Universidade Estadual de Campinas
Faculdade de Engenharia Mecânica
Departamento de Projeto Mecânico

Um Ambiente Integrado de Desenvolvimento de Software Aplicado à Robótica

Por: **Luiz Antônio de Freitas Coutinho**

Orientador: **Prof. Dr. João Maurício Rosário**

Dissertação apresentada à Faculdade de Engenharia Mecânica - FEM - Unicamp como parte dos requisitos exigidos para a obtenção do título de *MESTRE em ENGENHARIA MECÂNICA*.

Campinas - 1993

UNIVERSIDADE ESTADUAL DE CAMPINAS
FACULDADE DE ENGENHARIA MECÂNICA
DEPARTAMENTO DE PROJETO MECÂNICO

Tese de Mestrado

Um Ambiente Integrado de Desenvolvimento de Software Aplicado à Robótica

Autor: Luiz Antônio de Freitas Coutinho

Orientador: Prof. Dr. João Maurício Rosário

Aprovado por:

Prof. Dr. João Maurício Rosário - Presidente

Prof. Dr. Geraldo Nonato Telles

Prof. Dr. Paulo Martins Engel

Campinas-SP, 17 de Setembro de 1993

Dedico este trabalho a meus pais, como prova de eterna gratidão pelo apoio e incentivo recebidos ao longo de toda minha vida.

Agradecimentos

Ao quase-irmão Joselito, pela eterna amizade, participação ativa em todas as fases da minha vida acadêmica e pelo seu espírito crítico, com o qual eu muito aprendi.

Ao Professor e amigo João Maurício Rosário, pela orientação, pelo exemplo de dedicação ao trabalho e por ter me confiado a responsabilidade de executar este projeto.

À Liziane, pelo carinho, por suas sugestões e sobretudo, pelas palavras de conforto e encorajamento nos momentos mais difíceis deste trabalho.

Ao Professor Paulo Amaral, que fez despertar em mim o gosto pela robótica.

Ao amigo Waldemar Scudeller Jr., que com suas "dicas" valiosas me poupou muitas dores-de-cabeça.

Aos amigos Fançony, Fábio, Oswaldo, Rober e Rogério pela agradável convivência durante o transcorrer do meu curso de pós-graduação.

A Todas as pessoas que, de uma forma ou de outra, contribuíram para que os objetivos deste trabalho fossem alcançados.

À Coordenação de Aperfeiçoamento de Pessoal de Ensino Superior (CAPES) pelo apoio financeiro.

Este trabalho é parte integrante de um projeto de cooperação entre a Universidade Estadual de Campinas, a Petrobrás e o Geesthacht Institut da Alemanha, que tem como objetivo a automatização de intervenções em poços de petróleo submarinos em águas profundas.

" Uma máquina pode fazer o trabalho de cinquenta pessoas comuns. Máquina alguma pode fazer o trabalho de um homem incomum."

Elbert Hubbard

Resumo

Nesta dissertação é elaborado e implementado um ambiente integrado de desenvolvimento de *software* aplicado à robótica.

Trata-se de um aplicativo que integra:

1. Uma linguagem de alto nível para programação *off-line* de robôs de propósitos gerais, totalmente compatível com a linguagem C;
2. Um módulo de *CAD* responsável pela modelagem gráfica do robô e seu ambiente de trabalho;
3. Uma biblioteca de funções relacionadas à robótica;
4. Um gerenciador de arquivos;
5. Um editor de textos.

Ao final do trabalho, implementa-se alguns programas em um manipulador industrial com seis graus de liberdade

Abstract

This dissertation presents an integrated environment for software development dedicated to robotics.

It is a software package witch includes:

1. A high-level language for general purpose robots off-line programming, fully compatible with the C language;
2. A CAD module for graphical modeling of the robot and its work environment;
3. A library containing robotics related functions;
4. A file manager;
5. A text editor.

At the end of this work, some programs are implemented on an industrial manipulator with six degrees of freedom.

Prólogo

O progresso da exploração de reservas submarinas de petróleo e gás a grandes profundidades leva à necessidade do desenvolvimento de técnicas de automação submarina.

Aproximadamente 67 % da produção brasileira de petróleo é obtida através da exploração de reservas na plataforma continental. Novas reservas em grandes profundidades têm sido localizadas, porém sua exploração é ainda inviável. A medida que a profundidade cresce de 500 m para 2000 m, tecnologias que dispensam a intervenção humana (como a robótica), tornam-se mais e mais atraentes por questões de segurança para o homem, eficiência na realização da tarefa, custo da operação e independência das condições do mar em todas as atividades relacionadas à prospecção.

Levando-se em conta os riscos financeiros e danos ambientais envolvidos em uma possível falha no sistema de prospecção, torna-se evidente a necessidade do desenvolvimento de técnicas automatizadas de reparo e manutenção. Além disso, o custo de uma simples falha a uma grande profundidade, por si só já poderia justificar o desenvolvimento destas técnicas.

Durante um convênio de cooperação técnico-científica envolvendo a Universidade Estadual de Campinas, a PETROBRÁS e o Instituto Tecnológico de Geesthacht - GKSS, da Alemanha, surgiu a oportunidade da realização de um projeto conjunto no campo da automação de intervenções submarinas através da utilização de robôs [4, 25, 29] e telemanipuladores industriais adaptados.

Um telemanipulador Kraft (cedido pela PETROBRÁS) e um robô industrial MANUTEC r3 (cedido pelo GKSS) serviram de ponto de partida para o trabalho realizado no Laboratório de Automação e Robótica da Faculdade de Engenharia Mecânica da UNICAMP.

Com o objetivo de avaliar o desempenho tanto do robô quanto do telemanipulador, algumas alterações foram realizadas em ambos os sistemas originais. Tais alterações foram necessárias pois robôs e telemanipuladores são, por definição, completamente diferentes entre si. O simples fato de que o telemanipulador necessitava de um operador para realizar as tarefas

enquanto o robô podia ser programado através de uma linguagem de programação *off-line* já dava ao último grande vantagem sobre o primeiro. Porém outros aspectos também deviam ser analisados. Entre estes aspectos pode-se citar:

- **Custo.** O custo total da implantação do telemanipulador é inferior ao do robô.
- **Capacidade de Carga.** O telemanipulador possui acionadores hidráulicos enquanto o robô possui acionadores elétricos (motores *brushless*). Esta característica faz com que o manipulador tenha uma capacidade de carga bem superior à do robô.
- **Precisão, Exatidão e Repetibilidade.** Devido às características apresentadas no item anterior, aliadas ao fato de que no robô os sensores de junta são *encoders* e no manipulador, potenciômetros, pode-se concluir que o robô possui uma acurácia muito maior que a do telemanipulador.

Tornou-se evidente que qualquer comparação entre ambos só poderia ser realizada depois que o telemanipulador pudesse dispensar a intervenção de um operador durante o processo de execução da tarefa. Isto equivale a dizer que este deveria poder ser, como o robô, programável através de uma linguagem textual.

Optou-se então pelo desenvolvimento de um *software* (o mais genérico possível) que incorporasse uma linguagem de programação de robôs e pudesse ser aplicado ao caso particular do telemanipulador.

Trata-se de um *software* que integra uma linguagem de programação de robôs com os recursos necessários ao desenvolvimento da programação de tarefas. Programas desenvolvidos neste ambiente serão validados a partir da implementação em um manipulador industrial com seis graus de liberdade.

Dentre os fatores que motivam a escolha do tema, pode-se citar:

- As linguagens de programação de robôs industriais são, em geral, fortemente dependentes do *hardware*. Isto significa que a implementação de uma determinada linguagem não é uma tarefa muito simples levando-se em conta os diferentes tipos de robôs existentes.
- Poucas são as linguagens compiladas que fornecem recursos de simulação e visualização gráfica.
- Em geral, nas linguagens de programação de robôs, a *interface* com o usuário é muito pobre, o que dificulta a elaboração e depuração de programas mais complexos.

No capítulo 1 faz-se uma breve análise dos métodos utilizados para a programação de robôs e mostra-se as vantagens de se utilizar uma linguagem textual de programação *off-line*.

No capítulo 2 apresenta-se o ambiente ULTRA C Integrado (UCI) e detalha-se sua utilização.

O capítulo 3 apresenta a linguagem ULTRA C (Uma Linguagem Textual para Robótica e Automação usando C), sua sintaxe e funções.

O capítulo 4 foi reservado à descrição de uma importante ferramenta embutida no ambiente UCI: o modelador gráfico. Neste capítulo mostra-se sua utilização e potencialidades.

O capítulo 5 descreve as alterações realizadas no sistema Kraft original para que a programação através da linguagem ULTRA C pudesse ser implementada.

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Programação de Tarefas

1.1 Introdução

O sucesso da utilização de robôs em aplicações industriais se deve principalmente ao fato de que eles podem ser utilizados em uma larga gama de tarefas. Em praticamente todas as aplicações industriais existe um ciclo de operações que devem ser realizadas em uma determinada sequência. Ao término de um ciclo, um novo ciclo (não necessariamente igual ao anterior) se inicia. Enquanto o robô avança de um ciclo a outro, a sequência das operações pode variar para permitir que outras tarefas possam ser executadas. Estas alterações na sequência básica são devidas a modificações em certas condições externas relevantes para o sucesso da tarefa.

Toda esta potencialidade só pode ser explorada se o sistema controlador do robô puder ser facilmente programado. O termo programação, em robótica, significa o estabelecimento de um sequência de operações a serem executadas pelo robô.

A versatilidade de um robô está então, intimamente relacionada com a generalidade da sua estrutura física, com sua capacidade de sensoriamento e principalmente com a flexibilidade de programação de sua unidade de controle.

A forma como a programação é efetuada pode variar. Existem basicamente dois métodos de programação de robôs industriais [34]:

- Métodos diretos ou por aprendizagem (programação *on-line*);
- Métodos indiretos ou através de uma linguagem de programação de robôs (programação *off-line*).

Neste capítulo detalha-se ambos os métodos e suas variações. O primeiro método será visto *en passant* enquanto que o segundo terá maior atenção por ser o objeto de estudo desta dissertação.

1.2 Programação por Aprendizagem

O mais antigo método de programação de robôs consiste em se mover (diretamente ou através de algum dispositivo) as juntas do robô enquanto a unidade de controle do robô armazena as informações enviadas pelos sensores de junta para posterior reprodução do movimento.

A programação por aprendizagem é um método simples de ser usado e implementado por não requerer um computador de propósitos gerais.

Este método de programação é o mais utilizado na indústria devido ao fato da tecnologia de montagem em célula flexível ainda não estar suficientemente desenvolvida. Como as aplicações de robôs na indústria ainda estão restritas a tarefas fortemente repetitivas, em ambiente estático que não necessitam de grande precisão, a programação por aprendizagem ainda satisfaz à demanda industrial, na maior parte dos casos.

1.2.1 Formas de Programação por Aprendizagem

Quanto à forma como o operador move as juntas do robô, a programação por aprendizagem pode ser subdividida em três tipos:

1. O operador guia o elemento terminal do robô (*end-effector*), diretamente ou através de algum dispositivo de assistência muscular, enquanto a unidade de controle utiliza os sensores de junta para armazenar as informações referentes aos pontos importantes da trajetória. Como este método requer que o operador seja o responsável pelo movimento da estrutura física do robô, torna-se inviável utilizá-lo em robôs de porte e peso elevados ou com movimentos não reversíveis.
2. O operador guia um simulador físico que possui sensores equivalentes aos do robô e características geométricas idênticas ou em escala reduzida. Devido ao fato de que este método não requer o contato direto do operador com o robô, ele é indicado nas situações em que, por algum motivo, não seja possível o acesso ao ambiente de trabalho do robô ou na programação de robôs de grande porte e de estruturas não reversíveis. Este método tem como principal desvantagem o alto custo do simulador físico [11].

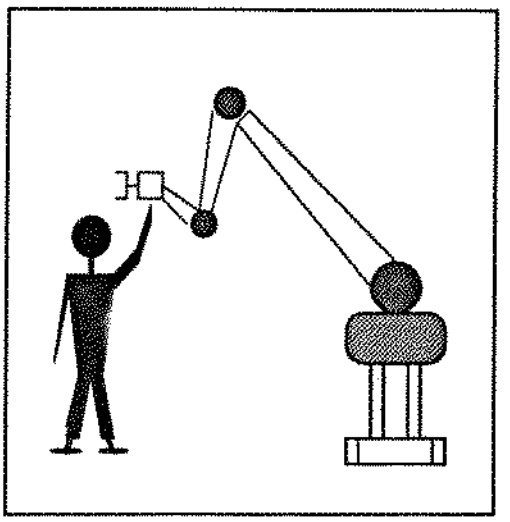


Figura 1.1: *Programação por aprendizagem. O operador guia o robô diretamente.*

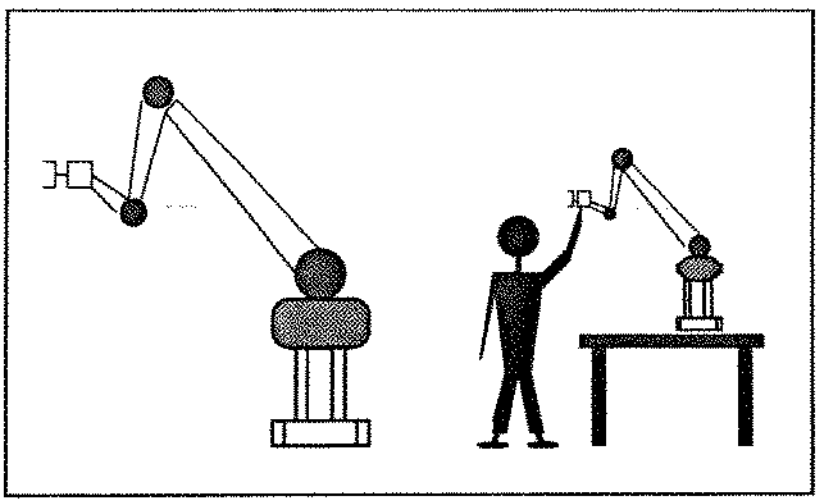


Figura 1.2: *Programação por aprendizagem utilizando um simulador físico.*

3. O operador utiliza um dispositivo conhecido como *teach pendant* ou *teach-in box* para mover cada junta do robô isoladamente ou fornecer a posição e orientação do elemento terminal. O *teach pendant* está, em geral, conectado à unidade de controle e possui teclas especiais que dão acesso a basicamente as mesmas funções do controlador.

   A figura 1.4 mostra um *teach pendant* comercial. O *teach pendant* pode conter indicadores visuais e outras teclas especiais. Alguns são tão sofisticados que permitem a utilização da programação *off-line*. Programas inteiros podem ser escritos, testados e executados através deles.

   Este método pode ser aplicado a qualquer tipo de robô e seu custo é consideravelmente inferior ao exposto anteriormente.

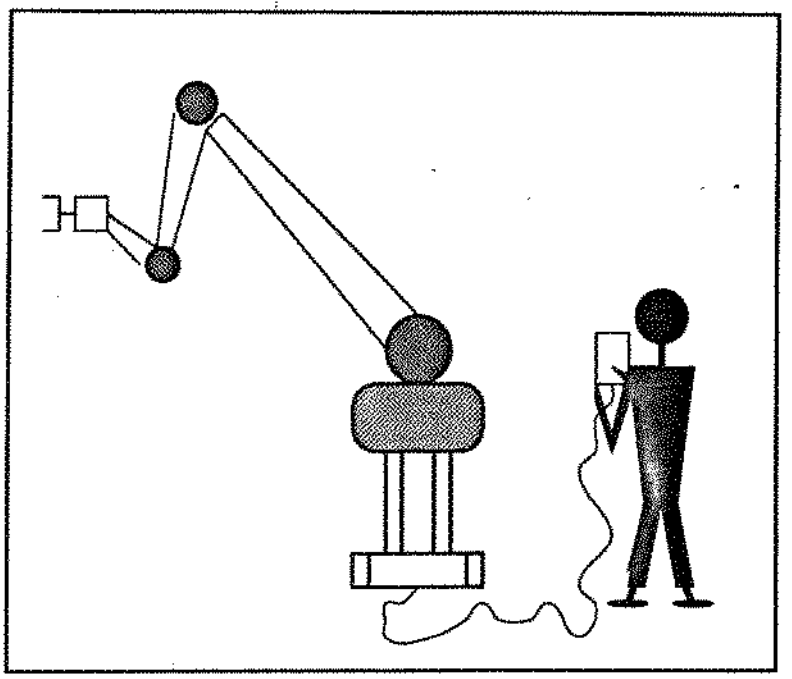


Figura 1.3: *Programação por aprendizagem utilizando o* teach pendant.

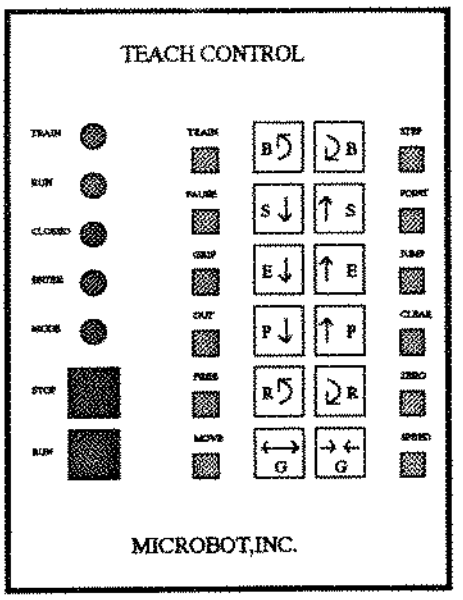


Figura 1.4: Teach pendant *do Sistema Robótico Alpha fabricado pela Microbot, Inc.*

Quanto à forma de armazenamento da tarefa, a programação por aprendizagem pode ser subdividida em dois tipos:

1. **Gravação ponto a ponto:** o operador grava apenas os pontos de interesse. A unidade de controle se encarregará de interpolar os pontos de tal forma que a trajetória final

seja contínua. Nesta forma de gravação, o operador deve estar atento aos obstáculos no interior do volume de trabalho do robô pois, em geral, o caminho que será seguido entre o ponto inicial e o ponto final da trajetória não é uma linha reta. Este método de gravação deve ser aplicado apenas nas tarefas que não requeiram um controle preciso da trajetória e da velocidade entre um ponto e outro. A principal vantagem deste modo de gravação é a pouca memória exigida já que apenas alguns pontos da trajetória precisam ser gravados.

2. **Gravação contínua**: a taxa com a qual os pontos da trajetória são amostrados e gravados é bastante elevada. A velocidade de reprodução do movimento é função das taxas de amostragem e de envio dos pontos pois o robô é conduzido continuamente por inércia. Este método é indicado para aplicações onde a precisão da trajetória e da velocidade são condições necessárias ao sucesso da tarefa. Não tem sentido a utilização do *teach-in box* com o método de gravação contínua. A grande desvantagem deste método está na grande quantidade de memória requerida para que uma tarefa relativamente curta seja programada.

Controladores mais sofisticados permitem aprimoramentos em ambas as formas de armazenamento. Por exemplo, se na gravação ponto a ponto, o operador puder escolher a forma de interpolação (linear, circular, etc) entre um ponto e outro, ou se puder definir a velocidade com a qual o elemento terminal deve passar por cada ponto, então pode-se aplicar este método em tarefas que necessitem de precisão de trajetória e velocidade.

1.2.2 Problemas com a Programação por Aprendizagem

Na programação por aprendizagem, o robô é *forçado* a seguir uma sequência de movimentos pré-determinada. Alterações nesta sequência, implementação de *loops*, tomadas de decisão e verificação *on-line* de dados provenientes de sensores externos são características de difícil, senão impossível, implementação [3, 9].

O tempo gasto durante a programação de uma tarefa relativamente simples pode ser muito grande pois cada movimento do robô deve ser realizado manualmente, tomando-se o cuidado para que não ocorram colisões durante a execução final. Caso algum erro de programação seja cometido tem-se que, em geral, refazer toda a tarefa. A consequência disto é que o robô tem seu tempo útil reduzido. Em aplicações onde a reprogramação de tarefas ocorre com uma frequência alta, a programação por aprendizagem pode ser completamente inviável.

Percebe-se que na programação por aprendizagem, principalmente na gravação contínua, o sucesso da tarefa depende muito da habilidade do operador. Em um ambiente no qual seja impossível o acesso do operador (como em intervenções submarinas ou em reatores nucleares) a situação é ainda mais complicada. Nestes casos, o operador deve se valer de um sistema de visão tridimensional. Ainda assim, a noção de profundidade fornecida por estes sistemas não é suficiente para a realização de tarefas que exijam alto grau de precisão. Na gravação contínua, todo movimento feito pelo operador é armazenado para posterior reprodução. Entretanto, se o operador encontrar dificuldades para realizar alguma operação, pontos indesejáveis estarão sendo armazenados. Estes pontos são completamente inúteis e consomem espaço de memória, tempo durante a execução e sua eliminação é quase impossível.

Nem todos estes problemas estão necessariamente presentes em um sistema de programação por aprendizagem. Alguns podem ser eliminados através de um aumento na solisticação do controlador do sistema. Outros, entretanto, são inerentes a este método de programação.

