UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA MECÂNICA

Descrição, Instalação, Programação e Funcionamento de um Robô Manipulador do tipo SCARA

Dissertação submetida à Universidade Federal de Santa Catarina para obtenção do grau de

Mestre em Engenharia Mecânica

Lucas Weihmann

Florianópolis, Outubro de 1999

Descrição, Instalação, Programação e Funcionamento de um Robô Manipulador do tipo SCARA

Eng. Lucas Weihmann

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Mecânica na área de concentração Projeto de Sistemas Mecânicos – Robótica, e aprovada em sua forma final pelo curso de Pós-Graduação de Engenharia Mecânica.

Prof. Raul Guenther, D. Sc. – Orientador

Prof. Júlio César Passos, Dr. – Coordenador do Curso

Banca Examinadora:

Prof. Altamir Dias, D. Sc

Prof. Edson Roberto De Pieri, Dr.

Prof. Marcelo Ricardo Stemmer, Dr.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O objetivo principal deste trabalho é fornecer o máximo de informações possíveis sobre o robô SCARA, instalado no início do ano de 1998 no Laboratório de Robótica da Universidade Federal de Santa Catarina, de forma a servir como referência para trabalhos futuros que venham a ser desenvolvidos nesta área. São apresentadas as principais características de cada um dos componentes físicos que compõem o manipulador, descritas as interações existentes entre cada um deles e as propriedades do robô visto como um todo. É apresentada uma descrição detalhada dos programas para movimentação e controle deste robô, e dos programas que garantem segurança na sua utilização. São mostradas as diretrizes para utilização do controle híbrido força/posição e para instalação de novos controladores.

Palavras-chave: SCARA, controle híbrido, robô Inter.

ABSTRACT

The aim of this work is to supply the maximum information about the SCARA robot, installed in the beginning of 1998 in the Laboratory of Robotics from the Federal University of Santa Catarina, in a way to serve as reference for further works that will be developed in this area. The main characteristics of each physical component that compose the manipulator, the interaction between each one of this components and the properties of the robot like a whole are presented. A detailed description of the programs for movement and control and also the programs that guarantee the safety use of the robot are present. The guidelines from the hybrid force/position control installed in the robot and the way to install new controllers are shown.

Key-words: SCARA, hybrid control, Inter robot.

Capítulo 1

INTRODUÇÃO

Na indústria atual é crescente o emprego de robôs nos mais diversos processos de fabricação. O que se observa até o presente momento é que estes robôs utilizam técnicas de controle bastante elementares pelo simples motivo que estas técnicas funcionam adequadamente para o tipo de tarefa para o qual estão sendo empregados.

A sofisticação das tarefas a serem executadas por robôs também vem crescendo e isto gera a necessidade de utilização de técnicas de controle mais complexas.

Algumas destas técnicas mais complexas, como o controle de força e o controle adaptativo, vem sendo tema de diversos trabalhos de dissertação e doutorado desenvolvidos aqui na UFSC e em diversos outros centros de pesquisa do Brasil e do mundo.

No entanto, aqui na UFSC, a maioria dos resultados destes trabalhos vem sendo obtida a partir de simulações, devido à inexistência de um robô disponível para a realização de testes práticos.

Com o intuito de romper esta barreira existente entre prática e teoria, foi adquirido pela UFSC, junto ao Instituto de Robótica da Universidade Federal de Zurique, um robô manipulador com configuração SCARA. Sua principal característica é possuir uma arquitetura aberta, ou seja, a possibilidade de programação e implementação de algoritmos de controle e não somente a programação de trajetórias como na maioria dos robôs industriais.

A aquisição deste robô abre um campo amplo para pesquisas principalmente na área de controle de força. O sensor de força, descrito no capítulo 2, foi instalado durante a fase de conclusão desta dissertação e, portanto, ainda não foi possível a realização de testes práticos. Mas sua utilização permitirá a realização de vários

experimentos nesta área que está sendo bastante pesquisada e cujas aplicações devem ser utilizadas em robôs industrias brevemente.

Este robô, mostrado na figura 1.1, recebeu o nome Inter e foi instalado em março de 1998 na sala do Laboratório de Automação Industrial (LAI) pertencente ao Laboratório de Controle em Micro Informática (LCMI).

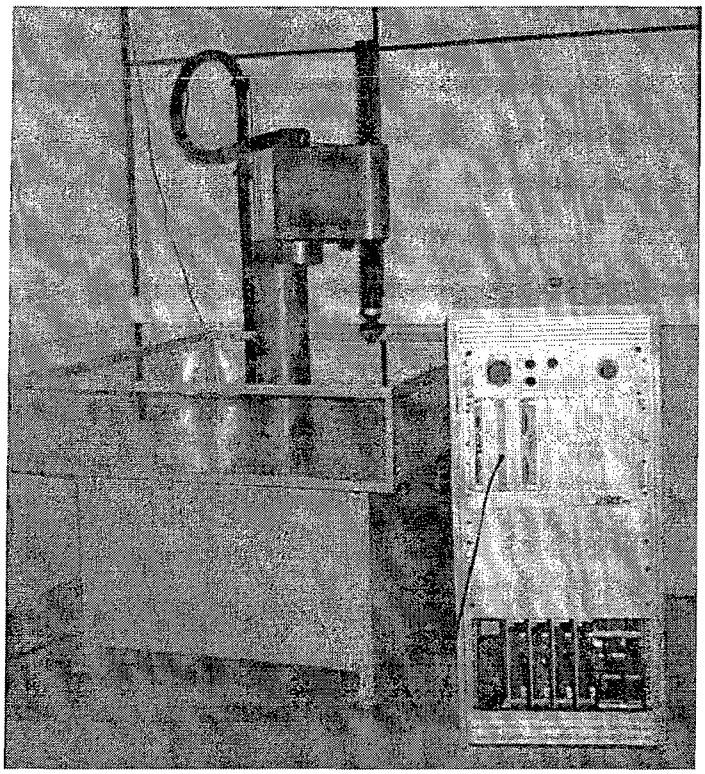


FIGURA 1.1 - O robô manipulador Inter e sua unidade de controle.

Até o presente momento foram produzidas apenas quatro unidades deste robô. Além desta unidade, adquirida pela UFSC, outras duas unidades estão sendo utilizadas numa indústria suíça na montagem de cabos coaxiais, em um projeto realizado em conjunto com a Universidade Federal de Zurique. E a última unidade permanece no Instituto de Robótica em Zurique para fins de pesquisa.

Devido à pequena quantidade de robôs produzida e devido ao seu caráter amplamente voltado para pesquisa, infelizmente não foi escrito nenhum manual completo sobre seu funcionamento. Existe uma documentação sobre os componentes do robô tratados individualmente e sobre os principais circuitos elétricos existentes. Mas não existe a descrição do robô como um conjunto, abordando as interações entre

seus vários componentes. Também não existe um manual que explique os programas utilizados na movimentação e controle deste robô.

Durante o período do curso de mestrado, acompanhei todo o processo de montagem do robô, tanto da parte mecânica quanto da parte elétrica. Participei também ativamente no processo de implementação dos programas do robô, tendo oportunidade de conhecer profundamente as principais características, problemas e capacidades do equipamento adquirido. Antes de ser enviado ao Brasil, o robô foi totalmente desmontado e corretamente acondicionado para evitar danos no transporte. Após a chegada do robô no Brasil, executei, com auxílio do pessoal do LCMI, a montagem e a colocação do robô em funcionamento.

O objetivo deste trabalho é justamente organizar todas estas informações adquiridas durante o processo de confecção do robô, suprindo a carência de manuais e dados explicativos. Sem dúvida o volume de informações a respeito de um robô que possui arquitetura aberta é muito maior que o de um robô comercial. E é praticamente impossível abordar todos os aspectos pois, num robô com estes recursos, muitas vezes os limites passam a ser a criatividade e a imaginação do programador. O que se procura neste trabalho é apresentar as informações e programas existentes e tentar mostrar caminhos para o desenvolvimento de trabalhos futuros.

No capítulo 2 serão apresentadas as principais características físicas do robô, descrevendo seus componentes principais e aspectos construtivos.

No capítulo 3 serão descritos os módulos fundamentais para o funcionamento do robô.

No capítulo 4 será feita uma revisão sobre o controle híbrido força/posição e serão apresentados os módulos de controle híbrido implementados no robô.

No capítulo 5 são colocadas as conclusões a respeito do trabalho desenvolvido e as perspectivas de futuros trabalhos nesta área.

Para a completa compreensão deste trabalho e para a utilização eficiente do robô, é necessário que se tenha um bom conhecimento da linguagem XOberon. Como

também não temos um manual explicando esta linguagem, seus principais aspectos e alguns exemplos que auxiliem sua compreensão foram descritos no apêndice A.

No apêndice B serão descritos todos os procedimentos necessários para garantir o funcionamento adequado do robô.

Os principais algoritmos e rotinas são apresentados no apêndice C.

Capítulo 2

DESCRIÇÃO GERAL DO ROBÔ

2.1 Introdução

O robô manipulador Inter foi construído, sob encomenda da UFSC, no Instituto de Robótica da Universidade Federal de Zurique (ETH), na Suíça. Trata-se de um robô do tipo SCARA para fins de pesquisa. A interface entre usuário e robô é feito através de um ambiente computacional chamado XOberon.

Este capítulo apresenta uma descrição do robô abrangendo seus componentes principais, sua construção mecânica, seus componentes periféricos, seus parâmetros dinâmicos e seu funcionamento. A intenção é proporcionar ao usuário as informações necessárias para se compreender o funcionamento do robô manipulador Inter. Incluímos aqui explicações técnicas bem como as principais informações contidas em catálogos.

2.2 Descrição Geral do Robô SCARA

O robô do tipo SCARA é uma versão especial de robô articulado. SCARA é sigla de *Selective Compliant Articulated Robot for Assembly*. Nesta configuração as duas primeiras juntas são de rotação, girando em torno de eixos verticais, trabalhando, portanto, num plano horizontal. A terceira junta é de translação, realizando deslocamentos no sentido vertical. O efetuador final realiza movimentos rotacionais sobre um eixo vertical, podendo ser adequado de acordo com o tipo de tarefa a ser executada. Assim, vemos que o robô SCARA possui quatro graus de liberdade. A representação esquemática de um robô do tipo SCARA está na figura 2.1.

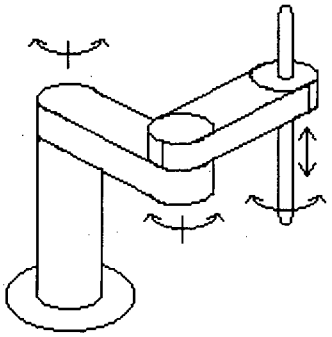


FIGURA 2.1 - Representação esquemática de um robô SCARA.

2.3 Dimensões e Sistemas de Coordenadas

A figura 2.2 apresenta um desenho esquemático do robô Inter, com suas principais dimensões, fornecidas em mm, e a alocação de seus sistemas de coordenadas.

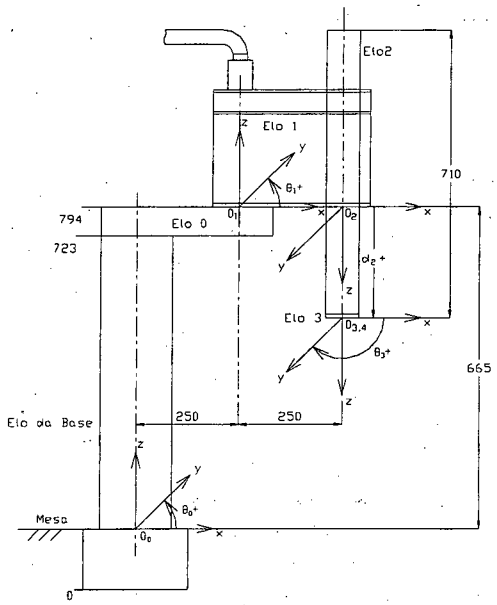


FIGURA 2.2 - Sistema de coordenadas do robô Inter.

Abaixo segue uma breve descrição de cada sistema de coordenadas:

$O_0 \rightarrow$ sistema de coordenadas cartesianas fixo à base do robô, utilizado como referência para definir a posição e a orientação do manipulador no espaço. Este espaço que tem como centro de seu sistema de coordenadas o ponto O_0 é chamado de espaço do manipulador ou espaço operacional.

$O_1 \rightarrow$ sistema de coordenadas fixo ao elo 0. Sua posição relativa ao sistema O_0 é alterada através de um giro θ_0 imposto pela junta 0 ao elo 0.

$O_2 \rightarrow$ sistema de coordenadas fixo ao elo 1. Sua posição relativa ao sistema O_1 é alterada através de um giro θ_1 imposto pela junta 1 ao elo 1.

$O_3 \rightarrow$ sistema de coordenadas fixo ao elo 2. Sua posição relativa ao sistema O_2 é alterada através de um deslocamento d_2 imposto pela junta 2 ao elo 2.

$O_4 \rightarrow$ sistema de coordenadas fixo ao elo 3. Sua posição relativa ao sistema O_3 é alterada através de um giro θ_3 imposto pela junta 3 ao elo 3. É o sistema de coordenadas do efetuador final.

É importante notar que não utilizamos a convenção para juntas e elos normalmente encontrada na literatura. A convenção aqui adotada visa concordar com a do ambiente XOberon. Isto se justifica pois na representação de um vetor nesta linguagem o primeiro termo é referenciado com o índice 0, o segundo com o índice 1 e assim por diante. Desta forma referenciamos a primeira junta e o primeiro elo móvel do manipulador com o índice zero para que haja concordância com a representação vetorial do XOberon.

A partir da definição destes sistemas de coordenadas temos a matriz com os parâmetros de Denavit-Hartenberg, apresentada na tabela 2.1.

Tabela 2.1 - Parâmetros de Denavit-Hartenberg

No. Elo	α_i	a_i	d_i	θ_i
0	0	250	665	θ_0
1	180	250	0	θ_1
2	0	0	d_2	0
3	0	0	0	θ_3

E a matriz de transformação T_4^0 do sistema de coordenadas O_4 para O_0 é:

$$T_4^0 = \begin{bmatrix} C3C01+S3S01 & -S3C01+C3S01 & 0 & 0{,}25(C01+C0) \\ C3S01-S3C01 & -S3S01-C3C01 & 0 & 0{,}25(S01+S0) \\ 0 & 0 & -1 & -d2+0{,}665 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

onde a representação dos senos e cossenos foi simplificada na forma:

$C3 = \cos(\theta_3)$

$C01 = \cos(\theta_0 + \theta_1)$

O sistema de coordenadas do manipulador final não é definitivo pois na direção z ainda podem ocorrer alterações devido à implementação de uma garra, do sensor de força ou de algum outro efetuador final.

2.4 Descrição Geral de Elos e Juntas

São apresentados a seguir os dados relativos a cada um dos componentes do conjunto elo/junta. A maioria destes dados foi obtida de catálogos. Alguns, no entanto, foram determinados a partir de experimentos ou estimados através de cálculos feitos a partir de informações nem sempre muito precisas. Portanto, principalmente os dados sobre momento de inércia e centro de massa podem apresentar incertezas consideráveis. Alguns dados permanecem em aberto pois não foi possível determiná-los devido à falta de informações.

Como estamos montando um conjunto a partir de vários componentes, é natural que algumas características como torque máximo e velocidade máxima de rotação não sejam os mesmos para cada um dos componentes. Naturalmente deve ser adotado para o conjunto a condição máxima suportada pelo seu elemento mais frágil. O cálculo dos valores efetivamente empregados para cada conjunto elo/junta é apresentado em 2.6.

2.4.1 Elo 0, Junta 0

Elo:
Massa: ~ 6,3 kg
Comprimento entre juntas: 250 mm

Centro de Massa: 118 mm
Momento de Inércia (C.M.):

Tipo do Motor: BLS 072
Vel. Máx.: 9000 rpm → 50 Hz → 942 rad/s
Torque Parado: 1,9 Nm
Torque de Pico: 11,1 Nm
Constante de Torque: 0,5 Nm/A
Massa: 1,9 kg
Momento de Inércia: $0,05x10^{-3}$ kg.m^2

Harmonic Drive: HFUC-32 100-20H SP2565
Torque de Saída: 137 Nm (para 2000 rpm)
Torque Máximo (de Pico): 333 Nm
Torque de Emergência: 647 Nm
Velocidade Máxima de Entrada: 4000 rpm → 67 Hz → 419 rad/s
Saída: 40 rpm → 0,67 Hz → 4,19 rad/s
Relação de Transmissão: n = -100
Momento de Inércia (lado motor): $1,69x10^{-4}$ kg.m^2
Massa: 3,2 kg
Rendimento: ~ 70%
Período na Velocidade Máxima: ~ 1,5 s

Amplificador: 220/12
Corrente de Trabalho: 2,74 A
Corrente Máxima: 6,6 A

2.4.2 Elo 1, Junta 1

Elo:
Massa: ~ 19,5 kg (todo o conjunto)
Comprimento entre juntas: 250 mm
Centro de Massa: 116 mm (todo o conjunto)
Momento de Inércia (C.M.):

Tipo do Motor: BLS 055
Vel. Máx.: 10.000 rpm → 167 Hz → 1043 rad/s
Torque Parado: 0,7 Nm
Torque de Pico: 4,65 Nm
Constante de Torque: 0,47 Nm/A
Massa: 1,4 kg
Momento de Inércia: $0,015x10^{-3}$ Nm

Harmonic Drive: HFUC-25 100-20H SP2878
Torque de Saída $_{(contínuo)}$: 67 Nm (para 2000 rpm)
Torque Máximo $_{(de\ pico)}$: 157 Nm
Torque de Emergência: 284 Nm

Velocidade Máxima de Entrada: 4000 rpm → 67 Hz → 419 rad/s
Saída: 40 rpm → 0,67 Hz → 4,19 rad/s

Velocidade Contínua Entrada: 3500 rpm → 58 Hz → 557 rad/s
Relação de Transmissão: n = 100
Momento de Inércia (lado motor): $0{,}413 \times 10^{-4}$ kg.m^2
Massa: 1,5 kg
Rendimento: ~ 70%
Período na Velocidade Máxima: ~ 1,5 s

Amplificador: 220/04
Corrente de Trabalho: 1,42 A
Corrente Máxima: 3,34 A

2.4.3 Elo 2, Junta 2

Tipo do Motor 2: idem ao motor 1

Redução:
Relação: 32/54
Momento de Inércia:
Rendimento:
Massa:

Spline:
Relação: 1 volta → 25mm
Momento de Inércia:
Massa:

Amplificador: 220/04
Corrente de Trabalho: 1,49 A
Corrente Máxima: 4,4 A

2.4.4 Elo 3, Junta 3

Tipo do Motor: idem ao motor 1

Redução A:
Relação: 1/4,5
Momento de Inércia:
Rendimento:
Massa:

Redução B:
Relação: 15/54
Momento de Inércia:

Rendimento:
Massa:

Spline:
Relação: 1/1
Momento de Inércia
Massa:

Amplificador: 220/04
Corrente de Trabalho: 1,49 A
Corrente Máxima: 4,4 A

2.5 Aspectos Construtivos

Os principais componentes do robô comprados prontos de outros fabricantes são os motores, harmonic drives (HD) e o ball-screw-spline. Os motores das quatro juntas são de corrente alternada. Nas juntas 0 e 1 é utilizada uma redução com HD, cuja saída movimenta os elos. Nas juntas 2 e 3 a redução é feita através de polias dentadas que movimentam o ball-screw-spline. Este componente por sua vez irá provocar o movimento dos elos 2 e 3. Veremos a seguir com maiores detalhes cada um destes componentes.

2.5.1 Harmonic Drive

O Harmonic Drive (HD) é um dispositivo mecânico utilizado para transmissão e transformação de torques e rotações. No robô são utilizados dois HDs: um acoplado ao motor 0 (para rotação do elo 0) e outro ao motor 1 (para rotação do elo 1).

O HD é composto de três peças principais, como mostrado na figura 2.3.

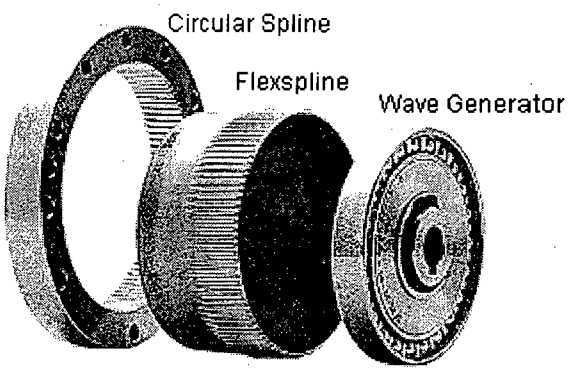


FIGURA 2.3 - Componentes do Harmonic Drive.

O Flexspline (FS) e o Wave Generator (WG) giram no interior do anel circular rígido Circular Spline (CS). O funcionamento do dispositivo está centrado no fato de que WG é elíptico. Para que possa haver rotação entre CS e WG, é necessário que FS seja flexível. A figura 2.4 ilustra uma rotação.

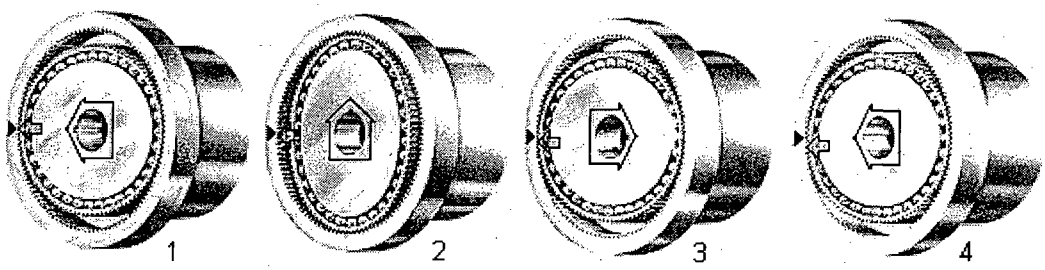
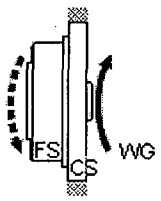


FIGURA 2.4 - Funcionamento do Harmonic Drive .

Note que em (1) foi feita uma marcação "alinhando" CS, FS e WG e que durante uma rotação qualquer há somente dois pontos de contato entre FS e CS. Após uma rotação completa de WG, existe uma diferença de dois dentes entre as engrenagens de FS e CS, conforme mostrado em (4). A relação de transmissão é definida pela quantidade de dentes de FS e CS. Para os HDs usados no robô, 100 rotações de WG equivalem a uma rotação de FS ou CS.

Há diferentes possibilidades de montagem e fixação do HD em relação a motor e elo. No caso do robô Inter foi utilizada o esquema de montagem mostrado na figura 2.5.



CS: fixo no elo anterior, que permanece parado.
WG: movimentado pelo motor, cuja carcaça está fixa no elo anterior.
FS: fixo no elo a ser movimentado.

FIGURA 2.5 - Esquema de montagem do Harmonic Drive.

Com esta montagem, vemos que o elo gira no sentido contrário do eixo do motor e como conseqüência a relação de transmissão vale -100.

2.5.2 Ball-Screw-Spline

O Ball-Screw-Spline, que chamaremos simplesmente de fuso, é o elemento mecânico que permite a movimentação dos elos 2 e 3, ou seja, o deslocamento vertical e a mudança da orientação do efetuador final. A figura 2.6 ilustra este mecanismo.

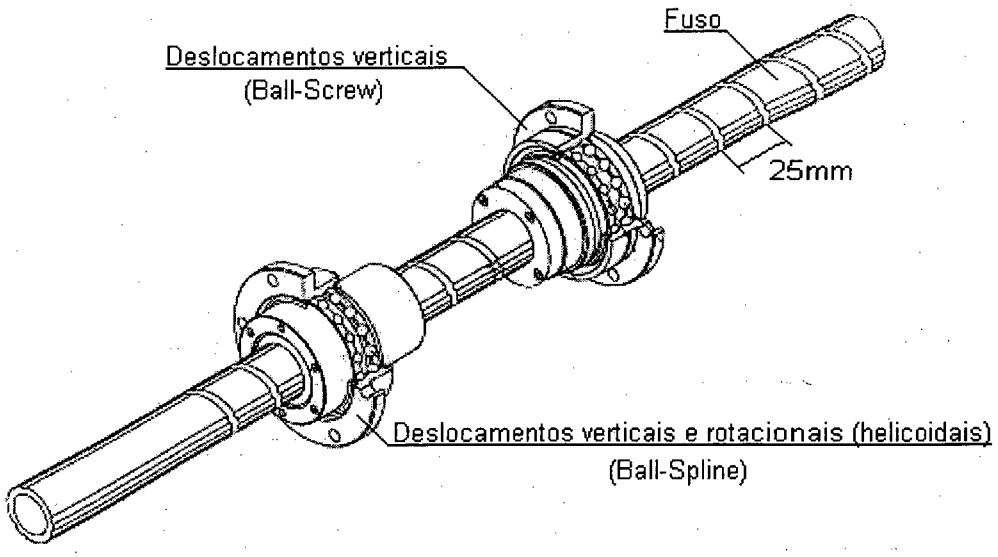
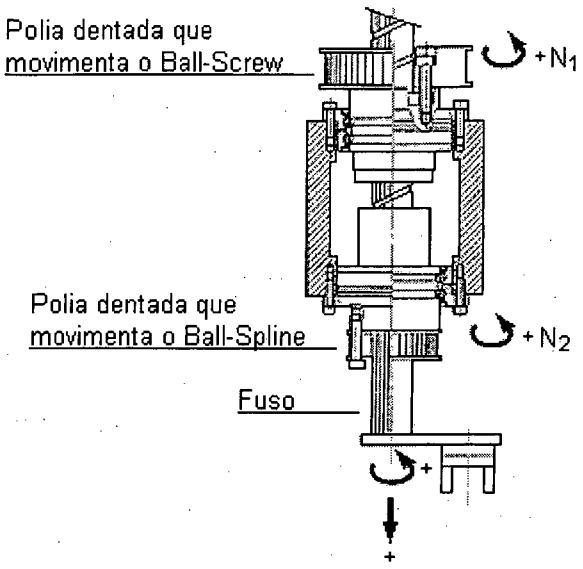


FIGURA 2.6 - Ball-Screw-Spline.

Nesta figura, a peça superior (ball-screw) provoca unicamente movimentos verticais no fuso, na razão de 25mm por volta. A peça inferior (ball-spline) provoca movimentos verticais, na mesma razão da anterior, e ao mesmo tempo rotaciona o fuso, gerando um movimento helicoidal.

Estas peças são acionadas pelos motores 2 e 3, respectivamente. A figura 2.7 ilustra a montagem deste mecanismo bem como as convenções adotadas de giros e deslocamentos.



p: razão de deslocamento do fuso

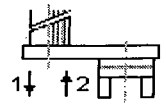
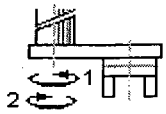
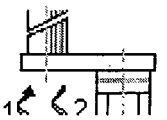
N_1: velocidade de rotação do Ball-Screw

N_2: velocidade de rotação do Ball-Spline

FIGURA 2.7 - Funcionamento do Ball-Screw-Spline.

O funcionamento deste mecanismo é resumido na tabela 2.2.

Tabela 2.2 - Movimentos do Ball-Screw-Spline

Movimento		Sentido de deslocamento	Engrenagem/transmissão Ball-Screw	Engrenagem/transmissão Ball-Spline	Movimento do fuso Vertical (velocidade)	Movimento do fuso Sentido de rot. (vel. rot. polias)
	1	vertical → para baixo rotação → 0	N_1 (direita)	0	$V=N_1.p$ ($N_1 \neq 0$)	0
	2	vertical → para cima rotação → 0	$-N_1$ (esquerda)	0	$V=-N_1.p$ ($N_1 \neq 0$)	0
	1	vertical → 0 rotação → direita	N_1	N_2 (direita)	0	N_2 (direita) ($N_1=N_2 \neq 0$)
	2	vertical → 0 rotação → esquerda	$-N_1$	$-N_2$ (esquerda)	0	$-N_2$ (esquerda) ($N_1=N_2 \neq 0$)
	1	vertical → para cima rotação → direita	0	N_2 ($N_2 \neq 0$)	$V=N_2.p$	N_2 (direita)
	2	vertical → para baixo rotação → esquerda	0	$-N_2$ ($-N_2 \neq 0$)	$V=-N_2.p$	$-N_2$ (esquerda)

2.5.3 Estabelecimento da Relação entre as Juntas 2 e 3

Visto o funcionamento do Ball-Screw-Spline, veremos como fica a relação entre as juntas 2 e 3.

A junta 2 provoca movimentos verticais no fuso na razão 25mm/volta da polia dentada.

A junta 3 provoca movimentos rotacionais no fuso na razão de 1volta/volta da polia dentada. Ao mesmo tempo, porém, esta junta provoca movimentos verticais no fuso, na mesma razão de 25mm/volta da polia. Isso significa que, ao se mudar a orientação do efetuador final, sua posição também será alterada, já que esta rotação provoca um deslocamento vertical do elo 2 (25mm para cima ou para baixo a cada volta, de acordo com o sentido de rotação).

Este efeito pode ser anulado se provocarmos um deslocamento na junta 2 de igual valor, mas no sentido contrário. Se nos referirmos aos motores, estes também deverão girar em sentidos contrários, mas em proporções diferentes, já que as reduções não são as mesmas. No caso acima apresentado, o motor 2 deverá girar numa velocidade menor que o motor 3 (e em sentido contrário, logicamente) para manter constante a posição horizontal do manipulador final. A relação correta entre estes motores é apresentada a seguir.

- a junta 2 possui uma única redução de 32/54 ou de 1/1,6875. Ou seja, para um deslocamento de 25mm, o motor 2 realiza 1,6875 voltas. Para um deslocamento de 260mm o ball-screw realiza 10,4 voltas e o motor 2 realiza 17,55 voltas.

- a junta 3 possui duas reduções: uma de 1/4,5 e outra de 15/54. A relação final fica sendo o produto dessas duas, o que resulta em 1/16,2. Isto significa que para cada volta do efetuador final (há ainda 25mm de deslocamento vertical) o motor 3 realiza 16,2 voltas. Assim, para um deslocamento de 260mm, o ball-spline também realiza 10,4 voltas, mas o motor 3 deve girar 168,48 vezes.

- a relação final entre os motores 2 e 3 fica sendo $^{27}/_{259,2} = {}^{1}/_{9,59}$. Ou seja, se quisermos manter constante a posição do efetuador final durante uma mudança em sua direção, a cada 9,59 voltas do motor 3 o motor 2 deve realizar uma no sentido contrário

A tabela 2.3 resume estas relações.

Tabela 2.3 - Relação entre as juntas 2 e 3

Posição dos motores	Relação	Para deslocamento de 260mm*	
motor 2	$q_{m2} = q_2.1,6875$	$q_2 = 10,4$ voltas	$q_{m2} = 17,55$ voltas
motor 3	$q_{m3} = q_3.16,2$	$q_3 = 10,4$ voltas	$q_{m3} = 168,48$ voltas

* Curso do elo 2, conforme definido via software. Ver também a figura do espaço de trabalho cartesiano para o eixo Z.

2.5.4 Motores

A figura 2.8 ilustra a montagem dos motores no robô.

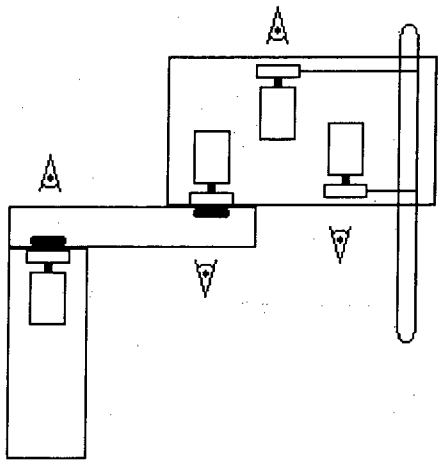


FIGURA 2.8 - Esquema de montagem dos motores.

A convenção adotada para o sentido positivo de rotação dos motores é a seguinte: (considerando a vista superior do motor, individualmente, como mostrado na figura anterior):

A cada motor está associado um encoder, que faz a leitura da posição de acordo com o sentido de rotação do eixo do motor. Desta forma é necessário estabelecer uma relação entre o sentido de rotação do eixo do motor e o que acontece no respectivo elo para que o sinal que o encoder envia ao controlador seja coerente com o movimento do elo (por exemplo, se o elo gira positivamente, é este o sinal que o controlador deve receber, independente do sentido de rotação do motor). Para adequar o sentido de rotação do elo com o encoder, utilizamos uma relação de transmissão positiva ou negativa, apresentada para cada elo na tabela 2.4.

Tabela 2.4 - Relação de transmissão dos motores

Motor	Giro do Motor	Giro do Elo	Relação
0	+	-	- 100
1	+	-	- 100
2	+	+	+ 54/32
3	+	+	+ 4,5.54/15

As relações mostradas na tabela foram determinadas considerando-se a disposição dos motores utilizados na montagem de cada elo e as convenções adotadas para sentidos positivos e negativos de rotação. Estas relações foram posteriormente verificadas na prática.

2.6 Espaço de Trabalho

O espaço de trabalho de um manipulador é definido pelo conjunto de pontos x,y, e z que podem ser atingidos por seu efetuador final. Ao definir o espaço de trabalho temos de utilizar um sistema de coordenadas como referência. Para definição do espaço de trabalho do robô Inter foi utilizado como referência o sistema de coordenadas identificado na figura 2.2 com o índice O_o. Este sistema define a posição e orientação do robô.

2.6.1 Espaço de Trabalho - Plano XY

Na figura 2.9 está ilustrado o espaço de trabalho do robô Inter no plano XY, gerado exclusivamente pelos movimentos de rotação das juntas 1 e 2.

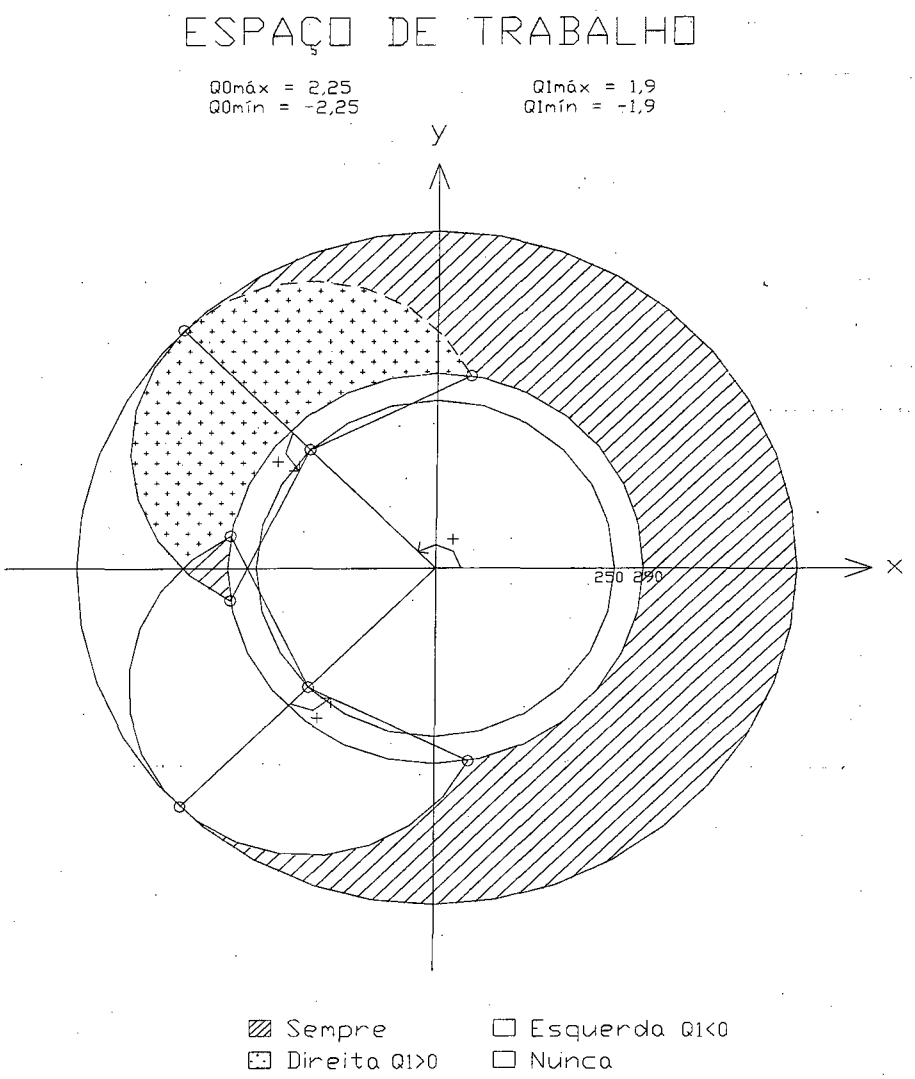


FIGURA 2.9 - Espaço de trabalho no plano XY.

O elo 1 realiza rotações de ± 2,25 rad (aproximadamente ± 129°). Já o elo 2 é limitado em ± 1,9 rad (≈ ±109°). É importante notar no desenho a convenção utilizada para definir o sentido dos giros, que é positivo de x^+ para y^+, e negativo de x^+ para y^-.

Aqui cabem algumas observações sobre singularidades: em posições singulares há movimentos infinitesimais que não são atingíveis, ou seja, o elo em questão não pode se mover em determinadas direções e consequentemente perde-se um ou mais graus de liberdade. Também, nas proximidades de configurações singulares, não existirá solução única para o problema da cinemática inversa - neste caso haverá infinitas soluções ou não haverá solução e velocidades limitadas no espaço cartesiano podem gerar velocidades ilimitadas no espaço de juntas - isso também vale para forças e torques. Cabe notar ainda que uma movimentação de θ_1^+

para θ_1^-, no espaço de trabalhando cartesiano, a junta 1 teria que passar por uma região singular, o que o próprio software não permite. Para informações detalhadas sobre singularidades o leitor pode consultar, como sugestão, *[5]*.

Na figura do espaço de trabalho é possível identificar 4 regiões distintas. Estas regiões estão relacionadas com a configuração do robô.

A configuração à direita é definida com $\theta_1>0$ e à esquerda para $\theta_1<0$. Estas configurações geram as áreas referenciadas como *Direita* e *Esquerda* na figura 2.9 e que, conforme será visto, só são atingidas em configurações particulares das juntas 0 e 1. A área referenciada como *Sempre* pode ser atingida com qualquer configuração, ou seja, tanto com $\theta_1>0$ como com $\theta_1<0$.

Suponha que a junta 0 tenha sofrido um deslocamento angular de 2,25 rad a partir da origem, ou seja, está no seu deslocamento máximo positivo (θ_0=2,25rad). Suponha ainda que o elo 1 esteja alinhado com o elo 0, ou seja θ_1=0. Se a partir desta posição apenas a junta 1 realizar um deslocamento angular de 0 a -1,9 rad, a extremidade do elo 1 percorrerá a linha tracejada da figura. Mantendo o deslocamento da junta 1 neste intervalo de 0 a –1,9 rad e deslocando a junta 0 de 2,25 à –2,25 rad, observa-se que nesta situação o interior da região (+) nunca é atingida. Se por outro lado a junta 1 deslocar-se no intervalo de 0 à +1,9 rad, para qualquer deslocamento permitido da junta 0, a região (.) nunca será atingida.

O interior da região (+), portanto, só é atingido pela extremidade do elo 1 se a junta 1 se deslocar à direita, ou seja, somente se $\theta_1 > 0$. Claro que se o limite de θ_0=2,25 rad fosse maior, digamos 3 rad, seria possível atingir o interior desta área com um θ_1 negativo. Como θ_0 é limitado fisicamente em 2,25 rad, isso não é possível. Assim, vemos que o interior da região denominada "Direita" só é atingido com $\theta_1 > 0$.

Para a região esquerda vale o análogo, ou seja, seu interior só é atingido pela extremidade do elo 1 com $\theta_1 < 0$.

É importante observar que θ_1 pode ser positivo ou negativo com $\theta_0 > 0$ e com $\theta_0 < 0$. O interior das regiões (+) e (.) é que somente é alcançado nas configurações particulares acima citadas. As áreas em branco claramente nunca são atingidas, qualquer que seja a combinação dos ângulos θ_0 e θ_1.

Ao se trabalhar no espaço cartesiano deve-se optar por uma das configurações descritas, pois não é possível passar de $\theta_1>0$ para $\theta_1<0$ e vice-versa. Isto reduz o espaço de trabalho do robô para a área gerada pela configuração (além da área comum a ambas configurações, claro). Há ainda um limite de ± 0,15 rad para θ_1, definido via software, como margem para a região singular.

Para passar de uma configuração para outra, deve-se obrigatoriamente trabalhar no espaço de juntas (isto é, faz-se o deslocamento da respectiva junta, no caso a junta 1, até se "ultrapassar" a configuração singular) e em seguida voltar a trabalhar no espaço cartesiano.

2.6.2 Espaço de Trabalho - Eixo Z

Na figura 2.10 estão definidas as posições máxima e mínima atingidas pelo efetuador final na direção z, representadas em milímetros. Os deslocamentos verticais do elo 2 podem ser referenciados segundo o sistema da junta ou segundo o sistema cartesiano, indicados na figura.

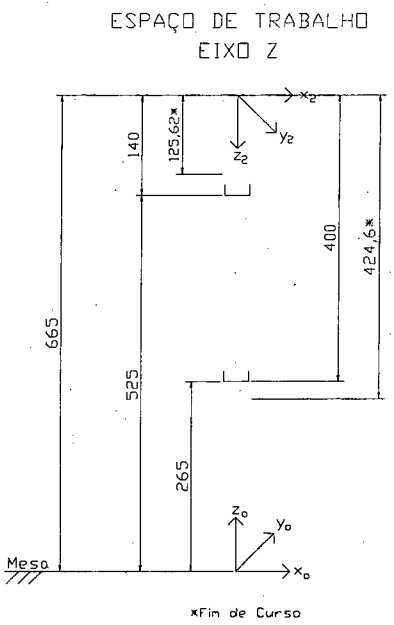


FIGURA 2.10 - Espaço de trabalho no eixo Z.

Conforme se pode ver na figura 2.10, há duas representações para o manipulador final: uma tracejada e outra em linha cheia. A primeira se refere aos limites mecânicos de deslocamento vertical do elo 2, determinados por sensores de

fim de curso. A segunda, em linha cheia, se refere aos limites definidos em software para estes deslocamentos. A tabela 2.5 resume os cursos do manipulador final:

Tabela 2.5 - Limites do eixo Z

Tipo de limitação	Diferença segundo referenciais apresentados no desenho	Curso (mm)
Limites Mecânicos	424,6 - 125,62	298,98
Limites Definidos pelo Software	400 - 140 ou 525 - 265	260

Portanto, no espaço de juntas, vemos que o espaço de trabalho na direção z fica compreendido entre 140 e 400 mm. Já no espaço cartesiano trabalhamos com valores entre 265 e 525 mm.

Estes limites no eixo z podem ser atingidos em qualquer ponto do espaço de trabalho definido para o plano x-y.

2.6.3 Limitações nos Deslocamentos do Robô

Na verdade todas as juntas são limitadas por software em relação aos limites impostos pelos sensores de fim de curso. Sempre que o robô chega a algum dos fins de curso, desliga automaticamente como medida de segurança. Assim, estando sujeito a oscilações antes de estabilizar em uma posição desejada, se utilizássemos o mesmo limite mecânico para o software, é bastante provável que o robô viesse a acionar o sensor de fim de curso ao se aproximar destes limites. Isto interromperia a seqüência de trabalho do robô e para evitar que isso venha a ocorrer seguidamente é que são estabelecidas estas limitações de curso via software.

A figura 2.11 exemplifica este caso, onde a limitação da junta, configurada por software, coincide com o sensor de fim de curso e a posição desejada é muito próxima do limite.

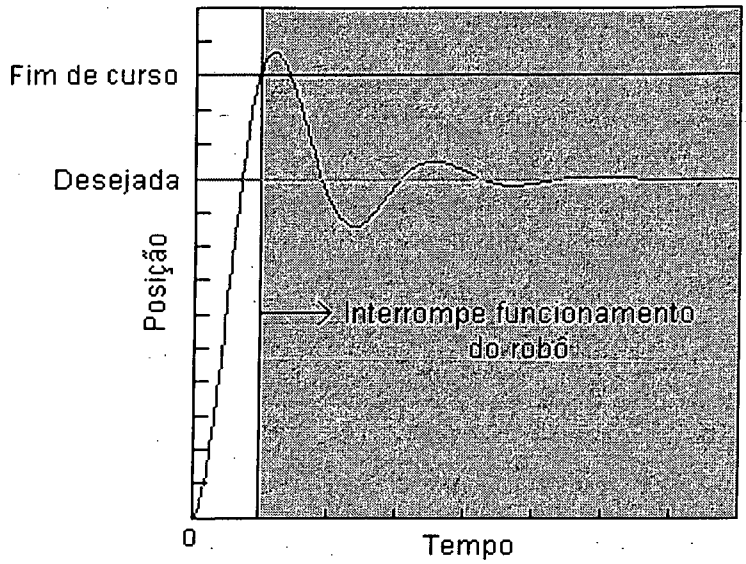


FIGURA 2.11 - Limites de deslocamento próximo ao sensor de fim de curso.

Neste caso, o sensor de fim de curso seria acionado mesmo a posição desejada estando no interior do espaço de trabalho.

2.7 - Dados e Tabelas de configuração

Dados de configuração podem ser entendidos como o conjunto básico de informações, exprimindo principalmente grandezas físicas inerentes ao robô. Estes dados somente devem ser alterados em situações especiais. Compreendem as velocidades, acelerações e momentos máximos para cada junta, posições de sincronização e todas as escalas de conversão que nos permitem utilizar diretamente valores no espaço de juntas. Veremos a seguir como foram determinados estes dados para o robô e como são utilizadas as escalas de conversão.

2.7.1 - Motores, HDs, Transmissões e Amplificadores

A tabela 2.6 apresenta um resumo dos dados obtidos dos catálogos de motores, HD e das reduções das transmissões.

