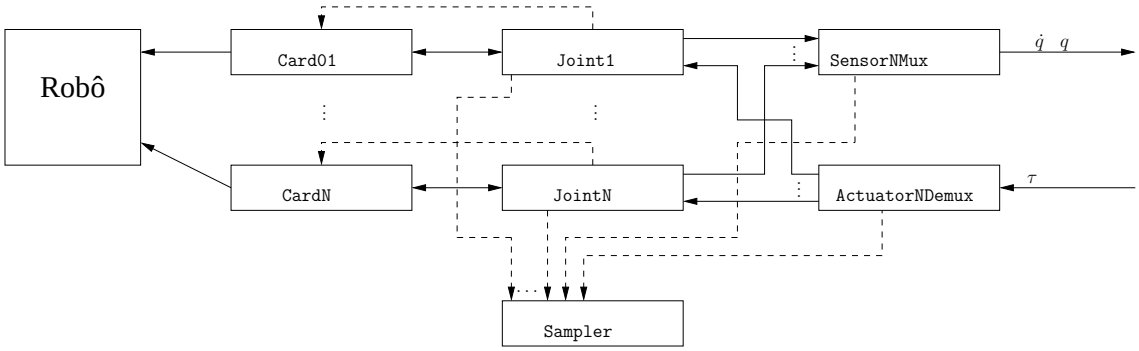


Figura 14: Diagrama de blocos das extensões `ActuatorNDemux` e `SensorNMux` com os componentes `Joint`.

É importante ressaltar que esses componentes são genéricos e podem ser aplicados para qualquer robô, independentemente do número de juntas, apenas recompilando o modelo do componente.

Um exemplo para uma extensão para um robô de duas juntas é apresentado em detalhes na Figura 15. Para esse sistema é necessário a criação dos seguintes componentes: `Card01`, `Card02`, `Joint1`, `Joint2`, `Sensor2Mux`, `Actuator2Demux` e `Sampler`.

Para facilitar o entendimento, explica-se o funcionamento do sistema para um ciclo equivalente a uma leitura de dados do robô, e uma escrita de tensão no motor. Neste exemplo considera-se que os componentes possuem uma prioridade e uma atividade definidos conforme a Tabela 2, onde a prioridade 1 é maior do que a prioridade 2.

Quando o componente `Sampler` for executado, o evento `Sample` será emitido e,

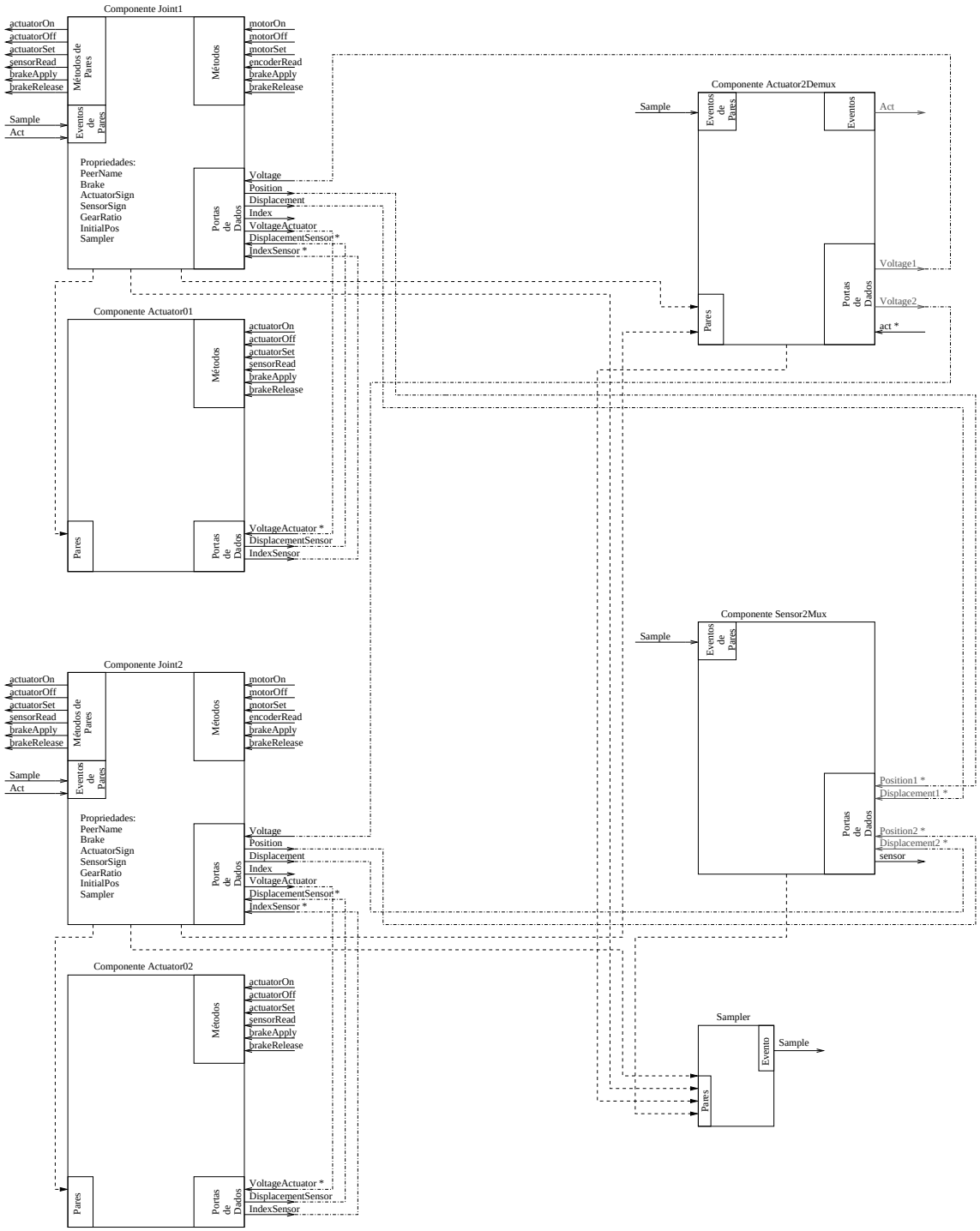


Figura 15: Sistema de duas juntas com `Sensor2Mux` e `Actuator2Demux`.

Tabela 2: Prioridade e atividade dos componentes.

Componente	Atividade	Prioridade
`Sampler`	`NonPeriodicActivity`	1
`Card01`	`SequentialActivity`	—
`Card02`	`SequentialActivity`	—
`Joint1`	`SequentialActivity`	—
`Joint2`	`SequentialActivity`	—
`Sensor2Mux`	`NonPeriodicActivity`	2
`Actuator2Demux`	`NonPeriodicActivity`	2

instantaneamente irá ocorrer o seguinte: os componentes `Actuator2Demux` e `Sensor2Mux` irão marcar uma *flag* indicando que o evento ocorreu e os componentes `Joint`s irão chamar os métodos `encoderRead` das suas respectivas `Card0x`s. Estes métodos irão atualizar as portas `Index`, `Position*` e `Displacement*`. A escrita na porta `Displacement*` irá ocasionar que o componente `Sensor2Mux` execute quando não houver nenhum outro componente com maior prioridade do que ele executando. A partir desde momento, o componente `Sensor2Mux` irá executar todos os *callbacks* causados pelas as portas `Position1*`, `Position2*`, `Displacement1*` e `Displacement2*`. Quando o último deles for executado, o `Sensor2Mux` detectará que todas as portas foram atualizadas e que o evento `Sample` ocorreu, e assim, atualizará a sua porta `sensor`.

Quando uma escrita na porta `act*` do componente `Actuator2Demux` ocorrer, isto irá causar a execução da *callback* associada a esta porta, de acordo com as prioridades do sistema. Na sua execução ela irá verificar que o `Sample` ocorreu, e então, ele escreverá as respectivas tensões nas portas `Voltage1` e `Voltage2` para seguir emitir o evento `Act`. Este evento será atendido pelos componentes `Joint`s de forma síncrona, fazendo com que uma escrita na porta `VoltageCard` ocorra.

Note que caso uma segunda escrita a porta `act*` ocorra, nada irá acontecer, pois este componente irá esperar o evento `Sample`.

3.4 Extensão do Componente `Controller` para um PID Independente por Junta

Da mesma forma que os componentes `Sensor` e `Actuator` da Figura 11, o componente `Controller` também necessita ser estendido para implementar a lei de controle desejada. Nesta seção o componente `Controller` é estendido para implementar um controlador PID independente para cada junta do robô.

Inicialmente será implementado um modelo de componente PID genérico, para que ele possa ser instanciado diversas vezes, com diferentes ganhos, no mesmo sistema, além de ser utilizando posteriormente em outras estratégias de controle.

3.4.1 Modelo de Componente de `PID`

Um controlador PID com saturação envolve três ganhos separados: O ganho proporcional K_p, o ganho integral K_i e o ganho derivativo K_d. A saturação é representada pelo sinal $\underline{u}/\overline{u}$, que significa o valor mínimo/máximo admitido na entrada da planta. O valor de erro $e(t)$ é a diferença entre o valor de referência e o valor da

saída. Ele é representado pela a equação:

$$\begin{cases} u(t) = K_p e(t) + K_i \int_0^t e(t)dt + K_d \dot{e}(t) & , u(t) \leq \overline{u} \text{ e } u(t) \geq \underline{u} \\ u(t) = \overline{u} & , u(t) > \overline{u} \\ u(t) = \underline{u} & , u(t) < \underline{u} \end{cases} \quad (3)$$

Uma das formas discretas do algoritmo PID é (ASTROM; WITTENMARK, 1984; HEMERLY, 1996):

$$\begin{cases} u[k] = u[k-1] + k_p(e[k] - e[k-1]) + k_i e[k] + k_d(e[k] - 2e[k-1] + e[k-2]) & , u[k] \leq \overline{u} \text{ e } u[k] \geq \underline{u} \\ u[k] = \overline{u} & , u[k] > \overline{u} \\ u[k] = \underline{u} & , u[k] < \underline{u} \end{cases} \quad (4)$$

Esta equação apresenta uma resposta melhor que uma simples discretização de (3) para sistemas com saturação, pois evita *windup* do controlador. A Figura 16 apresenta a interface do componente PID.

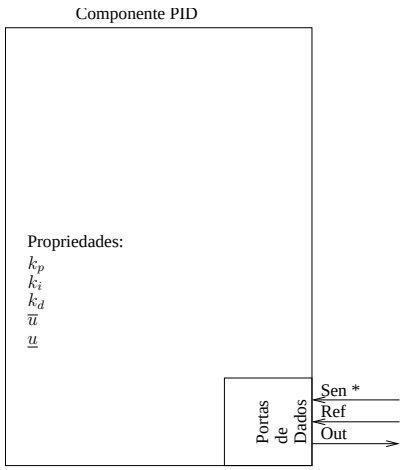


Figura 16: Modelo de componente `PID`.

Para não perder generalidade, todos os parâmetros dos PID são propriedades. Isto permite que diversos PIDs diferentes sejam criados a partir deste modelo. Esse modelo se comunica com outros componentes através das suas portas de dados. A porta de dado `Ref` contém o valor da referência do PID. A porta de dado `Sen*` recebe o valor do sensor, e por sua vez, está associada a uma *callback*, de forma que quando um valor é escrito nesta porta o componente é ativado.

Ao ser ativado, o componente irá ler o valor das suas portas `Ref` e `Sen*`, para então calcular o valor de u em (4). A seguir o valor é escrito na porta `Out`.

3.4.2 Modelo de Componente de `ControllerNPID`

Este modelo de componente tem por objetivo estender o `Controller` para realizar o controle através de N PIDs. A Figura 17 apresenta a interface do componente `ControllerNPID`.

Este componente adiciona ao `Controller` as portas necessárias para se comunicar com N componentes `PID`, onde N é um parâmetro definido em tempo de compilação. Ele reimplementa a função virtual `controller_now()` de forma a quebrar os vetores `reference` e `sensor`, para então escrever os valores nas respectivas $2N$ portas de dados (`Ref1`,...`RefN`) e (`Sen1`,...`SenN`). Uma *callback* também é

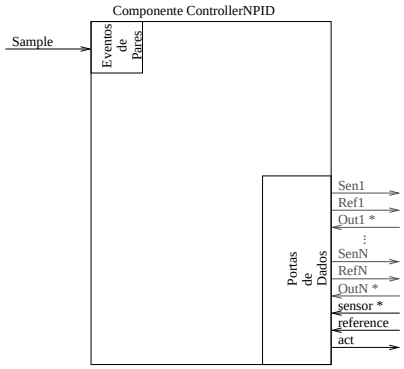


Figura 17: Modelo de componente `ControllerNPID`.

chamada quando um dado for escrito em qualquer uma das N portas de dados (`Out1`,...`OutN`). Essa *callback* tem um funcionamento similar a *callback* das portas (`Position1*`,...,`PositionN*`) do componente `SensorNMux`, e irá agrupar as saídas do PID no vetor `act`, somente atualizando o vetor `act` quando todas as portas forem atualizadas e o evento `Sample` tenha ocorrido. O diagrama de blocos do sistema é apresentado na Figura 18.

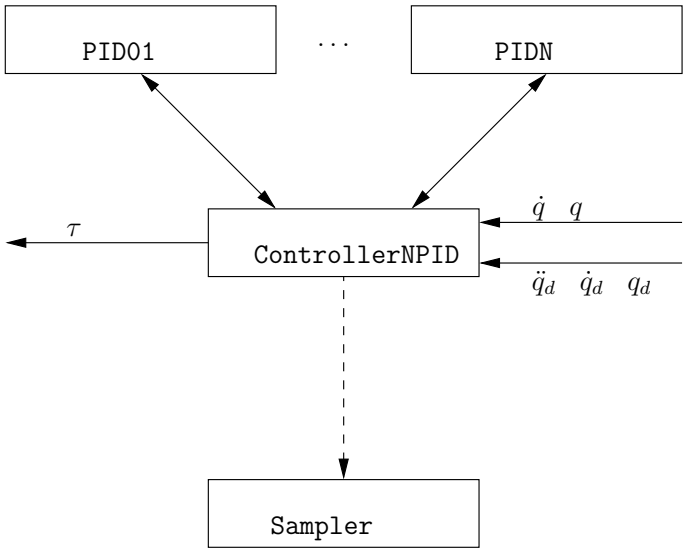


Figura 18: Diagrama de blocos da extensão `ControllerNPID`.

A Figura 19 demonstra a conexão de um sistema com dois controladores PID. Inicialmente é necessário conectar o `Controller2PID` ao `Sampler` para que ele possa ter acesso ao evento `Sample`. A seguir, as respectivas portas de dados do `PID1` e do `PID2` devem ser conectados com as portas de dados do `Controller2PID`.

Para facilitar o entendimento, explica-se o funcionamento do sistema para um ciclo equivalente ao calculo da lei de controle para um robô de duas juntas. Neste exemplo considera-se que os componentes possuem uma prioridade e uma atividade definidos conforme a Tabela 3.

Quando o `Sampler` dispara o evento `Sample`, uma *flag* é marcada no `Controller2PID` para indicar esta ocorrência. A seguir, quando um dado for escrito na porta `sensor`, a função `controller_now()` fará com que a porta `reference` e `sensor` sejam lidas. Então a porta `Ref1` é escrita, a seguindo pela a porta `Sen1`. Esta última

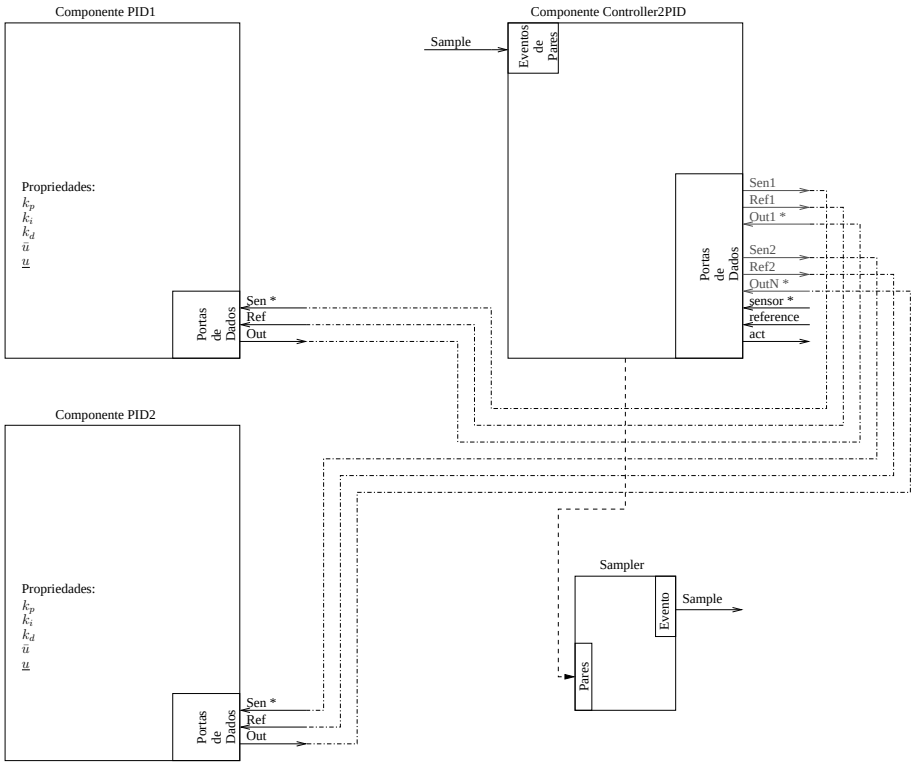


Figura 19: Sistema de dois controladores PID com `Controller2PID` e `PID`.

escrita acionará o componente `PID`, fazendo com que ele escreva o valor da saída de controle na sua porta `Out`. Essa escrita fará com que uma *callback* seja chamada futuramente. A seguir o mesmo se repete com as portas `Ref2` e `Sen2`. A seguir, as duas *callbacks* são executadas e a porta `act` é escrita pelo `Controller2PID`.

Tabela 3: Prioridade e atividade dos componentes.

Componente	Atividade	Prioridade
`Sampler`	`NonPeriodicActivity`	1
`PID1`	`SequentialActivity`	—
`PID2`	`SequentialActivity`	—
`Controller2PID`	`NonPeriodicActivity`	2

3.5 Extensão do Componente `Controller` para um Controle de Torque Calculado

Esta seção tem por objetivo estender o componente `Controller` para implementar um controlador de torque calculado (FU; GONZALES; LEE, 1987). Essa estratégia é representada em blocos na Figura 20. A sua ideia consiste em utilizar o modelo do robô para anular as suas não linearidades, como por exemplo a gravidade. Ao fazer cancelamento exato das não linearidades, pode-se impor a dinâmica desejada respeitando-se o grau relativo do sistema. Como tal cancelamento não ocorre na realidade, um PID é adicionado ao sistema para corrigir eventuais diferenças entre o modelo e o robô, bem como as perturbações do sistema.

Para esta estratégia de controle, é conveniente reescrever o modelo do robô manipulador (1) como:

$$\dot{x} = f(x) + g(x)u \tag{5}$$

onde $x = \begin{bmatrix} q & \dot{q} \end{bmatrix}$ representa o estado do robô e $u = \tau$ representa a entrada de torque do robô, sendo $f(x) = -D(q)^{-1}\left(H(q,\dot{q}) + G(q)\right)$ e $g(x) = D(q)^{-1}$.

Desta forma, é possível fazer uma linearização do modelo utilizando uma realimentação de estado, colocando uma entrada adequada como:

$$u = g^{-1}(x)(v - f(x)) \tag{6}$$

onde v é uma entrada qualquer.

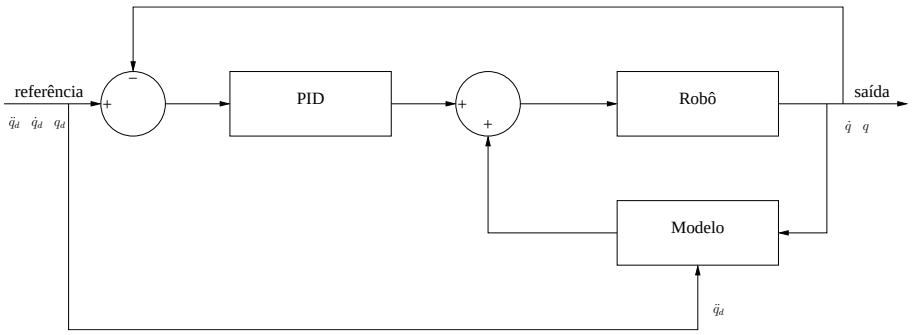


Figura 20: Estratégia de controle de torque calculado.

Substituindo (6) em (5), o modelo torna-se linear:

$$\dot{x} = v \tag{7}$$

É fácil perceber, que para robôs manipuladores, $g^{-1}(x)$ é equivalente a matriz $D(q)$ de (1), que por sua vez, é sempre não singular (FU; GONZALES; LEE, 1987).

Da Figura 20 percebe-se que neste caso, a entrada do robô é constituída de uma parcela relativa ao modelo, mais uma parcela relativa ao PID, segundo (6). Considerando que o modelo do robô possui os parâmetros $D_a(q)$, $H_a(q,\dot{q})$, e $G_a(q)$, o torque aplicado na entrada de um robô pode ser rescrito como:

$$\tau = D_a(q)\ddot{q}_d + H_a(q,\dot{q}) + G_a(q) + \begin{bmatrix} K_{p1} & \dots & K_{pN} \end{bmatrix}(q_d - q) \tag{8}$$
$$+ \begin{bmatrix} K_{i1} & \dots & K_{iN} \end{bmatrix}\left(\int_0^t q_d dt - \int_0^t q dt\right) + \begin{bmatrix} K_{d1} & \dots & K_{dN} \end{bmatrix}(\dot{q}_d - \dot{q})$$

A Equação (8) pode ser igualada a (1), onde caso os parâmetros do modelo sejam iguais aos parâmetros reais do robô, obtém-se:

$$D(q)(\ddot{q}_d - \ddot{q}) + \begin{bmatrix} K_{p1} & \dots & K_{pN} \end{bmatrix}(q_d - q) + \begin{bmatrix} K_{i1} & \dots & K_{iN} \end{bmatrix}\left(\int_0^t q_d dt - \int_0^t q dt\right) \tag{9}$$
$$+ \begin{bmatrix} K_{d1} & \dots & K_{dN} \end{bmatrix}(\dot{q}_d - \dot{q}) = 0$$

Esta equação ainda pode ser reescrita como:

$$D(q)\dddot{e} + K_d\ddot{e} + K_p\dot{e} + K_i e = 0 \tag{10}$$

onde $e = q - q_d$ e os ganhos K do controlador podem ser escolhidos de forma que o erro tenda assintoticamente para zero, dado que $D(q)$ é positiva definida (SCIAVICCO; SICILIANO, 2005).

A implementação desta estratégia no OROCOS segue o diagrama de blocos da Figura 21. O bloco PID apresentando na subseção 3.4.1, é reutilizado nesse caso. Os blocos `Model` e `ControllerNPIDCT` são implementados nesta seção.

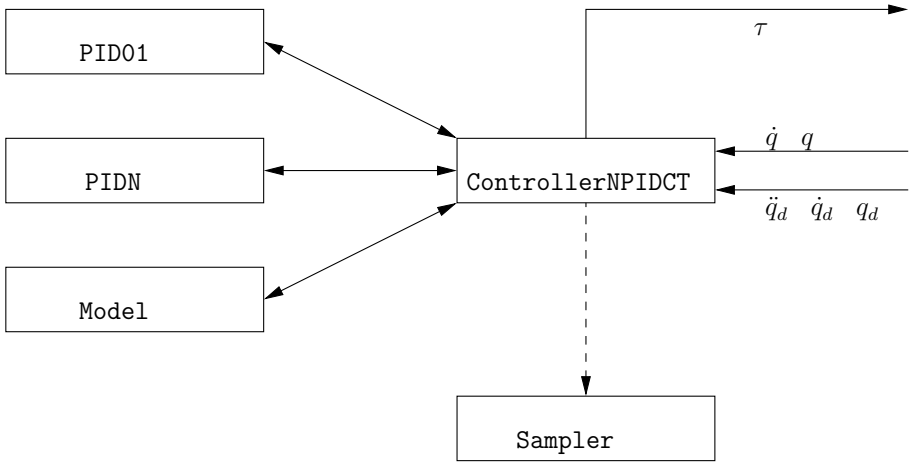


Figura 21: Diagrama de blocos do controlador de torque calculado.

3.5.1 Modelo de Componente de `Model`

Este componente tem a finalidade de calcular um modelo genérico, que a partir de uma entrada, gera uma saída. Nesta seção ele terá a finalidade de calcular o modelo inverso de um robô, ou seja, a partir da posição, da velocidade e da aceleração das juntas do robô, ele deverá gerar o torque aplicado para atingir esta condição, satisfazendo (1). Esse modelo de componente é implementado como um componente base, similar a `Controller`, e é apresentado na Figura 22. As portas de dados `In*` e `Out` são compostas por `vector` de *double*, onde o tamanho é definido no construtor do componente. Uma escrita na porta `In*` irá chamar a função virtual `model_now()`. Esta função deve ser reimplementada na extensão do componente `Model`. Ela deverá ser responsável por calcular o modelo inverso do robô, e colocar o valor do torque na sua porta `Out`.

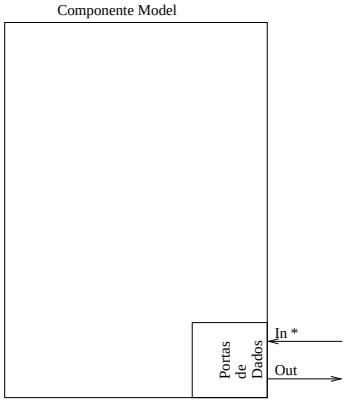


Figura 22: Modelo de componente `Model`.

3.5.2 Modelo de Componente de `ControllerNPIDCT`

Este modelo de componente tem por objetivo estender o `Controller` para realizar o controle através de N PIDs utilizando uma estratégia de torque calculado. A Figura 23 apresenta a interface do componente `ControllerNPIDCT`.

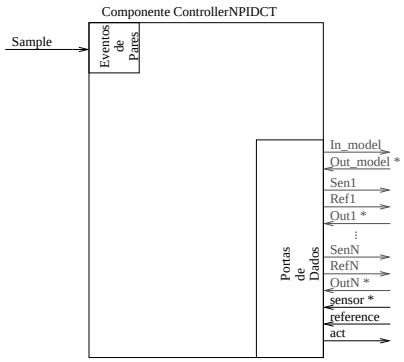


Figura 23: Modelo de componente `ControllerNPIDCT`.

Este componente possui uma implementação similar ao `ControllerNPID`, porém com as portas `In_model` e `Out_model`. A função virtual `controller_now` é reimplementada para colocar a posição e a velocidade angular medida do robô, junto com a aceleração angular desejada, na porta `In_model`. A seguir, a *callback* que é chamada quando um dado é escrito nas portas `OutN*` é modificada para ser chamada quando um dado for escrito na porta `Out_model*`. Essa *callback* irá quebrar o vetor da porta `Out_model*` e somar com os valores recebidos nas portas `OutN*` e quando todos os valores tiverem sido atualizados, ela irá escrever os dados na porta `Act`, finalizando assim, o ciclo de controle.