1.2.3 Tendências Atuais

Os fabricantes de robôs industriais têm mostrado uma crescente preocupação com a qualidade da *interface* homem-máquina. A introdução do *teach pendant* com controle através de *joystick* e o constante esforço em se obter uma estrutura perfeitamente balanceada para facilitar a movimentação do robô são exemplos claros desta preocupação. Do ponto de vista operacional, os fabricantes estão cada vez mais aceitando o conceito de movimento centrado na ferramenta e a utilização de um sistema de coordenadas múltiplos, que ajudarão a reduzir o tempo de programação e aumentarão a produtividade da célula de trabalho.

A sofisticação dos pacotes de funções fornecidos pelos fabricantes tende a aumentar à medida que os conceitos e algoritmos se tornam de conhecimento público. Isto se deve ao fato de que o limite mínimo de facilidades fornecidas por um pacote comercialmente aceitável aumenta a cada dia. Por outro lado, os fabricantes estão conscientes das vantagens advindas do uso da programação *off-line* através de linguagens textuais e aumentam as facilidades de programação por aprendizagem dos seus robôs pelo simples fato de ainda não possuírem tecnologia suficiente para introduzir a programação *off-line* a custos razoáveis.

O reconhecimento da linguagem falada [5, 6] em substituição à movimentação física do robô ou à utilização do *teach pendant* está sendo pesquisado. Entretanto, os resultados provenientes destas pesquisas ainda não podem ser considerados satisfatórios para aplicações industriais.

Embora o futuro seja mais promissor para a programação *off-line*, a programação por aprendizagem ainda tem muito o que avançar e deve ainda ser por mais alguns anos o método de programação de tarefas mais aplicado na indústria mundial.

A programação por aprendizagem é adequada para algumas aplicações como soldagem ponto a ponto, pintura, e transferência de peças. Em outras aplicações, entretanto, como em montagem mecânica e inspeção de peças, existe a necessidade de se especificar a ação desejada do robô em resposta a certos dados fornecidos por sensores, entrada de dados feita pelo operador ou detecção iminente de colisão. Nesses casos a programação de robôs necessita das facilidades fornecidas por uma linguagem de programação de computadores de propósitos gerais.

A programação *off-line* por linguagem textual, ao contrário da programação por aprendizagem permite a utilização de pontos determinados analiticamente, interação com sistemas de CAD/CAM e acesso a dados fornecidos por sensores externos. Além disso, não necessita da utilização do robô durante o processo de aprendizagem (ou alteração da programação), o que aumenta seu tempo útil.

1.3 Programação Textual

A programação *off-line* usando linguagens textuais pode ser considerada como o processo pelo qual os programas são desenvolvidos, parcial ou totalmente, sem a necessidade do uso do robô propriamente dito, através da utilização de uma linguagem de programação de computador. Recentes avanços na tecnologia de desenvolvimento de *hardware* e *software* estão tornando a programação de robôs por linguagens textuais, cada vez mais próximas da realidade industrial. Estes avanços incluem uma maior sofisticação dos controladores, melhor precisão de posicionamento e incremento no número e tipo dos sensores. Existe atualmente uma considerável atividade na área de pesquisa e desenvolvimento de novas técnicas de programação *off-line* de robôs, e espera-se que estas técnicas estejam sendo empregadas intensivamente na indústria dentro de poucos anos.

Em aplicações relacionadas com a produção em massa de determinado produto, como a soldagem numa linha de produção automobilística, a reprogramação de tarefas é mínima. Nestes casos, a programação por aprendizagem pode ser aplicada sem grandes problemas. Entretanto, para que robôs possam ser utilizados na produção de pequenos e médios lotes, onde o tempo de programação pode ser de substancial importância, o método de programação *off-line* usando linguagem textual é essencial. A crescente complexidade das aplicações usando robôs, particularmente as relacionadas à montagem, fazem as vantagens associadas à programação *off-line* por

linguagem textual cada vez mais atraentes. Estas vantagens podem ser resumidas em:

1. Redução do *downtime* (tempo em que o robô está fora da linha de produção), pois o robô pode continuar trabalhando enquanto sua próxima tarefa está sendo programada. Esta característica torna a utilização da flexibilidade do robô mais efetiva.
2. Não há necessidade do programador entrar em contato com o ambiente de trabalho do robô, que muitas vezes é hostil.
3. Portabilidade de programação. A programação de tarefas depende muito menos das características específicas de uma unidade de controle. Em um futuro bem próximo, quando os problemas relacionados à compatibilidade entre os diversos controladores estiverem resolvidos, uma aplicação poderá ser facilmente transportada para robôs de outro tipo sem haver a necessidade de alteração do programa.
4. Integração com sistemas CAD/CAM. Esta característica permite uma maior integração entre as fases de projeto e produção, reduzindo tempo, custos e aumentando a confiabilidade do processo produtivo.
5. Simplificação da programação de tarefas com alto grau de complexidade. As linguagens de programação de alto nível facilitam a elaboração de tarefas complexas pois possuem formas analíticas de geração de trajetórias e análise de dados provenientes de sensores externos além de várias estruturas de controle, *loops*, tomadas de decisão, etc.;
6. Segurança na geração dos programas. As linguagens de programação de robôs podem internamente (ou através da comunicação com sistemas CAD/CAM) gerar um modelo geométrico do ambiente de trabalho do robô e, através de algoritmos de detecção de colisão, produzir trajetórias seguras e simulá-las por *software* antes da execução final.

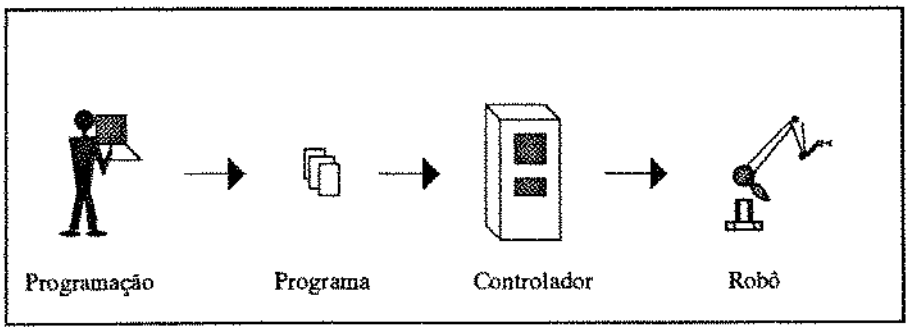


Figura 1.5: *Sistema de programação* off-line

1.3.1 Desenvolvimento da Programação Off-line

A técnica de programação *off-line* tem sido empregada no campo das máquinas de comando numérico (CNC) já há algum tempo, e muito conhecimento foi adquirido nesta área. Embora existam diferenças fundamentais entre a programação de máquinas de comando numérico e robôs industriais, algumas semelhanças entre os problemas e fases de desenvolvimento de sistemas *off-line* podem ser observadas. Torna-se inevitável traçar um paralelo entre o desenvolvimento destas máquinas e o desenvolvimento de técnicas de programação *off-line* de robôs industriais.

Os primeiros CNC's eram programados através de códigos simples que definiam movimentos da ferramenta, velocidades e coordenadas espaciais. Geralmente estes comandos eram específicos de determinados controladores.

A grande maioria dos robôs industriais de hoje tem controladores com características semelhantes às dos primeiros CNC's: são programados usando-se funções específicas do robô em questão.

Com a evolução das máquinas de comando numérico no cenário industrial, linguagens de programação de alto nível começaram a ser implementadas com o objetivo de fornecer maiores facilidades aos programadores. A programação passou a ser completamente *off-line* sendo, em geral, realizada longe do chão de fábrica. Com o desenvolvimento de linguagens textuais de comando numérico como APT e EXAPT, ferramentas e objetos passaram a ser descritos geometricamente usando-se pontos, curvas e superfícies. Interfaces apropriadas eram necessárias para a conversão dos códigos de um controlador específico.

Este é o estágio correspondente em que se encontra o desenvolvimento da programação *off-line* de robôs industriais. Linguagens textuais como RAPT e ROBEX incorporaram características de programação de alto nível similares às do APT. Sistemas gráficos como CATIA e GRASP usam técnicas de interação com CAD.

Assim como a programação de máquinas de comando numérico evoluiu em direção à integração com sistemas de CAD, a programação *off-line* de robôs industriais se desenvolverá de forma similar. A tendência atual indica que o próximo estágio será o desenvolvimento de sistemas multirobôs.

1.3.2 Evolução das Linguagens de Programação de Robôs

As linguagens de alto nível especializadas para a programação de tarefas de robôs industriais começaram a aparecer nos EUA em 1973. WAVE foi a primeira linguagem de programação de robôs. Foi desenvolvida no Laboratório de Inteligência Artificial de Stanford para programar um robô interligado a um sistema de visão.

AL (*Assembly Language*) foi a segunda geração das linguagens de programação de robôs desenvolvida em Stanford. Surgiu em 1974 com o objetivo de descrever o movimento de múltiplos braços em tarefas que exigiam coordenação. AL foi baseada na linguagem ALGOL (*Algorithmic Language*) e desenvolvida em Pascal e SAIL (*Stanford Artificial Intelligence Language*). Apesar de ser uma das primeiras linguagens de programação de robôs, AL ainda é considerada uma das melhores linguagens devido ao seu alto grau de estruturação e recursos implementados. Esta linguagem tem servido de modelo para muitos grupos de pesquisa ligados ao problema da programação de robôs em todo o mundo. Ver tabela 1.2.

A primeira linguagem textual comercial desenvolvida foi VAL (*Versatile Assembly Language*), apresentada em 1979 pela UNIMATION, para os robôs da linha PUMA (*Programmable Universal Machine for Assembly*). VAL tem a mesma estrutura do BASIC, com muitos comandos adicionados para possibilitar a programação de robôs. Foi implementada em linguagem C e *assembly* 6503. Uma segunda versão , chamada VAL II foi lançada em 1984 como uma extensão das capacidades de sua antecessora.

A IBM, no início dos anos setenta, desenvolveu suas primeiras linguagens de programação de robôs: EMILY e ML. A partir de 1976 a IBM iniciou o desenvolvimento de duas linguagens de programação de robôs: AUTOPASS (*Automatic Parts Assembly System*) e AML (*A Manufacturing Language*). Desde 1982, AML está sendo comercializada com os robôs fabricados pela IBM.

Desde então, várias linguagens de programação de robôs foram sendo desenvolvidas.

RAIL (*Robotic Automatix Incorporated Language*), desenvolvida pela Automatix, é uma linguagem de alto nível adequada tanto à visão quanto à manipulação. Como a maioria das linguagens de programação de robôs existentes, RAIL é interpretada. Foi desenvolvida em Pascal.

A linguagem RPL (*Robotic Programming Language*) foi criada pelo SRI (*Stanford Research Institute*) para facilitar o desenvolvimento, teste e *debbuging* de algoritmos de controle dedicados a pequenos sistemas de manufatura que consistam de poucos manipuladores, sensores

e equipamentos auxiliares.

Hoje, pode-se dizer que existem tantas linguagens de programação quanto fabricantes de robôs. Rembold relacionou em 1987 mais de 100 linguagens de programação para robôs [27]. Dentre estas linguagens pode-se citar: AML/E, ANIMATE/PLACE, ANIMATOR, ANORAD, FUNKY, GEOMAP, HELP, JARS, LAMA, LM, MAL, MAPLE, MCL, PRBE, RAPT, RCL, ROL, ROPL, PAL, SIGLA, SRL, e T3.

A tabela 1.1 mostra algumas linguagens de programação de robôs, os equipamentos para os quais foram desenvolvidas e quem as desenvolveu.

Linguagem	*Robô*	*Fabricante*
FUNKY	IBM	IBM
T3	T3	Cincinnati Milacron
ANORAD	Anorad	Anorad Corp.
EMILY	IBM	IBM
RCL	PACS	Rensselaer Polytechnic Institute
RPL	PUMA	SRI International
SIGLA	Sigma	Olivetti
VAL	PUMA	Unimation
AL	Stanford	Stanford Artificial Intelligence Lab.
HELP	Allegro	General Electric
MAPLE	IBM	IBM
MCL	------	McDonnell Douglas Corp.
PAL	Stanford	Purdue University
AUTOPASS	IBM	IBM

Tabela 1.1: *Linguagens de programação, equipamentos e respectivos fabricantes*

O grande problema relacionado às linguagens de programação de robôs industriais reside no fato de que a maioria destas linguagens não possui algumas características essenciais, como por exemplo:

- Estruturação;
- Tipologia de dados;
- Concorrência ou paralelismo;

- Interação com sensores externos complexos;
- Boa *interface* com o usuário;
- Portabilidade.

Somente o desenvolvimento em conjunto destas características, determinará o sucesso de uma nova linguagem de programação de robôs.

1.3.3 Níveis de Programação

Pode-se usar como indicador do grau de sofisticação de uma linguagem de programação de robôs, o nível de abstração em relação à tarefa programada. A literatura existente classifica as linguagens de programação de robôs em quatro níveis:

1. **Nível de Junta.** As linguagens classificadas neste nível requerem a programação individual de cada junta do robô para que uma determinada posição no espaço tridimensional seja alcançada.
2. **Nível de Manipulador.** Neste nível de programação o usuário se preocupa apenas em fornecer a posição e orientação do *end-effector* e o sistema se encarrega de obter, através do modelo geométrico inverso do robô, as posições de cada junta.
3. **Nível de Objeto.** Linguagens classificadas neste nível requerem apenas especificações relativas ao posicionamento de objetos no interior do volume de trabalho do robô. Torna-se necessário então, a existência de um modelo matemático representativo do ambiente de trabalho no qual o robô se encontra inserido.
4. **Nível de Objetivo.** Neste nível as tarefas são especificadas de forma genérica, como por exemplo, "Monte parte X da peça Y". Esta característica implica não somente na existência do modelo do ambiente mas também em um conjunto de dados relativos a uma determinada tarefa. Linguagens neste nível de abstração requerem um certo grau de "inteligência" para que seja possível a interpretação dos comandos e a geração de trajetórias livres de colisão.

A maioria dos sistemas desenvolvidos, tanto *on-line* quanto *off-line*, estão classificados no nível de Manipulador. Linguagens atualmente em desenvolvimento estão sendo direcionadas para o nível de objeto. As linguagens no nível objetivo ainda são consideradas um campo de pesquisa relativamente novo e seus resultados ainda pouco satisfatórios.

1.3.4 Requisitos Básicos de um Sistema Off-Line

Como mostrado na seção anterior, sistemas *off-line* podem ser classificados em diferentes níveis de programação. Sistemas diferentes, implicam em diferentes métodos de programação. Entretanto, todos os sistemas de programação *off-line* possuem certas características comuns, sem as quais se tornariam completamente inviáveis.

Sistemas de programação *off-line* bem sucedidos devem possuir algumas características básicas, entre elas:

1. Conhecimento do processo ou tarefa a ser programada.
2. Modelo matemático tridimensional do ambiente de trabalho do robô.
3. Conhecimento da geometria, cinemática e, eventualmente, do modelo dinâmico do robô.
4. Método de programação gráfica ou textual.
5. Verificação e validação das tarefas programadas. Por exemplo, checagem das restrições de junta (fins de curso, velocidade máxima, etc.) ou detecção de colisão.
6. *Interface* apropriada para a comunicação com o controlador do robô.
7. *Interface* homem-máquina amigável.

1.3.5 Problemas Relacionados à Programação Off-Line

A programação *off-line* de robôs apresenta três grandes problemas:

1. **Modelagem.** Requer a existência de um modelo teórico do ambiente no qual o robô está inserido. Os programas devem levar em consideração as diferenças existentes entre o modelo teórico e o mundo real.
2. **Interfaces.** Existe a necessidade de uma padronização de *interfaces*, sem a qual se torna quase impossível a compatibilidade entre robôs e sistemas de programação. Em geral, uma determinada linguagem se aplica exclusivamente a um robô (ou tipo) pois é fortemente dependente do *hardware*.
3. **Forma de Programação.** Dificuldade em gerar uma forma genérica de programação para os diversos tipos de robôs existentes e as inúmeras aplicações possíveis.

A seguir, discute-se com mais detalhes estes problemas e como eles afetam a criação de um sistema de programação *off-line* genérico.

Modelagem

Geralmente, as linguagens de programação de robôs possuem primitivas geométricas que permitem uma modelagem aproximada do robô e seu ambiente de trabalho.

O problema reside no fato de que estas primitivas nem sempre atendem às características reais e torna-se necessária a composição de primitivas para que seja possível uma representação aceitável do ambiente.

A composição de primitivas é um trabalho manual desgastante, demorado e passível de erros.

Uma solução para este problema é o interfaceamento com *softwares* de CAD. Para isto, torna-se necessária a criação de rotinas responsáveis por este interfaceamento.

Em um ambiente estático, a construção destas rotinas não é uma tarefa difícil. Porém, em geral, o ambiente de trabalho do robô é dinâmico e o modelo geométrico teórico deve corresponder sempre à realidade. Neste caso, o interfaceamento com *softwares* de CAD não é suficiente para resolver o problema.

Além da modelagem do ambiente de trabalho em si, existe a necessidade de se modelar o próprio robô. Toda linguagem de programação off-line de robôs possui um modelador capaz de representar mecanismos com juntas. Este modelador pode ser implementado de três formas diferentes:

1. Em um nível primário, o modelador é dedicado a um só tipo de robô. Embora esta abordagem simplifique a implementação, por outro lado, limita o escopo de aplicação da linguagem.
2. Em um nível mais elevado, encontram-se os modeladores capazes de representar uma determinada classe de estruturas. A separação dos robôs industriais em classes de estrutura semelhante, permite encontrar o melhor algoritmo de modelamento para cada classe. Mesmo neste tipo de abordagem, a obtenção do modelo geométrico inverso não é simples pois não existe uma solução que cubra todas as possibilidades de arranjos estruturais.
3. O mais alto nível de sofisticação de um modelador de robôs é o que permite a representação de juntas interconectadas que se movimentam como um grupo e não podem ser consideradas matematicamente independentes. A complexidade matemática presente nestes mecanismos impede qualquer tentativa de generalização do modelador. Em geral, estuda-se e implementa-se em apenas alguns poucos casos particulares.

Os sistemas de programação *off-line* de robôs estão, em sua maioria, no nível 1 de modelamento. Alguns poucos sistemas, no nível 2.

A incorporação de modelagem dinâmica ainda está limitada aos casos particulares e é considerada muito complexa para uma abordagem genérica.

Interfaces

Problemas relacionados ao interfaceamento entre o código gerado pela linguagem de programação e o código aceito pelos diversos controladores de robôs existentes talvez sejam os maiores obstáculos à unificação e padronização dos sistemas de programação *off-line*.

Cada fabricante produz seu próprio controlador que, via de regra, não é compatível com os controladores produzidos por outros fabricantes.

Algumas padronizações são sugeridas para que estes problemas sejam contornados:

1. **Sistema de Programação:** Alguns países europeus (Grã-Bretanha, França e Alemanha) e o Japão estudam as implicações de uma padronização nos sistemas de programação *off-line*. Os primeiros relatórios produzidos pelo grupo europeu indicam concordância em algumas idéias básicas acerca deste assunto.
2. **Sistema de Controle:** O incremento da utilização de robôs na indústria torna evidente a necessidade de uma padronização dos sistemas controladores. A incompatibilidade existente, tanto a nível de *hardware* quanto de *software*, torna impossível qualquer tentativa de unificação e padronização dos sistemas de programação *off-line*.
3. **Formato do Programa:** Cada linguagem de programação existente possui sua própria estrutura interna, comandos, forma de armazenamento de dados, etc. Padronizar o formato dos programas significa reduzir os esforços necessários à generalização das linguagens de programação.

Forma de Programação

Além da existência de um modelador eficiente, um sistema de programação *off-line* necessita de uma linguagem de programação adequada para manipular as informações provenientes deste modelador. Funções relacionadas aos movimentos do robô, tratamento de entradas e saídas provenientes de dispositivos periféricos, comandos de lógica, controle de sequência e sub-rotinas, compõem o escopo de uma linguagem de programação.

As funções relacionadas ao movimento do robô são de fundamental importância numa linguagem de programação. A generalização destas funções não é simples pois cada aplicação requer funções de movimentação diferentes.

A implementação das funções pode ser realizada, como exposto anteriormente, em quatro níveis: nível de junta, nível de manipulador, nível de objeto e nível de objetivo. A maioria das linguagens desenvolvidas estão classificadas no nível de manipulador. Estas linguagens deixam a cargo do programador a tarefa de gerar trajetórias livres de colisão. Funções gráficas que permitam a visualização do ambiente através de simulações facilitam este trabalho pois o programador pode verificar passo a passo a execução da tarefa, sem a necessidade de executá-la no sistema real.

Linguagens que permitam a programação de sistemas multirobô são de difícil implementação. À medida que a indústria tende aos sistemas flexíveis de manufatura, esta característica será cada vez mais necessária.

A questão da modularidade e estruturação da linguagem é de fundamental importância pois permite simplificações na programação de tarefas com alto grau de complexidade.