Tabela 2.6 - Características dos motores, harmonic drives e transmissões

Dispositivo	Tipo	RPM	Torque (Nm)	Torque Máx. (Nm)	Constante (Nm/A)
Motor 0	BLS-72	9.000	1,9	11,1	0,5
HD 0	32-100	3.600	137/100	333/100	-
Motor 1	BLS-55	10.000	0,7	4,65	0,47
HD 1	25-100	3.600	67/100	157/100	-
Motor 2	BLS-55	10.000	0,7	4,65	0,47
Redução 2	54/32	-	-	-	-
Motor 3	BLS-55	10.000	0,7	4,65	0,47
Redução 3	4,5.54/15	-	-	-	-

Os amplificadores possuem um software integrado onde é possível estabelecer a velocidade máxima que pode ser atingida pelos motores a eles associados. No robô Inter, a velocidade máxima de 3.600 rpm foi configurada na EPROM dos amplificadores para todos os motores. Isso se deve ao fato de os HDs limitarem a velocidade máxima das juntas 0 e 1, conforme se pode verificar na tabela acima. Para as juntas 2 e 3 este valor especificado para a velocidade máxima foi calculado com um bom coeficiente de segurança, pois a transmissão por correia dentada é mais robusta que pelo HD. Se no futuro for necessário uma velocidade maior para estas juntas, um estudo mais aprofundado deve ser feito. Ao ultrapassar esta velocidade de 3.600 rpm, o amplificador detecta um erro e pára os motores. Logo, a velocidade máxima de cada motor é na prática de 3.600 rpm.

A tabela 2.7 apresenta as correntes máximas permitidas nos amplificadores.

Tabela 2.7 - Características dos amplificadores

	Tipo	Corrente Máxima (A)	Corrente de Trabalho (A)
Amp. 0	220/12	13,8	6,0
Amp. 1	220/04	4,4	2,0
Amp. 2	220/04	4,4	2,0
Amp. 3	220/04	4,4	2,0

2.7.2 - Cálculo de Torques e Correntes

Os torques em cada eixo não devem exceder o limite máximo suportado pelo elemento mais frágil do conjunto motor-redução. Como o torque é proporcional à corrente que passa pelos motores, é necessário verificar também se cada um destes motores suporta a corrente máxima fornecida pelo amplificador associado. Caso a corrente máxima seja superior, o amplificador deve ter sua corrente máxima limitada. Esta configuração também é feita via software.

Abaixo apresentamos os cálculos para cada um dos elos:

Elo 0: limitado pelo HD, que suporta um torque menor que o motor.

Torque (limitado pelo HD): $\frac{137}{100} = 1{,}37 \text{ Nm} \rightarrow \frac{137[Nm]}{0{,}5[Nm/A]} = 2{,}74 \text{ A}$

Torque Máximo (limitado pelo HD): $\frac{333}{100} = 3{,}33 \text{ Nm} \rightarrow \frac{3{,}33[Nm]}{0{,}5[Nm/A]} = 6{,}66 \text{ A}$

Foi configurado via software para: Corrente: 2,74 A

Corrente Máxima: 6,66 A

Elo 1: limitado pelo HD, que suporta um torque menor que o motor.

Torque (limitado pelo HD): $\frac{67}{100} = 0{,}67 \text{ Nm} \rightarrow \frac{0{,}67[Nm]}{0{,}47[Nm/A]} = 1{,}42 \text{ A}$

Torque Máximo (limitado pelo HD): $\frac{157}{100} = 1{,}57 \text{ Nm} \rightarrow \frac{1{,}57[Nm]}{0{,}47[Nm/A]} = 3{,}34 \text{ A}$

Foi configurado via software para: Corrente: 1,42 A

Corrente Máxima: 3,34 A

Elos 2 e 3: as transmissões (por correia dentada) suportam torques de ordem muito maior que os motores. Assim, o torque é limitado pelos motores.

Torque (limitado pelo motor): $\frac{0,7[Nm]}{0,47[Nm/A]} = 1,49\,A$

Torque Máximo (limitado pelo amplificador): $\frac{4,65[Nm]}{0,47[Nm/A]} = 9,89\,A$ (motor)

$\therefore 4,4\,A$ (amplificador)

Os amplificadores não fornecem uma corrente de 9,89A e portanto o torque máximo nos motores é limitado pelos próprios amplificadores, ou seja, em 4,4A. Isso significa que, na prática, estes motores irão gerar no máximo 2,07 Nm de torque, ao invés dos 4,65 Nm que suportam.

Configuração do amplificador:

Corrente: 1,49 A

Corrente Máxima: 4,44 A

2.7.3 Calibração e Sincronização

A cada motor do robô Inter está associado um "encoder" de escala incremental. Sendo de escala incremental, estes dispositivos não armazenam a posição absoluta do eixo respectivo e sim o número de pulsos gerados através de uma rotação qualquer. Assim, ao se ligar o robô, não é possível saber em que posições absolutas estão os elos e desta forma é necessário realizar um processo de sincronização de cada "encoder".

Descreveremos rapidamente o processo de sincronização do elo 0, que vale para os demais. O elo 0 é primeiramente deslocado para a posição onde θ_0 deve ser zero rad, isto é, no sentido do eixo x_0^+. Em seguida é deslocado até o seu fim de curso positivo e este deslocamento angular é medido e registrado. Repete-se o procedimento para a posição máxima negativa. A partir destes dados é calculada a posição intermediária e a posição de sincronização.

A posição intermediária define a posição de sincronização somente para a junta 3, devido à dificuldade de tomarmos algum ponto como referência. Para as demais juntas, a posição de sincronização é definida para o deslocamento máximo positivo ou negativo, medido para cada uma delas. O sentido de deslocamento do elo durante o

processo de sincronização deve concordar com o valor utilizado como posição de sincronização. Se por exemplo a posição de sincronização for negativa, o elo deverá se deslocar no sentido negativo durante a sincronização.

As posições de sincronização (*SynchPos*) utilizadas são as mostradas na tabela 2.8.

Tabela 2.8 - Posição de sincronização

SynchPos	Eixo 0	Eixo 1	Eixo 2	Eixo 3
(rad)	-2,5102	2,0992	0,12562 (m)	-2,826

O problema apresentado por este processo de sincronização é que a posição de sincronização foi obtida a partir de um posicionamento não muito preciso dos elos, já que não se dispõe, no momento, de ferramentas adequadas para isto. Idealmente, esta calibração deveria ser feita posicionando-se o manipulador numa posição e orientação bem conhecidas de seu espaço de trabalho. Por exemplo, poderíamos ter uma ferramenta posicionada em um ponto preciso x, y e z no espaço cartesiano. Para sincronizar o robô bastaria posicionar o manipulador final do robô em contato com esta ferramenta para definir seu posicionamento absoluto. Teríamos desta forma um processo rápido e eficiente de sincronização.

2.7.4 - Conversão dos Pulsos do Encoder em Posição dos Elos

Os "encoders" são integrados aos motores e apresentam resolução de 4096x4 = 16.384 pulsos por volta do motor. Isto significa que trabalhamos com uma resolução de $2\pi/16.384 = 38{,}35 \times 10^{-3}$ rad para o posicionamento do motor.

Para transformar rotações do motor em movimentos das juntas do robô, devemos utilizar as relações de transmissão anteriormente apresentadas.

A tabela abaixo apresenta a escala para transformar pulsos do encoder em posição do robô:

Tabela 2.9 - Transformação de pulsos em radianos

Eixo	Escala robô [rad robô] [0 ... 2π/gear]	Pulsos do encoder
Eixo 0	[0 ... -0,0628]	[16383 ... 0]
Eixo 1	[0 ... -0,0628]	[16383 ... 0]
Eixo 2	[0 ... 3,72]	[16383 ... 0]
Eixo 3	[0 ... 0,387]	[16383 ... 0]

Esta escala não define nenhum tipo de limite de deslocamento para as juntas do robô. Para os eixos 0 e 1 o valor de -0,0628 rad não significa obviamente que o robô só pode trabalhar neste intervalo.

O deslocamento mínimo possível de ser observado no motor é 1 pulso. Para cada junta, este pulso representa um deslocamento angular distinto, de acordo com a relação de transmissão. O deslocamento angular mínimo possível de ser medido em cada junta é calculado a seguir e será referenciado como a resolução da junta:

Eixos 0 e 1 $\rightarrow$ 1 pulso = 0,0628 / 16384 = $3,835 \times 10^{-6}$ rad

Eixo 2 $\rightarrow$ 1 pulso = 3,72 / 16384 = $2,27 \times 10^{-4}$ rad = 227×10^{-6} rad

Eixo 3 $\rightarrow$ 1 pulso = 0,387 / 16384 = $2,36 \times 10^{-5}$ rad = $23,6 \times 10^{-6}$ rad

A resolução da junta 2 refere-se ao posicionamento angular do conjunto que provoca o deslocamento longitudinal. A relação entre posição angular e deslocamento longitudinal é de 25mm/volta. Logo, 227×10^{-6} rad correspondem a $227 \times 10^{-6} \times 2\pi \times 25$ = 0,0356mm, que é a resolução da junta 2 na direção z.

Estas resoluções indicam a máxima precisão de posicionamento que pode ser obtida com este robô, mas não determinam a precisão real de posicionamento do robô.

2.7.5 - Relação Entre Torques, Tensões, Correntes e Pulsos

Os amplificadores do robô trabalham numa faixa de tensão de entrada de ±10V, que corresponde á configuração de corrente máxima fornecida por cada amplificador. Por exemplo, uma tensão de +10V corresponde a uma corrente de 6,66A para o amplificador 0 e 3,33A para o amplificador 1, como pode ser visto na tabela 2.10.

A configuração do ambiente XOberon permite a criação de uma escala que converte os valores de torque fornecidos via software em pulsos (ticks) na interface Digital/Analógica ou vice-versa. Podemos, por exemplo, fornecer através do ambiente um torque desejado (em Nm) em uma junta e o software trata de converter automaticamente este valor em pulsos. Do mesmo modo, recebemos pulsos do encoder e tacômetro, que serão transformados em posições e velocidades da junta, respectivamente.

O número máximo de pulsos do sistema para a conversão D/A e A/D é de $2^{12} = 4096$ pulsos, o que também define, em parte, a resolução do sistema.

A tabela 2.10 apresenta as relações entre torques, tensões, correntes e pulsos:

Tabela 2.10 - Transformação de torques em pulsos

Nm → Pulsos	Gear	Máx. A amp. (A)	Torque máx. motor (Nm) (1)	Torque no robô (junta) (Nm)	Torque no motor (Nm)	Corrente no motor (A)	Tensão no amplificador (V)	Pulsos
Eixo 0	-100	6,66	3,33	[333... -333]	[3,33 ...-3,33]	[-6,66 ...6,66]	[-10...10]	[0...4095]
Eixo 1	-100	3,33	1,57	[157...-157]	[1,57 ...-1,57]	[-3,33 ...3,33]	[-10...10]	[0...4095]
Eixo 2	54/32	4,4	2,07	[-3,493 ...3,493	[-2,07 ...2,07]	[-4,4 ...4,4]	[-10...10]	[0...4095]
Eixo 3	4,5.54/32	4,4	2,07	[-33,53 ...33,53]	[-2,07 ...2,07]	[-4,4 ...4,4]	[-10...10]	[0...4095]

(1) torque máximo do motor na prática, conforme corrente máxima (configurada) fornecida pelo respectivo amplificador.

Exemplo: se na junta 0 desejamos aplicar 100Nm, podemos utilizar por exemplo o comando *Ampl0.Write(100)*. Automaticamente teremos a conversão deste valor em tensão e pulsos, conforme demonstrado abaixo:

-333 [Nm] → 10 [V]
100 [Nm] → x [V]

x = 100.10 / -333
x = -3,003 V

20 [V] → 4095 [pulsos]
3,003 [V] → x [pulsos]

x = 3,003.4095 / 20
x = 614 pulsos

Os 20 volts se referem à escala total dos amplificadores, ou seja, de -10 a +10 V.

Foi calculado o número de pulsos para 3,003V = 614.

Deve ser determinado, também, o número absoluto de pulsos para 0 V e deste valor deve ser subtraído o número de pulsos para o intervalo de 3,003 V. Desta forma é calculado o número de pulsos absolutos que representam os 100 Nm:

$$-3{,}003\ \text{V} = \frac{4095}{2} - 614 = 1434\ \text{pulsos}$$

Esta configuração é feita no arquivo *ROBAllBoot.Config* e o usuário não precisa se preocupar com conversões AD/DA, podendo trabalhar diretamente com valores de Nm para representar torques e radianos para representar a posição das juntas.

2.7.6 - Resumo

A tabela 2.11 apresenta um resumo dos dados de configuração.

Tabela 2.11 - Resumo dos dados de configuração

	Junta 0	Junta 1	Junta 2	Junta 3
Swmin (rad)	-2,5102	-2,0980	0,1256 (m)	-3,1478
Swmax (rad)	2,5335	2,0992	0,4246 (m)	2,5049
$\pm\frac{\text{Swmax}+\lvert\text{Swmin}\rvert}{2}$ (rad)	± 2,523	± 2,1017	0,298 (m)	± 2,8263
SynchPos (rad)	-2,5102	2,0992	0,1256 (m)	-2,826
Velocidade Máx. (rad/s)	3,77	3,77	0,888 (m/s)	23,27
Pos. Máx. Config. (rad)	2,25	1,9	0,4	2,5
Pos. Min. Config. (rad)	-2.25	-1,9	0,14	-2,5
Vel. Máx. Config. (rad/s)	3,00	3,00	0,88 (rad/s)	20
Vel. Mín. Config. (rad/s)	-3,00	-3,00	-0,88 (rad/s)	-20
Torque Máx. (Nm)	± 333	± 157	± 3,493	± 33,53
Acc. Máx. (rad/s^2)	± 80	± 100	± 3000 mm/s^2	± 500
Acc. Medida (rad/s^2)	95 (q_1 estendido) 139 (q_1 retraído)	175		

Os valores das acelerações máximas para cada junta (Acc. Máx. em rad/s^2) foram baseados em dados de um robô semelhante, existente no Instituto de Robótica na Suíça.

A aceleração medida vem de testes práticos. Nestes testes, q1 estendido refere-se à aceleração estimada com o elo 1 "estendido", ou seja, com θ_1=0. Da mesma forma, q1 retraído refere-se à aceleração medida com o elo 1 "retraído", ou seja, com θ_1=-1,9 rad. Note que com q1 retraído a aceleração do eixo 0 é bem maior (mais rápida), como era de se esperar.

As velocidades máximas foram calculadas através das seguintes expressões:

Juntas 0 e 1 $\rightarrow$ $3600.2\pi / 100.60 = 3,77$

Junta 2 $\rightarrow$ $\frac{3600x0,025}{60x54 / 32}$

Junta 3 $\rightarrow$ $\frac{3600x2\Pi}{60x54x4,5 / 15}$

2.8 Armário de Controle

No armário de controle estão montados todos os dispositivos de comunicação, controle e alimentação elétrica do robô Inter. A figura 2.12 mostra este armário, onde estão identificados os principais componentes.

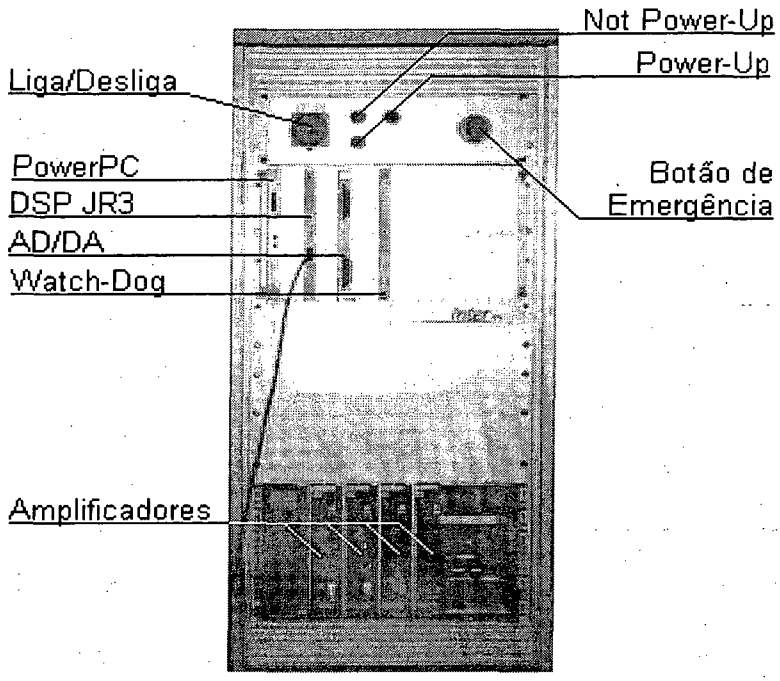


FIGURA 2.12 - Armário de controle.

Placa PowerPC:

- barramento VME (clock bus de 67 MHz)
- clock de 200 MHz
- memória RAM de 16 Mb
- cache de 256 Kb

Amplificadores: total de quatro. Podem ser configurados individualmente, através do software que os acompanha, de acordo com os motores que alimentam, podendo-se configurar correntes de trabalho e máxima. Esta configuração é salva na EPROM do respectivo amplificador. A comunicação entre o computador e o amplificador em questão é feita via serial (a entrada fica na parte frontal do armário). Cada amplificador pode trabalhar com momentos ou velocidades como valores

desejados. No interior do armário há uma chave associada a cada amplificador que muda o modo de trabalho de torque para velocidade (e vice-versa). Todos os softwares atualmente em uso foram escritos para o modo torque. Se o usuário desejar trabalhar no modo velocidade deverá fazer as respectivas alterações nos controladores.

Power-Up e Not-Power-Up: o botão power-up habilita a alimentação dos motores do robô. Assim, para o funcionamento do robô é necessário que este botão esteja acionado. O botão not-power-up, por sua vez, corta a alimentação dos motores, não desligando o armário de controle. Deve, portanto, ficar claro para o usuário a diferença entre os botões liga/desliga (que se refere ao armário de controle) e power-up/not-power-up (que se referem aos motores do robô).

Botão de emergência: pode ser acionado tanto no armário como no "controle remoto de emergência". Quando acionado, o robô pára imediatamente a execução da tarefa e vai para o estado "emergency". O botão power-up é automaticamente desacionado.

JR3: placa do sensor de força JR3, responsável pelo processamento digital dos sinais do sensor. Alimenta o sensor e recebe os dados via cabo e conector RJ45. Maiores detalhes na seção 1.8.

Watch-dog: dispositivo de segurança do sistema, monitorando o funcionamento dos dispositivos. Responsável pelo isolamento de corrente entre os dispositivos, evitando interferência entre placas e amplificadores (que trabalham com níveis de tensão e corrente muito maiores).

2.9 Sensor de Força

O sensor de força acoplado ao manipulador final do robô Inter, da marca JR3, possui 6 graus de liberdade para medição de forças e torques. Os sinais dos extensômetros de resistência (montados em ponte) são amplificados e convertidos para representações digitais de força e torque por um circuito eletrônico montado dentro do próprio sensor. Portanto, os sinais analógicos (de baixa amplitude) e o circuito A/D estão no interior da carcaça do sensor e protegidos de interferências eletromagnéticas. Estes dados são transmitidos para uma placa que acompanha o JR3, conectada à placa-mãe do armário de controle, através de um cabo serial e na taxa de 8 kHz para cada

eixo. O pacote de dados seriais inclui ainda informações sobre características do sensor, calibração e realimentação da tensão elétrica.

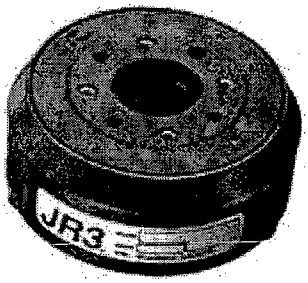


FIGURA 2.13 - O sensor de força.

A placa que acompanha o sensor de força, mostrado na figura 2.13, é responsável por receber e processar os dados seriais transmitidos pelo sensor, bem como monitorar e ajustar automaticamente a alimentação elétrica do JR3. Ela contém um processador digital de sinais (DSP)[1] que desempenha as seguintes funções: remoção de offset, detecção de saturação do sensor, desacoplamento e filtragem dos sinais, cálculo dos vetores de força e momento, detecção de picos e vales, cálculo de translação e rotação de coordenadas e monitoramento de "treshold", entre outras.

A placa possui uma memória RAM em que usuário e DSP podem ler e gravar dados - ou seja, o usuário pode reconfigurar o DSP através de comandos, por exemplo. A memória do DSP é de 16k de palavras de 2 bytes. Estes primeiros 8k de endereços são RAM, enquanto os 8k restantes são reservados para registradores de estado e funções afins. A localização das variáveis na memória[2] deve ser vista no manual do JR3.

Capacidade de carga

A capacidade de carga do sensor para forças e momentos é apresentada na tabela 2.12, conforme dados do manual.

[1] Daqui por diante utilizaremos a abreviatura DSP para nos referirmos indistintamente ao processador digital de sinais como à sua placa.
[2] Na verdade estes endereços são offsets de um endereço base para a RAM da placa-mãe configurados através de dip-switches do DSP. Maiores informações neste documento e no manual do JR3.

Tabela 2.12 - Capacidade de carga do sensor de força

	Fx	Fy	Fz
Força (N)	100	100	200
	Mx	My	Mz
Momento (Nm)	6,3	6,3	6,3

Filtros

Os dados enviados do sensor e desacoplados matematicamente pelo DSP passam por seis filtros passa-baixas em cascata. Cada filtro subsequente tem uma freqüência de corte de ¼ do anterior. A freqüência de corte de cada filtro é de $\frac{1}{16}$ da taxa de amostragem - assim, para uma taxa de 8 kHz, as freqüências de corte são 500 Hz, 125 Hz, 31,25 Hz etc. O atraso de cada filtro é aproximadamente o inverso da freqüência de corte, ou seja, para um filtro de 500 Hz este tempo seria de aproximadamente 2 ms. O manual do JR3 apresenta os gráficos das características dos seis filtros do DSP.

Vetores de força e momento são calculados tanto a partir dos dados desacoplados, bem como a partir dos dados após cada filtro.

Unidades

Diferentes unidades podem ser utilizadas no sensor de força. Atualmente elas estão configuradas para: N, Nm e mm.10.

Execução de comandos

O usuário dispõe de uma série de comandos para interagir com o sensor e com o DSP do JR3. A execução de um comando no JR3 é feita escrevendo-se o código do comando que se deseja executar em posições específicas da memória. Estas posições de execução são chamadas de command_word_2, command_word_1 e command_word_0 e seus endereços são:

command_word_2 → 0x00e5
command_word_1 → 0x00e6
command_word_0 → 0x00e7

A utilização de cada comando disponível no JR3 evidentemente varia conforme o comando, mas em geral o usuário escreve dados em várias posições de memória e finalmente escreve o código do comando desejado nas posições de execução. Os comandos mais usuais são executados em command_word_0, ficando os outros dois normalmente para situações de teste ou naquelas em que há possibilidade

de conflito, ou seja, tanto o DSP quanto usuário escreverem dados na mesma posição de memória ao mesmo tempo.

Por fim, o JR3 irá escrever um zero em command_word_0 pós a execução correta de um comando. Caso haja algum erro, um número negativo é escrito.

Offsets

O tratamento de offsets é bastante útil ao se utilizar diferentes ferramentas acopladas ao sensor de força e/ou em diferentes orientações do manipulador final. O DSP do JR3 é capaz de armazenar 16 conjuntos de offsets. Os valores de offsets atualmente em uso pelo sensor são armazenados nas variáveis "offsets", nas posições de memória de 0x0088 a 0x008d. Em 0x008e, que é a localização da variável offset_num, é indicado o número do offset atualmente em uso.

Os comandos relativos a offsets são:

- `Use Offsets:` 0x060#

Uso/Sintax/Modo de execução: escreve-se 0x060# em command_word_0 (0x00e7), sendo # o número do conjunto de offsets a ser utilizado, podendo variar de 0 a f (o que dá o total de 16).

Este comando carrega para as posições 0x0088 a 0x008d o conjunto de offsets referenciado por #, setando estes offsets na entrada # da tabela. Caso não seja necessário o uso de diferentes offsets, o usuário provavelmente não necessitará executar este comando, senso o offset 0 o de trabalho e default.

- `Set Offsets:` 0x0700

Modo de execução: escreve-se 0x0700 em command_word_0 (0x00e7).

Este comando lê os dados das variáveis offsets (0x0088 a 0x008d), colocando-os como offsets de trabalho e os armazena na tabela de offsets como o conjunto #, conforme indicado na variável offset_num (0x008e). O comando set offsets não é absolutamente necessário já que a escrita direta dos valores de offsets nas suas respectivas variáveis tem o mesmo resultado. Como o DSP verifica se houve mudança de offsets a cada 16 amostragens do sensor, este comando pode ser muito importante nos casos em que delays da ordem de 2 ms são críticos.

- `Reset Offsets:` 0x0800

Modo de execução: escreve-se 0x0800 em command_word_0 (0x00e7).

O comando reset offsets zera os dados de todos os eixos do sensor. Isto é bastante útil na remoção de cargas/taras. Por exemplo, ao se fixar uma ferramenta ao JR3, este estará lendo forças relativas ao seu peso, o que normalmente não interessa em aplicações práticas.

Este comando atualiza as variáveis de offsets (0x0088 a 0x008d) e armazena os seus dados na tabela de offsets sob o número #, conforme indicado por offset_num (0x008e).

Escalas

A variável full_scale indica as escalas atualmente em uso pelo sensor. A resolução do sensor é de 2^{14}, ou seja, um valor de força ou momento lido pelo JR3 é digitalizado em 14 bits. Assim, os dados em uma escala qualquer são tais que + 16384 é igual ao valor máximo da escala em questão. Se por exemplo o usuário for trabalhar numa faixa de valores pequena em relação à escala atual, estará perdendo resolução. Se por outro lado o usuário for trabalhar com valores de força/torque próximos ao valor máximo/mínimo da escala em uso, ela pode facilmente saturar. Assim, o usuário deve ter cuidado ao setar as escalas com que vai trabalhar, tendo em mente uma noção das forças que estarão envolvidas no seu experimento.

Os valores das escalas para forças e momentos podem ser setados pelo usuário através do comando set new full scales:

- `Set New Full Scales:` 0x0A00

Modo de execução: escreve-se 0x0A00 em command_word_0 (0x00e7). Os valores de full scale para Fx, Fy, Fz e Mx, My e Mz (0x0080 a 0x0085) são setados para as novas escalas do sensor.

Este comando demora em média 3 a 10 ms para ser executado. Durante este tempo, o bit indicador de system_busy da palavra errors_bits (0x00f1) é setado, havendo portanto uma condição de "erro". O comando set new full scales também calcula uma transformação de coordenadas como parte de seu processamento, baseado no valor da variável transform_num (0x0077). O usuário deve ficar atento quando da execução deste comando, pois se houver uma "incompatibilidade" entre transform_num pode-se gerar uma transformação de coordenadas diferente da desejada.

Avisos e Erros

Mensagens de aviso e erros do JR3 são apresentados nas seguintes posições de memória:

avisos (warning_bits) → 0x00f0
erros (error_bits) → 0x00f1

Cada uma destas palavras codifica um tipo de aviso ou erro, identificado do bit menos significativo (0) ao mais significativo (15). A especificação de cada tipo de erro e warning é apresentada no manual do JR3 e no módulo JR3.Mod.

Capítulo 3

DESCRIÇÃO DOS PRINCIPAIS MÓDULOS DE SOFTWARE DO ROBÔ

3.1 Introdução

O objetivo deste capítulo é apresentar os módulos de software utilizados na movimentação, descrevendo sua função principal. Maiores detalhes sobre os principais procedimentos destes módulos podem ser encontrados no apêndice C. Informações sobre o sistema operacional XOberon podem ser encontradas no apêndice A.

3.2 Função Básica dos Principais Módulos

Os módulos utilizados na programação do robô Inter podem ser divididos em duas grandes classes: os pertencentes ao sistema operacional XOberon, e aqueles escritos em XOberon, os quais possuem função específica na movimentação e controle do robô.

Não temos informações completas a respeito da estrutura dos módulos inerentes ao XOberon pois nos foi fornecido apenas o arquivo gerado após sua compilação. Não podemos alterar estes módulos, mas devemos conhecer sua função básica e saber utilizar as ferramentas colocadas por eles à nossa disposição.

Já os módulos escritos em XOberon exigem um estudo mais aprofundado pois qualquer módulo adicional que quisermos implementar no robô poderá afetar aqueles anteriormente instalados.

Muitos módulos pertencentes ao XOberon não são freqüentemente utilizados quando se está programando, de modo que não constarão da listagem e do diagrama apresentados a seguir.

Alguns módulos escritos em XOberon e que não estão sendo utilizados pelo robô no momento, também não serão apresentados.

A listagem a seguir mostra os principais módulos e sua função básica.

Módulos pertencentes ao sistema XOberon:

- PPCXOKernel - alocação de memória e coletor de lixo;
- XFiles - manipulação de arquivos;
- Reals – define características de números reais.
- Objects - gerenciamento de bibliotecas;
- XTexts - gerenciamento de textos de entrada e saída;
- MathL - funções matemáticas para LONGREALS;
- Math - funções matemáticas para REALS;
- XOberon - manipulação e controle de tarefas;
- SYSTEM - módulo de comandos com funções relacionadas com o sistema operacional utilizado;
- Base – utilizado na configuração de objetos. É uma extensão de *Objects*.
- ConfigP – utilizado na configuração de objetos;
- Const – define características de constantes.
- Peripherals – configuração de periféricos;

Módulos para movimentação e controle do robô:

- Drive - define características de cada uma das juntas do robô;
- GlobalDefs - define algumas constantes e variáveis utilizadas por vários módulos;
- StrConv - converte números hexadecimais e em decimais e vice-versa;
- RampFilter - faz o escalonamento dos valores desejados de controle;
- MatlabFile2 - gerencia a plotagem de dados;
- PathDefs - define os parâmetros para geração de trajetórias offline;
- Points - define pontos para uma seqüência de movimentos;
- ForceCtrl - algoritmo de controle de força;
- SynchCtrl - algoritmo de controle utilizado durante o processo de sincronização;
- PISpeedCtrl - algoritmo de controle de velocidade;
- StateCtrl - algoritmo de controle de posição;

- SCARACarthKin - transformações entre o espaço de juntas e o espaço operacional;
- Scara23Kinematics - transformação do espaço de juntas para o espaço das transmissões, devido a presença do Ball-Screw-Spline;
- DynCompScara - executa a compensação dinâmica;
- CartesianTransf - transformações entre o espaço operacional e o espaço da tarefa;
- HybridCtrl - módulo para o controle híbrido;
- Main3 - módulo fundamental para movimentação e controle do robô;
- MainTest3 - faz a interface com o usuário dos comandos do módulo Main3;
- RobotLib2 - módulo utilizado pelos demais módulos para a busca de objetos na biblioteca;
- Bahnplaner - módulo de geração de trajetórias;
- BPTest - faz a interface com o usuário dos comandos do módulo Bahnplaner;
- HCTest - faz a interface com o usuário dos comandos do módulo HybridCtrl;
- Inertia - módulo utilizado no cálculo dos parâmetros de inércia do robô;
- Circulo - módulo que gera uma trajetória circular;
- RegulaDemo - módulo que executa uma seqüência de movimentos com o robô;

Para estabelecer o relacionamento entre módulos é comum utilizar diagramas de importação. Nestes diagramas os módulos são apresentados como retângulos e as linhas unindo estes retângulos mostram as relações de importação entre estes módulos. Devido a grande quantidade de módulos existente, optou-se por representar as relações de importação através de números. Cada módulo recebe um número de identificação que será utilizado como referência para os módulos que quiserem importá-lo, como mostrado na figura 3.1. O número de identificação sempre deverá ser maior que o número dos módulos importados.

Nome do módulo

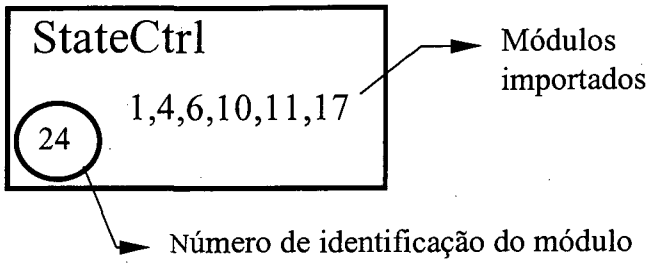


FIGURA 3.1 - Exemplo de representação dos módulos.

A figura 3.2 mostra os principais módulos e as relações de importação. Foram ordenados de forma que os importadores fiquem em linhas inferiores aos importados ou clientes. Estas relações de importação são importantes pois ao modificarmos um módulo devemos saber quais módulos são influenciados por esta modificação.

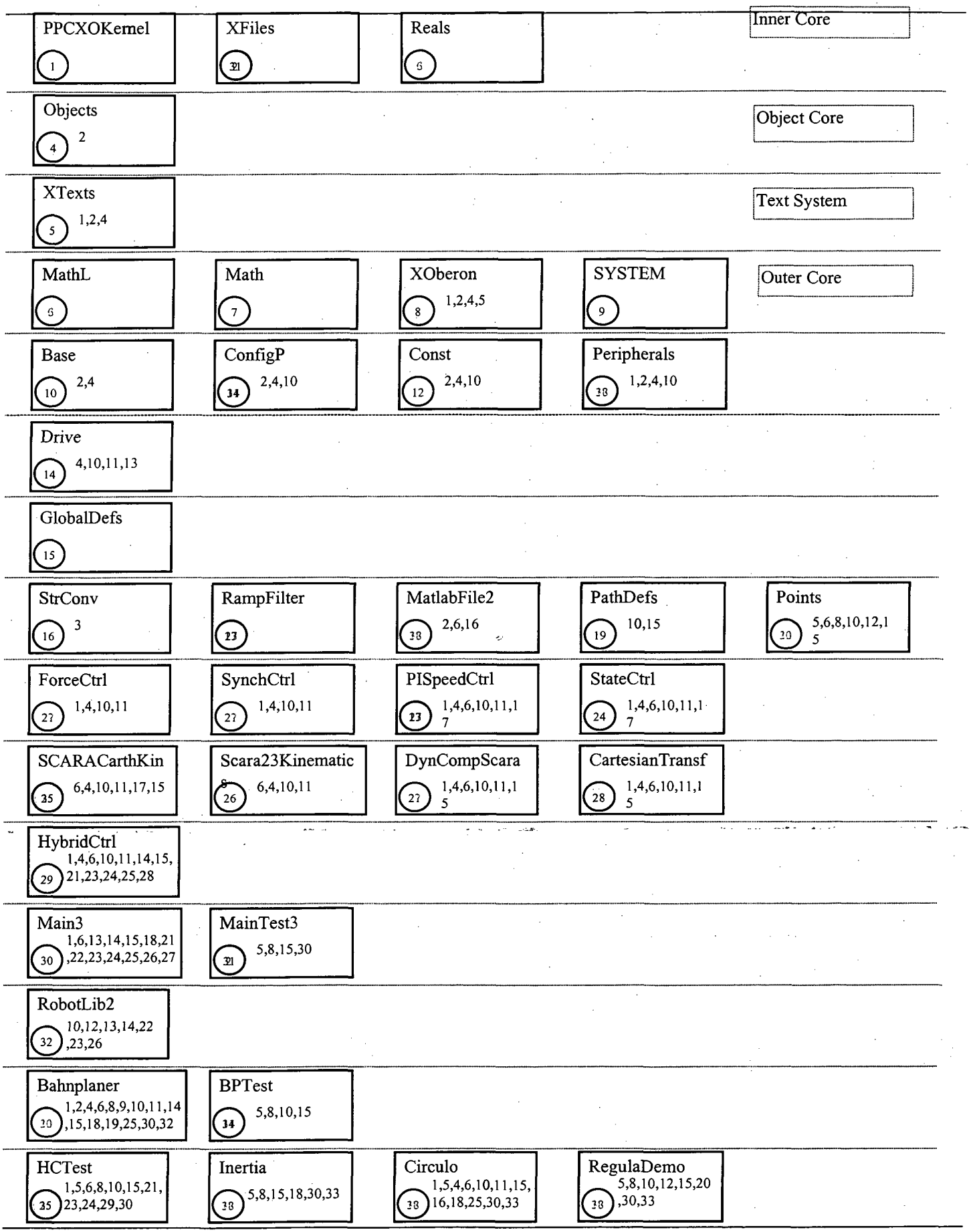


FIGURA 3.2 - Diagramas de importação

3.3 Arquivos de Configuração

Toda vez que o armário de controle é ligado, um arquivo (bootfile) é automaticamente carregado. O arquivo de configuração *ROBAllBoot.Config* é parte integrante do bootfile. Esse arquivo contém uma série de dados fundamentais para o funcionamento do robô.

A primeira parte deste arquivo estabelece os parâmetros de comunicação entre software e hardware e as escalas para conversão de unidades. O resultado disto é que durante a programação podemos utilizar diretamente rad e rad/s para determinar respectivamente as posições e velocidades das juntas do robô e podemos trabalhar com momentos em Nm, sem nos preocuparmos com qualquer tipo de conversão, como mostrado no capítulo 2. Como isto é definido uma única vez, não devendo ser alterado posteriormente, não nos ateremos nesta primeira parte.

A segunda parte trata da configuração de objetos. Nesta etapa são criadas instâncias de objetos e atribuídos valores aos seus campos. Estes objetos ficam a disposição em uma biblioteca, podendo ser buscados a qualquer momento.

A origem de alguns dados configurados como, por exemplo, posição e velocidades máximas das juntas foram explicadas no capítulo dois. Outros dados como os ganhos dos controladores foram em grande parte obtidos a partir de experimentos práticos.

O arquivo *ROBAllBoot* é apresentado integralmente em [7]. As linhas da segunda parte deste arquivo e as explicações relativas a cada um de seus campos são apresentadas no apêndice C.

A configuração de todos objetos constantes neste arquivo poderia ser feita após o armário de controle ter sido ligado através da utilização de chamadas externas. No entanto, a maioria destes dados configurados não é freqüentemente alterada e é conveniente que sejam carregados automaticamente, evitando que esqueçamos de chamar algum deles. Outro benefício é que se fizermos alguma alteração indesejada em algum módulo, o bootfile permanecerá inalterado e o robô com certeza continuará funcionando. Os procedimentos necessários para alterar o bootfile estão descritos no apêndice B.

3.4 O Módulo Drive

Para compreendermos o módulo *Drive* é importante definir exatamente cada uma das 4 juntas do robô. As juntas 0 e 1 são os mecanismos de acionamento dos elos 0 e 1 respectivamente, após a redução imposta pelos Harmonic Drives. A junta 3 é o mecanismo de acionamento do elo 3, após a redução imposta pelas polias dentadas.

Devido a transformação de movimento de rotação em translação através do Ball Screw Spline, a definição da junta 2 é um pouco mais complexa. Entendemos como junta 2 os valores relacionados com o movimento de translação do elo 2, apesar do mecanismo que provoca este movimento estar executando uma rotação. No entanto, o sinal de controle neste caso está vinculado ao mecanismo e não à junta.

A cada uma destas 4 juntas está associada uma série de propriedades como velocidade máxima, posição máxima, aceleração máxima e um estado, isto é, se está freiada e se seus amplificadores estão habilitados ou não. O sinal de controle e os dados sobre posição atual e velocidade atual também são informações associadas às juntas.

Cada junta é movimentada por um motor. Quando buscamos dados do encoder e do tacômetro via hardware, serão fornecidas as posições e velocidades do motor e não das juntas, pois as reduções e transformações ainda não foram levadas em consideração. Mas no arquivo de configuração foi definida uma escala de conversão que transforma os dados relativos ao motor em informações sobre as juntas, sem que precisemos nos preocupar com isto. E quando o motor está freiado e habilitado, a junta correspondente se encontra no mesmo estado. E todos os torques de controle são fornecidos relativamente às juntas(exceto para ajunta 2). Portanto, iremos sempre trabalhar com dados relativos às juntas e não aos motores.

O módulo *Drive* descreve o tipo básico *Drive* e contém os procedimentos para a configuração de objetos deste tipo. Ao término da configuração, cada objeto do tipo *Drive* possui todas as propriedades e os valores associados acima descritos. Apesar de estarmos utilizando a palavra *Drive*, estamos na verdade configurando cada junta do robô. A configuração das quatro juntas é feita através do arquivo *ROBAllBoot.Config* descrito no apêndice C. A linhas de interesse são:

```
Drive.DefDrive Drive0
(ampl=A0, counter=Counter0, enable=Enable0, break=Break0,
        sWMin=SW0Min, sWMax=SW0Max, vmax=3.0, vmin=-3.0, forcemax=333.0,
forcemin= 333.0, posmax=2.25, posmin=-2.25, amax=80, amin=-80) ~
Drive.DefDrive Drive1
(ampl=A1, counter=Counter1, enable=Enable1, break=Break1,
        sWMin=SW1Min, sWMax=SW1Max, vmax=3.0, vmin=-3.0, forcemax=157.0,
forcemin=-157.0, posmax=1.9, posmin=-1.9, amax=100.0, amin=-100.0) ~
Drive.DefDrive Drive2
(ampl=A2, counter=Counter2, enable=Enable2, break=Break2,
        sWMin=SW2Min, sWMax=SW2Max, vmax=0.88, vmin=-0.88, forcemax=6.975,
forcemin=-6.975, posmax=0.40, posmin=0.14, amax=3000.0, amin=-3000.0) ~
Drive.DefDrive Drive3
(ampl=A3, counter=Counter3, enable=Enable3, break=Break3,
        sWMin=SW3Min, sWMax=SW3Max, vmax=20.0, vmin=-20.0, forcemax=16.74,
forcemin=-16.74, posmax=2.5, posmin=-2.5, amax=500.0, amin=-500.0) ~
```

Estes comandos chamam o procedimento *DefDrive* do módulo *Drive*, desencadeando todo processo de configuração.

O módulo *Drive* pode ser encontrado integralmente em [7]. As partes principais deste módulo são descritas em detalhe no apêndice C.

3.5 O Módulo Main3

Main3 é o módulo principal do robô, pois coordena todo o seu funcionamento. Nenhum estado pode ser alterado e nenhum movimento pode ser realizado se o *Main3* não estiver funcionando corretamente.

O estado é uma característica fundamental do robô, pois define em que situação estão os freios, amplificadores, botoeira de emergência, botão liga-desliga entre outras coisas. A seguir mostraremos através de diagramas os principais estados do robô e descreveremos suas características. Nestes diagramas, os estados são representados através de retângulos, os comandos através de círculos e os flags através de losangos.

Flags são variáveis booleanas que podem assumir os valores *TRUE* e *FALSE*. Para a definição dos estados do robô, utilizamos um único flag que define se o robô está sincronizado ou não.

Na parte central do diagrama está o estado em estudo. Na parte superior estão os estados a partir dos quais o estado em estudo pode ser atingido e na parte inferior estão os estados que podem ser atingidos a partir do estado em estudo.

Mas antes de começarmos a explicar os estados do robô, é importante uma observação a respeito do botão Power Up (botão verde situado na porção superior do armário de controle). O módulo *Main3* foi escrito originalmente para outro robô semelhante a este, no qual Power Up era feito via software. No robô Inter, o botão de Power Up só pode ser acionado manualmente por motivos de segurança e , portanto, os comandos feitos via software não possuem mais efeito. Ao acionar Power Up não estamos habilitando os amplificadores, são duas coisas completamente distintas. A habilitação dos amplificadores é feita via software e só pode ser efetivada quando o botão de Power Up já estiver acionado. O desacionamento do botão Power Up, no entanto, ainda pode ser feito via software.

Estado *Initialized* - o robô deve passar automaticamente para este estado logo após o armário de controle ser ligado, pois no arquivo de configuração *ROBAllBoot* já está inserido o comando *CInit*. Também é possível atingir este estado a partir de outros estados, como mostrado no diagrama. A seqüência natural para colocar o robô em funcionamento seria sair deste estado através do comando *Synch* que inicia o processo de sincronização. As demais possibilidades possíveis são apresentadas no diagrama. Neste estado os amplificadores estão desabilitados, o robô está freiado e obrigatoriamente não sincronizado.

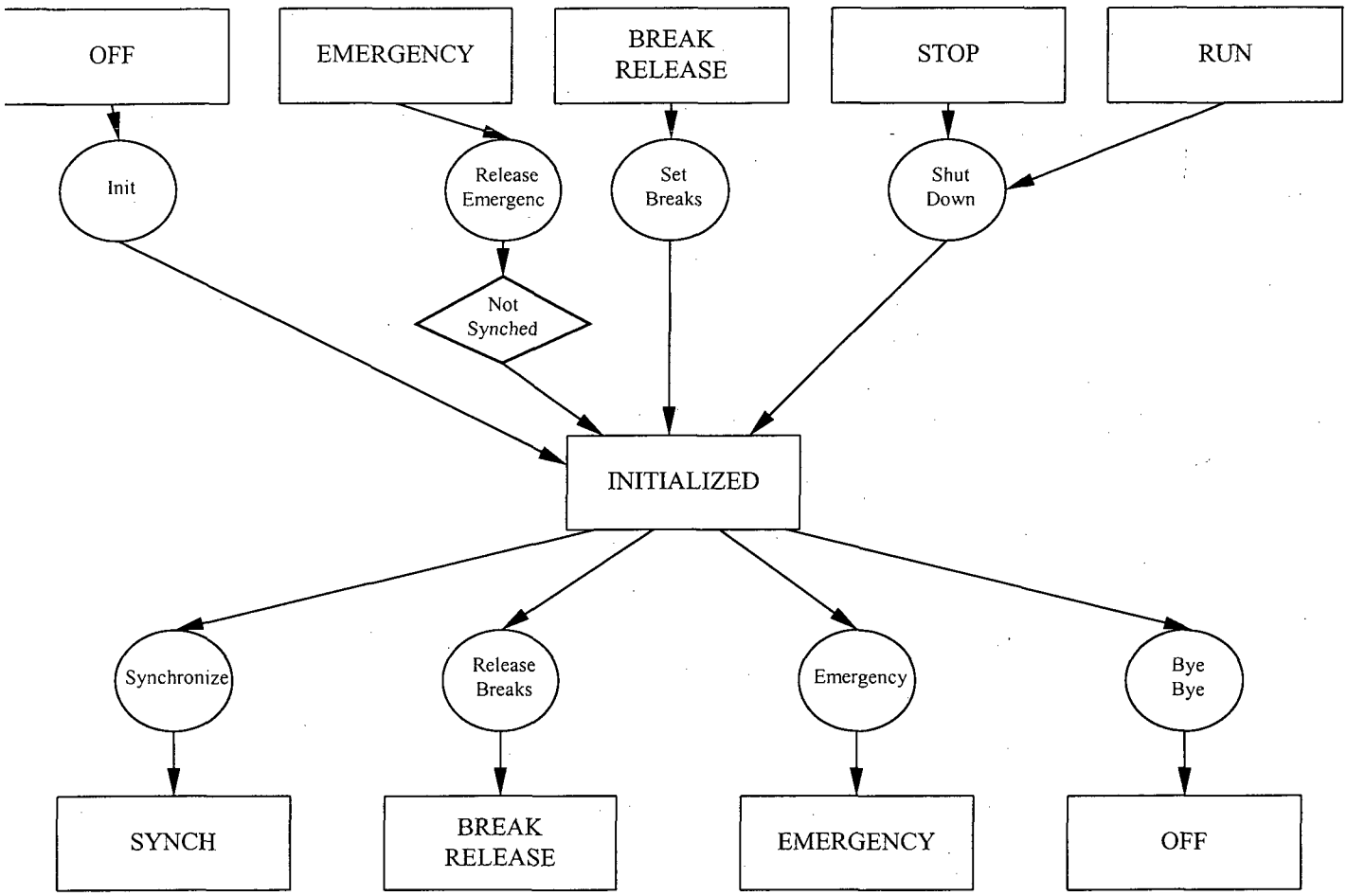


FIGURA 3.3 - O estado Initialized

Estado *Emergency* - este estado surge quando é acionada a botoeira de emergência ou quando algum módulo detecta algum situação fora do normal e aciona o comando emergência. É possível atingir este estado a partir de qualquer outro estado. Cuidado especial deve ser tomado para sair deste estado, pois o motivo que o gerou deve ser identificado e resolvido. Se o problema não tiver sido resolvido devemos utilizar o comando *ByeBye* que desliga o robô. Ao utilizar o comando ReleaseEmergency o robô pode continuar normalmente seus movimentos mesmo que

o problema que gerou este estado não tenha sido identificado e resolvido. Isto pode gerar um novo estado de emergência ou até provocar danos mais graves. Neste estado os amplificadores estão desabilitados e o robô está freiado. Ao sair deste estado com o comando *Release Emergency* o robô assume o estado *Stop* se já estiver sincronizado ou o estado *Initialized* se ainda não estiver sincronizado. Nestes dois casos, para continuar a trabalhar com o robô é necessário apertar o botão de Power Up, que é automaticamente desacionado quando entramos no estado *Emergency*.