3.6 Extensão do Componente `Controller` para com `FeedForward`

A estratégia de *FeedForward* é similar ao controle de torque calculado, com a diferença que o modelo é alimentado pela própria referência do robô. A Figura 24 demonstra o diagrama de blocos dessa estratégia.

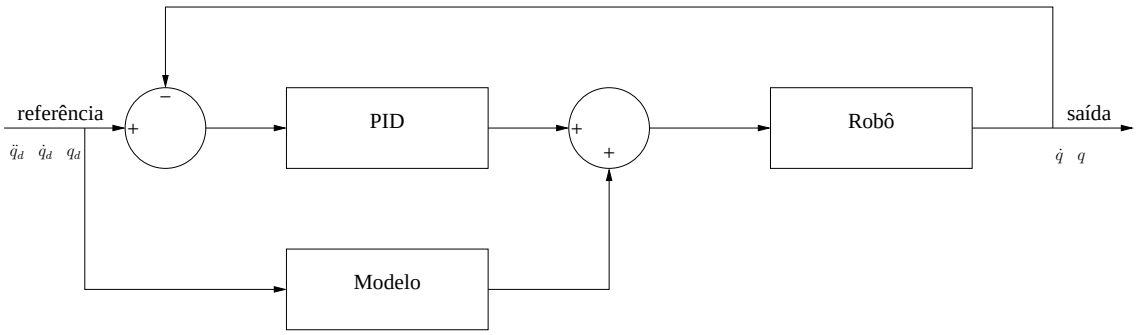


Figura 24: Estratégia de controle com `FeedForward`.

Neste caso, é conveniente escrever o modelo do robô na forma:

$$\dot{x}(t) = f(x(t), u(t)) \tag{11}$$

Supondo que os estados $x_d(t)$ são conhecidos ao longo de uma trajetória desejadas, o torque $u_d(t)$ também é conhecido, pode ser calculado por (1) e satisfaz:

$$\dot{x}_d(t) = f_a(x_d(t), u_d(t)) \tag{12}$$

onde $f_a(\cdot,\cdot)$ é o modelo do robô.

Subtraindo de (12) de (11), obtém-se

$$\delta\dot{x}(t) = h(x(t), x_d(t), u(t), u_d(t)) \tag{13}$$

onde $\delta x(t) = x(t) - x_d(t)$ e $h(\cdot,\cdot,\cdot,\cdot)$ representa o modelo gerado pela a diferença entre o robô $f(\cdot,\cdot)$ e o modelo utilizado no cálculo $f_a(\cdot,\cdot)$.

Caso o estado atual do robô x seja próximo do estado desejado x_d, o erro $\delta x(t)$ tenderá a ser pequeno e poderá ser tratado por um controlador PID, o qual fará com que $\delta x(t)$ tenda a zero.

Note que diferentemente da estratégia de torque calculado, é necessário que as posições e as velocidades desejadas do robô estejam sempre próximas as posições e as velocidades medidas.

A implementação desta estratégia no OROCOS segue a diagrama de blocos da Figura 21, onde o componente `ControllerNPIDCT` é substituído pelo componente `ControllerNPIDFF` e os demais blocos já foram implementados anteriormente.

3.6.1 Modelo de Componente de `ControllerNPIDFF`

Este modelo de componente estende o `Controller` para realizar o controle através de N PIDs utilizando uma estratégia com *FeedForward*. A sua interface é igual ao componente `ControllerNPIDCT` da Figura 23. A única diferença consiste na função virtual `controller_now`. Essa função irá colocar na porta `In_model` as posições, velocidades e acelerações desejadas, ao invés das posições e velocidades medidas. No restante, o funcionamento deste componente se assemelha ao componente `ControllerNPIDCT`.

3.7 Conclusão

Neste capítulo foi apresentada uma arquitetura para controle de robôs manipuladores. A definição dos blocos da Figura 11 torna o sistema modular e independente, separando as funções de controle, sensoriamento e atuação do sistema. Esta forma de implementação permite que qualquer parte do sistema seja substituída sem que as outras partes sejam refeitas, permitindo que o usuário se foque exatamente na parte que se deseja desenvolver. O bloco `Sampler` fornece o sincronismo, quando desejado na implementação do sistema.

Em um grau de liberdade abaixo, os componentes `SensorNMux` e `ActuatorNDemux` fornecem uma interface para N juntas de um robô, desde que a interface com o *hardware* respeite a interface do componente `Joint`. Além disso, eles servem de exemplos para futuras extensões dos componentes `Sensor` e `Actuator`.

Para o componente `Controller`, foram apresentadas três extensões diferentes que implementam três estratégias de controle. Nota-se que a medida que os componentes estão desenvolvidos, a implementação do sistema de controle torna-se mais fácil, sendo que na estratégia de controle da seção 3.6 utiliza os componentes implementados nas seções 3.4 e 3.5.

As portas da Figura 11 não foram conectadas para deixar o sistema versátil. Por exemplo, caso se deseje operar um sistema em malha aberta, pode-se não implementar o componente `Controller`, aplicar a atuação diretamente na porta `act*` do componente `Actuator` e medir a saída através do componente `Sensor`.

Por fim, a utilização do *framework* OROCOS nesta parte do projeto facilitou esta implementação, diminuindo o tempo deste trabalho.

4 PLACA AIC (ACTUATOR INTERFACE CARD)

As juntas de um robô manipulador são compostas de atuadores e sensores. Os atuadores têm a função de realizar um movimento na junta, enquanto os sensores são os responsáveis por medir este movimento. Uma forma muito comum de atuador em robôs manipuladores são os motores de corrente contínua (motores DC). Este tipo de motor é fácil de ser controlado, em virtude de ser bem representado por um modelo linear, além de apresentar um movimento suave, e uma boa relação torque/volume.

Um sensor típico para motores é o *encoder* incremental de quadratura. Este instrumento permite medir o deslocamento e o sentido de giro de um motor. Ele consiste em um disco furado acoplado ao eixo do motor, com dois sensores ópticos acoplados e posicionados com uma defasagem de 90 graus elétricos entre si. Essa defasagem permite obter uma resolução quatro vezes maior que o número de furos do disco.

Com a finalidade de atuar e medir sobre uma junta de um robô manipulador, foi desenvolvida uma placa de acionamento nomeada AIC. Essa placa é baseada em um microcontrolador dsPIC (MICROCHIP, 2006) e será utilizada na implementação da arquitetura proposta no Capítulo 3.

Com finalidade servir como uma ponte entre as juntas de um robô manipulador e um computador *host*, a placa AIC terá a função de interpretar os comandos enviados do computador e realizar a operação correspondente. As mensagens de comando são enviadas utilizando um barramento CAN (BOSCH, 1991), como mostra a Figura 25. O barramento CAN é utilizado para troca de dados em tempo real, pois ele apresenta baixo custo, um bom desempenho e uma alta popularidade, além de ser um protocolo aberto (XUEMEI; LIANGZHONG, 2007).

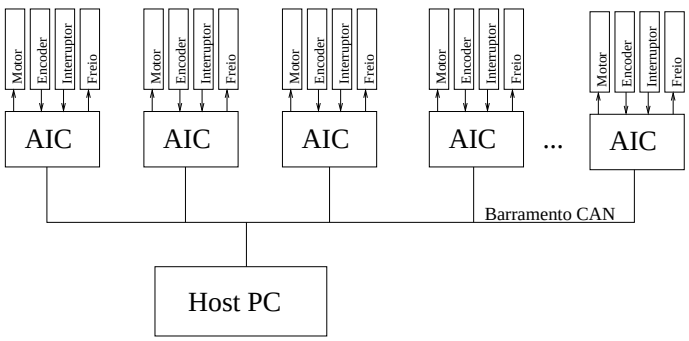


Figura 25: Topologia de comunicação com as placas AICs.

Para não limitar a AIC à esta aplicação, todas as funções são implementadas de forma genérica e separadas em bibliotecas. Desse modo, a implementação de

novas funcionalidades fica facilitada. Por exemplo, o desenvolvimento de um PID na própria placa AIC é uma alternativa que pode ser desenvolvida sem dificuldades.

Este capítulo trata em detalhes a implementação da placa AIC. A seção 4.1 apresenta as características necessárias para a implementação da comunicação da placa AIC com o computador *host* no barramento CAN . A seção 4.2 apresenta os detalhes do *hardware* da placa, enquanto a seção 4.3 apresenta o *software*.

4.1 Protocolo CAN (*Controller Area Network*)

O protocolo CAN foi originalmente desenvolvido pela a Bosch, em 1985, para ser utilizado como meio de comunicação entre componentes de veículos automotivos. A sua especificação pode ser encontrada em (BOSCH, 1991). O protocolo foi amplamente aceito pela indústria e a sua padronização se deu através da ISO 11898 em 1993. Outras variações específicas deste protocolo foram padronizadas, como a ISO 11898-2 para um barramento CAN de alta velocidade e a ISO 11898-3 para para um barramento CAN de baixa velocidade tolerante à falhas.

O CAN é um protocolo *broadcast*, onde todos os nós são conectados a um barramento e, por consequência, recebem as mesmas mensagens de forma simultânea. Através de filtros de *hardware* é possível selecionar as mensagens recebidas pelo nó, tornando-o um barramento *multicast*. Ele necessita de somente um canal de dados para transmitir os seus bits, sendo que os valores desses bits são: recessivos ou dominantes.

Uma das características mais importantes do protocolo CAN é a dominância de bit no barramento. O seu conceito pode ser representado com nós conectados a um barramento, onde suas saídas utilizam uma configuração de coletor aberto utilizando um resistor de *pull-up*, conforme representado na Figura 26. Neste caso, o valor recessivo corresponde ao VCC, enquanto o valor dominante corresponde ao GND. Somente quando os transistores dos nós A e B estiverem cortados, o barramento possuirá o valor recessivo. Quando um dos dois transistores passar para o estado saturado, o valor do barramento passará para dominante, independentemente do valor do outro nó.

A Figura 26 também mostra um exemplo dos nós A e B transmitindo simultaneamente. Enquanto os nós transmitem o mesmo valor, não há nenhum problema e o barramento assume este valor. No momento em que os dois nós transmitem bits diferentes, o barramento sempre terá o valor dominante.

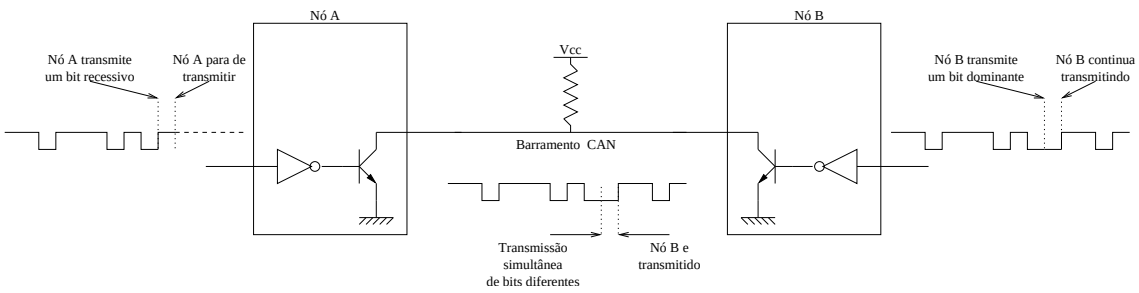


Figura 26: Característica da dominância em um barramento.

Este conceito é importante, uma vez que ele permite associar prioridades às mensagens e não aos nós. Desta forma é possível garantir que mensagens vitais são

enviadas em detrimento de mensagens de menor importância. O barramento CAN utiliza tal conceito para realizar uma arbitragem não destrutiva de mensagens. Ele faz isto ao sincronizar os envios de mensagens e inserir um campo de arbitragem no início do quadro. Desta forma, cada estação fica monitorando o barramento durante a transmissão do seu bit, e caso ela detecte que o barramento está com um bit diferente do qual ela está enviando, a estação irá cancelar a sua transmissão. Assim o barramento CAN lida com a colisão de mensagens sem destruir todas as mensagens que colidiram e sem perder tempo no barramento. No caso da Figura 26, a mensagem do nó B tem uma maior prioridade do que a do nó A, e portanto, ela é transmitida no barramento, enquanto a mensagem do nó A fica aguardando o barramento ser liberado.

Grande parte do desempenho do protocolo CAN está na sua capacidade de detectar erros. O protocolo CAN estabelece quatro tipos de detecção de erros: monitoramento (o transmissor compara o bit que ele está transmitindo com o bit que está sendo recebido no barramento), verificação de CRC (*Cyclic Redundancy Check*) utilizando um polinômio de grau 15, inserção de *bit stuffing* durante a mensagem e, por fim, verificação de quadro no recebimento da mensagem. Desta forma, a chance de um erro passar despercebido pelo o barramento é de $4.7 \times 10^{-11} f_e$, onde f_e é a taxa de erros (BOSCH, 1991).

Quando um nó do barramento detecta um erro em uma mensagem sendo enviada, ele pode informar a todos os outros nós que o erro ocorreu. Essas mensagens corrompidas são então automaticamente retransmitidas pelo transmissor. Os próprios nós do barramento conseguem auto-detectar problemas ou falhas críticas, e neste caso, podem se desconectar do barramento.

A taxa de comunicação no barramento CAN pode variar entre sistemas, porém para um determinado sistema, ela é fixa. O tamanho das mensagens também pode ser variável, porém finito e qualquer nó conectado ao barramento pode iniciar a transmissão de uma mensagem.

4.1.1 Tipos de Quadros

O protocolo CAN apresenta três tipos básicos de quadros:

- Quadro de Dados: tipo de quadro utilizado para transportar dados;
- Quadro Remoto: tipo de quadro utilizado para requisitar a transmissão de um quadro de dados de um outro nó;
- Quadro de Erro: quadro que sinaliza um erro;

Nesta seção, considera-se que o bit dominante é representado como 0 e que o bit recessivo é representado como 1 para simplificar o raciocínio. Neste trabalho é utilizada a versão 2.0A do protocolo CAN, e somente os tipos de quadros desta versão são explicados.

4.1.1.1 Quadro de Dados

O quadro de dados é gerado por um nó, quando ele desejar transmitir alguma informação. A Figura 27 apresenta o quadro de dados. Como todos os outros quadros, o quadro de dados começa com um bit de início de quadro SOF (*Start-Of-Frame*). Este bit, dominante, só pode ser colocado no barramento quando o

mesmo estiver ocioso, ou seja, quando nenhum quadro estiver sendo transmitindo. Ele também serve para informar a todos os nós que um quadro vai começar a ser transmitido.

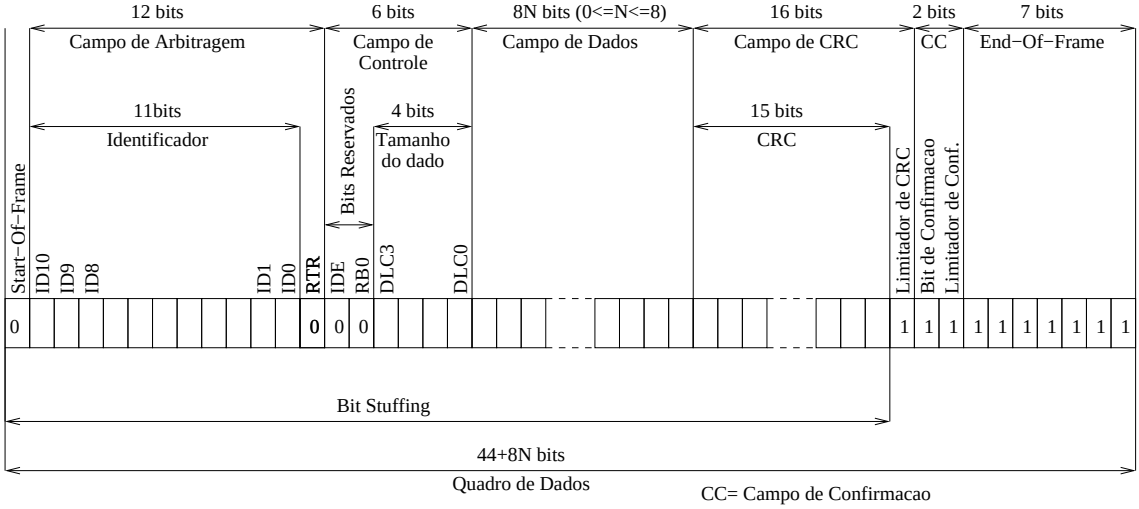


Figura 27: Quadro de dados padrão do CAN.

A seguir do SOF, tem-se o campo de arbitragem. Este campo tem uma grande importância, pois com o SOF, todos os nós que tiverem uma mensagem para transmitir irão começar a transmitir simultaneamente. Neste caso, enquanto as duas mensagens forem iguais, não há problema. A partir do momento que elas foram diferentes, segundo a característica do CAN, a mensagem que tiver o bit dominante irá ganhar a arbitragem e continuará sendo transmitida. De modo correspondente, a mensagem que tiver o bit recessivo, irá ter um erro de monitoramento acusando que ela perdeu a arbitragem e irá interromper a sua transmissão. O nó desta mensagem irá esperar o barramento ficar livre para retransmiti-lá .

O campo de arbitragem é composto por 11 bits chamados de Identificador, e 1 bit chamado RTR. Os 11 bits servem para definir a prioridade da mensagem e identificar o seu significado (isto possibilita 2048 mensagens diferentes). Caso sejam necessários mais identificadores, a versão 2.0B do protocolo deve ser utilizada. O protocolo CAN permite a possibilidade de selecionar as mensagens que são recebidas utilizando filtros e máscaras nestes bits. O bit RTR serve para indicar se este quadro é remoto ou de dados, e neste caso deve ser um bit dominante.

O próximo campo é utilizado para o controle do quadro. Ele começa com dois bits dominantes que estão reservados para futuras expansões, e mais 4 bits que indicam o tamanho do dado a ser transportado nesta mensagem. Eles podem variar de *dddd* a *rddd* indicando de 0 a 8 *bytes* respectivamente.

Depois do campo de controle, vem o campo de dados, cujo o tamanho dependerá do campo de controle. Ele terá $8N$ bits, onde N varia de 0 a 8, com os bits mais significativos transmitidos em primeiro lugar. Para dar mais robustez ao protocolo tem-se um campo de CRC, que é composto por um CRC de 15 bits e um bit recessivo (limitador de CRC) para finalizar o fim deste campo e dar tempo para que os receptores verifiquem se o CRC recebido está correto.

Para indicar que a mensagem foi entregue, existe o campo de confirmação. Ele consiste em 2 bits recessivos, sendo que o primeiro bit deve ser sinalizado como dominante pelos nós que receberam a mensagem sem erro de CRC. E o último bit

(limitador de confirmação) deve ser sempre recessivo para sinalizar o término do campo de confirmação.

Por fim, uma sequência de 7 bits recessivos sinalizam o final do quadro EOF (*End-Of-Frame*). Note que todos os campos, com exceção do campo de reconhecimento e do EOF, estão sujeitos a *bit-stuffing* no sexto bit, ou seja, quando cinco bits do mesmo nível forem transmitidos, o sexto bit possuirá o valor contrário aos outros cinco.

4.1.1.2 Quadro Remoto

O quadro remoto tem a finalidade de solicitar que um determinado nó envie uma informação através de um quadro de dados. O quadro remoto possui os seguintes campos do quadro de dados: SOF, campo de arbitragem, campo de controle, campo de CRC, campo de confirmação e EOF. O quadro remoto é demonstrado na Figura 28.

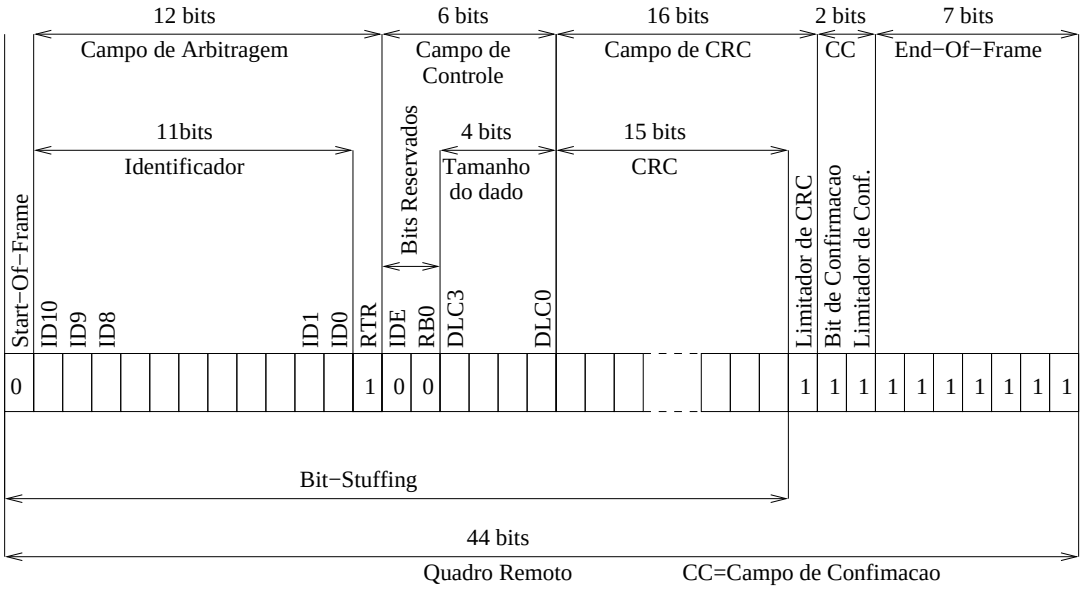


Figura 28: Quadro remoto do CAN.

As diferenças entre o quadro de dados e o remoto, é que o quadro remoto deve enviar um bit recessivo no RTR do campo de arbitragem, e que o identificador tem a finalidade de informar qual é o quadro de dados que deve ser enviado, e os 4 bits de tamanho do campo de controle devem conter o tamanho dos dados do quadro de dados.

Assim como o quadro de dados pode ser o padrão ou o estendido, o quadro remoto também suporta os dois formatos.

4.1.1.3 Quadro de Erro

O quadro de erro é gerado por qualquer nó que detectar um erro no barramento, no instante em que ele detecta o erro. O quadro de erro mostrado na Figura 29. Ele é composto por dois campos: um campo de sinalização de erro e um campo delimitador de erro. O campo delimitador de erro é composto por 8 bits recessivos e serve para reiniciar a comunicação de forma limpa após um erro no barramento.

O campo de sinalização de erro é composto por 6 bits consecutivos do mesmo

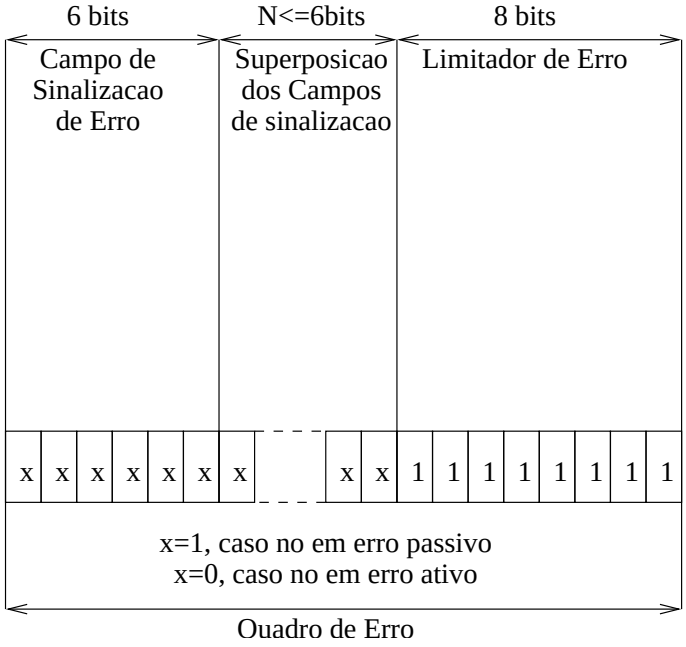


Figura 29: Quadro de erro do CAN.

nível, de forma que quando emitido, ele irá gerar erros de *bit-stuffing* nos demais nós. O valor do bit neste campo é definido pelo estado do nó. Caso o nó esteja no estado erro ativo, ele irá enviar 6 bits dominantes, de forma que todos os outros nós irão receber esse quadro, e irão acusar erro na mensagem. Caso o nó esteja em erro passivo, ele irá enviar 6 bits recessivos. Assim, os outros nós só irão ser informados do erro caso o transmissor tenha enviado o quadro, caso contrário, a transmissão continuará de forma normal.

O campo de sinalização pode ter um tamanho variável. Isto é necessário para que todos os quadros de erro sejam superpostos. Por exemplo, somente um nó detecta um erro no barramento, nesse caso, ele irá começar a emitir o campo de sinalização e no pior caso possível, o sexto bit deste campo irá causar um erro de *bit-stuffing* nos outros nós. Então os outros nós irão começar a enviar os seus quadros de erros com os 6 bits no campo de sinalização. Os dois campos de sinalização ficam superpostos, somando um total 12 bits de mesmo valor.

O quadro de erro é gerado nas seguintes situações:

- Quando o CRC calculado pelo receptor for diferente do enviado na mensagem (mensagem corrompida);
- Quando o *bit-stuffing* for violado entre o SOF e o limitador do CRC, ou seja, quando 6 bits com o mesmo valor forem recebidos de forma consecutiva (geralmente um quadro de erro gerado por outro nó);
- Quando o transmissor detectar um bit dominante no campo EOF, ou no bit limitador de campo, ou no bit limitador de CRC, ou no espaço entre quadros (espaço onde nenhum nó pode escrever no barramento);
- Quando um bit transmitido é monitorado com outro valor fora da fase de arbitragem, ou do bit de confirmação;

O quadro de erro não é gerado nos seguintes casos:

- Quando o transmissor não detecta um bit dominante no bit de confirmação, isto significa que nenhum nó recebeu a mensagem e portanto ela deve ser retransmitida;
- Quando um bit transmitido é monitorado com outro valor durante a fase de arbitragem, isto significa que o nó perdeu a arbitragem e deve parar de transmitir;
- Quando uma mensagem inválida é recebida pelo receptor, por exemplo, um nó configurado em uma frequência diferente do sistema irá se encaixar neste caso;

4.1.2 Camada Física

O meio físico do barramento CAN é definido pela ISO11898 como uma linha de dois condutores, com um terra de retorno comum. Nas terminações do barramento são adicionados terminadores de linha com a impedância característica da linha. Os condutores podem ser do tipo par trançado ou paralelos, com ou sem blindagem, dependendo do isolamento eletromagnético necessário na aplicação.

O tamanho máximo do barramento irá depender principalmente da velocidade do barramento. O tamanho máximo é limitado em 1 km, porém a uma taxa de 1 MHz ele é limitado a 40 m.

De acordo com a norma DIN41652, é recomendado utilizar conectores DB9 com uma pinagem segundo a Tabela 4.

Tabela 4: Pinagem do barramento CAN.