Neste capítulo procurou-se mostrar os diferentes métodos de programação de robôs industriais, suas características básicas, vantagens e desvantagens. Fez-se um breve histórico dos sistemas de programação *off-line* existentes. Maiores detalhes sobre cada uma das linguagens de programação de robôs não caberiam no escopo deste capítulo. Apresenta-se porém, a tabela 1.2 que resume algumas das principais características das linguagens mais utilizadas. Nesta tabela aparece a linguagem ULTRA C que será detalhada posteriormente. O item *módulos de suporte* que aparece na tabela 1.2 representa o grau de conforto oferecido ao usuário. Este item será o objeto de estudo do próximo capítulo onde será apresentado o ambiente integrado UCI do qual a linguagem ULTRA C é parte integrante.

Características	AL	AML	HELP	JARS	MCL	RAIL	RPL	VAL	Ultra C
Textual	■	■	■	■	■	■	■	■	■
Via Menú		■							
Subrotinas				■			■		■
Extensão					■				
Nova	■	■	■			■		■	
Frame	■			■	■	■		■	■
Espaço de Junta		■		■		■			■
Vetor	■	■		■	■				■
Rotação	■			■	■				■
Trajetória						■			■
Matriz	■			■					■
Ång. de Euler	■	■		■		■		■	■
Multi-robô	■		■		■				
Labels		■	■	■	■		■	■	■
If-Then	■	■	■	■	■	■	■	■	■
If-Then-Else	■	■	■	■	■	■	■		■
While-Do	■	■	■	■	■	■	■		■
Do-Until	■	■		■		■	■		■
Case	■			■		■	■		■
For	■	■		■		■	■		■
Begin-End	■	■		■		■			■
Proc/Func/Subr	■	■	■	■	■	■	■	■	■
Editor	■	■	■	■		■	■	■	■
Ger. de Arquivos	■	■	■	■		■	■	■	■
Interpretador	■	■	■			■	■	■	
Compilador	■			■	■		■		■
Simul. Gráfica	■	■			■				■
Macros	■		■		■				■
Bibliotecas	■	■							■
Ajuda	■								■

- MODALIDADE DA LINGUAGEM
- TIPO DE LINGUAGEM
- TIPOS DE DADOS GEOMÉTRICOS
- REPRESENTAÇÃO E ESPECIFICAÇÃO DAS ROTAÇÕES
- CAPACIDADE DE CONTROLAR VÁRIOS ROBÔS
- ESTRUTURAS DE CONTROLE
- MÓDULOS DE SUPORTE

Tabela 1.2: *Características de algumas linguagens de programação de robôs.*

Capítulo 2

O Ambiente ULTRA C Integrado

2.1 Introdução

Um ambiente integrado de desenvolvimento de *software* é um pacote fechado onde o programador encontra todos os recursos necessários para desenvolver seus programas. Neste pacote, em geral, encontra-se um editor de textos simples, um gerenciador de arquivos, um compilador e um montador. Outros aplicativos podem estar inclusos, dependendo do objetivo do ambiente e de seu nível de sofisticação.

O Turbo C e o Turbo Pascal da Borland International,Inc. são exemplos de ambientes integrados de desenvolvimento de *software* de grande sucesso. O segredo deste sucesso não foram somente preço e eficiência, mas sim o fato de que um novo ambiente integrado de desenvolvimento havia sido criado. A entrada e edição do código-fonte, a verificação da sintaxe, a compilação, *linkedição*, execução e depuração foram etapas trazidas para dentro de um único pacote e tornadas disponíveis através de menus facilmente utilizáveis. Estas atividades vinham, até então, de pacotes independentes de softwares especializados.

A grande vantagem em se utilizar um ambiente integrado é que todos os aplicativos incluídos possuem uma grande compatibilidade na troca de dados. Isto acarreta uma maior facilidade e rapidez no desenvolvimento dos programas.

Neste capítulo mostra-se a criação e implementação de um ambiente integrado de desenvolvimento de *software* aplicado à robótica.

2.2 Componentes do Ambiente UCI

Todo ambiente de desenvolvimento de *software* está baseado em uma linguagem de programação. A sintaxe desta linguagem, os recursos presentes e necessários afetam bastante a forma como o ambiente será estruturado.

O ambiente UCI está baseado na linguagem ULTRA C (Uma Linguagem Textual para Robótica e Automação usando C) que será detalhada no próximo capítulo. ULTRA C é uma extensão da biblioteca padrão do ANSI C [16, 30, 31] contendo funções especialmente criadas para possibilitar o desenvolvimento de programas voltados para a automação, em especial, a robótica.

O UCI tem uma estrutura similar à do Turbo C. É composto basicamente de:

- Um **editor de textos**;
- Um **compilador C**;
- Um **montador**;
- Um **gerenciador de arquivos**;
- Um **conjunto de bibliotecas de funções**;
- Um **modelador gráfico**.

Tanto o compilador quanto o montador são os idênticos aos utilizados pelo Turbo C, por isso, existe uma compatibilidade total entre os dois ambientes. Isto significa que qualquer código-fonte escrito no Turbo C pode ser compilado no UCI e vice-versa, desde que a biblioteca relacionada à robótica (*UC.LIB*) seja transferida para o Turbo C.

Todos os recursos presentes no UCI são acessados através de *pop-up menus* com memória, o que torna sua utilização mais fácil e eficiente.

2.3 Utilização do UCI

O programa UCI.EXE é o responsável pelo gerenciamento de todos os recursos do sistema. A execução deste programa fornece a tela mostrada na figura 2.1.

A tela de trabalho do UCI é dividida em quatro partes:

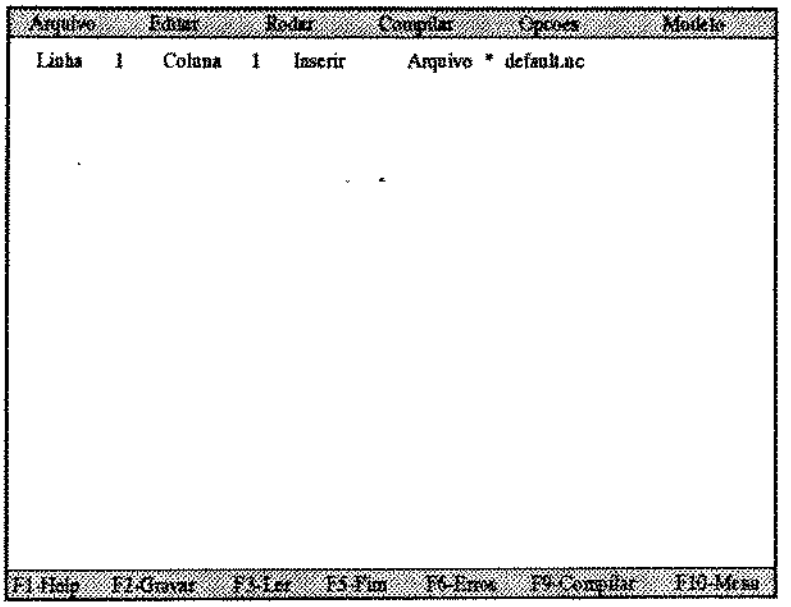


Figura 2.1: *Tela inicial do ambiente integrado UCI*

- **Menu Principal.** É a primeira linha da tela. Nesta linha encontram-se as portas para a navegação do sistema. Cada uma das palavras-chaves contidas no menu principal dá acesso a um determinado recurso do sistema;
- **Linha de Status.** É a segunda linha da tela. Na linha de *status* estão contidos alguns dados de interesse para o programador.
- **Área de Trabalho.** São todas as linhas abaixo da linha de *status* exceto a última linha da tela. Na área de trabalho o programador escreve seu programa-fonte.
- **Linha de Hot-Keys.** É a ultima linha da tela. Nela encontram-se algumas teclas especiais que dão acesso diretamente a algumas facilidades do sistema, dispensando-se o uso das janelas de navegação.

Quando o usuário entra no ambiente UCI, o cursor aparece na primeira linha e primeira coluna da área de trabalho. Quando o cursor estiver na área de trabalho, significa que o usuário está utilizando o editor de textos do UCI.

Os comandos de edição são basicamente os mesmos usados no editor do Turbo C [16].

Para facilitar a utilização do sistema de janelas, algumas regras de navegação foram adotadas:

1. O pressionamento da tecla **F10** faz a mudança entre o menu principal e a área de trabalho.

2. Em qualquer menu do sistema o usuário pode utilizar as setas do teclado para mudar a opção ativa.

3. A escolha de uma opção é realizada em dois passos:
   a) primeiro deve-se tornar a opção ativa através das setas do teclado.
   b) efetiva-se a escolha pressionando-se a tecla **ENTER**.

4. Pressionar a primeira letra de uma opção equivale a realizar os dois passos mencionados no item anterior.

5. Estando o cursor na área de trabalho, o pressionamento da tecla **ALT** em conjunto com a primeira letra de uma opção do menu principal, equivale a pressionar **F10** (ou seja, ir para o menu principal) e realizar as operações descritas nos itens 2, 3 ou 4.

6. Em qualquer menu ou sub-menu do sistema, o pressionamento da tecla **ESC** apresenta o menu anterior. Caso o usuário esteja no menu principal, o cursor será devolvido à área de trabalho.

7. Todos os menus e sub-menus do sistema possuem memória, ou seja, guardam a última opção do usuário.

8. Quando o usuário escolher uma opção pela primeira vez em uma sessão de trabalho, o primeiro item do sub-menu (caso exista) estará ativo.

2.4 Navegação Através do Sistema de Janelas

A partir do menu principal, o usuário pode ter acesso à edição de textos, arquivos, compilação, *linkedição*, customização do ambiente, e modelagem geométrica do ambiente de trabalho do robô.

Para uma melhor utilização do UCI torna-se indipensável o conhecimento de seu sistema de janelas. No decorrer desta seção, mostra-se todo o sistema de menus e sub-menus disponível para o usuário.

2.4.1 A Opção *Arquivo*

A opção *Arquivo* é a primeira opção do menu principal. Algumas funções do gerenciador de arquivos se encontram disponíveis dentro desta opção.

A escolha da opção *Arquivo* apresenta um sub-menu vertical com seis itens como mostra a figura 2.2:

1. *Ler*;
2. *Novo*;
3. *Gravar*;
4. *Salvar como...*;
5. *Mudar Diretória*;
6. *Fim*.

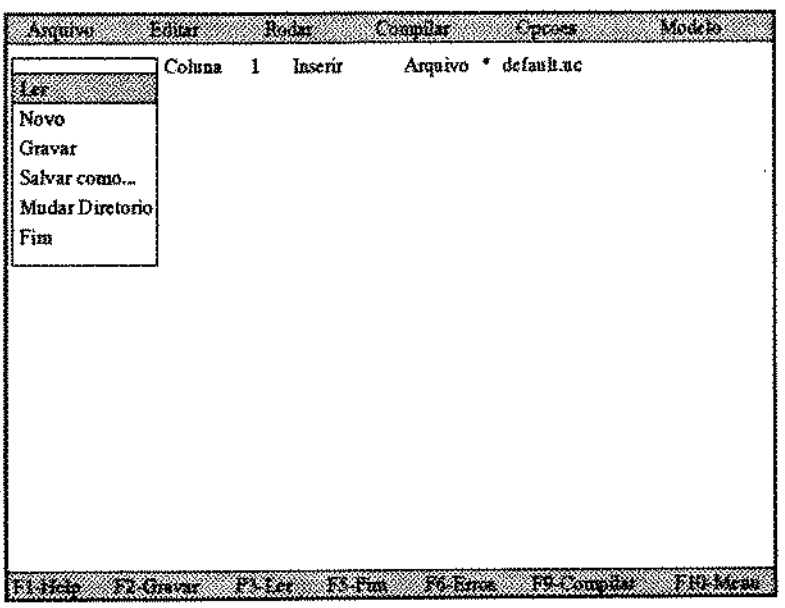


Figura 2.2: *O sub-menu da opção* Arquivo

Quando o usuário escolher a opção *Arquivo* pela primeira vez em uma sessão de trabalho, o item *Ler* estará ativo como mostrado na figura.

A Opção *Ler*

A opção *Ler* permite ao usuário transferir um programa-fonte escrito em ULTRA C de uma unidade de disco rígido ou flexível para a área de trabalho.

Ao escolher a opção *Ler*, o sistema exibirá a tela mostrada na figura 2.3.

Abaixo da opção *Ler* o sistema pede um *path* para o arquivo que o usuário deseja ler. Neste momento o usuário podem entrar com qualquer *path* válido no DOS desde que não contenha os caracteres-curinga '*' e '?'.

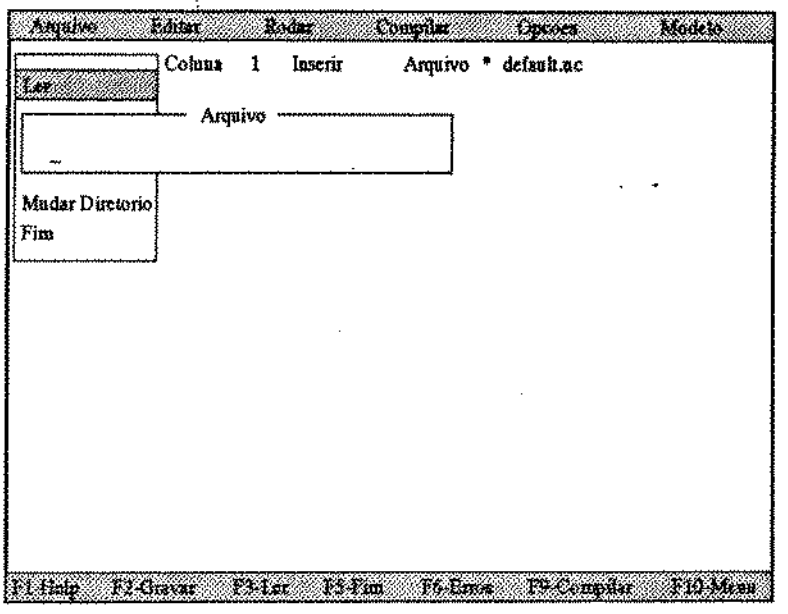


Figura 2.3: *A opção* Ler

Se no *path* fornecido pelo usuário não contiver informações relativas ao *drive* ou ao diretório em que o arquivo se encontra, o UCI irá procurar o arquivo no diretório *default* (o diretório corrente).

Caso o usuário especifique um nome de arquivo válido, este arquivo será lido e seu conteúdo aparecerá na área de trabalho, pronto para ser editado. Caso a área de trabalho esteja ocupada por um outro arquivo, o UCI perguntará ao usuário se deseja gravar as alterações realizadas antes da leitura do novo arquivo.

A *hot-key* **F3** aciona diretamente a janela correspondente à opção *Ler*.

É importante lembrar que a qualquer momento o usuário pode abandonar a opção *Ler* simplesmente pressionando a tecla **ESC**.

A Opção *Novo*

Esta opção não possui sub-menu. Ela limpa a área de trabalho e atribui o nome "default.uc" ao arquivo corrente. Se o arquivo contido na área de trabalho tiver sofrido alguma alteração, o UCI perguntará ao usuário se ele deseja que as alterações sejam gravadas antes que a área de trabalho seja apagada.

A Opção *Gravar*

Assim como a opção *Novo*, a opção *Gravar* também não possui sub-menu. Ela realiza a operação inversa à realizada pela opção *Ler*, ou seja, transfere um arquivo da área de trabalho para uma unidade de disco.

Caso o nome do arquivo corrente seja "default.uc", o UCI pedirá um novo nome.Como na opção *Ler*, o usuário pode especificar o *path* completo do arquivo ou simplesmente seu nome. Caso apenas o nome seja fornecido, o UCI usará o diretório *default*.

A *hot-key* **F2** aciona diretamente a opção *Gravar*.

A Opção *Salvar como ...*

Neste opção, o usuário pode salvar o arquivo corrente com um outro nome. Este recurso é útil na hora de criar versões de teste para os programas.

Quando o usuário escolher esta opção, o UCI apresentará a tela mostrada na figura 2.4.

Da mesma forma que nas opções *Ler* e *Gravar*, o usuário poderá fornecer o *path* completo ou apenas o nome do arquivo. O nome que o usuário fornecer passará a ser o nome do arquivo corrente.

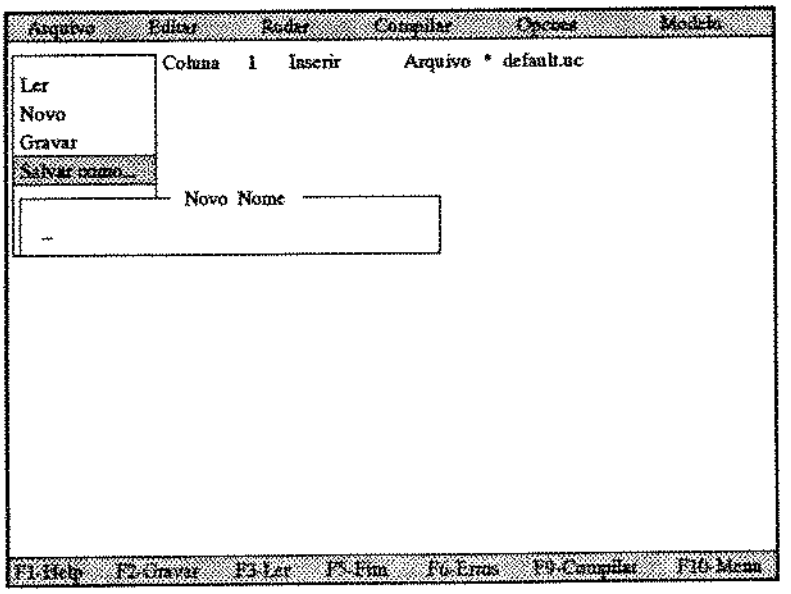


Figura 2.4: *A opção* Salvar como...

A Opção *Fim*

A escolha desta opção, encerra uma sessão de trabalho do UCI. O programa termina sua execução e o controle é devolvido ao sistema operacional.

Caso alguma alteração tenha sido feita no arquivo corrente, o UCI perguntará ao usuário se deseja gravá-lo antes de terminar a sessão.

A *hot-key* **F5** aciona diretamente a opção *Fim*.

2.4.2 A Opção *Editar*

É a segunda opção do menu principal. Não possui sub-menu. Sua ativação transfere o cursor para a área de trabalho, permitindo ao usuário a utilização do editor de textos do UCI.

2.4.3 A Opção *Rodar*

É a terceira opção do menu principal. Assim como a opção *Editar*, também não possui sub-menu. Sua função é permitir que o usuário execute o programa corrente. Caso necessário, o programa será compilado, montado e, posteriormente executado. Se o código executável (.EXE) for encontrado no diretório *output* (veja seção *A opção Opções*) e o programa corrente não tiver sofrido alterações, a compilação não será necessária.

Caso algum erro de compilação seja detectado, o UCI informará ao usuário de sua ocorrência.

Maiores informações sobre a compilação e a interpretação das mensagens de erro geradas pelo UCI serão fornecidas na próxima seção.

2.4.4 A Opção *Compilar*

A quarta opção do menu principal permite acessar diretamente o compilador C contido no UCI. Deve-se utilizar esta opção quando se desejar realizar a compilação e/ou montagem de um programa-fonte sem que este seja executado.

Quando a opção *Compilar* está ativa, o UCI apresenta a tela mostrada na figura 2.5.

Cabem aqui algumas explicações sobre a compilação e a *linkedição*.

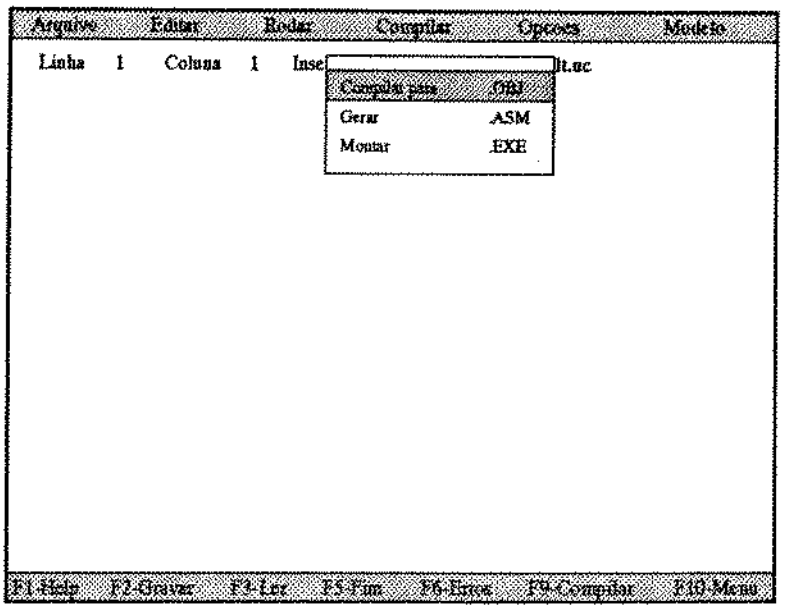


Figura 2.5: *O sub-menu da opção* Compilar

O código-fonte (extensão .UC) é uma sequência de comandos que descrevem as ações que o usuário deseja que o computador realize. Antes que um programa possa ser executado, deverá ser submetido ao compilador e, depois, ao *linker* do UCI.