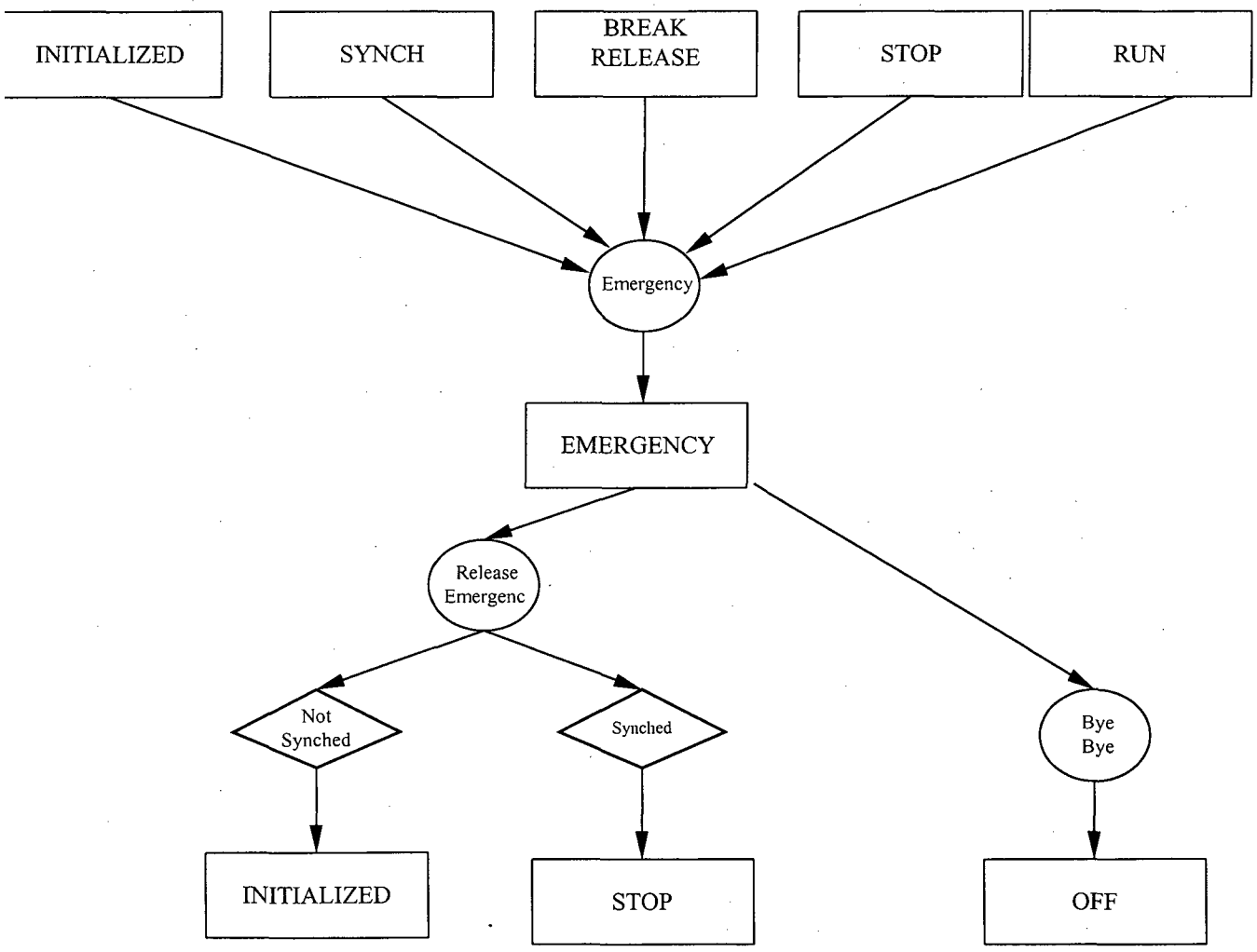


FIGURA 3.4 O estado Emergency

Estado *Synch* - estado transitório de sincronização. Neste estado o robô está em processo de sincronização, movimentando seus elos até os sensores de fim de curso. Se o processo transcorrer normalmente, o robô passa automaticamente ao estado *Stop*. Caso algum erro seja detectado pelo programa de sincronização ou se a botoeira de emergência for acionada, o robô passa para o estado *Emergency*.

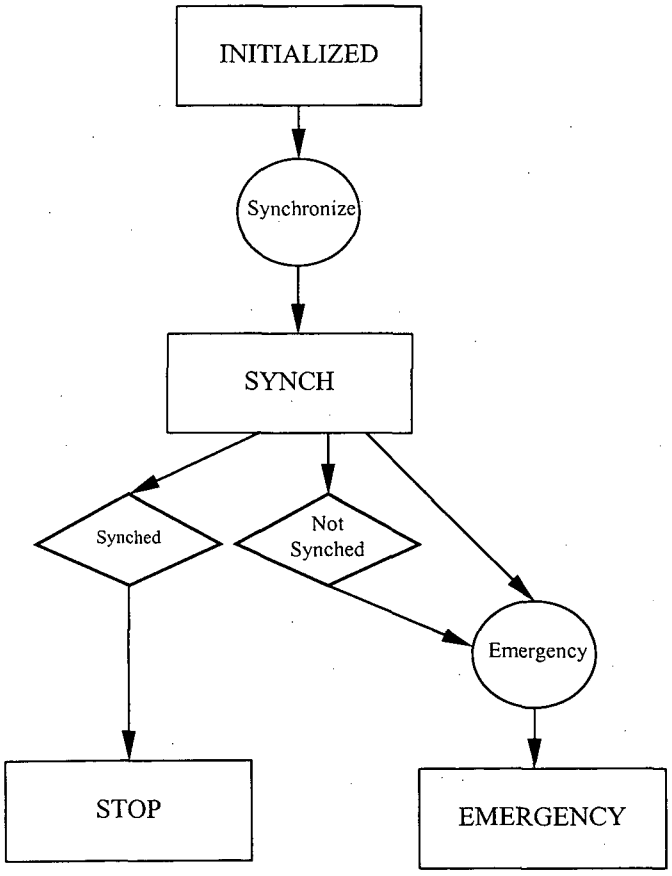


FIGURA 3.5 - O estado Synch

Estado *Stop* - é um estado de espera do robô, bastando acionar o comando de *Release Stop* para colocar o robô no estado *Run*. O robô somente pode assumir este estado quando estiver sincronizado. Normalmente utilizamos este estado quando fazemos uma pausa na movimentação do robô. Neste estado não existe risco de acidente pois os amplificadores estão desabilitados e o robô freiado.

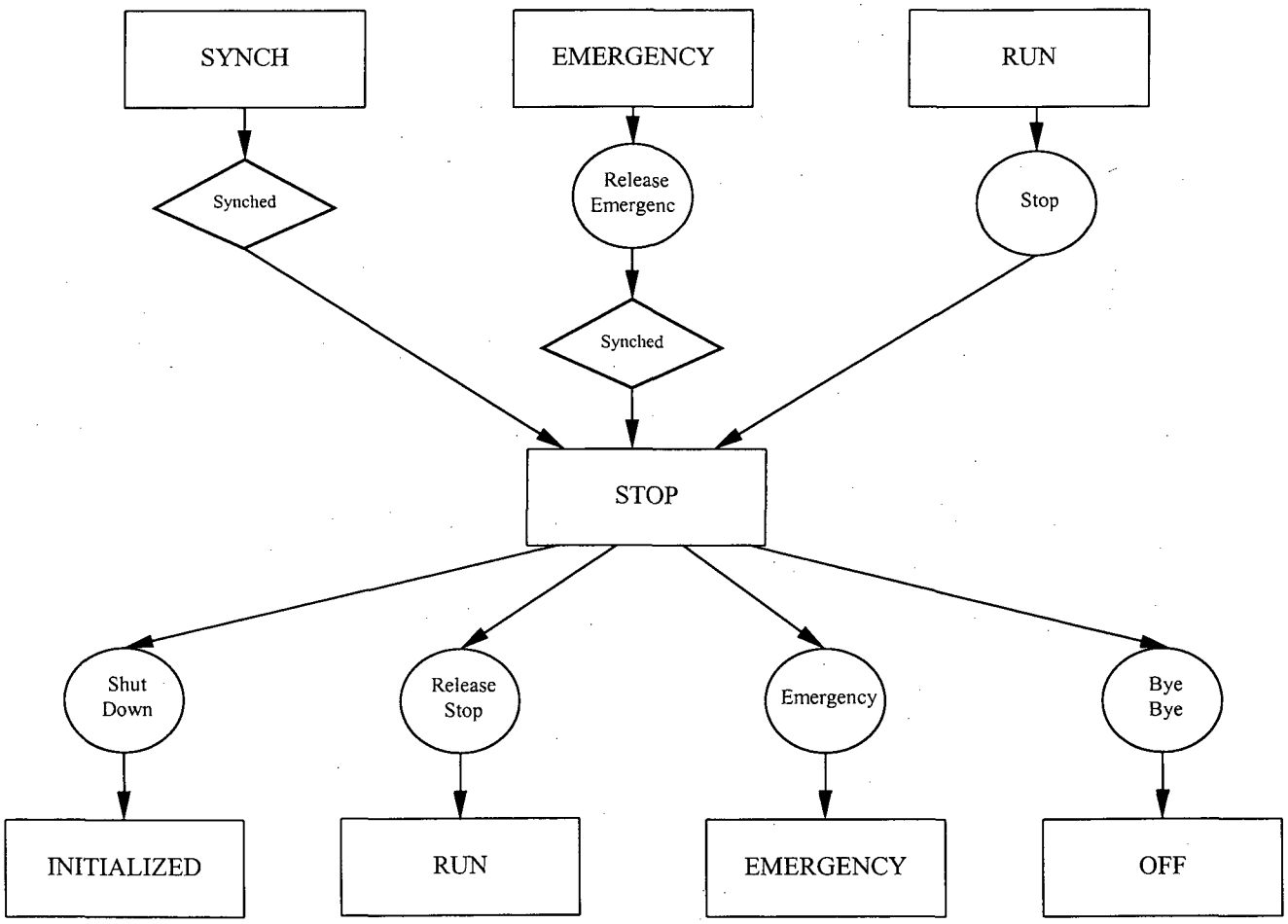


FIGURA 3.6 - O estado Stop

Estado *Run* - à exceção do processo de sincronização, todos os movimentos do robô são executados com o robô neste estado. Como neste estado os amplificadores estão habilitados e os freios estão soltos existe possibilidade de acidente. Para reduzir esta possibilidade devemos evitar de entrar no interior da região que delimita o espaço de trabalho do robô. É importante também que sempre que terminarmos de movimentar o robô ele não permaneça neste estado. O mais simples é passar ao estado *Stop*, de onde podemos retornar ao estado *Run* sem maiores problemas quando quisermos voltar a movimentá-lo.

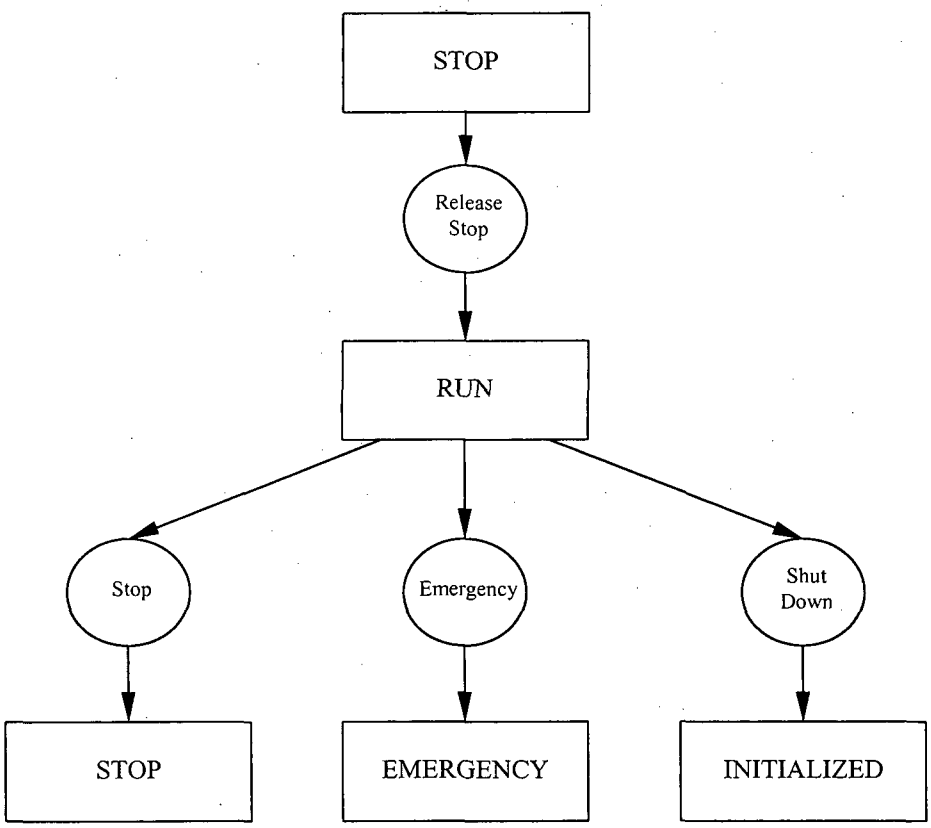


FIGURA 3.7 - O estado Run

Estado *BreakRelease* - neste estado o robô pode ser posicionado manualmente sem riscos de acidentes pois apesar dos freios estarem soltos, os amplificadores estão desabilitados. Não são muitas as situações em que utilizamos este estado. Talvez para teste dos sensores de final de curso ou quando quisermos aproximar ou afastar o robô de objetos ou superfícies. No estado *BreakRelease* o robô não está sincronizado.

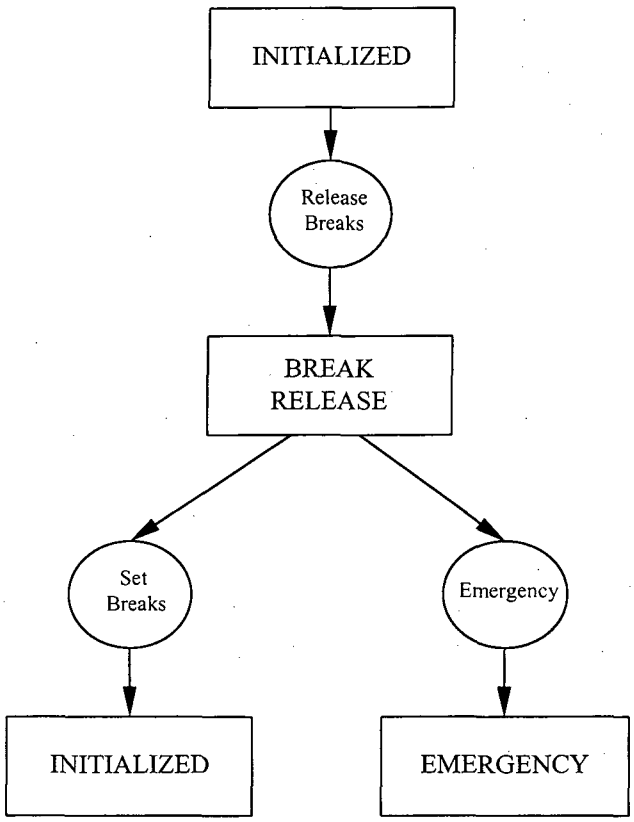


FIGURA 3.8 - O estado Break Release

Estado *OFF* - neste estado o robô está desligado. Isto não significa que o armário de controle esteja desligado, apenas que não existem nenhum processo relativo ao robô rodando no microprocessador (Target). Os processos relativos ao sistema XOberon continuam em funcionamento no microprocessador. A única forma de sairmos do estado *OFF* é acionando o comando *Init*.

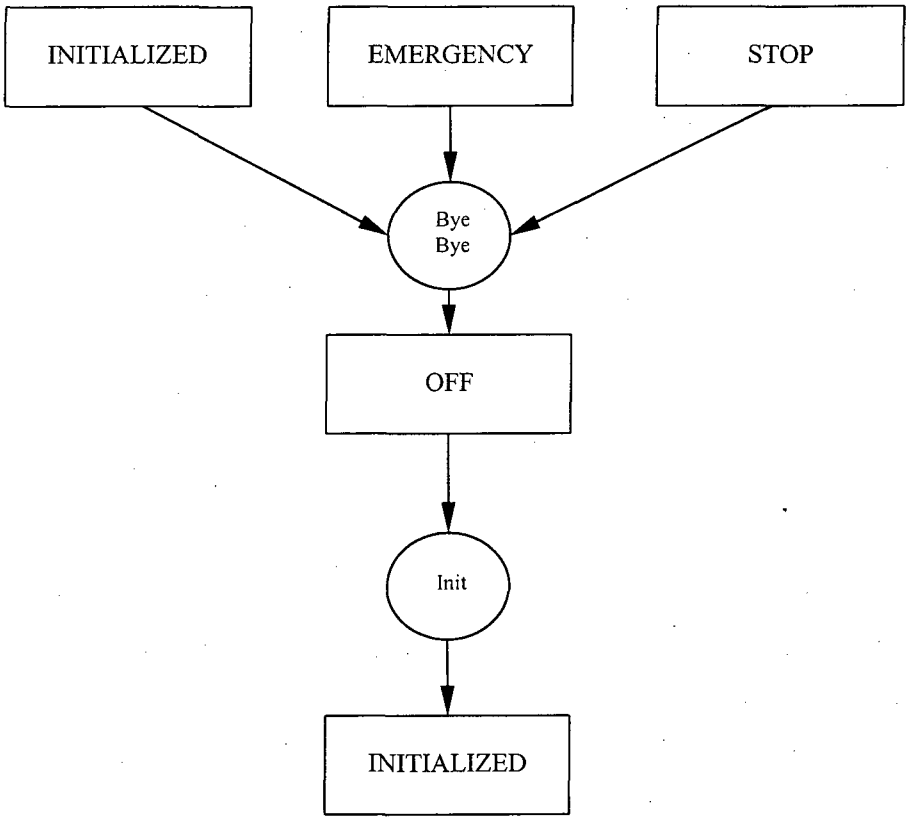


FIGURA 3.9 - O estado OFF

A tarefa *MAIN* associada ao *MainEvent* de nome *mainevent* do módulo *Main3* faz o controle dos estados do robô. Nesta tarefa é verificado se existe um novo comando para troca de estados e se este comando é válido para o estado atual em que se encontra o robô. Se o comando for válido o estado é alterado e todas as alterações relativas a este novo estado são executadas. Por exemplo, para passarmos do estado *Run* para o estado *Stop* precisamos apenas acionar o comando *ReleaseStop*. A tarefa *MAIN* se encarregará de acionar os comandos para desabilitar os amplificadores e frear os motores. O estado dos *flags* também é monitorado por esta tarefa, que os

modifica quando pertinente. Esta tarefa é instalada através do comando *Init* e desinstalada através do comando *ByeBye*.

Além desta tarefa, o módulo *Main3* possui a tarefa *Controller*, associada ao *Every Event* de nome *ctrlEvent*. Esta tarefa também é instalada através do comando *Init* e possui um clock de um milisegundo. A função desta tarefa é fornecer os valores de controle para as juntas. Quando a tarefa *MAIN* é terminada, como conseqüência, a tarefa *Controller* é desinstalada.

As principais tarefas e procedimentos do módulo Main3 são apresentados no apêndice C. Para quem tiver interesse, a versão completa do módulo *Main3* pode ser encontrada em [7].

O módulo *Main3* sem dúvida é o mais importante na movimentação e controle do robô Inter. Qualquer modificação feita neste módulo deve ser cuidadosamente avaliada.

3.6 O Módulo Bahnplaner

Este módulo é responsável pela geração de trajetórias do robô. Calcula a posição e velocidade desejadas, dada a posição inicial, posição final, velocidade e aceleração do movimento. Funciona no espaço cartesiano ou no espaço de juntas. A movimentação do robô somente pode ser feita ponto a ponto. Não existe geração de trajetórias curvas ou de trajetórias ponto a ponto, passando por outros pontos intermediários.

A estratégia utilizada por este módulo é dividir a trajetória em dois ou três segmentos conforme a situação apresentada. Se não for possível atingir a velocidade desejada com a aceleração especificada durante a movimentação, a trajetória é dividida em 2 segmentos. Se for possível atingir a velocidade máxima, a trajetória é dividida em 3 segmentos. A figura abaixo apresenta ao gráficos de posição, velocidade e aceleração para estes dois casos. *Posf* representa a posição final, *tf* representa o tempo final, *Vdes* representa a velocidade desejada, *Acdes* representa a aceleração desejada e *Valc* representa a velocidade máxima atingida durante o movimento. No caso da divisão da trajetória em dois segmentos, a *Vdes* nunca é alcançada.

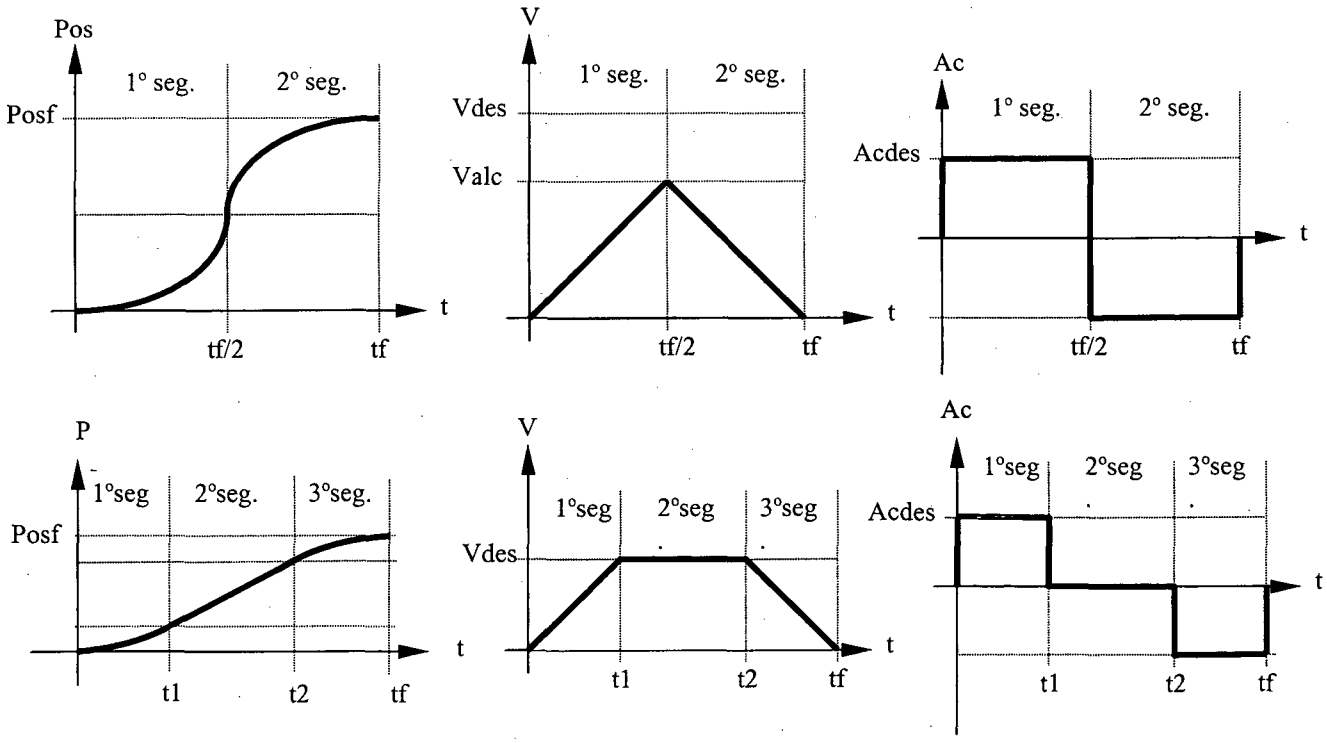


FIGURA 3.10 - Divisão da trajetória em segmentos

O tempo de execução da trajetória é determinado pela velocidade e aceleração desejadas.

Após dividir a trajetória em 2 ou 3 segmentos, é feito o cálculo da posição, velocidade e aceleração na extremidade de cada segmento. Todos os pontos intermediários são calculados em tempo real, durante a execução do movimento.

Além de definir o espaço em que queremos trabalhar, podemos determinar se queremos fazer uma movimentação relativa ou absoluta. Na movimentação relativa o robô se movimenta a partir do ponto em que se encontra nas distâncias especificadas. Na movimentação absoluta, o robô se movimenta do ponto em que se encontra para o ponto indicado. Quando trabalhamos no espaço de juntas devemos fornecer dados relativos à cada junta ao chamar os comandos. Já no espaço cartesiano devemos fornecer dados relativos a cada uma das direções cartesianas.

Os comandos para movimentação do robô são:

PROCEDURE MoveRelSingleJoint*(axis: LONGINT; dq, speed, acc: REAL): BOOLEAN;

Movimenta a junta indicada (axis) na distância *dq* em relação à posição atual, com a velocidade *speed* e a aceleração *acc*.

PROCEDURE MoveRelAllJoint*(dq0, dq1, dq2, dq3, speed, acc: REAL): BOOLEAN;

Movimenta as juntas 0, 1, 2 e 3 nas distâncias *dq0*, *dq1*,*dq2* e *dq3* respectivamente, relativamente à posição em que se encontram, com a velocidade *speed* e a aceleração *acc*. Não é possível especificar uma velocidade para cada junta pois isto traria como conseqüência que as juntas poderiam terminar seus movimentos em tempos diferentes. A solução encontrada foi calcular um vetor deslocamento *d* cujos componentes são as distâncias de deslocamento de cada junta. Este vetor deslocamento é calculado através da fórmula:

$$d = \sqrt{dq0^2 + dq1^2 + dq2^2 + dq3^2}$$

A velocidade especificada ao chamar o procedimento é a velocidade deste vetor deslocamento. Para calcular a velocidade de cada junta esta velocidade é decomposta em função do deslocamento a ser realizado em cada junta, através da fórmula exemplificada a seguir para a junta 1:

$$speed1 = \frac{speed}{d} \cdot dq1$$

onde:

*speed*1 é a velocidade da junta 1.

O grande problema disto é que trabalhamos simultaneamente com duas unidades distintas pois nas juntas 0, 1 e 3 usamos radianos e rad/s enquanto que na junta 2 utilizamos metros e m/s. Como os valores absolutos em radianos são normalmente maiores que aqueles fornecidos em metros, temos de cuidar para não especificar valores muito altos para velocidade e aceleração que não possam ser atingidos pela junta 2. Em contrapartida, ao utilizar valores que possam ser atingidos pela junta 2, estamos reduzindo a velocidade das juntas 0, 1 e 3 desnecessariamente, tornando o robô mais lento. Se durante a movimentação alguma das juntas ultrapassar a velocidade máxima especificada, o movimento do robô é interrompido.

PROCEDURE MoveRelXYZ*(dx, dy, dz, speed, acc: REAL): BOOLEAN;

Movimenta o robô no espaço cartesiano nas distâncias *dx*, *dy* e *dz*, relativamente à posição que se encontra, com a velocidade *speed* e a aceleração *acc*. As unidades utilizadas são metros, m/s e m/s^2. Novamente a velocidade *speed* é decomposta em cada direção, proporcionalmente ao comprimento *dx*, *dy* e *dz*

PROCEDURE MoveRelPhi*(dphi, speed, acc: REAL): BOOLEAN;

Movimenta o robô no espaço cartesiano, provocando o giro *dphi* em relação à posição que se encontra, com a velocidade *speed* e a aceleração *acc*. As unidades utilizadas são radianos, rad/s e rad/s^2.

PROCEDURE MoveRelXYZPhi*(dx, dy, dz, dphi, speed, acc: REAL): BOOLEAN;

Movimenta o robô no espaço cartesiano na distâncias *dx*, *dy* e *dz* e provocando o giro *dphi* em relação a sua posição atual, com a velocidade *speed* e a aceleração *acc*. Neste caso, apesar de também trabalharmos com unidade distintas, as conseqüências são outras. Como normalmente o valor absoluto de *dphi* é maior que os outros, ao decompormos a velocidade, a maior parcela de velocidade ficará para executar o giro *dphi*, tornando os outros movimentos mais lentos. Para aumentar a velocidade do robô

bastaria aumentar a velocidade especificada. O problema que surge é que o limite de velocidade para este tipo de movimento é completamente distinto daquele utilizado quando não existe o giro *dphi* ou quando *dphi* é igual a zero. Fica difícil estabelecer limites de velocidade máxima para este tipo de movimento e o que normalmente acontece é que valores conservativos são tomados como referência e a velocidade do robô fica bem abaixo da máxima possível.

PROCEDURE MoveAbsSingleJoint*(axis: LONGINT; q, speed, acc: REAL): BOOLEAN;

Movimenta a junta especificada (axis) para a posição absoluta *q*, com a velocidade *speed* e a aceleração *acc*.

PROCEDURE MoveAbsAllJoint*(q0, q1, q2, q3, speed, acc: REAL): BOOLEAN;

Movimenta as juntas 0, 1, 2 e 3 respectivamente para as posições absolutas *q0*, *q1*, *q2* e *q3*, com velocidade *speed* e aceleração *acc*. As mesmas observações feitas para o procedimento *MoveRelAllJoint* são válidas também para este procedimento.

PROCEDURE MoveAbsXYZ*(x, y, z, speed, acc: REAL): BOOLEAN;

Movimenta o robô no espaço cartesiano, do ponto que se encontra para o ponto definido pelas coordenadas *x*,*y* e *z*, com a velocidade *speed* e a aceleração *acc*.

PROCEDURE MoveAbsPhi*(phi, speed, acc: REAL): BOOLEAN;

Movimenta o robô no espaço cartesiano, posicionando o efetuador final na orientação absoluta *phi*, com velocidade *speed* e aceleração *acc*.

PROCEDURE MoveAbsXYZPhi*(x, y, z, phi, speed, acc: REAL): BOOLEAN;

Movimenta o robô no espaço cartesiano para o ponto *x*, *y*, *z* e orientação *phi*, com velocidade *speed* e aceleração *acc*. As mesmas observações feitas para o procedimento *MoveRelXYZPhi* são válidas para este caso.

Para executar qualquer um destes movimentos nos utilizamos de um módulo auxiliar chamado *BPTest.Mod*. Os procedimentos deste módulo possuem os mesmos nomes e o que eles fazem é ler através de um Scanner os valores desejados descritos na linha de chamada e enviar estes dados para o procedimento correspondente do módulo *Bahnplaner*. Por exemplo, se quisermos movimentar o robô no espaço

cartesiano para o ponto X=0.3m, Y=0.26m, Z=0.35m, com velocidade speed=0.2 m/s e aceleração acc=1.0 m/s^2, devemos escrever:

XSystem.Call BPTest.MoveAbsXYZ 0.3 0.26 0.35 0.2 1.0~

Existe um arquivo chamado *BPTest.Tool* onde já estão à disposição vários comandos de movimentação do robô. O usuário pode adicionar à este arquivo novos comandos com os movimentos que julgar importante ou com aqueles movimentos que serão repetidos diversas vezes.

Os outros procedimentos importantes do módulo Bahnplaner são:

```
PROCEDURE CalcTrack*(ctrlEvent : XOK.Event);

PROCEDURE PrepareIm(abs: BOOLEAN; Vector: GD.Vector4; speedIn, accIn:
LONGREAL):BOOLEAN;

PROCEDURE Move(absrel: BOOLEAN; d: GD.Vector4; speed, acc: REAL): BOOLEAN;
```

O procedimento *Move* gerencia toda a execução do movimento, chamando o procedimento *PrepareIm*, instalando a tarefa *CalcTrack* e obtendo a autorização para movimentação do robô junto ao módulo *Main3*. É acionado automaticamente toda vez que um dos comandos de movimentação for executado.

O procedimento *PrepareIm* prepara a execução da trajetória, dividindo-a e segmentos e calculando posições, velocidades e acelerações no início e no final de cada segmento.

CalcTrack é uma tarefa associada à variável *ctrlEvent* do tipo *Every Event*. Esta tarefa é instalada pelo procedimento *Move* e possui clock de 5 milisegundos. Calcula os valores desejados de posição, velocidade e aceleração ao longo de toda trajetória. Todos os cálculos são realizados em tempo real e novos dados são gerados a cada clock. Estes valores desejados são enviados para a tarefa *Controller* do módulo *Main3* que realiza o controle.

Os procedimentos apresentados a seguir reduzem a quantidade de cálculos que necessitam ser realizados em tempo real. O número de segmentos e a posição, velocidade e aceleração no início e no fim de cada segmento são calculados offline e armazenados em um arquivo. Estes procedimentos lêem este arquivo e utilizam os

dados previamente calculados. Os pontos no interior de cada segmento ainda tem de ser calculados online. Atualmente estes procedimentos não estão sendo utilizados, pois o microprocessador tem velocidade suficiente para executar sem problemas todos os cálculos de forma online.

```
PROCEDURE ReadPath*(name: ARRAY OF CHAR; VAR p: PD.Planner);
PROCEDURE MovePathCart*(s: ARRAY OF CHAR): BOOLEAN;
PROCEDURE MovePathRob*(s: ARRAY OF CHAR): BOOLEAN;
```

Ao utilizar este módulo na geração de trajetórias, devemos estar cientes de suas limitações e dos cuidados a serem tomados ao especificar uma trajetória.

3.7 O Módulo StateCtrl

Neste módulo está o algoritmo de controle de posição do robô. Apesar da formulação um pouco complicada, veremos que o controlador é do tipo PD, sem nenhum tipo de compensação dinâmica.

Este módulo foi escrito originalmente para um robô cujo sinal de controle era uma velocidade e não um momento como é feito atualmente. Isto implica na utilização de algoritmos de controle completamente diferentes. O algoritmo de controle original foi mantido como comentário neste módulo pois se algum dia for necessário alterar o modo de controle de momento para velocidade, já teremos um algoritmo pronto.

Como o algoritmo atual foi escrito a partir de modificações introduzidas neste algoritmo original, muitas variáveis configuradas não estão sendo efetivamente utilizadas.

Existe um algoritmo de nome *AlgoTorq* do tipo PID neste módulo, o qual ainda não foi testado.

O algoritmo utilizado no momento possui o nome *Algo*. Após a descrição da função básica de cada procedimento, apresentaremos com maiores detalhes este algoritmo.

```
PROCEDURE (c : StateCtrl) Assign* (o : Base.Object; name : ARRAY OF CHAR) :
BOOLEAN;
PROCEDURE AttrMsg(obj: StateCtrl; VAR M: O.AttrMsg);
PROCEDURE Handler*(obj: O.Object; VAR M: O.ObjMsg);
```

```
PROCEDURE NewStateCtrl*;
PROCEDURE DefStateCtrl*;
```

Estes 5 procedimentos configuram os objetos do tipo *StateCtrl*, colocando-os na biblioteca.

```
PROCEDURE (c : StateCtrl) ChangePara*(VAR stateCtrlPara : StateCtrlPara);
```

Este procedimento altera os ganhos do controlador.

```
PROCEDURE (c : StateCtrl) ChangeFilterPara*(VAR  filterPara: FilterBSnO.FilterPara);
```

O filtro somente era utilizado com o algoritmo original, não estando mais em uso. Este procedimento alterava os coeficientes do filtro.

```
(*
PROCEDURE (c : StateCtrl) Algo*(istpos, sollpos, istspeed, sollspeed : REAL) : REAL;
END Algo;
*)
```

É o algoritmo original. Permanece no interior do módulo sob forma de comentário. Possui o mesmo nome do algoritmo atualmente utilizado.

```
PROCEDURE (c : StateCtrl) AlgoTorq*(istpos, sollpos, istspeed, sollspeed : REAL) : REAL;
```

Algoritmo do tipo PID. Nunca foi testado, e portanto os seus ganhos não estão ajustados.

```
PROCEDURE (c : StateCtrl) Algo*(istpos, sollpos, istspeed, sollspeed : REAL) : REAL;
VAR
        posdiff, poskorr,deltaspeed, forceout : REAL; p : StateCtrlPara;
BEGIN
        p:= c.stateCtrlPara;
        sollspeed:= RF.RampFilter(sollspeed, c.rs);
        sollpos:= RF.RampFilter(sollpos, c.rp);
        poskorr:=(sollspeed + istspeed)*p.systemdelayDiv2;
        posdiff:= sollpos - istpos - poskorr;
        deltaspeed:= sollspeed-istspeed;

        (* I-Part *)
        p.integ := p.integ+sollpos-istpos;
        IF ((p.integ>0)&(p.integ>p.iLimit)) THEN
                p.integ:= p.iLimit
        ELSIF ((p.integ<0)&(p.integ<-p.iLimit)) THEN
                p.integ:= -p.iLimit
        END;
```

```
        forceout:= (posdiff*p.kPos+deltaspeed+p.i*p.integ)*p.ineDivclock;
        IF c.bsFilter#NIL THEN forceout:= c.bsFilter.Algo(forceout) END;
        RETURN forceout
END Algo;
```

Este é o algoritmo de controle atualmente utilizado. Os valores configurados através do arquivo *ROBAlllBoot.Config* são:

- junta 0 - kPos=400.0, kVel=1.0, systemdelay=0.0, i=0.0, iLimit=0.0, inertia=0.1, clock=0.001, ineDivclock=100.
- junta 1 - kPos=400.0, kVel=1.0, systemdelay=0.0, i=0.0, iLimit=0.0, inertia=0.07, clock=0.001, ineDivclock=100.
- junta 2 - kPos=400.0, kVel=1.0, systemdelay=0.0, i=0.0, iLimit=0.0, inertia=0.0001, clock=0.001, ineDivclock=100.
- junta 3 - kPos=400.0, kVel=1.0, systemdelay=0.0, i=0.0, iLimit=0.0, inertia=0.001, clock=0.001, ineDivclock=100.

Algumas variáveis do Algo original foram inicialmente utilizadas e depois zeradas por não apresentarem resultados eficientes.

No processo de configuração, estas variáveis que definem os ganhos do controlador são campos do objeto *StateCtrl.StateCtrlPara*. No método *Algo*, como a variável *c* é do tipo *StateCtrl*, para nos referenciarmos a estas variáveis, deveríamos escrever *c.StateCtrlPara*. No entanto, na primeira linha deste módulo escreveu-se:

```
p : = c.stateCtrlPara;
```

com o único intuito de simplificar a escrita das variáveis. Para identificar a variável *i* por exemplo, podemos escrever apenas *p.i* ao invés de *c.stateCtrlPara.i*.

E como esta variável *p.i* é igual a zero para todas as juntas, a parcela integral é anulada e o algoritmo fica mais simples.

A variável *systemdelay* é calculada por *systemdelayDiv2*. Como *systemdelay* vale zero para todas as juntas, *systemdelayDiv2* também vale e como conseqüência, *poskorr* também é igual a zero.

Lembrando que o filtro não está sendo utilizado, vamos rescrever este procedimento já considerando estas simplificações.

```
PROCEDURE (c : StateCtrl) Algo*(istpos, sollpos, istspeed, sollspeed : REAL) : REAL;
VAR
        posdiff,deltaspeed, forceout : REAL; p : StateCtrlPara;
BEGIN
        p:= c.stateCtrlPara;
        sollspeed:= RF.RampFilter(sollspeed, c.rs);
        sollpos:= RF.RampFilter(sollpos, c.rp);
        posdiff:= sollpos - istpos;
        deltaspeed:= sollspeed-istspeed;
        forceout:= (posdiff*p.kPos*p.ineDivclock+deltaspeed*p.ineDivclock);

        RETURN forceout
END Algo;
```

Ao chamar este procedimento, devemos informar as posições e velocidades atuais e desejadas. Como resposta obtemos um torque de controle.

Este procedimento é na verdade um método associado ao tipo *StateCtrl*. Na configuração são criados os objetos *StateCtrl0, StateCtrl1, StateCtrl2* e *StateCtrl3*. Estes objetos são buscados da biblioteca pelo módulo *Main3* e assumem respectivamente os nomes *stateCtrl[0], stateCtrl[1], stateCtrl[2]* e *stateCtrl[3]*. Logo, cada um destes objetos possui este método associado e no momento que o chama, fornece automaticamente também o valor dos ganhos configurados para cada um deles. Para calcular os valores de controle para as 4 juntas, este método é chamado 4 vezes a cada milisegundo, uma vez para cada junta.

A segunda e terceira linhas utilizam o módulo *RampFilterLib* para escalonar os valores de velocidade e posição desejados. Como o clock do controlador é 1 milisegundo e o clock do gerador de trajetórias é 5 milisegundos, o controlador trabalharia com o mesmo valor desejado durante 5 ciclos. O módulo *RamFilterLib* possibilita através deste escalonamento que a cada novo ciclo do controlador exista um novo valor desejado. Isto tende a tornar o movimento do robô mais suave.

Na quarta linha é calculado o erro de posição e na quinta linha é calculado o erro de velocidade.

A sexta linha apresenta o algoritmo de controle propriamente dito:

```
forceout:= (posdiff*p.kPos*p.ineDivclock+deltaspeed*p.ineDivclock);
```

Comparemos este algoritmo de controle com o controlador do tipo PD apresentado a seguir:

$$Tau= (qd -q)*kp + (vd -v)*kv,$$

onde:

- Tau é o torque de controle;
- qd é a posição desejada;
- q é a posição atual;
- vd é a velocidade desejada;
- v é a velocidade atual;
- kp é o ganho proporcional;
- kv é o ganho derivativo.

Fica bastante fácil estabelecer a relação entre estes dois algoritmos:

- forceout=Tau;
- posdiff=qd-q;
- p.kPos*p.ineDivclock=kp;
- deltaspeed= vd-v;
- p.ineDivclock= kv.

Logo, o algoritmo de controle nada mais é do que um controlador PD.

Os ganhos foram ajustados individualmente para cada junta de forma prática. Na primeira etapa aumentou-se gradativamente o valor do ganho *kPos* até o robô começar a vibrar. Foi ajustado como ganho um valor 40% menor do que o valor do ganho no momento que iniciou a vibração.

Na segunda etapa repetiu-se o procedimento para o ganho *ineDivclok*. No final destes processos feitos individualmente para cada uma das juntas, foram obtidos os valores dos ganhos anteriormente descritos.

O controlador do tipo PD reconhecidamente não é o ideal para controle de posição. E a determinação dos ganhos para este controlador foi feita de forma

totalmente prática, sem um embasamento científico profundo. Apesar desses fatores, o robô apresenta, em linhas gerais, um desempenho razoável.

Logicamente este controlador pode e deve ser melhorado. O controlador atual não está escrito de forma clara e possui muitas linhas que não apresentam mais utilidade. No entanto, não faz parte do escopo deste trabalho a implementação e análise de diferentes tipos de controladores. Veremos a seguir como seria o processo de implementação de um novo controlador, de forma a direcionar e facilitar trabalhos futuros que venham a ser realizados nesta área de controle.

A maneira mais simples de implementar um novo algoritmo de controle é escrever no interior do módulo *StateCtrl* um novo método utilizando o mesmo nome Algo e colocar o método *Algo* atualmente em uso como comentário. Ao também utilizar este mesmo nome para o novo controlador, não há necessidade de alterar os demais módulos, pois estes também fazem referência a este nome.

No interior deste novo *Algo* podemos realizar todos os cálculos necessários para implementar o novo controlador. Os dados fornecidos a este *Algo* continuam sendo as posições e velocidades atuais e desejadas. Se necessitarmos a aceleração atual, podemos também calculá-la no interior deste módulo ou então em algum outro procedimento auxiliar.

Existem outras maneiras de implementar um novo controlador, as quais provavelmente necessitem de modificações em outros módulos existentes. Estas modificações mais complexas devem ser feitas com bastante cuidado pois podem comprometer todo o funcionamento do robô.

Na medida do possível devemos evitar de modificar o módulo *Main3* que é a base do funcionamento do robô. Se instalarmos um novo controlador sem alterar este módulo e após isto o robô não apresentar mais um funcionamento adequado, fica fácil de identificar o erro pois com certeza estará no novo módulo implementado e não no módulo *Main3*. É conveniente também guardar uma cópia da versão antiga do módulo a ser alterado, possibilitando retornar ao estado de funcionamento anterior, caso o novo controlador apresente algum problema.

Capítulo 4

CONTROLE HÍBRIDO

4.1 Introdução

Existem muitas tarefas em que as estratégias de controle de posição não apresentam resultados satisfatórios. Para montagem e polimento de peças, por exemplo, é mais adequado utilizar técnicas que controlem as forças de interação entre o manipulador e o ambiente. Uma das técnicas de controle de força utilizada é o controle híbrido.

A idéia básica do controle híbrido é determinar as direções em que se deseja fazer controle de força e as direções em que se deseja fazer o controle de posição. O controle híbrido permite que seja utilizada uma lei de controle distinta para cada grau de liberdade do robô. No caso do manipulador com configuração SCARA, seria possível, por exemplo, definir leis de controle diferentes para as direções x, y e z do espaço cartesiano e para a orientação phi.