Pino	Nome	Descrição
1	—	Reservado
2	CAN_L	Bit dominante é representado como nível baixo de tensão
3	CAN_GND	Terra do barramento CAN
4	—	Reservado
5	CAN_SHLD	Blindagem do barramento (Opcional)
6	GND	Terra (Opcional)
7	CAN_H	Bit dominante é representado como nível alto de tensão
8	—	Reservado
9	CAN_V+	Alimentação externa ao barramento (Opcional)

Conforme especificado na ISO11898, o barramento CAN possui uma camada física que utiliza uma transmissão diferencial através dos pinos CAN_L e CAN_H. Quando o barramento não está sendo utilizado, o seu valor é considerado recessivo, onde os dois sinais CAN_L e CAN_H ficam flutuando. Quando um valor dominante for representado no barramento, o sinal CAN_L é forçado para o nível baixo, enquanto o sinal CAN_H é forçado para o nível alto.

O pino 9 pode ser utilizado como uma alimentação externa para alimentar os *transceivers* e os opto-acopladores dos nós do barramento. Desta forma, consegue-se isolar o barramento CAN do resto do circuito elétrico dos nós, dispensando o uso de conversores DC/DC. Esta alimentação deve ser externa e nenhum nó deve gerar esta alimentação. Ela deve estar na faixa de $7\ V \leq V \leq 13\ V$ e em geral deve ser localizada no meio do barramento.

Apesar de reservado, o pino 8 pode ser utilizado como linha de erro do sistema. Nas terminações do barramento um resistor de 124 Ω deve ser conectado entre os pinos 2 e 7 do conector, para reduzir as reflexões das ondas (CIA, 1994).

4.2 *Hardware* da AIC

O *hardware* da placa AIC, vista na Figura 30, é constituído de duas partes: periféricos capazes de executar as tarefas necessárias às juntas de um robô e um processador para coordenar o uso dos periféricos. O segundo grupo comanda as operações do robô. O primeiro grupo serve de suporte ao segundo, fazendo com que as ordens do processador sejam executadas. Por exemplo, um processador pode querer acionar um motor com uma determinada tensão, porém ele não é capaz de fornecer a corrente e a tensão necessárias para o motor. Desta forma, o processador irá comandar um periférico que é capaz de realizar esta operação.

Os esquemáticos da placa AIC estão localizados no apêndice A.

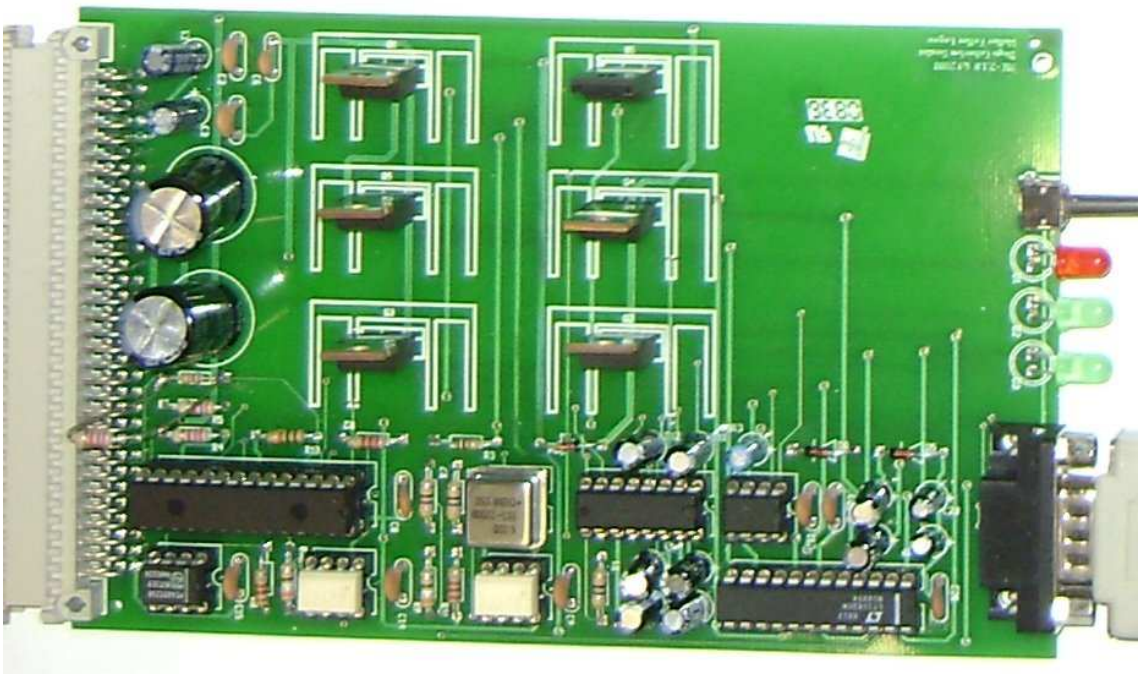


Figura 30: Placa AIC.

O diagrama de blocos da AIC é apresentado na Figura 31. A placa AIC contém um microprocessador dsPIC30F4012 (MICROCHIP, 2007), que é especializado em controle de motores. Ele é um processador de 16 bits, com 48 *KBytes* de memória de programa e com controladores adequados a finalidade da placa como: QEI (*Quadrature Encoder Interface*), PWM (*Pulse Width Modulation*), UART (*Universal Asynchronous Receiver Transmitter*), CAN e pinos de I/O.

A UART é um módulo de comunicação serial assíncrona *full-duplex*. A esse controlador é adicionado um *transceiver* RS-232 para poder conectar a placa AIC em uma porta serial de um computador. Dois pinos de I/O são posicionados nos pinos CTS e RTS do conector RS-232 para a implementação de controle de fluxo de comunicação por *hardware*. Apesar de servir de *debug*, a interface serial não é um método eficaz para a comunicação da placa AIC com um PC durante o seu

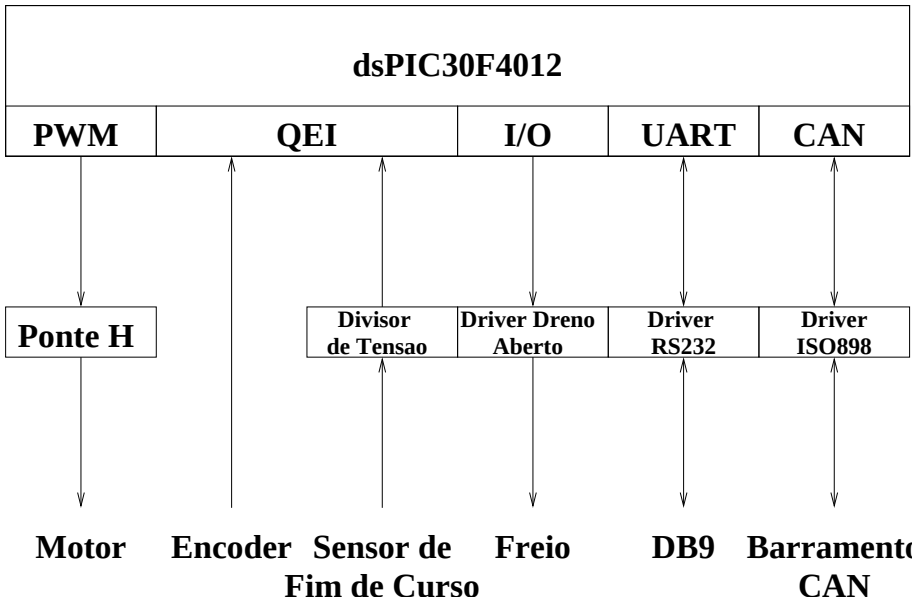


Figura 31: Diagrama de blocos da placa AIC.

funcionamento, devido a baixa taxa de transmissão desse módulo em comparação com o módulo CAN.

O QEI é um módulo que fornece uma interface para *encoders* incrementais. Ele consiste de um decodificador de quadratura que interpreta os sinais do *encoder* e um contador que guarda a contagem. A interface com o *encoder* de quadratura não necessita de nenhum componente adicional para o seu funcionamento. Porém, um divisor de tensão é colocado para adaptar o sinal de entrada aos níveis de tensão do microprocessador.

O PWM é um módulo especializado na geração de múltiplos PWMs, que são úteis no acionamento de motores DC. Na placa AIC, o motor é acionado através de uma ponte H. Tal ponte é um circuito eletrônico que alimenta um motor DC em sentido direto, reverso, ou parado. A ponte H é composta por transistores MOSFETs, pois eles suportam correntes maiores sem aquecerem muito, além de terem diodos de *flywheel* encapsulados. A ponte H é acionada por um *driver* de ponte H com MOSFET, que recebe os sinais da interface de controle de motor via PWM. Essa topologia permite que a tensão aplicada ao motor varie de -60 V até 60 V, dependendo do valor da alimentação Vpp.

O CAN é um módulo que implementa o protocolo CAN. A essa interface é adicionado um *transceiver* CAN que implementa o padrão ISO11898. Esse padrão define que a alimentação do *transceiver* vem do barramento. Desta forma é necessário utilizar opto-acopladores para conectar o *transceiver* ao microprocessador. Isto garante uma maior robustez ao barramento, apesar de aumentar consideravelmente o atraso de propagação do sistema.

A saída para o freio é feita utilizando um pino de I/O do dsPIC em conjunto com um transistor em dreno aberto.

Na placa AIC, o processador é excitado por um clock de 6 MHz, esse clock é multiplicado 16 vezes internamente por um PLL, obtendo-se uma frequência de operação de 96 MHz e um ciclo de máquina de 24 MHz.

4.3 *Software* da AIC

O *software* da placa AIC pode ser dividido em basicamente duas partes: uma parte que é implementada no dsPIC e outra parte que é implementada em um PC. O lado do dsPIC tem como objetivo, implementar os comandos recebidos do barramento CAN e executá-los. Este lado é composto por uma biblioteca de acesso aos periféricos e um programa principal que fica recebendo e executando os comandos de um PC.

No lado do PC, foi desenvolvida uma interface que permite ao usuário enviar comandos a AIC. Essa interface é feita através de classes, de forma que o usuário não precise se preocupar com a sua implementação e tenha uma visão bem próxima da placa.

4.3.1 *Software* do dsPIC

Para dar início ao desenvolvimento da AIC foram implementadas funções de baixo nível para dar suporte às aplicações de alto nível.

Para o freio, foram implementadas funções que acionam e liberam o mesmo. O *hardware* foi projetado também de tal modo que durante a inicialização da placa, o freio fiquei sempre ativado.

O QEI contém em *hardware* um contador de 16 bits. Devido a um *bug* (MICROCHIP, 2008), o bit mais significativo deste contador não funciona, e portanto somente 15 bits devem ser utilizados. Uma variável binária de 15 bits consegue armazenar 32768 pulsos. O contador é armazenado em uma variável de 31 bits em *software*, aumentando assim o número máximo de pulsos que podem ser contados. Essa variável incrementa ou decrementa conforme o giro do motor. Para essa variável foram implementadas funções que: retornam o seu valor, zeram o seu valor, retornam e zeram o seu valor. O valor da leitura de pulsos pode ser traduzido em radianos segundo o número de pulsos passado na inicialização do *encoder*. Funções que retornam o estado do sensor de fim de curso também foram implementadas.

Para o motor foram implementadas funções que permitem configurar a frequência de operação do PWM, bem como habilitar e desabilitar os motores, ou ainda modificar o ciclo de trabalho do PWM. Durante a inicialização da placa, nenhuma tensão é aplicada ao motor.

Para comunicar a placa com um PC, foram implementadas funções para a UART do dsPIC e para o controlador CAN. A UART funciona com uma taxa de 19200 bps, com o formato 8N1, sem controle de fluxo. Somente funções para enviar caracteres e strings estão disponíveis. Nenhuma função de recebimento serial foi implementada até o momento. Isso porque a serial tem apenas uma função de *debug* nesta placa. Para enviar comandos, o barramento CAN é utilizado.

No lado do CAN, foram implementadas funções para suportar todos os quadros do protocolo CAN, bem como para configurar os filtros e as máscaras de recebimento e a frequência de operação do barramento. Ele suporta frequência de até 1 MHz e funções de escrita e recebimento foram implementadas. Estas funções são apenas um acesso ao *hardware* do processador e nenhum *buffer* de recebimento foi implementado. Assim, caso uma mensagem chegue pelo barramento antes que a anterior seja retirada do *hardware*, a última mensagem irá sobrescrever a anterior.

Todas as funções estão documentadas nos seus cabeçalhos que estão em anexo no apêndice B.

Para coordenar a utilização dessas funções, foi implementado um programa principal que recebe as mensagens através do barramento CAN e executa-as. Esse programa utiliza o sistema operacional FreeRTOS (THE FREERTOS PROJECT, 2009). O FreeRTOS é um mini *kernel* de tempo real, de código aberto, que suporta processadores da família dsPIC.

Utilizar um sistema operacional apresenta vantagens como a implementação de suporte a tarefas e corrotinas, além de diversas bibliotecas de comunicação entre processos.

O programa principal tem a função de inicializar a placa AIC. Para isto existe a função `aic_initialize`, onde se define a tensão máxima do motor, bem como a frequência de operação do PWM e o número de pulsos por revolução do *encoder*. O *watchdog* é inicializado na função `wdt_initialize` para reiniciar a placa, caso a função `wdt_ping` não seja chamada por 100 ms. A função `xTaskCreate` dispara uma tarefa periódica.

Essa tarefa fica constantemente monitorando o modulo CAN para verificar se alguma mensagem foi recebida. Caso nenhuma mensagem tenha sido recebida, o programa é adormecido e espera a sua próxima execução. Caso alguma mensagem tenha sido recebida, o programa principal irá chamar as funções necessárias para executar o comando associado a mensagem recebida. Quando um comando é recebido, o programa irá reiniciar o *watchdog* impedindo que a placa reinicie.

O *watchdog* é utilizado por questões de segurança, para que, caso a placa AIC fique 100 ms sem receber nenhum comando, ela será reiniciada. Ao ser reiniciada, a placa leva a junta a um estado seguro, onde o motor está parado e o freio acionado. O código do programa principal está no apêndice C.

O tempo de processamento no dsPIC de cada um dos comandos é apresentado na Tabela 5. Estes tempos foram medidos utilizando um osciloscópio e os demais pinos de I/O do dsPIC. Quando o dsPIC recebia um comando, ele colocava uma tensão de 5 V no pino RB0, após executar o comando, ele colocava uma tensão de 0 V. Ao fazer isto de forma periódica, um pulso quadrado aparece no osciloscópio. Os comandos de ler *status* e aplicar tensão demoram mais tempo, pois eles implementam cálculos com números em ponto flutuante, sem um *hardware* específico para isto. Os demais comandos apresentam um tempo muito baixo por se tratarem de operações de modificar bits do dsPIC.

Tabela 5: Tempo de processamento de cada comando na placa AIC.

Comando	Tempo
Ler *status*	69 us
Aplicar tensão no motor	65 us
Aplicar freio	1 us
Desligar freio	1 us
Ligar motor	1 us
Desligar motor	1 us

4.3.2 Protocolo AIC - *host*

A comunicação entre a placa AIC e o computador *host* é realizada utilizando o barramento CAN. Tal barramento é utilizado por apresentar baixos tempos de

latência, sendo assim adequado para aplicações de tempo real, além de ser um protocolo aberto. Devido ao baixo número de comandos necessários para a placa AIC, e a quantidade de juntas que um robô manipulador usualmente tem, o protocolo CAN na versão 2.0A é utilizado. Neste caso, o identificador possui somente 11 bits.

O campo identificador do barramento CAN, o qual contém o significado da mensagem, é divido em duas partes: a parte mais significativa da mensagem irá conter o comando a ser realizado, enquanto a parte menos significativa irá endereçar a placa que deve realizar o comando. A Tabela 6 demonstra está divisão.

Tabela 6: Protocolo AIC - *host*.

Bits	Descrição	
10-5	Comando	Dado
	000000 = Reservado	——
	000001 = Reinicia AIC	——
	000010 = Desligar motor	——
	000011 = Aplicar freio	——
	010000 = Aplicar tensão no motor	8 *bytes*
	100000 = Ler *status*	8 *bytes*
	110000 = Desligar freio	——
	110001 = Ligar motor	——
4-0	Endereço da AIC	
	00000 Reservado	
	00001 AIC 1	
	00010 AIC 2	
	⋮	
	11111 AIC 31	

Esta atribuição de comandos é feita de forma a priorizar a segurança. Os comandos que levam a junta a um estado sem movimento, como desligar motor e aplicar freio têm uma prioridade maior que os outros. Desta forma, eles ganham o controle do barramento no caso de uma transmissão simultânea no barramento CAN.

O mesmo raciocínio vale para as juntas. A AIC 1 possui uma prioridade maior que a AIC 2, que por sua vez possui uma prioridade maior que a AIC 3, e assim por diante. Desta forma também é possível priorizar juntas consideradas mais importantes no robô manipulador. A macro `AIC` do programa principal do dsPIC define o número da AIC.

Todos os comandos, com exceção do ler *status*, devem ser enviados pelo *host* para serem executados na devida AIC. O comando ler *status* contém a informação do *encoder* e do interruptor de fim de curso da junta e portanto ele deve ser enviado pela AIC. O *host* pode solicitar o envio deste pacote da AIC, através do envio de um quadro remoto com o mesmo identificador. Este comando, contém no campo de dados o deslocamento da junta desde a última leitura e o valor do interruptor de fim de curso. O deslocamento está nos 4 *bytes* mais significativos, em radianos, representado em IEEE-754 com precisão simples, e o valor do interruptor está no bit menos significativo. Os demais bits são reservados para futuras expansões.

O comando para aplicar tensão no motor, contém o valor de tensão a ser aplicado no seu campo de dados, representado em IEEE-754 com precisão dupla.

4.3.3 *Software* do *host*

Esta parte do *software* é feita para ser executada em um computador *host* que controla a AIC. Ela é feita para rodar em um computador utilizando o sistema operacional Linux-2.6.x (THE LINUX KERNEL ARCHIVES, 2009), com o *patch* de tempo real do RTAI (RTAI - THE REALTIME APPLICATION INTERFACE FOR LINUX FROM DIAPM, 2009).

O Linux é um sistema operacional convencional que a princípio procura partilhar o tempo de processamento entre todas as tarefas do usuário, e por causa disto, ele não fornece um escalonamento em tempo real. O *patch* do RTAI é uma extensão do *kernel* do Linux, que fornece capacidades de tempo real ao sistema. Ele utiliza o conceito de HAL (*Hardware Abstraction Layer*), inserindo uma camada entre o *hardware* e o Linux para capturar as interrupções do sistema. Desta forma o RTAI assume o controle do *hardware* e executa o Linux como uma tarefa de *background* quando nenhuma outra tarefa de tempo real estiver sendo executada.

Uma das desvantagens desta abordagem, é que ao se programar uma tarefa de tempo real, não se pode utilizar funções do sistema operacional. Caso se utilize funções do sistema operacional, como por exemplo `printf` da biblioteca padrão C, o escalonador irá executar o sistema operacional. Este, por sua vez, irá realizar todas as suas tarefas, incluindo o `printf`, e então irá retornar para a tarefa de tempo real. Note que quando o sistema operacional executar, a temporização não será garantida.

Para comunicar o computador com a placa AIC através do barramento CAN, uma placa PCICan da Kvaser (WELCOME TO KVASER - ADVANCED CAN SOLUTIONS FOR HARDWARE, SOFTWARE, CONSULTING AND EDUCATION, 2009) foi utilizada. Ela é uma placa PCI que suporta o protocolo CAN na sua versão padrão e estendida. Os seus *drivers* para o barramento são de acordo com a ISO11898-2, que define os barramentos CAN de alta taxa de dados. Ela apresenta a vantagem de possuir tanto a sua documentação como o seu *software* abertos.

Porém o seu *driver* apresenta um problema: ele utiliza funções do sistema operacional, como filas e chamadas às funções do sistema operacional. Desta forma a temporização da comunicação entre o *host* e a placa AIC fica comprometida. Para resolver este problema, o *driver* original do fabricante foi reescrito de forma a não comprometer a temporização da comunicação. Esta reescrita foi feita de forma similar à biblioteca CAN do dsPIC e ela fornece um acesso básico ao *hardware*.

As suas funções são bem similares as funções de *drivers* de caracteres:

- `rt_pcican_close(void *handler)`: função que serve para finalizar a placa PCICan, desconectando o nó do barramento;
- `rt_pcican_open(void **handler, int channelNr)`: função que inicializa a placa e canal da placa, atualmente só é suportado o canal 0 da placa;
- `rt_pcican_read(void *handler, CAN_MSG *m)`: lê a última mensagem do barramento;
- `rt_pcican_write(void *handler, CAN_MSG *m)`: escreve uma mensagem no barramento;
- `rt_pcican_ioctl(void *handler, unsigned int cmd, void *buffer)`: envia um comando para o barramento. Eles podem ser os seguintes:

- `IOCTL_SETBUSPARAM`: para configurar a frequência do barramento;
- `IOCTL_BUSON`: para conectar o nó no barramento;
- `IOCTL_BUSOFF`: para desconectar o nó do barramento;
- `IOCTL_PCICAN_STATUS`: para informar o *status* do nó no barramento;

Essas funções funcionam somente como acesso ao *hardware*, e nenhum fluxo de controle é implementado. Caso o usuário tente enviar uma mensagem enquanto a placa estiver enviando uma mensagem, um erro é retornado e a mensagem não é enviada. Caso a placa receba duas mensagens consecutivas, sem que o usuário chame a função `rt_pcican_read`, a primeira mensagem é perdida.

A estrutura usada na leitura/escrita de mensagens é a seguinte:

```
typedef struct s_can_msg {
        unsigned long id;               ///< CAN ID
        unsigned char flags;            ///< CAN type
        unsigned char data [8];         ///< CAN data
        unsigned char length;           ///< size of data
} CAN_MSG;
```

O campo `flags` indica o tipo de quadro, isto é, se ele será um quadro de dados ou remoto, e se será um quadro padrão ou estendido. O campo `id` deve conter o identificador da mensagem, enquanto o campo `data` contém a mensagem em si. Por fim, o campo `length` contém o tamanho dos dados da mensagem. O cabeçalho deste *driver* se encontra no apêndice D.

Este *driver* é compilado como um módulo do *kernel* do Linux e utiliza uma interface similar ao módulo LXRT do RTAI para realizar a comunicação entre o usuário e o sistema operacional. Ele funciona de modo similar as chamadas de sistema no Linux. As funções citadas causam uma interrupção de *software* no sistema operacional. Estas funções são tratadas pela a função `rtai_lxrt_handler(void)`, que salva os argumentos na pilha e realiza os trabalhos necessários para chavear de modo usuário para supervisor, e por fim chama a função do módulo que faz o trabalho.

Com as funções do meio de comunicação implementadas, uma biblioteca que implemente o protocolo de comunicação foi construída. Inicialmente esta biblioteca define um canal de comunicação utilizando a classe `AIC_COMM` em linguagem C++. Esta classe tem por objetivo definir a forma de transmissão e recepção de comandos, para que a comunicação fique independente do barramento.

A classe `AIC_COMM` define duas funções puramente virtuais que realizam a troca de comandos: `send_command` e `get_status`. A função `send_command` envia um comando de acordo com a Tabela 6, enquanto a função `get_status` recebe o *status* da placa, este é solicitado com o comando ler *status*. Uma função virtual, em C++, significa que a função pode ser redefinida em uma classe derivada a partir desta classe (STROUSTRUP, 2000). Isto permite que diversas implementações do canal de comunicação da biblioteca possam ser construídas, sem precisar alterar o restante do código.

Então, foi criada uma classe para cada um dos periféricos da AIC. Estas classes têm por finalidade enviar os seus comandos para os periféricos da placa AIC, utilizando a classe `AIC_COMM`. A relação entre as classes criadas e os comandos da placa AIC está apresentada na Tabela 7.

Tabela 7: Classes e comandos da placa AIC.

Classe	Comandos
`AIC_BRAKE`	Acionar Freio
	Liberar Freio
`AIC_MOTOR`	Ativar Motor
	Aplicar Tensão no Motor
	Desligar Motor
`AIC_ENCODER`	Ler *Status*
`AIC_INDEX`	Ler *Status*

Todas estas classes são agrupadas em uma classe chamada `AIC`, a fim de representar a placa como um todo. Por fim, uma classe derivada das classes `AIC` e `AIC_COMM` é necessária. Esta classe possui o nome de `AIC_CAN` e implementa as funções virtuais de `AIC_COMM`, fornecendo assim um meio de transporte para os comandos enviados pelas outras classes. A Figura 32 apresenta a hierarquia de classes da biblioteca implementada.

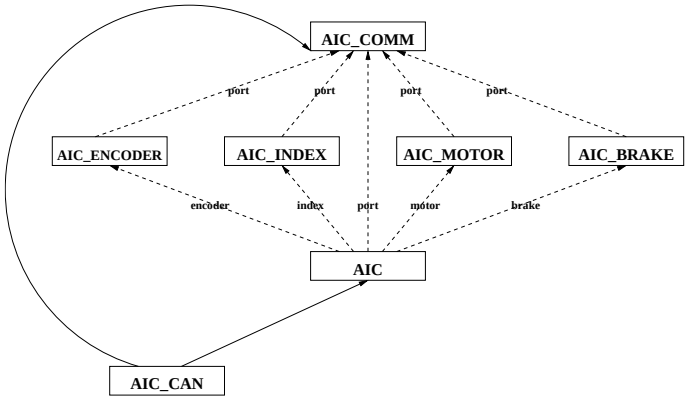


Figura 32: Hierarquia de classes da biblioteca AIC.

Na Figura 32, as linhas cheias significam que uma classe é derivada da outra. A base da linha é a classe derivada, enquanto que a seta é a classe base. As linhas tracejadas possuem uma relação onde as classes com as setas são atributos das classes que originam as setas.

Este tipo de abordagem encapsula todos os detalhes e permite que o meio de comunicação seja futuramente alterado, sem precisar modificar a API disponível ao usuário. Isto assegura que o código do usuário possa permanecer inalterado ao longo das futuras modificações. Outra vantagem desta implementação é a sobrecarga de operadores. Desta forma o usuário utiliza a biblioteca de uma maneira bem próxima à placa. O exemplo abaixo demonstra a utilização da biblioteca.

```
#include <aiccan.h>

int main(int argc,char *argv[])
{
        //criando comunicação com a placa AIC07
        AIC aic07=new AIC_CAN(7);
```

```
float position;

//liberando o freio da placa
aic07->brake.release();
//acionando o motor da placa
aic07->motor.on();
//aplicando 7 V no motor da placa
aic07->motor=7.0;
//lendo o deslocamento da placa
position=aic07->encoder.read();
.
.
.
```

No código, o comando `new AIC_CAN(7)` instancia um objeto referente a classe `AIC_CAN` para comunicar com a placa detentora do ID 7 no seu processador dsPIC. A seguir, funções como `brake.release()` e `motor.on()` enviam os seus respectivos comandos, de acordo com a Tabela 6. O operador `=` é sobrecarregado na classe `AIC_MOTOR` para aplicar a tensão do PWM na placa AIC para 7 V. Por fim, a função `encoder.read()` retorna o valor do *encoder* da placa AIC.

Para avaliar o desempenho temporal do *driver* e da biblioteca foram realizadas sucessivas medidas do tempo que o comando `encoder.read()` demora para ser realizado. A Figura 33 apresenta o resultado. As amostras desta figura possuem uma média de 237.35 us, com uma variância de 11.85 us. Está variância é pequena e confirma a baixa latência do barramento CAN.