O processo de compilação, aplicado aos arquivos .UC, produz arquivos de código-objeto com a extensão .OBJ. Estes arquivos contêm instruções em linguagem de máquina que fazem sentido somente para os microprocessadores Intel 8088, 8086, 80186, 80286, 80386 ou 80486. O compilador irá produzir um arquivo .OBJ com o mesmo nome do arquivo .UC compilado.

Ao ser executado, o *linker* ou montador toma um ou mais arquivos .OBJ e, como seu próprio nome diz, processa-os juntos para produzir um arquivo executável com a extensão .EXE. O *linker* pode também processar o código proveniente de bibliotecas padrões pré-compiladas. Uma destas bibliotecas contém as funções necessárias para a utilização da linguagem ULTRA C.

Como pode-se ver na figura 2.5, o sub-menu da opção *Compilar* apresenta três itens:

1. *Compilar para .OBJ*;
2. *Gerar .ASM*;
3. *Montar .EXE*.

A Opção *Compilar para .OBJ*

Nesta opção, o UCI faz uma chamada apenas ao compilador C para que este produza o código-objeto do programa corrente.

Quando esta opção é acionada, o UCI inicia a compilação do programa-fonte contido na área de trabalho. Se o programa não foi alterado desde a sua última compilação, então o UCI avisa que não será necessária uma nova compilação.

A figura 2.6 mostra a fase de compilação de um programa escrito em ULTRA C.

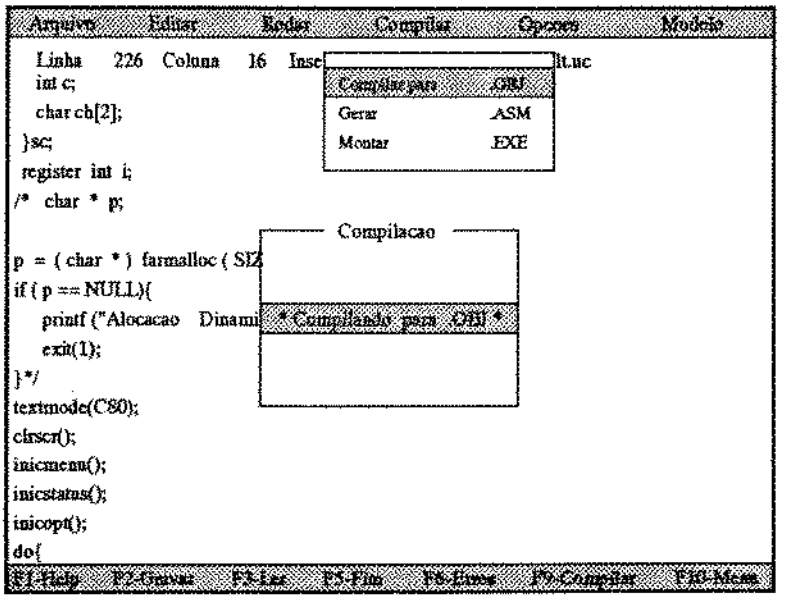


Figura 2.6: *Fase de compilação de um programa*

Qualquer erro detectado durante a compilação impedirá a geração do arquivo .OBJ.

Warnings são avisos de que existe a possibilidade do programador ter cometido um erro durante a edição do programa. Na maior parte dos casos, o *warning* é apenas uma advertência feita pelo compilador quando encontra uma operação que pode (ou não) conter um engano cometido pelo programador. Um exemplo é a atribuição de valores com tipos de dados incompatíveis ou a utilização de uma variável em uma expressão, antes que um valor lhe seja atribuído.

Aparentemente inofensivos, os *warnings* são as maiores causas de *bugs*. A presença de *warnings* **não** impede que o arquivo .OBJ seja gerado.

Qualquer erro ou *warning* detectado durante a fase de compilação será mostrado pelo UCI como ilustra a figura 2.7.

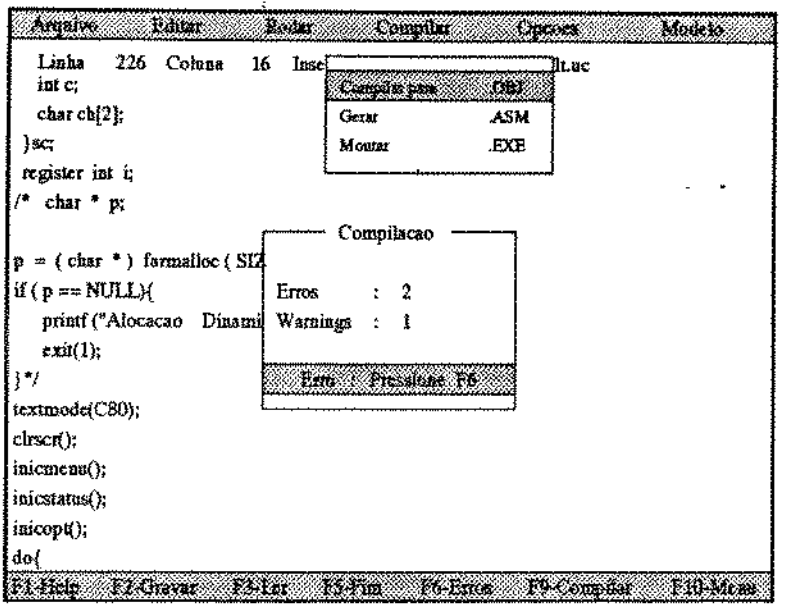


Figura 2.7: *Erros e warnings detectados durante a compilação*

Quando um erro ou *warning* é encontrado, o UCI permite ao usuário, através do pressionamento da *hot-key* **F6**, ver as mensagens geradas pelo compilador.

A figura 2.8 mostra as mensagens de erro provenientes da compilação realizada anteriormente.

A primeira palavra de uma mensagem gerada indica se ela é proveniente de um erro ou *warning*.

Logo a seguir vem o *path* do arquivo compilado.

O número que aparece em seguida, indica a linha do programa-fonte a que se reporta a mensagem.

Logo após o número da linha, vem uma breve explicação sobre a causa do erro ou *warning*.

Se o usuário pressionar **ENTER** sobre uma mensagem, a tela de mensagens dará lugar à área de trabalho e o cursor será posicionado na primeira coluna da linha em que o erro (ou *warning*) ocorreu.

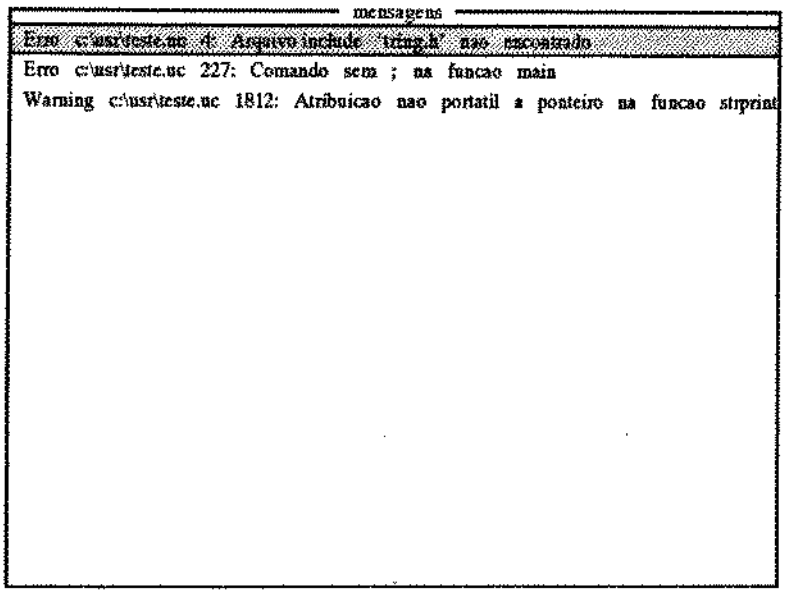


Figura 2.8: *Tela de mensagens após a compilação*

A Opção *Gerar .ASM*

Esta opção oferece ao usuário a possibilidade de gerar um arquivo texto .ASM que contém código em assembler. A grande vantagem em se fazer isto é que se o usuário necessita de maior velocidade do programa ou acesso a algumas características específicas impossíveis de serem realizadas em ULTRA C, não será necessário desenvolvimento de um programa em assembler. Basta escrever um programa equivalente em ULTRA C, gerar o arquivo .ASM e efetuar as alterações necessárias.

Esta opção também submete o programa corrente ao compilador, da mesma forma que a opção anterior.

A Opção *Montar .EXE*

A última opção do sub-menu *Compilar* realiza a geração do arquivo .OBJ e posteriormente o *linka* com as bibliotecas padrões, gerando o arquivo executável .EXE.

O usuário não precisa se preocupar com quais bibliotecas o seu arquivo deve ser linkado. O UCI verifica isto automaticamente.

2.4.5 A Opção *Opções*

Algumas características do UCI são customizáveis. Esta opção do menu principal oferece ao usuário a possibilidade de *customizar* seu ambiente de trabalho.

Quando esta opção é acionada, o UCI mostra a tela ilustrada na figura 2.9.

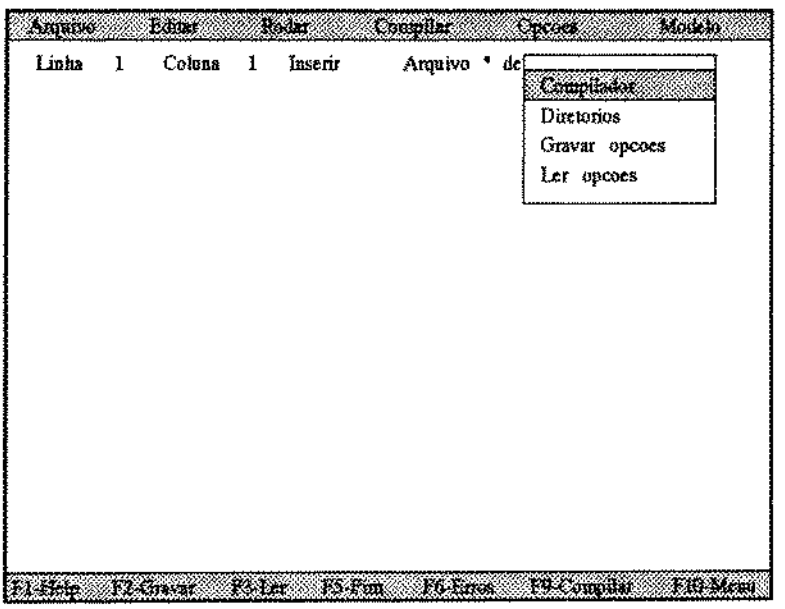


Figura 2.9: *Opções de customização do UCI*

O sub-menu de customização do UCI apresenta quatro itens:

1. *Compilador*;
2. *Diretórios*;
3. *Gravar opções*;
4. *Ler Opções*;

A Opção *Compilador*

Nesta opção o usuário pode determinar algumas características que serão levadas em consideração pelo compilador durante a geração do código objeto.

Quando esta opção é acionada, o UCI mostra um sub-menu como ilustrado na figura 2.10.

Como mostra a figura, a opção *Compilador* permite acessar quatro itens:

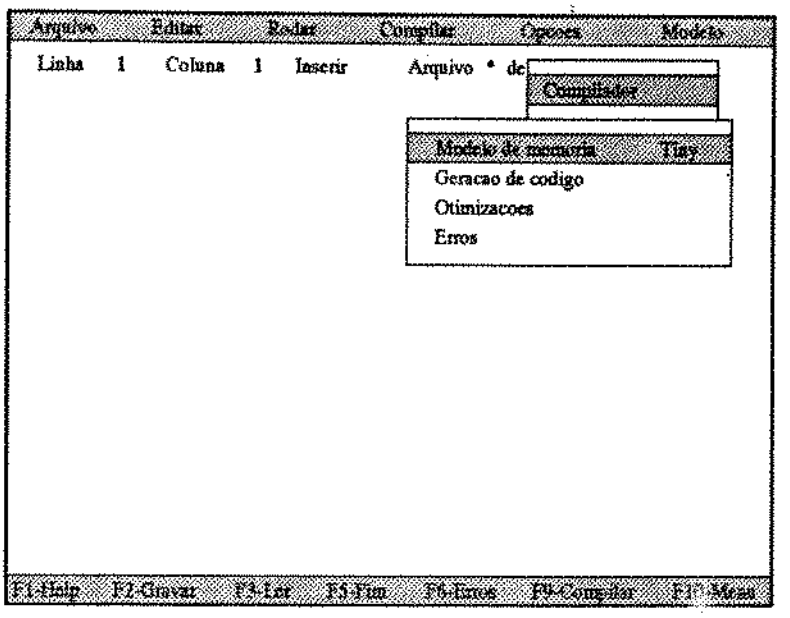


Figura 2.10: *O sub-menu da opção* Compilador

1. **Modelo de Memória**. Nesta opção o usuário determina qual o modelo de memória que o compilador deve levar em consideração. A ativação desta opção fornece o sub-menu mostrado na figura 2.11. No ANSI C foram definidos seis modelos de memória [16, 30, 31] diferentes:

   a) **Tiny** : Todos os registradores de segmento recebem o mesmo valor e todo endereçamento é feito com apenas 16 bits. Isto significa que código, dados, e a pilha (ou *stack*) devem estar contidos em um mesmo segmento de memória de 64K. A utilização deste modelo de memória torna a execução do programa bastante rápida;

   b) **Small** : Todo o código deve estar contido em um segmento de 64K e todos os dados devem estar em um segundo segmento de 64K. Todos os ponteiros possuem 16 bits. A execução é tão rápida quanto no modelo tiny;

   c) **Medium** : Todos os dados do programa devem estar contidos em um único segmento de 64K mas o código pode utilizar múltiplos segmentos. Todos os ponteiros de dados têm 16 bits mas todas as chamadas (*calls*) e saltos (*jumps*) utilizam ponteiros de 32 bits. Esta característica faz com que programas compilados neste modelo de memória tenham um rápido acesso aos dados mas uma lenta execução das instruções;

   d) **Compact** : Todo o código deve estar contido em um único segmento de 64K mas os dados podem utilizar múltiplos segmentos. Entretanto, nenhum item de dado pode exceder 64K. Todos os ponteiros de dados possuem 32 bits mas *calls* e *jumps* utilizam ponteiros de 16 bits. Acesso lento aos dados com rápida execução de instruções;

   e) **Large** : Ambos, código e dados, podem utilizar múltiplos segmentos de 64K. Todos os

ponteiros possuem 32 bits. Entretanto, nenhum item de dados pode exceder 64K. Baixa velocidade de execução e acesso aos dados;

f) **Huge** : Tanto código quanto dados podem utilizar múltiplos segmentos de memória. Todos os ponteiros possuem 32 bits. Os itens de dados podem exceder 64K. Programas compilados neste modelo de memória possuem a mais baixa velocidade de execução.

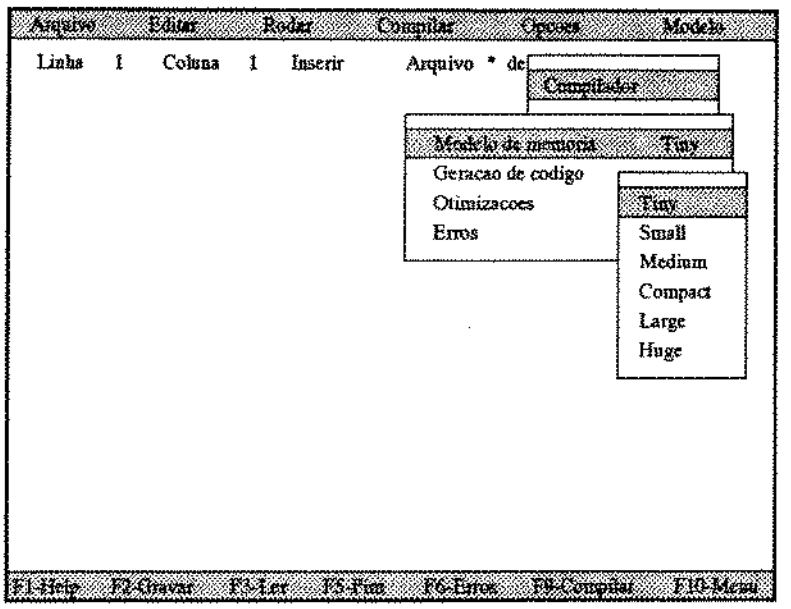


Figura 2.11: *O sub-menu* Modelo de memória

2. **Geração de código**. Nesta opção o usuário pode definir algumas características relacionadas ao código gerado pelo compilador. A figura 2.12 mostra o efeito da ativação desta opção;

   Quando a opção *Geração de código* está ativada, o UCI mostra um sub-menu contendo as seguintes palavras-chaves:

   (a) **Instruções** : Se o programa for compilado com a opção *8086/8088*, o código .OBJ gerado somente conterá instruções reconhecidas por estes microprocessadores e, consequentemente, pelas gerações de microprocessadores posteriores a eles. Neste caso, o programa poderá ser executado por todos os microprocessadores da linha Intel 80XXX. Se o programa for compilado com a opção *80186/80286*, o código .OBJ gerado conterá instruções reconhecidas pelos microprocessadores Intel 8088 e 8086 assim como instruções exclusivas dos microprocessadores 80186 e 80286. Isto significa que programas compilados com a última opção não podem ser executados pelos microprocessadores 8088 ou 8086;

   (b) **Ponto Flutuante** : Se o programa for compilado com a opção *8087/80187*, todos

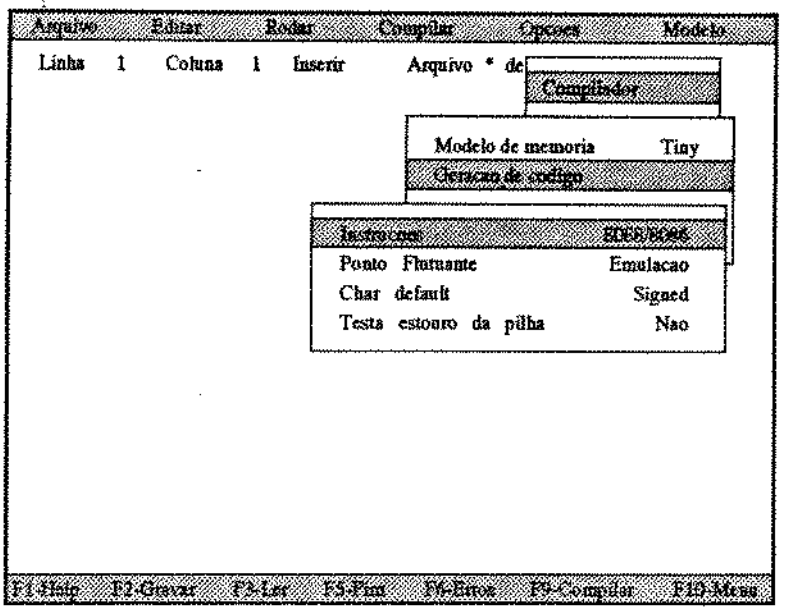


Figura 2.12: *O sub-menu* Geração de Código

os cálculos em ponto flutuante serão realizados através de instruções específicas destes co-processadores matemáticos. Isto resulta numa maior velocidade de execução. Se os referidos co-processadores (ou gerações posteriores a eles) não estiverem presentes, deve-se compilar o programa com a opção *Emulação*. Neste caso, todas as operações em ponto flutuante serão emuladas por *software*;

(c) **Char default** : Em ULTRA C, assim como em C, existe um tipo de dado chamado *char* que define variáveis, funções ou ponteiros como sendo do tipo caracter. Em geral, toda variável definida como *char* ocupa oito bits na memória. Se o programa for compilado com a opção *Signed*, todos as variáveis definidas como char poderão conter valores entre -127 e +127. Se o programa for compilado com a opção *Unsigned*, estas variáveis poderão conter valores entre 0 e 255;

(d) **Testa estouro da pilha** : Toda vez que um *call* ou *jump* é executado pelo microprocessador, o endereço de retorno é armazenado num local da memória chamado pilha ou *stack*. Quando uma instrução de retorno é encontrada, o endereço de retorno é retirado da pilha. Instruções *push* e *pop* também afetam a pilha. Quando o tamanho da pilha cresce muito, devido a múltiplas chamadas sem que haja retorno ou a um grande número de instruções *push* sem o respectivo *pop*, existe o risco de haver uma colisão com áreas importantes da memória. Se o programa for compilado com a opção *Sim*, o compilador acrescentará ao código gerado, uma rotina de verificação toda vez que acessos à pilha são realizados. Caso um estouro

da pilha seja detectado, o programa é encerrado, uma mensagem de erro é gerada e o controle é devolvido ao sistema operacional. Com a opção *Sim* ativa, o programa gerado torna-se mais lento. Se o programa for compilado com a opção *Não*, nenhuma verificação será realizada;

3. **Otimizações.** Nesta opção o usuário pode otimizar a utilização das instruções geradas pelo compilador. A ativação desta opção é mostrada na figura 2.13;

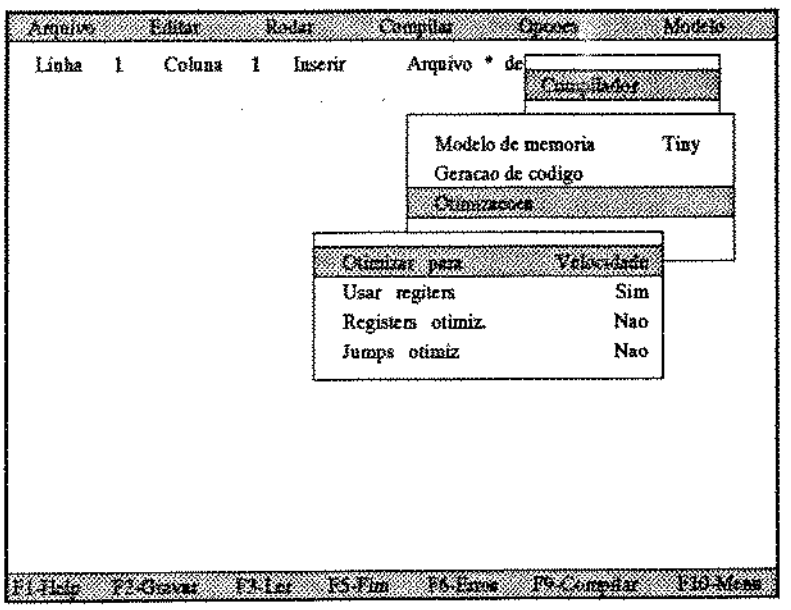


Figura 2.13: *O sub-menu* Otimizações

O uso de registradores internos para armazenar valores de variáveis, incluindo os parâmetros formais das funções, é um método comum empregado pelos compiladores durante a otimização.