O objetivo deste capítulo é apresentar o módulo de controle híbrido implementado no robô Inter, explicando seu funcionamento, as simplificações feitas e suas restrições. Como este módulo não está em seu estágio final de desenvolvimento, no final do capítulo são apresentadas algumas sugestões para aqueles interessados em continuar a trabalhar com o controle híbrido.

4.2 Sistemas de Coordenadas

O controle híbrido trabalha com diversos sistemas de coordenadas também chamados de espaços. São freqüentes as transformações de posição, velocidade e aceleração entre estes sistemas. Portanto, é necessário que cada um destes sistemas seja bem definido.

Sistema de coordenadas do robô ou operacional ou espaço operacional - sistema de coordenadas cartesiano, fixo a base do robô. Este sistema é utilizado para

definir a posição absoluta x_0, y_0, z_0 e phi_0 do robô. Para identificar qualquer variável relacionada a este sistema utilizaremos 'o' subscrito.

Sistema de coordenadas da tarefa ou espaço da tarefa - sistema de coordenadas cartesiano criado para definir uma tarefa a ser executada pelo robô, tendo como referência o espaço operacional. Como o robô Inter possui 4 graus de liberdade, a definição deste sistema restringe-se a deslocamentos de origem e a rotações θ_T ao redor do eixo z_0. Para identificar qualquer variável relacionada a este sistema utilizaremos 'T' subscrito.

Sistema de coordenadas do efetuador final ou espaço do efetuador final - sistema de coordenada cartesiano fixo ao efetuador final. Neste sistema será feita a leitura de forças e momentos. Para identificar qualquer variável relacionada a este sistema utilizaremos 'E' subscrito.

Sistema de coordenadas das juntas ou espaço das juntas - por não se tratar exatamente de um sistema de coordenadas, pois para cada junta é definido um sistema diferente, utilizaremos preferencialmente palavra espaço ao invés de sistema. Este espaço define de forma única a configuração do manipulador através do posicionamento individual de cada uma de suas juntas. Para identificar as posições das juntas 0, 1, 2 e 3 utilizaremos respectivamente as variáveis θ_0, θ_1, d_2 e θ_3. Para velocidades utilizaremos $\dot{\theta}_0$, $\dot{\theta}_1$, $\dot{d}_2$ e $\dot{\theta}_3$, e para forças ou momentos τ_0, τ_1, F_2 e τ_3. O vetor q será utilizado para representar a posição de todas as juntas e o vetor τ os momentos.

4.3 Técnicas de Controle

O controle híbrido pode ser realizado no espaço operacional ou no espaço da tarefa. Durante o desenvolvimento do módulo de controle híbrido, verifiquei maior facilidade em executar este controle no espaço da tarefa. Portanto, nos concentraremos em descrever este caso. Esta técnica de controle normalmente utiliza o cancelamento dinâmico e o desacoplamento. Este desacoplamento permite que sejam utilizados controladores distintos para cada uma das direções do espaço de tarefas. Mostraremos os passos que devem ser seguidos para chegar nesta situação descrita.

Iniciaremos revendo a bem conhecida equação da dinâmica do robô em contato com o meio.

$$H(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = \tau - J_T^{\ T} f_T \tag{4.1}$$

onde:

$H(q)$ é a matriz de inércia do manipulador;

$C(q,\dot{q})$ é a matriz dos torques centrípedos e de coriolis;

$G(q)$ é a matriz dos torques gravitacionais;

τ é o vetor que representa os torques aplicados nas juntas;

J_T é o jacobiano da tarefa;

q, $\dot{q}$ e $\ddot{q}$ são os vetores que representam as posições, velocidades e acelerações das juntas respectivamente;

f_T é o vetor que representa as forças aplicadas no manipulador no sistema da tarefa.

Para transformar velocidades de um espaço ou sistema para outro utilizamos Jacobianos. Na transformação de velocidades do espaço de juntas para o espaço da tarefa utilizamos a relação:

$$\dot{p}_T = J_T \dot{q} \tag{4.2}$$

onde:

$\dot{p}_T$ é o vetor que determina as velocidades no espaço da tarefa;

J_T é o jacobiano da tarefa;

$\dot{q}$ é o vetor que define as velocidades das juntas do manipulador.

Para determinar a relação entre as acelerações entre estes dois sistemas, devemos diferenciar esta equação, chegando ao seguinte resultado:

$$\ddot{p}_T = J_T \ddot{q} + \dot{J}_T \dot{q} \tag{4.3}$$

Isolando $\ddot{q}$ temos:

$$\ddot{q} = J_T^{-1}(\ddot{p}_T - \dot{J}_T \dot{q}) \tag{4.4}$$

onde:

J_T^{-1} é o inverso do jacobiano da tarefa;

$\dot{J}_T$ é a derivada do jacobiano da tarefa;

$\ddot{p}_T$ é o vetor aceleração no espaço da tarefa;

$\ddot{q}$ é o vetor aceleração das juntas.

Substituindo a eq.(4.4) na eq.(4.1) obtemos:

$$H(q)[J_T^{-1}(\ddot{p}_T - \dot{J}_T \dot{q})] + C(q,\dot{q})\dot{q} + G(q) = \tau - J_T^T f_T \tag{4.5}$$

Se utilizarmos a lei de controle

$$\tau = \hat{H}(q) J_T^{-1}[a_T - \dot{J}_T \dot{q}] + \hat{C}(q,\dot{q})\dot{q} + \hat{G}(q) + J_T^T f_T \tag{4.6}$$

onde o sobrescrito ^ indica que alguns parâmetros destas matrizes foram estimados, obteremos em malha fechada, substituindo a eq.(4.6) na eq.(4.5):

$$\ddot{p}_T = a_T \tag{4.7}$$

onde a_T é uma lei de controle complementar desejada.

Desta forma está feito o desacoplamento entre as direções do espaço de tarefas.

Na direção em que queremos controlar a posição, podemos utilizar uma lei de controle do tipo:

$$a_{TP} = \ddot{p}_{TD} + k_{VP}(\dot{p}_{TD} - \dot{p}_T) + k_{PP}(p_{TD} - p_T) \tag{4.8}$$

onde:

a_{TP} é a lei de controle complementar nas direções em que desejamos controlar a posição;

k_{VP} é a matriz diagonal dos ganhos derivativos do controlador de posição;

k_{PP} é a matriz diagonal dos ganhos proporcionais do controlador de posição.

O subscrito 'D' representa valores desejados.

Rescrevendo esta lei em forma de matrizes temos:

$$\begin{bmatrix} a_{TPx} \\ a_{TPy} \\ a_{TPz} \\ a_{TPphi} \end{bmatrix} = \left(\begin{bmatrix} \ddot{x}_{TD} \\ \ddot{y}_{TD} \\ \ddot{z}_{TD} \\ \ddot{phi}_{TD} \end{bmatrix} + \begin{bmatrix} k_{VPx} & 0 & 0 & 0 \\ 0 & k_{VPy} & 0 & 0 \\ 0 & 0 & k_{VPz} & 0 \\ 0 & 0 & 0 & k_{VPphi} \end{bmatrix} \cdot \begin{bmatrix} \dot{x}_{TD} - \dot{x}_T \\ \dot{y}_{TD} - \dot{y}_{TD} \\ \dot{z}_{TD} - \dot{z}_{TD} \\ \dot{phi}_{TD} - \dot{phi}_{TD} \end{bmatrix} + \begin{bmatrix} k_{PPx} & 0 & 0 & 0 \\ 0 & k_{PPy} & 0 & 0 \\ 0 & 0 & k_{PPz} & 0 \\ 0 & 0 & 0 & k_{PPphi} \end{bmatrix} \cdot \begin{bmatrix} x_{TD} - x_T \\ y_{TD} - y_{TD} \\ z_{TD} - z_{TD} \\ phi_{TD} - phi_{TD} \end{bmatrix} \right)$$

Em malha fechada obtemos:

$$\ddot{\tilde{p}}_T + k_{VP}\dot{\tilde{p}}_T + k_{PP}\tilde{p}_T = 0 \qquad (4.10)$$

onde $\tilde{p}_T = p_{TD} - p_T$ e representa o erro de posição.

Para o controle de força podemos utilizar uma lei de controle do tipo:

$$a_{TF} = k_e^{-1}[\ddot{f}_{TD} + k_{VF}(\dot{f}_{TD} - \dot{f}_T) + k_{PF}(f_{TD} - f_T)] \qquad (4.11)$$

onde:

a_{TF} é a lei de controle complementar nas direções em que se deseja controlar a força;

k_e^{-1} é o inverso da matriz que representa a constante elástica das diversas direções do meio em que se deseja aplicar a força;

k_{VF} é a matriz diagonal que representa os ganhos derivativos do controlador de força;

k_{PF} é a matriz diagonal dos ganhos proporcionais do controlador de força.

Em malha fechada obtemos:

$$\ddot{\tilde{f}}_{TD} + k_{VF}\dot{\tilde{f}}_{TD} + k_{PF}\tilde{f}_{TD} = 0 \qquad (4.12)$$

Com matrizes diagonais para definir os ganhos do controlador fica garantido o desacoplamento entre as direções cartesianas do sistema de coordenadas da tarefa. Para selecionar as direções em que desejamos controlar força e as direções em que desejamos controlar posição utilizamos as matrizes de seleção. No caso do controle no espaço de tarefas, as matrizes de seleção serão compostas apenas por zeros e uns. A lei de controle a_T fica então:

$$a_T = \Omega_T a_{TP} + \overline{\Omega}_T a_{TF} \tag{4.13}$$

onde:

Ω_T é a matriz de seleção de posições;

$\overline{\Omega}_T$ é a matriz de seleção de forças;

e:

$$\Omega_T + \overline{\Omega}_T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.14}$$

Desta forma pode ser calculado um sinal de controle de força e posição para cada uma das direções do sistema cartesiano e as matrizes de seleção garantem que somente os sinais desejados sejam efetivamente utilizados. Se no nosso software for possível selecionar as direções em que se quer controlar a força, não será necessária a utilização das matrizes de seleção.

Para escrever a lei de controle completa temos de conhecer as matrizes de inércia, coriolis e centrípeta, gravitacional, o jacobiano da tarefa, sua derivada e sua inversa. Apresentaremos a seguir estas matrizes definidas para um robô com configuração SCARA genérico, não levando em consideração os parâmetros do robô Inter. Para a definição dos Jacobianos foi considerado que o espaço da tarefa é definido apenas através de um deslocamento de origem no plano $x_0 - y_0$ e uma rotação θ_T em torno do eixo z_0.

A matriz de inércia é dada por

$$H(q)=\begin{bmatrix} I_0+I_A+m_0 l_{c0}^{\ 2}+(m_1+m_A)l_0^2+(\frac{m_1}{k}+m_A)(2l_0 l_1\cos\theta_1)+(\frac{m_1}{k^2}+m_A)l_1^2 & I_A+(\frac{m_1}{k}+m_A)(l_0 l_1\cos\theta_1)+(\frac{m_1}{k^2}+m_A)l_1^2 & 0 & -I_3 \\ I_A+(\frac{m_1}{k}+m_A)(l_0 l_1\cos\theta_1)+(\frac{m_1}{k^2}+m_A)l_1^2 & I_A+(\frac{m_1}{k^2}+m_A)l_1^2 & 0 & -I_3 \\ 0 & 0 & m_A & 0 \\ -I_3 & -I_3 & 0 & I_3 \end{bmatrix}$$

onde:

I_0 é o momento de inércia do elo 0;

I_A é a soma dos momentos de inércia dos elos 1, 2 e 3;

I_3 é o momento de inércia do elo 3;

m_0 é a massa do elo 0;

m_1 é a massa do elo 1;

m_A é a soma das massas dos elos 1,2 e 3;

l_0 é o comprimento do elo 0;

l_{c0} é a distância do centro de massa do elo 1 em relação à junta 1;

l_1 é o comprimento do elo 1;

l_{c1} é a distância do centro de massa do elo 0 em relação à junta 0;

θ_1 é o ângulo de giro da junta 1;

k constante que define a relação entre o comprimento do elo 1 e seu centro de massa:

$$k=\frac{l_{c1}}{l_1}$$

A matriz dos torques centrípetos e de coriolis é dada por:

$$C(q,\dot{q})=\begin{bmatrix} -(\frac{m_1}{k}+m_A)l_0 l_1(\text{sen}\,\theta_1)\dot{\theta}_1 & -(\frac{m_1}{k}+m_A)l_0 l_1(\text{sen}\,\theta_1)(\dot{\theta}_0+\dot{\theta}_1) & 0 & 0 \\ -(\frac{m_1}{k}+m_A)l_0 l_1(\text{sen}\,\theta_1)\dot{\theta}_0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (4.16)$$

onde:

θ_i é o ângulo de giro da junta 1;

$\dot{\theta}_0$ é a velocidade da junta 0;

$\dot{\theta}_1$ é a velocidade da junta 1.

A matriz dos torques gravitacionais é dada por:

$$G(q)=\begin{bmatrix} 0 \\ 0 \\ m_A g \\ 0 \end{bmatrix} \quad (4.17)$$

onde:

g é a aceleração da gravidade.

O Jacobiano da tarefa é dado por:

$$J_T=\begin{bmatrix} l_0 S_{(\theta T-\theta 0)}+l_1 S_{(\theta T-(\theta 0+\theta 1))} & l_1 S_{(\theta T-(\theta 0+\theta 1))} & 0 & 0 \\ l_0 C_{(\theta T-\theta 0)}+l_1 C_{(\theta T-(\theta 0+\theta 1))} & l_1 C_{(\theta T-(\theta 0+\theta 1))} & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 1 & 0 & -1 \end{bmatrix} \quad (4.18)$$

onde foi utilizada a seguinte notação:

S é o seno do valor subscrito dentro de parêntesis;

C é o cosseno do valor subscrito dentro de parêntesis.

Por exemplo, $S_{(\theta T-(\theta 0+\theta 1))}$ é o seno de $\left(\theta_T-\left(\theta_0+\theta_1\right)\right)$.

A inversa do Jacobiano da tarefa é:

$$J_T^{-1}=\begin{bmatrix} \frac{C_{(\theta T-(\theta 0+\theta 1))}}{l_0 S_1} & -\frac{S_{(\theta T-(\theta 0+\theta 1))}}{l_0 S_1} & 0 & 0 \\ \frac{-l_1 C_{(\theta T-(\theta 0+\theta 1))}-l_0 C_{(\theta T-\theta 0)}}{l_0 S_1} & \frac{l_1 S_{(\theta T-(\theta 0+\theta 1))}+l_1 S_{(\theta T-(\theta 0+\theta 1))}}{l_0 S_1} & 0 & 0 \\ 0 & 0 & -1 & 0 \\ -\frac{C_{(\theta T-\theta 0)}}{l_1 S_1} & \frac{S_{(\theta T-\theta 0)}}{l_1 S_1} & 0 & -1 \end{bmatrix} \quad (4.19)$$

E sua derivada é:

$$\dot{J}_T=\begin{bmatrix} -l_0 C_{(\theta T-\theta 0)}\dot{\theta}_0-l_1 C_{(\theta T-(\theta 0+\theta 1))}(\dot{\theta}_0+\dot{\theta}_1) & -l_1 C_{(\theta T-(\theta 0+\theta 1))}(\dot{\theta}_0+\dot{\theta}_1) & 0 & 0 \\ l_0 S_{(\theta T-\theta 0)}\dot{\theta}_0+l_1 S_{(\theta T-(\theta 0+\theta 1))}(\dot{\theta}_0+\dot{\theta}_1) & l_1 S_{(\theta T-(\theta 0+\theta 1))}(\dot{\theta}_0+\dot{\theta}_1) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \quad (4.20)$$

Estão assim definidos todos os componentes da lei geral de controle.

4.4 Considerações e Simplificações

A falta do sensor de força dificulta a implementação do controle híbrido pela dificuldade de simular leituras de força durante a realização de testes práticos.

Toda implementação foi dirigida de forma que quando for instalado o sensor sejam necessários apenas pequenos ajustes.

Nos testes práticos foi utilizada uma força desejada igual a zero. O resultado disto é que nas direções em que o controle de força é selecionado é possível movimentar o manipulador manualmente. Neste caso apenas as forças de atrito atuam em sentido contrário ao deslocamento.

Com o sensor de força, o comportamento do manipulador seria diferente. Ao deslocarmos manualmente o manipulador, os torques de controle necessários para atingir a força zero atuariam já compensando qualquer forma de atrito existente. Mesmo neste caso mais simples é impossível simular a presença do sensor de força.

A falta do sensor também impede que as técnicas de controle de força sejam testadas. Após a instalação do sensor deverão ser feitos testes práticos para definir o melhor tipo de controlador a ser utilizado e efetuar o ajuste dos ganhos.

Na definição do sistema de coordenadas da tarefa foram feitas duas considerações simplificadoras.

A primeira é que este sistema foi definido a partir do sistema de coordenadas operacional, considerando apenas um deslocamento x_{OT} e y_{OT} da origem no plano $x_0 - y_0$ e um giro θ_T ao redor de z_0. Não foram, portanto, considerados giros ao redor de x_0 e y_0 nem deslocamentos da origem na direção z_0. Esta consideração não impõe nenhuma restrição além daquelas naturalmente impostas pela configuração do manipulador do tipo SCARA pois este possui apenas quatro graus de liberdade. Não teria sentido definir ângulos de giro ao redor de x_0 e y_0 pois o manipulador não conseguiria atingir orientações definidas a partir destes giros. O fato da origem não poder ser deslocada no sentido z_0 também não cria maiores problemas pois não limita o espaço de trabalho do robô. A origem z_T sempre irá concordar com a origem z_0.

A segunda consideração é que θ_T é constante. Isto implica que a princípio não podemos trabalhar com o controle híbrido em superfícies curvas.

No entanto, veremos que as matrizes nas quais θ_T entra como componente e as quais são utilizada para transformação de coordenadas, são calculadas em procedimentos independentes. Nada impede que um novo θ_T seja definido a cada instante, desde que após cada mudança estas matrizes sejam recalculadas. O que falta realmente é definir uma tarefa na qual θ_T seja variável, criar um procedimento que estabeleça uma função para esta variação de θ_T e a partir deste procedimento chamar os procedimentos que recalculem as matrizes necessárias.

No controlador o fato de θ_T ser constante tem outras implicações. O cálculo da derivada do jacobiano da tarefa é diferente para θ_T constante ou não constante. Neste caso o cálculo da derivada teria de ser refeito.

Nesta etapa de implementação não foi feita a compensação dinâmica e o desacoplamento. Foi utilizada uma lei de controle para cada direção do espaço da

tarefa, desprezando totalmente qualquer tipo de acoplamento. Ao invés de utilizar a lei de controle completa dada por:

$$\tau = \hat{H}(q)\, J_T^{-1}\,[a_T - \dot{J}_T \dot{q}] + \hat{C}(q,\dot{q})\dot{q} + \hat{G}(q) + J_T^{\,T} f_T$$

usamos apenas:

$$\tau = a_T = \Omega_T a_{TP} + \overline{\Omega}_T a_{TF}$$

Para controlar posições durante o controle híbrido foi utilizado o mesmo controlador do tipo PD utilizado no controle de posição normal do robô, apenas com ganhos distintos.

Para controlar forças foi implementado um controlador do tipo PD cujo desempenho ainda não pode ser testado.

Logo, a_{TP} e a_{TF} ficam na forma:

$$\begin{aligned} a_{TP} &= k_{VP}(\dot{p}_{TD} - \dot{p}_T) + k_{PP}(p_{TD} - p_T) \\ a_{TF} &= k_e^{-1}[k_{VF}(\dot{f}_{TD} - \dot{f}_T) + k_{PF}(f_{TD} - f_T)] \end{aligned} \tag{4.21}$$

Esta hipótese de não utilizar a compensação dinâmica é sem dúvida bastante simplificadora mas para casos em que são consideradas pequenas velocidades de deslocamento, este tipo simples de controlador pode apresentar resultados satisfatórios.

Para uma futura implementação do controlador com compensação dinâmica não seria necessário fazer nenhuma alteração nos módulos existentes. Os valores de controle calculados entrariam como parcela da lei completa. Teriam, isto sim, de ser criados novos módulos para o cálculo dos demais termos como jacobianos e matrizes de inércia e um módulo que calculasse os novos torques de controle.

Outra limitação apresentada por este controle híbrido diz respeito à geração de valores desejados. Atualmente é possível estabelecer somente valores desejados constantes para velocidades e forças. Não existe um módulo ou procedimento responsável pela geração destes valores através de funções variáveis no tempo. Também não é possível estabelecer uma trajetória desejada. A idéia é utilizar o módulo *Bahnplaner* para gerar trajetórias no espaço de tarefas. Em princípio não

haveria problema pois neste módulo é indiferente em qual sistema cartesiano esteja sendo feita esta geração de trajetórias. No entanto sua estrutura teria de ser modificada pois atualmente este módulo envia os valores desejados diretamente para o módulo *Main3* realizar o controle. No controle híbrido os valores desejados devem ser enviados para os módulos que efetivamente executam este controle. Falta portanto a implementação de uma rotina que detecte se o controle híbrido está ativado e neste caso envie os dados para o local correto.

A modificação necessária para que estes módulos possam ser utilizados em casos genéricos não é muito simples pois existe uma variedade muito grande de tarefas e cada uma destas tarefas pode impor condições diferentes. No momento que for definida uma atividade específica para o robô ou mesmo um conjunto de atividades fica bem mais fácil executar as alterações necessárias neste módulo para que seja possível fornecer também valores desejados de posição e valores variáveis de velocidade e força.

No momento temos de nos limitar a utilizar apenas velocidades e forças constantes que podem ser estabelecidas para cada uma das direções do espaço de tarefas.

4.5 Transformação de Coordenadas

Para implementar o controle híbrido é necessária uma série de transformações entre os sistemas de coordenadas anteriormente apresentados. A maioria destas transformações pode ser feita com a utilização de matrizes, mas algumas delas ficam mais simples utilizando cálculos recursivos.

TRANSFORMAÇÃO DE POSIÇÕES

Do espaço de juntas para o espaço operacional (Cinemática Direta):

$$\begin{aligned} x_O &= l_1 C_{(\theta 0+\theta 1)} + l_0 C_{(\theta 1)} \\ y_O &= l_1 S_{(\theta 0+\theta 1)} + l_0 S_{(\theta 1)} \\ z_O &= 0.665 - d_2 \\ phi_O &= \left(\theta_o + \theta_1 - \theta_3\right) \end{aligned} \tag{4.22}$$

Do espaço operacional para o espaço de juntas (Cinemática Inversa):

$$\theta_1 = \arctan\left(\frac{\sqrt{1-\left(\frac{x_0^{\,2}+y_0^{\,2}-l_0^{\,2}-l_1^{\,2}}{2l_0l_1}\right)^2}}{\left(\frac{x_0^{\,2}+y_0^{\,2}-l_0^{\,2}-l_1^{\,2}}{2l_0l_1}\right)}\right)$$

$$\theta_0 = \arctan\left(\frac{y_0}{x_0}\right) + \arctan\left(\frac{l_1 S_1}{l_0 + l_1 C_1}\right) \tag{4.23}$$

$$d_3 = 0.665 - z_0$$

$$\theta_3 = \theta_0 + \theta_1 - phi_0$$

Do espaço operacional para o espaço da tarefa:

$$\begin{bmatrix} x_T \\ y_T \\ z_T \\ phi_T \end{bmatrix} = -\begin{bmatrix} C_{\theta T} & S_{\theta T} & 0 & 0 \\ -S_{\theta T} & C_{\theta T} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_{OT} \\ y_{OT} \\ z_{OT} \\ \theta_T \end{bmatrix} + \begin{bmatrix} C_{\theta T} & S_{\theta T} & 0 & 0 \\ -S_{\theta T} & C_{\theta T} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_O \\ y_O \\ z_O \\ phi_O \end{bmatrix} \tag{4.24}$$

Do espaço da tarefa para o espaço operacional:

$$\begin{bmatrix} x_O \\ y_O \\ z_O \\ phi_O \end{bmatrix} = \begin{bmatrix} x_{OT} \\ y_{OT} \\ z_{OT} \\ \theta_T \end{bmatrix} + \begin{bmatrix} C_{\theta T} & -S_{\theta T} & 0 & 0 \\ S_{\theta T} & C_{\theta T} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_T \\ y_T \\ z_T \\ phi_T \end{bmatrix} \tag{4.25}$$

TRANSFORMAÇÃO DE VELOCIDADES

Do espaço de juntas para o espaço operacional:

$$\begin{bmatrix} \dot{x}_O \\ \dot{y}_O \\ \dot{z}_O \\ \dot{phi}_O \end{bmatrix} = \begin{bmatrix} -(l_0 S_{\theta 0} + l_1 S_{(\theta 0+\theta 1)}) & -l_1 S_{(\theta 0+\theta 1)} & 0 & 0 \\ (l_0 C_{\theta 0} + l_1 C_{(\theta 0+\theta 1)}) & l_1 C_{(\theta 0+\theta 1)} & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 1 & 1 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} \dot{\theta}_0 \\ \dot{\theta}_1 \\ \dot{d}_2 \\ \dot{\theta}_3 \end{bmatrix} \tag{4.26}$$

Do espaço operacional para o espaço de juntas:

$$\begin{bmatrix} \dot{\theta}_0 \\ \dot{\theta}_1 \\ \dot{d}_2 \\ \dot{\theta}_3 \end{bmatrix} = \begin{bmatrix} \frac{l_1 C_{(\theta 0+\theta 1)}}{l_0 l_1 S_{\theta 1}} & \frac{l_1 S_{(\theta 0+\theta 1)}}{l_0 l_1 S_{\theta 1}} & 0 & 0 \\ \frac{-(l_0 C_{\theta 0} + l_1 C_{(\theta 0+\theta 1)})}{l_0 l_1 S_{\theta 1}} & \frac{-(l_0 S_{\theta 0} + l_1 S_{(\theta 0+\theta 1)})}{l_0 l_1 S_{\theta 1}} & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \frac{-l_0 C_{\theta 0}}{l_0 l_1 S_{\theta 1}} & \frac{-l_0 S_{\theta 0}}{l_0 l_1 S_{\theta 1}} & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} \dot{x}_O \\ \dot{y}_O \\ \dot{z}_O \\ \dot{phi}_O \end{bmatrix} \tag{4.27}$$

Do espaço operacional para o espaço da tarefa:

$$\begin{bmatrix} \dot{x}_T \\ \dot{y}_T \\ \dot{z}_T \\ \dot{phi}_T \end{bmatrix} = \begin{bmatrix} C_{\theta T} & S_{\theta T} & 0 & 0 \\ -S_{\theta T} & C_{\theta T} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \dot{x}_O \\ \dot{y}_O \\ \dot{z}_O \\ \dot{phi}_O \end{bmatrix} \tag{4.28}$$

Do espaço da tarefa para o espaço operacional:

$$\begin{bmatrix} \dot{x}_O \\ \dot{y}_O \\ \dot{z}_O \\ \dot{phi}_O \end{bmatrix} = \begin{bmatrix} C_{\theta T} & S_{\theta T} & 0 & 0 \\ -S_{\theta T} & C_{\theta T} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \dot{x}_T \\ \dot{y}_T \\ \dot{z}_T \\ \dot{phi}_T \end{bmatrix} \tag{4.29}$$

TRANSFORMAÇÃO DE FORÇAS OU MOMENTOS

Do espaço do efetuador final para o espaço operacional, onde $\beta = (\theta_3 - \theta_0 - \theta_1)$:

$$\begin{bmatrix} Fx_O \\ Fy_O \\ Fz_O \\ Mz_O \end{bmatrix} = \begin{bmatrix} C_\beta & -S_\beta & 0 & 0 \\ -S_\beta & -C_\beta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} Fx_E \\ Fy_E \\ Fz_E \\ Mz_E \end{bmatrix} \tag{4.30}$$

Do espaço operacional para o espaço de juntas.

$$\begin{bmatrix} \tau_0 \\ \tau_1 \\ F_2 \\ \tau_3 \end{bmatrix} = \begin{bmatrix} -(a_0 S_0 + a_1 S_{(0+1)}) & (a_0 C_0 + a_1 C_{(0+1)}) & 0 & 1 \\ -a_1 S_{(0+1)} & a_1 C_{(0+1)} & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} \cdot \begin{bmatrix} Fx_O \\ Fy_O \\ Fz_O \\ Mz_O \end{bmatrix} \tag{4.31}$$

Do espaço operacional para o espaço da tarefa.

$$\begin{bmatrix} Fx_T \\ Fy_T \\ Fz_T \\ Mz_T \end{bmatrix} = \begin{bmatrix} C_{\theta T} & S_{\theta T} & 0 & 0 \\ -S_{\theta T} & C_{\theta T} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} Fx_O \\ Fy_O \\ Fz_O \\ Mz_O \end{bmatrix} \tag{4.32}$$

Do espaço da tarefa para o espaço operacional.

$$\begin{bmatrix} Fx_O \\ Fy_O \\ Fz_O \\ Mz_O \end{bmatrix} = \begin{bmatrix} C_{\theta T} & -S_{\theta T} & 0 & 0 \\ S_{\theta T} & C_{\theta T} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} Fx_T \\ Fy_T \\ Fz_T \\ Mz_T \end{bmatrix} \tag{4.33}$$

Como dito anteriormente, foi escolhido implementar o controle híbrido no espaço de tarefas. Logo, os valores desejados de posição, velocidade e força devem ser definidos neste espaço. Como as forças são lidas no espaço do efetuador final, deverão ser transformadas para o espaço da tarefa para que a lei de controle possa ser aplicada.

Para posições e velocidades, a leitura é feita no espaço de juntas e os valores também deverão ser transformados para o espaço da tarefa. No final, o sinal de controle gerado no espaço da tarefa deverá ser transformado para o espaço de juntas para que possa ser enviado ao amplificador.

Mostramos a seguir os passos utilizados no software para implementar o controle híbrido:

1.definição do sistema de coordenadas da tarefa;

2.definição dos valores desejados de velocidade e força no sistema de coordenadas da tarefa;

3.leitura da força no sistema de coordenadas do efetuador final;

4.transformação das forças lidas para o espaço da tarefa;

5.leitura de posições e velocidades no espaço de juntas;

6.transformação das posições e velocidades lidas para o espaço da tarefa;

7.aplicação da lei de controle para definição dos torques(forças) de controle no espaço da tarefa;

8.aplicação das matrizes de seleção;

9.transformação dos valores de controle do espaço da tarefa para o espaço de juntas.

4.6 Módulos do Controle Híbrido

A implementação do controle híbrido foi dividida em 3 módulos que são: *HybridCtrl*, *CartesianTransf* e *HCTest*. Descreveremos aqui a função básica de cada um. O detalhamento de cada um destes módulos pode ser encontrado no apêndice C.

No módulo *CartesianTransf* estão todos os procedimentos para transformação de coordenadas à exceção das que envolvem o espaço de juntas, pois estas já haviam sido implementadas no módulo *ScaraCarthKin*. Neste módulo também é criado o objeto *Frame* que define o espaço da tarefa através do ângulo θ_T e das distâncias x_{OT} e y_{OT}.

O módulo *HybridCtrl* cria o objeto *HC*. Este objeto irá conter todas as informações para a realização do controle híbrido como os ganhos dos controladores de posição, velocidade e força, o tipo de controle a ser utilizado em cada direção do espaço de tarefas e as matrizes de seleção. A este objeto também está associado o método *Algo* que gerencia toda a seqüência de transformações de um sistema para outro, chama os algoritmos de controle e aplica as matrizes de seleção. Deste método saem os valores de controle já no espaço de juntas.

O módulo *HCTest* tem duas funções básicas. Faz toda interface com o usuário fornecendo e recendo informações e gerencia o funcionamento do controle híbrido através das tarefas *HCCtrl* e *MainHCCtrl* que são respectivamente do tipo *EveryEvent* e *MainEvent*.

4.7 O Controle Híbrido em Funcionamento

Para colocar o controle híbrido efetivamente em funcionamento devemos seguir os seguintes passos.

Configurar os objetos *Frame0* e *HyCtrl*, colocando-os na biblioteca, através dos comandos:

```
CartesianTrans.DefCarth Frame0 ( teta=0.665, x0Task=0.0, y0Task=0.03) ~
XSystem.Call HybridCtrl.DefHC HyCtrl
(clock=0.001, FCtrlX=HFCtrlX, FCtrlY=HFCtrlY, PISCtrlX=HSCtrlX, PISCtrlY=HSCtrlY,
PISCtrlZ=HSCtrlZ, PISCtrlPhi=HSCtrlPhi, kin=SCCKin, frame=Frame0, drive0=Drive0,
drive1=Drive1, drive2=Drive2, drive3=Drive3,
XCtrl='speed', YCtrl='speed', ZCtrl='speed', PZCtrl='speed') ~
```

Chamar o procedimento *Init* através do comando:

```
XSystem.Call  HCTest.Init ~
```

Chamar o procedimento *StartHC* através do comando:

```
XSystem.Call  HCTest.StartHC ~
```

Este procedimento chama o procedimento *InitTask*. Após este comando, o controle híbrido está em funcionamento pois as tarefas *HCCtrl* e *MainHCCtrl* já estão instaladas. Na direção x está sendo feito o controle de força e nas demais direções está sendo feito o controle de velocidade. Os valores desejados de força e velocidade são

no momento zero. Nesta situação, como a força desejada na direção xT é zero, é possível movimentar o manipulador manualmente nesta direção. Nas demais direções, a velocidade desejada é zero e o controlador atua de forma a garantir que a velocidade permaneça zero. O manipulador fica portanto rígido nestas direções, não permitindo movimentos.

Nestas direções onde é feito o controle de velocidade nunca será possível movimentar o robô manualmente pois sempre termos um sinal de controle atuando. Para movimentar o robô em algumas desta direções temos de utilizar comandos que especifiquem a direção, velocidade e aceleração desejadas.

Para alterar o ângulo teta e definir um novo espaço da tarefa utilizamos o comando:

```
XSystem.Call  HCTest.TSetTeta0.78 ~(* teta in rad *)
```

Neste caso estamos rotacionando o espaço de tarefa em aproximadamente 45 graus, definindo novas direções x e y para o espaço da tarefa.

Para alterar o tipo de controlador em uma das direções do espaço de tarefa utilizamos o comando:

```
XSystem.Call  HCTest.TChangeCtrl    0   'speed' ~ (** haxis, controller ('pos' OR 'speed' OR 'force' OR 'noctrl') *)
```

As direções x, y, z e phi são identificadas pelos números 0, 1, 2 e 3 respectivamente. Neste exemplo apresentado estamos alterando para velocidade o tipo de controlador a ser utilizado na direção x, que anteriormente estava em força.

Para alterar a velocidade de alguma das direções utilizamos o comando:

```
XSystem.Call  HCTest.TSetAxisSpeedAcc 0   0.05    1.0 ~(* haxis, speed acc *)
```

Neste caso estamos estabelecendo a velocidade na direção x do espaço da tarefa em 0.05 m/s e a aceleração em 1.0 m/s^2. Se na direção x estiver selecionado o controle de velocidade, o manipulador irá se movimentar nesta direção com esta velocidade. Se nas outras direções permanecer a velocidade desejada igual a zero, o manipulador irá se deslocar apenas na direção x do espaço da tarefa enquanto que nas demais direções não deverá sofrer nenhum tipo de deslocamento.

Para parar o controle híbrido devemos utilizar o comando:

```
XSystem.Call  HCTest.StopHC ~
```

Estes são os principais comandos do controle híbrido. Após a execução de cada comando surge uma mensagem na tela informando se este comando foi realizado com sucesso ou não.

4.8 Modificações e Melhorias Propostas

Estes módulos para o controle híbrido com certeza não estão no seu estágio final de desenvolvimento. Existe bastante coisa para ser corrigida e melhorada.

A maior problema continua sendo a ausência do sensor de força. O controle híbrido exige que o controle de força seja feito de forma adequada. Quando for instalado o sensor de força, não bastará apenas alterar alguns comandos para efetivar o controle híbrido. Terão de ser feitos uma série de testes apenas com o controle de força puro até que se obtenha um resultado satisfatório que possa ser utilizado no controle híbrido.

Na implementação do controle híbrido não foi feito nenhum tipo de compensação dinâmica. Para tarefas em que pequenas velocidades e acelerações são utilizadas, isto não é um problema muito grave. Mas no momento em que forem feitas trajetórias mais rápidas e complexas deverá ser implementada a compensação dinâmica. Talvez a presença do sensor de força também exija respostas dinâmicas mais rápidas o que torna imprescindível a utilização de um algoritmo de compensação dinâmica.

O fato de não podermos ainda especificar uma trajetória para as direções do espaço de tarefa é sem dúvida uma limitação. Foi mostrado em capítulo anterior as diretrizes para fazer esta importante alteração. No entanto, deve ser reenfatizado o comentário de que seria mais fácil promover esta alteração após definida uma tarefa específica a ser executada.

A importância destes módulos de controle híbrido é deixar pronta uma estrutura básica que realmente funciona e que apesar de apresentar algumas limitações já foi utilizada em alguns testes práticos. São naturalmente necessárias algumas

modificações e a instalação do sensor de força para que o controle híbrido possa ser efetivamente utilizado.

Capítulo 5

CONCLUSÃO

Este trabalho teve como objetivo descrever o funcionamento do robô Inter, procurando facilitar o entendimento da pessoa interessada em trabalhar com o robô.

Não se trata de uma dissertação nos moldes tradicionais, envolvendo um problema a ser enfocado, estudado, discutido e dentro do possível testado, e no final são apresentadas conclusões sobre o assunto.

Este trabalho é basicamente uma coletânea de informações, sejam elas obtidas durante o acompanhamento da fabricação do robô e do desenvolvimento de softwares ou de catálogos.

Seria bastante fácil descrever apenas os comandos utilizados na movimentação do robô. Mas isto fugiria ao objetivo do trabalho que é permitir que outras pessoas venham a desenvolver trabalhos de pesquisa com este robô, escrevendo seus próprios programas. A pesquisa exige um conhecimento bem mais profundo sobre o robô, não apenas sobre os softwares de controle mas também sobre seus componentes mecânicos.

Apenas no capítulo 4 é feito um estudo sobre o controle híbrido de força/posição e são apresentados os módulos implementados no robô para a realização desta tarefa. A falta do sensor de força não permitiu maiores evoluções neste sentido, mas a estrutura básica do problema está montada.

A maior contribuição deste trabalho ao meu entender foi instalar o robô e deixá-lo em condições de ser utilizado por outras pessoas. Todos os problemas que surgem normalmente quando se está fabricando um produto novo consumiram um tempo razoável e infelizmente não permitiram que fossem aprofundados alguns pontos essenciais na área de controle. Descrevo a seguir uma série de trabalhos que podem ser

desenvolvidos com o robô e espero conseguir despertar o interesse de pessoas para que trabalhem nesta área.

O robô Inter é um instrumento de pesquisa e como tal deve estar em freqüente evolução. Suas limitações atuais não devem ser consideradas como defeitos ou problemas e sim como desafios a serem superados no processo de desenvolvimento.

O controlador de trajetórias atualmente em uso é bastante rudimentar. As ferramentas de plotagem são restritas. O programa de geração de trajetórias também apresenta limitações. O sensor de força foi implementado na fase final desta dissertação e nenhum tipo de teste foi realizado.

Tendo em vista o acima exposto, sugiro uma série de trabalhos que podem ser desenvolvidos com este robô:

1.Desenvolvimento e teste de novos algoritmos de controle de posição;

2.Implementação de algoritmo de compensação dinâmica;

3.Desenvolvimento e teste de novos algoritmos de controle de força;

4.Aperfeiçoamento dos módulos existentes de controle híbrido de força/posição;

5.Desenvolvimento de uma aplicação prática a ser realizada pelo robô, utilizando técnicas de controle de força, como por exemplo gravação de vidros.

6.Aperfeiçoamento das ferramentas de plotagem existentes

Estas são apenas algumas sugestões de trabalhos. Como dito na introdução, os limites são a criatividade e imaginação do pesquisador. Nada impede que sejam feitas pesquisas nas áreas de geração de trajetórias, estudo da flexibilidade das juntas para o desenvolvimento de técnicas de controle, desenvolvimento de técnicas para efetuar o contato com superfícies, desenvolvimento de técnicas para evitar colisões ou regiões singulares, desenvolvimento de técnicas de controle de visão a serem integradas ao robô.

As possibilidades de pesquisa são amplas e espero que este importante instrumento colocado a nossa disposição seja efetivamente utilizado.

REFERÊNCIAS BIBLIOGRÁFICAS

1 ASADA, H.; SLOTINE, J.-J. E., ***Robot Analysis and Control.*** Nova Iorque: John Wiley and Sons, 1986. 266p.

2 LEWIS, F. D.; ABDALLAH, C. T.; DAWSON, D. M., ***Control of Robot Manipulators.*** Nova Iorque: MacMillan, 1993. 424p.

3 MÖSSENBÖCK, H., ***Object-Oriented Programming in Oberon-2.*** Berlim: Springer-Verlag, 1993. 278p.

4 REISER, M., ***The Oberon System:Users Guide and Programming Manual.*** Addison-Wesley, 1992.

5 SPONG, M. W.; VIDYASAGAR, M., ***Robot Dynamics and Control.*** Nova Iorque: John Wiley and Sons, 1989. 336p.

6 VESTLI, S. J., **Handbook - Denia/Xoberon**. *Technical report, Instituo de Robótica da Universidade Federal de Zurique, Suíça, 1997.*

7 WEIHMANN, L., **Módulos principais e arquivos de configuração do robô manipulador Inter.** *Technical report, Laboratório de Robótica da universidade Federal de Santa Catarina, 1998.*

8 YOSHIKAWA, T., **Dynamic Hybrid Position/Force Control of Robot Manipulators**: Description of Hand Constraints and Calculation of Joint Driving Force. *IEEE Journal of Robotics and Automation,* v.RA-3, p.386-92, 1987.

APÊNDICE A

A LINGUAGEM XOBERON

A.1 Introdução

A linguagem XOberon é bastante nova e ainda está em fase de aperfeiçoamento. Não existe portanto um manual completo que sirva como guia para seu aprendizado. O que temos atualmente são livros sobre Oberon, precursor do Xoberon, e algumas apostilas.

Neste anexo as informações disponíveis são organizadas, sendo também acrescentados exemplos e explicações, de forma a facilitar o aprendizado desta linguagem.

Como o objetivo deste anexo é auxiliar o leitor a entender os programas atualmente existentes no robô Inter adquirido pela UFSC e num estágio posterior escrever seus próprios programas, muitos tópicos apresentados nas apostilas e livros que não estão diretamente relacionados com a utilização do robô, não serão aqui abordados.

Na metodologia proposta para este aprendizado, o leitor será requisitado freqüentemente a ler as apostilas e alguns capítulos dos livros abaixo citados:

- The Oberon System - User Guide and Programmer's Manual, ***Martin Reiser[4];***
- Object-Oriented Programming in Oberon-2, ***Hanspeter Mössenböck [3];***
- Handbook - Denia/XOberon, ***Sjur J. Vestli [6];***

Após cada leitura requisitada, serão discutidos os pontos principais e aqueles que tenham gerado alguma dúvida e, conforme a necessidade, serão apresentados alguns exemplos.

A.2 Descrição Geral

A linguagem Oberon foi desenvolvida na Universidade Técnica de Zurique - Suíça. Segue as características das linguagens Pascal e Modula-2.

O XOberon foi desenvolvido no Instituto de Robótica desta Universidade e é uma extensão do Oberon. Esta extensão confere à linguagem as características de sistema operacional em tempo real.

No XOberon é possível definir tarefas e o tempo de execução destas tarefas. Se por algum motivo o tempo estipulado não puder ser cumprido, o sistema detectará o erro e o programa será finalizado, enviando uma mensagem de erro.

Supondo que temos um robô cujo valor de controle deve ser atualizado a cada 5 milisegundos. Caso a computação envolvida seja muito complexa e o computador só consiga completar os cálculos a cada 10ms, o programa será parado. Se o programa não fosse parado o sinal de controle teria um atraso que aumentaria 5ms em cada ciclo. Isto pode acarretar conseqüências graves, principalmente quando for necessário algum tipo de sincronização com outros robôs ou sistemas mecatrônicos.

Para programação em tempo real é fundamental uma linguagem confiável e que realmente possua as características de sistema operacional em tempo real. O XOberon atende a estes requisitos.

A.3 Linguagem de Programação

Algumas orientações para o leitor a respeito da formulação deste apêndice.

Programas ou mesmo parte deles serão escritos em letras com fonte menor que o texto normal.

Nomes de módulos, procedimentos ou variáveis inseridos no texto, assim como palavras em inglês cuja tradução descaracterizaria seu significado serão escritos em itálico. *Module*, *procedure e pointer* foram respectivamente traduzidas como módulo, procedimento e ponteiro.

Como início do aprendizado do XOberon deverá ser lido o apêndice A do livro Object-Oriented Programmming in Oberon-2 [3]. O leitor que tiver um conhecimento

de Pascal ou mesmo em C provavelmente não terá dificuldades nesta etapa. Aquele que não estiver familiarizado com programação orientada a objetos ou com ponteiros não deverá se ater nestes tópicos pois serão devidamente explicados posteriormente.

Para descrever a lógica e a sintaxe do XOberon utilizaremos o módulo apresentado a seguir. Este módulo faz parte dos programas do robô Inter mas sua função não será explicada no momento.

```
MODULE RampFilterLib;
        (* mecos Robotics AG / Robot Install GmbH *)
        (* Autor: E.Nielsen; Date: 960914 *)
        (* Version 960914 *)

IMPORT
        XTexts, Base, Objects;

CONST
        Version = 960914;
        Dist = 15.23;

TYPE
        RampType* =     RECORD
                new, old, start, step: REAL;
                steps, count: LONGINT;
              END;

VAR
        version- : LONGINT;

PROCEDURE InitRampFilter* (time, clock, old: REAL; VAR c: RampType);
BEGIN
        c.steps := ENTIER(time/clock - 0.5);
        c.step := 0.0;
        c.count := 0;
        c.old := old;
        c.new := c.old;
        c.start := c.new;
END InitRampFilter;

PROCEDURE RampFilter* (new:  REAL; VAR c: RampType):REAL;
BEGIN
        IF c.count < c.steps THEN INC(c.count) END;
        IF new # c.old THEN
                c.new := new;
                c.step := (c.new - c.old) / (c.steps+1);
                c.start := c.old + c.step;
                c.count := 0;
                c.old := c.new;
                RETURN c.start
        ELSIF c.count >= c.steps THEN RETURN c.new
        ELSE RETURN c.start + c.count * c.step
        END;
END RampFilter;
```

```
BEGIN
        version := Version;

END RampFilterLib.
```

A estrutura da linguagem XOberon é modular. Cada módulo é uma coleção de declarações de constantes, tipos, variáveis, procedimentos e pode ser compilado separadamente.