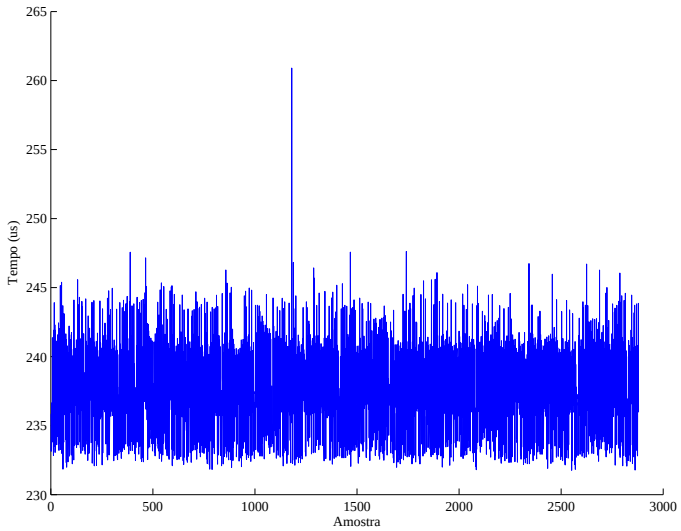


Figura 33: Temporização de sucessivas chamadas de `encoder.read()`.

A média pode ser melhor analisada. Idealmente, o tempo de barramento neste comando é composto por um quadro remoto de 44 bits solicitando a leitura do *encoder*, mais um quadro de dados padrão de 108 bits, totalizando assim 152 bits. Com uma frequência de 1 MHz, isto corresponde a 152 us. Este valor é ideal, pois ainda existem bits gerados devido a *bit-stuffing*. Quando o dsPIC recebe o quadro remoto, ele necessita de tempo para processar a informação antes de responder. Este

tempo foi medido e apresentado na Tabela 5 e corresponde a 60 us. Diminuindo estes valores da média, restam 26 us, que podem ser atribuídos ao *driver* de tempo real e também à própria placa PCIcan.

4.3.4 AIC como Componente do OROCOS

A placa AIC é modelada por um componente no OROCOS, de forma a oferecer o suporte ao *hardware* do robô para o componente `Joint` da seção 3.2. O modelo de componente `AIC` mapeia todas as funcionalidades da placa AIC na interface utilizada pelo componente `Joint` no OROCOS. A sua interface é representada na Figura 34.

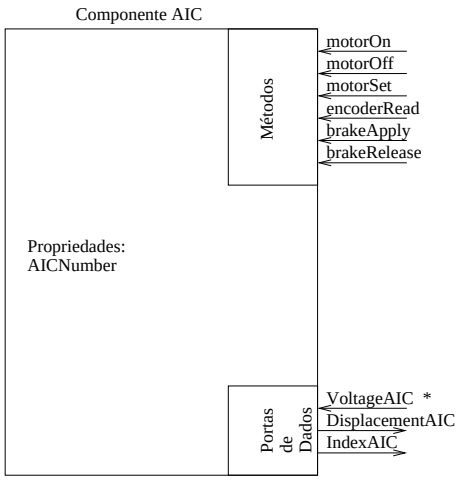


Figura 34: Modelo de componente AIC.

Os métodos apresentados pelo componente AIC são diretamente mapeados nas respectivas funções da classe AIC. Desta forma, se um outro componente quiser utilizá-los, basta que ele se conecte como par do componente AIC. Os métodos `motorOn`, `motorOff`, `brakeApply` e `brakeRelease` retornam um valor booleano para informar se o respectivo comando foi enviado pelo barramento e não possuem nenhum argumento. O método `motorSet`, também retorna um valor booleano, e possuem como argumento o valor de tensão a ser passado para a placa. O método `encoderRead` não possui argumento e retorna o deslocamento do eixo do motor em radianos, desde a última leitura. Estas funcionalidades foram implementadas como métodos, pois desta forma, irão reagir de forma síncrona com o componente que está utilizando os métodos.

Do ponto de vista do componente AIC, as portas de dados `DisplacementAIC` e `IndexAIC` são somente de escrita e são atualizadas quando o método `encoderRead` é chamado, de forma redundante com o valor de retorno do método. A porta `VoltageAIC*` é somente de leitura, o * indica que uma escrita nesta porta irá causar a emissão de um evento. Este evento será tratado em uma *callback* do próprio componente AIC dono da porta, que reage ao evento de forma assíncrona com a escrita na porta. Essa *callback* chama o método `motorSet` com o argumento igual ao valor escrito na porta.

A propriedade `AICNumber` é utilizada para configurar o componente, a partir de um arquivo XML, na sua função `configureHook()` e irá amarrar o ID da placa AIC com o componente instanciado. A função de inicialização do componente `startHook()` contém uma chamada da função `encoder.read()` da classe AIC para zerar o *encoder* da placa.

4.4 Conclusão

Neste capítulo foi apresentado em detalhes o desenvolvimento de uma placa de acionamento com os três graus de liberdade abaixo:

- independência de função: apesar de que, neste trabalho, ela é utilizada de forma específica para acionamento de robôs, toda a parte de *software* da placa foi feita de forma genérica a fim de possibilitar outros usos.
- independência de robô: ela é capaz de acionar qualquer robô que utilize motores DC com uma tensão de até 60 V, de medir *encoders* incrementais com sensores de fim de curso e de acionar freios eletromecânicos.
- independência de barramento: a API da placa é feita de forma genérica, suportando outras implementações com outros barramentos de comunicação.

5 IMPLEMENTAÇÃO DA ARQUITETURA NO ROBÔ JANUS

Este capítulo apresenta a implementação da arquitetura definida no Capítulo 3 para o robô Janus. O *hardware* das juntas é acionado utilizando a placa AIC e o barramento CAN, conforme descrito no Capítulo 4.

A arquitetura proposta será implementada para controlar as duas juntas do sistema de visão do robô Janus, neste capítulo, e futuros trabalhos podem abranger outras partes do robô. Os componentes `Sensor2Mux` e `Actuator2Demux` agrupam as duas juntas do sistema de visão, e a seguir três controladores diferentes são utilizados.

Para a utilização dos controladores de torque calculado e *feedforward*, é necessário a obtenção do modelo dinâmico do robô. Esse modelo pode ser obtido de forma analítica utilizando-se métodos difundidos na literatura, como os métodos de Newton-Euler ou Lagrange-Euler (CRAIG, 1989; FU; GONZALES; LEE, 1987). No entanto, para utilização efetiva de tais modelos, parâmetros como massa e momento de inércia dos elos devem ser conhecidos, o que não ocorre neste caso.

A obtenção desses parâmetros a partir das características geométricas do robô é viável utilizando-se de diversas simplificações, como aproximar as formas geométricas dos elos por formas geométricas regulares e considerar a sua distribuição de massa uniforme. Essas simplificações não são reais, pois os elos do robô são ocos e têm em seu interior atuadores, sensores e cabeamento, de forma que as suas distribuições de massa estão longe de ser uniforme. Assim, uma alternativa para obtenção do modelo do robô Janus confiável, que reflita o sistema real, é a identificação dos parâmetros do seu modelo, o que é feito neste capítulo.

Este capítulo está dividido da seguinte forma: na seção 5.1 o robô Janus é apresentado, a seguir a seção 5.2 apresenta o desenvolvimento analítico do modelo do robô Janus. A seção 5.3 demonstra o método utilizado para identificar as constantes do modelo e a sua validação do mesmo, e por fim, a seção 5.4 apresenta a implementação da arquitetura proposta.

5.1 Robô Janus

O robô Janus é um protótipo de um robô antropomórfico, com um sistema de visão estéreo. A sua geometria foi inspirada na parte superior do corpo humano, como mostra a Figura 35. As suas juntas são compostas por motores DC e *encoders* incrementais. Elas podem atingir uma velocidade de rotação máxima de cerca de 60 graus por segundo. Todas as juntas possuem o mesmo tipo de interface.

Figura 35: Robô Janus.

Originalmente, o robô Janus utilizava amplificadores analógicos e módulos PID para o seu controle. Os módulos PID implementavam o controlador PID por *hardware* através do *chip* LM628 (NATIONAL, 2003), montados em dois PCs industriais baseados no processador 80486 executando o sistema operacional Linux. O sistema sofreu um *retrofiting* (LAGES; BRACARENSE, 2003), sendo os amplificadores analógicos substituídos por PWMs digitais.

O sistema de visão do robô Janus é composto por três elos interligados por duas juntas rotacionais, possuindo duas câmeras de vídeo na extremidade do terceiro elo, como mostra a Figura 36.

5.2 Modelagem

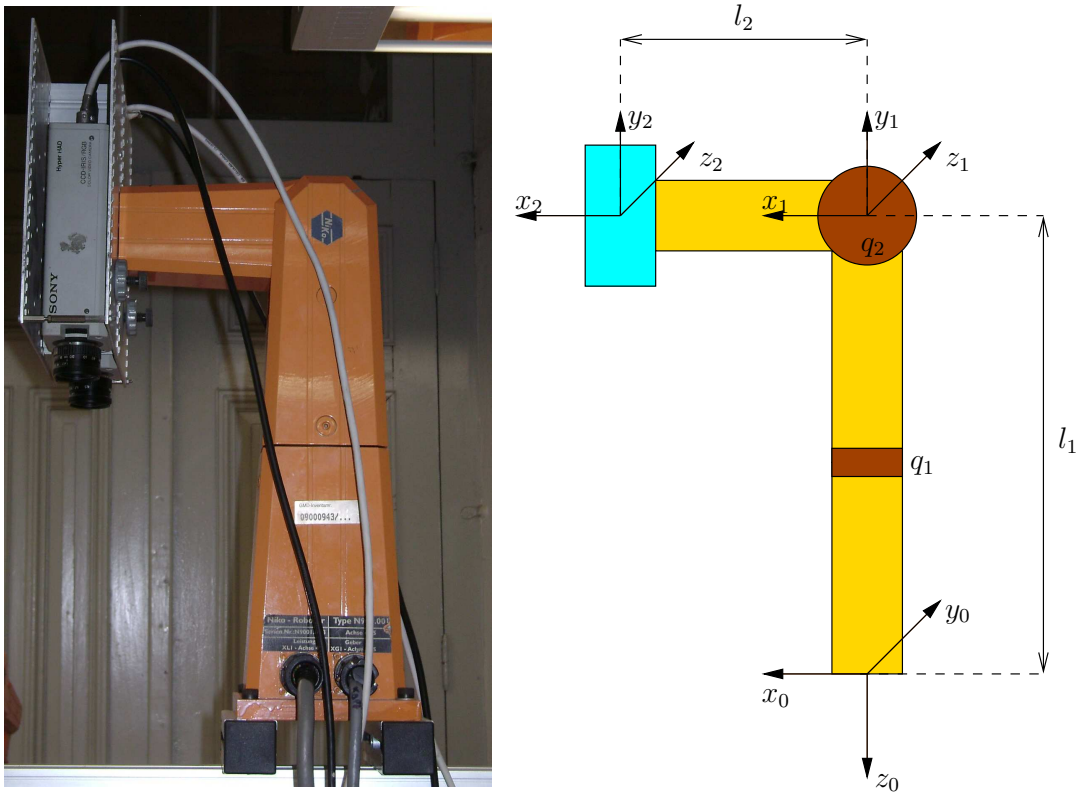
Existem dois tipos básicos de modelos para representar um robô manipulador: modelos cinemáticos e modelos dinâmicos.

O modelo cinemático representa o movimento do robô em relação ao sistema de coordenadas da base, o qual é fixo, sem considerar as forças ou torques que geram este movimento. Assim ele descreve o movimento espacial de um robô como uma função do tempo, através das relações entre os ângulos de cada junta e a posição e a orientação da ferramenta do robô no sistema de coordenadas da base.

O modelo dinâmico é o conjunto de equações matemáticas que representam o comportamento dinâmico do robô, isto é, a relação entre a força/torque aplicada/o em cada junta, e o movimento do robô em si.

Inicialmente, para se modelar um robô, é necessário atribuir sistemas de coordenadas associados a cada junta do robô. A convenção de Denavit-Hartenberg (DENAVIT; HARTENBERG, 1955; FU; GONZALES; LEE, 1987) propõe uma forma sistemática para esta atribuição. Ela facilita a obtenção dos modelos do robô. A Figura 36(b) demonstra os eixos atribuídos para a cabeça de visão robótica do robô Janus.

Na Figura 36(b), o sistema referencial da base é representando pelos eixos $\{x_0, y_0, z_0\}$



(a) Foto. (b) Diagrama.

Figura 36: Sistema de visão do robô Janus.

e é fixado na base da cabeça. Os sistemas $\{x_1, y_1, z_1\}$ e $\{x_2, y_2, z_2\}$ se movimentam de acordo com as juntas q_1 e q_2 respectivamente. O deslocamento de cada junta q_i é medido como sendo o deslocamento do eixo x_{i-1} até o eixo x_i sobre o eixo z_{i-1}. Na Figura 36(b) tem-se o robô com as seguintes posições angulares: $q_1 = 0$ radianos e $q_2 = 0$ radianos.

Cada junta q_i está também associada a um elo i que tem um conjunto de parâmetros mecânicos representados no eixo $\{x_i, y_i, z_i\}$, como os momentos de inércia: Ii_{xx}, Ii_{yy} e Ii_{zz}; os produtos de inércia: Ii_{xy}, Ii_{xz} e Ii_{yz}; a posição do seu centro de massa: $\bar{x}_i$, $\bar{y}_i$ e $\bar{z}_i$. Além de sofrer a ação da gravidade g e possuir uma massa m_i.

5.2.1 Modelo Cinemático

A utilização das convenções de Denavit-Hartenberg permite obter diretamente as matrizes de transformações de base:

$$
{}^0A_1 = \begin{bmatrix} \cos q_1 & 0 & -\operatorname{sen} q_1 & 0 \\ \operatorname{sen} q_1 & 0 & \cos q_1 & 0 \\ 0 & -1 & 0 & -l_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^1A_2 = \begin{bmatrix} \cos q_2 & \operatorname{sen} q_2 & 1 & 0 \\ \operatorname{sen} q_2 & \cos q_2 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^0A_2 = \begin{bmatrix} \cos q_1 \cos q_2 & -\cos q_1 \operatorname{sen} q_2 & -\operatorname{sen} q_1 & l_2 \cos q_1 \cos q_2 \\ \operatorname{sen} q_1 \cos q_2 & -\operatorname{sen} q_1 \operatorname{sen} q_2 & \cos q_1 & l_2 \operatorname{sen} q_1 \cos q_2 \\ -\operatorname{sen} q_2 & -\cos q_2 & 0 & -l_1 - l_2 \operatorname{sen} q_2 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

onde a matriz ${}^iA_j$ é chamada de matriz de transformação homogênea, que serve para mapear um vetor descrito no sistema de coordenadas $\{x_j, y_j, z_j\}$ em um vetor descrito no sistema de coordenadas $\{x_i, y_i, z_i\}$.

O modelo cinemático direto do sistema de visão do robô Janus, que mapeia a posição e a orientação do sistema de coordenadas da ferramenta, descrito no sistema de coordenadas da base, em função da posição angular das juntas, pode ser obtido diretamente da matriz de transformação homogênea ${}^0A_2$, que pode ser escrito na forma:

$$
\begin{cases} x = l_2 \cos q_1 \cos q_2 \\ y = l_2 \operatorname{sen} q_1 \cos q_2 \\ z = -l_1 - l_2 \operatorname{sen} q_2 \\ \alpha = q_1 \\ \beta = q_2 \\ \gamma = -90^o \end{cases} \tag{14}
$$

onde os ângulos α, β e γ, representam a orientação do sistema de coordenadas da ferramenta em relação ao sistema de coordenadas da base, utilizando os respectivos ângulos de *Roll*, *Pitch*, *Yaw*. Devido à geometria deste manipulador, não é possível posicionar a ferramenta com uma orientação *Yaw* diferente de -90^o.

5.2.2 Modelo Dinâmico

O modelo dinâmico de um robô é definido pela relação entre o movimento do robô e o torque aplicado nas suas juntas. O movimento do robô é descrito pela posição, velocidade e aceleração das suas juntas. Mais especificamente, deseja-se obter um modelo não linear na forma:

$$
\begin{aligned} \dot{x} &= f(x, u) \\ y &= h(x) \end{aligned}
$$

onde x é o estado do sistema, u é a entrada do sistema (aqui será considerada a tensão aplicada nos motores), y é a saída do sistema (tipicamente posição das juntas, no caso de robôs) e $f(\cdot, \cdot)$ e $h(\cdot)$ são funções não lineares.

O modelo dinâmico do robô, considerando como entrada o torque aplicado nas juntas, pode ser obtido a partir da equação de Lagrange-Euler (FU; GONZALES; LEE, 1987):

$$
\frac{d}{dt}\left(\frac{\partial L}{\partial \dot{q}_i}\right) - \frac{\partial L}{\partial q_i} = \tau_i \text{ para } i = 1, 2 \tag{15}
$$

onde L é a função de Lagrange $(K - P)$; sendo K a energia cinética total do robô, P a energia potencial total do robô, q_i a posição angular da junta i e τ_i o torque generalizado aplicado à junta i.

Para um robô manipulador de N juntas, a função de Lagrange pode ser reescrita como (FU; GONZALES; LEE, 1987):

$$
L = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{i} \sum_{k=1}^{i} \left[\operatorname{Tr}\left(U_{ij} J_i U_{ik}^T\right) \dot{q}_j \dot{q}_k \right] + \sum_{i=1}^{N} m_i g^\star \left({}^0A_i^i \bar{r}_i\right) \tag{16}
$$

onde U_{ij} é definido como $U_{ij} = \frac{\partial^0 A_i}{\partial q_j}$, que pode ser interpretado como os efeitos do movimento da junta j sobre os pontos descritos pelo sistema de coordenadas $\{x_i, y_i, z_i\}$; J_i é uma matriz dependente dos parâmetros das juntas, descrita em (17); $g*$ é um vetor representando a gravidade ($g^\star = [\ 0 \quad 0 \quad -g \quad 0\ ]^T$) e ${}^i\bar{r}_i$ representa o centro de gravidade do elo i representado no sistema de coordenadas i.

$$J_i = \begin{bmatrix} \frac{-I_{xx}+I_{yy}+I_{zz}}{2} & I_{xy} & I_{xz} & m_i\bar{x}_i \\ I_{xy} & \frac{I_{xx}-I_{yy}+I_{zz}}{2} & I_{yz} & m_i\bar{y}_i \\ I_{xz} & I_{yz} & \frac{I_{xx}+I_{yy}-I_{zz}}{2} & m_i\bar{z}_i \\ m_i\bar{x}_i & m_i\bar{y}_i & m_i\bar{z}_i & m_i \end{bmatrix} \tag{17}$$

Para o sistema de visão da cabeça do robô Janus são obtidas as seguintes matrizes:

$$U_{11} = \begin{bmatrix} -\operatorname{sen} q_1 & 0 & -\cos q_1 & 0 \\ \cos q_1 & 0 & -\operatorname{sen} q_1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$U_{21} = \begin{bmatrix} -\operatorname{sen} q_1 \cos q_2 & \operatorname{sen} q_1 \operatorname{sen} q_2 & -\cos q_1 & -l_2 \operatorname{sen} q_1 \cos q_2 \\ \cos q_1 \cos q_2 & -\cos q_1 \operatorname{sen} q_2 & -\operatorname{sen} q_1 & l_2 \cos q_1 \cos q_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$U_{22} = \begin{bmatrix} -\cos q_1 \operatorname{sen} q_2 & -\cos q_1 \cos q_2 & 0 & -l_2 \cos q_1 \operatorname{sen} q_2 \\ -\operatorname{sen} q_1 \operatorname{sen} q_2 & -\operatorname{sen} q_1 \cos q_2 & 0 & -l_2 \operatorname{sen} q_1 \operatorname{sen} q_2 \\ -\cos q_2 & \operatorname{sen} q_2 & 0 & -l_2 \cos q_2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Aplicando-se (16) em (15) se obtém:

$$\begin{aligned} \tau_i &= \frac{1}{2}\sum_{j=i}^{N}\sum_{k=1}^{j}\left[\operatorname{Tr}\left(U_{jk}J_jU_{ji}^T\right)\ddot{q}_k\right] + \sum_{j=i}^{N}\sum_{k=1}^{j}\sum_{m=1}^{j}\left[\operatorname{Tr}\left(U_{jkm}J_jU_{ji}^T\right)\dot{q}_k\dot{q}_m\right] \\ &\quad - \sum_{j=i}^{N} m_j g^\star\left(U_{ji}^j\bar{r}_j\right) \end{aligned} \tag{18}$$

onde U_{jkm} é definido como $U_{jkm} = \frac{\partial U_{jk}}{\partial q_m}$ e pode ser interpretado como a influência do movimento das juntas k e m sobre pontos no sistemas de coordenadas j. Para o sistema de visão da cabeça do robô Janus são obtidas as seguintes matrizes:

$$U_{111} = \begin{bmatrix} -\cos q_1 & 0 & \operatorname{sen} q_1 & 0 \\ -\operatorname{sen} q_1 & 0 & -\cos q_1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$U_{211} = \begin{bmatrix} -\cos q_1 \cos q_2 & \cos q_1 \operatorname{sen} q_2 & \operatorname{sen} q_1 & -l_2 \cos q_1 \cos q_2 \\ -\operatorname{sen} q_1 \cos q_2 & \operatorname{sen} q_1 \operatorname{sen} q_2 & -\cos q_1 & -l_2 \operatorname{sen} q_1 \cos q_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$U_{221} = U_{212} = \begin{bmatrix} \operatorname{sen} q_1 \operatorname{sen} q_2 & \operatorname{sen} q_1 \cos q_2 & 0 & l_2 \operatorname{sen} q_1 \operatorname{sen} q_2 \\ -\cos q_1 \operatorname{sen} q_2 & -\cos q_1 \cos q_2 & 0 & -l_2 \cos q_1 \operatorname{sen} q_2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$U_{222} = \begin{bmatrix} -\cos q_1 \cos q_2 & \cos q_1 \operatorname{sen} q_2 & 0 & -l_2 \cos q_1 \cos q_2 \\ -\operatorname{sen} q_1 \cos q_2 & \operatorname{sen} q_1 \operatorname{sen} q_2 & 0 & -l_2 \operatorname{sen} q_1 \cos q_2 \\ \operatorname{sen} q_2 & \cos q_2 & 0 & l_2 \operatorname{sen} q_2 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Sabe-se que para robôs manipuladores, (18) pode ser ainda escrita na forma

$$\tau = D(q)\ddot{q} + H(q,\dot{q})\dot{q} + G(q) \tag{19}$$

onde $D(q)$ é a matriz de inércia generalizada, $H(q,\dot{q})$ é o vetor de forças centrífugas e de Coriolis e $G(q)$ é o vetor de forças gravitacionais.

Assim, aplicando-se (19) com os parâmetros da cabeça de visão do Janus, pode-se escrever:

$$\begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix} \begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} + \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & 0 \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} + \begin{bmatrix} 0 \\ C_{21} \end{bmatrix} \tag{20}$$

com:

$$\begin{aligned} D_{11} &= \theta_1(\cos q_2)^2 + \theta_2(\operatorname{sen} q_2)^2 + \theta_3 \operatorname{sen} q_2 \cos q_2 + \theta_4 \\ D_{12} &= D_{21} = \theta_5 \operatorname{sen} q_2 + \theta_6 \cos q_2 \\ D_{22} &= \theta_1 + \theta_2 \\ H_{11} &= 2(\theta_2 - \theta_1) \operatorname{sen} q_2 \cos q_2 \dot{q}_2 \\ H_{12} &= (\theta_5 \cos q_2 - \theta_6 \operatorname{sen} q_2)\dot{q}_2 \\ H_{21} &= (\theta_1 - \theta_2) \operatorname{sen} q_2 \cos q_2 - \theta_3((\cos q_2)^2 - (\operatorname{sen} q_2)^2)\dot{q}_1/2 \\ C_{21} &= \theta_7 \cos q_2 + \theta_8 \operatorname{sen} q_2 \end{aligned}$$

onde θ_i são as seguintes constantes:

$$\begin{aligned} \theta_1 &= (-I_{2xx} + I_{2yy} + I_{2zz})/2 + 2l_2 m_2 \bar{x}_2 + l_2^2 m_2 \\ \theta_2 &= (I_{2xx} - I_{2yy} + I_{2zz})/2 \\ \theta_3 &= -2l_2 m_2 \bar{y}_2 - 2I_{2xy} \\ \theta_4 &= (I_{2xx} + I_{2yy} - I_{2zz})/2 + I_{1yy} \\ \theta_5 &= I_{2xz} + l_2 m_2 \bar{z}_2 \\ \theta_6 &= I_{2yz} \\ \theta_7 &= m_2 g(\bar{x}_2 + l_2) \\ \theta_8 &= m_2 g \bar{y}_2 \end{aligned}$$

Porém este modelo ainda não é suficiente, uma vez que o acionamento das juntas do robô Janus é feito por tensão, e não se tem sensores disponíveis que permitam medir o torque aplicado em cada junta. Desta forma é necessário estender o modelo para relacionar essas duas grandezas.

O modelo de um motor DC com imã permanente pode ser representado por:

$$V = L_a \frac{di_a}{dt} + i_a R_a + K_a \dot{q}_i \tag{21}$$

$$\tau = K_T i_a \tag{22}$$

onde a L_a é a indutância da armadura, R_a é a resistência de armadura, V é a tensão aplicada, K_a é a constante de força contra-eletromotriz e K_T é a constante de torque.

Segundo (FITZGERALD, 2002) a constante L_a é geralmente pequena e pode ser desprezada. Desta forma, pode-se escrever o torque aplicado ao sistema em função da tensão:

$$\tau_i = \frac{K_{Ti}}{R_{ai}}(V_i - K_{ai}\dot{q}_i) \tag{23}$$

Por fim, para tornar o modelo mais realista, deve-se incluir o efeito das perdas devido ao atrito nos mancais e na transmissão, uma vez que o modelo obtido de (15) não o leva em consideração. Essas perdas são normalmente modeladas como um torque que é apenas função da velocidade da junta $\dot{q}_i$ (GROTJAHN; HEIMANN; ABDELLATIF, 2004). Esta função não linear é escrita como a soma dos termos do atrito viscoso e do atrito seco:

$$\tau_{fi} = r_{1i}\dot{q}_i + r_{2i}sign(\dot{q}_i) \tag{24}$$

onde r_1 é a constante de atrito viscoso, r_2 é a constante de atrito seco, τ_f é o torque gerado pelo o atrito, que se opõem ao movimento. A função $sign(\cdot)$ é definida como:

$$sign(x) = \begin{cases} 0, & x = 0 \\ \frac{x}{|x|}, & x \neq 0 \end{cases}$$

Assim, (20) pode então ser reescrita como:

$$u = \begin{bmatrix} V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix} \begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} + \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & 0 \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} \tag{25}$$

$$+ \begin{bmatrix} 0 \\ C_{21} \end{bmatrix} + \begin{bmatrix} \theta_9\dot{q}_i + \theta_{11}sign(\dot{q}_1) \\ \theta_{10}\dot{q}_2 + \theta_{12}sign(\dot{q}_2) \end{bmatrix}$$

onde:

$$\begin{aligned} \theta_i &= \tfrac{R_{ai}}{K_{Ti}}\bar{\theta}_i \\ \theta_9 &= K_{a1} + \tfrac{R_{a1}}{K_{T1}}r_{11} \\ \theta_{10} &= \tfrac{R_{a1}}{K_{T1}}r_{21} \\ \theta_{11} &= K_{a2} + \tfrac{R_{a2}}{K_{T2}}r_{12} \\ \theta_{12} &= \tfrac{R_{a2}}{K_{T2}}r_{22} \end{aligned} \tag{26}$$

que representa o modelo da cabeça de visão do robô Janus.