Uma otimização descontrolada pode gerar alguns efeitos estranhos e catastróficos. Por isso, alguns compiladores, como o Turbo C, permitem que o usuário selecione entre diversas estratégias de otimização, incluindo a desabilitação total do processo de otimização.

A ativação da opção *Otimizações* apresenta um sub-menu com as seguintes palavras-chaves:

(a) **Otimizar para** : Nesta opção o usuário pode fazer com que o compilador escolha entre tamanho ou rapidez na geração da sequência de código;

(b) **Usar registers** : Quando a opção *Sim* estiver ativa, o compilador tentará utilizar registradores internos para armazenar os valores das variáveis definidas como *auto*. Caso a opção *Não* esteja ativa, o modificador de tipo *register* será ignorado;

(c) **Registers otimiz.** : Quando a opção *Sim* está ativa o compilador tentará evitar os movimentos desnecessários de dados entre a RAM e os registradores internos. Cuidados devem ser tomados quando se alterar o valor de uma variável através da utilização de ponteiros pois, neste caso, o compilador será incapaz de detectar tais alterações;

(d) **Jump otimiz.** : Quando a opção *Sim* está ativa, o compilador tentará reduzir o tamanho do código gerado, eliminando as operações ramificadas redundantes, comprimindo os *loops* e os comandos *switch...case*;

4. **Erros.** Nesta opção o usuário define a quantidade de erros e *Warnings* que a compilação de um programa pode gerar. A ativação desta opção é mostrada na figura 2.14;

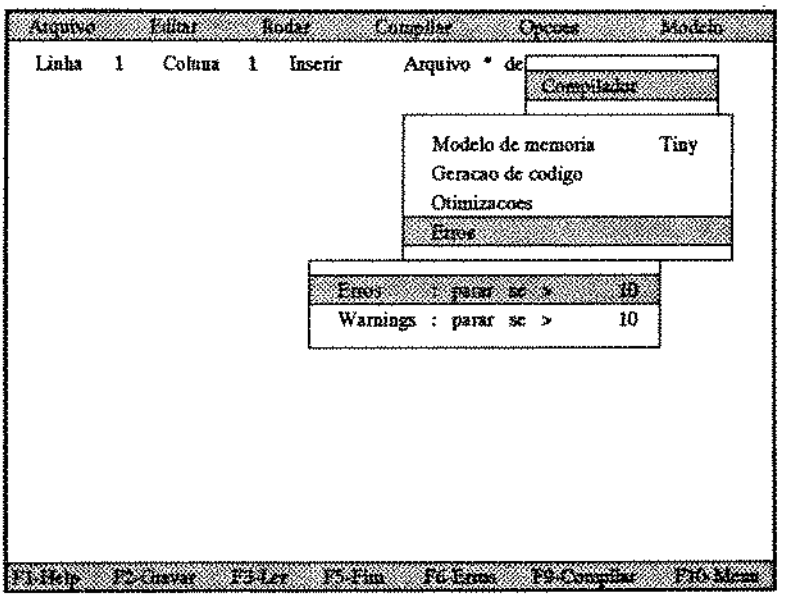


Figura 2.14: *O sub-menu* Erros

A Opção *Diretórios*

Nesta opção o usuário pode definir os diretórios utilizados pelo UCI. A figura 2.15 mostra a ativação desta opção.

O UCI utiliza quatro diretórios além do diretório corrente:

1. **Include.** Onde se encontram os arquivos-cabeçalhos;
2. **Output.** Onde são armazenados os arquivos .UC,.OBJ, .ASM e .EXE;

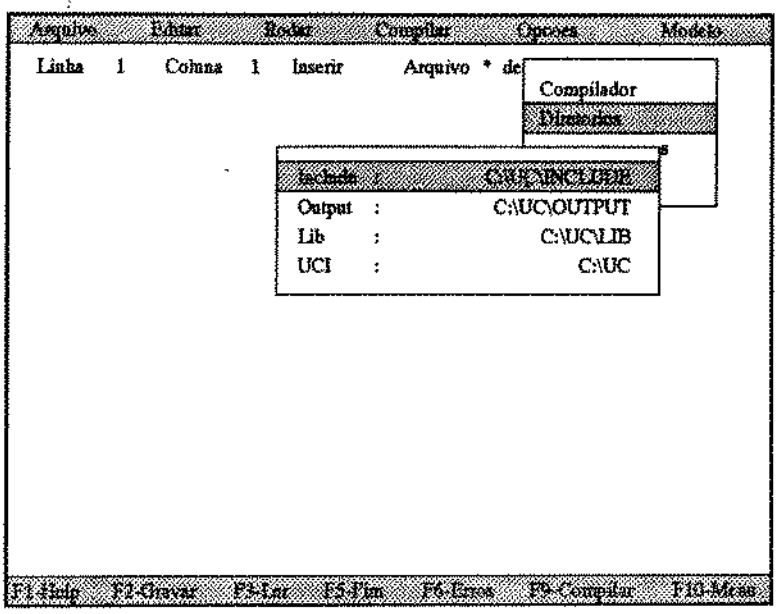


Figura 2.15: *O sub-menu* Diretórios

3. **Lib.** Onde se encontram as bibliotecas de funções;

4. **UCI.** Onde se encontram os principais programas do UCI, inclusive o UCI.EXE;

A Opção *Gravar opções*

Nesta opção o usuário pode gravar em um arquivo as alterações feitas no menu de *Opções* para que não seja necessária a alteração toda vez que se iniciar uma sessão de trabalho.

A ativação desta opção é mostrada na figura 2.16.

A Opção *Ler opções*

Nesta opção, o usuário pode recuperar o arquivo de alterações gravado no item anterior.

2.4.6 A Opção *Modelo*

Nesta opção, o usuário tem acesso ao modelador gráfico embutido no UCI. Nele, o usuário pode modelar graficamente tanto o robô quanto seu ambiente de trabalho.

A utilização do modelador gráfico do UCI será detalhada no capítulo *O Modelador Gráfico*.

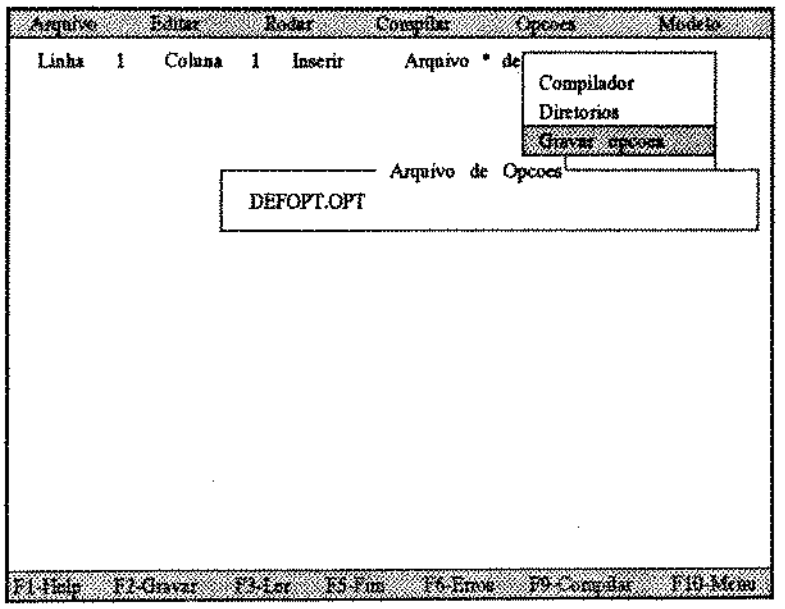


Figura 2.16: *Salvando as alterações efetuadas*

Neste capítulo, procurou-se detalhar o ambiente UCI, suas características básicas e forma de utilização. Todos os recursos apresentados pelo UCI possuem uma única finalidade: fornecer subsídios para que o usuário possa realizar a programação de tarefas, utilizando a linguagem ULTRA C, de forma cômoda e eficiente. A origem, sintaxe, tipologia de dados, funções e características básicas da linguagem ULTRA C serão itens abordados no próximo capítulo.

Capítulo 3

A Linguagem ULTRA C

3.1 Introdução

Como dito anteriormente, a linguagem de programação de robôs ULTRA C (Uma Linguagem Textual para Robótica e Automação usando C) é uma extensão das bibliotecas do ANSI C [16, 30, 31]. Algumas funções relacionadas à automação, em especial, à robótica foram desenvolvidas e compiladas em uma biblioteca que deve ser *linkada* com o programa do usuário.

Dentre os motivos que levaram à escolha da linguagem C pode-se citar:

1. A linguagem C é muito poderosa no sentido de que permite o acesso a todos os recursos do sistema tais como manipulação de bits, bytes, palavras e endereços de memória;
2. É uma linguagem portátil. Por *portabilidade* entenda-se a capacidade de se transportar um programa de uma plataforma de *hardware* para outra com um mínimo de alteração;
3. É uma linguagem estruturada;
4. É, tradicionalmente, uma linguagem compilada;
5. Tem um *núcleo* pequeno, ou seja, possui apenas 32 palavras-chaves (27 provenientes do padrão Kernighan & Richie mais 5 adicionadas pelo comitê de padronização ANSI). Apenas a título de comparação, o BASIC para IBM PC possui 159 palavras-chaves;
6. Permite quase todas as conversões de tipos. Por exemplo, pode-se misturar variáveis definidas como tipo caracter e variáveis definidas como tipo inteiro em uma mesma expressão aritmética;
7. É uma linguagem de programadores. Surpreendentemente, nem todas as linguagens de programação foram desenvolvidas para programadores. Considere o exemplo clássico de

duas linguagens que não foram desenvolvidas para programadores: COBOL e BASIC. COBOL foi desenvolvida para permitir que um *leigo* pudesse ler e (presumivelmente) entender o programa. BASIC foi desenvolvida para permitir que alguém que possuísse um mínimo de conhecimento de programação pudesse programar um computador para resolver problemas bastante simples. Nenhuma destas linguagens, ao contrário de C, tem como objetivo a otimização do processo de desenvolvimento, o aumento da eficiência e *performance* do código gerado ou a quantidade de recursos disponíveis para o programador.

3.2 A Linguagem C

A linguagem C foi pela primeira vez implementada por Dennis Ritchie em um computador DEC PDP-11 sob o sistema operacional UNIX. C é o resultado de um processo de evolução iniciado com uma antiga linguagem chamada BCPL. BCPL foi desenvolvida por Martin Richards, e influenciou uma linguagem chamada B, criada por Ken Thompson. B evoluiu para a linguagem C nos anos 70.

Por muitos anos, o padrão C foi o fornecido com o sistema operacional UNIX versão 5 [17]. Com o aumento da popularidade dos microcomputadores, um grande número de implementações de C foram desenvolvidas. Em pouco tempo, os programas desenvolvidos em cada uma destas implementações tornaram-se completamente incompatíveis entre si. Para resolver este problema, em 1983 formou-se um comitê que definiria o padrão a ser adotado em todas as implementações futuras da linguagem C. Surgiu então o ANSI C no qual se baseia a linguagem ULTRA C.

A tabela 3.1 mostra a posição da linguagem C em relação a outras linguagens de programação de computador [16].

Embora C seja considerada uma linguagem de nível médio, isto não significa que ela seja uma linguagem com poucos recursos ou de difícil compreensão.

C é considerada uma linguagem estruturada com alguma semelhança com o Algol e o Pascal. De forma mais formal, o que define a característica de estruturação de uma linguagem de programação é a *compartimentalização de código e dados*. Isto é, a linguagem estruturada pode *esconder* do resto do programa todas as informações necessárias para se realizar determinada tarefa. Esta característica é geralmente obtida através da utilização de variáveis locais temporárias. Desta forma, torna-se possível escrever sub-rotinas onde os eventos dentro delas

Linguagens de Alto Nível	Ada
	Modula-2
	Pascal
	Cobol
	Fortran
	Basic
Linguagens de Nível Medio	C
	Forth
	Macro-Assembler
Linguagens de Baixo Nível	Assembler
	Linguagem de Máquina

Tabela 3.1: *Classificação das linguagens de programação quanto ao nível de programação.*

não provoquem efeitos colaterais em outras partes do programa.

A tabela 3.2 mostra algumas linguagens de programação classificadas em estruturadas e não estruturadas [16].

Não Estruturadas	Linguagem de Máquina
	Assembler
	Macro-Assembler
	Forth
	Fortran
	Basic
	Cobol
Estruturadas	C
	Ada
	Modula-2
	Pascal

Tabela 3.2: *Classificação das linguagens de programação quanto à estruturação.*

Funções, tanto em C como em ULTRA C, são blocos de código onde ocorre toda a atividade do programa. Uma função, depois de depurada, pode ser utilizada em várias partes do programa ou em outros programas. Esta característica possibilitou a criação das funções relacionadas à robótica que ULTRA C utiliza.

3.3 Sintaxe da Linguagem ULTRA C

ULTRA C possui basicamente a mesma sintaxe do ANSI C. A diferença entre as duas está no acréscimo de uma biblioteca de funções relacionadas à automação e robótica (*UCLIB*) e na criação de alguns novos tipos de dados.

Todo programa escrito em ULTRA C deve ter como primeira instrução a macro-definição *start_uc( )* e como última instrução a macro-definição *end_uc( )*. Estas duas macros são responsáveis respectivamente pela alocação e liberação dinâmica de memória necessária para a armazenagem dos dados relativos aos *links* do robô.

Toda função deve ser referenciada pelo seu nome seguido dos parâmetros entre parênteses. Os parênteses devem estar presentes mesmo quando a função não necessitar de parâmetros. Por isso, em todo o texto, qualquer referência a uma determinada função apresentará os parênteses.

Deve-se sempre ter em mente que o compilador utilizado no UCI assume que os nomes de variáveis, funções, etc., podem conter caracteres minúsculos ou maiúsculos. Assim, uma função definida como *approach( )* deve ser utilizada sempre com caracteres minúsculos. Chamadas à *Approach( )* ou *APPROACH( )* resultarão em erro de compilação.

Nas seções seguintes, apresenta-se os novos tipos de dados bem como cada uma destas funções, sua finalidade, parâmetros de entrada e dados de saída.

3.4 Tipos de dados

Além dos tipos de dados existentes no ANSI C [16, 30, 31], ULTRA C possui os seguintes tipos previamente incorporados:

VPOINT

Coordenadas virtuais (x,y) de um ponto no espaço tridimensional. Por ponto virtual entenda-se a representação na tela gráfica de um ponto no espaço cartesiano. Este tipo está definido no arquivo cabeçalho *UC.H* como:

```
typedef struct{
    int x, y, z;
} VPOINT;
```

O parâmetro *z* definido no tipo *VPOINT* representa a coordenada z no espaço tridimensional.

POINT

Coordenadas reais *(x,y,z)* de um ponto no espaço tridimensional. Este tipo está definido no arquivo cabeçalho *UC.H* como:

```
typedef struct{
        int x, y, z;
} POINT;
```

FRAME

As variáveis definidas com este tipo contêm informação de posição *(x,y,z)* e orientação *(a,b,c)* no espaço tridimensional. Este tipo está definido no arquivo *UC.H* como:

```
typedef struct{
        POINT p;
        EULER o;
} FRAME;
```

CPATH

Conjunto de até *PATHSIZE frames* que formam um caminho no espaço cartesiano. Este tipo está definido no arquivo *UC.H* como:

```
typedef struct{
        POINT[PATHSIZE];
}CPATH;
```

JOINT

Conjunto de até *JOINTSIZE* variáveis de junta. Este tipo está definido no arquivo *UC.H* como:

```
typedef struct{
    float theta[JOINTSIZE];
}JOINT;
```

JPATH

Conjunto de até *PATHSIZE joints* que formam um caminho no espaço de juntas. Este tipo está definido no arquivo *UC.H* como:

```
typedef struct{
    JOINT[PATHSIZE];
}JPATH;
```

SHAPE

Contêm informações sobre as primitivas geométricas utilizadas em ULTRA C. As primitivas são detalhadas no capítulo *O Modelador Gráfico*. Este tipo está definido no arquivo *UC.H* como:

```
typedef struct{
    char tipo;
    unsigned parm[8];
    int x[60],y[60],z[60];
}SHAPE;
```

OBJECT

Contêm informações sobre os objetos (primitivas geométricas acrescidas de informações de posição e orientação) utilizadas nas *estruturas* rígidas. Este tipo está definido no arquivo

UC.H como:

```
typedef struct{
        SHAPE s;
        int x,y,z;
        float a, b, c;
        int seq;
}OBJECT;
```

STRUCTURE

Conjunto de até *STRSIZE* objetos agrupados de forma rígida com posição e orientação. Toda estrutura possui um *ponto de pega* que determina a posição e orientação do elemento terminal do robô caso seja necessária a sua manipulação. Os *links* do robô também são definidos através de estruturas. Este tipo está definido no arquivo *UC.H* como:

```
typedef struct{
        OBJECT o[STRSIZE];
        int count;
        int x, y, z;
        float a, b, c;
        int seq;
        FRAME tp;
}STRUCTURE;
```

3.5 Funções

Todas as funções da linguagem ULTRA C estão agrupadas na biblioteca *UC.LIB*. Além das provenientes do padrão ANSI, ULTRA C possue as seguintes funções:

3.5.1 Funções de Definição

São funções relacionadas à definição de dados que serão manipulados pelo programa.

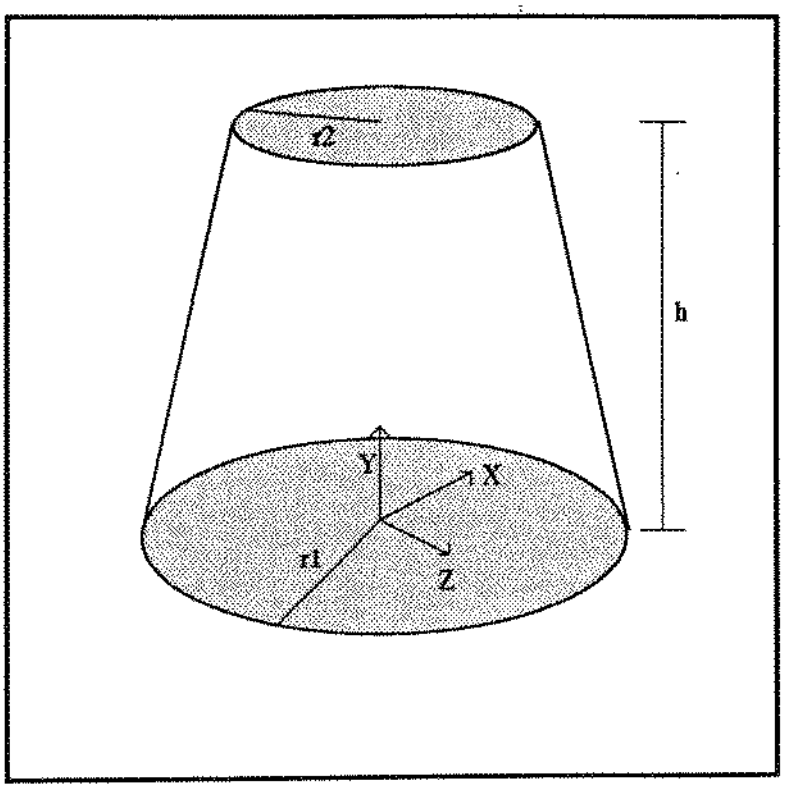


Figura 3.1: *Definição da primitiva* cone.

cone( )

Definição: SHAPE cone (unsigned r1,unsigned r2, unsigned h, unsigned v);

Descrição: Esta função tem como objetivo a geração de um SHAPE tipo tronco de cone com raio das bases dados em milímetros pelos parâmetros **r1 e r2**, comprimento em milímetros dado pelo parâmetro **h** e número de arestas dado pelo parâmetro **v**. Ver figura 3.1.