Analisaremos a seguir a estrutura do módulo *RampFilterLib*. Será mostrada também a convenção adotada na nomenclatura de constantes, variáveis e procedimentos utilizados na programação do Inter. Esta convenção não é obrigatória mas facilita a compreensão dos programas e estabelece uma padronização.

Passemos à análise da estrutura do módulo *RampFilterLib*:

MODULE - Inicia novo módulo. Segue o nome do módulo com a primeira letra maiúscula e talvez alguma intermediária também maiúscula, no caso de um nome composto. Não são permitidos espaços no nome.

```
MODULE RampFilterLib;
```

COMENTÁRIOS - Os comentários podem ser inseridos em qualquer ponto do programa. Ao iniciar o comentário devemos usar os símbolos "(*" e ao terminar usamos "*)". O compilador ignora o que estiver no interior destes símbolos.

```
(* mecos Robotics AG / Robot Install GmbH *)
(* Autor: E.Nielsen; Date: 960914 *)
(* Version 960914 *)
```

IMPORT - Caso o módulo em questão necessite de alguma variável, tipo ou procedimento de algum outro módulo, este deve ser importado para que seus dados possam ser utilizados.

```
IMPORT
        XTexts, Base, Objects;
```

O módulo *RampFilterLib* importa portanto os módulos *XTexts, Base e Objects*. Supondo que o módulo *Base* tem uma variável chamada *point*, para utilizarmos esta variável dentro do módulo *RampFilterLib*, deveremos nos referir a ela como:

```
Base.point
```

CONST - Associa um identificador com uma constante. A primeira letra dos identificadores deve ser maiúscula.

```
CONST
    Version = 960914;
    Dist = 15.23;
```

Neste caso os identificadores são *Version* e *Dist.*

TYPE - No apêndice de Mössenböck [3] é fornecida a definição dos diferentes tipos existentes. Neste módulo, *RampType* é do tipo *RECORD* que consiste num número fixo de elementos chamados campos, sendo que cada um destes campos pode ser de diferentes tipos.

```
TYPE
    RampType* =  RECORD ...
        new, old, start, step: REAL;
        steps, count: LONGINT;
    END;
```

Neste caso, os campos de *RampType* são do tipo *Basic Type*. As variáveis *new*, *old*, *start* e *step* são do tipo *REAL* e as variáveis *steps* e *count* são do tipo *LONGINT*. O asterisco após *RampType* indica que é exportado. Isto significa que se outro módulo importar *RampFilterLib* poderá utilizar e modificar os campos deste tipo. Se após um tipo houver um sinal de menos, este poderá ser lido por outros módulos mas não modificado. Se não houver nenhum destes sinais, este tipo simplesmente não existirá para o módulo importador.

Quando tivermos campos do mesmo tipo, podemos defini-los conjuntamente, separando-os por vírgulas. Após o nome do identificador, devemos usar dois pontos para então definir seu tipo.

O nome do *RECORD* deve ter a primeira letra maiúscula e cada um de seus campos deve ter seu nome escrito em minúsculas. Após a descrição dos vários campos de um tipo, deve vir obrigatoriamente *END*, determinando seu final.

VAR - Introduz uma variável ao definir um identificador e um tipo de dado para este identificador.

```
VAR
        version- : LONGINT;
```

Neste caso o identificador *version* é do tipo *LONGINT* e deve ser escrito com letras maiúsculas. O sinal de menos indica que esta variável é exportada mas não pode ser alterada por outros módulos. No apêndice de Mössenböck [3] estão os exemplos dos tipos de variáveis existentes.

PROCEDURE - A definição de procedimento está no apêndice A de Mössenböck [3] . A análise dos procedimentos de *RampFilterLib* será feita após a conclusão da discussão sobre o módulo em geral.

BEGIN - Inicia o corpo do módulo. Toda vez que o módulo é carregado, as informações contidas em seu corpo serão executadas.

É possível executar procedimentos chamando-os a partir do corpo de um módulo. Mas é preferível utilizar chamadas externas pois desta forma é muito mais fácil de manter o controle sobre o que está ocorrendo. Chamadas externas são comandos escritos em uma tela qualquer do ambiente XOberon e executados pelo usuário no momento desejado. Portanto, no corpo de um módulo deve ser escrito somente aquilo que for realmente indispensável.

```
BEGIN
```

```
version := Version;
```

END - Indica o término do módulo. Segue o nome do módulo e o ponto final.

```
END RampFilterLib.
```

Feita a análise da estrutura do módulo, passaremos à estrutura dos procedimentos.

Os dois tipos básicos de procedimentos são *proper procedures* e *function procedures*. Estes últimos normalmente fazem parte de uma expressão e fornecem como resultado um valor que é um dos operandos desta expressão. Dentro do corpo de um *function procedure* normalmente existe um *return statment*. Os *proper procedures* podem ser utilizados para inicialização de variáveis como no procedimento *InitRampFilter* apresentado a seguir.

```
PROCEDURE InitRampFilter* (time, clock, old: REAL; VAR c: RampType);
BEGIN
        c.steps := ENTIER(time/clock - 0.5);
        c.step  := 0.0;
        c.count := 0;
        c.old := old;
        c.new  := c.old;
        c.start  := c.new;
END InitRampFilter;
```

Faremos a análise deste procedimento explicando sua estrutura.

PROCEDURE - Inicia o procedimento. Segue o nome, sempre com a primeira letra maiúscula. O asterisco indica que este procedimento é exportado, isto é, um módulo que importe *RampFilterLib* pode ativar este procedimento. Entre parêntesis é feita a lista com a definição dos parâmetros formais. Estes parâmetros formais podem ser do tipo valor ou variável, identificados na lista através da presença ou ausência de *VAR*.

```
PROCEDURE InitRampFilter* (time, clock, old: REAL; VAR c: RampType);
```

Time, clock e *old* são identificadores do tipo *REAL* e assumem os valores fornecidos pelo procedimento que chamou *InitRampFilter*. A variável *c* é do tipo *RampType*. Quando um módulo chama *InitRampFilter*, deve fornecer um identificador para a variável *c*. Os valores são então calculados para este novo identificador e retornam para o módulo que o chamou.

Posteriormente criaremos o módulo *ChamaRamp* que irá chamar o procedimento *InitRampFilter* de forma a compreendermos melhor este mecanismo.

BEGIN - Inicia o corpo do procedimento.

Antes de *BEGIN* podem existir declarações de constantes, variáveis e tipos, no mesmo molde das apresentadas na análise do módulo, só que neste caso permanecem locais ao procedimento. Neste procedimento em particular não houve necessidade de declarações.

```
BEGIN
        c.steps := ENTIER(time/clock - 0.5);
        c.step  := 0.0;
        c.count := 0;
        c.old := old;
        c.new  := c.old;
        c.start  := c.new;
```

Segue uma série de *assignments* onde as variáveis assumem o valor especificado por uma expressão. Para escrever o operador do *assignment* utiliza-se ":=".

ENTIER é um procedimento pré-declarado e fornece o valor inteiro de uma expressão.

END - Assim como no módulo, o procedimento também deve ser finalizado, mas neste caso com ponto e vírgula.

```
END InitRampFilter;
```

O procedimento *RampFilter* possui estrutura semelhante a *InitRampFilter* e a descrição de seu funcionamento será feita posteriormente.

Criaremos a seguir um módulo que chame os procedimentos *InitRampFillter* e *RampFilter*.

```
MODULE  ChamaRamp;
        (* módulo para chamar RampFilter *)
        (* possui apenas função didática. Não é
          utilizado como software funcional do robô*)
IMPORT
        RF := RampFilterLib;
CONST
        Time = 0.02;
        Clock = 0.005;
        Pos1 = 0.0;
        Pos2 = 1.0;
        Pos3 = 2.0;
VAR
        rpos : RF.RampType;

PROCEDURE CRamp;
VAR
        i, j : INTEGER; posdesej2, posdesj3 : ARRAY  4 OF REAL;
BEGIN
        RF.InitRampFilter(Time, Clock, Pos1, rpos);
        FOR i := 0 TO 3 DO
                posdesej2[ i ] := RF.RampFilter(Pos2, rpos);
        END;
        FOR j := 0 TO 3 DO
                posdesej3[ j ] := RF.RampFilter(Pos3, rpos);
        END;
END CRamp;
BEGIN
END ChamaRamp.
```

Passaremos à análise deste módulo. Para iniciar um novo módulo devemos atribuir um nome a ele.

```
MODULE  ChamaRamp;
```

A seguir podemos fazer alguns comentários.

```
(* módulo para chamar "RampFilter" *)
(* possui apenas função didática. Não é
  utilizado como software funcional do robô*)
```

Como chamaremos procedimentos do módulo *RampFilterLib*, deveremos importá-lo. Também será utilizada uma variável do tipo *RampType*. O nome foi abreviado, o que evita de escrevê-lo por extenso toda vez que for utilizada alguma variável ou tipo do módulo importado.

```
IMPORT
        RF := RampFilterLib;
```

Declaração de algumas constantes que serão utilizadas em operações posteriores.

```
CONST
        Time = 0.02;
        Clock = 0.005;
        Pos1 = 0.0;
        Pos2 = 1.0;
        Pos3 = 2.0;
```

Declaração da variável *rpos*, que é do tipo *RampType*.

```
VAR
        rpos : RF.RampType;
```

Feitas as declarações do módulo, podemos escrever o procedimento. Inicialmente devemos nomeá-lo.

```
PROCEDURE CRamp;
```

A seguir, fazemos as declarações do procedimento. As variáveis *i* e *j* são do tipo inteiro enquanto *posdesej2* e *posdesej3* são vetores de 4 elementos reais.

```
VAR
        i, j : INTEGER; posdesej2, posdesj3 : ARRAY  4 OF REAL;
```

Feitas as declarações, podemos iniciar o corpo do procedimento. Analisaremos o corpo já levando em consideração sua interação com o módulo *RampFilterLib*.

```
BEGIN
        RF.InitRampFilter(Time, Clock, Pos1, rpos);
```

Nesta linha o procedimento *InitRampFilter* do módulo *RampFilterLib* é chamado. Os identificadores *time*, *clock*, *old* assumem respectivamente os valores de *Time*, *Clock*, *Pos1* enviados pelo procedimento *CRamp*.

No procedimento *InitRampFilter* é verificado se estas constantes são do tipo *REAL* e se *rpos* é do tipo *RampType*. Feita esta conferência, são calculadas as expressões e os vários campos de *rpos* assumem os valores abaixo apresentados.

```
rpos.steps := ENTIER(0.02/0.005 - 0.5); rpos.steps := 3.0;
rpos.step := 0.0;
rpos.count := 0;
rpos.old := 0.0;
rpos.new := 0.0;
rpos.start := 0.0;
```

É indispensável que todas variáveis definidas sejam inicializadas. A utilização de variáveis não inicializadas pode levar a resultados estranhos e em certos casos perigosos.

Cabe aqui um pequeno parêntesis sobre utilização de matrizes e vetores em XOberon. Para nos referenciarmos aos vários termos de um vetor, utilizamos a seguinte notação:

```
nomedovetor[indice].
```

Por exemplo, se quisermos somar o terceiro termo do vetor *pos* com o quinto termo do vetor *vel*, escrevemos:

```
pos[2]+vel[4],
```

pois o primeiro termo de matrizes e vetores em XOberon tem por definição o índice zero. Continuamos com a descrição dos procedimentos *CRamp* e *InitRampFilter*.

O procedimento *CRamp* somente passa para as linhas seguintes, apresentadas abaixo, após o procedimento *InitRampFilter* ter sido concluído.

```
FOR i := 0 TO 3 DO
        posdesej2[ i ] := RF.RampFilter(Pos2, rpos);
END;
```

Estas linhas apresentam um *statment* do tipo *FOR*, cuja estrutura é apresentada no apêndice de Mössenböck [3] e que neste caso repete quatro vezes o que estiver em seu interior. Aqui o procedimento *CRamp* chama o procedimento *RampFilter*, enviando o valor declarado de *Pos2* e os valores atuais de *rpos* que podem ser alterados durante a execução do módulo chamado.

Vamos verificar o que ocorre em cada uma das repetições do *loop* criado pelo *statement FOR*.

Os valores atuais de *rpos* são aqueles calculados em *InitRampFilter*.

Primeira repetição

A variável *i* possui o valor 0. O procedimento *RampFilter* do módulo *RampFilterLib* é chamado.

Primeira linha do corpo do procedimento RampFilter

```
IF c.count < c.steps THEN INC(c.count) END;
```

Como *rpos.count* é menor que *rpos.steps*, seu valor é incrementado em uma unidade. O procedimento pré-declarado *INC*, incrementa a variável entre parêntesis em uma unidade.

```
rpos.count := 1;
```

Segunda linha do corpo do procedimento RampFilter

```
IF new # c.old THEN
```

No processo de chamada, o identificador *new* assume o valor de *Pos2* definido nas constantes declaradas do módulo *CRamp* e é diferente de *rpos.old*, cujo valor atual é 0.0. Logo a seqüência após o *IF* deve ser executada.

Terceira a sétima linhas do corpo do procedimento RampFilter

São alterados os campos de *rpos*.

```
rpos.new := 1.0;
rpos.step := ( 1.0 - 0.0)/ ( 3+1); rpos.step:= 0.25;
rpos.start := 0.0 + 0.25; rpos.step := 0.25;
rpos.count := 0;
rpos.old := 1.0;
```

Oitava linha do corpo do procedimento RampFilter

No cabeçalho do procedimento *RampFilter* aparece, após as declarações, a palavra *REAL*, como visto a seguir.

```
PROCEDURE RampFilter* (new:  REAL; VAR c: RampType):REAL;
```

Isto significa que este procedimento retorna ao término de sua execução um valor do tipo *REAL* para o procedimento que o chamou.

Na oitava linha é feito exatamente isto. O valor atual de *rpos.start* é enviado para o procedimento *CRamp* através do *statment RETURN*. Neste procedimento, a variável *posdesej2[0]* assume este valor.

```
posdesej2[0] := 0.25;
```

As demais linhas não interessam nesta repetição uma vez que a condição *IF* já foi atendida.

Segunda repetição

A variável *i* possui o valor 1. O procedimento *RampFilter* do módulo *RampFilterLib* é chamado.

Primeira linha do corpo do procedimento RampFilter

Como *rpos.count* (zerado no ciclo anterior pela Sexta linha do procedimento *RampFilter*) é menor que *rpos.steps* (que continua com o valor 3.0, calculado pelo procedimento *InitRampFilter*), o valor de *rpos.count* é incrementado.

```
rpos.count := 1;
```

Segunda linha do corpo do procedimento RampFilter

Como *new* assume novamente o valor de *Pos2* e que agora é igual *rpos.old*, as linhas após *IF* não são executadas.

Nona linha do corpo do procedimento RampFilter

Como *rpos.count* é menor que *rpos.steps* o *statment ELSIF* também não é executado.

Décima linha do corpo do procedimento RampFilter

```
ELSE RETURN c.start + c.count * c.step
```

Retorna ao procedimento *CRamp* o valor de:

```
rpos.start+ rpos.count*rpos.step

0.25    +    1    *  0.25  = 0.5;
```

A variável *posdesej2[1]* assume este valor atual.

```
posdesej2[1] := 0.5;
```

Terceira repetição

A variável *i* possui o valor 2. O procedimento *RampFilter* do módulo *RampFilterLib* é chamado.

Primeira linha do corpo do procedimento RampFilter

```
rpos.count := 2;
```

Os requisitos da segunda e nona linhas novamente não são atendidos.

Décima linha do corpo do procedimento RampFilter

```
0.25+ 2*0.25 = 0.75;

posdesej2[2] := 0.75;
```

Quarta repetição

A variável *i* possui o valor 3. O procedimento *RampFilter* do módulo *RampFilterLib* é chamado.

Primeira linha do corpo do procedimento RampFilter

```
rpos.count := 3;
```

O requisito da segunda linha não é atendido.

Nona linha do corpo do procedimento RampFilter

```
ELSIF c.count >= c.steps THEN RETURN c.new
```

Como *rpos.count* = *rpos.steps* esta linha é executada e o procedimento retorna o valor de *rpos.new*.

```
posdesej2[3] := 1.0;
```

Terminada a quarta repetição passamos para as próximas linhas do procedimento *CRamp*.

```
FOR j := 0 TO 3 DO
        posdesej3[ j ] := RF.RampFilter(Pos3, rpos);
END;
```

Este *statment* é semelhante ao anterior e também faz com que se repita quatro vezes aquilo que está em seu interior.

Primeira repetição

A variável *j* possui o valor 0.

Primeira linha do corpo do procedimento RampFilter

Como *rpos.count* = *rpos.steps*, seu valor não é incrementado. Verificar que *rpos.count* permanece com o valor anteriormente calculado.

Segunda linha do corpo do procedimento RampFilter

Agora *new* assume o valor de Pos3. Como *rpos.old* é igual a 1.0, *new* é diferente de *rpos.old* e as linhas após o *IF* devem ser executadas.

Terceira a sétima linhas do corpo do procedimento RampFilter

```
rpos.new := 2.0;
rpos.step := ( 2.0 -1.0 ) / ( 3.0 +1); rpos.step := 0.25;
rpos.start := 1.0 + 0.25; rpos.start := 1.25;
rpos.count := 0;
rpos.old := 2.0;
```

Oitava linha do corpo do procedimento RampFilter

Retorna o valor para o procedimento *CRamp.*

```
Posdesej3[0] := 1.25;
```

Segunda, terceira e quarta repetições seguem de forma semelhante aos apresentados anteriormente.

No segundo ciclo,

```
posdesej3[1] := 1.5.
```

No terceiro ciclo,

```
posdesej3[2] := 1.75.
```

No quarto ciclo,

```
posdesej3[3] := 2.0.
```

Ao final do quarto ciclo está concluído o procedimento *CRamp.*

Como resultado obtivemos as variáveis abaixo descritas com seus respectivos valores atuais:

```
posdesej2[0] := 0.25;
posdesej2[1] := 0.5;
posdesej2[2] := 0.75;
posdesej2[3] := 1.0;
posdesej3[0] := 1.25;
posdesej3[1] := 1.5;
posdesej3[2] := 1.75;
```

```
posdesej3[3] := 2.0.
```

Está concluído o exemplo que realmente é bastante trivial mas fornece as noções básicas de programação e sintaxe da linguagem XOberon.

O apêndice do livro de Mössenböck [3] não é um manual completo sobre a linguagem mas pode servir como uma boa fonte de consulta para quem estiver programando em XOberon, principalmente quando surgirem dúvidas a respeito de sua sintaxe.

A.4 Particularidades do XOberon

Até agora vimos basicamente as características da linguagem Oberon que também se aplicam a linguagem XOberon. A seguir vamos estudar as características que conferem ao XOberon a qualidade de sistema operacional em tempo real e as ferramentas utilizadas para a geração de textos de saída e fornecimento de dados ao sistema.

Para isso devemos ler o 'Handbook - Denia/XOberon' [6] páginas 6 à 11.

A.4.1 Gerando texto de saída

Em relação a este tópico, a apostila está bastante clara. Cabe salientar que *XOberon.Log* é uma janela que normalmente está aberta, sendo bastante comum e conveniente gerar a saída do texto para esta janela.

Existe uma possibilidade de variação na geração de textos de saída. O módulo *Hello2* será rescrito, apresentando esta variação e a saída de dados será igualmente feita na tela *XOberon.Log*.

```
MODULE Hello2;

IMPORT XOberon, Xtexts;

VAR
        w : Xtexts.Writer;

PROCEDURE World;

BEGIN
        ASSERT (XOberon.Writer(w));
        XTexts.WriteString ( w, 'HelloWorld' );
        XTexts.WriteLn (w);
        XTexts.Append ( XOberon.Log(), w.buf);
```

```
END World;

BEGIN

END Hello2.
```

Apesar de não apresentados na apostila, os parêntesis após *XOberon.Log* são necessários em ambos os casos.

A linha *ASSERT (XOberon.Writer(w));* escrita no corpo do procedimento tem a mesma função da linha *XTexts.OpenWriter(w);* escrita no corpo do módulo apresentado na apostila. A forma utilizando *ASSERT* é mais freqüentemente encontrada nos módulos existentes do robô Inter.

WriteString e *WriteReal* são os procedimentos mais utilizados. Demais procedimentos para escrever outros tipos de números ou caracteres assim como procedimentos para mudança de atributos como cor e fonte estão descritos no livro "The Oberon Guide", capítulo 14, páginas 149 à 151.

A.4.2 Entrada de dados

Em XOberon, para executar um procedimento chamando-o externamente, usa-se normalmente o comando *XSystem.Call* seguido do nome do procedimento.

O *Scanner*, apesar de poder ser utilizado em qualquer texto especificado e de ter sua posição de início de leitura determinada, normalmente é utilizado para leitura dos dados escritos logo após o comando de chamada de um procedimento. Isto é feito pelas funções *XOberon.Par.text* e *XOberon.Par.pos* .

Supondo que queremos alterar os valores de *kp* e *kv* de um controlador do tipo PD. Para isso temos um procedimento chamado *ChangePara*, pertencente ao módulo *Change*, que altera os valores atuais de *kp* e *kv*, dada a junta e os novos valores desejados. Para alterar os valores da junta 1, o comando teria o seguinte formato.

```
XSystem.Call Change.ChangePara 1 500.0 100.0 ~
```

A linha

```
XTexts.OpenScanner(s, XOberon.Par.text, XOberon.Par.pos);
```

determina exatamente que o *Scanner* deva começar sua leitura de dados a partir do final do comando de chamada, que neste caso apresentado seria após a palavra *ChangePara*. Após a chamada, os valores seriam lidos e alocados nas variáveis desejadas.

Também existe uma variação para a utilização do *Scanner* freqüentemente encontrada nos programas do robô Inter.

Abaixo segue o módulo *TextIP* apresentado na apostila, rescrito com esta variação.

```
MODULE TextIP;

IMPORT XTexts, XOberon;

VAR
i1, i2 : LONGINT;

PROCEDURE Multiplication*;
VAR
        w : XTexts.Writer;
        s : XTexts.Scanner;
        ok : BOOLEAN;
BEGIN
        ASSERT(XOberon.Scanner(s));
        ok := TRUE;
        XTexts.ReadNextInt(s, i1, ok);
        XTexts.ReadNextInt(s, i2, ok);
        ASSERT(XOberon.Writer(w));
        IF ok THEN
```

A partir deste ponto os programas são semelhantes. Única diferença é que novamente não é necessário escrever no corpo do módulo a linha *XTexts.OpenWriter(w);* pois foi utilizado o comando *ASSERT*.

Neste exemplo, *ASSERT(XOberon.Scanner(s));* tem a mesma função de *XTexts.OpenScanner(s, XOberon.Par.text, XOberon.Par.pos);*, posicionando o *Scanner* logo após o comando de chamada deste procedimento.

A função *XTexts.ReadNextInt(s, i1, ok);* faz a leitura do próximo inteiro, já atribuindo à variável *i1* o valor lido. Também já é feita a conferência se o valor lido realmente é um inteiro. Se houver algum erro, a variável *ok* passa a assumir o valor *FALSE* e o próximo valor nem é lido.

Esta variante facilita o processo de leitura de dados com o *Scanner*.

Mais informações a respeito de *Scanner* podem ser obtidas no livro 'The Oberon Guide', capítulo 14.3, páginas 143 à 148.

A.4.3 Programando sistemas de tempo real

O capítulo 'Programming Real-Time Systems' da apostila [6] aborda de forma superficial o assunto programação em tempo real. O leitor deve ler atentamente as páginas 33 à 46 desta apostila que fornece uma explicação mais detalhada.

Os dois tipos de eventos mais utilizados na programação de sistemas em tempo real são o *Main Event* e *Every Event*. Portanto as explicações e exemplos serão direcionados para compreensão destes eventos.

O *Main Event* é utilizado para implementação de tarefas que são executadas somente quando houver tempo computacional disponível, isto é, quando não houver uma tarefa de tempo crítico sendo executada. Por isto são consideradas tarefas de tempo não crítico (NTC).

O *Every Event* implementa tarefas de tempo crítico (TC). Neste caso a tarefa tem de ser obrigatoriamente executada no tempo especificado.

Abaixo será apresentado um pequeno módulo que instala um *Main Event* e um *Every Event*.

```
MODULE InstalarEventos;

IMPORT
        XOK :=PPCXOKernel;

CONST
        clock = 5;

VAR
        ctrlEvent : XOK.EveryEvent;
        prinEvent : XOK.MainEvent;
        contador : REAL;

PROCEDURE Incrementador (e : XOK.Event);
VAR
BEGIN
        INC (contador);
END Incrementador;

PROCEDURE Principal ( e : XOK.Event);
```

```
VAR
BEGIN
        REPEAT  (* não faz nada *)
        UNTIL contador>1000.0;
        ctrlEvent.UnInstall;
END Principal.

PROCEDURE Instalar*;
VAR
BEGIN
        contador := 0.0;
        XOK.InitMain( prinEvent); prinEvent.Install ( Principal); prinEvent.Notify;
        XOK.InitEvery( ctrlEvent, clock, XOK.ONEms); ctrlEvent.Install ( Incrementador);
END Instalar;

BEGIN
        NEW(prinEvent);
        NEW(ctrlEvent);

END InstalarEventos.
```

Passaremos à análise deste módulo, explicando como se efetua a implementação de tarefas NTC e TC.

Iniciamos o módulo nomeando-o.

```
MODULE InstalarEventos;
```

É necessário importar o módulo *PPCXOKernel*. Neste módulo estão definidos os eventos. Não é necessário entender este módulo, devemos apenas saber usá-lo como uma ferramenta.

```
IMPORT

        XOK :=PPCXOKernel;
```

É definida a constante *clock*. Esta constante será posteriormente usada na especificação do tempo de clock do *ctrlEvent*.

```
CONST

        clock = 5;
```

Nas declarações é necessário especificar o tipo de evento a ser utilizado. Neste caso, *ctrlEvent* é do tipo *EveryEvent* e *prinEvent* é do tipo *MainEvent*, os quais são pré-definidos no módulo *PPCXOKernel*. Foi declarada também uma variável do tipo *REAL* que será utilizada no módulo.

```
VAR
        ctrlEvent : XOK.EveryEvent;
        prinEvent : XOK.MainEvent;
        contador : REAL;
```

Antes de entrarmos nos procedimentos, vamos verificar o corpo do módulo.

```
BEGIN
        NEW(prinEvent);
        NEW(ctrlEvent);

END InstalarEventos.
```

O corpo do módulo é executado quando o programa é carregado no sistema. Caso o módulo ainda não esteja carregado e algum procedimento deste módulo for chamado, as declarações e o corpo do módulo serão carregados antes da execução do procedimento. Ao utilizar o comando *NEW*, estamos alocando espaço na memória para executar estes eventos.

Desta forma, os eventos acima descritos com certeza terão seu espaço reservado na memória antes da execução de qualquer procedimento deste módulo. Também seria possível fazer esta alocação a partir do procedimento *Instalar*. Neste caso teríamos de ter o cuidado de usar este procedimento uma única vez enquanto o módulo estiver carregado. Do contrário, cada vez que acionássemos o procedimento *Instalar*, reservaríamos desnecessariamente um novo espaço na memória para estes eventos.

Iniciaremos analisando o módulo *Instalar*. Não houve necessidade de declarações. Neste caso a palavra *VAR* poderia ser suprimida.

```
PROCEDURE Instalar*;
VAR
BEGIN
```

Na primeira linha de seu corpo a variável contador é inicializada.

```
contador := 0.0;
```

A seguir é feita a implementação da tarefa associada ao evento *prinEvent*.

```
XOK.InitMain( prinEvent); prinEvent.Install ( Principal); prinEvent.Notify;
```

Esta seqüência é obrigatória e se repete cada vez que tivermos de instalar um *Main Event*. É importante identificar corretamente o nome do evento e da tarefa. Neste caso o nome do evento é *prinEvent* e o nome da tarefa é *Principal*.

Somente quando fazemos *prinEvent.Notify* a tarefa *Principal* passa a ser executada.

A seguir é instalada a tarefa associada ao evento *ctrlEvent*.

```
XOK.InitEvery( ctrlEvent, clock, XOK.ONEms); ctrlEvent.Install ( Incrementador);
END Instalar;
```

Após o comando *XOK.InitEvery*, devemos especificar dentro dos parêntesis e nesta ordem, nome do evento, tempo de execução e base de tempo.

O tempo deverá ser sempre um número inteiro. Ao invés de utilizar a constante pré-definida *clock*, poderíamos digitar diretamente um valor para o tempo desejado de cada ciclo.

```
XOK.InitEvery( ctrlEvent, 5, XOK.ONEms);
```

As bases de tempo existentes são:

- *ONEsec*, equivalente a um segundo;

- *ONEms*, equivalente a um milisegundo;

- *ONEts*, equivalente a um décimo de milisegundo.

Neste módulo, como *clock* é igual a cinco e a base de tempo usada é *ONEms*, a tarefa associada ao evento será executada a cada 5 milisegundos.

O comando *ctrlEvent.Install* instala a tarefa *Incrementador*. Não é necessário utilizar *Notify*. A partir deste ponto as duas tarefas estão em funcionamento.

Vamos verificar o que ocorre em *Incrementador*. Ao escrever a tarefa deverá obrigatoriamente aparecer *(e : XOK.Event)* sendo que a variável *e* pode ser substituída por qualquer outra variável desejada.

Esta tarefa se repete obrigatoriamente a cada cinco milisegundos e o que ela faz é simplesmente incrementar o valor de *contador* em uma unidade a cada ciclo. Sua execução irá terminar somente quando for desinstalada por algum outro procedimento

```
PROCEDURE Incrementador (e : XOK.Event);
VAR
BEGIN
        INC (contador);
END Incrementador;
```

Simultaneamente à tarefa *Incrementador* o sistema estará executando a tarefa *Principal*. Simultaneamente talvez não seja a palavra correta pois o sistema somente irá executar esta tarefa quando houver disponibilidade de tempo. A tarefa *Incrementador* nunca é interrompida antes de chegar ao final de um ciclo. Já a tarefa *Principal* pode ser interrompida em qualquer ponto de sua execução se houver necessidade de executar um *Every Event*.

Vamos verificar o que ocorre em *Principal*.

```
PROCEDURE  Principal ( e : XOK.Event);
VAR
BEGIN
        REPEAT  (* não faz nada *)
        UNTIL contador>1000.0;
        ctrlEvent.UnInstall;
END Principal.
```

Novamente é necessário escrever *(e:XOK.Event)*.

Ao contrário do *Every Event*, o *Main Event* por sua definição seria executado somente uma vez. Se quisermos que seja executado mais de uma vez temos de criar algum tipo de *loop* interno. Neste caso foi utilizado *REPEAT - UNTIL*, mas outras formas também são possíveis como por exemplo *LOOP - EXIT*.

Utilizando *REPEAT - UNTIL* a tarefa irá se repetir até que uma condição seja satisfeita.

Neste programa a condição é que a variável *contador* seja maior que mil. Enquanto esta condição não for satisfeita, a tarefa irá se repetir sem fazer nada, pois não temos nenhuma operação ou comando a ser executado.

Como descrito anteriormente, a tarefa *Incrementador* incrementa a variável *contador*. Quando esta variável atingir o valor 1001, a tarefa *Principal* sai de seu loop e antes de ser terminada chama o evento *ctrlEvent* e desinstala a tarefa a ele associada, que neste caso é *Incrementador*.

O *Main Event* não pode ser desinstalado enquanto estiver no interior de um loop. Sua execução irá terminar quando alguma condição for satisfeita. Já o *Every Event* necessita ser desinstalado.

Para iniciar estas tarefas utilizaríamos por exemplo comando:

```
XSystem.Call InstalarEventos.Instalar ~
```

Se o módulo *InstalarEventos* ainda não estivesse carregado, seriam executados inicialmente suas declarações e seu corpo.

Resumindo o que ocorre após a chamada do procedimento *InstalarEventos*.

1- as duas tarefas são instaladas.

2- a tarefa *Incrementador* começa a incrementar o valor de *contador*.

3- quando a variável *contador* atingir 1001, a tarefa *Incrementador* é desinstalada e a tarefa *Principal* é terminada.

Para rodar novamente, bastaria repetir o comando de chamada pois no procedimento *Instalar* a variável *contador* é zerada.

Esta é a idéia geral da utilização de *Main Events* e *Every Events* que são os eventos principais na programação de sistemas em tempo real.

O leitor pode questionar porque utilizar um Main Event ao invés de utilizar um procedimento normal. Existem duas razões principais.

A primeira é que o *Main Event* possui prioridade de execução sobre procedimentos normais.

A segunda é que quando a execução de um programa fica presa no interior de um loop dentro de um Main Event, é possível fazer chamadas externas para a execução de outros comandos. No entanto, quando ficamos presos no interior de um loop de um procedimento normal, não é possível executar outros comandos. Temos de esperar que o loop termine.

A.5 Configurando e Utilizando Objetos

Muitos módulos utilizados pelo robô Inter estão escritos em programação orientada para objetos. Não é intenção deste capítulo explicar esta metodologia de programação pois existe uma grande quantidade de livros sobre o assunto. O leitor que não tiver nenhum conhecimento sobre programação orientada para objetos deverá inicialmente ler os capítulos 1 a 4 do livro Object-Oriented Programming in Oberon-2.

A.5.1 Configurando a Placa, Sensores, Atuadores e Encoders

Em XOberon, além de configurar os objetos utilizados na movimentação e controle do robô, devemos também configurar os periféricos como sensores e atuadores. Veremos superficialmente esta configuração de periféricos pois após o robô estar em funcionamento, dificilmente será necessário alterá-la.

Mostraremos a seguir os principais comandos utilizados na configuração do microprocessador, dos sensores, atuadores e contadores do robô.

Iniciaremos definindo o microprocessador. O comando

```
IPCarrier.DefBoard VIP610 AT 0EDFF6000H
```

define um microprocessador do tipo VIP610 no endereço de memória 0EDFF6000H.

A seguir definimos os tipos de sinais existentes, relacionando-os com um endereço no microprocessador. O comando

```
IPPrec.DefMod AInMod BOARD VIP610 AT 0H CHANNELS 6
```

define seis canais de entrada analógica de dados, começando a partir do endereço 0 no microprocessador VIP610.

A definição dos canais das saídas analógicas de dados, da interface digital e dos contadores é feito de maneira semelhante.

```
IPDAC.DefMod AOutMod BOARD VIP610 AT 100H CHANNELS 6

IPDig48.DefMod DigitalMod BOARD VIP610 AT 300H CHANNELS 48

IPQuad.DefMod CountMod BOARD VIP610 AT 200H CHANNELS 4.
```

Feita a definição dos tipos de sinais, podemos configurar os objetos individualmente. Todos os objetos do tipo sensor, atuador e contador são definidos no módulo *Peripherals.Mod.* Ao definir algum destes objetos devemos fornecer um nome, tipo, canal e, conforme o caso, uma escala.

O comando

```
Peripherals.DefDActuator Break0 ON DigitalMod CHANNEL 17
```

define um atuador de nome *Break0* do tipo digital no canal 17.

A escala é utilizada por exemplo no caso de contadores como visto no capítulo 2. A escala a nível de configuração nos permite por exemplo trabalhar diretamente com radianos para definir posição das juntas sem nos preocuparmos com todo processo de conversão AD do sinal fornecido pelo *encoder.*

O comando

```
Peripherals.DefCounter Counter0 ON CountMod CHANNEL 0 SCALE countScale0
```

define um contador de nome *Counter0* do tipo contador no canal 0 com escala *countScale0.*

Toda definição de periféricos é feita no arquivo *ROBAllBoot.Config* do diretório *INTER* do robô.

A.5.2 Configurando um Objeto

A maioria dos objetos utilizados no robô também são configurados no arquivo *ROBAllBoot.Config* enviado ao microprocessador no momento em que o sistema é ligado.

Os objetos configurados devem ser uma extensão de *Objects* definido no módulo *Base* apresentado a seguir:

```
TYPE
        Object*=POINTER TO ObjDesc;
        ObjDesc*=RECORD(Objects.ObjDesc)
                PROCEDURE (obj : Object; name: ARRAY OF CHAR) : BOOLEAN;
        END;
```

Ao fazermos com que este objeto seja uma extensão de *Objects.ObjDesc*, podemos também utilizar suas propriedades que são:

- retirar e colocar objetos no banco de dados;
- possibilidade de inspeção dos dados.

Para criar um objeto necessitamos obrigatoriamente dos procedimentos *DefObject*, *NewObject* e do método *Assign*.

Mostraremos a seguir como escrever estes procedimentos, criando o objeto *CarthKin*, usado na determinação dos parâmetros cinemáticos do robô.

Iniciamos com a definição do tipo *CarthKin*.

```
TYPE
        CarthKin* = POINTER TO CarthKinDesc;
        CarthKinDesc* = RECORD (Base.ObjDesc)
                z0*, l1*, l2*, l111*, l212*, l112*, rMin*, rMax* : LONGREAL;
                d* : ARRAY 4 OF Drive.Drive;
                minq-, maxq- :ARRAY 4 OF REAL;
        END;
```

CarthKin é um ponteiro para *CarthKinDesc* que por sua vez é extensão de *Base.ObjDesc*, de forma que possamos nos utilizar de suas propriedades.

Dos novos campos introduzidos, alguns são do tipo *LONGREAL*, outros são vetores de quatro elementos reais e outro é um vetor de quatro elementos do tipo *Drive*.

Este tipo *Drive* é um objeto previamente definido e que será visto em maiores detalhes em capítulo posterior.

Feita a definição do tipo *CarthKin*, prosseguimos escrevendo o procedimento básico para configurar o objeto.

```
PROCEDURE DefCarthKinematic*;
VAR k : CarthKin; ok :=BOOLEAN;
BEGIN
        NewCarthKinematic; k:=O.NewObj(CarthKin);
        ok := Parser.Parse(k) ;
END;
END DefCarthKinematic;
```

Esta estrutura se repete toda vez que formos configurar um objeto. É importante identificar corretamente o nome do objeto que neste caso é *CarthKin* e que deve concordar com o nome do tipo anteriormente definido.

Na primeira linha do corpo é chamado o procedimento *NewCarthKinematic*, cuja estrutura é apresentada a seguir.

```
PROCEDURE NewCarthKinematic*;
VAR k : CarthKin;
BEGIN
        NEW(k); k.handle:=Handler;
        O.NewObj:=k;
END NewCarthKinematic;
```

Este comando gera um novo objeto, alocando um espaço na memória. A variável *k* tem de ser do tipo *CarthKin*. Esta é a versão mais breve possível de um comando para gerar um objeto.

Concluído o procedimento *NewCarthKinematic*, retornamos ao procedimento *DefCarthKinematic* que executa o comando *Parser.Parse(k)*. Este comando chama o método *Assign* apresentado a seguir, já enviando os parâmetros formais necessários.

```
PROCEDURE (k : CarthKin) Assign* (o : Base.Object; name : ARRAY OF CHAR) :
BOOLEAN;
    VAR    value : Base.Value; a : LONGINT;
    BEGIN
        IF (o=NIL) THEN RETURN FALSE
        ELSIF (o IS Base.Value) THEN    value:=o(Base.Value);
            IF  name = 'Z0' THEN
                k.z0:=value.x;
            ELSIF name = 'L1' THEN
                k.l1:=value.x;
            ELSIF name = 'L2' THEN
                k.l2:=value.x;
            ELSE RETURN k.Assign^(o, name);
            END
        ELSIF o IS Drive.Drive THEN
            IF name = 'drive0' THEN k.d[0]:=o(Drive.Drive)
            ELSIF name = 'drive1' THEN k.d[1]:=o(Drive.Drive)
            ELSIF name = 'drive2' THEN k.d[2]:=o(Drive.Drive)
            ELSIF name = 'drive3' THEN k.d[3]:=o(Drive.Drive)
            END;
        ELSE    RETURN k.Assign^ (o, name)
        END; RETURN TRUE;
END Assign;
```

O método *Assign* estabelece a correspondência entre o nome fornecido na configuração e a variável que deve assumir o valor atribuído a este nome.

Veremos como é a linha referente a configuração deste objeto, pertencente ao arquivo *ROBAllBoot.Config*.

```
SCARACarthKin.DefCarthKinematics SCCKin (Z0=0.665, L1=0.25, l2=0.25,
drive0=Drive0, drive1=Drive1, drive2=Drive2, drive3=Drive3);
```

Nesta configuração o comando *DefCarthKinematics* é chamado e toda a seqüência apresentada é executada, inclusive o método *Assign*. O método *Assign* lê cada nome e seu valor correspondente escritos dentro dos parêntesis e faz com que a campo desejado de *Carthkin* assuma este valor.

Nas linhas

```
IF  name = 'Z0' THEN
```

```
k.z0:=value.x;
```

a variável *k.z0* assume o valor 0.665.

Se todo método *Assign* pode ser executado sem nenhum tipo de conflito, o comando *Parser.Parse* assume o valor booleano *TRUE* e a configuração foi completada.

Ao final deste processo temos um objeto do tipo *CarthKin* de nome *SCCKin* e com os campos:

- z0 = 0.665;
- l1 = 0.25;
- l2 = 0.25;
- d[0] = Drive0;
- d[1] = Drive1;
- d[2] = Drive2;
- d[3] = Drive3;

Estes dados estão numa biblioteca e ainda não podem ser utilizados por outros módulos. Mostraremos como ter acesso a estes dados, apresentando o módulo a seguir.

```
MODULE Get;
        (* utilizar dados do objeto SCCKin *)

IMPORT
        SCK := SCARACarthKin, O :=Objects, Base;

VAR
        kin : SCK.CarthKin;

PROCEDURE GetScaraCarthKin (name : ARRAY OF CHAR; VAR k: SCK.CarthKin) :
BOOLEAN;

VAR
        obj: Base.Object;
BEGIN
        Base.GetObj( name, obj);
        IF (obj = NIL) OR ~(obj IS SCK.CarthKin) THEN
                RETURN FALSE;
        ELSE k := obj(SCK.CarthKin);
                RETURN TRUE;
        END;
END GetScaraCarthKin;

PROCEDURE Init;
VAR
```

```
        multi : REAL;
BEGIN
        kin.l111:=kin.l1*kin.l1:
        kin.l212 := kin.l2*kin.l2;
        kin.l112 := kin.l1*kin.l2;
        multi := kin.l1*kin.l2/kin.z0
END Init;

PROCEDURE Configuration;
VAR
kinConfigurated : BOOLEAN;
BEGIN
        kinConfigurated := GetScaraCarthKin('SCCKin', kin);
        IF kinConfigurated THEN Init
        ELSE HALT(120);
        END;
END Configuration;

BEGIN
        Configuration;

END Get.
```

No momento que este módulo é carregado, o procedimento *Configuration* é automaticamente executado. Este procedimento chama o procedimento *GetScaraCarthKin* que busca da biblioteca o objeto de nome *SCCKin* do tipo *CarthKin*. Inicialmente é feita a conferência se existe um objeto na biblioteca de nome *SCCKin* e se este objeto é do tipo *SCK.CarthKin*. Se estiver correto, a variável *kin* assume os valores de *SCCKin*. Na verdade, estamos criando um ponteiro de nome *kin* que aponta para este objeto *SCCKin*. Para utilizar os vários campos deste objeto devemos utilizar agora o nome do ponteiro. No procedimento *Init* por exemplo, alguns campos do objeto *SCCKin* são utilizados com o nome do ponteiro *kin* para o cálculo de expressões.

A configuração de objetos na linguagem XOberon não é muito simples e infelizmente não temos informações precisas sobre o que cada linha destes procedimentos faz exatamente. O que normalmente fazemos quando queremos criar um novo objeto é utilizar uma configuração de um objeto já existente e adequar esta configuração a este novo objeto.

Desta forma concluímos o exemplo sobre configuração de objetos. Existem variações de acordo com o tipo de objeto a ser configurado, mas a idéia básica permanece a mesma.

APÊNDICE B

O ROBÔ EM FUNCIONAMENTO

B.1 Introdução

Este capítulo descreve as ações necessárias para colocar o robô em funcionamento, abrangendo desde o estabelecimento da comunicação entre o microcomputador (Host) e o microprocessador (Target) até os comandos utilizados para movimentá-lo.

B.2 Estabelecendo a Comunicação.

O Host é um microcomputador utilizado para fazer a interface com o robô. O software XOberon deve ser instalado neste microcomputador. Não necessita de grande capacidade de processamento pois todos os módulos são carregados no microprocessador (Target) no momento que o armário de controle é ligado. O Host funciona apenas para enviar comandos ao Target que se encarrega de executá-los. O Target do robô Inter é um microprocessador PowerPC de 200MHz de clock.

A comunicação entre Host e Target pode ser feita ponto a ponto ou via Ethernet. A comunicação ponto a ponto é mais segura, mas optou-se durante a instalação do robô por utilizar a comunicação via Ethernet de forma a poder continuar fazendo proveito dos benefícios da rede.

Qualquer micro que esteja ligado na rede possui obrigatoriamente uma placa Ethernet e um IP number, fornecido pelo administrador da rede. O Target esta instalado na rede do LCMI e possui o IP number 150.162.14.204. Atualmente está sendo utilizado o microcomputador ARARA como Host, cujo IP number é 150.162.14.201. O procedimento a seguir descrito para estabelecer a comunicação usa como exemplo os IP numbers acima referenciados. Este procedimento somente precisará ser refeito se quisermos alterar o micro utilizado como Host ou se for

alterado o IP number do Target. Na verdade, qualquer microcomputador da rede pode acessar o Target. Este procedimento de ajustar os IP numbers é feito para que o Target saiba onde procurar o bootfile no momento em que o armário de controle é ligado. O bootfile está atualmente no diretório tftpboot do microcomputador ARARA e possui o nome Megula.

O primeiro passo para executar este ajuste de IP numbers é pendurar um terminal no Target. Para isto deve-se usar a porta Serial1 do Target, identificada em sua parte posterior, situada no interior do armário de controle. Esta porta possui 9 pinos, sendo que o 2 é para transmissão, o 3 para recebimento e o 5 é o terra. No terminal, que pode ser o próprio Host, normalmente encontramos uma porta serial de 9 ou 25 pinos e os canais utilizados são o 2 e o 3 para transmissão e recebimento e o 5 para terra no de nove pinos e o 7 para terra no de 25 pinos. Em alguns micros é possível que seja necessário inverter canais 2 e 3 .