5.3 Identificação

Para identificar as constantes que compõem o modelo dinâmico do robô, é necessário reescrever (25) na forma:

$$u = \phi(q, \dot{q}, \ddot{q})\theta$$

onde:

$$\phi = \begin{bmatrix} \phi_{11} & \cdots & \phi_{1(12)} \\ \phi_{21} & \cdots & \phi_{2(12)} \end{bmatrix}$$

$$
\begin{aligned}
\phi_{11} &= \cos q_2^2 \ddot{q}_1 - 2 \operatorname{sen} q_2 \cos q_2 \dot{q}_1 \dot{q}_2 \\
\phi_{12} &= \operatorname{sen} q_2^2 \ddot{q}_1 + 2 \operatorname{sen} q_2 \cos q_2 \dot{q}_1 \dot{q}_2 \\
\phi_{13} &= \operatorname{sen} q_2 \cos q_2 \ddot{q}_1 - (\operatorname{sen} q_2^2 - \cos q_2^2) \dot{q}_1 \dot{q}_2 \\
\phi_{14} &= \ddot{q}_1 \\
\phi_{15} &= \operatorname{sen} q_2 \ddot{q}_2 + \cos q_2 \dot{q}_2^2 \\
\phi_{16} &= \cos q_2 \ddot{q}_2 - \operatorname{sen} q_2 \dot{q}_2^2 \\
\phi_{17} &= \phi_{18} = \phi_{1(10)} = \phi_{1(12)} = \phi_{24} = \phi_{29} = \phi_{2(11)} = 0 \\
\phi_{19} &= \dot{q}_1 \\
\phi_{1(11)} &= sign(\dot{q}_1) \\
\phi_{21} &= \ddot{q}_2 + \operatorname{sen} q_2 \cos q_2 \dot{q}_1^2 \\
\phi_{22} &= \ddot{q}_2 - \operatorname{sen} q_2 \cos q_2 \dot{q}_1^2 \\
\phi_{23} &= (\operatorname{sen} q_2^2 - \cos q_2^2) \dot{q}_1^2 / 2 \\
\phi_{25} &= \operatorname{sen} q_2 \ddot{q}_1 \\
\phi_{26} &= \cos q_2 \ddot{q}_1 \\
\phi_{27} &= \cos q_2 \\
\phi_{28} &= \operatorname{sen} q_2 \\
\phi_{2(10)} &= \dot{q}_2 \\
\phi_{2(12)} &= sign(\dot{q}_2)
\end{aligned}
$$

Assim, as constantes desconhecidas θ_i são lineares com os regressores de dados ϕ, que dependem das posições, velocidades e acelerações das juntas. Deste modo é possível utilizar um método de identificação através da aplicação de uma tensão no robô e da medida do seu deslocamento.

Com estes dados, uma abordagem de mínimos quadrados é utilizado para estimar as constantes do modelo do robô Janus. A solução do estimador de mínimos quadrados é dada pelo a pseudo inversa à esquerda da matriz ϕ de u:

$$\theta = \left(\phi^T \phi\right)^{-1} \phi^T u \tag{27}$$

Para se aplicar a entrada em malha aberta, utiliza-se a arquitetura proposta, excluindo o controlador. A Figura 37 demonstra o sistema utilizando os componentes descritos nos Capítulos 3 e 4 no OROCOS.

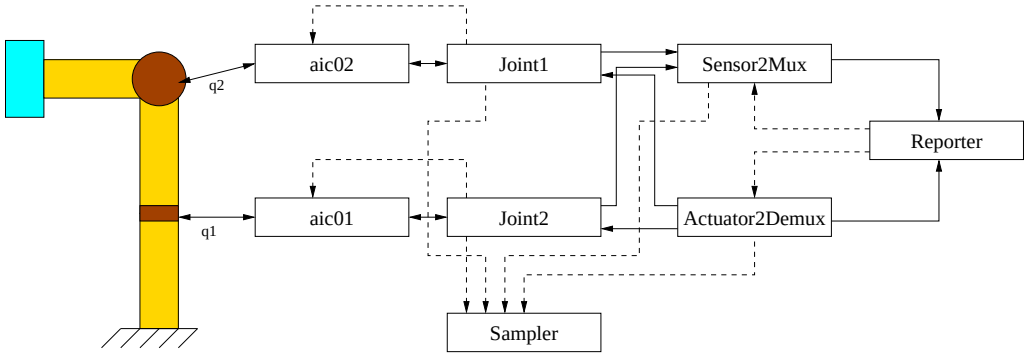
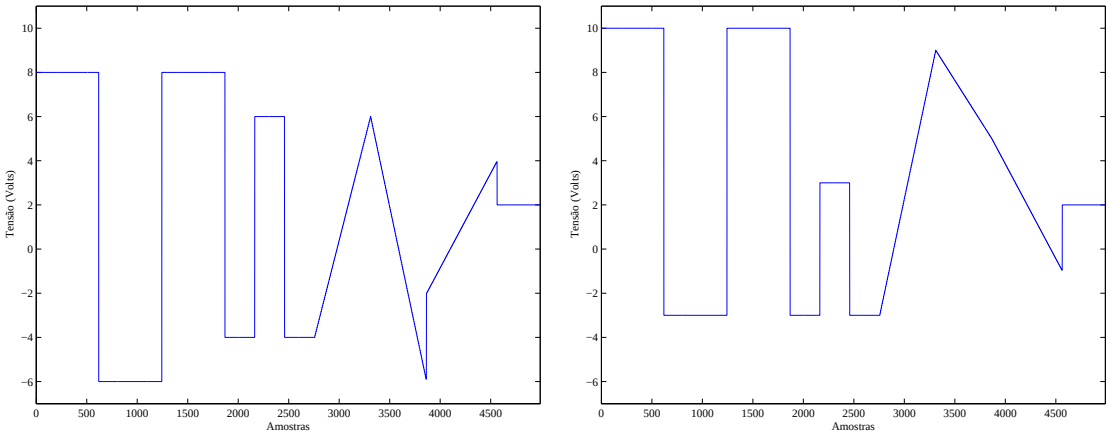


Figura 37: Sistema em malha aberta.

Os componentes `Sensor2Mux` e `Actuator2Demux` interagem com 2 componentes `Joint`. Estes que utilizam os componentes `AIC` para acessar o *hardware* das juntas. O componente `Reporter` é instanciado a partir do componente `FileReporting`, descrito na subseção 2.2.3, e é utilizado para armazenar os dados em um arquivo. O componente `Sampler` gera o evento `Sample` a cada 2 ms, e a entrada é colocada

na porta `act*` de forma periódica também. A Figura 38 mostra a tensão aplicada nas juntas durante este ensaio. Esta entrada é escolhida para excitar o sistema de forma adequada. Um conjunto de 5000 pontos é amostrado para ser utilizado na identificação.



(a) Junta 1. (b) Junta 2.

Figura 38: Tensão aplicada na identificação.

Os *encoders* do robô Janus fornecem uma medida de deslocamento relativo entre um instante de amostragem e outro. Deste modo, dividindo-se pelo período de amostragem é possível obter a velocidade angular média (no período de amostragem) de cada junta $\dot{q}_i$. A posição q_i é obtida através do somatório do deslocamento e a aceleração $\ddot{q}_i$ através da diferença do deslocamento.

Sabe-se que a diferenciação numérica, feita na estimação da aceleração das juntas, amplifica o ruído presente nas medidas e pode inviabilizar a obtenção das constantes. Para eliminar o ruído das medidas utiliza-se um filtro FIR de 10 coeficientes com uma frequência de corte de 36 Hz, o suficiente para eliminar o ruído e sem afetar a dinâmica do sistema. Para um período de amostragem de 2 ms tem-se:

$$\begin{aligned} y[n] \quad = \quad & = (1.2x[n] + 3.26x[n-1] + 8.88x[n-2] + 15.9x[n-3] + \\ & 20.76x[n-4] + 20.76x[n-5] + 15.9x[n-6] + \\ & 8.88x[n-7] + 3.26x[n-8] + 1.2x[n-9]) \times 10^{-2} \end{aligned}$$

A Figura 39 apresenta a resposta em frequência do filtro utilizado. As Figuras 40 e 41 comparam os dados de velocidade e aceleração da junta 1 com o filtro FIR e sem o filtro.

Aplicando método descrito nesta seção, obtém-se os valores expressos em (28). Vale ressaltar que tais valores não correspondem diretamente aos parâmetros das juntas e sim as combinações deles, conforme (26).

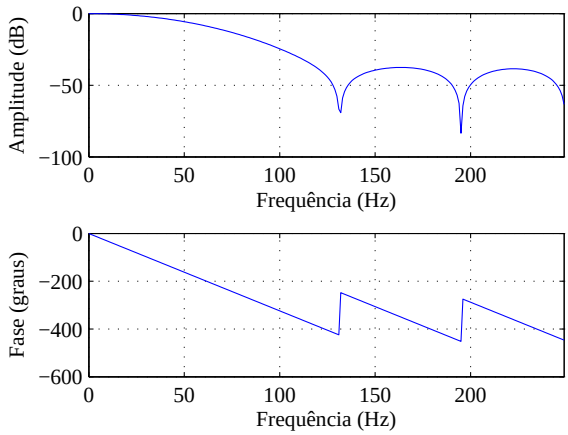
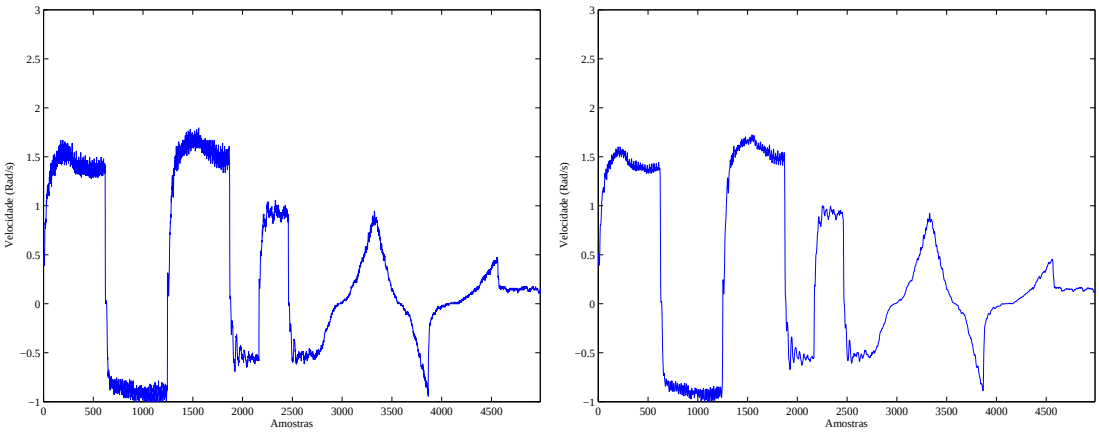
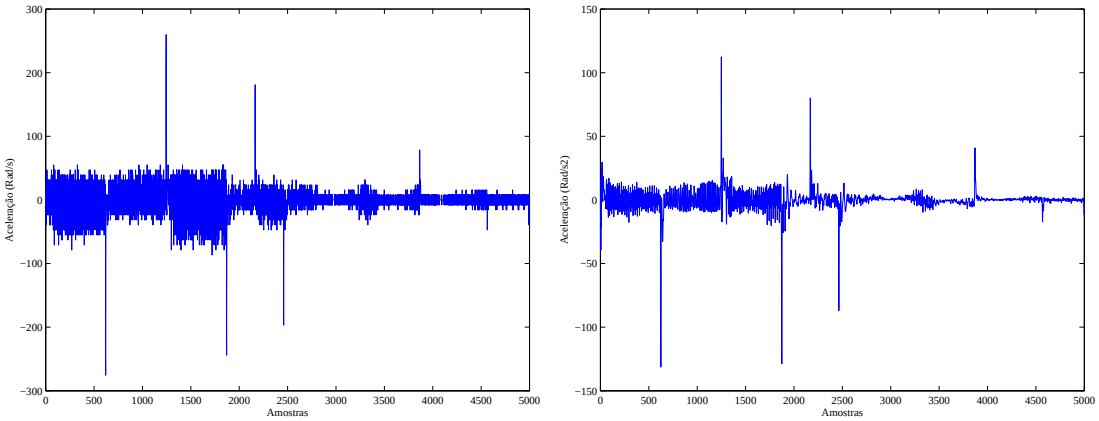


Figura 39: Filtro aplicado aos dados.



(a) Medida. (b) Filtrada.

Figura 40: Velocidade da junta 1.



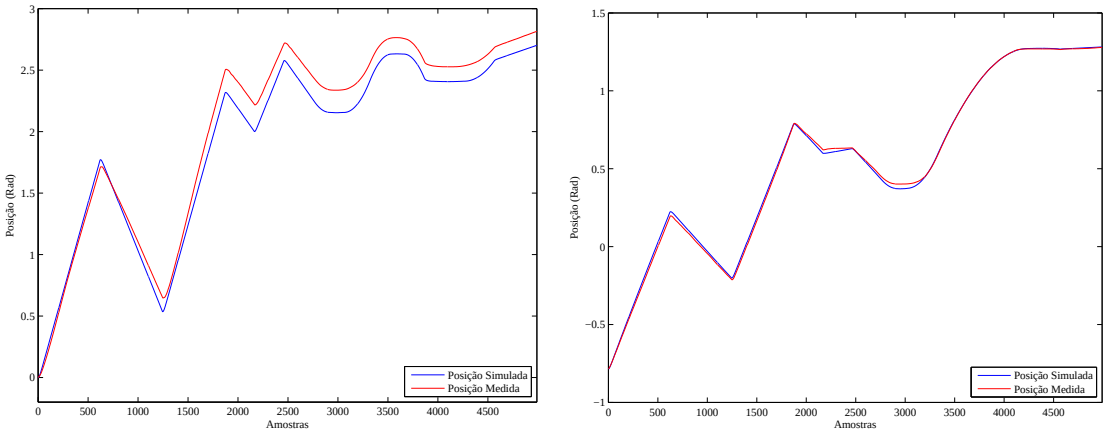
(a) Medida. (b) Filtrada.

Figura 41: Aceleração da junta 1.

$$
\begin{aligned}
\theta_1 &= 0.0396 \\
\theta_2 &= 0.0668 \\
\theta_3 &= -0.0074 \\
\theta_4 &= 0.0223 \\
\theta_5 &= 0.0026 \\
\theta_6 &= 0.0594 \\
\theta_7 &= 1.4129 \\
\theta_8 &= 0.3066 \\
\theta_9 &= 4.5720 \\
\theta_{10} &= 9.2553 \\
\theta_{11} &= 1.3594 \\
\theta_{12} &= 1.1472
\end{aligned} \tag{28}
$$

As Figuras 42 e 43(a) mostram uma comparação entre o modelo utilizando as constantes estimadas e o robô real. O modelo é escrito na forma (29). Então, aplica-se a entrada de tensão utilizada na identificação, vista na Figura 38, no modelo obtido. Utilizando o método de integração numérico Runge-Kutta de quarta ordem com um passo de 2 ms, obtém-se o comportamento do sistema ao longo do tempo, que é similar ao comportamento real do robô.

$$
\begin{aligned}
\begin{bmatrix} \ddot{q}_1 \\ \ddot{q}_2 \end{bmatrix} = \begin{bmatrix} D_{11} & D_{12} \\ D_{21} & D_{22} \end{bmatrix}^{-1} \Bigg(\begin{bmatrix} V_1 \\ V_2 \end{bmatrix} - \begin{bmatrix} H_{11} & H_{12} \\ H_{21} & 0 \end{bmatrix} \begin{bmatrix} \dot{q}_1 \\ \dot{q}_2 \end{bmatrix} \\
- \begin{bmatrix} 0 \\ C_{21} \end{bmatrix} - \begin{bmatrix} \theta_9 \dot{q}_i + \theta_{11} sign(\dot{q}_1) \\ \theta_{10} \dot{q}_2 + \theta_{12} sign(\dot{q}_2) \end{bmatrix} \Bigg)
\end{aligned} \tag{29}
$$

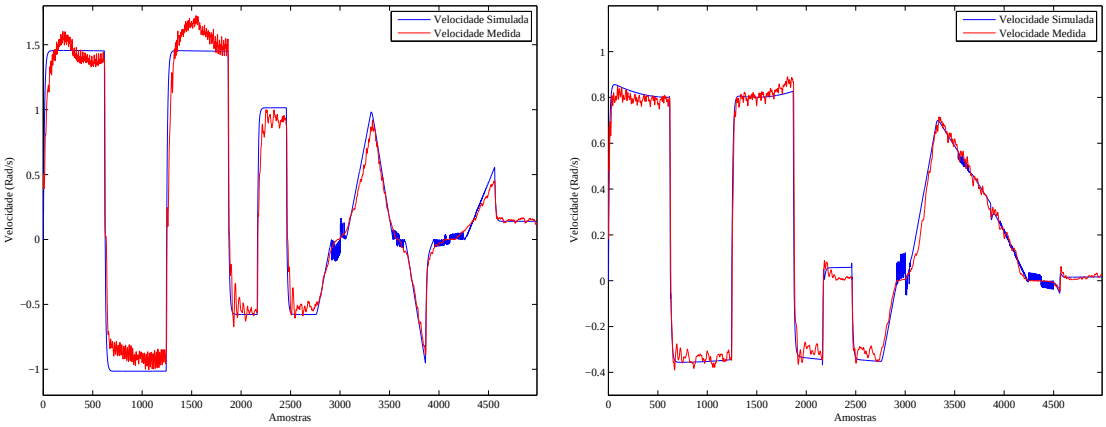


(a) Junta 1. (b) Junta 2.

Figura 42: Posição angular das juntas.

5.3.1 Validação do Modelo

Comparar a simulação do modelo obtido com dados medidos é provavelmente a forma mais usual de se validar um modelo (AGUIRRE, 2004), com o cuidado de não usar os dados utilizados para se obter o modelo. Dessa forma, um novo ensaio foi realizado, aplicando uma nova entrada de tensão, mostrada na Figura 44, nas juntas



(a) Junta 1. (b) Junta 2.

Figura 43: Velocidade angular das juntas.

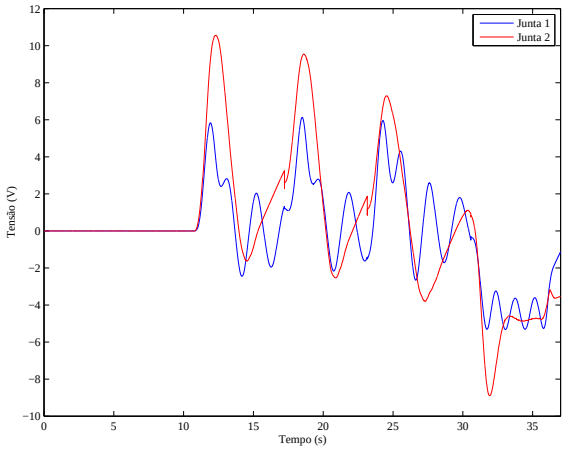
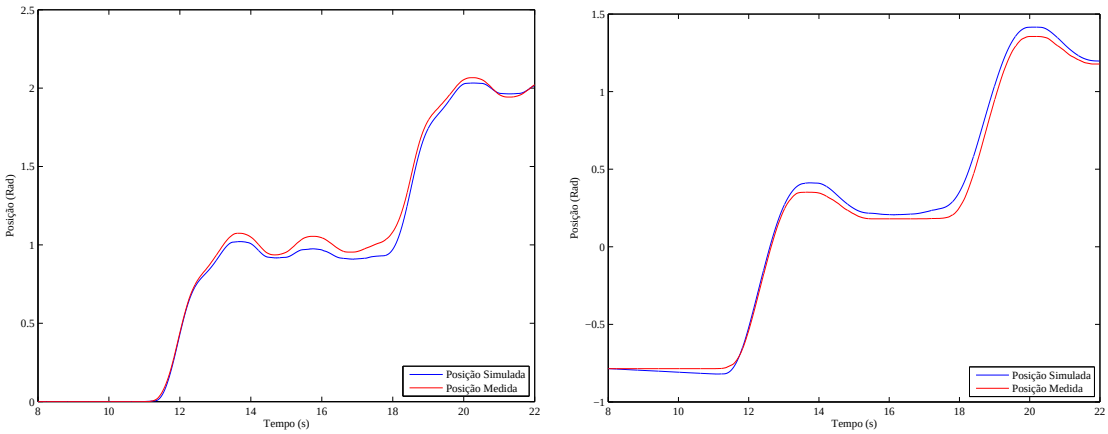


Figura 44: Tensões aplicadas nas juntas.

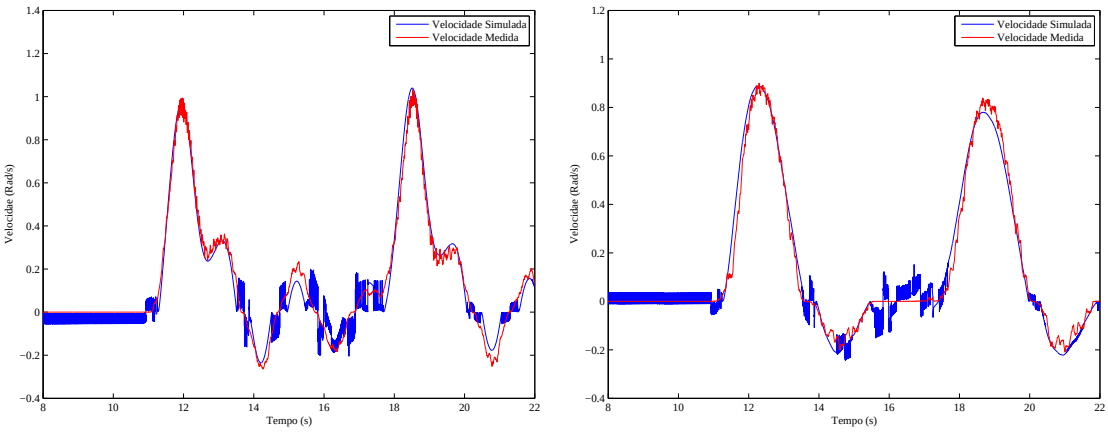
do robô. Essa mesma entrada é aplicada ao modelo (29), para então ser comparada com a trajetória real do robô.

Assim, dado um ponto inicial do sistema, o vetor do estado de (29) é inicializado com o estado medido do robô, e então o modelo é simulado. As Figuras 45 e 46 mostram o resultado obtido na simulação do modelo e o resultado do experimento com o robô real a partir de 8 s.



(a) Junta 1. (b) Junta 2.

Figura 45: Posição angular das juntas.



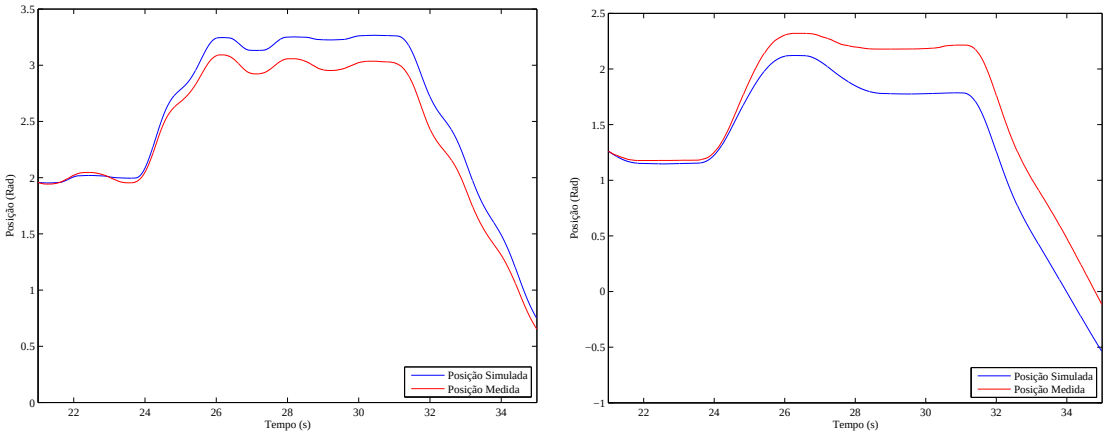
(a) Junta 1. (b) Junta 2.

Figura 46: Velocidade angular das juntas.

Na Figura 45(b) nota-se que, no início da simulação, o modelo começa a se distanciar dos dados reais. Isto se deve ao fato de que o robô se encontra no fim de curso desta junta. Deste modo, uma força mecânica é aplicada ao robô. Esta força não é considerada no modelo, justificando tal discrepância.

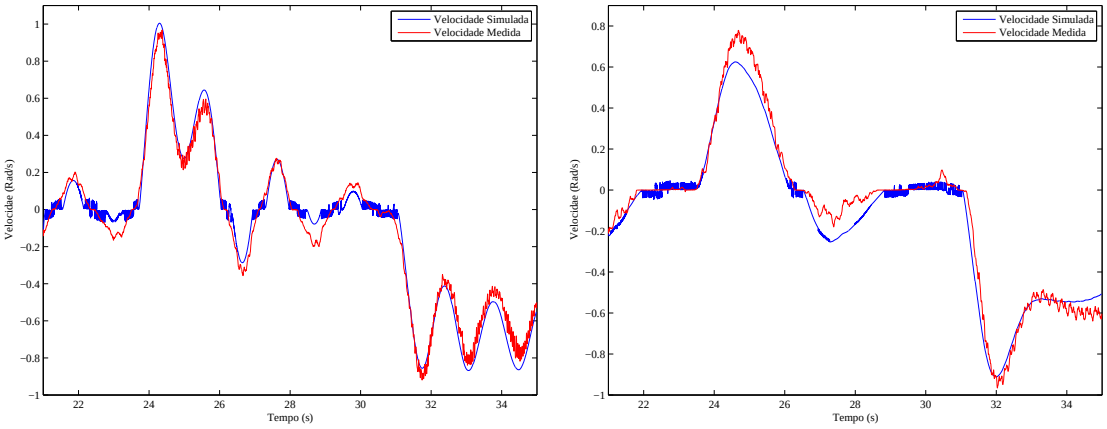
O erro que ocorre nas Figuras 45(a) e 45(b), com o passar do tempo, é esperado. Este erro surge devido à integração do modelo, pois ela vai acumulando os pequenos erros entre o modelo real e o modelo com os parâmetros estimados ao longo do tempo, como por exemplo, eventuais dinâmicas não modeladas. Isto não chega a ser um problema, pois para realizar controle, basta que o modelo funcione por um determinado horizonte temporal. Nas Figuras 46(a) e 46(b) nota-se que a velocidade do modelo se aproxima de forma satisfatória dos dados reais.