Retorno: retorna o tipo SHAPE resultante se os parâmetros são válidos ou um *SHAPE nulo* em caso contrário.

defhome( )

Definição: int defhome (JOINT j);

Descrição: Esta função tem como objetivo a definição de uma posição de *home* para o robô. Toda vez que houver uma chamada à função **home( )**, o robô executará um movimento interpolado no espaço de junta de sua posição corrente até a posição definida pelo ponto **j**.

Retorno: retorna zero caso o ponto **j** esteja dentro do volume de trabalho do robô e um código de erro em caso contrário.

defobserver( )

Definição: int defobserver (POINT p1,POINT p2);

Descrição: Esta função tem como objetivo a definição da posição do observador no interior do volume de trabalho do robô. O observador será posicionado no ponto *p1*, olhando para o ponto *p2*. Os pontos *p1* e *p2* devem estar dentro do volume definido através da função **defvolume( )**. Todas as funções relacionadas à simulação gráfica considerarão o ponto de vista definido por esta função como a posição corrente do observador em relação à cena.

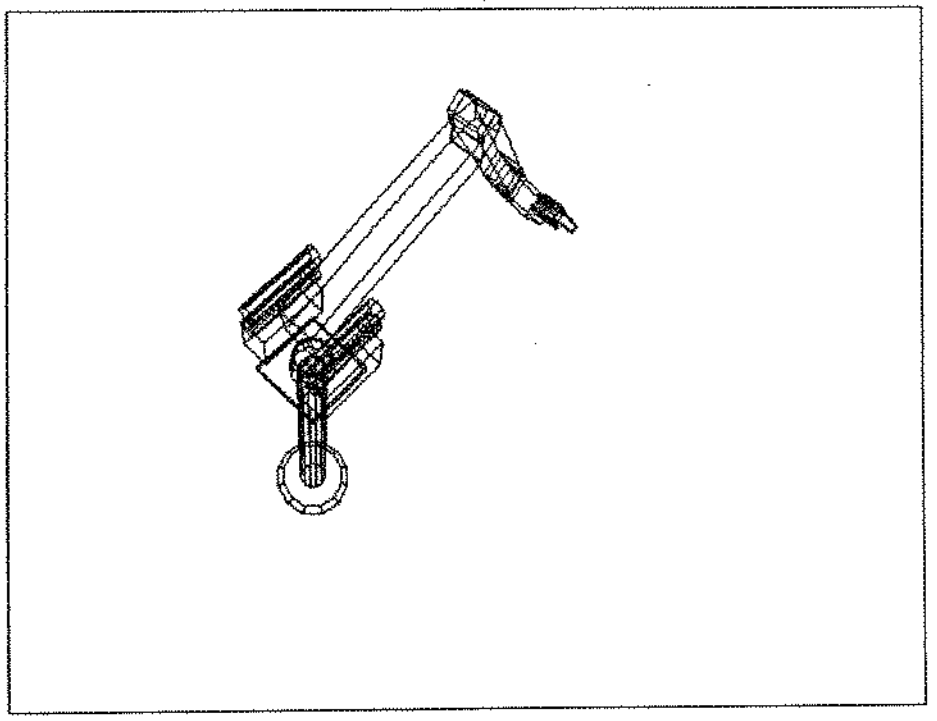


Figura 3.2: *Vista superior de uma operação simulada.*

A figura 3.2 mostra uma operação simulada utilizando o modelo gráfico do manipulador Kraft. Por uma questão de clareza, optou-se por apresentar somente o manipulador, omitindo-se os objetos definidos no interior do seu volume de trabalho. Nesta simulação, o observador está posicionado a uma altura de aproximadamente três metros da base do manipulador.

Na figura 3.3, apresenta-se um recurso adicional fornecido pela linguagem ULTRA C: múltiplas vistas de uma mesma operação. Nesta segunda simulação, o observador está posici-

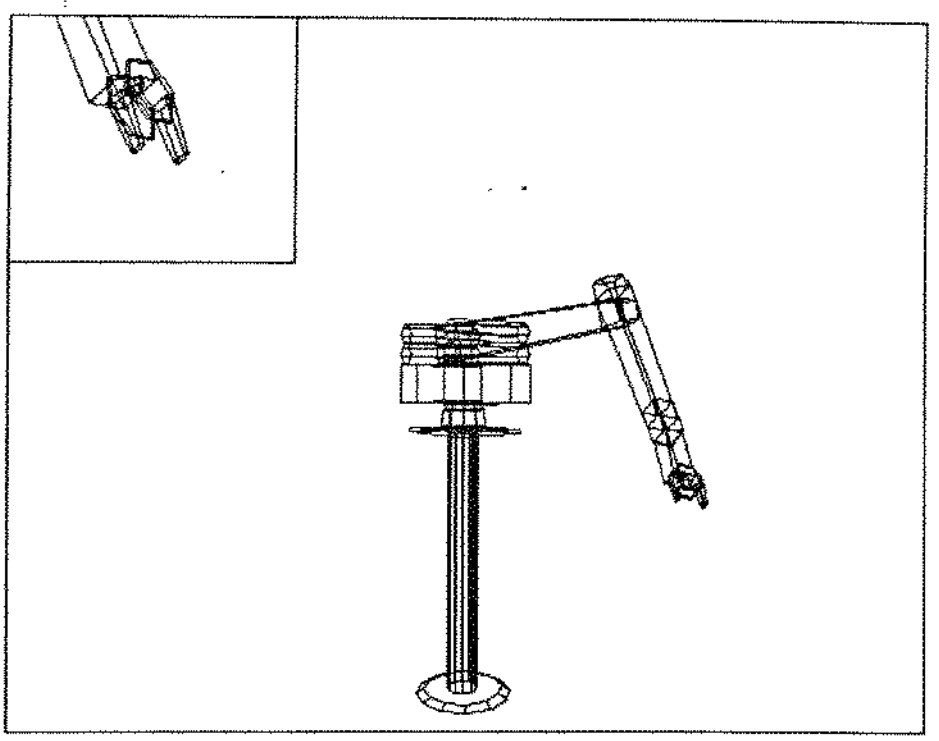


Figura 3.3: *Vista lateral com o elemento terminal em detalhe.*

onado lateralmente, vendo também, em *close*, o elemento terminal do manipulador. Um efeito bastante interessante foi obtido nesta simulação: o elemento terminal (no canto superior esquerdo da figura) é acompanhado durante toda execução da tarefa, simulando a presença de uma câmera posicionada sempre junto a ele. Este efeito foi conseguido utilizando-se a função **getframe( )** para redefinir a posição do observador durante o movimento do manipulador.

Retorno: retorna zero se os parâmetros são válidos e um código de erro em caso contrário.

defvolume( )

Definição: int defvolume (int maxx,int maxy,int maxz);

Descrição: Esta função tem como objetivo a definição do volume máximo que será utilizado na simulação gráfica. O volume definido por esta função deve englobar o *volume de trabalho* do robô. Após esta definição, as coordenadas tridimensionais deverão estar dentro dos seguintes intervalos:

1. X : [-maxx,maxx];

2. Y : [-maxy,maxy];

3. Z : [-maxz,maxz].

Retorno: retorna zero se os parâmetros são válidos e um código de erro em caso contrário.

denavit( )

Definição: int denavit (int n,float alpha,float a, float d, float theta, float type, float min, float max,float maxvel);

Descrição: Esta função tem como objetivo a definição de todos os parâmetros necessários à caracterização geométrica e cinemática de cada junta.
O parâmetro **n** indica o número da junta. A numeração das juntas pode variar de 1 a 13.
Os parâmetros **alpha, a, d** e **theta** são os parâmetros de Denavit-Hartemberg [7, 10, 14, 26] para a junta **n**.
O parâmetro **type** indica se a junta é prismática ou rotacional. Pode assumir os seguintes valores:

1. PRI : junta prismática;

2. ROT : junta rotacional.

Os parâmetros **min** e **max** definem as restrições mecânicas de posição da junta em questão. Se a junta for do tipo prismática, este parâmetro deve ser fornecido em *mm*, caso seja rotacional, em *rad*.
O parâmetro **maxvel** define a velocidade máxima da junta. A unidade de medida deve estar de acordo com o tipo de junta: *mm/s ou rad/s*, dependendo do caso.

Retorno: retorna zero se os parâmetros são válidos e um código de erro em caso contrário.

doubledisc( )

Definição: SHAPE doubledisc (unsigned r1,unsigned r2, unsigned a, unsigned l);

Descrição: Esta função tem como objetivo a geração de um SHAPE tipo duplodisco com raio dos discos dados (em mm) pelos parâmetros **r1 e r2**, largura (em mm) dada pelo

parâmetro **a** e distância entre os centros dos discos (em mm) dado pelo parâmetro **l**. O parâmetro **l** deve ser maior ou igual ao módulo da diferença entre os raios. Ver figura 3.4.

Retorno: retorna o tipo SHAPE resultante se os parâmetros são válidos ou um *SHAPE nulo* em caso contrário.

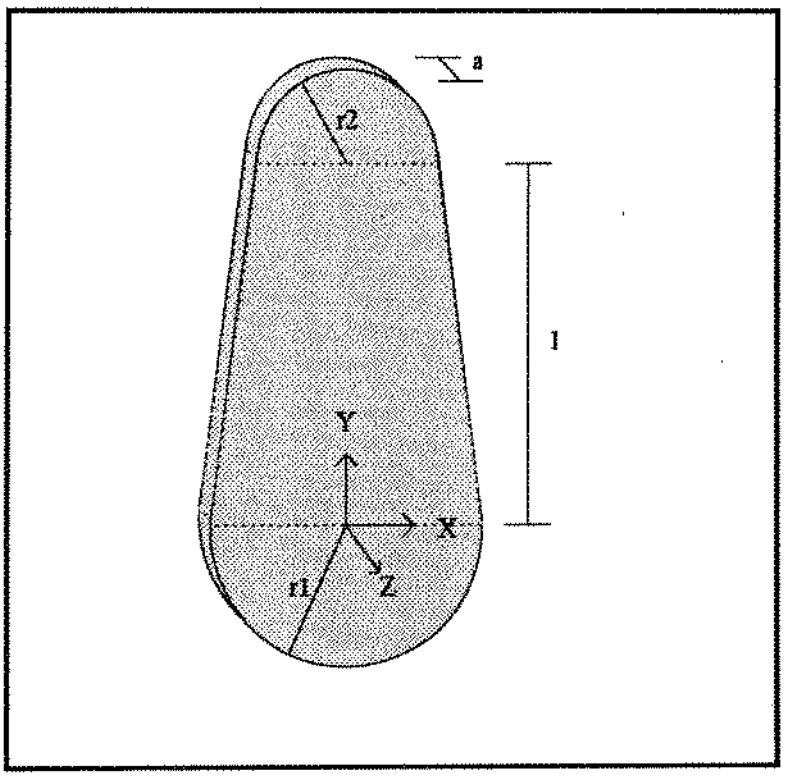


Figura 3.4: *Definição da primitiva* doubledisc.

interpol( )

Definição: int interpol (int tipo);

Descrição: Esta função tem como objetivo *setar* o tipo de interpolação a ser realizada pelas funções de movimentação. O parâmetro **tipo** pode ter os seguintes valores:

1. PTP : a interpolação utilizada será ponto-a-ponto, ou seja, linear no espaço de juntas;
2. LINEAR : a interpolação utilizada será linear no espaço cartesiano;

Retorno: retorna zero se o parâmetro for válido e um código de erro em caso contrário.

obj( )

Definição: OBJECT obj (SHAPE s, POINT p, float a, float b, float c, int seq);

Descrição: Esta função tem como objetivo criar um objeto a partir de uma primitiva **s**. A posição do novo objeto será **p** (em mm) e sua orientação será (a,b,c) (em rad). O parâmetro **seq** define a sequência de rotação em torno dos eixos coordenados:

1. XYZ : em torno do eixo X, em torno do eixo Y e em torno do eixo Z;
2. XZY : em torno do eixo X, em torno do eixo Z e em torno do eixo Y;
3. YXZ : em torno do eixo Y, em torno do eixo X e em torno do eixo Z;
4. YZX : em torno do eixo Y, em torno do eixo Z e em torno do eixo X;
5. ZXY : em torno do eixo Z, em torno do eixo X e em torno do eixo Y;
6. ZYX : em torno do eixo Z, em torno do eixo Y e em torno do eixo X;

Retorno: retorna o objeto obtido ou um *OBJECT nulo* caso os parâmetros não sejam válidos.

pyramid( )

Definição: SHAPE pyramid (unsigned a1,unsigned b1, unsigned a2, unsigned b2, unsigned h);

Descrição: Esta função tem como objetivo a geração de um SHAPE tipo tronco de pirâmide com largura das bases dadas (em mm) pelos parâmetros **a1** e **a2**, comprimento das bases dadas (em mm) pelos parâmetro **b1** e **b2** e altura dada (em mm) pelo parâmetro **h**. Ver figura 3.5.

Retorno: retorna o tipo SHAPE resultante se os parâmetros são válidos ou um *SHAPE nulo* em caso contrário.

setninterpol( )

Definição: int setninterpol (unsigned n);

Descrição: Esta função tem como objetivo determinar o número de pontos de discretização a serem utilizados pelas rotinas de interpolação de trajetória.

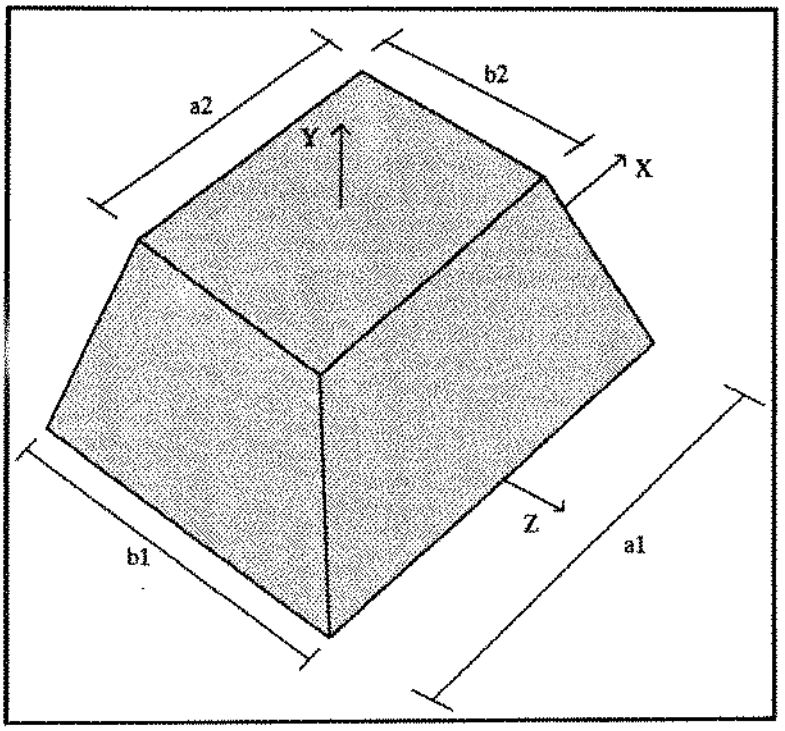


Figura 3.5: *Definição da primitiva* pyramid.

Retorno: retorna zero se o parâmetro **n** for válido ou um código de erro em caso contrário.

square( )

Definição: SHAPE square (unsigned a,unsigned b);

Descrição: Esta função tem como objetivo a geração de um SHAPE tipo retângulo com comprimento dado (em mm) pelo parâmetro **a** e largura dada (em mm) pelo parâmetro **b**. Ver figura 3.6.

Retorno: retorna o tipo SHAPE resultante se os parâmetros são válidos ou um *SHAPE nulo* em caso contrário.

3.5.2 Funções Matemáticas

DGM( )

Definição: FRAME DGM (JOINT J);

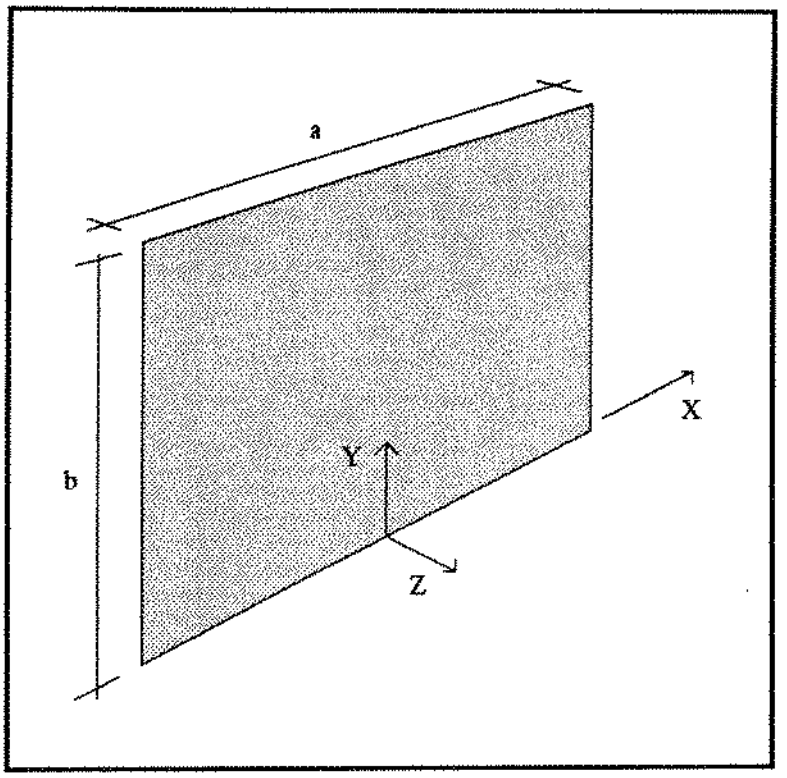


Figura 3.6: *Definição da primitiva* square.

Descrição: Esta função tem como objetivo calcular a posição e orientação do elemento terminal do robô (garra, ou ferramenta caso estejam presentes) através do modelo geométrico direto do robô[+garra[+ferramenta]]. Os valores de junta levados em consideração no cálculo são fornecidos através do parâmetro **J**.

Retorno: retorna o *frame* resultante.

IGM( )

Definição: JOINT IGM (FRAME F);

Descrição: Esta função tem como objetivo calcular os valores das juntas do robô para que o elemento terminal do robô[+garra[+ferramenta]] atinja o referencial definido por **F**.

Retorno: retorna os valores das variáveis de junta resultante ou um JOINT *OUTOFLIM* caso o ponto esteja fora do volume de trabalho do robô..

distance( )

Definição: unsigned distance (POINT p1, POINT p2);

Descrição: Esta função tem como objetivo calcular a distância entre dois pontos no espaço cartesiano.

Retorno: retorna a distância entre os pontos.

FtoM( )

Definição: M4X4 FtoM (FRAME F);

Descrição: Esta função tem como objetivo calcular a matriz de transformação homogênea definida por **F**. A convenção adotada para os ângulos de Euler [7, 14] foi ZXZ. Se

$$F = \begin{bmatrix} x \\ y \\ z \\ \alpha \\ \beta \\ \gamma \end{bmatrix} \tag{3.1}$$

Então

$$M = \begin{bmatrix} -s\alpha c\beta s\gamma + c\alpha c\gamma & -s\alpha c\beta c\gamma - c\alpha s\gamma & s\alpha s\beta & x \\ c\alpha c\beta s\gamma + s\alpha c\gamma & c\alpha c\beta c\gamma - s\alpha s\gamma & -c\alpha s\beta & y \\ s\beta s\gamma & s\beta c\gamma & c\beta & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.2}$$

Retorno: retorna a matriz resultante.

mulMM( )

Definição: M4X4 mulMM (M4X4 M1, M4X4 M2);

Descrição: Esta função tem como objetivo calcular a matriz de transformação homogênea resultante do produto *M1*M2*.

Retorno: retorna a matriz resultante.

transpnt( )

Definição: POINT transpnt (M4X4 M, POINT P);

Descrição: Esta função tem como objetivo realizar a transformação homogênea definida por **M** sobre o ponto **P**.

Retorno: retorna o ponto resultante.

transshp( )

Definição: SHAPE transshp (M4X4 M, SHAPE S);

Descrição: Esta função tem como objetivo realizar a transformação homogênea definida por **M** sobre o *shape* **S**.

Retorno: retorna o *shape* resultante.

transobj( )

Definição: OBJECT transobj (M4X4 M, OBJECT O);

Descrição: Esta função tem como objetivo realizar a transformação homogênea definida por **M** sobre o objeto **O**.

Retorno: retorna o objeto resultante.

transstr( )

Definição: OBJECT transstr (M4X4 M, STRUCTURE S);

Descrição: Esta função tem como objetivo realizar a transformação homogênea definida por **M** sobre a estrutura **S**.

Retorno: retorna a estrutura resultante.