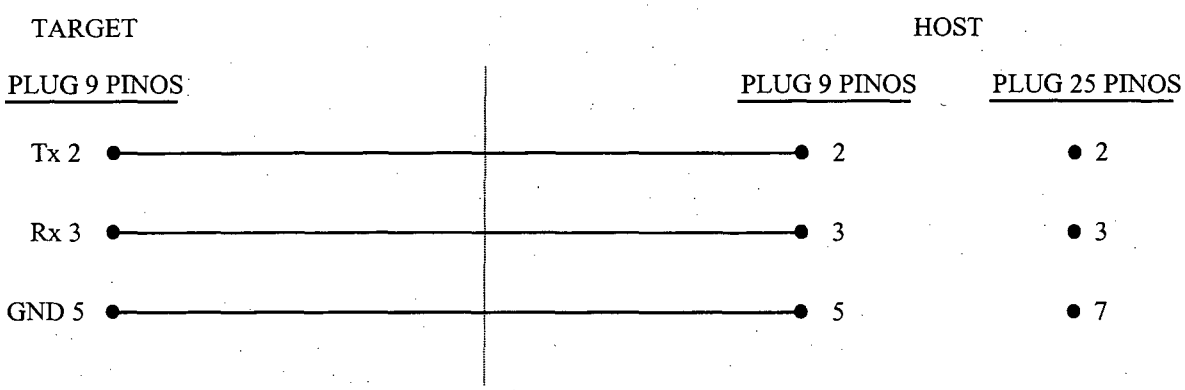


FIGURA B.1 - Diagrama para estabelecer comunicação com o Target.

Ajustado o cabo serial, abrir um HyperTerminal no Windows95. Clicar em qualquer dos ícones e fornecer qualquer nome. Informar qual porta iremos utilizar. No microcomputador ARARA utilizamos a porta COM2. Automaticamente surge PortSettings, onde devemos utilizar os seguintes dados:

- 9600 ;
- 8 bit;
- 1 stoppbit;
- Parity None;
- Hardware;.

Surge então uma tela vazia e o microprocessador deve ser resetado. Durante o processo de reset, apertar repetidamente a tecla Esc para estabelecer a comunicação. Se a comunicação foi estabelecida, surge o prompt PPC1-Bug>. Estamos no Bug do Target.

Devemos agora setar os IPnumbers. Para isso digitamos niot e ENTER.

Surge uma série de informações e devemos rolar até encontrar os campos mostrados a seguir:

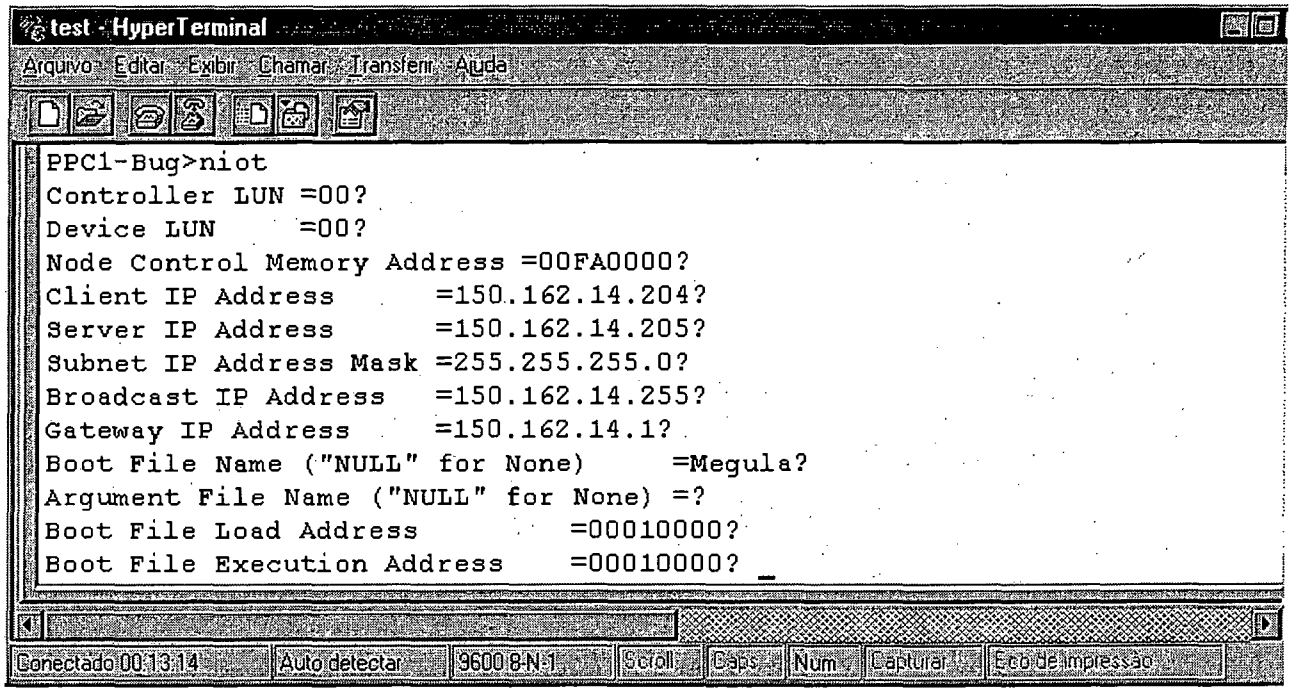


FIGURA B.2 - Tela do Hyper Terminal.

Os campos de interesse são:

- Client IP Address: 150.162.14.204 (IP number do Target);
- Server IP Address: 150.162.14 201 (IP number do Host);
- Subnet IP Address Mask: 255.255.255.0 (fornecido pelo administrador da rede);
- Broadcast IP Address: 150.162.14.255 (fornecido pelo administrador da rede);
- Gateway IP Address: 150.162.14.1 (fornecido pelo administrador da rede);
- Boot File Name: Megula (nome do arquivo de onde será feito o Boot);

Estes campos devem ser verificados e conforme a necessidade alterados. Feito isto podemos desconectar o terminal e desligar o Target.

O arquivo de configuração Megula deve ser carregado automaticamente toda vez que ligarmos o microprocessador. Este arquivo é enviado para o Target via TFTP.

Temos portanto de inicialmente abrir um servidor TFTP. Atualmente estamos usando o TFTP Server 95, situado no diretório Tftpd. Ao abrir a tela de TFTP, na opção *Configure* devemos determinar o diretório de onde será enviado o bootfile para o Target. No momento está em C:\tftpboot. Se por algum motivo qualquer quisermos alterar o local de armazenamento do bootfile, devemos também alterar o campo *Configure* do TFTP.

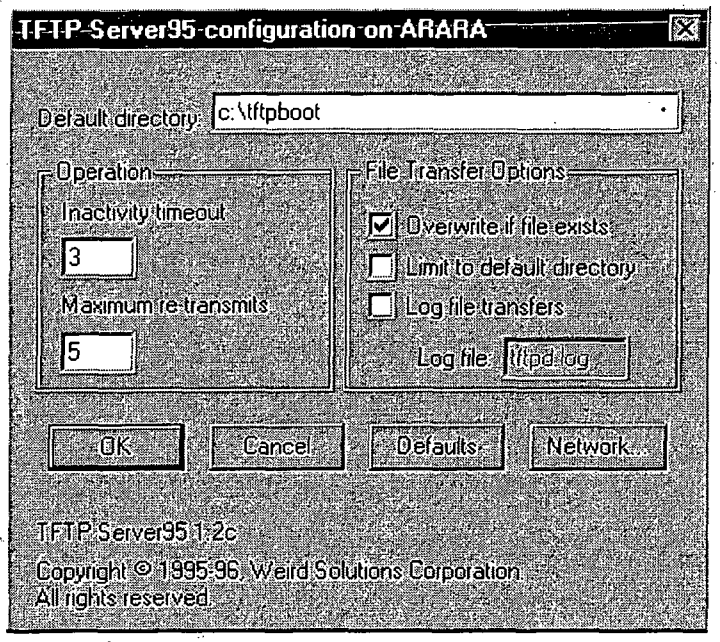


FIGURA B.3 - Configuração do TFTP Server95.

Após abrir o programa de TFTP e estabelecer as informações na opção *configure* podemos ligar o armário de controle. Na tela do programa de TFTP aparece a situação da transferência do arquivo Megula. Se não ocorreu nenhum problema ao carregar o bootfile, o status no programa de TFTP deve ser Complete e deve acender uma pequena luz verde situada na porção central do armário de controle. Como este arquivo é carregado via Ethernet, as vezes ocorrem falhas na transferência. Nestes casos aparece uma mensagem de erro na tela no programa de TFTP. Se isto ocorrer basta resetar o Target para fazer nova tentativa. Se após algumas tentativas persistir o erro, provavelmente o problema não está na rede e os IP numbers especificados devem ser conferidos. Também é importante verificar se o caminho para buscar o bootfile está correto.

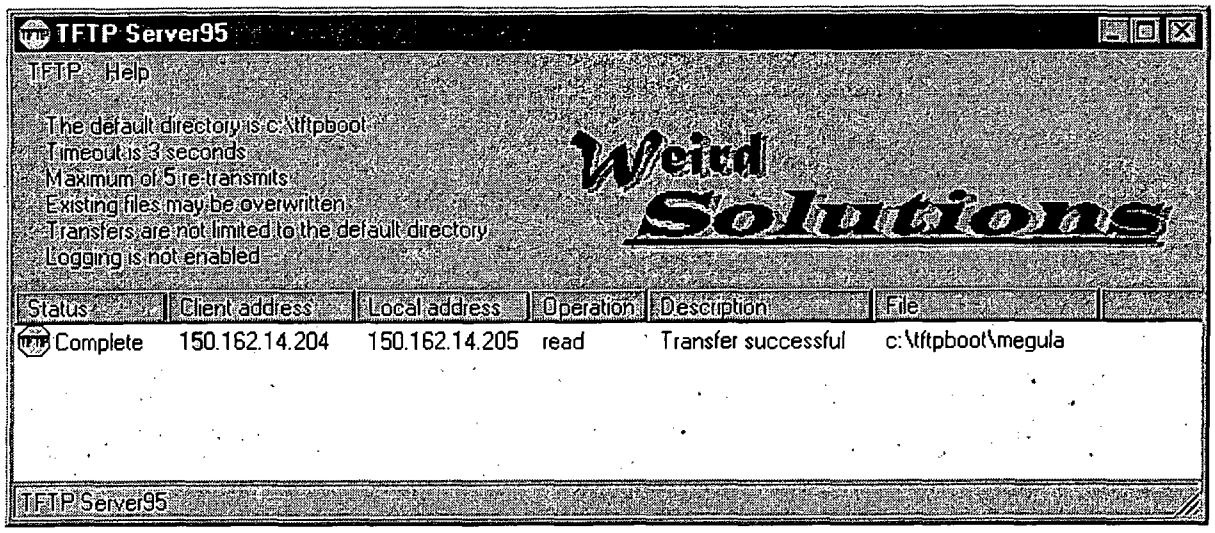


FIGURA B.4 - Tela principal do TFTP Server95.

No bootfile já existe um comando que executa o procedimento *Init* do módulo *Main3*. Este procedimento já instala as tarefas *MAIN* e *Controller*. A luz verde que aparece na região central do armário de controle é o sinal que indica o funcionamento do *WatchDog*.

Além dos processos instalados para o funcionamento do robô, existem uma série de processos que ocorrem simultaneamente e que são importantes na comunicação e no funcionamento do XOberon. Estes processos são automaticamente instalados quando ligamos o armário de controle.

B.3 XOberon Panel e Robot Panel

A seguir temos de iniciar o programa XOberon. O XOberon normalmente exige uma tela grande para comportar todas as informações abertas simultaneamente. Como nem sempre um monitor grande está disponível, o ambiente de trabalho do XOberon pode ser dividido em quatro partes mas somente uma delas fica visível no monitor. Ao iniciar o XOberon surge a tela a seguir apresentada.

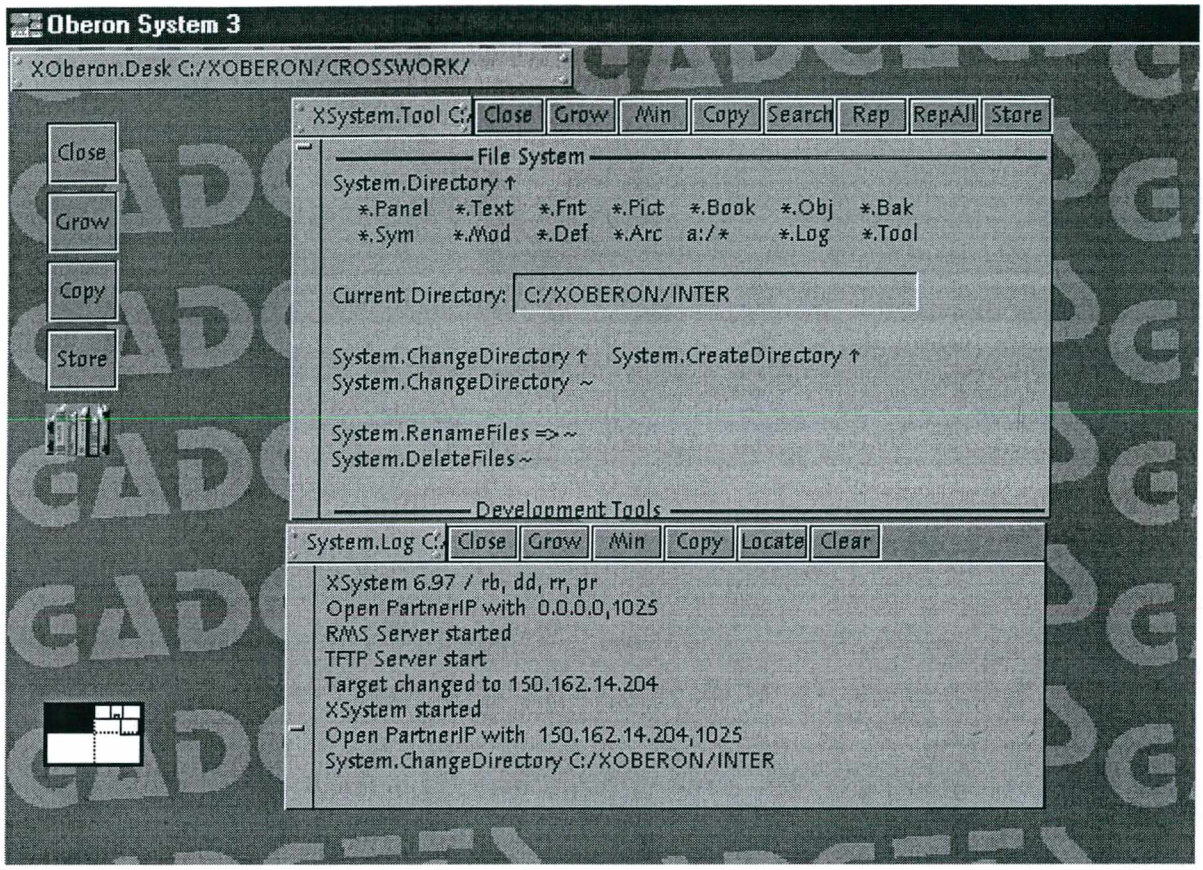


FIGURA B.5 - Tela do XOberon.

Na porção inferior esquerda existe um pequeno quadrado dividido em 4 chamado navigator. É através dele que nos movimentos pelo ambiente do XOberon, necessitando apenas clicar na porção que se deseja visualizar na tela. Ao clicar na porção superior direita do navigator surge o Robot.Panel e o XOberon.Panel.

XOberon.Panel e Robot.Panel são ferramentas que facilitam a execução de certos comandos. Cada botão representa um atalho para executar um comando. Por exemplo, quando clicamos em Call estamos na verdade executando o comando XSystem.Call.

No XOberon.Panel existem uma série de comandos utilizados para obter informações a respeito do funcionamento do Target. Para saber qual o Target que deve ser contatado, devemos escrever o IP number de nosso microprocessador obrigatoriamente entre aspas no quadro abaixo dos botões Start e Stop e pressionar a tecla ENTER. Para efetivar a escolha do Target e poder iniciar a troca de informações entre o Host e o Target, basta clicar o botão Start do XOberon.Panel. Ao fazer isto o Host sabe de onde buscar e para onde mandar as informações ou comandos

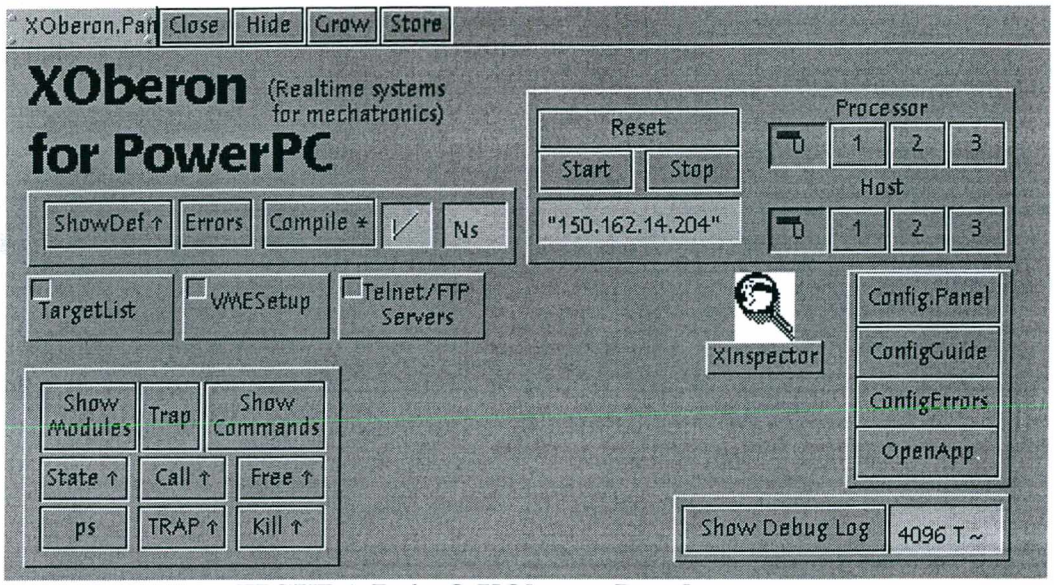


FIGURA B.6 - O XOberon Panel.

Para verificar a lista de processos em andamento logo após o robô ter sido ligado, devemos clicar o botão ps do XOberon.Panel. Deverá surgir uma tela auxiliar com o IP number do Target com a listagem de todos processos. Sempre que iniciarmos uma nova sessão, devemos acionar ps para que esta tela auxiliar apareça pois todas as mensagens enviadas do Target para o Host são escritas nesta tela, que deverá permanecer aberta enquanto durar a sessão.

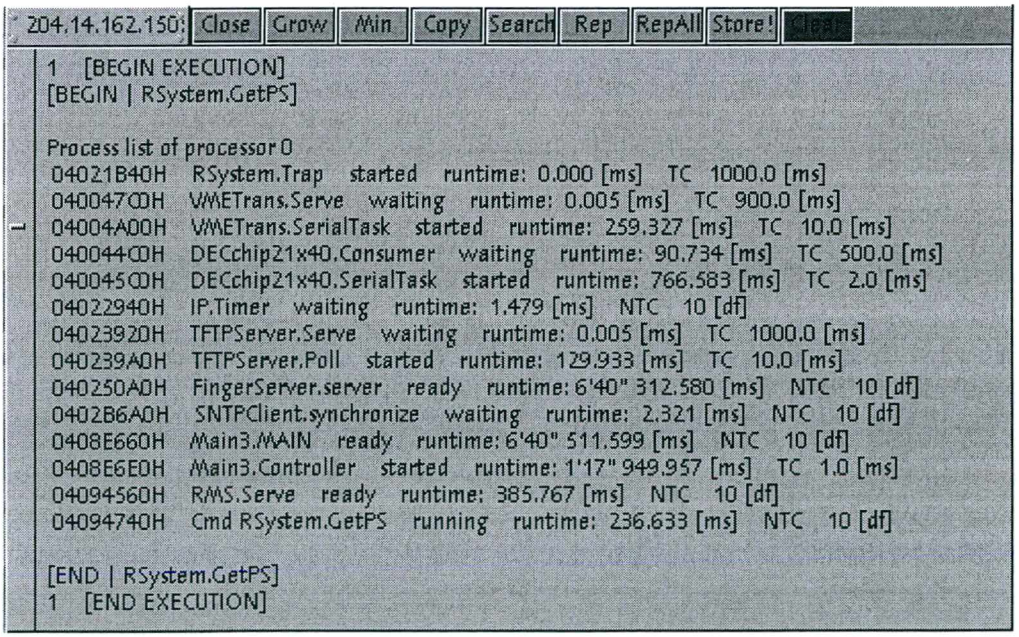


FIGURA B.7 - Processos instalados quando o armário de controle é ligado.

Na listagem de processos podemos ver que *MAIN* e *Controller* do módulo *Main3* estão ativos. Algumas informações adicionais podem ser conferidas como o

tipo de processo (TC ou NTC), tempo total de execução, status e clock para os processos TC.

Outros comandos bastante utilizados do XOberon.Panel são Show Modules, Show Commands, State, Call, Free e Kill.

Enquanto o XOberon.Panel trabalha com os comandos do XOberon, o Robot.Panel trabalha com os comandos para movimentação, controle e verificação do estado do robô.

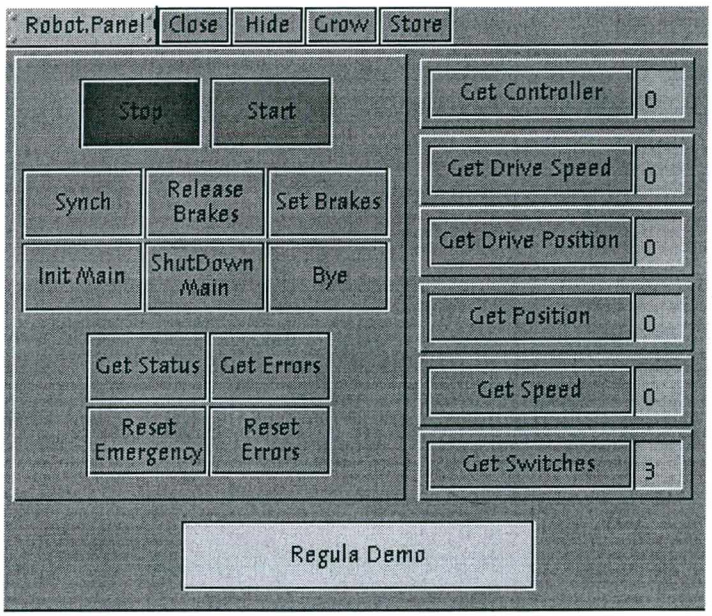


FIGURA B.8 - O Robot Panel.

Após conferir os processos, devemos verificar o estado do robô antes de executar qualquer outro comando. Para isto, clicamos no botão Get Status do Robot Panel e deverá aparecer na tela o seguinte:

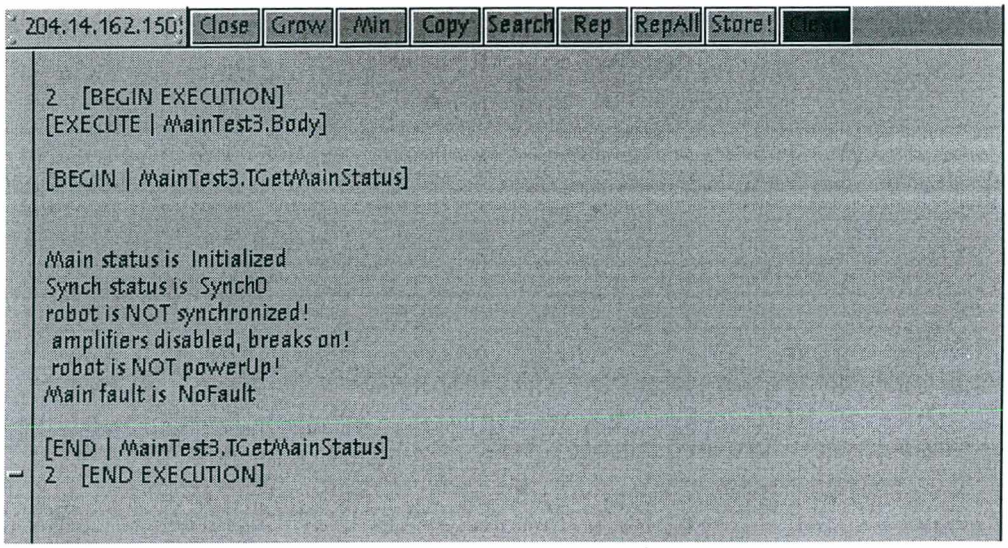


FIGURA B.9 - O estado inicial do robô.

Este comando fornece todas as informações a respeito do estado atual do robô. Logo que o robô é ligado deve obrigatoriamente se encontrar nesta situação.

Main status is Initialized significa que o estado do MAIN é Initialized.

Synch status is Synch0 significa que o estado da sincronização é Synch0.

Robot is NOT synchronized significa que o robô não está sincronizado.

Amplifiers disabled, breaks on significa que os amplificadores estão desabilitados e os motores estão freiados.

Robot is NOT powerUp significa que o botão de Power Up não está acionado.

Main fault is NoFault significa que não foi identificado nenhum erro no funcionamento do robô.

Verificado o estado do robô podemos fazer Power Up apertando o botão verde no armário de controle e iniciar o processo de sincronização clicando o botão Synch do Robot.Panel. Durante o processo de sincronização o robô executa uma série de movimentos até parar na posição especificada para o final do processo de sincronização. No final do processo de sincronização deve aparecer na tela a mensagem Synch End.

Após o processo de sincronização o robô se encontra no estado Stop. Para poder iniciar qualquer tipo de movimento devemos clicar o botão Start que habilita os amplificadores e solta os freios.

Feito isso, o robô está apto a realizar os movimentos desejados. Clicando no botão RegulaDemo por exemplo, o robô executa uma série de movimentos pré-programados.

Faremos um pequeno resumo das ações necessárias para colocar o robô em funcionamento, excluindo aquelas executadas apenas uma vez durante a instalação do robô, as quais garantem que o bootfile seja carregado corretamente.

1. Abrir o Programa TFTP Server.

2. Ligar o Armário de Controle.

3. Aguardar enquanto o bootfile está sendo carregado. O término ocorre quando surgir Complete na tela do programa TFTP Server e aperecer no armário de controle a luz verde que indica a presença do WatchDog.

4. Abrir o Programa XOberon.

5. Clicar o Botão de Start no XOberon.Panel, verificando se o IP number do TARGET já está correto.

6. Verificar os processos clicando no botão ps do XOberon.Panel. Surge uma tela com o IP number do Target.

7. Verificar o estado do robô clicando no botão GetStatus do Robot.Panel.

8. Apertar o botão de Power Up no armário de controle.

9. Sincronizar o robô, clicando no botão Synch do Robot.Panel.

10. Após o processo de sincronização, levar o robô para o estado Run, clicando no botão Start do Robot.Panel.

11. Executar os movimentos desejados.

B.4 As Ferramentas de Comandos.

Para executar movimentos utiliza-se normalmente os comandos do módulo BPTest. Para evitar que tenhamos de escrever todo o comando a cada vez que quisermos executá-los, é conveniente que criemos bibliotecas de comandos. Estas bibliotecas são criadas com a terminação Tool. Atualmente utilizamos BPTest.Tool e MainTest3.Tool. A biblioteca MainTest3 se refere mais a comandos que alterem o estado do robô e que forneçam informações a respeito do movimento e sensores. Já a biblioteca BPTest é utilizada para a movimentação do robô. Apresentamos a seguir alguns comandos desta biblioteca e exemplificaremos sua utilização.

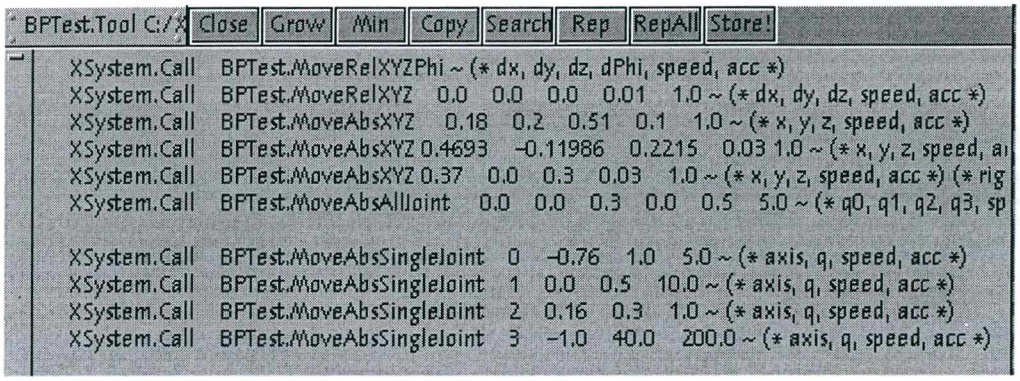


FIGURA B.10 - Principais comandos de movimentação.

Ao fazer XSystem.Call BPTest.MoveAbsSingleJoint estamos executando o procedimento MoveAbsSingleJoint do módulo BPTest. Este procedimento lê os dados compreendidos entre o final do comando e o sinal ~. Após este sinal são feitos comentários indicando quais os dados a serem fornecidos. Neste caso apresentado deve ser fornecido o elo que se deseja movimentar, a posição para a qual queremos movimentar este elo, a velocidade de deslocamento e a aceleração.

Estes dados podem ser facilmente alterados através dos comandos normais de edição e novos comandos podem ser adicionados nesta biblioteca.

Novas bibliotecas de comandos direcionadas para um devido fim também podem ser criadas. Por exemplo, se quisermos fazer uma demonstração do robô podemos criar a ferramenta Apresentacao.Tool com uma lista de comandos que devem ser executados em seqüência nesta demonstração.

B.5 Gerando um novo Bootfile

Sempre que compilamos um módulo é gerado um arquivo com terminação Obx. Este arquivo é entendido pelo microprocessador e deve ser carregado quando quisermos executar algum de seus procedimentos.

O Boot File além de configurar os objetos carrega estes arquivos do tipo Obx.

No entanto, quando modificamos um módulo e o compilamos, estas modificações não são transferidas automaticamente para o arquivo de boot. É necessário gerar um novo boot file.

Para gerar um novo boot file utilizamos a ferramenta Inter.Tool apresentada a seguir.

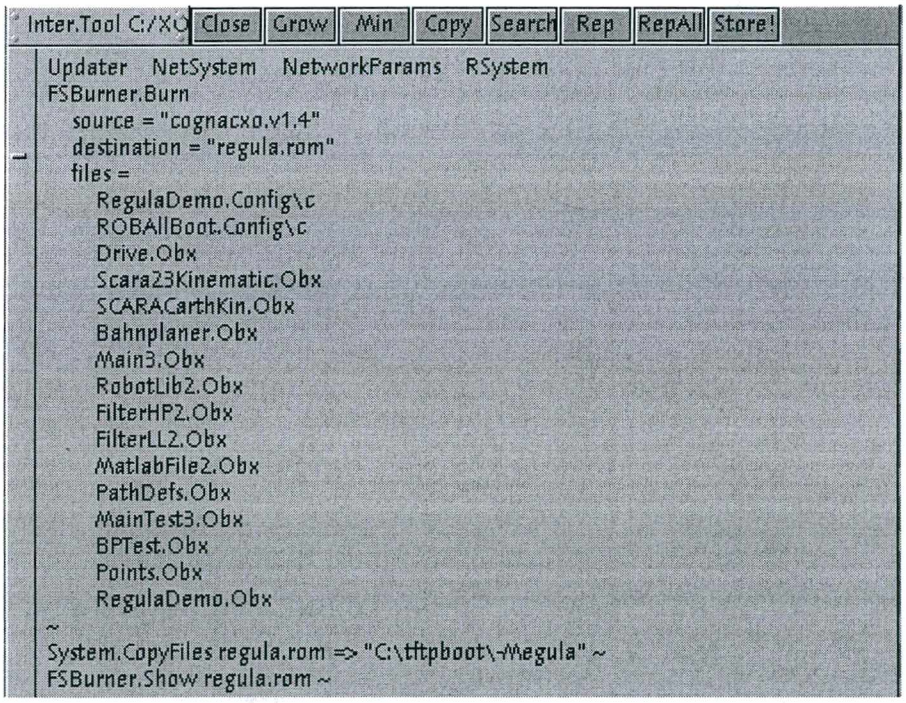


FIGURA B.11 - O Inter Tool.

Se compilarmos um módulo existente, não há necessidade de modificações em Inter.Tool. No entanto, se compilarmos um módulo novo é quisermos que este módulo seja carregado no processo de boot, basta incluir o nome deste módulo na listagem dos módulos acima apresentada com terminação Obx. Por exemplo, se criarmos e compilarmos o módulo Visao, para que este módulo seja carregado no processo de

boot basta escrever Visao.Obx abaixo de RegulaDemo.Obx mas obrigatoriamente antes do sinal ~.

Para gerar o novo boot file devemos clicar em FSBurner.Burn. Desta forma criamos no Host o arquivo regula.rom com todas as informações necessárias para o processo de boot. A seguir devemos clicar em System.CopyFiles que copia o arquivo regula.rom para o diretório C:\tftpboot com o nome Megula.

Está completo o processo de criação do novo arquivo de boot. Quando ligarmos novamente o armário de controle este novo arquivo será carregado.

O traço na frente do nome Megula é necessário. Se quisermos utilizando outro Drive, diretório ou arquivo para armazenar o bootfile, devemos alterar o arquivo Inter.Tool.

B.6 Compilação de Módulos

Na linguagem XOberon existem relações de importação entre os módulos, como visto no capítulo 4. Após compilarmos um módulo surgem na tela algumas informações a respeito desta compilação. Se somente aparecerem na tela alguns números os quais indicam o tamanho do módulo, isto significa que os demais módulos importadores não foram afetados por esta alteração e não precisam ser novamente compilados. No entanto, se ao compilarmos um módulo e surgir na tela a mensagem new symbol file, isto significa que temos de compilar novamente todos os módulos que importam este módulo e assim sucessivamente, criando uma reação em cadeia. Muitas vezes é difícil verificar quais os módulos que devem ser compilados novamente. Para facilitar esta tarefa utilizamos o arquivo Version.Text do diretório c:\XOberon\Inter. Neste arquivo aparece a listagem de todos os módulos em seqüência de compilação. Se modificarmos um módulo e na sua compilação for exigido que os módulos que o importem também sejam compilados, basta compilar os módulos que estejam abaixo deste no arquivo Version.Text. Sempre que um novo módulo for criado deverá ser acrescentado na posição adequada no interior deste arquivo.

B.7 Plotagem

A plotagem de dados é uma ferramenta essencial em pesquisa pois é a melhor forma de avaliar o desempenho do robô.

A linguagem XOberon não possui uma boa ferramenta de plotagem e para resolver este problema foi utilizada uma solução combinando os recursos do XOberon com os do software Matlab.

Em XOberon foi criado o módulo MatlabFiles2 que gera um arquivo de nome mlabdata.m que contém os dados a serem plotados.

A plotagem está atualmente restrita a oito diferentes possibilidades de conjuntos de dados a serem plotados. Quando o robô é ligado, nenhum tipo de plotagem está selecionado. Para fazer isto é necessário chamar um dos procedimentos de plotagem disponíveis.

As possibilidades de plotagem são:

- SetKreisPlot - fornece as posições e velocidades desejadas e realizadas, plotando x em função de y no espaço cartesiano. Utilizado somente quando o robô estiver descrevendo uma trajetória circular. Para todos os demais movimentos utilizar os procedimentos a seguir descritos.

- SetXYPlot - fornece as posições e velocidades desejadas e realizadas nas direções x e y do espaço cartesiano, plotando este valores em função do tempo.

- SetZPhiPlot - fornece as posições e velocidades desejadas e realizadas nas direções z e phi do espaço cartesiano, plotando este valores em função do tempo.

- SetQ0Q1Plot - fornece as posições e velocidades desejadas e realizadas pelas juntas 0 e 1, plotando este valores em função do tempo.

- SetQ2Q3Plot - fornece as posições e velocidades desejadas e realizadas pelas juntas 2 e 3, plotando este valores em função do tempo.

- SetTau0Tau1Plot - fornece os torques de controle efetivamente aplicados e os torques de controle compensados das juntas 0 e 1, plotando este valores em função do tempo.

- SetTau2Tau3Plot - fornece os torques de controle efetivamente aplicados e os torques de controle compensados das juntas 2 e 3, plotando este valores em função do tempo.

Os comandos para selecionar algum destes tipos de plotagem podem ser encontrado no arquivo BPTest.Tool.

Definido o tipo de plotagem que desejamos, o arquivo mlabdata é automaticamente gerado quando realizamos qualquer novo movimento. Este arquivo já contém os comandos para efetivar a plotagem e para criar a legenda no Matlab.

A dificuldade se encontra na transferência do arquivo do Target para o Host que deve ser feita via TFTP. Existem duas soluções possíveis.

A primeira é utilizar algum software para transferência direta dos dados do Target para o Host. Não está sendo utilizada no momento pois não temos um software para o Windows95 que possa ser automatizado para esta tarefa. É necessário configurar cada vez que formos utilizá-lo.

A segunda é buscar o arquivo do Target através do sistema UNIX e posteriormente transferir deste sistema para o Windows95 via TFTP. Apesar desta solução parecer mais complexa, está sendo adotada no momento por permitir uma maior automatização.

No início da sessão devemos nos conectar à uma outra máquina com o nome de usuário robinter para e cuja senha é conhecida pelo coordenador do laboratório. O programa TFTP Server 95 utilizado para o carregamento do boot file deve permanecer aberto. Após gerado o arquivo mlabdata no Target, basta digitar tftpcommand na tela referente ao usuário robinter. Feito isto o arquivo é transferido diretamente para o diretório tftpboot e está pronto para ser utilizado pelo Matlab.

No Matlab devemos ter o cuidado de sempre trabalhar no diretório tftpboot. Para visualizar o gráfico basta digitar mlabdata.

A figura a seguir mostra um gráfico gerado durante uma movimentação do robô, estando o procedimento SetQ0Q1Plot selecionado.

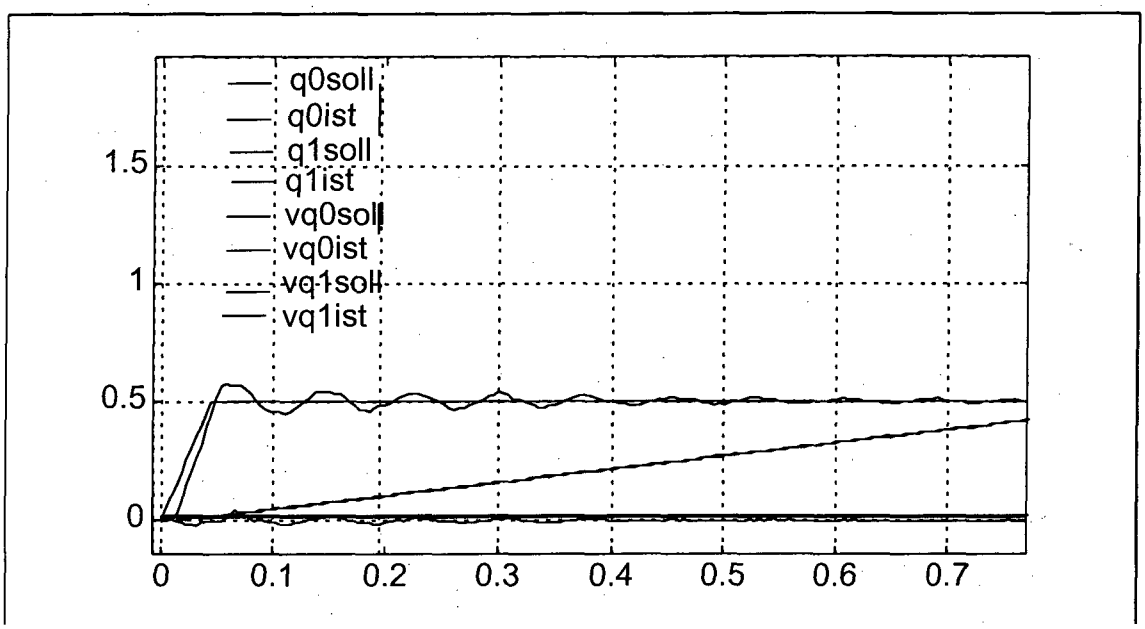


FIGURA B.12 - Exemplo de plotagem.

A matriz com os dados a serem plotados é armazenada na variável A. Esta matriz pode ser trabalhada na forma que quisermos, usando comandos do Matlab. É possível por exemplo plotar apenas as posições desejadas e realizadas, caso os dados de velocidade não sejam de interesse no momento. Também é possível definir uma função erro que compare os valores desejados e realizados ao longo de uma trajetória, ferramenta que pode ser bastante útil na verificação do desempenho de diferentes tipos de controladores.

Sempre que um novo arquivo é gerado no Target, o anterior é apagado.

Resumiremos a seguir os passos necessários para a plotagem de gráficos de movimentação do robô.

1. Selecionar um dos tipos de plotagem em BPTest.Tool;

2. Verificar se TFTP Server 95 está aberto. A opção overwrite if file exist deve estar obrigatoriamente selecionada;

3. Conectar outra máquina através do programa Telnet, utilizando como user name robinter;

4. Iniciar Matlab e mudar para o diretório C:\tftpboot no microcomputador ARARA;

5. Movimentar o robô na trajetória em que se deseja verificar os dados de plotagem;

6. Na tela referenta à conexão via Telnet efetuada com a outra máquina digitar tftpcommand;

7. No Matlab digitar mlabdata;

Ao digitar tftpcommand estamos na verdade executando o arquivo tftpcommand. Este arquivo possui uma série de comandos que automatizam o processo de transferência de arquivos via TFTP.

A plotagem de dados desta forma atualmente utilizada não é muito amigável. Se quisermos plotar conjuntos de dados diferentes dos já estabelecidos teremos de fazer uma série de mudanças em vários módulos existentes. Também não existe a possibilidade de plotar dois tipos de gráficos distintos simultaneamente. O processo de transferência de dados é complicado mas pode ser facilitado com a aquisição de um software adequado para transferência de dados via TFTP que funcione no Windows95.

Estão sendo desenvolvidos no ETH programas para integração do XOberon com o Visual Basic. Quando concluída esta integração, é possível que tenhamos a nossa disposição recursos de visualização e plotagem bastante superiores aos atualmente utilizados.

APÊNDICE C

DETALHAMENTO DOS PRINCIPAIS MÓDULOS DE SOFTWARE DO ROBÔ

C.1 Introdução

Neste apêndice estão explicadas as principais informações contidas nos módulos de software do robô.

C.2 O Arquivo ROBAllBoot

As linhas de interesse do arquivo ROBAllBoot.Config são apresentadas a seguir.

```
Const.DefScalar BaseClock=0.001~
```

Define o valor da constante *BaseClock* em segundos. Esta constante especifica o *clock* do controlador do robô. Um novo sinal de controle é gerado a cada milisegundo. Se quisermos aumentar ou diminuir o *clock* do controlador, temos de alterar o valor desta constante na configuração.

```
Drive.DefDrive Drive0 (ampl=A0, counter=Counter0, enable=Enable0, break=Break0,
sWMin=SW0Min, sWMax=SW0Max, vmax=3.0, vmin=-3.0, forcemax=333.0, forcemin=-
333.0, posmax=2.25, posmin=-2.25, amax=80, amin=-80) ~
```

Cria o objeto *Drive0* que estabelece os principais parâmetros da junta 0.

Em *ampl=A0* o campo *ampl* assume valores e características do objeto *A0*, o qual é configurado na primeira parte do arquivo *ROBAllBoot*. Este objeto, além de possuir vários comandos associados, define para qual endereço deve ser enviado o sinal de controle do amplificador da junta 0, já executando todas as conversões de unidades. Ao fazer *ampl=A0*, podemos portanto utilizar para o campo *ampl* todas propriedades previamente definidas para *A0*.

Um dos comandos associados ao objeto A0 é o comando *Write*, utilizado justamente para enviar o torque de controle para a junta. Para enviar um torque de controle para a junta 0, devemos escrever algo semelhante a *Drive0.ampl.Write(105)*, onde 105 é o valor do torque desejado em Nm. Dizemos semelhante pois não podemos esquecer que o objeto *Drive0* está na biblioteca e pode ser buscado com qualquer nome desejado. Se tivermos por exemplo um módulo de nome *TesteDrive* que busque da biblioteca o objeto *Drive0* com o nome *junta0*, para enviar um torque de controle deveríamos escrever o comando *junta0.ampl.Write(105)*.

Em *counter=Counter0* o campo *counter* assume os valores e características do objeto *Counter0* anteriormente configurado. Este objeto *Counter0* define o endereço e escala de conversão para o sinal do encoder que fornece a posição da junta 0. Se quisermos saber a posição da junta 0, já fornecida em radianos, devemos escrever algo semelhante à *Drive0.counter.Read()*.

Em *enable=Enable0* o campo *enable* assume os valores e características do objeto *Enable0* anteriormente configurado. Este objeto *Enable0* define o endereço do sinal para habilitar e desabilitar o amplificador. Para habilitar o amplificador devemos escrever algo semelhante à *Drive0.enable.Write(TRUE)* e para desabilitar devemos escrever *Drive0.enable.write(FALSE)*.

Em *brake=Brake0* o campo *brake* assume os valores e características do objeto *Brake0* anteriormente configurado. Este objeto *Brake0* define o endereço do sinal para acionar e desacionar o freio da junta 0. Para freiar a junta devemos escrever algo semelhante à *Drive0.break.Write(TRUE)* e para soltar o freio devemos escrever *Drive0.break.Write(FALSE)*.

Em *sWMin=SW0Min* o campo *sWMin* assume os valores e características do objeto *SW0Min* anteriormente configurado. Este objeto *SW0Min* define o endereço do sinal do sensor de fim de curso no sentido negativo. Para ler o estado deste sensor devemos escrever algo semelhante à *Drive0.sWMin.Read()*. Se o resultado desta leitura for *TRUE*, o sensor está acionado. Se for *FALSE*, o sensor não está acionado.

Em *sWMax=SW0Max* o campo *sWMax* assume os valores e características do objeto *SW0Max* anteriormente configurado. Este objeto *SW0Max* define o endereço do

sinal do sensor de fim de curso no sentido positivo. Para ler o estado deste sensor devemos escrever algo semelhante à *Drive0.sWMax.Read()*. Se o resultado desta leitura for *TRUE*, o sensor está acionado. Se for *FALSE*, o sensor não está acionado.

Em *vmax=3.0* e *vmin=-3.0* são definidas as velocidades máxima e mínima da junta 0 em rad/s.

Em *posmax=2.25* e *posmin=-2.25* são definidas as posições máxima e mínima da junta 0 em rad.