As Figuras 47 e 48 mostram o resultado obtido na simulação do modelo e o resultado do experimento com o robô real, em outra janela de tempo, a partir de 22 s. Estas figuras confirmam os resultados anteriores e mostram que o modelo apresenta um comportamento bastante próximo do robô para um horizonte de simulação de aproximadamente 1 s.



(a) Junta 1. (b) Junta 2.

Figura 47: Posição angular das juntas.

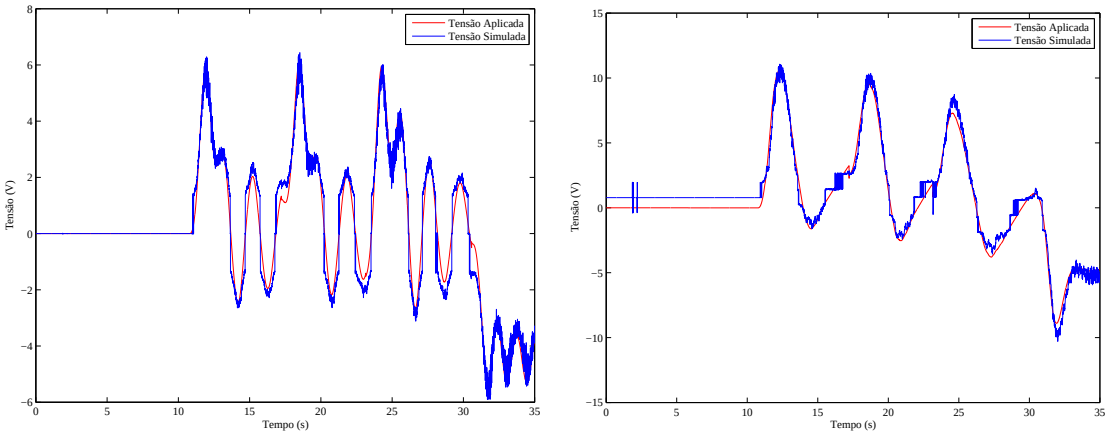


(a) Junta 1. (b) Junta 2.

Figura 48: Velocidade angular das juntas.

Outro teste que deve ser feito para validar as constantes identificadas, é a verificação de (25). Neste caso utiliza-se os dados de posição, velocidade e aceleração estimados e medidos, para então calcular qual é a tensão que deve ser aplicada ao modelo de forma a satisfazer (25). Utilizando os dados obtidos a partir da aplicação da tensão na Figura 44 obtém-se as Figuras 49(a) e 49(b).

Na Figura 49(b) nota-se novamente o efeito do fim de curso da junta 2. O modelo calcula que somente com uma tensão de aproximadamente 1 V é que ela ficará parada naquela posição. Neste caso, onde o somatório dos erros ao longo do tempo não existe, percebe-se que o modelo estima a tensão de forma próxima a tensão aplicada no robô para todo o tempo. Isto valida este modelo para a utilização de técnicas de controle baseadas em modelo, como será feito na próxima seção.



(a) Junta 1. (b) Junta 2.

Figura 49: Tensão.

5.4 Implementação dos Controladores

Para implementar a arquitetura no sistema de visão do robô Janus, é necessário implementar cada um dos blocos da Figura 11. Os blocos `Sensor` e `Actuator` são utilizados de forma similar ao caso em malha aberta, descrita na seção 5.3, pelo o diagrama de blocos da Figura 37. Para a geração da referência, utiliza-se uma modificação do gerador de trajetória do OROCOS: `nAxisGeneratorPos`, descrito na subseção 2.2.4. Essa modificação é feita para que ele gere, também, uma aceleração de referência. Para a sua utilização, também é necessário incluir um segundo componente que gere um vetor com a posição, a partir do sinal `sensor`, e que forme o sinal `reference` a partir dos dados de referência. Esse componente é chamado `BridgeRef` pois fará uma ponte entre o gerador de trajetória do OROCOS e a arquitetura proposta. Ele é mostrado na Figura 50.

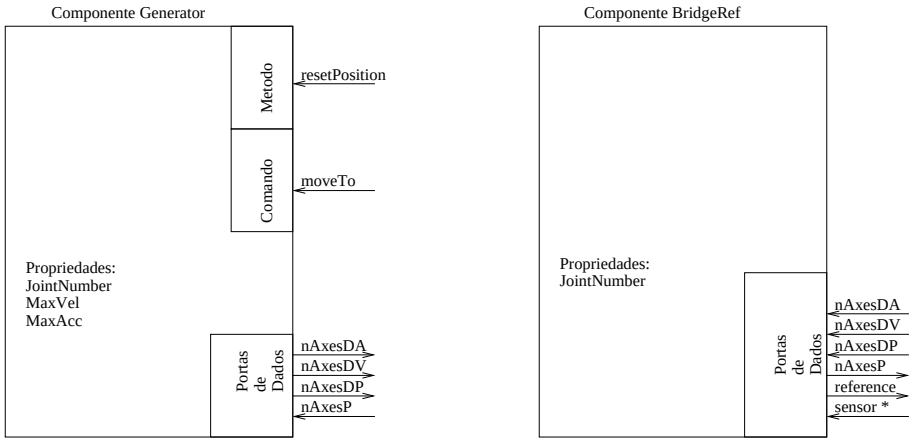


Figura 50: Modelo de componente `BridgeRef`.

O componente `BridgeRef` pega o valor da posição dos sensores através da porta `sensor*` e coloca a posição atual do robô na porta `nAxesP`. Para o controlador, ele agrupa as portas `nAxesDP`, `nAxesDV` e `nAxesDA` na sua porta `reference`. Ele é ativado quando um novo dado é escrito na porta `sensor*`.

Para o armazenamento de dados, é utilizado o componente `FileReporting`, des-

crito na subseção 2.2.3. Nesta implementação, ele é conectado as portas de dados do componente `Controller` para armazenar os valores de referência, saída e atuação. Os comandos do usuário são passados para o sistema utilizando o `TaskBrowser`, descrito na subseção 2.2.1. Ele é conectado ao componente `nAxisGeneratorPos` que gera a trajetória do sistema.

O sistema é inicializado utilizando o componente `DeploymentComponent`, descrito na subseção 2.2.2. Através de um *script*, as portas de dados são conectadas entre si, bem como os componentes são conectados como pares. Ele também é utilizado para descrever a atividade e a prioridade dos componentes. O apêndice E contém o *script* de inicialização do sistema para o controle de torque calculado. Nele os componentes são inseridos no sistema e seguem os passos de inicialização da Figura 7, no Capítulo 2.

Os demais componentes são instanciados conforme o tipo de controle descrito a seguir.

5.4.1 Controlador Independente por Junta

Para um controle independente por juntas, é necessário utilizar o `Controller2PID` junto com dois blocos PID para implementar o controle. O diagrama de blocos da Figura 51 demonstra a comunicação entre os componentes, bem como as suas funções, onde os blocos hachurados ressaltam os componentes com comportamento de tempo real.

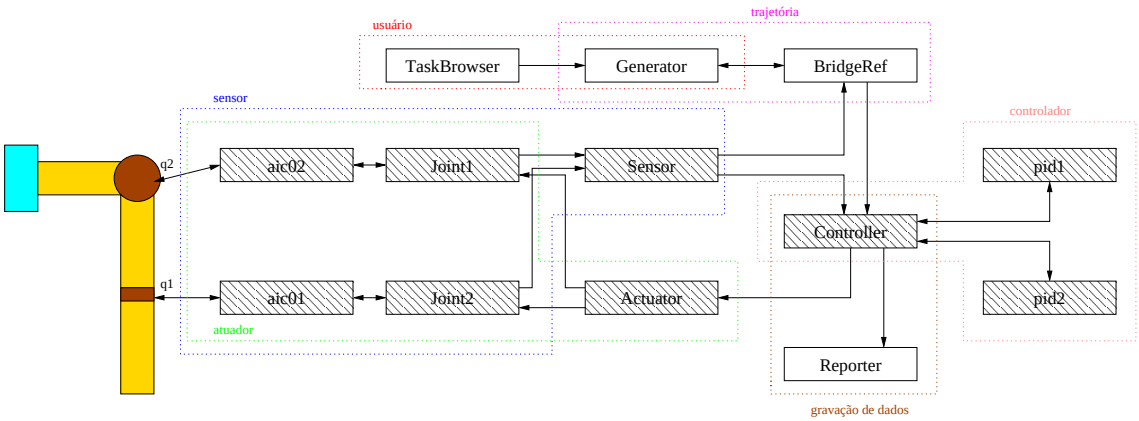


Figura 51: Diagrama de blocos do sistema com um controlador independente por junta.

A Tabela 8 resume a atribuição de atividades e prioridades dos componentes para esta implementação, enquanto a Figura 52 mostra o *timeline* dos componentes do sistema fora de escala. Os tempos apresentados nessa figura representam as médias dos tempos nos ensaios realizados. Note que os componentes `TaskBrowser`, `Reporter` e `Generator` não estão presentes no *timeline* pois eles não funcionam de forma síncrona com o ciclo de controle.

Um ciclo de controle do sistema funciona da seguinte forma:

- Em 0, o componente `Sampler` irá ser ativado, dando início ao ciclo de controle através do evento `Sample`. Os componentes `Joint1`, `Joint2`, `Sensor`, `Actuator` e `Controller` reagem de forma síncrona ao evento com prioridade 1. Inicialmente o componente `Joint1` atende ao evento, chamando o método

Tabela 8: Configuração do sistema.

Componente	Modelo	Atividade	Prioridade	Período
`Sampler`	`Sampler`	`PeriodicActivity`	1	10 ms
`aic01`	`AIC`	`SequentialActivity`	—	—
`aic02`	`AIC`	`SequentialActivity`	—	—
`Joint1`	`Joint`	`SequentialActivity`	—	—
`Joint2`	`Joint`	`SequentialActivity`	—	—
`Sensor`	`Sensor2Mux`	`NonPeriodicActivity`	2	—
`BridgeRef`	`BridgeRef`	`NonPeriodicActivity`	97	—
`Generator`	`nAxisGenPos`	`PeriodicActivity`	98	10ms
`Controller`	`Controller2PID`	`NonPeriodicActivity`	3	—
`pid1`	`PID`	`SequentialActivity`	—	—
`pid2`	`PID`	`SequentialActivity`	—	—
`Actuator`	`Actuator2Demux`	`NonPeriodicActivity`	3	—
`Reporter`	`FileReporting`	`PeriodicActivity`	100	10ms
`TaskBrowser`	`TaskBrowser`	`NonPeriodicActivity`	255	—

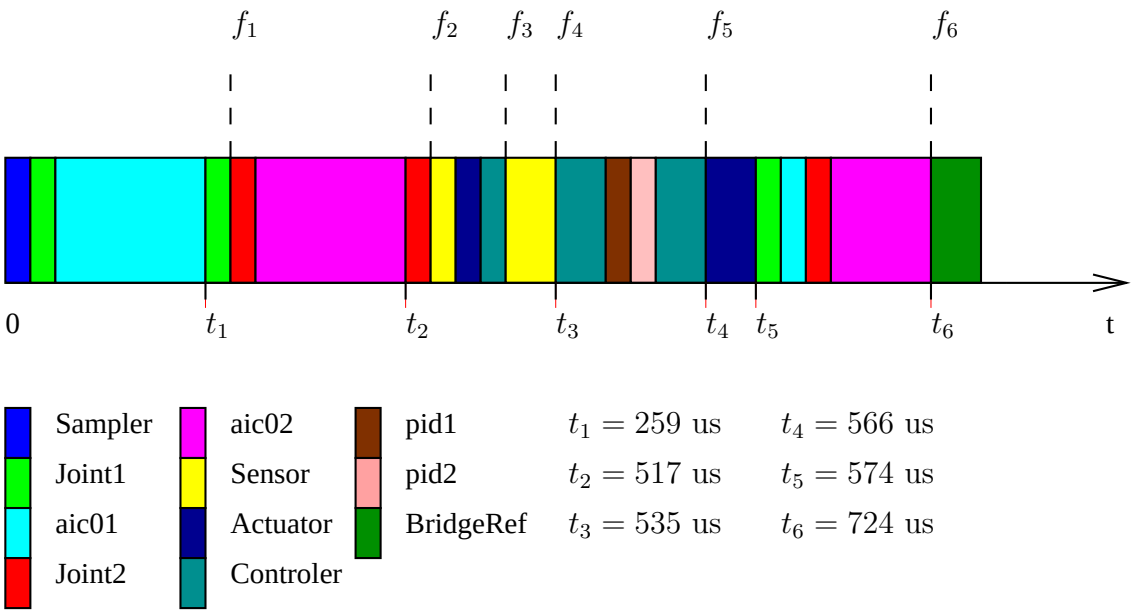


Figura 52: *Timeline* do sistema de controle independente por junta.

`sensorRead` do componente `aic01`. Esse método é instantaneamente executado, gerando uma leitura no barramento, demorando o tempo necessário para o tráfego de um quadro remoto, um quadro de dados com 8 *bytes*, mais a leitura do *encoder*. Esse método atualizará as portas do componente `Joint1`.

- Em f_1, as portas `Displacement` e `Position` de `Joint1` são atualizadas, o que gera dois eventos que devem ser atendidos pelo o componente `Sensor` de forma assíncrona. Como `Sensor` tem prioridade 2, ele fica aguardando todos os processos com prioridade 1 executarem. A seguir, o componente `Joint2`, com prioridade 1, repete o processo, e os demais componentes (`Sensor`, `Actuator` e `Controller`) executam as suas funções `sample_now` devido à ocorrência do evento `Sample`.

- Em f_3, todos os componentes com prioridade 1 já executaram, então o componente `Sensor` reage aos eventos gerados em f_1 e f_2, e a seguir, a porta `sensor` é atualizada. Isto gera um evento ao qual os componentes `Controller` e `BridgeRef` devem reagir.

- Em f_4, como a prioridade do `Controller` é maior que a do `BridgeRef`, o `Controller` irá executar. Ele escreverá os dados nas portas dos dois PIDs, o que fará com que eles executem com a prioridade do `Controller`, devido à atividade sequencial dos mesmos. A seguir, eles geram eventos que forçam o `Controller` a executar novamente, atualizando a porta `act`. Esse fato gera um evento ao qual o `Actuator` deve reagir.

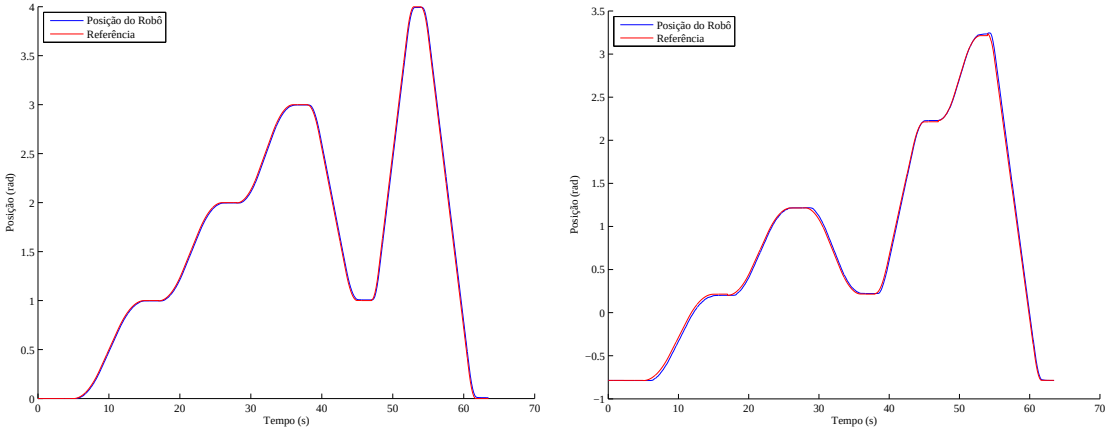
- Em f_5, como a prioridade do `Actuator` é maior que a do `BridgeRef`, o `Actuator` irá executar. Ele irá desmontar o vetor de atuação, e escrever cada valor de atuação no seu respectivo componente `Joint`. A seguir, o evento `Act` é gerado, causando a execução dos componentes `Joint` de forma síncrona. Os componentes `Joint` irão aplicar a tensão nos componentes `AIC`. Note que somente o componente `aic02` demora para realizar está tarefa. Isso ocorre devido à implementação do *driver* PCIcan. Na primeira escrita no barramento, esse estará sem nenhum dado. Na segunda escrita, o *driver* ficará fazendo *polling*, esperando que a primeira mensagem libere o barramento. Este tempo corresponde a um quadro de dados com 8 *bytes*.

- Em f_6, nada com maior prioridade que `BridgeRef` precisa executar, e então, esse é executado. A seguir, o processador fica livre para executar os demais componentes do sistema.

As Figuras 53(a) e 53(b) demonstram o comportamento do robô ao longo de um ensaio. Nota-se que a saída segue a referência com uma precisão satisfatória.

5.4.2 Controlador de Torque Calculado

A filosofia de componentes do OROCOS permite que uma nova implementação de controle seja realizada simplesmente trocando um determinando componente. Assim, o componente `Controller2PID` da subseção anterior é trocado pelo componente `Controller2PIDCT`. Para essa estratégia de controle, é necessário se ter o modelo dinâmico do robô. Assim, o modelo identificado neste capítulo é implementado na extensão do componente `Model` chamada `ModelJH`. Desta forma, a Tabela 8 recebe as duas linhas da Tabela 9.



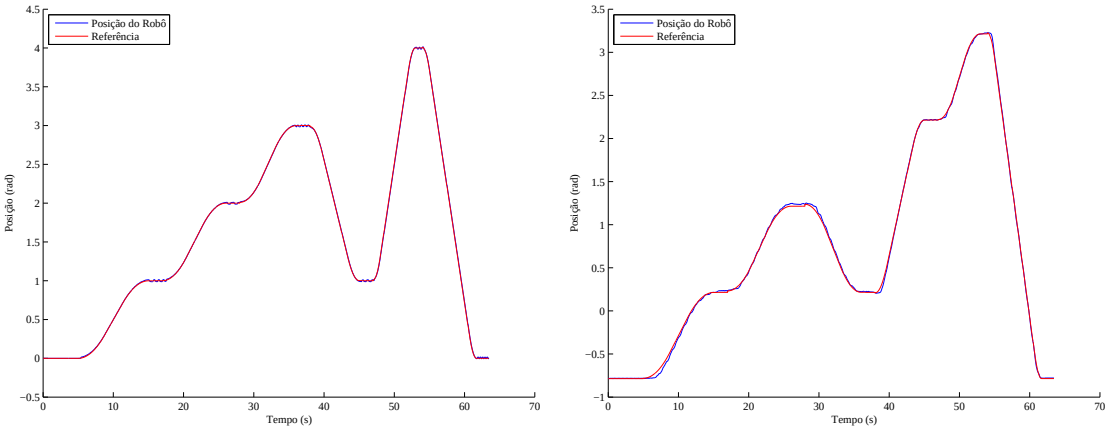
(a) Junta 1. (b) Junta 2.

Figura 53: Posição utilizando controlador PID independente por junta.

Tabela 9: Componentes adicionados.

Componente	Modelo	Atividade	Prioridade	Período
`Controller`	`Controller2PIDCT`	`NonPeriodicActivity`	3	—
`model`	`ModelJH`	`SequentialActivity`	—	—

O funcionamento do sistema continua basicamente o mesmo, e por isso não será novamente explicado. Ao refazer as medidas temporais, nota-se que há um acréscimo de 10 us no tempo do controlador, devido ao cálculo do modelo do robô. As Figuras 54(a) e 54(b) demonstram o comportamento do robô ao longo de um ensaio. De forma similar ao controlador anterior, o resultado é satisfatório.



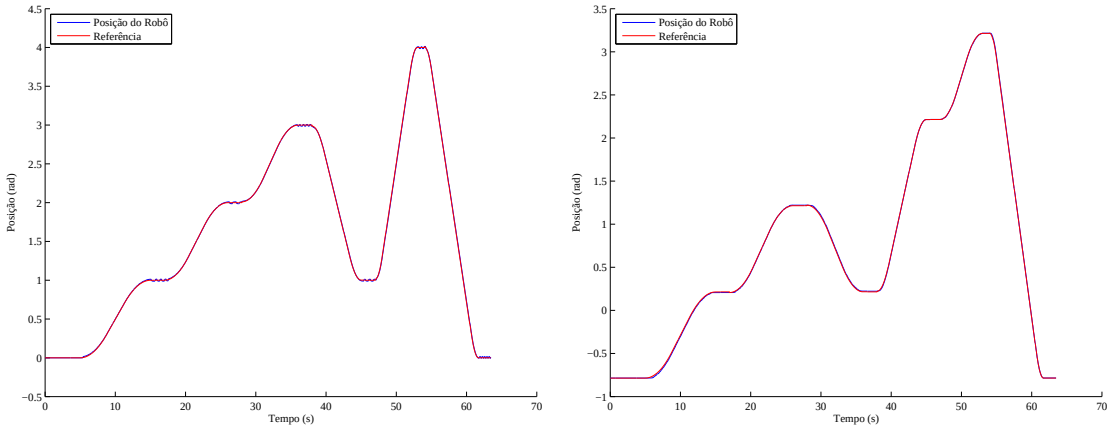
(a) Junta 1. (b) Junta 2.

Figura 54: Posição utilizando controlador com torque calculado.

5.4.3 Controlador com `FeedForward`

Este controle é implementado através da substituição do componente `Controller2PIDCT` pelo componente `Controller2PIDFF`, sem outras mudanças. As Figuras 55(a) e

55(b) apresentam a posição angular das juntas no ensaio. Nota-se que a posição segue a referência desejada.



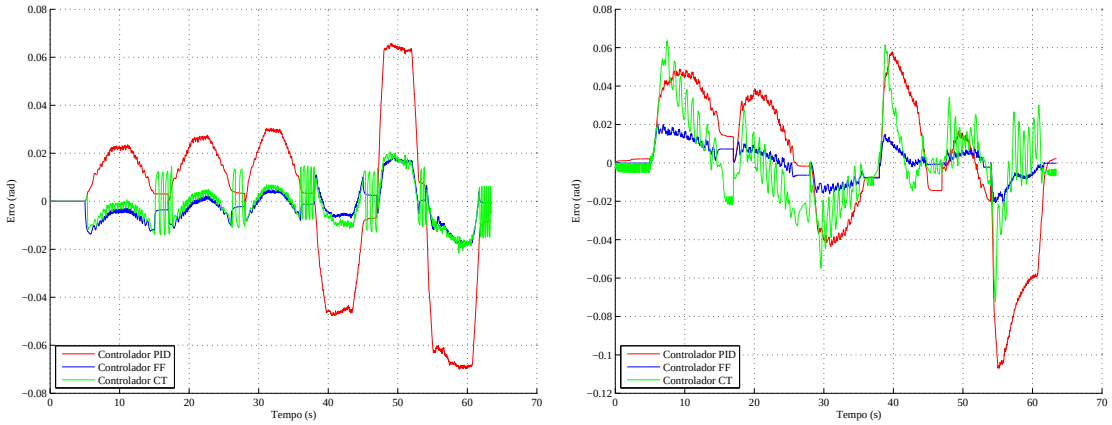
(a) Junta 1. (b) Junta 2.

Figura 55: Posição utilizando controlador com *FeedForward*.

5.5 Conclusão

Em relação ao modelo estimado, a resposta do sistema obtida experimentalmente e a resposta obtida da simulação do modelo, permite concluir que o modelo representa bem o sistema real. Fato que é comprovado nas estratégias de controle que o utilizam.

Em relação à implementação da arquitetura, nota-se que a mesma ficou flexível, onde ela foi utilizada por três estratégias de controles diferentes, além de ter sido utilizada em malha aberta para a obtenção dos dados da identificação. Note que em todos os passos do controle, o escalonamento em tempo real é garantido. As Figuras 56(a) e 56(b) comparam os desempenhos dos controladores utilizados, através da demonstração do erro entre a referência desejada e a posição do robô.



(a) Junta 1. (b) Junta 2.

Figura 56: Erro ao longo do tempo.

Para não avaliar o desempenho dos controladores de forma apenas visual e subjetiva, os critérios IAE, ISE, ITAE e ITSE são apresentados nas Tabelas 10 e 11.

Como era esperado, os controladores de torque calculado e com *FeedForward* apresentam um melhor desempenho que o controlador com um PID independente por junta.

Note que para uma avaliação mais genérica sobre as estratégias de controle, seria necessário a implementação dos controladores utilizando ganhos ótimos, a partir de um critério, nos controladores PID. É possível que um controlador esteja apenas com os ganhos mal ajustados, uma vez que eles foram sintonizados de forma experimental.

Tabela 10: Critérios de desempenho para junta 1.

	IAE	ISE	ITAE	ITSE
PID	0.02488325517	0.00109109795	0.03298157035	0.00164850109
CT	0.00691766951	0.00007775767	0.00880522929	0.00010901467
FF	0.00577014140	0.00006052742	0.00692017596	0.00008206409

Tabela 11: Critérios de desempenho para junta 2.

	IAE	ISE	ITAE	ITSE
PID	0.02746701262	0.00131288324	0.03108200523	0.00171017576
CT	0.01524780852	0.00042537226	0.01441801559	0.00037510700
FF	0.00667365531	0.00007236350	0.00631226583	0.00006480212

6 CONCLUSÃO

Foi proposta neste trabalho uma arquitetura aberta para controle de robôs manipuladores. Isso foi feito com a implementação de classes bases que definem a interface da arquitetura. Prosseguindo, foram implementadas extensões dessas classes para agrupar N juntas de um robô utilizando a interface do componente `Joint`, e três tipos de controladores diferentes. Ao definir a interface `Joint`, abstraiu-se o *hardware* da junta, possibilitando que qualquer placa de acionamento seja utilizada na arquitetura.

Paralelamente foi implementada a placa de acionamento AIC, que junto com a arquitetura proposta, demonstram o funcionamento da mesma quando implementada em um robô real. O sistema utilizado foi o sistema de visão do robô Janus, para o qual foram desenvolvidos os modelos cinemático e dinâmico. Este último também teve os seus parâmetros identificados para a implementação das estratégias baseadas no modelo.

A utilização do *framework* OROCOS cumpriu o seu papel, diminuindo o tempo de implementação da arquitetura ao oferecer um conjunto de primitivas, as quais foram utilizadas no desenvolvimento da arquitetura. Além do que, possibilitou a utilização de componentes já implementados por outros usuários do OROCOS, como o gerador de trajetória e o componente que grava os arquivos de dados. Por outro lado, o tempo de aprendizado do *framework* OROCOS foi consideravelmente grande, devido à sua pobre documentação. Este trabalho além de apresentar como contribuição, uma documentação dos conceitos utilizados no OROCOS, ele define também uma arquitetura para a implementação de controladores, com base nas primitivas disponibilizadas pelo OROCOS, servindo como uma base de integração para novos desenvolvedores.

Por ser um *framework* baseado em componentes, a arquitetura ficou modular. Isto deixa os blocos da arquitetura independentes entre si, permitindo a exclusão de alguns deles quando desejado (como no ensaio em malha aberta), ou a substituição de parte do sistema (como na variação do controlador de torque calculado para o controlador com *feedforward*), sem que os demais componentes precisem de alterações. Isto permite que um usuário se concentre exatamente no seu ponto de interesse, sem que ele precise perder tempo nos demais detalhes do sistema.