MtoF( )

Definição: FRAME MtoF (M4X4 M);

Descrição: Esta função tem como objetivo calcular um *frame* que represente a transformação homogênea definida por **M**. A conversão **M** em **F** requer que a os ângulos de Euler (α,β,γ) sejam obtidos a partir da matriz de rotação formada pelas três primeiras linhas e três primeiras colunas de **M**. A convenção adotada para os ângulos de Euler foi ZXZ. No caso em que $\beta = 0$ ou $\beta = \pi rad$, apenas a soma dos ângulos α e γ pode ser obtida. Convencionou-se então adotar, neste caso, $\gamma = 0$. Se

$$M = \begin{bmatrix} n_x & s_x & a_x & x \\ n_y & s_y & a_y & y \\ n_z & s_z & a_z & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.3}$$

Então

$$F = \begin{bmatrix} x \\ y \\ z \\ \alpha \\ \beta \\ \gamma \end{bmatrix} \tag{3.4}$$

Onde, se $a_x \neq 0$ e $a_y \neq 0$, tem-se

$$\alpha = Atan2(a_x, -a_y) \tag{3.5}$$

$$\beta = Atan2(-c\alpha s_x - s\alpha s_y, c\alpha n_x + s\alpha n_y) \tag{3.6}$$

$$\gamma = Atan2(s\alpha a_x - c\alpha a_y, a_z) \tag{3.7}$$

Ou, se $a_x = a_y = 0$ e $a_z > 0$, tem-se

$$\alpha = Atan2(n_y, s_y) \tag{3.8}$$

$$\beta = 0 \tag{3.9}$$

$$\gamma = 0 \tag{3.10}$$

Ou, se $a_x = a_y = 0$ e $a_z < 0$, tem-se

$$\alpha = Atan2(n_y, -s_y) \tag{3.11}$$

$$\beta = \pi \tag{3.12}$$

$$\gamma = 0 \tag{3.13}$$

Retorno: retorna o *frame* resultante.

3.5.3 Funções Gráficas

São funções relacionadas à simulação do ambiente de trabalho e dos movimentos do robô.

drawshp( )

Definição: int drawshp (SHAPE s, POINT p, float a, float b, float c, int seq);

Descrição: Esta função tem como objetivo mostrar na tela gráfica a primitiva **s** no ponto **p** girada, em relação à sua origem, dos ângulos (a,b,c) (em rad). A sequência de rotação será fornecida pelo parâmetro **seq**. Ver função **obj( )**.

Retorno: retorna zero se os parâmetros são válidos e um código de erro em caso contrário.

drawobj( )

Definição: void drawobj (OBJECT o);

Descrição: Esta função tem como objetivo mostrar na tela gráfica o objeto **o**.

Retorno: retorna void.

drawrobot( )

Definição: void drawrobot (void);

Descrição: Esta função tem como objetivo mostrar na tela gráfica a posição atual do robô.

Retorno: retorna void.

drawstr( )

Definição: void drawstr (STRUCTURE s);

Descrição: Esta função tem como objetivo mostrar na tela gráfica a estrutura **s**.

Retorno: retorna void

end_gr( )

Definição: void end_gr (void);

Descrição: Esta função tem como objetivo encerrar o modo gráfico inicializado através da função **start_gr( )**, retornando ao modo texto.

Retorno: retorna void.

virtualxy( )

Definição: VPOINT virtualxy (POINT p);

Descrição: Esta função tem como objetivo determinar o ponto (x,y) em coordenadas da tela que representa a projeção ortogonal do ponto **p** no espaço cartesiano em um plano perpendicular ao vetor definido pela posição do observador e pelo ponto para onde ele olha.

Retorno: retorna o ponto tipo VPOINT resultante.

start_gr( )

Definição: void start_gr (void);

Descrição: Esta função tem como objetivo inicializar o modo gráfico.

Retorno: retorna void.

3.5.4 Funções de Movimentação

São funções relacionadas ao movimento do robô, dos objetos e estruturas no interior do ambiente de trabalho.

approach( )

Definição: int approach (POINT p,unsigned d,int dir);

Descrição: Esta função tem como finalidade fazer com que o robô se aproxime do ponto **p** da distância em milímetros especificada no parâmetro **d** segundo a orientação definida por **dir** em relação ao referencial inercial.

O parâmetro **dir** pode ter os seguintes valores:

1. DX : Sentido positivo do eixo X;
2. DY : Sentido positivo do eixo Y;
3. DZ : Sentido positivo do eixo Z;
4. -DX : Sentido negativo do eixo X;
5. -DY : Sentido negativo do eixo Y;
6. -DZ : Sentido negativo do eixo Z;

Retorno: retorna zero se a operação foi bem sucedida ou um código de erro em caso contrário.

closegripper( )

Definição: int closegripper (unsigned d);

Descrição: Esta função tem como objetivo o controle do fechamento de uma garra associada ao robô. Ao terminar sua execução, a garra do robô terá atingido a abertura especificada em milímetros pelo parâmetro **d**.

Retorno: retorna zero se a operação foi bem sucedida ou o valor em milímetros do fechamento obtido. Retorna -1 se a garra não estiver presente.

depart( )

Definição: int depart (unsigned d, int dir);

Descrição: Esta função tem como finalidade fazer com que o robô se afaste (do ponto corrente) de uma distância (em mm) especificada no parâmetro **d** segundo a orientação definida por **dir** em relação ao referencial inercial. Ver função **approach( )**.

Retorno: retorna zero se a operação foi bem sucedida ou um código de erro em caso contrário.

dropstr( )

Definição: int dropstr (void);

Descrição: Esta função tem como objetivo fazer com que a estrutura acoplada ao elemento terminal do robô [+garra[+ferramenta]] através da função **takestr( )** seja liberada. A estrutura será mantida na posição e orientação em que foi liberada.

Retorno: retorna -1 se não existir uma estrutura acoplada. Retorna zero em caso contrário.

followj( )

Definição: int followj (JPATH j, int sentido);

Descrição: Esta função tem como objetivo fazer com que o robô siga uma determinada trajetória definida pelo parâmetro **j** no espaço de juntas. O parâmetro **sentido** indica o sentido a ser seguido:

1. FORWARD : o movimento se dará na sequência em que os pontos foram definidos no caminho **j**;
2. BACKWARD : o movimento se dará na sequência inversa em que os pontos foram definidos no caminho **j**.

Retorno: retorna zero se os parâmetros são válidos e um código de erro em caso contrário.

followc( )

Definição: int followc (CPATH c, int sentido);

Descrição: Esta função tem como objetivo fazer com que o robô siga uma determinada trajetória definida pelo parâmetro **c** no espaço cartesiano. O parâmetro **sentido** indica o sentido a ser seguido. Ver função **followj( )**.

Retorno: retorna zero se os parâmetros são válidos e um código de erro em caso contrário.

home( )

Definição: int home (void);

Descrição: Esta função tem como objetivo fazer com que o robô se posicione no *home* definido pela função **defhome( )**.

Retorno: retorna zero caso o ponto *home* tenha sido definido anteriormente e um código de erro em caso contrário.

movejoint( )

Definição: int movejoint (int n, float val);

Descrição: Esta função tem como objetivo mover a junta especificada no parâmetro **n** para a coordenada de junta especificada no parâmetro **val**. O parâmetro **val** deve estar dentro dos limites **[min,max]** definidos na função **denavit( )**.

Retorno: retorna zero se os parâmetros são válidos e um código de erro em caso contrário.

moveobj( )

Definição: OBJECT moveobj (OBJECT o, POINT p);

Descrição: Esta função tem como objetivo atualizar as coordenadas espaciais (x,y,z) do objeto **o**.

Retorno: retorna o objeto **o** com suas coordenadas espaciais atualizadas ou um *OBJECT nulo* caso os parâmetros não sejam válidos.

moveshape

Definição: SHAPE moveshape (SHAPE s, POINT p);

Descrição: Esta função tem como objetivo atualizar as coordenadas espaciais (x,y,z) da primitiva **s**. Uma nova primitiva será gerada a partir da translação da origem da primitiva **s** para o ponto **p**.

Retorno: retorna a primitiva **s** com suas coordenadas espaciais atualizadas ou um *SHAPE nulo* caso os parâmetros não sejam válidos.

movestr( )

Definição: STRUCTURE movestr (STRUCTURE s, POINT p);

Descrição: Esta função tem como objetivo atualizar as coordenadas espaciais (x,y,z) da estrutura **s**. Uma nova estrutura será gerada a partir da translação da origem da estrutura **s** para o ponto **p**.

Retorno: retorna a estrutura **s** com suas coordenadas espaciais atualizadas ou uma *STRUCTURE nula* caso os parâmetros não sejam válidos.

moveto( )

Definição: int moveto (FRAME p);

Descrição: Esta função tem como objetivo fazer com que o robô se mova da posição atual para a posição dada pelo parâmetro **p**. A interpolação utilizada será a fornecida na última chamada à função **interpol( )**.

Retorno: retorna zero caso a operação seja bem sucedida ou um código de erro em caso contrário.

opengripper( )

Definição: int opengripper (unsigned d);

Descrição: Esta função tem como objetivo o controle da abertura de uma garra associada ao robô. Ao terminar sua execução, a garra do robô terá atingido a abertura especificada em milímetros pelo parâmetro **d**. Se **d** = *TOTAL*, haverá uma abertura total da garra.

Retorno: retorna zero se a operação foi bem sucedida ou o valor em milímetros da abertura obtida. Se a garra não estiver presente, retorna -1.

orientation( )

Definição: int orientation (int cod);

Descrição: Esta função tem como objetivo determinar o tipo de orientação que o elemento terminal do robô deverá seguir. O parâmetro **cod** determina se o movimento será livre ou fixo. Se o movimento for livre, a orientação do elemento terminal do robô será dada pelas funções de movimento. Se o movimento for fixo, a orientação corrente do elemento terminal será mantida. O parâmetro **cod** pode ter os seguintes valores:

1. FREE : movimento livre do elemento terminal;
2. FIXED : orientação fixa do elemento terminal.

Retorno: retorna zero se os parâmetros forem válidos ou um código de erro em caso contrário.

rotobj( )

Definição: OBJECT rotobj (OBJECT o, float a, float b, float c, int seq);

Descrição: Esta função tem como objetivo girar a primitiva do objeto **o**, em torno da origem, dos ângulos (a,b,c) dados em radianos.O parâmetro **seq** define a sequência de rotação em torno dos eixos coordenados. Ver função **drawshp( )**.

Retorno: retorna o objeto **o** com sua primitiva atualizada ou um *OBJECT nulo* caso os parâmetros não sejam válidos.

rotshape( )

Definição: SHAPE rotshape (SHAPE s, float a, float b,float c, int seq);

Descrição: Esta função tem como objetivo girar a primitiva **s**, em torno da origem, dos ângulos (a,b,c) dados em radianos.O parâmetro **seq** define a sequência de rotação em torno dos eixos coordenados da mesma forma que na função **rotobj( )**.

Retorno: retorna a primitiva **s** atualizada ou um *SHAPE nulo* caso os parâmetros não sejam válidos.

rotstr( )

Definição: STRUCTURE rotstr (STRUCTURE s, float a,float b, float c, int seq);

Descrição: Esta função tem como objetivo girar todas as primitivas da estrutura **s**, em torno da origem, dos ângulos (a,b,c) dados em radianos.O parâmetro **seq** define a sequência de rotação em torno dos eixos coordenados da mesma forma que na função **rotobj( )**.

Retorno: retorna a estrutura **s** com suas primitivas atualizadas ou uma *STRUCTURE nula* caso os parâmetros não sejam válidos.

rotxyz( )

Definição: POINT rotxyz (POINT p1, POINT p2, float a, float b, float c);

Descrição: Esta função tem como objetivo girar o ponto **p1** em torno do ponto **p2** (ambos dados em milímetros) dos ângulos (a,b,c) dados em radianos.

Retorno: retorna o ponto tipo POINT resultante.

speed( )

Definição: int speed (unsigned vel);

Descrição: Esta função tem como objetivo determinar o percentual da velocidade máxima para os movimentos do robô. O parâmetro **vel** pode conter valores inteiros no intervalo]0,100].

Retorno: retorna zero se o parâmetro **vel** for válido ou um código de erro em caso contrário.

takestr( )

Definição: int takestr (STRUCTURE S);

Descrição: Esta função tem como objetivo fazer com que o elemento terminal do robô [+garra[+ferramenta]] se desloque da posição corrente até a posição indicada pelo *frame de pega* da estrutura **S** e *acople* o *frame* desta estrutura. Ou seja, a estrutura ficará rigidamente ligada ao robô. Qualquer movimento realizado pelo robô afetará também esta estrutura realizando uma simulação de pega de um objeto. Em uma situação real, esta função deve ser sempre seguida

pela função **closegripper( )** ou pela ativação de uma saída (digital ou analógica) que atue sobre o dispositivo responsável pela pega.

Retorno: retorna -1 se já existe uma estrutura acoplada. Retorna zero se a operação for bem sucedida e um código de erro em caso contrário.

3.5.5 Funções de Gerais

São as que não se enquadram nos tipos anteriores.

defmsg( )

Definição: int defmsg (int cod, char far *msg);

Descrição: Esta função tem como objetivo a definição de uma mensagem de erro. Toda vez que houver uma chamada à função **writemsg(cod)**, será mostrada a mensagem de erro fornecida no parâmetro **msg**.

Retorno: retorna zero caso seja uma nova mensagem de erro e 1 caso seja uma redefinição de mensagem.

getgripper( )

Definição: int getgripper (void);

Descrição: Esta função tem como objetivo obter o valor da abertura da garra do robô.

Retorno: retorna o valor em milímetros da abertura da garra ou -1 se a garra não estiver presente.

getframe( )

Definição: FRAME getframe (void);

Descrição: Esta função tem como objetivo obter a posição (x,y,z) e orientação (a,b,c) correntes do *end-effector*.

Retorno: retorna uma variável tipo FRAME contendo a posição e orientação corrente do *end-effector*.

loadrobot( )

Definição: int loadrobot (int type);

Descrição: Esta função tem como objetivo ler os dados referentes a um determinado tipo de robô previamente gravados em disco. Existe a intenção de se criar uma biblioteca contendo arquivos com dados referentes a vários tipos de robôs existentes. À medida que estes arquivos forem sendo criados, o parâmetro **type** tem seu limite incrementado. Ou seja, **type** pode variar de 1 (que representa o telemanipulador KRAFT) até o número de robôs contidos nesta biblioteca.

Retorno: retorna 0 se os arquivos referentes ao robô estiverem presentes ou um código de erro em caso contrário.

loadstr( )

Definição: STRUCTURE loadstr (char huge *arquivo);

Descrição: Esta função tem como objetivo ler uma estrutura previamente gravada em um arquivo em disco cujo nome é fornecido pelo parâmetro **arquivo**.

Retorno: retorna a estrutura lida se o arquivo for válido ou uma *STRUCTURE nula* em caso contrário.

hmatrix( )

Definição: M4X4 hmatrix (JOINT j, int i);

Descrição: Esta função tem como objetivo construir a matriz homogênea de transformação do referencial inercial até o referencial conectado ao link **i**, considerando os valores de junta fornecidos no parâmetro **j**.

Retorno: retorna a matriz homogênea se os parâmetros forem válidos ou uma matriz nula em caso contrário.

savestr( )

Definição: int savestr (STRUCTURE s, char huge *arquivo);

Descrição: Esta função tem como objetivo gravar a estrutura **s** no arquivo em disco cujo nome é fornecido pelo parâmetro **arquivo**.

Retorno: retorna zero se a operação for bem sucedida ou um código de erro em caso contrário.

seterror( )

Definição: int seterror (int cod);

Descrição: Esta função tem como objetivo determinar o procedimento a ser adotado quando uma condição de erro for encontrada. O parâmetro **cod** pode assumir os seguintes valores:

1. NONE : nenhuma ação será tomada. Neste modo, o usuário deverá sempre verificar os valores de retorno das funções chamadas para se certificar de que nenhuma condição de erro foi detectada.
2. WRITE : quando um erro for detectado, uma mensagem será impressa e o programa continuará a ser executado.
3. ABORT: quando um erro for detectado, uma mensagem será impressa e o programa será encerrado.

Retorno: retorna zero se o parâmetro é válido ou um código de erro em caso contário.

writemsg( )

Definição: char writemsg (int cod);

Descrição: Esta função tem como objetivo imprimir na tela de textos a mensagem de erro pré-definida pela função **defmsg(cod,msg)**. A mensagem será impressa e o controle será devolvido ao programa depois que o usuário pressionar uma tecla qualquer.

Retorno: retorna zero se a mensagem não estiver definida ou o código da tecla pressionada em caso contrário.

3.6 Adição de Funções à Biblioteca UC.LIB

A principal vantagem de se ter um sistema baseado em bibliotecas reside no fato de que novas funções podem ser criadas pelo usuário. Para acrescentar uma nova função à biblioteca UC.LIB, basta que se crie a referida função no editor do UCI sem a função **main( )** característica dos programas em C e ULTRA C. Depois, esta função deve ser compilada para *.OBJ* e acrescentada à biblioteca *UC.LIB*. Para acrescentá-la à biblioteca, deve-se utilizar o programa *TLIB.EXE* contido no Turbo C ou qualquer outro gerenciador de bibliotecas.

Suponha que se deseje criar as funções **func1( ), func2( ) e func3( )**. Depois de se escrever as funções, em um arquivo *FUNC.UC*, por exemplo, deve-se compilar este arquivo, gerando o arquivo *FUNC.OBJ*. Para adicionar as novas funções à biblioteca *UC.LIB* utilizando-se o *TLIB.EXE* basta digitar na linha de comando do DOS: *TLIB UC.LIB +FUNC.OBJ*. Depois desse passo, as funções **func1( ), func2( ) e func3( )** podem ser utilizadas em qualquer programa escrito em ULTRA C.

O apêndice C mostra dois exemplos de programação utilizando a linguagem ULTRA C.

Neste capítulo, algumas características básicas da linguagem ULTRA C foram apresentadas. Optou-se por não se ater a detalhes de implementação de cada uma das funções. Em vez disso, tentou-se fornecer uma idéia global da linguagem, sua tilização e potencialidades. Algumas funções apresentadas neste capítulo estão intimamente relacionadas com a modelagem gráfica do robô e seu ambiente de trabalho. O próximo capítulo trata exatamente da utilização do modelador gráfico incorporado no UCI.

Capítulo 4

O Modelador Gráfico

4.1 Introdução

Todo sistema de programação *off-line* de robôs por linguagem textual deve permitir ao programador modelar graficamente o robô e seu ambiente de trabalho. A modelagem permite a simulação gráfica da execução do programa como uma forma de validação. O ambiente integrado UCI possui um modelador gráfico que permite, através de primitivas básicas, gerar modelos que representem o robô e objetos inseridos em seu volume de trabalho.

Os modeladores gráficos mais simples possuem, em geral, apenas uma primitiva geométrica básica, geralmente o paralelepípedo [1, 23]. A modelagem de um ambiente com certo grau de complexidade através de paralelepípedos não pode ser considerada eficiente. Serve apenas para dar ao programador uma noção do movimento que será realizado pelo robô. Não pode ser aplicada na implementação de algoritmos precisos de teste de colisão em tempo real pois o número de primitivas necessárias para se modelar um sólido simples é muito grande. Supondo-se que este sólido é um simples cone, pode-se ter idéia da quantidade de paralelepípedos necessários para modelá-lo com certo grau de realismo.

Apenas a título de ilustração, apresenta-se na figura 4.1 uma operação simulada de aproximação do robô Manutec r3 montado sobre uma plataforma móvel, em relação a um dos painéis de operação do *template manifold*.