Em *amax=80.0* e *amin=-80.0* são definidas as acelerações máxima e mínima da junta 0 em rad/s^2.

Em *forcemax=333.0* e *forcemin=-333.0* são definidos os momentos máximo e mínimo em Nm que podem ser aplicados na junta 0.

Após o arquivo de configuração ter sido carregado, o objeto *Drive0* está na biblioteca. Para utilizarmos os dados deste objeto temos de criar um ponteiro para o local onde está armazenado este objeto. O nome deste ponteiro é definido pelo programador e toda vez que formos utilizar um dos campos deste objeto, devemos utilizar o nome do ponteiro e não o do objeto. Por exemplo, se criarmos o ponteiro *d[0]* que aponte para *Drive0* e se quisermos ler a posição da junta 0 devemos escrever *d[0].counter.Read()*.

```
Drive.DefDrive Drive1 (ampl=A1, counter=Counter1, enable=Enable1, break=Break1,
sWMin=SW1Min, sWMax=SW1Max, vmax=3.0, vmin=-3.0, forcemax=157.0, forcemin=-
157.0, posmax=1.9, posmin=-1.9, amax=100.0, amin=-100.0) ~
```

Cria o objeto *Drive1* que define os principais parâmetros da junta 1, de forma semelhante ao mostrado para a junta 0.

Os dados sobre velocidade máxima e mínima, força máxima e mínima, posição máxima e mínima e aceleração máxima e mínima configurados devem ser sempre consultados quando existir alguma dúvida a respeito dos limites da junta. Estes dados podem ser alterados se houver algum motivo que justifique esta mudança. Se por

exemplo forem feitos testes práticos que estabeleçam que a aceleração máxima da junta 1 pode ser superior ao valor atualmente utilizado, este campo pode ser alterado.

```
Drive.DefDrive Drive2(ampl=A2, counter=Counter2, enable=Enable2, break=Break2,
sWMin=SW2Min, sWMax=SW2Max, vmax=0.88, vmin=-0.88, forcemax=6.975, forcemin=-
6.975, posmax=0.40, posmin=0.14, amax=3000.0, amin=-3000.0) ~
```

Cria o objeto *Drive2* que define os principais parâmetros da junta 2, de forma semelhante ao mostrado para a junta 0.

Para a junta 2 cabem alguns comentários adicionais. Por se tratar de uma junta de translação, a velocidade é definida em m/s, a posição em metros e a aceleração em m/s^2. A força máxima permanece em Nm.

```
Drive.DefDrive Drive3 (ampl=A3, counter=Counter3, enable=Enable3, break=Break3,
sWMin=SW3Min, sWMax=SW3Max, vmax=20.0, vmin=-20.0, forcemax=16.74, forcemin=-
16.74, posmax=2.5, posmin=-2.5, amax=500.0, amin=-500.0) ~
```

Cria o objeto *Drive3* que define os principais parâmetros da junta 3, de forma semelhante ao mostrado para a junta 0.

```
Scara23Kinematic.DefScara23Kinematic SC23Kin (steigZ=0.025) ~
```

Cria o objeto *SC23Kin* que estabelece o acoplamento cinemático entre as juntas 2 e 3.

Em *steigZ=0.025* é definido que para cada volta das polias dentadas acopladas ao sistema, o fuso se desloca 0.025 metros, como visto no capítulo 2. O sentido de deslocamento é determinado pelo sentido de rotação das polias dentadas.

```
SCARACarthKin.DefCarthKinematic SCCKin (Z0=0.665, L1=0.25, L2=0.25, drive0=Drive0,
drive1=Drive1, drive2=Drive2, drive3=Drive3) ~
```

Cria o objeto *SCCKin* que estabelece os parâmetros cinemáticos do robô.

Em *Z0=0.665* é definida a distância em metros entre a origem do eixo Z no sistema de coordenadas cartesianas do robô e o sistema de coordenadas definido para a junta 2. Este valor é utilizado para transformar dados do espaço de juntas para o espaço cartesiano e vice-versa.

Em *L1=0.25* e *L2=0.25* é definido o comprimento do elo 0 e do elo 1. Estes valores são diretamente utilizados na cinemática direta e inversa do robô.

Nos demais campos são definidos ponteiros para os objetos *Drive0, Drive1, Drive2* e *Drive3* definidos anteriormente.

SynchCtrl.DefLimitSWCtrl SynchCtrl0 (maxSynchDist=5.5, synchSpeedSlow=-0.01, synchSpeedFast=-0.1, synchPos= -2.5102) ~

Cria o objeto *SynchCtrl0* que define os parâmetros para a sincronização da junta 0.

O campo *maxSynchDist=5.5* define o deslocamento máximo em rad que a junta 0 pode descrever antes de atingir o sensor de final de curso. Como foi definido um valor maior que a amplitude máxima de deslocamento desta junta (4.5 rad), este valor é usado para identificar algum sensor danificado.

O campo *synchSpeedSlow* define a velocidade lenta sincronização em rad/s enquanto *synchSpeedFast* define a velocidade rápida de sincronização em rad/s. Se o processo de sincronização estiver muito lento é possível acelerá-lo aumentando em módulo o valor de *synchSpeedFast*. O campo *synchSpeedSlow* não deve ser alterado pois garante a precisão do processo de sincronização.

O campo *synchPos* define a posição da junta 0 ao acionar o sensor de final de curso. Está diretamente associada a posição adotada como centro do sistema de coordenadas do robô e a orientação dos seus eixos cartesianos. Se quisermos definir uma nova orientação para os eixos X e Y do sistema de coordenadas do robô, deveremos também alterar de forma coerente a posição de sincronização para que as transformações cinemáticas sejam válidas.

Os valores de *synchSpeedSlow*, *synchSpeedFast* e *synchPos* são negativos pois a sincronização da junta 0 é feita no sentido negativo.

```
SynchCtrl.DefLimitSWCtrl SynchCtrl1
(maxSynchDist=5.0, synchSpeedSlow=0.02, synchSpeedFast=0.1, synchPos= 2.0980) ~
```

Cria o objeto *SynchCtrl1* que define os parâmetros de sincronização da junta 1. Cabem os mesmos comentários e observações feitos ao objeto *SynchCtrl0*, exceto que neste caso a sincronização é feita no sentido positivo, gerando valores positivos para *synchSpeedSlow*, *synchSpeedFast* e *synchPos*.

```
SynchCtrl.DefLimitSWCtrl SynchCtrl2
(maxSynchDist=0.27, synchSpeedSlow=-0.003, synchSpeedFast=-0.01, synchPos= 0.12562)
```

Cria o objeto *SynchCtrl2* que define os parâmetros de sincronização da junta 2. Os campos *synchSpeedFast* e *synchSpeedSlow* são definidos em m/s e *maxSynchDist* e *synchPos* em metros. As velocidades são definidas no sentido negativo, mas como no espaço de trabalho a posição da junta 2 é sempre positiva (ver capítulo 2), *synchPos* também é positivo.

```
SynchCtrl.DefLimitSWCtrl SynchCtrl3
(maxSynchDist=6.3, synchSpeedSlow=-0.02, synchSpeedFast=-0.5, synchPos= -2.826) ~
```

Cria o objeto *SynchCtrl3* que define os parâmetros de sincronização da junta 3. Cabem os mesmos comentários e observações feitos ao objeto *SynchCtrl0*.

```
Const.DefScalar piscClock=0.005 ~
```

Define o valor da constante *piscClock* em segundos.

```
PISpeedCtrl.DefPISpeedCtrl SpeedCtrl0
(p=400.0, i=15000.0, iLimit=0.1, clock=BaseClock, rampTime=piscClock, a=0.0, b=0.0,
c=0.0, offset=0.0, bsFilter=BSDummy0) ~
```

Cria o objeto *SpeedCtrl0* que define os ganhos do controlador de velocidade da junta 0, as constantes de tempo e as constantes da função utilizada para compensação do atrito nas juntas.

Em *p=400* é definido o ganho proporcional do controlador de velocidade da junta 0.

Em *i=15000* é definido o ganho integral do controlador de velocidade da junta 0.

Em *iLimit=0.1* é definido o valor máximo que a parcela integral pode fornecer ao controlador.

Em *clock=BaseClock* o campo *clock* assume o valor anteriormente definido para *BaseClock*. É utilizada nos cálculos internos do controlador.

Em *rampTime=piscClock* o campo *rampTime* assume o valor anteriormente definido para *pisClock*. Para compreender a razão deste campo é necessário algum comentário a respeito da função do módulo *RamFilterLib*. O sinal de controle possui um clock de um milisegundo. Na geração de trajetórias, cada novo dado é gerado a cada 5 milisegundos, conforme será definido posteriormente neste arquivo de configuração. Desta forma apresentada o controlador trabalharia cinco vezes com o mesmo valor desejado. A função do módulo *RamFilterLib* é fazer um escalonamento do valor desejado, de forma que a cada clock do controlador exista um novo valor desejado, gerando trajetórias mais suaves. O campo *rampTime* é utilizado em alguns cálculos dentre deste módulo e deve obrigatoriamente concordar com o clock do módulo de geração de trajetórias.

Foi implementada no robô uma função que define a relação entre a velocidade da junta e o torque de atrito correspondente. Os campos *a*, *b*, *c*, *offset* definem as constantes desta função. No processo de configuração estes campos foram zerados, de forma que ao inicializar o robô não exista compensação de atrito pois esta se mostrou oportuna em alguns casos mas inapropriada em outros. Atualmente, para efetivar esta compensação, a definição de valores para estes campo deve ser feita a partir de chamadas externas. Quando forem definidas constantes que apresentem resultados

genéricos satisfatórios, estes valores podem ser diretamente utilizados na configuração.

```
PISpeedCtrl.DefPISpeedCtrl SpeedCtrl1

(p=80.0, i=12000.0, iLimit=0.1, clock=BaseClock, rampTime=piscClock, a=0.0, b=0.0, c=0.0,
offset=0.0, bsFilter=BSDummy1) ~

PISpeedCtrl.DefPISpeedCtrl SpeedCtrl2

(p=0.1, i=3.0, iLimit=1.0, clock=BaseClock, rampTime=piscClock, a=0.0, b=0.0, c=0.0,
offset=0.0, bsFilter=BSDummy2) ~

PISpeedCtrl.DefPISpeedCtrl SpeedCtrl3

(p=1.0, i=35.0, iLimit=0.1, clock=BaseClock, rampTime=piscClock, a=0.0, b=0.0, c=0.0,
offset=0.0, bsFilter=BSDummy3) ~
```

Criam os objetos *SpeedCtrl1*, *SpeedCtrl2* e *SpeedCtrl3* que definem os ganhos do controlador de velocidade, as constantes de tempo e as constantes da função utilizada para compensação do atrito nas juntas 1, 2 e 3 respectivamente.

```
PISpeedCtrl.DefPISpeedCtrl HSCtrlX

(p=1200.0, i=40000.0, iLimit=0.01, clock=BaseClock, rampTime=piscClock, a=0.0, b=0.0,
c=0.0, offset=0.0, bsFilter=BSDummy0) ~
```

Cria o objeto *HSCtrlX*. Este objeto possui os mesmos campos de *SpeedCtrl0* e sua função é semelhante. A diferença é que seus dados são utilizados quando o controle híbrido do robô estiver acionado. O controle híbrido é feito no espaço de coordenadas cartesianas, portanto não estamos falando mais de juntas e sim de direções cartesianas. Neste caso são definidos os ganhos para a direção cartesiana X, que são totalmente diferentes daqueles definidos anteriormente para o controle no espaço das juntas. O funcionamento do controle híbrido será explicado em capítulo posterior.

```
PISpeedCtrl.DefPISpeedCtrl HSCtrlY

(p=1200.0, i=40000.0, iLimit=0.01, clock=BaseClock, rampTime=piscClock, a=0.0, b=0.0,
c=0.0, offset=0.0, bsFilter=BSDummy1) ~
```

Cria o objeto *HSCtrlY* que define os ganhos na direção cartesiana Y.

```
PISpeedCtrl.DefPISpeedCtrl HSCtrlZ
(p=0.0, i=0.0, iLimit=0.0, clock=BaseClock, rampTime=piscClock, a=0.0, b=0.0, c=0.0,
offset=0.0, bsFilter=BSDummy2) ~
```

Cria o objeto *HSCtrlZ* que define os ganhos na direção cartesiana Z.

```
PISpeedCtrl.DefPISpeedCtrl HSCtrlPhi
(p=0.0, i=0.0, iLimit=0.0, clock=BaseClock, rampTime=piscClock, a=0.0, b=0.0, c=0.0,
offset=0.0, bsFilter=BSDummy3) ~
```

Cria o objeto *HSCtrlPhi* que define os ganhos para orientação Phi.

```
Const.DefScalar BahnplanerClock=0.005 ~
```

Defme o valor da constante *BahnplanerClock* em segundos. Esta constante especifica o clock do gerador de trajetórias do robô. Um novo valor é gerado a cada 5 milisegundos. Se quisermos aumentar ou diminuir o clock do gerador de trajetórias, temos de alterar o valor desta constante na configuração.

```
StateCtrl.DefStateCtrl StateCtrl0
(kPos=400.0, kVel=1.0, acc=80.0, systemdelay=0.0, i=0.0, iLimit=0.0, inertia=0.1,
clock=BaseClock, rampTime=BahnplanerClock, bsFilter=BSDummy0) ~
```

Cria o objeto *StateCtrl0* que define os ganhos do controlador de posição da junta 0. O funcionamento do controlador de posição e a definição de seus ganhos serão descritos posteriormente.

Para os campos *clock* e *rampTime* cabem os mesmos comentários feitos anteriormente para o objeto *SpeedCtrl0.*

```
StateCtrl.DefStateCtrl StateCtrl1
```

(kPos=400.0, kVel=1.0, acc=100.0, systemdelay=0.0, i=0.0, iLimit=0.0, inertia=0.07, clock=BaseClock, rampTime=BahnplanerClock, bsFilter=BSDummy1) ~

Cria o objeto *StateCtrl1* que define os ganhos do controlador de posição da junta 1.

StateCtrl.DefStateCtrl StateCtrl2

(kPos=400.0, kVel=1.0, acc=100.0, systemdelay=0.0, i=0.0, iLimit=0.0, inertia=0.0001, clock=BaseClock, rampTime=BahnplanerClock, bsFilter=BSDummy2) ~

Cria o objeto *StateCtrl2* que define os ganhos do controlador de posição da junta 2.

StateCtrl.DefStateCtrl StateCtrl3

(kPos=400.0, kVel=1.0, acc=500.0, systemdelay=0.0, i=0.0, iLimit=0.0, inertia=0.001, clock=BaseClock, rampTime=BahnplanerClock, bsFilter=BSDummy3) ~

Cria o objeto *StateCtrl3* que define os ganhos do controlador de posição da junta 0.

StateCtrl.DefStateCtrl HStCtrlX

(kPos=0.01, kVel=0.0, acc=10.0, systemdelay=0.00, i=0.0, iLimit=0.00, inertia=0.0, clock=BaseClock, rampTime=BahnplanerClock, bsFilter=BSDummy0) ~

Cria o objeto *HStCtrlX* que define os ganhos do controlador na direção X do espaço cartesiano, quando o controle híbrido do robô estiver acionado.

StateCtrl.DefStateCtrl HStCtrlY

(kPos=0.01, kVel=0.0, acc=10.0, systemdelay=0.00, i=0.0, iLimit=0.00, inertia=0.0, clock=BaseClock, rampTime=BahnplanerClock, bsFilter=BSDummy1) ~

Cria o objeto *HStCtrlY* que define os ganhos do controlador na direção Y do espaço cartesiano, quando o controle híbrido do robô estiver acionado.

```
StateCtrl.DefStateCtrl HStCtrlZ
(kPos=0.01, kVel=0.0, acc=10.0, systemdelay=0.00, i=0.0, iLimit=0.0, inertia=0.000,
clock=BaseClock, rampTime=BahnplanerClock, bsFilter=BSDummy2) ~
```

Cria o objeto *HStCtrlZ* que define os ganhos do controlador na direção Z do espaço cartesiano, quando o controle híbrido do robô estiver acionado.

```
StateCtrl.DefStateCtrl HStCtrlPhi
(kPos=0.01, kVel=0.0, acc=10.0, systemdelay=0.0, i=0.0, iLimit=0.0, inertia=0.00,
clock=BaseClock, rampTime=BahnplanerClock, bsFilter=BSDummy3) ~
```

Cria o objeto *HStCtrlPhi* que define os ganhos do controlador referentes a orientação Phi do espaço cartesiano, quando o controle híbrido do robô estiver acionado.

```
ForceCtrl.DefForceCtrl ForceCtrl0(clock=BaseClock, p=1.0, i=0.0, d=0.0, iLimit=0.0) ~
```

Cria o objeto *ForceCtrl0* que define os ganhos do controlador de força na junta 0. O controlador de força não está sendo utilizado pois o sensor de força ainda não foi implementado. Assim que ocorrer a implementação, os ganhos devem ser definidos e configurados.

```
ForceCtrl.DefForceCtrl ForceCtrl1(clock=BaseClock, p=1.0, i=0.0, d=0.0, iLimit=0.0) ~
ForceCtrl.DefForceCtrl ForceCtrl2(clock=BaseClock, p=1.0, i=0.0, d=0.0, iLimit=0.0) ~
ForceCtrl.DefForceCtrl ForceCtrl3(clock=BaseClock, p=1.0, i=0.0, d=0.0, iLimit=0.0) ~
```

Criam os objeto *ForceCtrl1*, *ForceCtrl2* e *ForceCtrl3* que definem os ganhos dos controladores de força das juntas 1, 2 e 3 respectivamente.

```
ForceCtrl.DefForceCtrl HFCtrlX(clock=BaseClock, p=0.0, i=0.0, d=0.0, iLimit=0.0) ~
ForceCtrl.DefForceCtrl HFCtrlY(clock=BaseClock, p=0.0, i=0.0, d=0.0, iLimit=0.0) ~
ForceCtrl.DefForceCtrl HFCtrlZ(clock=BaseClock, p=0.0, i=0.0, d=0.0, iLimit=0.0) ~
```

```
ForceCtrl.DefForceCtrl HFCtrlPhi(clock=BaseClock, p=0.0, i=0.0, d=0.0, iLimit=0.0) ~
```

Criam os objetos *HFCtrlX, HFCtrlY, HFCtrlz* e *HFCtrlPhi* que definem os ganhos do controlador de força nas direções cartesianas X, Y, Z e Phi, utilizados quando o controle híbrido estiver acionado.

```
DynCompScara.DefDynamic Dynamic0
(j0=0.23, m0=15.0, l0=0.25, lc0=0.1, j1=0.16, m1=12.0, l1=0.25, lc1=0.18, j2=0.10, m2=2.0,
l2=0.0, lc2=0.0, j3=0.10, m3=1.5, l3=0.0, lc3=0.0) ~
```

Cria o objeto *Dynamic0* que define os parâmetros dinâmicos do robô. Alguns destes dados foram obtidos de catálogos e outros determinados experimentalmente. Como descrito no capítulo 2, existe uma grande incerteza na definição de alguns destes parâmetros. Existe um módulo que executa a compensação dinâmica, mas seus resultados foram observados somente através de plotagens e não na prática.

Será feita uma breve descrição de cada um destes parâmetros:

1. j0 - momento de inércia do elo 0 em kg*m^2;
2. m0 - massa do elo 0 em kg;
3. l0 - comprimento do elo 0 em metros;
4. lc0 - posição do centro de massa do elo 0 em metros;
5. j1 - momento de inércia do elo 1 em kg*m^2;
6. m1 - massa do elo 1 em kg;
7. l1 - comprimento do elo 1 em metros;
8. lc1 - posição do centro de massa do elo 1 em metros;
9. j2 - momento de inércia do elo 2 em kg*m^2;
10. m2 - massa do elo 2 em kg;
11. l2 - comprimento do elo 2 em metros;
12. lc2 - posição do centro de massa do elo 2 em metros;
13. j3 - momento de inércia do elo 3 em kg*m^2;
14. m3 - massa do elo 3 em kg;
15. l3 - comprimento do elo 3 em metros;

16. lc3 - posição do centro de massa do elo 3 em metros;

```
Main3.CInit ~
```

Não se trata de configuração de objetos. Esta linha já chama o procedimento *CInit* do módulo *Main3*. O funcionamento deste módulo será explicado posteriormente.

```
Bahnplaner.DefBP BPIm (clock=BahnplanerClock) ~
```

Cria o objeto *BPIm* que assume no campo *clock* o valor anteriormente definido para *BahnplanerClock*. Este objeto será buscado pelo módulo de geração de trajetórias.

Chegamos ao final do arquivo de configuração *ROBAllBoot*.

C.3 O Módulo Drive

Como o procedimento para configuração de objetos já foi visto em capítulo anterior, não entraremos em detalhes de configuração, nos atendo mais em explicar os campos do tipo *Drive* mostrado a seguir e os métodos a ele associados.

```
TYPE
 BaseType* = RECORD
    force*, acc*, speed*, pos* : REAL;
 END;

 Drive* = POINTER TO DriveDesc;
 DriveDesc* = RECORD (Base.ObjDesc)
  id* : INTEGER;
  counter*: P.Counter;
  ampl*: P.AnalogActuator;
  tacho*: P.AnalogSensor;
  tachoFilter*: FilterTP.TP;
  forceSens*: P.AnalogSensor;
  enable*: P.DigitalActuator;
  break*: P.DigitalActuator;
```

```
    sWSynch*, sWMin*, sWMax*: P.DigitalSensor;
    istValues*, maxValues*, minValues* : BaseType;
    swsynch*, swmin*, swmax* : BOOLEAN;
  END;
```

O tipo *Drive* é um ponteiro para *DriveDesc*, que por sua vez é uma extensão de *Base.ObjDesc*. Alguns campos de *DriveDesc* já foram comentados no capítulo 3.2. Explicaremos os novos campos e faremos alguns comentários adicionais a respeito dos campos anteriormente descritos.

O campo *id* : INTEGER*; cria uma variável de nome *id* do tipo inteiro.

O campo *counter*: P.Counter*; cria uma variável de nome *counter* do tipo *P.Counter*. *P.Counter* foi definido no módulo *Peripherals*. No processo de configuração é estabelecido um endereço onde será lido o valor do encoder associado à junta.

O campo *ampl*: P.AnalogActuator*; cria uma variável de nome *ampl* do tipo *P.AnalogActuador*. Este tipo foi definido no módulo *Peripherals*. No processo de configuração é estabelecido um endereço onde é feita a comunicação com os amplificadores.

O campo *tacho*: P.AnalogSensor*; cria uma variável de nome tacho do tipo *P.AnalogSensor*. Este tipo foi definido no módulo *Peripherals*. No processo de configuração é estabelecido um endereço onde será lido o valor do tacômetro associado à junta. Este campo é utilizado para ler a velocidade da junta.

O campo *tachoFilter*: FilterTP.TP*; cria uma variável de nome *tachoFilter* do tipo *FilterTP.TP*. O sinal fornecido pelo tacômetro nunca foi muito satisfatório, apresentando bastante ruído. Este módulo foi criado para melhorar este sinal através de um filtro. No entanto, o resultado também não foi bom e não está sendo utilizado no momento. Atualmente o sinal de velocidade está sendo obtido através da derivada do sinal de posição, sem a utilização do sinal do tacômetro.

O campo *forceSens*: P.AnalogSensor*; cria uma variável de nome *forceSensor* do tipo *P.AnalogSensor*. Este tipo foi definido no môdulo *Peripherals*. No processo de configuração é estabelecido um endereço onde será lida a força aplicada a junta

correspondente. Como o sensor de força ainda não está instalado, este campo não possui utilidade no momento e provavelmente terá de ser reformulado após a implementação do sensor.

A linha *enable*: P.DigitalActuator*; gera o campo *enable* do tipo *P.DigitalActuator*. Este tipo foi definido no módulo *Peripherals*. No processo de configuração é estabelecido um endereço onde é feita a habilitação dos amplificadores.

O campo *break*: P.DigitalActuator*; cria uma variável de nome *break* do tipo *P.DigitalActuator*. Este tipo é definido no módulo *Peripherals*. No processo de configuração é estabelecido um endereço onde o freio do motor pode ser acionado.

Os campos *sWSynch*, sWMin*, sWMax*: P.DigitalSensor;* criam as variáveis *sWSynch*, *sWMin* e *sWMax* que são do tipo *P.DigitalSensor*. O campo *sWSynch* não está sendo utilizado no momento. No processo de configuração são definidos endereços onde pode ser lida a situação dos sensores de final de curso.

A linha *istValues*, maxValues*, minValues* : BaseType;* gera os campos *istValues*, *maxValues* e *minValues* do tipo *BaseType*. *BaseType*, por sua vez, possui os campos *force*, *acc*, *speed* e *pos*. Na configuração são determinados os valores máximos e mínimos de força, aceleração, velocidade e posição da junta. Estes valores são armazenados nos campos *maxValues* e *minValues*. Já o campo *istValues* armazena os valores atuais de posição e velocidade da junta, lidos através dos campos previamente descritos.

A linha *swsynch*, swmin*, swmax* : BOOLEAN*; gera os campos *swsynch*, *swmin* e *swmax* que são do tipo booleano.

Os procedimentos a seguir listados são utilizados na configuração dos objetos do tipo *Drive*:

```
PROCEDURE (d : Drive) Assign* (o : Base.Object; name : ARRAY OF CHAR):BOOLEAN;
PROCEDURE AttrMsg(obj: Drive; VAR M: O.AttrMsg);
PROCEDURE Handler*(obj: O.Object; VAR M: O.ObjMsg);
PROCEDURE NewDrive*;
PROCEDURE DefDrive*;.
```

Ao criar um objeto é possível criar procedimentos associados a este objeto chamados de métodos. Mostraremos a seguir os métodos associados ao objeto *Drive*, explicando sua função básica.

```
PROCEDURE (d: Drive) Read*;
BEGIN

IF d.counter#NIL THEN d.istValues.pos:= d.counter.Read() ELSE d.istValues.pos:= 0.0
END;
IF d.tacho#NIL THEN
 IF d.tachoFilter#NIL THEN d.istValues.speed:=d.tachoFilter.Algo(d.tacho.Read())
  ELSE d.istValues.speed:=d.tacho.Read()
  END;
 ELSE d.istValues.speed:= 0.0
 END;
        IF d.forceSens#NIL THEN d.istValues.force:=d.forceSens.Read() ELSE
        d.istValues.force:= 0.0 END;
        IF d.sWMin#NIL THEN d.swmin:= d.sWMin.Read() ELSE d.swmin:= FALSE
        END;
        IF d.sWMax#NIL THEN d.swmax:= d.sWMax.Read() ELSE d.swmax:= FALSE
        END;
        IF d.sWSynch#NIL THEN d.swsynch:= d.sWSynch.Read() ELSE d.swsynch:=
 FALSE END;
END Read;
```

Para facilitar a compreensão, vamos supor que em outro módulo foi criado um ponteiro de nome *d[1]* que aponte para um objeto do tipo *Drive*. Para chamar o método *Read* deveremos escrever *d[1].Read* em qualquer ponto no interior deste outro módulo.

Antes de ler a informação contida nos endereços indicados é conferido se realmente existe algum dado naquele canal ou se está vazio. Quando perguntamos se *d.counter#NIL* estamos conferindo se existe algo a ser lido em *d.counter*. *NIL* significa que aquele endereço está vazio. Feita esta conferência é lido o valor ou estado correspondente àquele endereço. Neste método são lidos na seqüência a posição da junta, a velocidade da junta, a força aplicada na junta (somente quando o sensor estiver implementado) e o estado dos sensores de final de curso. Este valores serão armazenados em *d[1].istValue.pos*, *d[1].istValue.speed*, *d[1].istValue.force*, *d[1].swmin* e *d[1].swmax* respectivamente. Feita a leitura estes dados podem ser utilizados por exemplo para o cálculo dos valores de controle.

```
PROCEDURE (d: Drive) Init*;
```

Este método inicializa o tacômetro se este estiver sendo utilizado.

```
PROCEDURE (d: Drive) EnableAmp*;
```

Método utilizado para habilitar o amplificador.

PROCEDURE (d: Drive) DisableAmp*;

Método utilizado para desabilitar o amplificador.

PROCEDURE (d: Drive) BreakOn*;

Método utilizado para acionar os freios.

PROCEDURE (d: Drive) BreakOff*;

Método utilizado para soltar os freios.

PROCEDURE (d: Drive) EmergencyStop*;

Este método habilita o amplificador e solta o freio.

PROCEDURE (d: Drive) SuperVisor*(): BOOLEAN;

Este método supervisiona o estado dos sensores de final de curso. O robô deve trabalhar somente com os sensores desacionados. Assim que um deles for acionado, o sinal é detectado e o robô é parado. Este método complementa o procedimento *SuperVisor* do módulo *Main3*.

C.4 O Módulo Main3

Para instalar um novo tipo de controlador é necessário um bom conhecimento sobre esta tarefa. Sobre os demais procedimentos do módulo *Main3* nos restringiremos a escrever somente a primeira linha e a explicar sua função básica.

Os procedimentos são apresentados na ordem em que aparecem no módulo *Main3* a exceção da tarefa *Controller* que será apresentada no final.

PROCEDURE SetTestDrive*(driveNo : LONGINT) : BOOLEAN;

Este procedimento é utilizado quando quisermos trabalhar com apenas uma junta, deixando as demais inativas. Ao informar a junta que queremos utilizar, fica impossibilitada a habilitação dos amplificadores para as demais juntas.

PROCEDURE DynamicOnOff*() : BOOLEAN;

Através deste procedimento tornamos a compensação dinâmica ativa ou inativa. Como a compensação dinâmica ainda não foi implementada na prática, deve permanecer inativa até que testes complementares sejam realizados.

```
PROCEDURE GetAuthority* (VAR authorityNr : LONGINT) : BOOLEAN;
PROCEDURE AuthorityFree* (authorityNr : LONGINT) : BOOLEAN;
PROCEDURE AuthorityCheck* (authorityNr : LONGINT) : BOOLEAN;
PROCEDURE AuthorityReset*; (** Nur fuer Supereruser!! nicht in Programmen verwenden!! *)
```

A linguagem XOberon permite que vários processos do tipo *Every Event* ocorram simultaneamente. Existem atualmente 3 módulos que geram valores desejados de posição, velocidade e aceleração. Nada impede que estes processos ocorram simultaneamente e nada limita a criação de novos processos. Os procedimentos envolvendo o processo de Autorização foram criados como forma de evitar que dois ou mais processos enviem simultaneamente valores desejados de posição, velocidade ou aceleração. Ao instalarmos uma tarefa que gera valores desejados devemos primeiramente acionar o procedimento *GetAuthority* para obter autorização para estabelecer valores desejados. Se outro processo estiver com a autorização, ela não é concedida e a tarefa não é instalada. Se nenhum processo estiver com a autorização, ela é concedida e um número é fornecido. Este número funciona como uma senha. Toda vez que formos estabelecer novos valores desejados devemos também informar este número, o qual é checado pelo procedimento *AuthorityCheck.* Ao desinstalar a tarefa devemos chamar o comando *AuthorityFree* para permitir que outros processos possam obter a autorização. O comando *AuthorityReset* retira a autorização de qualquer um que a esteja usando, interrompendo o processo. Este é um procedimento extremamente perigoso pois os valores desejados de posição e velocidade no momento da interrupção passam a ser os valores de controle. Se por exemplo interrompermos um movimento durante a fase de aceleração, o valor de aceleração atual será mantido como valor desejado e o robô permanecerá acelerando até que o botão de emergência ou o sensor de fim de curso sejam acionados ou a velocidade máxima seja ultrapassada. Se existir algum obstáculo, é possível que ocorra uma colisão antes de que uma destas condições sejam satisfeitas. Por isto o

comando *AuthorityReset* não deve sob hipótese nenhuma ser chamado a partir de outros módulos.

Deve ficar claro que quando quisermos implementar novos módulos que gerem valores desejados, devemos sempre utilizar os comandos *GetAuthority* e *AuthorityFree.*

```
PROCEDURE GetPos*(driveNo : LONGINT) : REAL;
```

Este procedimento lê a posição atual da junta indicada. No parâmetro formal *driveNo* identificamos qual a junta sobre a qual se quer a informação. Se quisermos por exemplo saber a posição da junta 1, devemos escrever

```
pos[1] : = GetPos(1);
```

Na variável *pos[1]* será armazenado o valor da posição atual da junta 1.

```
PROCEDURE GetRobPos*(VAR posVect : GD.Vector4);
```

Este procedimento lê a posição atual das quatro juntas do robô. A posição das juntas 0,1 e 3 é fornecida em radianos e a posição da junta 2 é fornecida em metros.

```
PROCEDURE GetSollPos*(driveNo : LONGINT) : REAL;
```

Este procedimento lê a posição desejada da junta indicada.

```
PROCEDURE GetSollSpeed*(driveNo : LONGINT) : REAL;
```

Este procedimento lê a velocidade desejada da junta indicada.

```
PROCEDURE GetRobSollPosSpeed*(VAR posVect, speedVect : GD.Vector4);
```

Este procedimento lê as posições e velocidades desejadas das 4 juntas do robô.

```
PROCEDURE GetRobSollSpeed*(VAR speedVect : GD.Vector4);
```

Este procedimento lê as velocidades desejadas das 4 juntas do robô.

```
PROCEDURE GetSollDriveSpeed*(driveNo : LONGINT) : REAL;
```

Este procedimento lê as velocidades desejadas das 4 juntas do robô, antes da transformação da velocidade angular em velocidade linear na junta 2. Logo, todos os valores são fornecidos em rad/s.

PROCEDURE GetSpeed*(driveNo : LONGINT) : REAL;

Este procedimento lê a velocidade atual da junta indicada.

PROCEDURE GetRobSpeed*(VAR speedVect : GD.Vector4);

Este procedimento lê as velocidades atuais das 4 juntas do robô.

PROCEDURE GetRobPosSpeed*(VAR posVect : GD.Vector4; VAR speedVect : GD.Vector4);

Este procedimento lê as posições e as velocidades atuais das 4 juntas do robô.

PROCEDURE GetDrivePos*(driveNo : LONGINT) : REAL;

Este procedimento lê as posição atual da junta indicada do robô antes da transformação do movimento angular da junta 2 em movimento linear. Logo, todos os valores são fornecidos em rad.

PROCEDURE GetDriveSpeed*(driveNo : LONGINT) : REAL;

Este procedimento lê a velocidade da junta indicada do robô antes da transformação do movimento angular da junta 2 em movimento linear. Logo, todos os valores são fornecidos em rad/s.

PROCEDURE GetDrivesSwitches*(driveNo : LONGINT; VAR swsynch, swmin, swmax : BOOLEAN);

Fornece o estado em que se encontram os sensores de fim de curso da junta indicada.

PROCEDURE GetController*(driveNo : LONGINT) : LONGINT;

Fornece o tipo de controlador utilizado no momento pela junta indicada.

PROCEDURE GetSystemState*(VAR mainStatus, synchronizeStatus, mainFault, powerUpState : LONGINT;

Fornece o estado do robô (*mainStatus*), a situação no processo de sincronização (*synchronizeStatus*), se existe alguma falha (*mainFault*) e a condição de Power Up (*powerUpState*).

PROCEDURE GetSynchDist*(driveNo : LONGINT) : REAL;

Fornece a posição de sincronização da junta indicada, isto é, a posição da junta quando o sensor de fim de curso é acionado.

```
PROCEDURE GetSensForce*(VAR f : GD.Vector6);
PROCEDURE AdjustForceOffset*() : BOOLEAN;
PROCEDURE SetMaxForce*(force : GD.Vector6);
PROCEDURE ResetMaxForce*;
```

Estes quatro procedimentos referem-se aos valores fornecidos pelo sensor de força. Como o sensor ainda não foi instalado, estes procedimentos provavelmente deverão ser modificados posteriormente.

```
PROCEDURE ChangeCtrl*(ctrl : LONGINT) : BOOLEAN;
```

A exceção do controle híbrido, todos os demais controles são executados no espaço de juntas. Existem atualmente três possibilidades distintas de controladores para as juntas. Podemos controlar velocidade da junta, posição da junta e a força ou momento da junta. Para cada um destes casos existe um algoritmo de controle diferente. Quando utilizamos este procedimento, estamos modificando o tipo de controle usado nas juntas. No espaço de juntas não é possível trabalhar com controladores de tipos diferentes para cada junta. O controlador mais utilizado atualmente é o controlador de posição que será visto em mais detalhes posteriormente. O controle de velocidade pode ser útil em alguns casos, mas é preciso de cuidado na sua utilização pois ao estabelecer uma velocidade para uma junta ela somente irá parar quando estabelecermos velocidade zero ou quando algum dispositivo de segurança ou supervisão for acionado. Como o sensor de força ainda não está implementado, todos os ganhos do controlador de força foram igualados a zero para que ele não possa efetivamente ser realizado. Estes ganhos devem ser ajustados quando da implementação do sensor de força.

```
PROCEDURE ChangePIPara*(driveNo : LONGINT; p, i, iLimit : REAL) : BOOLEAN;
```

Altera os ganhos do controlador de velocidade da junta indicada.

```
PROCEDURE ChangeDynPara*(axe : LONGINT; j, m, l, lc : REAL) : BOOLEAN;
```

Altera os parâmetros dinâmicos do elo indicado.

```
PROCEDURE ChangeFrictPara*(driveNo : LONGINT; a, b, c, off : REAL) : BOOLEAN;
```

Como comentado anteriormente, foi criada uma curva relacionando a velocidade da junta e o torque de atrito correspondente. Este procedimento altera os coeficientes desta curva para a junta indicada. Após a configuração, estes valores *a,b,c e off* são estabelecidos de forma que não exista compensação de atrito. Para introduzir a compensação de atrito temos de utilizar este procedimento.

PROCEDURE ChangeStatePara*(driveNo : LONGINT; kPos, acc, systemdelay, i, iLimit, inertia : REAL) :BOOLEAN;

Altera os ganhos do controlador de posição para a junta indicada.

PROCEDURE ChangeSFilterPara*(driveNo : LONGINT; a0, a1, b0, b1 : REAL) : BOOLEAN;

Altera os ganhos de um filtro de segunda ordem. Atualmente não está sendo utilizado nenhum tipo de filtro.

PROCEDURE ChangeForcePara*(driveNo : LONGINT; p : REAL) : BOOLEAN;

Altera os ganhos do controlador de força para a junta indicada.

PROCEDURE ReleaseStopProc;

Habilita os amplificadores e solta os freios das 4 juntas do robô.

PROCEDURE StopProc;

Desabilita os amplificadores e solta os freios das 4 juntas do robô.

PROCEDURE InitSynchCtrl;

Inicializa os dados sobre as juntas, preparando o processo de sincronização.

PROCEDURE PowerUp;

Este procedimento não tem mais utilidade pois o botão de Power Up tem de ser acionado manualmente.

PROCEDURE WatchDog;

Este procedimento verifica se existe comunicação entre o microprocessador e o robô. A falha de comunicação poderia acarretar conseqüência muito graves pois toda supervisão feita via software estaria inutilizada. Como pequeno exemplo, os sinais dos

sensores de fim de curso não poderiam ser lidos e o robô poderia ultrapassar os limites do seu espaço de trabalho.

Funciona da seguinte maneira. Existe um canal cujo valor booleano deve ser alterado a cada 10 milisegundos. A tarefa *Controller* se encarrega de chamar este procedimento que apenas altera o estado booleano atual a cada execução. Quando por algum problema de comunicação este valor booleano não for alterado, o botão de Power Up cai imediatamente, garantido a integridade do robô e a segurança dos usuários. Esta verificação não tem nada a ver com a comunicação entre o microcomputador utilizado como Host e o microprocessador utilizado como Target, pois esta comunicação normalmente é feita via Internet e está sujeita a delays que não comprometem significativamente a segurança dos usuários e a integridade do robô.

PROCEDURE SuperVisor;

Supervisiona o estado dos sensores, a situação do botão de emergência, se existe algum erro nos amplificadores e as forças aplicadas no sensor de força. Se alguma destas condições estiver fora das normais, é acionado o estado de emergência. Este procedimento também é chamado pela tarefa *Controller*. Isto significa que este controle é feito a cada um millisegundo pois o clock desta tarefa está especificado atualmente neste valor.

PROCEDURE StopFrictMeasure*;
PROCEDURE Move(axis : LONGINT; pos, speed : LONGREAL) : BOOLEAN;
PROCEDURE FrictionMeasure*(VAR viscosFrictA, viscosFrictB, staticFrict : GD.Vector4) :

Estes procedimentos foram utilizados para medição dos dados de atrito das juntas que contribuíram para o estabelecimento dos coeficientes da função de atrito.

PROCEDURE SynchProc;

Este procedimento coordena o processo de sincronização das juntas do robô.

PROCEDURE ReleaseAllBreaks;

Este procedimento solta os freios das 4 juntas do robô.

PROCEDURE SetAllBreaks;

Este procedimento aciona os freios das 4 juntas do robô.

PROCEDURE WaitForPreSynchPos;

Este procedimento não é mais utilizado. Anteriormente o processo de sincronização era iniciado através da movimentação manual do robô que foi excluída por razões de segurança.

PROCEDURE GetEMErrors*(VAR ePU, eSVPU, eSVENDSW0, eSVENDSW1, eSVENDSW2, eSVENDSW3, eSVENDSW4, eSVEMBUT,eCTRLFORCE, eSYNCH, eSYNCHTIME, eSYNCHPRETIME, eSVMF0, eSVMF1, eSVMF2, eSVMF3, eSVMF4, eSVMF5, eSVAMPERR0, eSVAMPERR1, eSVAMPERR2, eSVAMPERR3 : BOOLEAN);

Este procedimento fornece o motivo pelo qual o estado Emergency foi acionado. É uma ferramenta importante que não deve ser esquecida quando o motivo da emergência for aparentemente desconhecido.

PROCEDURE ResetEMErrors*;

Após verificado o motivo que gerou o estado de emergência devemos utilizar este procedimento para resetar os erros.

PROCEDURE Configuration() : BOOLEAN;

Este procedimento é muito importante pois busca da Biblioteca os objetos configurados, atribuindo ponteiros com novos nomes a eles. Os objetos do tipo *Drive*, criados pelo arquivo de configuração com os nomes *Drive0*, *Drive1*, *Drive2* e *Drive3* são buscados da biblioteca e assumem os nomes *d[0]*, *d[1]*, *d[2]* e *d[3]* respectivamente. Estes últimos nomes é que serão válidos sempre que quisermos utilizar os vários campos destes objetos.

O nome do objeto criado pelo arquivo de configuração aparece entre parêntesis, sempre antes do nome do ponteiro. No *exemplo* *GetStateCtrl('StateCtrl0', stateCtrl[0]);*, *StateCtrl0* é o nome do objeto na biblioteca e *stateCtrl[0]* é o nome do ponteiro que deve ser posteriormente referenciado.

PROCEDURE NearAfterSynchPos() : BOOLEAN;

PROCEDURE MoveToEndSwitch;

Estes dois procedimentos auxiliam o processo de sincronização.

PROCEDURE PowerDown*;

O botão de Power Up não pode ser acionado via software mas pode ser desacionado. Isto acontece automaticamente quando o sinal de emergência é acionado. Este procedimento aciona emergência, desacionando o botão de Power Up.

PROCEDURE MAIN(e : XOK.Event);

Este procedimento ou tarefa já foi anteriormente comentado quando foram abordados os estados do robô. Neste procedimento é feito todo o controle, gerenciando a troca de estados e verificando se existem novos comandos. Também é responsável pela desinstalação da tarefa *Controller*.

PROCEDURE CInit*;

Comando que inicializa o robô. Passa o robô do estado *OFF* para o estado *Initialized*.

PROCEDURE CSynch*;

Inicia o processo de sincronização, passando o robô para o estado temporário *Synch*.

PROCEDURE CBye*;

Comando que muda o estado do robô para *OFF*.

PROCEDURE CReleaseAllBreaks*;

Comando que muda o estado do robô para *Break Release*.

PROCEDURE CSetAllBreaks*;

Comando que muda o estado do robô para *Initialized* quando o estado anterior era *Break Release*.

PROCEDURE CStop*;

Comando que muda o estado do robô para *Stop*.

PROCEDURE CReleaseStop*;

Comando que muda o estado do robô para *Run*.

PROCEDURE CShutDown*;

Comando que muda o estado do robô para *Initialized*.

PROCEDURE CResetEmergency*;

Comando que muda o estado do robô, saindo do estado *Emergency*.

PROCEDURE SetDriveSpeed*(driveNo : LONGINT; sollspeed : REAL) : BOOLEAN;

Procedimento utilizado para especificar a velocidade desejada da junta indicada. Como não confere a autorização, não deve ser utilizado em programas.

PROCEDURE SetDriveForce*(driveNo : LONGINT; sollforce : REAL) : BOOLEAN;

Procedimento utilizado para especificar o momento desejado da junta indicada. Como não confere a autorização, não deve ser utilizado em programas.

PROCEDURE SetDrivePosSpeed*(driveNo : LONGINT; sollpos, sollspeed : REAL) : BOOLEAN;

Procedimento utilizado para especificar a posição e a velocidade desejadas da junta indicada. Como não confere a autorização, não deve ser utilizado em programas.

PROCEDURE SetRobPosSpeed*(authorityNr : LONGINT; sollpos : GD.Vector4; sollspeed : BOOLEAN;

Procedimento utilizado para especificar a posição e a velocidade desejadas das 4 juntas do robô. Este procedimento e os dois seguintes foram criados justamente para checar se os valores desejados são provenientes de um programa que possui a autorização para isto. Sempre que for criado um programa que gere valores desejados de velocidade, posição ou força, estes procedimentos devem obrigatoriamente ser utilizados.

PROCEDURE SetRobSpeed*(authorityNr : LONGINT; sollspeed : GD.Vector4): BOOLEAN;

Procedimento utilizado para especificar a velocidade desejada das 4 juntas do robô.

PROCEDURE SetRobForce*(authorityNr : LONGINT; sollforce : GD.Vector4): BOOLEAN;

Procedimento utilizado para especificar o momento desejado das 4 juntas do robô.