Em relação à placa AIC, essa cumpriu bem o seu papel para o acionamento do sistema de visão do robô Janus. A utilização do barramento CAN, devido a sua baixa latência, permitiu atingir melhores estimativas de velocidade, e a obtenção de um período de até 2 ms em um ciclo de sensoriamento/atuação. Teoricamente este período pode ser ainda diminuído. Neste ponto, a implementação do *driver* PCIcan foi fundamental, uma vez que ela permitiu o acesso ao *hardware* sem precisar passar

pelo o sistema operacional. Isto garante um escalonamento de tempo real as tarefas.

Por fim, para o robô Janus, este trabalho se mostrou adequado demonstrando que a arquitetura pode ser aplicada em outros robôs.

Para trabalhos futuros deseja-se aplicar esta arquitetura nos braços do robô Janus e em outros robôs. Para outros robôs, deve-se implementar um componente de acordo com a interface de `Joint`, ou utilizar a placa AIC. Também deseja-se utilizar a biblioteca BFL do OROCOS para realizar uma nova identificação dos parâmetros do modelo do robô Janus, bem como utilizar um filtro de Kalman estendido para estimar a velocidade e assim melhorar o desempenho dos controladores, em especial o de torque calculado. Por fim, pretende-se continuar com o desenvolvimento da arquitetura, através da implementação de bibliotecas para controle, onde diversos tipos de componente `Controller` serão implementados, bem como o desenvolvimento de uma biblioteca com diferentes componentes de placas de acionamento, permitindo que o usuário possa escolher a opção mais adequada a sua necessidade, economizando tempo de projeto.

REFERÊNCIAS

AGUIRRE, L. A. **Introdução à Identificação de Sistemas**. Minas Gerais: UFMG, 2004.

ASTROM, K. J.; WITTENMARK, B. **Computer Controlled Systems**: theory and design. Indianapolis: Prentice Hall Professional Technical Reference, 1984.

BAILO, C.; ALDERSON, G.; YEN, J. **OMAC Requirements V1.1**. Detroit: General Motors Corporation, 1994.

BOSCH, R. **CAN Specification Version 2.0**. Stuttgart: Robert Bosch GmbH, 1991.

BRUYNINCKX, H. Open robot control software: the orocos project. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 2001, Seoul. **Proceedings...** Piscataway: IEEE Press, 2001. v.3, p.2523–2528.

CIA. **CiA Draft Standard 102 Version 2.0 CAN Physical Layer for Industrial Applications**. Nuremberg: CAN in AUTOMATION, 1994.

COMPONENTS for control the OROCOS project. Disponível em: http://www.orocos.org/ocl/. Acesso em: 05 ago. 2009.

CRAIG, J. J. **Introduction to Robotics Mechanics and Control**. 2.ed. Boston: Addison-Wesley, 1989.

DENAVIT, J.; HARTENBERG, R. S. A kinematic notation for lower-pair mechanisms based on matrices. **Trans ASME Journal of Applied Mechanics**, New York, v.23, p.215–221, 1955.

FITZGERALD, A. E. **Electric machinery**. New York: McGraw-Hill, 2002.

FORD, W. What is an open architecture robot controller? In: IEEE INTERNATIONAL SYMPOSIUM ON INTELLIGENT CONTROL, 1994, Columbus. **Proceedings...** Piscataway: IEEE Press, 1994. p.27–32.

FU, K. S.; GONZALES, R. C.; LEE, C. S. G. **Robotics Control, Sensing, Vision and Intelligence**. New York: McGraw-Hill, 1987. (Industrial Engineering Series).

GASPAR, M. D. **Uma Biblioteca Configurável para Controle Tempo Real de Robôs Manipuladores**. 2003. 129 f. Dissertação (Mestrado), Universidade Federal de Minas Gerais, Belo Horizonte, 2003.

GROTJAHN, M.; HEIMANN, B.; ABDELLATIF, H. Identification of Friction and Rigid-Body Dynamics of Parallel Kinematic Structures for Model-Based Control. **Multibody System Dynamics**, Dordrecht, v.11, p.273–294, 2004.

HAYES-ROTH, F.; ERMAN, L.; TERRY, A.; HAYES-ROTH, B. Domain-specific software architectures: distributed intelligent control and management. In: IEEE SYMPOSIUM ON COMPUTER-AIDED CONTROL SYSTEM DESIGN, 1992, Napa. **Proceedings...** Piscataway: IEEE Press, 1992. p.117–128.

HEMERLY, E. M. **Controle por Computador de Sistemas Dinâmicos**. São Paulo: Edgard-Blücher, 1996.

LAGES, W. F.; BRACARENSE, A. Q. Robot Retrofitting: a perspective to small and medium size entreprizes. In: AUSTRIAN BRAZILIAN AUTOMATION DAY, 3., 2003, São Bernardo do Campo. **Proceedings...** São Paulo: RECOPE/MANET, 2003. p.1–11.

LI, W.; HENRIK, C. I.; ANDERS, O. Architecture and its implementation for robots to navigate in unknown indoor environments. **Chinese Journal of Mechanical Engineering**, Beijing, v.18, n.3, p.366–370, 2005.

LIANDONG, P.; XINHAN, H. Implementation of a PC-based Robot Controller with Open Architecture. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND BIOMIMETICS, 2004, Shenyang. **Proceedings...** Piscataway: IEEE Press, 2004. p.790–794.

LLOYD, J. E. **RCCL Reference Manual**. Montréal: McGill Research Centre for intelligent Machines, McGill University, 1992.

LLOYD, J. E. **RCI Reference Manual**. Montréal: McGill Research Centre for intelligent Machines, McGill University, 1992.

MICROCHIP. **DSPIC30F Family Reference Manual**. Chandler: Microchip Technology Incorporated, 2006.

MICROCHIP. **DSPIC30F4011/4012 Data Sheet**. Chandler: Microchip Technology Incorporated, 2007.

MICROCHIP. **DSPIC30F4011/4012 Rev. A2/A3 Silicon Errata**. Chandler: Microchip Technology Incorporated, 2008.

MILLER, D.; LENNOX, R. An object-oriented environment for robot system architectures. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND AUTOMATION, 1990, Cincinnati. **Proceedings...** Piscataway: IEEE Press, 1990. v.1, p.352–361.

NATIONAL. **LM628/LM629 Precision Motion Controller**. Santa Clara: National Semiconductor Corporation, 2003.

OROCOS Bayesian Filtering Library Think Smarter. Disponível em: http://www.orocos.org/bfl/. Acesso em: 05 ago. 2009.

OROCOS Kinematics and Dynamics Smarter in Motion. Disponível em: http://www.orocos.org/kdl/. Acesso em: 05 ago. 2009.

OROCOS Real Time Toolkit Smarter Real Time. Disponível em: http://www.orocos.org/rtt/. Acesso em: 05 ago. 2009.

OSACA Handbook Version 2.0. Disponível em: http://www.osaca.org/documentation_and_software/handbook.htm. Acesso em: 05 ago. 2009.

PIRJANIAN, P.; HUNTSBERGER, T. L.; TREBI-OLLENNU, A.; AGHAZARIAN, H.; DAS, H.; JOSHI, S. S.; SCHENKER, P. S. Campout: a control architecture for multirobot planetary outposts. In: THE INTERNATIONAL SOCIETY FOR OPTICAL ENGINEERING SPIE, 2000, Boston, USA. **Proceedings...** Bellingham: SPIE, 2000. v.4196, n.1, p.221–230.

PRITSCHOW, G.; ALTINTAS, Y.; JOVANE, F.; KOREN, Y.; MITSUISHI, M.; TAKATA, S.; BRUSSEL, H. van; WECK, M.; YAMAZAKI, K. Open Controller Architecture - Past, Present and Future. **CIRP Annals - Manufacturing Technology**, New York, v.50, n.2, p.463–470, 2001.

RTAI - The Realtime Application Interface for Linux from DIAPM. Disponível em: https://www.rtai.org/. Acesso em: 05 ago. 2009.

SCIAVICCO, L.; SICILIANO, B. **Modelling and Control of Robot Manipulators**. 2nd ed. New York: Springer, 2005.

SOETENS, P. **The TaskBrowser Component**. Leuven: Flanders' Mechatronics Technology Centre, 2007.

SOETENS, P. **The Orocos Component Builder's Manual**. Leuven: Flanders' Mechatronics Technology Centre, 2009.

SOETENS, P. **The Deployment Component**. Leuven: Flanders' Mechatronics Technology Centre, 2009.

SOETENS, P. **The Reporting Component**. Leuven: Flanders' Mechatronics Technology Centre, 2009.

STEWART, D.; VOLPE, R.; KHOSLA, P. Design of dynamically reconfigurable real-time software using port-based objects. **IEEE Transactions on Software Engineering**, Piscataway, v.23, n.12, p.759–776, Dec. 1997.

STROUSTRUP, B. **The C++ Programming Language**. Boston: Addison-Wesley Longman Publishing, 2000.

SWEVERS, J.; VERDONCK, W.; DE SCHUTTER, J. Dynamic Model Identification for Industrial Robots. **Control Systems Magazine**, Piscataway, v.27, n.5, p.58–71, Oct. 2007.

TAVARES, D. M.; AROCA, R. V.; PAULA CAURIN, G. A. de. Upgrade of a SCARA Robot Using OROCOS. In: IASTED INTERNATIONAL CONFERENCE ON ROBOTICS AND APPLICATIONS, 13., 2007, Würzburg. **Proceedings...** Calgary: ACTA Press, 2007. p.252–257.

THE FREERTOS Project. Disponível em: http://www.freertos.org. Acesso em: 05 ago. 2009.

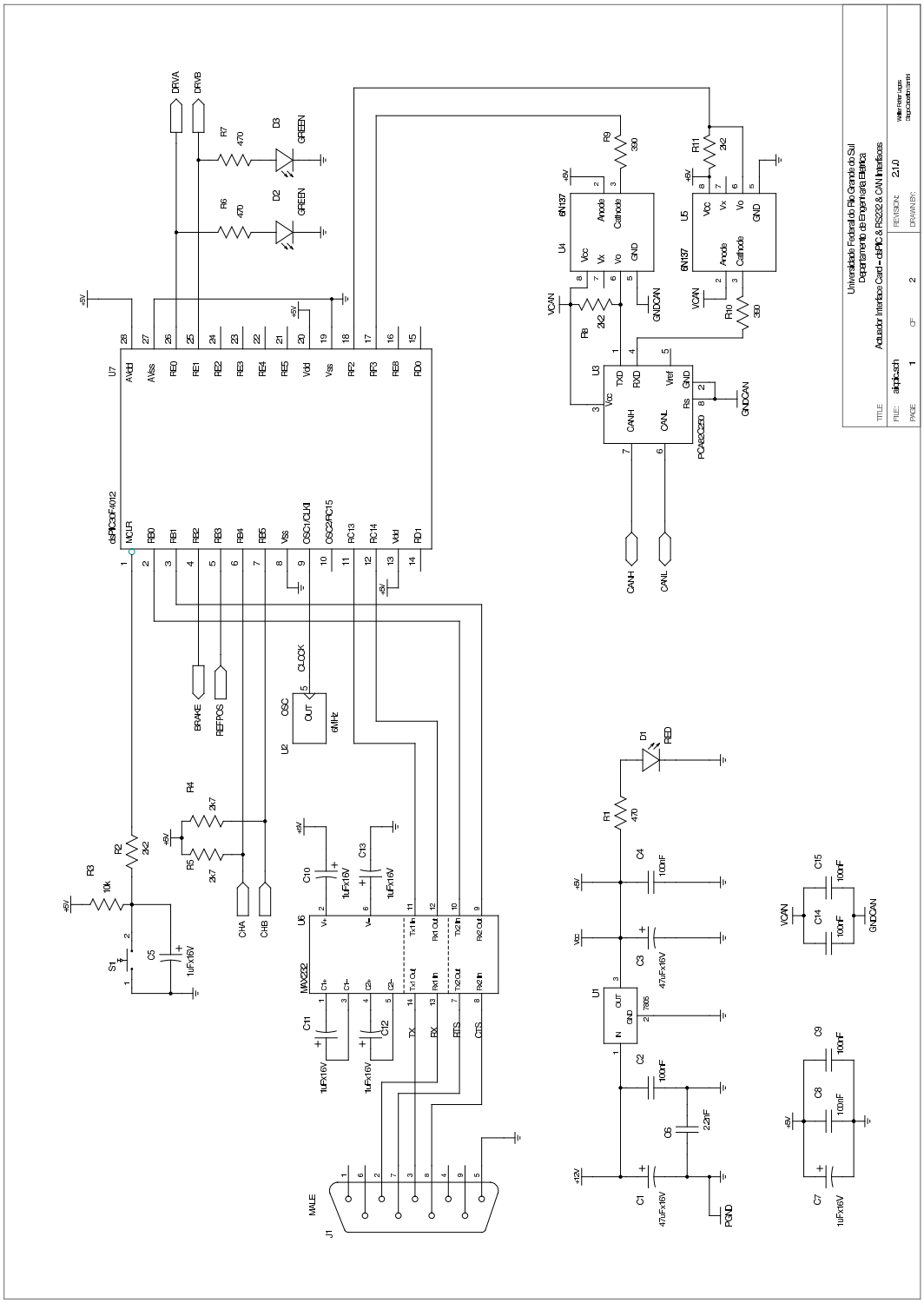
THE LINUX Kernel Archives. Disponível em: http://www.kernel.org/. Acesso em: 05 ago. 2009.

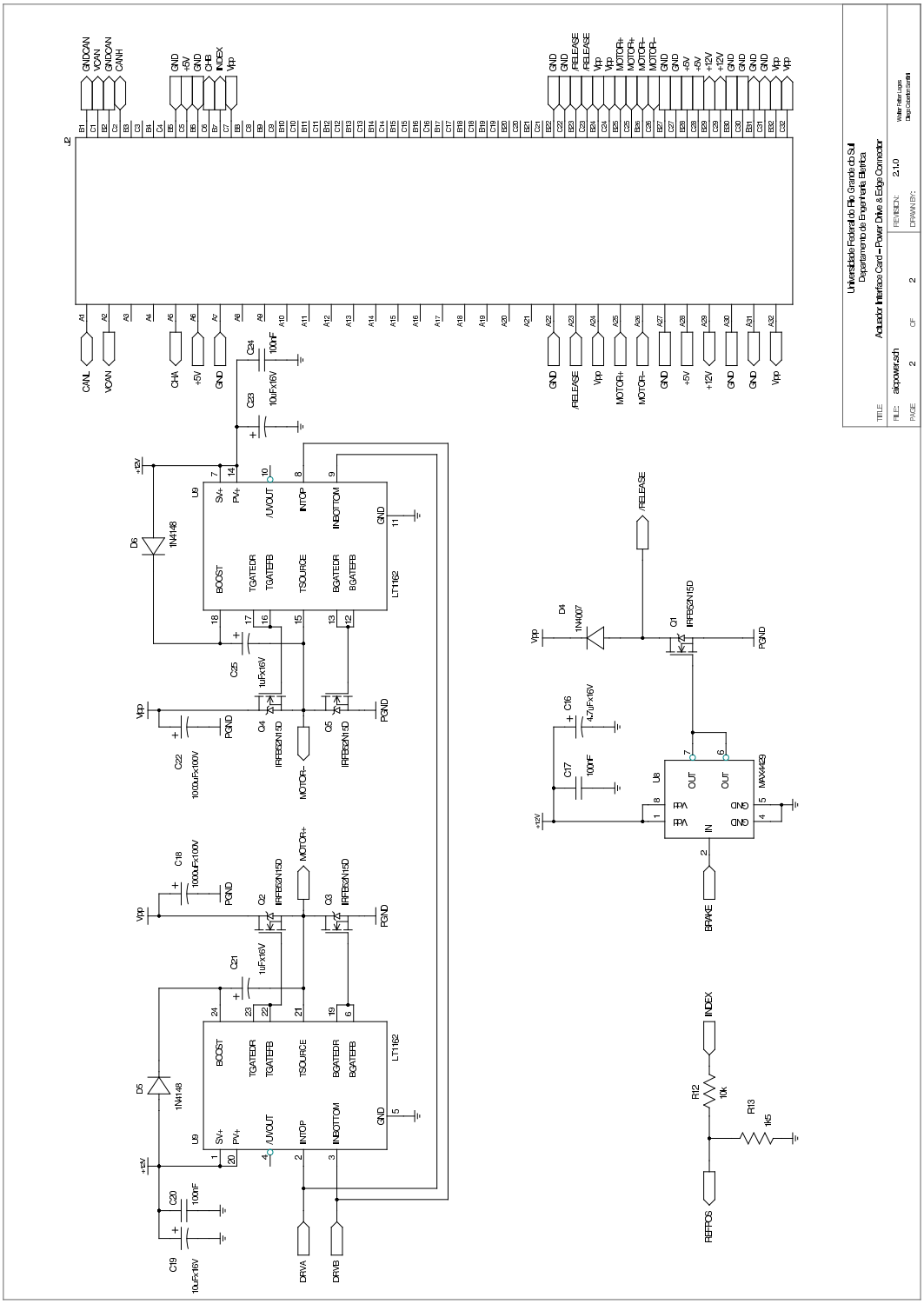
THE OROCOS Project Smarter Control in Robotics and Automation. Disponível em: http://www.orocos.org/. Acesso em: 05 ago. 2009.

WELCOME to Kvaser - Advanced CAN Solutions for Hardware, Software, Consulting and Education. Disponível em: http://www.kvaser.com/. Acesso em: 05 ago. 2009.

XUEMEI, L.; LIANGZHONG, J. Study on control system architecture of modular robot. In: IEEE INTERNATIONAL CONFERENCE ON ROBOTICS AND BIOMIMETICS, 2007, Sanya. **Proceedings...** Piscataway: IEEE Press, 2007. p.508–512.

APÊNDICE A DIAGRAMAS ESQUEMÁTICOS DA AIC


Universidade Federal do Rio Grande do Sul
Departamento de Engenharia Elétrica
Atuador Interface Can - dsPIC & RS232 & CAN Interfaces
TITLE:
FILE: atuador.sch
PAGE 1 OF 2
REVISION: 2.1.0
DRAWN BY:
DRVA
DRVB
R7 470
D3 GREEN
R6 470
D2 GREEN
+5V
U7 dsPIC30F4012
MCLR
RB0
RB1
RB2
RB3
RB4
RB5
Vss
OSC1/CLKI
OSC2/PLCI5
RC13
RC14
Vdd
RD1
AVdd
AVss
RE0
RE1
RE2
RE3
RE4
RE5
Vdd
Vss
RF2
RF3
RF8
RD0
BRAKE
REFPOS
CLOCK
U2 OSC
OUT
6MHz
R4 2k7
R5 2k7
R2 2k2
R3 10k
S1
C5 1uF/x16V
CHA
CHB
U6 MAX232
C10 1uF/x16V
C13 1uF/x16V
C11 1uF/x16V
C12 1uF/x16V
J1 MALE
R9 390
U4 6N137
R8 2k2
U3 PCA82C250
CANH
CANL
R11 2k2
U5 6N137
R10 390
VCAN
GNDCAN
D1 RED
R1 470
C4 100nF
C3 47uF/x16V
U1 78l05
C2 100nF
C6 2.2nF
C1 47uF/x16V
PGND
C14 100nF
C15 100nF
C9 100nF
C8 100nF
C7 1uF/x16V


Universidade Federal do Rio Grande do Sul
Departamento de Engenharia Elétrica
Acionador Interface Card – Power Drive & Edge Connector
TITLE
FILE: acpowercsch
PAGE: 2 OF 2
REVISION: 2.1.0
DRAWN BY:
J2
U9
LT1162
U8
MAX4429
D4
1N4007
D5
1N4148
D6
1N4148
Q1
IRFBG30N5D
Q2
IRFBG30N5D
Q3
IRFBG30N5D
Q4
IRFBG30N5D
Q5
IRFBG30N5D
C16
4.7uFx16V
C17
100nF
C18
1000uFx100V
C19
10uFx16V
C20
100nF
C21
1uFx16V
C22
1000uFx100V
C23
10uFx16V
C24
100nF
C25
1uFx16V
R12
10k
R13
1k5
GNDCAN
VCAN
GNDCAN
CANH
GND
+5V
GND
CHB
INDEX
Vpp
GND
GND
/RELEASE
/RELEASE
Vpp
Vpp
MOTOR+
MOTOR+
MOTOR-
MOTOR-
GND
+5V
+5V
+12V
+12V
GND
GND
GND
Vpp
Vpp
CANL
VCAN
CHA
+5V
GND
GND
/RELEASE
Vpp
MOTOR+
MOTOR-
GND
+5V
+12V
GND
GND
Vpp
+12V
Vpp
PGND
MOTOR-
MOTOR+
PGND
DRVA
DRVB
/RELEASE
PGND
BRAKE
INDEX
REFPOS

APÊNDICE B CABEÇALHOS DAS BIBLIOTECAS DA AIC

```
/*************************************************************************
                      Actuator Interface Card
                      I/O Interface Library

        Copyright (C) 2005-2008 Walter Fetter Lages <w.fetter@ieee.org>
                      2008      Diego Caberlon Santini <diegos@ece.ufrgs.br>
    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

    You can also obtain a copy of the GNU General Public License
    at <http://www.gnu.org/licenses>.

 *************************************************************************
 2008.06.16     Rewrite by Diego Caberlon Santini for dsPIC
 */

#ifndef _AICIO_H
#define _AICIO_H

/** @file aicio.h
 *      I/O Interface Library
 *      @author Walter Fetter Lages <w.fetter@ieee.org>
 *      @author Diegos Caberlon Santini <diegos@ece.ufrgs.br>
 */

/** @defgroup aicio AIC I/O Interface
@{
 */

/** Reference frequency = 24MHz
 */
#define REF_FREQ 24e6

/** Default Switching Frequency = 20KHz
 */
#define SW_FREQ 20e3

/** H-bridge turn-off delay
 */
#define TURNOFF_DELAY 625e-9

/** PWM minimal count value for frequency
 */
#define MIN_COUNT_FREQ (unsigned int)100

/** PWM maximal count value for frequency
 */
#define MAX_COUNT_FREQ (unsigned int)32768

/** Initializes brake
 *      @param base Base address for on board devices (DEPRECATED)
 */
extern void brake_initialize(unsigned long base);

/** Applies the electromagnetic brake
 */
extern void brake_apply(void);

/** Releases the electromagnetic brake
 */
extern void brake_release(void);
```

```
/** Initializes Index
 *      @param base Base address for on board devices (DEPRECATED)
 *      @since AIC-1.4.2
 */
extern void index_initialize(unsigned long base);

/** Finalizes Index
 */
extern void index_finalize(void);

/** Reads the sync-switch
 *      @return the sync-switch state
 *      @since AIC-1.4.2
 */
extern unsigned long index_read(void);

/** Sets the frequency of PWM in cycles of 24MHz
 *      @param count number of cycles from 100 to 32768. Default
 *              value is 1200
 */
extern void pwm_set_count(unsigned int count);

/** Gets the frequency of PWM in cycles of 24MHz
 *      @return count number of cycles
 */
extern unsigned pwm_get_count(void);
/** Sets the frequency of PWM
 *      @param frequency frequency in Hz from 732.42Hz to 240KHz. Default
 *              value is 20KHz
 */
extern void pwm_set_freq(float frequency);

/** Returns the frequency of PWM
 *      @return PWM frequency in Hz
 */
extern float pwm_get_freq(void);

/** PWM initialization
 *      @param base Base address for on board devices (DEPRECATED)
 *      @param min_count MIN_COUNT (DEPRECATED)
 *      @param count frequency of PWM in cycles of 24MHz. Number of cycles
 *               from 10 to 65530.
 *      @since AIC-1.4.2
 */
extern void pwm_initialize(unsigned long base,unsigned int min_count,unsigned int count);

/** Finalizes the PWM
 *      @since AIC-1.4.2
 */
extern void pwm_finalize(void);

/** Sets the duty-cycle of PWM
 *      @param dutycycle to be used by PWM as a float from 0 to 1.0
 *      @return the count value programmed to the PWM timer
 */
extern  unsigned long pwm_set_duty_float(float dutycycle);

/** Turns the PWM on
 */
extern void pwm_on(void);

/** Turns the PWM off
 */
extern void pwm_off(void);

/** Initializes the motor driver
 *      @param base Base address for on board devices
 *      @param voltage Motor Voltage
 *      @param freq PWM frequency
 */

extern void motor_initialize(unsigned long base,float voltage,float freq);

/** Finalizes the motor driver
 */
extern void motor_finalize(void);

/** Turns the motor driver on
 */
extern void motor_on(void);

/** Turns the motor driver off
 */
extern void motor_off(void);

/** Sets the voltage to be applied by the motor driver
 *      @param voltage to be applied by the motor driver
 *      @return the duty-cycle of the associated PWM
 */
extern float motor_set(float voltage);

/** Clears the counter on the quadrature decoder chip
 *      @since AIC-1.5.0
 */
extern void encoder_clear(void);

/** Initializes Encoder, clears the counter on the quadrature decoder chip
 *      @param base Base address for on board devices
 *      @param pulses Number of pulses per revolution of the encoder, considering the
```

```
 *              quadrature decoder
 *      @since AIC-1.5.0
 */
extern void encoder_initialize(unsigned long base,long pulses);

/** Finalizes Encoder
 *      @since AIC-1.5.0
 */
extern void encoder_finalize(void);

/** Gets the count on the quadrature decoder chip
 *      @return the number of pulses (in 2-complement) since the last time
 *              the counter was cleared
 *      @since AIC-1.5.0
 */
extern long encoder_get_count(void);


/** Gets the count and clears the counter on the quadrature decoder chip
 *      @return the number of pulses (in 2-complement) since the last time
 *              the counter was cleared
 *      @since AIC-1.5.0
 */
extern long encoder_get_count_and_clear(void);


/** Reads the counter on the quadrature decoder chip and converts the motion
 * to radians
 *      @return the displacement in radians since the last time
 *              the encoder was cleared
 *      @since AIC-1.5.0
 */
extern float encoder_read(void);

/** Reads the counter on the quadrature decoder chip, converts the motion
 * to radians and clears the counter
 *      @return the displacement in radians since the last time
 *              the encoder was cleared
 *      @since AIC-1.5.0
 */
extern float encoder_read_and_clear(void);


/** Turns an AIC on
 *      @since AIC-1.5.0
 */
extern void aic_on(void);

/** Turns an AIC off
 *      @since AIC-1.5.0
 */
extern void aic_off(void);

/** Initializes an AIC
 *      @param base Base address for on board devices
 *      @param vm Motor Voltage
 *      @param freq PWM frequency
 *      @param np Pulses per encoder revolution
 *      @since AIC-1.5.0
 */
extern void aic_initialize(unsigned long base,float vm,float freq,long np);

/** Finalizes an AIC
 *      @since AIC-1.5.0
 */
extern void aic_finalize(void);

/**
@}
 */


#endif
```

```
/*****************************************************************************

                        Actuator Interface Card
                            Serial Transmit
        Copyright (C) 2008 Diego Caberlon Santini <diegos@ece.ufrgs.br>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

    You can also obtain a copy of the GNU General Public License
    at <http://www.gnu.org/licenses>.

*****************************************************************************
2008.06.13 -> Start by Diego Caberlon Santini, for now it is just transmission
 */
#ifndef SERIAL_H
#define SERIAL _H

/** @file serial.h
 *      Serial Function
 *      @author Diego Caberlon Santini <diegos@ece.ufrgs.br>
 */

/** @defgroup serial Serial Function
@{
 */

#include "buffer.h"

/** Max Buffer Size
 */
#define SERIAL_SIZE BUFFER_SIZE

/*****************************************************************************/
/** Finalizes serial
 */
extern void serial_end(void);
/*****************************************************************************/
/** Transmit data
 *      @param data Pointer to a string
 *      @param size sizeof string
 *      @return 0 OK and -1 to error
 */
extern int serial_write(char *data, int size);
/*****************************************************************************/
/** Transmit data
 *      @param data Pointer to a string
 *      @return 0 OK and -1 to error
 */
extern void prints(char *data);
/*****************************************************************************/
/** Initializes serial with 19200, 8n1, no flux control
 */
extern void serial_init(void);
/*****************************************************************************/
#endif
```

```
/*****************************************************************************

                        Actuator Interface Card
                              Can Lib
        Copyright (C) 2008 Diego Caberlon Santini <diegos@ece.ufrgs.br>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

    You can also obtain a copy of the GNU General Public License
    at <http://www.gnu.org/licenses>.

*****************************************************************************
2008.07.14 -> Start by Diego Caberlon Santini
*/
#ifndef CANLIB_H
#define CANLIB_H

/** @file canlib.h
 *      CanLib Functions
 *      @author Diego Caberlon Santini <diegos@ece.ufrgs.br>
 */

/** @defgroup Can Can Group
@{
 */

typedef int canHandle;
typedef int canStatus;

#define CANHANDLER 0x7777

#define BAUD_1M 0x01

#define canFILTER_SET_CODE_STD  3
#define canFILTER_SET_MASK_STD  4
#define canFILTER_SET_CODE_EXT  5
#define canFILTER_SET_MASK_EXT  6

#define canMSG_RTR              0x0001      // Message is a remote request
#define canMSG_STD              0x0002      // Message has a standard ID
#define canMSG_EXT              0x0004      // Message has an extended ID

/** Opens a CAN channel to AIC
 *      @param channelNumber number of channel (only 0)
 *      @param bus flags (DEPRECATED)
 *      @return Handler to CAN channel
 */
extern canHandle canOpenChannel(unsigned int channelNumber, unsigned int flags);

/** Sets CANBus parameters
 *      @param hnd handler of channel
 *      @param freq frequency of canbus
 *      @param tseg1 number of Time Quantum in tseg1 (DEPRECATED)
 *      @param tseg1 number of Time Quantum in tseg2 (DEPRECATED)
 *      @param sjw number of Time Quantum in Synchronized Jump Width (DEPRECATED)
 *      @param noSamp number of Samples (DEPRECATED)
 *      @param syncmode (DEPRECATED)
 *      @return 0 OK, -1 error
 */
extern canStatus canSetBusParams(const canHandle hnd, \
                                 long freq, unsigned int tseg1, unsigned int tseg2, \
                                 unsigned int sjw, unsigned int noSamp, unsigned int syncmode);

/** Turn on channel
 *      @param hnd handler of channel
 *      @param bus flags (DEPRECATED)
 *      @return 0 OK, -1 error
 */
extern canStatus canBusOn(const canHandle hnd);

/** Turn off channel
 *      @param hnd handler of channel
 *      @param bus flags (DEPRECATED)
 *      @return 0 OK, -1 error
 */
extern canStatus canBusOff(const canHandle hnd);

/** Sets Mask or Filter to CAN identifier
 *      @param hnd handler of channel
 *      @param envelope Value to set
 *      @param flag Mask or Filter identifier
 *      @return 0 OK, -1 error
 */
extern canStatus canAccept(const canHandle hnd, const long envelope, const unsigned int flag);

/** Read a CANBus message
 *      @param hnd handler of channel
```

```
 *      @param id Pointer to CAN ID
 *      @param msg Pointer to CAN data
 *      @param dlc Pointer to CAN data size
 *      @param flag Pointer to Type of CAN message
 *      @param time (DEPRECATED)
 *      @return 0 OK, -1 error
 */
extern canStatus canRead(const canHandle hnd, long *id, void *msg, unsigned int *dlc,
                  unsigned int *flag, unsigned long *time);

/** Write a CANBus message
 *      @param hnd handler of channel
 *      @param id Pointer to CAN ID
 *      @param msg Pointer to CAN data
 *      @param dlc Pointer to CAN data size
 *      @param flag Pointer to Type of CAN message
 *      @return 0 OK, -1 error
 */
extern canStatus canWrite(const canHandle hnd, long id, void *msg, unsigned int dlc, unsigned int flag);

#endif
```

```
/*****************************************************************************
                        Actuator Interface Card
                           WatchDog Interface
        Copyright (C) 2008 Diego Caberlon Santini <diegos@ece.ufrgs.br>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

    You can also obtain a copy of the GNU General Public License
    at <http://www.gnu.org/licenses>.

*****************************************************************************
2008.10.28 -> Start by Diego Caberlon Santini
*/
#ifndef WDT_H
#define WDT_H

/** @file wdt.h
 *      WatchDog Interface
 *      @author Diego Caberlon Santini <diegos@ece.ufrgs.br>
 */

/** @defgroup buffer WatchDog Group
@{
 */

/*****************************************************************************/
/** Initializes WatchDog
 */
extern void wdt_initialize(void);
/*****************************************************************************/

/*****************************************************************************/
/** Finalize WatchDog
 */
extern void wdt_finalize(void);
/*****************************************************************************/

/*****************************************************************************/
/** Ping WatchDog
 */
extern void wdt_ping(void);
/*****************************************************************************/

#endif
```