Passaremos agora à análise da tarefa *Controller*. Apresentaremos esta tarefa dividindo-a em blocos e explicando a função de cada um destes blocos. O leitor não deve se preocupar no momento em entender cada linha desta tarefa.

```
PROCEDURE Controller(ctrlEvent : XOK.Event);
VAR
  i : LONGINT; dummy, dummy2 : REAL; dummyBol : BOOLEAN; istpos, istspeed,
  solltorq, absistspeed, deltaForce, oldoutForce : GD.Vector4; sign: REAL; frictforce:
  ARRAY 4 OF REAL;
BEGIN
```

Inicia o procedimento e cria as variáveis locais ao procedimento.

```
WatchDog;
        (* IF ~FTSensA.ReadForce(sensForce) THEN emergency:= TRUE;
emCTRLFORCE:= TRUE END; *)
SuperVisor;
```

Chama os procedimentos *WatchDog* e *SuperVisor*, cujas funções foram descritas anteriormente. A linha relativa ao sensor de força não está sendo utilizada pois este ainda não foi instalado.

```
FOR i:=0 TO MaxDrives-1 DO
 d[i].Read;
 IF d[i].tacho=NIL THEN
 d[i].istValues.speed:= (d[i].istValues.pos-oldValues[i].pos)/baseClock;
                IF d[i].tachoFilter#NIL THEN
                d[i].istValues.speed:= d[i].tachoFilter.Algo(d[i].istValues.speed)
                END;
 END;
 oldValues[i].pos:= d[i].istValues.pos;
END;
```

Faz a leitura da posição das juntas e calcula as velocidades caso o tacômetro não esteja sendo utilizado.

```
(*
(** BEGIN TESTHACK *)
```

```
        IF (testdrive>=0)&(testdrive<MaxDrives) THEN sollValues[testdrive].speed:=
    ADDAEltec.ReadAD(1)/10.0 END;
    (** END TESTHACK *)
    *)
```

Estas linhas foram utilizadas quando se acoplou ao robô um gerador de pulsos para alguns testes de vibração. Não está mais sendo utilizado no momento.

As linhas seguintes descrevem o gerenciamento dos controladores das juntas 0 e 1.

```
(* Roboter Drive 0-1, drive phi1 & drive phi2 *)

FOR i:=0 TO 1 DO
        CASE ctrlStatus[i] OF
          GD.SNoCtrl : outValues[i].force:= 0.0;
  | GD.SSynchCtrl :
            IF synchCtrl[i].synchPara.synchSpeedFast>0 THEN synchCtrl[i].inSynchSW:=
            d[i].swmax
    ELSE synchCtrl[i].inSynchSW:= d[i].swmin
    END;
          sollValues[i].speed:= synchCtrl[i].Algo(d[i].istValues.pos);
    IF synchCtrl[i].message=SynchCtrl.SynchNow THEN
              synchCtrl[i].synchdist:=d[i].istValues.pos;d[i].counter.Reset(synchCtrl[i].sy
              nchPara.synchPos);
             d[i].istValues.pos:= synchCtrl[i].synchPara.synchPos;
              oldValues[i].pos:= d[i].istValues.pos; piSpeedCtrl[i].Init(d[i].istValues.pos);
 synchCtrl[i].message:= SynchCtrl.SynchNowDone;
    END;
    IF ~enabled[i] THEN piSpeedCtrl[i].piPara.integ:= 0.0 END;
    outValues[i].force:= piSpeedCtrl[i].Algo
            (d[i].istValues.pos,      d[i].maxValues.acc,      d[i].istValues.speed,
            sollValues[i].speed);
  | GD.SPISpeedCtrl :
    IF ~enabled[i] THEN piSpeedCtrl[i].piPara.integ:= 0.0 END;
    outValues[i].force:= piSpeedCtrl[i].Algo
            (d[i].istValues.pos,      d[i].maxValues.acc,      d[i].istValues.speed,
            sollValues[i].speed);
    IF frictMes THEN
     IF posdir[i] THEN posreached[i]:= d[i].istValues.pos>sollValues[i].pos
     ELSE posreached[i]:= d[i].istValues.pos<sollValues[i].pos
    END;
    IF posreached[i] THEN
            IF ~measureDone[i] THEN messforce[i]:= outValues[i].force; messspeed[i]:=
            d[i].istValues.speed; measureDone[i]:= TRUE END;
     sollValues[i].speed:= 0.0
     END;
    END;
  | GD.SStateCtrl :
          outValues[i].force:=  stateCtrl[i].Algo(d[i].istValues.pos,  sollValues[i].pos,
          d[i].istValues.speed, sollValues[i].speed);
```

```
(* teste do novo state control
            outValues[i].force:=                piSpeedCtrl[i].Algo(d[i].istValues.pos,
            d[i].maxValues.acc, d[i].istValues.speed, outValues[i].speed);
 *)
 | GD.SForceCtrl :
  outValues[i].force:= forceCtrl[i].Algo(sollValues[i].force, sollValues[i].force);
  IF inertial THEN
   IF i=1 THEN
            dummyBol:= MF.Sample(outValues[0].force, SHORT(dynacomptorq[0]),
            outValues[1].force, SHORT(dynacomptorq[1]), (sollValues[0].speed),
            (d[0].istValues.speed), (sollValues[1].speed), (d[1].istValues.speed), timer);
            timer:=timer+0.001;
   END;
   IF sollValues[i].force>0.0 THEN
     IF (d[i].istValues.speed>(d[i].maxValues.speed/2 )) OR
            (d[i].istValues.pos>(d[i].maxValues.pos-(d[i].maxValues.pos-
            d[i].minValues.pos)/4)) THEN
smoothstop:=TRUE;
END;
     ELSE
     IF (d[i].istValues.speed<(d[i].minValues.speed/2 )) OR
            (d[i].istValues.pos<((d[i].maxValues.pos-
            d[i].minValues.pos)/4+d[i].minValues.pos)) THEN
       smoothstop:=TRUE;
     END;
    END;
  END;
  IF smoothstop THEN sollValues[i].force:=sollValues[i].force*0.99;
    IF (ABS(sollValues[1].force)<0.5 )&(ABS(sollValues[0].force)<0.5) THEN
sollValues[i].force:=0.0; outValues[i].force:=0.0;
            inertial:=FALSE; smoothstop:=FALSE;
    END;
   END;
  | GD.SMesCtrl :
    IF frictMes& ~measureDone[i] THEN
    moved[i]:= ABS(endpos[i]-d[i].istValues.pos)>epsilon[i];
          IF moved[i] THEN messforce[i]:= outValues[i].force; measureDone[i]:=
          TRUE; outValues[i].force:= 0.0 END;
    (* ELSE do nothing *)
    END;
    ELSE
    HALT(GD.Main3Halt);
   END;
END;
```

Existem 6 possibilidades para o estado do controlador. *SNoCtrl* quando nenhum tipo de controlador estiver atuando, *SSychCtrl* utilizado exclusivamente durante o processo de sincronização, *SPISpeedCtrl* quando o controle de velocidade estiver sendo empregado, *SStateCtrl* quando o controle de posição estiver sendo empregado, *SForceCtrl* quando o controle de força estiver sendo empregado e *SMesCtrl* utilizado apenas durante o processo de medição dos parâmetros para o

estabelecimento das curvas de atrito. Vamos concentrar o interesse no *SStateCtrl* pois ele faz o controle de posição que é mais freqüentemente utilizado.

```
| GD.SStateCtrl :
        outValues[i].force:= stateCtrl[i].Algo(d[i].istValues.pos, sollValues[i].pos,
        d[i].istValues.speed, sollValues[i].speed);
(* teste do novo state control
        outValues[i].force:= piSpeedCtrl[i].Algo(d[i].istValues.pos,
        d[i].maxValues.acc, d[i].istValues.speed, outValues[i].speed);
*)
```

É importante verificar que nenhum cálculo é feito nestas linhas. Elas apenas chamam o método *Algo* associado ao objeto *stateCtrl[i]*. Neste método é que todos os cálculos de controle são executados. Ao chamar *Algo*, informamos a posição e velocidades atuais e desejados. A partir destes dados é computado o valor de controle para cada uma das juntas. O valor de controle da junta 0 é armazenado em *outValues[0].force* e o da junta 1 em *outValues[1].force*. As linhas que aparecem como comentário faziam parte de um controlador anteriormente utilizado. Podem ser excluídas.

A grande vantagem desta formulação é que se quisermos alterar o algoritmo de controle precisamos alterar apenas o método *Algo* do módulo *StateCtrl*. O módulo *Main3* não precisa ser alterado.

```
(* Roboter Drive 2-3, drive z & drive phiz *)
kin23.TransQ23PToZPhiP
   (d[2].istValues.pos, d[3].istValues.pos, istZPhiZValues[2].pos, istZPhiZValues[3].pos);
kin23.TransQ23VToZPhiV
   (d[2].istValues.speed, d[3].istValues.speed, istZPhiZValues[2].speed,
   istZPhiZValues[3].speed);
```

A junta 2 apresenta como particularidade a transformação do movimento angular em movimento linear. Além disso, como descrito anteriormente, existe o acoplamento entre os movimentos das juntas 2 e 3. Quando falamos de *ZphiZValues* estamos nos referindo aos valores de posição, velocidade e força das juntas após a transformação do movimento angular em linear e considerado o acoplamento. Quando falamos em *istValues* estamos nos referindo aos valores de posição, velocidade e força

das engrenagens, após a redução feita em relação ao motor mas antes da transformação do movimento angular em linear e desconsiderando o acoplamento. Para nos referenciarmos a estes últimos valores usaremos a palavra *Drive*. O controle das juntas 2 e 3 é feito nos *Drives*. No entanto, no processo de sincronização trabalhamos com o espaço de juntas. As linhas acima apresentadas transformam os valores atuais de posição e velocidade do espaço dos *Drives* para o espaço de juntas, para que seus valores possam ser utilizados no processo de sincronização.

```
FOR i:=2 TO 3 DO
 CASE ctrlStatus[i] OF
  GD.SPreSynchCtrl :
   IF synchCtrl[i].synchPara.synchSpeedFast>0 THEN
     synchCtrl[i].inSynchSW:= d[i].swmax
   ELSE synchCtrl[i].inSynchSW:= d[i].swmin
   END;
   IF synchCtrl[i].inSynchSW THEN sollZPhiZValues[i].speed:= 0.0
   ELSE sollZPhiZValues[i].speed:= synchCtrl[i].synchPara.synchSpeedFast
   END;
  | GD.SSynchCtrl :
   IF synchCtrl[i].synchPara.synchSpeedFast>0 THEN
     synchCtrl[i].inSynchSW:= d[i].swmax
   ELSE synchCtrl[i].inSynchSW:= d[i].swmin
   END;
   sollZPhiZValues[i].speed:= synchCtrl[i].Algo(istZPhiZValues[i].pos);
   IF synchCtrl[i].message=SynchCtrl.SynchNow THEN
    IF i=2 THEN
              ASSERT (synchCtrl[3].message=SynchCtrl.Synched, 123);
                   (* Ueberpuefen ob Achse3 synchronisiert wurde *)
                    (* z-achse synchpos wird mit dem momentanen istwert von drive3
          berechnet; drive3 muss schon synchronisiert sein *)
          kin23.TransQ23PToZPhiP
                    (d[2].istValues.pos,                    d[3].istValues.pos-
                    synchCtrl[3].synchPara.synchPos+        synchCtrl[3].synchdist,
                    synchCtrl[2].synchdist, dummy);
                     kin23.TransZPhiPToQ23P      (synchCtrl[2].synchPara.synchPos,
                    istZPhiZValues[3].pos,  synch23Values[2].pos, dummy);
                    d[2].counter.Reset(synch23Values[2].pos);
                        d[2].istValues.pos:= synch23Values[2].pos;
                       oldValues[2].pos:= d[2].istValues.pos;
                        piSpeedCtrl[2].Init(d[2].istValues.pos);
    ELSIF i=3 THEN
                    ASSERT (synchCtrl[2].message#SynchCtrl.Synched, 124);
                    (* Sicherstellen, dass Achse2 noch nicht synchronisiert wurde *)
                   synchCtrl[3].synchdist:= d[3].istValues.pos;
                   d[3].counter.Reset(synchCtrl[3].synchPara.synchPos);
                    d[3].istValues.pos:= synch23Values[3].pos;
                    oldValues[3].pos:= d[3].istValues.pos;
                     piSpeedCtrl[3].Init(d[3].istValues.pos);
    END;
    synchCtrl[i].message:= SynchCtrl.SynchNowDone;
```

```
    END;
  ELSE
   (* do nothing *)
  END;
 END;
```

As linhas acima descrevem o controle utilizado apenas no processo de sincronização das juntas 2 e 3.

```
kin23.TransZPhiFToQ23F
   (sollZPhiZValues[2].force,    sollZPhiZValues[3].force,    sollValues[2].force,
   sollValues[3].force);
kin23.TransZPhiVToQ23V
    (sollZPhiZValues[2].speed,   sollZPhiZValues[3].speed,   sollValues[2].speed,
    sollValues[3].speed);
kin23.TransZPhiPToQ23P
    (sollZPhiZValues[2].pos, sollZPhiZValues[3].pos, sollValues[2].pos, sollValues[3].pos);
```

Estas linhas transformam os valores desejados de posição, força e velocidade do espaço de juntas para o espaço dos *Drives*. Na geração de valores desejados trabalhamos normalmente no espaço de juntas. Como o controle é feito no espaço dos *Drives*, temos de transformar os valores desejados para este espaço.

```
FOR i:=2 TO 3 DO
 CASE ctrlStatus[i] OF
  GD.SNoCtrl :
   outValues[i].force:= 0.0;
  | GD.SPreSynchCtrl :
   IF ~enabled[i] THEN piSpeedCtrl[i].piPara.integ:= 0.0 END;
   outValues[i].force:= piSpeedCtrl[i].Algo
    (d[i].istValues.pos, d[i].maxValues.acc, d[i].istValues.speed, sollValues[i].speed);
  | GD.SSynchCtrl :
   IF ~enabled[i] THEN piSpeedCtrl[i].piPara.integ:= 0.0 END;
   outValues[i].force:= piSpeedCtrl[i].Algo
    (d[i].istValues.pos, d[i].maxValues.acc, d[i].istValues.speed, sollValues[i].speed);
  | GD.SPISpeedCtrl :
   IF ~enabled[i] THEN piSpeedCtrl[i].piPara.integ:= 0.0 END;
   outValues[i].force:= piSpeedCtrl[i].Algo
    (d[i].istValues.pos, d[i].maxValues.acc, d[i].istValues.speed, sollValues[i].speed);
   IF frictMes THEN
    IF posdir[i] THEN posreached[i]:= istZPhiZValues[i].pos>sollZPhiZValues[i].pos
    ELSE posreached[i]:= istZPhiZValues[i].pos<sollZPhiZValues[i].pos
    END;
    IF posreached[i] THEN
```

```
                IF ~measureDone[i] THEN messforce[i]:= outValues[i].force;
          messspeed[i]:= d[i].istValues.speed; measureDone[i]:= TRUE END;
        sollZPhiZValues[i].speed:= 0.0
       END;
      END;
    | GD.SStateCtrl :
          outValues[i].force:= stateCtrl[i].Algo(d[i].istValues.pos, sollValues[i].pos,
      d[i].istValues.speed, sollValues[i].speed);
      (* teste do novo state control
          outValues[i].force:= piSpeedCtrl[i].Algo(d[i].istValues.pos,
      d[i].maxValues.acc, d[i].istValues.speed, outValues[i].speed);
      *)
    | GD.SForceCtrl :
      outValues[i].force:= forceCtrl[i].Algo(sollValues[i].force, sollValues[i].force);
    | GD.SMesCtrl :
      IF frictMes& ~measureDone[i] THEN
       moved[i]:= ABS(endpos[i]-istZPhiZValues[i].pos)>epsilon[i];
            IF moved[i] THEN measureDone[i]:= TRUE; messforce[i]:=
        outValues[i].force; outValues[i].force:= 0.0 END;
      (* ELSE do nothing *)
      END
    ELSE
     HALT(GD.Main3Halt);
    END;
  END;
```

Este bloco faz o controle das juntas 2 e 3 e é semelhante ao gerenciador de controladores utilizados para as juntas 0 e 1. Apresenta algumas diferenças no controle durante o processo de sincronização e no controle de força. O controlador *SStateCtrl* utiliza o mesmo algoritmo utilizado para as juntas 0 e 1 apenas com ganhos diferentes.

```
FOR i:=0 TO MaxDrives-1 DO
 (* max drive force limiting *)
 IF outValues[i].force>d[i].maxValues.force THEN
   outValues[i].force:= d[i].maxValues.force
 ELSIF outValues[i].force<-d[i].maxValues.force THEN
   outValues[i].force:= -d[i].maxValues.force
 END;
END;
```

Estas linhas verificam se o valor de controle não é superior ao valor máximo permitido e o limitam se for o caso.

```
istpos[0]:= d[0].istValues.pos; istpos[1]:= d[1].istValues.pos;
```

```
istpos[2]:= istZPhiZValues[2].pos; istpos[3]:= istZPhiZValues[3].pos;
istspeed[0]:= d[0].istValues.speed; istspeed[1]:= d[1].istValues.speed;
istspeed[2]:= istZPhiZValues[2].speed; istspeed[3]:= istZPhiZValues[3].speed;
FOR i:=0 TO MaxDrives-1 DO
  solltorq[i]:= outValues[i].force;
END;
```

Copia os valores atuais de posição para o vetor *istpos[i]*, os valores atuais de velocidade para o vetor *istspeed[i]* e os valores de controle para o vetor *solltorq[i]*. A única função é facilitar a utilização destes valores, utilizando um vetor de nome comum para as quatro juntas.

```
IF dynComp THEN dynamic.Comp(istpos, istspeed, solltorq, dynacomptorq);
END;
```

Fornece valores de controle considerando a compensação dinâmica, os quais são armazenados no vetor *dynacomptorq*. Estes valores estão sendo utilizados apenas para plotagem e não como parte integrante do controlador.

```
FOR i:=0 TO MaxDrives-1 DO
  absistspeed[i]:= ABS(istspeed[i]);
  IF istspeed[i]<0 THEN sign:= -1.0 ELSE sign:= 1.0 END;
        frictforce[i]:=piSpeedCtrl[i].frictPara.offset+(sign*(piSpeedCtrl[i].frictPara.a+p
    iSpeedCtrl[i].frictPara.b*SHORT(absistspeed[i]))*(1.0-M.exp(-
    piSpeedCtrl[i].frictPara.c*SHORT(absistspeed[i])));
```

Calcula os torques decorrentes da compensação de atrito e os armazena no vetor *frictforce*.

```
  (* d[i].ampl.Write(SHORT(dynacomptorq[i]+frictforce[i])); *)
```

Soma os torques decorrentes da compensação dinâmica e da compensação de atrito. Permanece como comentário enquanto a compensação dinâmica não for efetivada.

```
  oldoutForce[i]:= outValues[i].force;
  d[i].ampl.Write(outValues[i].force+frictforce[i]);
END;
```

Estas linhas transmitem para os amplificadores os valores de controle e encerram o loop criado pelo último *IF*.

```
(*
(** BEGIN TESTHACK *)
IF (testdrive>=0)&(testdrive<MaxDrives) THEN
  ADDAEltec.WriteDA(0, d[testdrive].istValues.speed*10.0);
  ADDAEltec.WriteDA(1, outValues[testdrive].force/33.0);
END;
(** END TESTHACK *)
*)
```

Estas linhas foram utilizadas quando se acoplou ao robô um gerador de pulsos para alguns testes de vibração. Não está mais sendo utilizado no momento.

```
END Controller;
```

Fim da tarefa *Controller*.

O procedimento *Controller* é grande e bastante complexo e a análise individual de cada uma de suas linhas extrapolaria o objetivo deste capítulo que é fornecer uma visão geral sobre ele. A descrição de função de cada bloco visa facilitar a tarefa do leitor que tiver necessidade de compreender detalhadamente este procedimento para efetuar mudanças futuras neste módulo.

C.5 Módulos do Controle Híbrido

A seguir são descritos os principais módulos do controle híbrido.

C.5.1 Módulo CartesianTransf

Iniciaremos analisando o módulo *CartesianTransf*.

```
MODULE CartesianTransf;
(** Autor: L.Weihmann, R.Hueppi; Date: 971028 *)
(** Version 971028 *)
```

```
IMPORT Math:=MathL, Base, O:= Objects, Parser:=ConfigP, GD := GlobalDefs, XOK:=
PPCXOKernel;

CONST
 Version = 971028;
 X* = 0; Y* = 1; Z* = 2; Phi* = 3;
 Mz*=3; My*=4; Mx*=5;
```

Inicia o módulo, define os módulos a serem importados e as constantes.

```
TYPE
 RotMatrix* = ARRAY 2,2 OF LONGREAL;
 Frame* = POINTER TO RECORD
    r*, rt* : RotMatrix;
    teta*, x0Task*, y0Task* : LONGREAL
  END;

 CarthTrans* = POINTER TO CarthTransDesc;
 CarthTransDesc* = RECORD (Base.ObjDesc)
  frame*, swapframe : Frame;
 END;
```

Define os tipos *RotMatrix*, *Frame* e o objeto *CarthTrans*. *RotMatrix* é uma matriz 2x2. O tipo *Frame* define o espaço cartesiano através dos campos *teta*, *x0Task* e *y0Task* que na nomenclatura utiliza na teoria apresentada de controle híbrido referem-se respectivamente a x_{OT}, y_{OT} e θ_T. As matrizes 2x2 *r* e *rt* fazem justamente a transformação entre forças e velocidades na direção x e y pois as velocidades na direção z e phi, as forças na direção z e os momentos em torno de z são idênticos no espaço da tarefa e no espaço cartesiano e portanto não necessitam ser transformados.

O objeto *CarthTrans* possui dois campos do tipo *Frame*. Um destes campos define o espaço de tarefas e o outro é utilizado apenas quando se quer alterar este espaço.

```
VAR
 version- : LONGINT;
```

Cria a variável version.

```
PROCEDURE (c : CarthTrans) Assign* (o : Base.Object; name : ARRAY OF CHAR) :
BOOLEAN;
PROCEDURE AttrMsg(obj: CarthTrans; VAR M: O.AttrMsg);
PROCEDURE Handler*(obj: O.Object; VAR M: O.ObjMsg);
```

```
PROCEDURE NewCarthTrans*;
PROCEDURE DefCarthTrans*;
```

Estes procedimentos são utilizados na configuração de objetos.

```
PROCEDURE (c : CarthTrans) RotationalCartTask*(teta, x0Task, y0Task : LONGREAL);
```

Este procedimento calcula as matrizes *r* e *rt* que fazem as transformações entre os sistemas da tarefa e o operacional a partir dos dados *teta*, *x0Task* e *y0Task*. Sempre que quisermos alterar algum destes dados e em conseqüência a sistema de coordenadas da tarefa, este procedimento deve ser chamado para que as novas matrizes *r* e *rt* sejam calculadas.

```
PROCEDURE (c : CarthTrans) PosCartToTask*(pCart : GD.Vector4; VAR pTask :
GD.Vector4);
```

Dada a posição do manipulador no espaço cartesiano, este procedimento calcula a posição no espaço da tarefa.

```
PROCEDURE (c : CarthTrans) PosTaskToCart*( pTask : GD.Vector4; VAR pCart :
GD.Vector4);
```

Dada a posição no espaço da tarefa, este procedimento calcula a posição no espaço operacional.

```
PROCEDURE (c : CarthTrans) VelCartToTask*( vCart : GD.Vector4; VAR vTask :
GD.Vector4);
```

Dada a velocidade no espaço operacional, calcula a velocidade no espaço da tarefa.

```
PROCEDURE (c : CarthTrans) VelTaskToCart*( vTask : GD.Vector4; VAR vCart :
GD.Vector4);
```

Dada a velocidade no espaço da tarefa, calcula a velocidade no espaço operacional.

PROCEDURE (c : CarthTrans) ForceToolToCart*(fTool : GD.Vector4; pr : GD.Vector4; VAR fCart : GD.Vector4);

Dada a força no espaço do efetuador final e a posição do manipulador, calcula a força no espaço operacional.

PROCEDURE (c : CarthTrans) ForceCartToTask*(fCart : GD.Vector4; VAR fTask : GD.Vector4);

Dada a força no espaço operacional, calcula a força no espaço da tarefa.

PROCEDURE (c : CarthTrans) ForceTaskToCart*(fTask : GD.Vector4; VAR fCart : GD.Vector4);

Dada a força no espaço da tarefa, calcula a força no espaço operacional.

END CartesianTransf.

Termina o módulo.

As transformações entre o espaço de juntas e o espaço operacional são executadas dentro do módulo *SCARACarthKin*.

Para configurar o objeto *CarthTrans* podemos utilizar por exemplo o seguinte comando:

CartesianTrans.DefCarth Frame0 (teta=0.665, x0Task=0.0, y0Task=0.03) ~

Ao executar este comando criaríamos o objeto *Frame0* que define o espaço da tarefa através do giro teta e dos deslocamento *x0Task* e *y0Task*. Este objeto permanece na biblioteca até que seja buscado por algum outro módulo. É possível criar vários objetos do tipo *CarthTrans* e desta forma definir diferentes espaços da tarefa. Assim,

de acordo com a tarefa a ser executado, poderíamos buscar da biblioteca o espaço da tarefa desejado que já estaria pré-defímido.

C.5.2 Módulo HybridCtrl

Passaremos a seguir à análise do módulo *HybridCtrl.*

```
MODULE HybridCtrl;
 (** ETHZ Institut fuer Robotik *)
 (** Autor: R.Hueppi, L.Weihmann; Date: 970811 *)
 (** Version 970811 *)

IMPORT
 Base, O:= Objects, Parser:= ConfigP, Math:=MathL, ForceCtrl, PISpeedCtrl,
        D:= Drive, SCK:= SCARACarthKin, GD:= GlobalDefs, CT:=CartesianTransf,
StateCtrl, XOK:= PPCXOKernel;

CONST
 Version = 970811;
 X* = 0; Y* = 1; Z* = 2; Phi* = 3;
 Force* = 0; Speed* = 1; Pos* = 2; NoCtrl* = 3;
```

Inicia o módulo, faz alguns comentários, define os módulos a serem importados e as constantes.

```
TYPE
 SelMatrix = POINTER TO RECORD
     speed, force : ARRAY 4,4 OF LONGREAL;
   END;
 Ctrl* = ARRAY 4 OF INTEGER;
 HC*=POINTER TO HCDesc;
 HCDesc*=RECORD(Base.ObjDesc)
  clock* : REAL; (* hybrid controller clock in sec *)
  kin* : SCK.CarthKin;
  frame* : CT.CarthTrans;
  fCtrl* : ARRAY 4 OF ForceCtrl.FCtrl;
  piSpeedCtrl* : ARRAY 4 OF PISpeedCtrl.PISpeedCtrl;
  stateCtrl* : ARRAY 4 OF StateCtrl.StateCtrl;
  selMat*, selMatSwap : SelMatrix;
  d* : ARRAY 4 OF D.Drive;
  ctrl* : Ctrl;
  newCtrl* : BOOLEAN;
 END;
```

Define o tipo *SelMatrix* que possui os campos *speed* e *force* que são matrizes 4x4. Estas matrizes serão utilizadas para fazer a seleção das direções onde se deseja controlar a força e a velocidade.

Define o tipo *Ctrl* que é um vetor de quatro componentes.

Define o objeto *HC*. Este objeto é a base do funcionamento do controle híbrido. Veremos o significado de cada um de seus campos.

O campo *clock* define o clock utilizado durante o controle híbrido.

O campo *kin* define uma variável do tipo *SCK.CarthKin*. Desta forma podemos fazer uso dos métodos associados a este objeto que são utilizados na transformação de coordenadas.

O campo *frame* define uma variável do tipo *CT.CarthTrans* cuja finalidade foi discutida anteriormente.

O campo *fctrl* define um vetor de quatro componentes do tipo *ForceCtrl.FCtrl*. Este tipo define os ganhos no controle de força e a ele está associado o método *Algo* utilizado no controle de força.

O campo *piSpeedCtrl* define o vetor de quatro componentes do tipo *PISpeedCtrl.PISpeedCtrl*. Este tipo define os ganhos do controle de velocidade e a ele está associado o método *Algo* utilizado no controle de velocidade.

O campo *stateCtrl* define um vetor de quatro componentes do tipo *StateCtrl.StateCtrl*. Este tipo define os ganhos do controlador de posição e a ele está associado o método *Algo* utilizado no controle de posição.

Os campos *selMat* e *selMatSwap* são do tipo *SelMatrix* definido anteriormente. *SelMat* é efetivamente utilizado nos cálculos, enquanto que *selMatSwap* é utilizado apenas no processo para alterar as matrizes de seleção quando desejado.

O campo d define um vetor de quatro componentes do tipo *D.Drive*. O objeto *D.Drive* já foi amplamente discutido no capítulo 3.

O campo *ctrl* é do tipo *Ctrl*. Este campo é utilizado para definir qual o tipo de controle que deve ser utilizado em cada uma das direções do espaço da tarefa. A partir das informações contidas neste campo serão calculadas as matrizes de seleção.

O campo *newCtrl* define apenas uma variável do tipo booleana.

```
VAR
 version- : LONGINT;
```

Define a variável global version.

```
PROCEDURE (c : HC) Assign* (o : Base.Object; name : ARRAY OF CHAR) : BOOLEAN;
PROCEDURE AttrMsg(obj: HC; VAR M: O.AttrMsg);
PROCEDURE Handler*(obj: O.Object; VAR M: O.ObjMsg);
PROCEDURE NewHC;
PROCEDURE DefHC*;
```

Procedimentos utilizados na configuração do objeto *HC*.

```
PROCEDURE (c : HC) NewCtrl*(ctrl : ARRAY OF INTEGER);
```

Este procedimento deve ser utilizado toda vez que quisermos alterar o tipo de controle a ser utilizado nas direções cartesianas do espaço da tarefa. Ao receber a variável *ctrl*, este procedimento recalcula as matrizes de seleção.

O método *Algo* apresentado a seguir faz todo o gerenciamento de transformações entre os diferentes sistemas de coordenadas. Além disso este método chama os algoritmos de controle que são métodos associados aos objetos *ForceCtrl*, *PISpeedCtrl* e *StateCtrl*. Faremos a análise deste método dividindo-o em blocos.

```
PROCEDURE (c : HC) Algo*(istposR, istspeedR, istforceT, sollPosF, sollSpeedF, sollAccF,
sollForceF : GD.Vector4; VAR outForceR : GD.Vector4);
```

Inicia o método *Algo* associado ao objeto *HC*. Ao chamar este método devemos informar a posição do robô e a velocidade no espaço de juntas, as forças aplicadas no robô lidas no espaço do efetuador final, a posição, velocidade, aceleração e força desejadas no espaço da tarefa. Como resultado este método fornece os torques de controle já no espaço de juntas.

```
VAR
 i : LONGINT; done, rechts  : BOOLEAN;
 pCart,  vCart, istposC, istspeedC, istposT, istspeedT, pFrame, vFrame,
 outForceFrame, outSpeedFrame, outSpeed, outPosFrame, outForce : GD.Vector4;
 fFrame, fCart, outTForceFrame : GD.Vector4;
```

```
trigo : SCK.Trigotyp; jak : SCK.JakobiMatrix;
```

Define as variáveis globais.

```
BEGIN
 (* transformations *)
 c.kin.TrigoCalc(istposR[0], istposR[1], trigo);
 c.kin.PosRobotToCart(istposR, trigo, pCart, rechts);
 c.kin.JakobiCalc(trigo, jak);
 c.kin.SpeedRobotToCarth(istspeedR,  jak, vCart);
 c.frame.PosCartToTask(pCart, pFrame);
 c.frame.VelCartToTask(vCart, vFrame);
 c.frame.ForceToolToCart(istforceT,  istposR, fCart);
 c.frame.ForceCartToTask( fCart, fFrame);
```

Faz todas as transformações necessárias a partir dos dados atuais e dos desejados.

```
(* initialisations *)
IF c.newCtrl OR c.frame.swapped THEN
  c.newCtrl:= FALSE; c.frame.swapped:= FALSE;
  FOR i:= 0 TO 3 DO
    CASE c.ctrl[i] OF
      Speed : c.piSpeedCtrl[i].Init(SHORT(pFrame[i]));
      | Pos : c.stateCtrl[i].Init(SHORT(pFrame[i]));
      | Force : c.fCtrl[i].Init();
    ELSE
      HALT(GD.HybridCtrlHalt)
    END;
  END;
END;
```

Caso sejam definidas novas direções de controle ou caso seja alterado o espaço da tarefa, os métodos *Init* associados aos objetos *PISpeedCtrl*, *StateCtrl* e *ForceCtrl* devem ser executados para que as informações a respeito do posicionamento atual sejam atualizadas.

```
(* controllers *)
FOR i:= 0 TO 3 DO
  CASE c.ctrl[i] OF
                          (* como o sensor de força ainda não foi instalado,
                istforce=sollforce. Isto significa que fFrame[i]=sollForce[i]*)
                          Force : outForceFrame[i]:= c.fCtrl[i].Algo(SHORT(sollForceF[i]),
                SHORT(sollForceF[i]));
    | Speed : outSpeedFrame[i]:= c.piSpeedCtrl[i].Algo
                                  (SHORT(pFrame[i]),SHORT(sollAccF[i]),
                          SHORT(vFrame[i]), SHORT(sollSpeedF[i]));
    | Pos :  outSpeed[i]:= c.stateCtrl[i].Algo
```

```
                                    (SHORT(pFrame[i]),          SHORT(sollPosF[i]),
                              SHORT(vFrame[i]), SHORT(sollSpeedF[i]));
    | NoCtrl : outForce[i]:= 0.0;
   ELSE
    HALT(GD.HybridCtrlHalt)
   END;
  END;
```

Sabendo qual o tipo de controle a ser executado em cada uma das direções cartesianas do espaço de tarefa fornecido pela variável *c.ctrl[i]*, estas linhas fazem com que apenas o controlador adequado seja chamado. Por exemplo, se na direção x quisermos controlar apenas força, será chamado o método *c.fCtrl.Algo*. Como resultado destas linhas temos o valor de controle em cada direção do espaço de tarefas.

```
(* transformation controller forces to frame forces *)
FOR i:= 0 TO 3 DO
  outForceFrame[i]:= outForceFrame[i]*c.selMat.force[i,i];
  outSpeedFrame[i]:= outSpeedFrame[i]*c.selMat.speed[i,i];
  outTForceFrame[i]:=outForceFrame[i]+outSpeedFrame[i];
END;
```

Estas linhas aplicam as matrizes de seleção sobre os valores de controle. Em princípio não necessitaríamos de matrizes de seleção pois os valores de controle são calculados já levando em consideração as direções selecionadas. No entanto, estas matrizes são importante para garantir que o valor de controle seja zero para a combinação tipo de controle/direção em que não se deseja fazer o controle. Do contrário, valores calculados anteriormente poderiam entrar incorretamente na composição do valor final de controle.

```
(* back transformations *)
c.frame.ForceTaskToCart(outTForceFrame, fCart);
c.kin.ForceCarthToRobot(fCart, jak,  outForceR);
END Algo;
```

Estas linhas transformam os valores de controle do espaço de tarefas para o espaço de juntas.

```
PROCEDURE (c : HC) Init*();
VAR  i : LONGINT;  minq, maxq : GD.Vector4;
BEGIN
 FOR i:= 0 TO 3 DO
   minq[i]:= c.d[i].minValues.pos; maxq[i]:= c.d[i].maxValues.pos
 END;
 c.kin.Init(minq, maxq);
```

```
 c.NewCtrl(c.ctrl);
END Init;
```

Inicializa o controle híbrido, determinando os valores máximos de posição para as juntas e chamando os métodos *Init* do objeto *CarthKin* e *NewCtrl* do objeto *HC*. Toda vez que se inicia o controle híbrido este último método deve ser chamado para que as matrizes de seleção sejam calculadas.

```
BEGIN
 version := Version;
END HybridCtrl.
```

Termina o módulo HybridCtrl.

C.5.3 Módulo HCTest

Apesar de conter as bases para o funcionamento do controle híbrido, o módulo *HybridCtrl* não possui nenhum procedimento para interface com o usuário e para fornecer os valores de controle para o módulo *Main3*, de onde são enviados para os amplificadores. Estas tarefas são executadas pelo módulo *HCTest* analisado a seguir.

```
MODULE HCTest;
 (** Autor: L.Weihmann, R.Hueppi; Date: 2.12.97 *)
 (** Version 2 12 97 *)

IMPORT
 XT:= XTexts, Base, XO:=XOberon, M:=Main3, HCtrl:= HybridCtrl, GD := GlobalDefs,
 XOK:= PPCXOKernel, PISpeedCtrl, StateCtrl, ForceCtrl, Math:= MathL;

CONST
 Version = 21296; X=0; Y=1; Z=2; Phi=3;

VAR
 version- : LONGINT;
 hc : HCtrl.HC;
 hcev : XOK.EveryEvent;
 mainhcev : XOK.MainEvent;
 start, init, run : BOOLEAN;
 hcAuthority : LONGINT;
        istposR, istspeedR, istforceT, outForceR, sollPosF, sollSpeedF, sollAccF, sollForceF,
nullvector : GD.Vector4;
        piPara : PISpeedCtrl.PIPara; stateCtrlPara : StateCtrl.StateCtrlPara; forcePara :
ForceCtrl.FPara;
 ctrl: HCtrl.Ctrl;
```

Inicia o módulo, faz alguns comentários, define os módulos a serem importados, define constantes e variáveis globais.

PROCEDURE GetHC(name : ARRAY OF CHAR; VAR hc : HCtrl.HC) : BOOLEAN;

Este procedimento busca da biblioteca um objeto do tipo *HC*, cujo nome é especificado durante a chamada através da variável *name,* e este objeto passa a ser utilizado com o nome da variável *hc* também especificada durante a chamada.

PROCEDURE ChangeAxisSpeedAcc*(haxis : LONGINT; speed, acc : REAL) : BOOLEAN;

Este procedimento altera a velocidade na direção desejada e retorna a variável booleana *TRUE* caso a operação tenha sido realizada com sucesso.

PROCEDURE ChangeAxisForce*(haxis : LONGINT; force : REAL) : BOOLEAN;

Este procedimento altera a força na direção desejada e retorna a variável booleana *TRUE* caso a operação tenha sido realizada com sucesso.

PROCEDURE ChangeTeta*(teta : REAL) : BOOLEAN;

Este procedimento altera a variável teta, definindo um novo espaço da tarefa. No interior deste procedimento é chamado o método *RotationCartTask* para que as novas matrizes de transformação *r* e *rt* sejam calculadas.

PROCEDURE ChangePIPara*(haxis : LONGINT; p, i, iLimit : REAL) : BOOLEAN;

Este procedimento altera os ganhos do controlador de velocidade na direção desejada do espaço da tarefa.

PROCEDURE ChangeStatePara*(haxis : LONGINT; kPos, acc, systemdelay, i, iLimit, inertia : REAL) : BOOLEAN;

Este procedimento altera os ganhos do controlador de posição na direção desejada do espaço da tarefa.

PROCEDURE ChangeForcePara*(haxis : LONGINT; p, i, d, iLimit : REAL) : BOOLEAN;

Este procedimento altera os ganhos do controlador de força na direção desejada do espaço da tarefa.

PROCEDURE TSetAxisSpeedAcc*; (** haxis, speed acc *)

Este procedimento é acionado através de chamada externa onde é informada a direção do espaço da tarefa em que se deseja alterar a velocidade, a nova velocidade desejada e a nova aceleração desejada. Estes dados são lidos através da utilização do Scanner e enviados para o procedimento *ChangeAxisSpeedAcc* comentado anteriormente que efetivamente altera a velocidade desejada.

PROCEDURE TSetAxisForce*; (** haxis, force *)

Este procedimento é acionado através de chamada externa onde é informada a direção do espaço da tarefa em que se deseja alterar a força e a nova força desejada. Estes dados são lidos através da utilização do Scanner e enviados para o procedimento *ChangeAxisForce* comentado anteriormente que efetivamente altera a força desejada.

PROCEDURE TSetTeta*; (** teta *)

Este procedimento é acionado através de chamada externa onde é informado o novo ângulo teta desejado. Este dado é lido através da utilização do Scanner e enviado para o procedimento *ChangeTeta* comentado anteriormente que efetivamente altera o ângulo teta.

PROCEDURE TSetSpeedCtrlPara*; (** haxis, p, i, iLimit *)

Este procedimento é acionado através de chamada externa onde é informada a direção do espaço da tarefa em que se deseja alterar os parâmetros do controlador de velocidades e os novos parâmetros a serem utilizados. Estes dados são lidos através da utilização do Scanner e enviados para o procedimento *ChangePIPara* comentado anteriormente para que estes parâmetros sejam efetivamente alterados.

PROCEDURE TSetForceCtrlPara*; (** haxis, p, i, d, iLimit *)

Este procedimento é acionado através de chamada externa onde é informada a direção do espaço da tarefa em que se deseja alterar os parâmetros do controlador de força e os novos parâmetros a serem utilizados. Estes dados são lidos através da utilização do Scanner e enviados para o procedimento *ChangeForcePara* comentado anteriormente para que estes parâmetros sejam efetivamente alterados.

PROCEDURE TSetStateCtrlPara*; (** haxis, kPos, acc, systemdelay, i, iLimit, inertia *)

Este procedimento é acionado através de chamada externa onde é informada a direção do espaço da tarefa em que se deseja alterar os parâmetros do controlador de posição e os novos parâmetros a serem utilizados. Estes dados são lidos através da utilização do Scanner e enviados para o procedimento *ChangeStatePara* comentado anteriormente para que estes parâmetros sejam efetivamente alterados.

PROCEDURE TChangeCtrl*; (** haxis, controller ('pos' 'speed' 'force' 'noctrl') *)

Este procedimento é utilizado para alterar através de chamada externa o tipo de controlador utilizado numa das direções do espaço da tarefa. Estes dados são lidos com a utilização do Scanner e enviados para o método *NewCtrl* que efetivamente executa a mudança, calculando as novas matrizes de seleção.

```
PROCEDURE HCCtrl(e : XOK.Event);
VAR
 dummy : BOOLEAN;
BEGIN
 run:= (M.status=GD.Run);
 IF run THEN
  (* M.GetSensForce(fTool6); *)
  M.GetRobPosSpeed(istposR, istspeedR);
                (* istforceT[X]:= fTool6[0]; istforceT[Y]:= fTool6[1]; istforceT[Z]:=
       fTool6[2]; istforceT[Phi]:= fTool6[5]; *)
  istforceT:= nullvector;
                hc.Algo(istposR, istspeedR, istforceT,sollPosF, sollSpeedF, sollAccF,
       sollForceF, outForceR);
  dummy := M.SetRobForce(hcAuthority, outForceR);
 ELSE dummy := M.SetRobForce(hcAuthority, nullvector);
```

```
 END;
END HCCtrl;
```

Esta tarefa é instalada pelo *EveryEvent* de nome *hcev* e tem um clock de 1 milisegundo. Inicialmente verifica se o robô está no estado *Run*. Depois faz a leitura dos valores atuais de força, posição e velocidade. As linhas referentes a leitura de força estão como comentário pois ainda não podem ser utilizadas. A linha *istforceT=nullvector* determina que todas forças atuais sejam iguais a zero. Após é chamado o método *Algo* discutido anteriormente. Do método *Algo* retorna a variável *outForceR* que são os valores de controle no espaço de juntas. Estes valores de controle são setados no módulo *Main3* através do comando *SetRobForce*. O número com a autorização também deve ser enviado.

```
PROCEDURE InitTask;
BEGIN
 sollForceF[X]:= 0.0;
        sollSpeedF[X]:= 0.0; sollAccF[X]:=  0.0; sollSpeedF[Y]:= 0.0; sollAccF[Y]:=  0.0
sollSpeedF[Z]:= 0.0; sollAccF[Z]:=  0.0;
 sollSpeedF[Phi]:= 0.0; sollAccF[Phi]:=  0.0;
 ctrl[X]:= HCtrl.Force; ctrl[Y]:= HCtrl.Speed; ctrl[Z]:= HCtrl.Speed; ctrl[Phi]:= HCtrl.Speed;
 hc.NewCtrl(ctrl); M.GetRobPosSpeed(istposR, istspeedR);
END InitTask;
```

Inicializa o robô quando este está parado, atribuindo o valor zero para velocidade e aceleração desejadas. Determina que seja feito o controle de força na direção x e o controle de velocidade nas direções y, z e phi. Faz a leitura da posição atual do robô.

```
PROCEDURE MainHCCtrl(e : XOK.Event);
VAR
 dummy : BOOLEAN;
BEGIN
 REPEAT
   IF ABS(istposR[1])<0.15 THEN start := FALSE END;
   IF ABS(istposR[1])>1.8 THEN start := FALSE END;
   IF ABS(istposR[0])>2.1 THEN start := FALSE END;
 UNTIL ~start OR ~run ;
 start:= FALSE; hcev.UnInstall;
 dummy := M.SetRobForce(hcAuthority, nullvector);
 dummy := M.AuthorityFree(hcAuthority);
 dummy:= M.ChangeCtrl(GD.SPISpeedCtrl);
END MainHCCtrl;
```

Esta tarefa é instalada pelo *MainEvent* de nome *mainhc*. Fica no interior do loop *REPEAT-UNTIL* até que as variáveis *run* ou *start* sejam *FALSE*. No interior do

loop fica verificando se alguns limites de posição não são ultrapassados. Estes limites evitam que o robô entre em posições singulares. Ao sair do loop, desinstala o *EveryEvent*, seta a força zero para cada junta, libera a autorização e muda o controlador do módulo *Main3* para controlador de velocidade. Durante o controle híbrido, o controlador do módulo *Main3* deve obrigatoriamente estar em controle de força.

PROCEDURE StartHC*;

Este procedimento inicia o funcionamento do controle híbrido. Muda o controlador do módulo *Main3* para controle de força, pede a autorização para poder setar os valores desejados de controle, chama o procedimento *InitTask* e instala as tarefas *MainHCCtrl* e *HCCtrl*. Se alguma destas atividades não pode ser completada com sucesso, envia uma mensagem de erro para a tela.

PROCEDURE StopHC*;

Este procedimento para o funcionamento do controle híbrido ao fazer com que a variável *start* assuma o valor *FALSE*. Envia dados para a tela, informando se a desinstalação foi feita com sucesso.

PROCEDURE Init*;

Este procedimento inicializa o controle híbrido ao chamar o procedimento *GetHC* que busca da biblioteca o objeto do tipo *HybridCtrl*. Este objeto é buscado com o nome *hc* e sempre que quisermos utilizar um campo deste objeto devemos usar o nome *hc*. Por exemplo, ao utilizar o campo *clock* devemos escrever *hc.clock*.

```
BEGIN
 version := Version; NEW(hcev); NEW(mainhcev); init:= FALSE;
 NEW(piPara); NEW(stateCtrlPara); NEW(forcePara);
END HCTest.
```

Aloca espaço na memória para os eventos *hcev* e *mainhcev* e para os objetos *piPara*, *stateCtrlPara* e *forcePara*. Termina o módulo.