APÊNDICE C PROGRAMA PRINCIPAL DA AIC

```
/*****************************************************************************

                      Actuator Interface Card
                     FreeRTOS Daemon on AIC-2.0.0
        Copyright (C) 2008 Diego Caberlon Santini <diegos@ece.ufrgs.br>

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

    You can also obtain a copy of the GNU General Public License
    at <http://www.gnu.org/licenses>.

*****************************************************************************
2008.06.12 -> Start FreeRTOS test to AIC-2.0.0 by Diego Caberlon Santini from Demos apps.
*****************************************************************************
*/
#include <p30f4012.h>

/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "croutine.h"

#include "aicnet.h"
#include "canlib.h"
#include "wdt.h"
#include "aicio.h"

/*-----------------------------------------------------------*/

#define mainDAEMON_TAKS_STACK_SIZE                              ( configMINIMAL_STACK_SIZE)

#define mainDAEMON_TASK_PRIORITY                                ( tskIDLE_PRIORITY + 3 )

/* The execution period of the check task. */
#define mainDAEMON_TASK_PERIOD                          ( ( portTickType ) 1 )

#define AIC 0x00000003


/*-----------------------------------------------------------*/
_FOSC(EC_PLL16 & CSW_FSCM_OFF);
_FWDT(WDT_OFF & WDTPSA_64 & WDTPSB_4);
_FBORPOR(MCLR_EN & RST_IOPIN & PBOR_OFF & PWRT_64 & RST_PWMPIN & PWMxH_ACT_HI & PWMxL_ACT_HI);
/*-----------------------------------------------------------*/

static void vDaemonTask( void *pvParameters );
extern long pulses_count_last;
extern long pulses_count_debug;
/*-----------------------------------------------------------*/

void end(void){
        for(;;){}
}

/*
 * Create the demo tasks then start the scheduler.
 */
int main( void )
{


        aic_initialize(0x0, 12.0, 20e3,2048*4);
```

```
        /* Create the test tasks defined within this file. */
        xTaskCreate( vDaemonTask, ( signed portCHAR * ) "DAEMON", mainDAEMON_TAKS_STACK_SIZE, NULL, mainDAEMON_TASK_PRIORITY, NULL );


        /* Finally start the scheduler. */
        vTaskStartScheduler();

        /* Will only reach here if there is insufficient heap available to start
        the scheduler. */
        return 0;
}
/*-----------------------------------------------------------*/

static void vDaemonTask( void *pvParameters )
{
        /* Used to wake the task at the correct frequency. */
        portTickType xLastExecutionTime;
        canHandle canhnd=0;
        canStatus status=0;
        long id=0;
        long long msg=0;
        unsigned int dlc=0;
        unsigned int flag=0;
        long double volt;
        float encoder;
        long index;
        int i;

        canhnd=canOpenChannel(0,0);

        if(canhnd < 0){
                end();
        }

        status=canSetBusParams(canhnd,BAUD_1M,0,0,0,0,0);

        if(status < 0){
                end();
        }

        status=canAccept(canhnd, 0x0000001F, canFILTER_SET_MASK_STD);
        status=canAccept(canhnd, AIC, canFILTER_SET_CODE_STD);

        if(status < 0){
                end();
        }

        status=canBusOn(canhnd);

        if(status < 0){
                end();
        }

        wdt_initialize();

        for(;;){
                xLastExecutionTime = xTaskGetTickCount();
                status=canRead(canhnd, &id, (void*)&msg, &dlc, &flag, NULL);
                if(status >= 0){
                        switch(id >> 5){
                                case AIC_RESET:
                                        end();
                                        break;
                                case AIC_MOTOR_OFF:
                                        wdt_ping();
                                        motor_off();
                                        break;
                                case AIC_BRAKE_APPLY:
                                        wdt_ping();
                                        brake_apply();
                                        break;
                                case AIC_MOTOR_ACT:
                                        wdt_ping();
                                        for(i=0;i<8;i++){
                                                ((unsigned char *)(&volt))[7-i]=((unsigned char *)(&msg))[i];
                                        }
                                        motor_set((float)volt);
                                        break;
                                case AIC_STATUS:
                                        wdt_ping();
                                        if(flag && canMSG_RTR){
                                                encoder=encoder_read_and_clear();
                                                index=(long)index_read();
                                                for(i=0;i<4;i++)
                                                        ((unsigned char *)(&msg))[3-i]=((unsigned char *)(&encoder))[i];
                                                for(i=0;i<4;i++)
                                                        ((unsigned char *)(&msg))[7-i]=((unsigned char *)(&index))[i];
                                                canWrite(canhnd, id, (void*)&msg, 8, canMSG_STD);
                                        }
                                        break;
                                case AIC_BRAKE_RELEASE:
                                        wdt_ping();
                                        brake_release();
                                        break;
                                case AIC_MOTOR_ON:
                                        wdt_ping();
                                        motor_on();
                                        break;
                                default:
```

```
                                        end();
                                        break;
                                }
                        }
                }

                end();
        }
```

APÊNDICE D CABEÇALHO DO DRIVER RTAI PCICAN

```
/*
 * Copyright (C) 2008 Diego Caberlon Santini
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License as
 * published by the Free Software Foundation; either version 2 of the
 * License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
 */

#ifndef _RTAI_PCICAN_H
#define _RTAI_PCICAN_H

#define  FUN_EXT_RTAI_PCICAN  14

#define _PCICANOPEN                     0
#define _PCICANCLOSE                    1
#define _PCICANREAD                     2
#define _PCICANWRITE                    3
#define _PCICANIOCTL                    4
#define _PCICANDEBUG                    5

///< IOCTL_SETBUSPARAM set bit timing to CANBus using VCanBusParam
#define IOCTL_SETBUSPARAM               0

#define BAUD_1M        (-1)     //CAN 1Mbit

typedef struct {
        signed long freq;
        unsigned char sjw;
        unsigned char tseg1;
        unsigned char tseg2;
        unsigned char samp3;
} VCanBusParams;

///< IOCTL_BUSON turn on CANBus
#define IOCTL_BUSON                     1

///< IOCTL_BUSOFF turn off CANBus
#define IOCTL_BUSOFF                    2

///< IOCTL_PCICAN_STATUS return status of pcican
#define IOCTL_PCICAN_STATUS             3

typedef struct s_can_msg {
        unsigned long id;               ///< CAN ID
        unsigned char flags;            ///< CAN type
        unsigned char data [8];         ///< CAN data
        unsigned char length;           ///< size of data
} CAN_MSG;


#define canMSG_RTR              0x0001      // Message is a remote request
#define canMSG_STD              0x0002      // Message has a standard ID
#define canMSG_EXT              0x0004      // Message has an extended ID

#endif /* !_RTAI_SERIAL_H */

#ifdef __KERNEL__

#include <rtai.h>
RTAI_SYSCALL_MODE int rt_pcican_read(void *handler, CAN_MSG *m);
RTAI_SYSCALL_MODE int rt_pcican_close(void *handler);
RTAI_SYSCALL_MODE int rt_pcican_open(void **handler, int channelNr);
RTAI_SYSCALL_MODE int rt_pcican_write(void *handler, CAN_MSG *m);
RTAI_SYSCALL_MODE int rt_pcican_ioctl(void *handler, unsigned int cmd, void *buffer);
```

```
RTAI_SYSCALL_MODE int rt_pcican_debug(void *handler, int *m);

#else /* !__KERNEL__ */

#include <rtai_lxrt.h>

/** Open a CANBus
          *     @param handler Pointer to receive CANBus handler
          *     @param channelNR channel to open
          *     @return 0 OK and -1 to error
 */
RTAI_PROTO(int, rt_pcican_open, (void **handler, int channelNr))
{
        struct { void **handler; long channelNr; } arg = { handler, channelNr };
        return rtai_lxrt(FUN_EXT_RTAI_PCICAN, SIZARG, _PCICANOPEN, &arg).i[LOW];
}

/** Close a CANBus
 *      @param handler Pointer to CANBus handler
 *      @return 0 OK and -1 to error
 */
RTAI_PROTO(int, rt_pcican_close, (void *handler))
{
        struct { void *handler;} arg = { handler};
        return rtai_lxrt(FUN_EXT_RTAI_PCICAN, SIZARG, _PCICANCLOSE, &arg).i[LOW];
}

/** Read from CANBus
 *      @param handler Pointer to CANBus handler
 *      @param m CAN message
 *      @return 0 OK and -1 to error
 */
RTAI_PROTO(int, rt_pcican_read, (void *handler, CAN_MSG *m))
{
        struct { void *handler; void *m; } arg = { handler, m };
        return rtai_lxrt(FUN_EXT_RTAI_PCICAN, SIZARG, _PCICANREAD, &arg).i[LOW];
}

/** Write to CANBus
 *      @param handler Pointer to CANBus handler
 *      @param m CAN message
 *      @return 0 OK and -1 to error
 */
RTAI_PROTO(int, rt_pcican_write, (void *handler, CAN_MSG *m))
{
        struct { void *handler; void *m; } arg = { handler, m };
        return rtai_lxrt(FUN_EXT_RTAI_PCICAN, SIZARG, _PCICANWRITE, &arg).i[LOW];
}

/** Close a CANBus
 *      @param handler Pointer to CANBus handler
 *      @param cmd Command to CANBus Controller
 *      @param buffer Buffer to command
 *      @return 0 OK and -1 to error
 */
RTAI_PROTO(int, rt_pcican_ioctl, (void *handler, unsigned int cmd, void *buffer))
{
        struct { void *handler; unsigned long cmd; void *buffer; } arg = { handler, cmd, buffer };
        return rtai_lxrt(FUN_EXT_RTAI_PCICAN, SIZARG, _PCICANIOCTL, &arg).i[LOW];
}
#endif
```

APÊNDICE E ARQUIVO DE CONFIGURAÇÃO DO DEPLOYMENTCOMPONENT

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "cpf.dtd">
<properties>
        <simple name="Import" type="string"><value>.</value></simple>
        <struct name="aic01" type="OCL::AIC">
                <simple name="AutoConf" type="boolean"><value>1</value></simple>
                <simple name="AutoStart" type="boolean"><value>1</value></simple>
                <simple name="LoadProperties" type="string"><value>xml/aic01.xml</value></simple>
                <struct name="Ports" type="PropertyBag">
                        <simple name="IndexAIC" type="string"><value>index1</value></simple>
                        <simple name="DisplacementAIC" type="string"><value>displacement1</value></simple>
                        <simple name="VoltageAIC" type="string"><value>voltage1</value></simple>
                </struct>
        </struct>
        <struct name="Joint1" type="OCL::Joint">
                <simple name="AutoConf" type="boolean"><value>1</value></simple>
                <simple name="AutoStart" type="boolean"><value>1</value></simple>
                <simple name="LoadProperties" type="string"><value>xml/joint1.xml</value></simple>
                <struct name="Peers" type="PropertyBag">
                        <simple type="string"><value>aic01</value></simple>
                        <simple type="string"><value>Sampler</value></simple>
                        <simple type="string"><value>Actuator</value></simple>
                </struct>
                <struct name="Ports" type="PropertyBag">
                        <simple name="Position" type="string"><value>p1</value></simple>
                        <simple name="Displacement" type="string"><value>v1</value></simple>
                        <simple name="Voltage" type="string"><value>t1</value></simple>
                        <simple name="IndexAIC" type="string"><value>index1</value></simple>
                        <simple name="DisplacementAIC" type="string"><value>displacement1</value></simple>
                        <simple name="VoltageAIC" type="string"><value>voltage1</value></simple>
                </struct>
        </struct>
        <struct name="aic02" type="OCL::AIC">
                <simple name="AutoConf" type="boolean"><value>1</value></simple>
                <simple name="AutoStart" type="boolean"><value>1</value></simple>
                <simple name="LoadProperties" type="string"><value>xml/aic02.xml</value></simple>
                <struct name="Ports" type="PropertyBag">
                        <simple name="IndexAIC" type="string"><value>index2</value></simple>
                        <simple name="DisplacementAIC" type="string"><value>displacement2</value></simple>
                        <simple name="VoltageAIC" type="string"><value>voltage2</value></simple>
                </struct>
        </struct>
        <struct name="Joint2" type="OCL::Joint">
                <simple name="AutoConf" type="boolean"><value>1</value></simple>
                <simple name="AutoStart" type="boolean"><value>1</value></simple>
                <simple name="LoadProperties" type="string"><value>xml/joint2.xml</value></simple>
                <struct name="Peers" type="PropertyBag">
                        <simple type="string"><value>aic02</value></simple>
                        <simple type="string"><value>Sampler</value></simple>
                        <simple type="string"><value>Actuator</value></simple>
                </struct>
                <struct name="Ports" type="PropertyBag">
                        <simple name="Position" type="string"><value>p2</value></simple>
                        <simple name="Displacement" type="string"><value>v2</value></simple>
                        <simple name="Voltage" type="string"><value>t2</value></simple>
                        <simple name="IndexAIC" type="string"><value>index2</value></simple>
                        <simple name="DisplacementAIC" type="string"><value>displacement2</value></simple>
                        <simple name="VoltageAIC" type="string"><value>voltage2</value></simple>
                </struct>
        </struct>
        <struct name="Sampler" type="OCL::Sampler">
                <struct name="Activity" type="PeriodicActivity">
                        <simple name="Period" type="double"><value>0.01</value></simple>
                        <simple name="Priority" type="short"><value>1</value></simple>
                        <simple name="Scheduler" type="string"><value>ORO_SCHED_RT</value></simple>
                </struct>
                <simple name="AutoConf" type="boolean"><value>1</value></simple>
                <simple name="AutoStart" type="boolean"><value>1</value></simple>
        </struct>
        <struct name="Sensor" type="OCL::Sensor2Mux">
                <struct name="Activity" type="NonPeriodicActivity">
                        <simple name="Priority" type="short"><value>2</value></simple>
                        <simple name="Scheduler" type="string"><value>ORO_SCHED_RT</value></simple>
                </struct>
```

```
        <simple name="AutoConf" type="boolean"><value>1</value></simple>
        <simple name="AutoStart" type="boolean"><value>1</value></simple>
        <struct name="Peers" type="PropertyBag">
                <simple type="string"><value>Sampler</value></simple>
        </struct>
        <struct name="Ports" type="PropertyBag">
                <simple name="Position1" type="string"><value>p1</value></simple>
                <simple name="Displacement1" type="string"><value>v1</value></simple>
                <simple name="Position2" type="string"><value>p2</value></simple>
                <simple name="Displacement2" type="string"><value>v2</value></simple>
                <simple name="SensorPort" type="string"><value>sensor</value></simple>
        </struct>
</struct>
<struct name="Actuator" type="OCL::Actuator2Demux">
        <struct name="Activity" type="NonPeriodicActivity">
                <simple name="Priority" type="short"><value>3</value></simple>
                <simple name="Scheduler" type="string"><value>ORO_SCHED_RT</value></simple>
        </struct>
        <simple name="AutoConf" type="boolean"><value>1</value></simple>
        <simple name="AutoStart" type="boolean"><value>1</value></simple>
        <simple name="LoadProperties" type="string"><value>xml/actuator.xml</value></simple>
        <struct name="Peers" type="PropertyBag">
                <simple type="string"><value>Sampler</value></simple>
        </struct>
        <struct name="Ports" type="PropertyBag">
                <simple name="ActPort" type="string"><value>act</value></simple>
                <simple name="Voltage1" type="string"><value>t1</value></simple>
                <simple name="Voltage2" type="string"><value>t2</value></simple>
        </struct>
</struct>
<struct name="Controller" type="OCL::Controller2PIDCT">
        <struct name="Activity" type="NonPeriodicActivity">
                <simple name="Priority" type="short"><value>3</value></simple>
                <simple name="Scheduler" type="string"><value>ORO_SCHED_RT</value></simple>
        </struct>
        <simple name="AutoConf" type="boolean"><value>1</value></simple>
        <simple name="AutoStart" type="boolean"><value>1</value></simple>
        <struct name="Peers" type="PropertyBag">
                <simple type="string"><value>Sampler</value></simple>
        </struct>
        <struct name="Ports" type="PropertyBag">
                <simple name="SensorPort" type="string"><value>sensor</value></simple>
                <simple name="RefPort" type="string"><value>ref_vector</value></simple>
                <simple name="OutPort" type="string"><value>act</value></simple>
                <simple name="Ref1" type="string"><value>ref1</value></simple>
                <simple name="Sen1" type="string"><value>sen1</value></simple>
                <simple name="Out1" type="string"><value>out1</value></simple>
                <simple name="Ref2" type="string"><value>ref2</value></simple>
                <simple name="Sen2" type="string"><value>sen2</value></simple>
                <simple name="Out2" type="string"><value>out2</value></simple>
                <simple name="InModel" type="string"><value>inm</value></simple>
                <simple name="OutModel" type="string"><value>outm</value></simple>
        </struct>
</struct>
<struct name="pid1" type="OCL::PID">
        <simple name="AutoConf" type="boolean"><value>1</value></simple>
        <simple name="AutoStart" type="boolean"><value>1</value></simple>
        <simple name="LoadProperties" type="string"><value>xml/pid1.xml</value></simple>
        <struct name="Ports" type="PropertyBag">
                <simple name="Ref" type="string"><value>ref1</value></simple>
                <simple name="Sen" type="string"><value>sen1</value></simple>
                <simple name="Out" type="string"><value>out1</value></simple>
        </struct>
</struct>
<struct name="pid2" type="OCL::PID">
        <simple name="AutoConf" type="boolean"><value>1</value></simple>
        <simple name="AutoStart" type="boolean"><value>1</value></simple>
        <simple name="LoadProperties" type="string"><value>xml/pid2.xml</value></simple>
        <struct name="Ports" type="PropertyBag">
                <simple name="Ref" type="string"><value>ref2</value></simple>
                <simple name="Sen" type="string"><value>sen2</value></simple>
                <simple name="Out" type="string"><value>out2</value></simple>
        </struct>
</struct>
<struct name="model" type="OCL::ModelJH">
        <simple name="AutoConf" type="boolean"><value>1</value></simple>
        <simple name="AutoStart" type="boolean"><value>1</value></simple>
        <struct name="Ports" type="PropertyBag">
                <simple name="In" type="string"><value>inm</value></simple>
                <simple name="Out" type="string"><value>outm</value></simple>
        </struct>
</struct>
<struct name="Bridge" type="OCL::BridgeRef">
        <struct name="Activity" type="NonPeriodicActivity">
                <simple name="Priority" type="short"><value>97</value></simple>
                <simple name="Scheduler" type="string"><value>ORO_SCHED_RT</value></simple>
        </struct>
        <simple name="AutoConf" type="boolean"><value>1</value></simple>
        <simple name="AutoStart" type="boolean"><value>1</value></simple>
        <simple name="LoadProperties" type="string"><value>xml/bridge.xml</value></simple>
        <struct name="Ports" type="PropertyBag">
                <simple name="SensorPort" type="string"><value>sensor</value></simple>
                <simple name="nAxesDP" type="string"><value>naxesdp</value></simple>
                <simple name="nAxesDV" type="string"><value>naxesdv</value></simple>
                <simple name="nAxesDA" type="string"><value>naxesda</value></simple>
                <simple name="nAxesP" type="string"><value>mp_vector</value></simple>
                <simple name="RefPort" type="string"><value>ref_vector</value></simple>
        </struct>
</struct>
```

```
        <struct name="Generator" type="OCL::Generator">
                <struct name="Activity" type="PeriodicActivity">
                        <simple name="Period" type="double"><value>0.01</value></simple>
                        <simple name="Priority" type="short"><value>98</value></simple>
                        <simple name="Scheduler" type="string"><value>ORO_SCHED_OTHER</value></simple>
                </struct>
                <simple name="AutoConf" type="boolean"><value>1</value></simple>
                <simple name="AutoStart" type="boolean"><value>1</value></simple>
                <simple name="LoadProperties" type="string"><value>xml/generator.xml</value></simple>
                <struct name="Ports" type="PropertyBag">
                        <simple name="nAxesP" type="string"><value>mp_vector</value></simple>
                        <simple name="nAxesDP" type="string"><value>naxesdp</value></simple>
                        <simple name="nAxesDV" type="string"><value>naxesdv</value></simple>
                        <simple name="nAxesDA" type="string"><value>naxesda</value></simple>
                </struct>
        </struct>
        <simple name="Import" type="string"><value>/home/diegos/new/OROCOS/build/lib</value></simple>
        <struct name="Reporting" type="OCL::FileReporting">
                <struct name="Activity" type="PeriodicActivity">
                        <simple name="Period" type="double"><value>0.01</value></simple>
                        <simple name="Priority" type="short"><value>99</value></simple>
                        <simple name="Scheduler" type="string"><value>ORO_SCHED_OTHER</value></simple>
                        </struct>
                <simple name="AutoConf" type="boolean"><value>1</value></simple>
                <simple name="AutoStart" type="boolean"><value>1</value></simple>
                <simple name="LoadProperties" type="string"><value>xml/Reporting.xml</value></simple>
                <struct name="Peers" type="PropertyBag">
                        <simple type="string"><value>Actuator</value></simple>
                        <simple type="string"><value>Sensor</value></simple>
                        <simple type="string"><value>Sampler</value></simple>
                        <simple type="string"><value>Joint1</value></simple>
                        <simple type="string"><value>Joint2</value></simple>
                        <simple type="string"><value>Bridge</value></simple>
                        <simple type="string"><value>Controller</value></simple>
                        <simple type="string"><value>model</value></simple>
                        </struct>
                </struct>
</properties>
```