O modelador gráfico embutido no UCI possui as seguintes primitivas básicas (*shapes*):

1. SQUARE: utilizado para modelar planos e objetos nos quais a espessura não tenha muita importância no decorrer da tarefa. Tem a vantagem de possuir apenas dois parâmetros

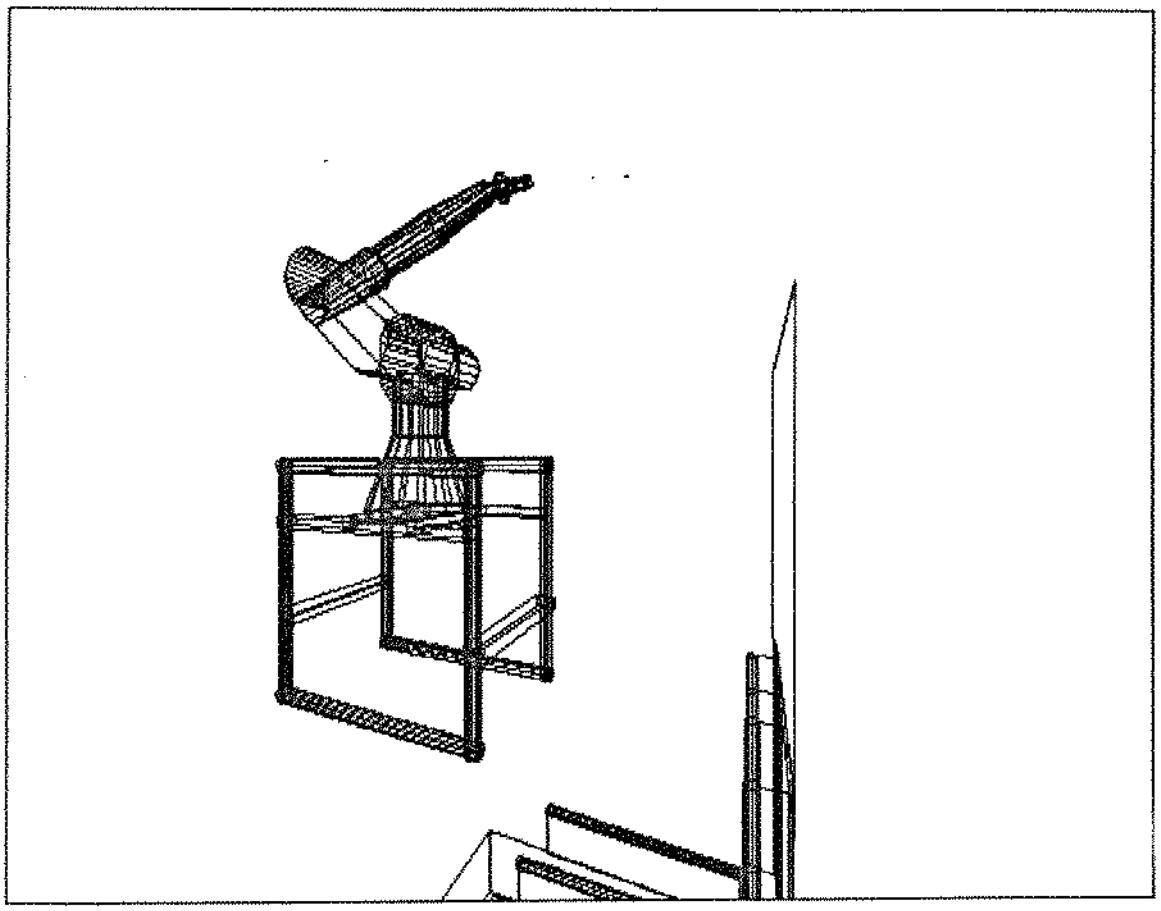


Figura 4.1: *Simulação de aproximação do painel de operação com o robô Manutec r3 montado sobre plataforma móvel.*

para sua definição e é representado com apenas quatro pontos;

2. PYRAMID: apesar do nome, o *shape* pyramid não serve apenas para se modelar pirâmides. As formas das bases e a distância entre elas podem ser alteradas. Na verdade,o triângulo, o retângulo, trapézio, o cubo, o paralelepípedo, a pirâmide e o tronco de pirâmide são casos particulares da primitiva PYRAMID. Possue cinco parâmetros e é representada com oito pontos;

3. CONE: da mesma forma que a primitiva PYRAMID, a primitiva CONE assume diversos papéis durante a modelagem tanto do robô quanto de seu ambiente de trabalho. Devido à flexibilidade de seus parâmetros, esta primitiva resume as características do círculo, do disco, do cilindro, do cone e do tronco de cone. Na verdade, a primitiva CONE pode representar mais que isto. O fato de que o número de arestas verticais pode variar de 3 a 16, permite a esta primitiva representar os papéis do triângulo, retângulo, pentágono, hexágono, etc.(polígonos regulares com número de lados menor que 17), além dos sólidos

gerados pela combinação com os demais parâmetros. Possue quatro parâmetros e o número de pontos pode variar de 6 a 32;

4. DOUBLEDISC: é uma primitiva pouco comum em modeladores gráficos. Representa dois discos de raios diferentes com uma determinada distância entre seus centros. Pode ser usado para modelar grades de proteção a sistemas com polias, rodas de raios distintos ou esteiras rolantes. Pode representar o círculo, o disco e o próprio *duplodisco*. Possue quatro parâmetros e sempre será representado por 60 pontos.

Pode-se perceber que simulações bastante realistas podem ser obtidas com estas primitivas básicas sem que uma grande quantidade de memória seja requerida. Detalhes sobre as primitivas, como definí-las e utilizá-las serão vistos nas próximas seções.

4.2 Regras de Modelagem

Em ULTRA C, cada sólido a ser modelado, por mais complexo que seja, deve ser decomposto em suas primitivas básicas. Às primitivas acrescenta-se informações de posição e orientação (*FRAME*) e se obtém uma entidade chamada objeto. Os objetos podem ser agrupados em entidades chamadas estruturas. Cada estrutura pode conter até *STRSIZE* objetos e cada objeto, uma primitiva. Ou seja, pode-se afirmar que:

- *Shapes* são primitivas geométricas básicas que possuem apenas informação sobre a forma do sólido que estão modelando. Não possuem qualquer informação sobre a posição e a orientação em relação ao referencial inercial;
- *Objects* são entidades geométricas formadas pela adição de um referencial a um *shape*. Simbolicamente pode-se dizer que:
  $OBJECT = SHAPE + FRAME$
  Os objetos em sí já são suficientes para modelar um sólido simples como o paralelepípedo, o cilindro e o cone;
- *Structures* são entidades geométricas formadas por um conjunto de objetos posicionados em relação a um referencial. Além disso, toda estrutura possui um referencial chamado *referencial de pega* ou *ponto de pega* que nada mais é que um *FRAME* associado à posição e orientação com que o robô terá que se posicionar caso seja necessária a manipulação desta estrutura. Ou seja, simbolicamente pode-se dizer que:
  $STRUCTURE = (OBJECT\ 1 + OBJECT\ 2 + ... + OBJECT\ N) + FRAME + PEGA$
  onde N pode variar de 1 a *STRSIZE*.

Um detalhe deve sempre ser observado pelo usuário: embora existam formas que podem ser representadas por mais de uma primitiva, o usuário deve sempre optar por usar os *shapes* mais simples. Por exemplo, se um determinado sólido pode ser representado tanto por um *shape* CONE quanto por um *shape* PYRAMID, deve-se optar pelo *shape* PYRAMID por questões de velocidade durante a simulação.

A criação de *Shapes*, *Objects* e *Structures* pode ser realizada passo-a-passo, através das funções *square()*, *pyramid()*, *conc()*, *doubledisc()*, *obj()*, *str()*, *etc.* Esta forma de modelagem é tediosa e demorada. Com o objetivo de automatizar a modelagem de ambientes complexos, desenvolveu-se um modelador gráfico que tira do usuário a tarefa de manipulação destas funções, tornando o processo de modelagem mais amigável.

4.3 Chamadas ao Modelador Gráfico

O usuário pode ter acesso ao modelador gráfico de duas maneiras:

1. Utilizando a opção *Modelar* do menu principal do UCI;
2. Executando o programa *MODEL.EXE* a partir do *prompt* do sistema operacional.

Ao realizar uma chamada ao modelador gráfico, a tela ilustrada na figura 4.2 será apresentada.

No lado esquerdo da tela se encontra a área de trabalho do modelador gráfico. No lado direito encontra-se o menu que dá acesso a algumas funções do modelador. A utilização destas funções será apresentada na próxima seção.

4.4 Utilização do Modelador Gráfico

Deve-se encarar o processo de modelagem da seguinte maneira: inicia-se com a análise, partição ou subdivisão de um sólido complexo em seus componentes básicos mais simples que possam ser representados pelas primitivas básicas. Em seguida, determina-se onde deverá estar fixado o referencial do sólido modelado. Criam-se então cada um dos objetos (primitivas dotadas de uma posição e orientação em relação ao referencial do sólido). Todos os objetos criados serão considerados rigidamente ligados ao referencial do sólido. Estes passos nada mais representam senão a construção de uma estrutura com *frame* nulo. Se esta estrutura puder ser manipulada

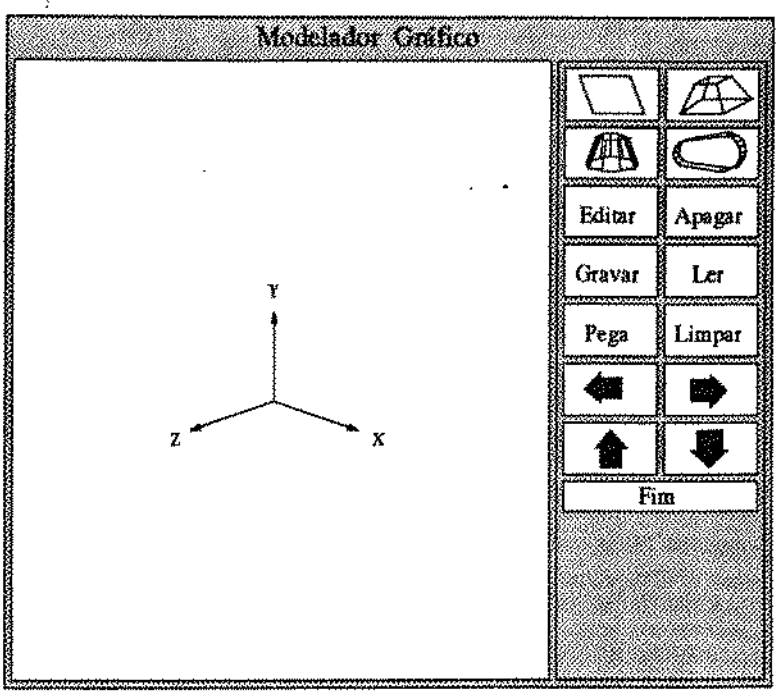


Figura 4.2: *O modelador gráfico MODEL.EXE*

pelo robô, deve-se atribuir a ela um referencial de *pega*. Esta estrutura pode agora ser gravada em disco e lida posteriormente por um programa ULTRA C, poupando ao usuário, a tarefa de redefinição dos objetos. Repete-se o processo para todos os sólidos a serem modelados. As estruturas lidas por um programa podem então ser dotadas, através das funções *movstr()* e *rotstr()*, de uma posição e uma orientação em relação ao referencial do robô.

O modelador gráfico *MODEL* suporta todos os passos descritos e permite ao usuário maior flexibilidade durante a edição de um modelo gráfico. Cada retângulo do lado direito da tela possui uma função específica. Ao se realizar uma chamada ao modelador, a primeira opção do menu se encontra ativa. As setas do teclado mudam a opção ativa. A tecla *ENTER* efetiva a opção ativa. As quatro primeiras opções (de cima para baixo) correspondem às primitivas geométricas descritas no capítulo anterior. As demais opções dão suporte à editoração, leitura e gravação de arquivos e alteração do ponto de vista do observador em relação ao referencial do modelo.

Mostra-se a seguir cada uma destas opções.

Sólidos Tipo Square

A escolha desta opção permite a criação de um objeto com primitiva retângulo no espaço tridimensional. Todos os parâmetros requeridos para a definição do objeto são fornecidos na parte inferior do menu. Estes parâmetros são (ver figura 3.6):

1. **A** : largura do retângulo (em milímetros);
2. **B** : Comprimento do retângulo (em milímetros);
3. **X** : Coordenada X do centro do retângulo (em milímetros);
4. **Y** : Coordenada Y da base do retângulo (em milímetros);
5. **Z** : Coordenada Z do centro do retângulo (em milímetros);
6. **a** : Ângulo de rotação em torno do eixo X (em graus);
7. **b** : Ângulo de rotação em torno do eixo Y (em graus);
8. **c** : Ângulo de rotação em torno do eixo Z (em graus);
9. **seq** : Sequência de rotação que pode ser XYZ, XZY, YXZ, YZX, ZXY ou ZYX.

Sólidos Tipo Pyramid

A escolha desta opção permite a criação de um objeto com primitiva tronco de pirâmide no espaço tridimensional. Todos os parâmetros requeridos para a definição do objeto são fornecidos na parte inferior do menu. Estes parâmetros são (ver figura 3.5):

1. **A1** : Largura da base inferior (em milímetros);
2. **B1** : Comprimento da base inferior (em milímetros);
3. **A2** : Largura da base superior (em milímetros);
4. **B2** : Comprimento da base superior (em milímetros);
5. **H** : Distância entre as bases (em milímetros);
6. **X** : Coordenada X do centro da base inferior (em milímetros);
7. **Y** : Coordenada Y do centro da base inferior (em milímetros);
8. **Z** : Coordenada Z do centro da base inferior (em milímetros);
9. **a** : Ângulo de rotação em torno do eixo X (em graus);

10. **b** : Ângulo de rotação em torno do eixo Y (em graus);
11. **c** : Ângulo de rotação em torno do eixo Z (em graus);
12. **seq** : Sequência de rotação que pode ser XYZ, XZY, YXZ, YZX, ZXY ou ZYX.

Sólidos Tipo Cone

A escolha desta opção permite a criação de um objeto com primitiva tipo tronco de cone no espaço tridimensional. Todos os parâmetros requeridos para a definição do objeto são fornecidos na parte inferior do menu. Estes parâmetros são (ver figura 3.1):

1. **R1** : Raio da base inferior (em milímetros);
2. **R2** : Raio da base superior (em milímetros);
3. **L** : Distância entre as bases (em milímetros);
4. **N** : Número de arestas. Deve estar no intervalo]3,16];
5. **X** : Coordenada X do centro da base inferior (em milímetros);
6. **Y** : Coordenada Y do centro da base inferior (em milímetros);
7. **Z** : Coordenada Z do centro da base inferior (em milímetros);
8. **a** : Ângulo de rotação em torno do eixo X (em graus);
9. **b** : Ângulo de rotação em torno do eixo Y (em graus);
10. **c** : Ângulo de rotação em torno do eixo Z (em graus);
11. **seq** : Sequência de rotação que pode ser XYZ, XZY, YXZ, YZX, ZXY ou ZYX.

Sólidos Tipo Doubledisc

A escolha desta opção permite a criação de um objeto com primitiva tipo *duplodisco* no espaço tridimensional. Todos os parâmetros requeridos para a definição do objeto são fornecidos na parte inferior do menu. Estes parâmetros são (ver figura 3.4):

1. **R1** : Raio do disco inferior (em milímetros);
2. **R2** : Raio do disco superior (em milímetros);
3. **A** : Largura dos discos (em milímetros);

4. **L** : Distância entre os centros (em milímetros). Note que

$$L \geq | R1 - R2 | \tag{4.1}$$

5. **X** : Coordenada X do centro do disco inferior (em milímetros);
6. **Y** : Coordenada Y do centro do disco inferior (em milímetros);
7. **Z** : Coordenada Z do centro do disco inferior (em milímetros);
8. **a** : Ângulo de rotação em torno do eixo X (em graus);
9. **b** : Ângulo de rotação em torno do eixo Y (em graus);
10. **c** : Ângulo de rotação em torno do eixo Z (em graus);
11. **seq** : Sequência de rotação que pode ser XYZ, XZY, YXZ, YZX, ZXY ou ZYX.

A Opção Editar

Esta opção permite ao usuário alterar os parâmetros de qualquer objeto definido anteriormente. O usuário escolhe o objeto a ser editado através das teclas **F5** e **F6**. Quando uma destas teclas é pressionada, ocorre uma varredura dos objetos por ordem direta (**F5**) ou inversa (**F6**) de criação e o objeto escolhido é mostrado na tela em cor preta. Os demais objetos são mostrados em cor branca.

Durante o processo de edição dos parâmetros, o usuário tem a opção de manter ou alterar cada parâmetro dos objetos. Para mantê-los basta pressionar *ENTER* sobre o parâmetro, para alterá-lo, deve digitar o novo valor sobre o valor antigo e pressionar *ENTER*. Para se desistir de efetuar uma edição, basta pressionar a tecla *ESC*.

A Opção Apagar

Esta opção permite ao usuário apagar qualquer objeto definido anteriormente. O usuário escolhe o objeto a ser apagado utilizando as teclas **F5** e **F6**. O objeto apagado será removido da estrutura.

A Opção Gravar

Nesta opção o usuário pode gravar em um arquivo em disco a estrutura criada. O nome do arquivo deverá ser fornecido na parte inferior do menu. Se o usuário informar um nome

de arquivo válido, a estrutura será gravada e poderá posteriormente ser lida através da função *loadstr()* do ULTRA C.

A Opção Ler

Nesta opção o usuário pode transferir uma estrutura contida em um arquivo em disco, para a tela de edição do *MODEL.EXE*. O nome do arquivo deve ser fornecido na parte inferior do menu.

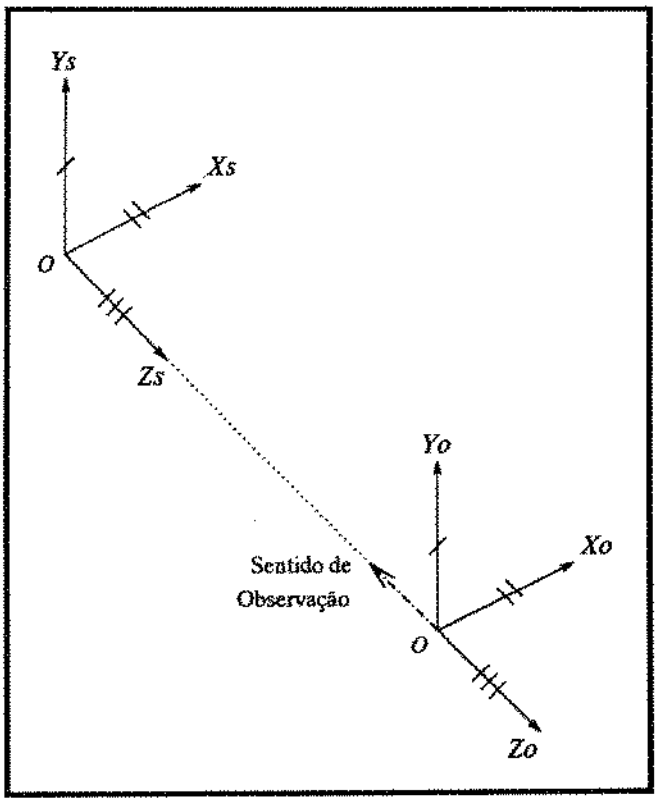


Figura 4.3: *Posição inicial do observador.*

A Opção Pega

Esta opção permite a definição da posição e orientação que o elemento terminal do robô[+garra[+ferramenta]] deve assumir para realizar a manipulação desta estrutura.

A Opção Limpar

Esta opção apaga todos os objetos da estrutura. Pede uma confirmação do usuário para realizar o apagamento.

Mudança de Perspectiva

As outras opções se referem aos movimentos do observador em relação ao referencial da estrutura.

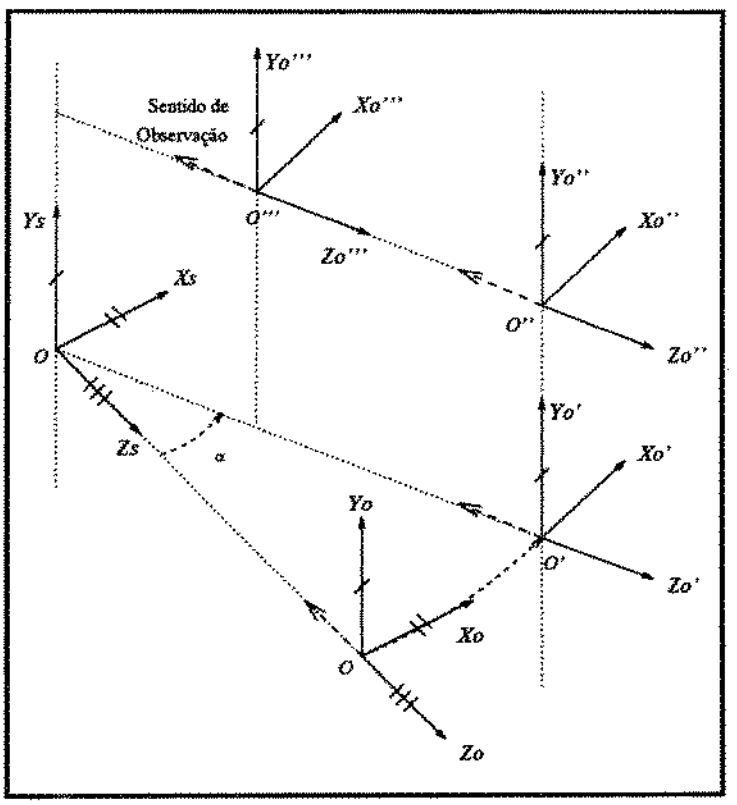


Figura 4.4: *Mudança de perspectiva.*

Inicialmente o referencial do observador é colocado conforme mostra a figura 4.3. Ou seja, o observador está a uma determinada distância do referencial da estrutura, com coordenadas $x = 0$ e $y = 0$ olhando para a origem do referencial da estrutura.

O usuário pode alterar seu ponto de vista em relação à estrutura para obter uma melhor visualização. As setas *para cima* e *para baixo* que aparecem no menu principal, provocam a translação do referencial do observador ao longo de um eixo paralelo ao eixo y (eixo vertical) do referencial da estrutura. As setas *para direita* e *para esquerda* provocam a rotação do referencial

do observador em relação ao referencial da estrutura sobre um círculo contido em um plano paralelo ao plano xz. O raio do círculo pode ser alterado realizando os efeitos *zoom* e *unzoom*. As teclas **F3** e **F4** diminuem e aumentam, respectivamente, o raio deste círculo. A perspectiva original pode ser sempre obtida através do pressionamento da tecla **F9**. Todas as opções de movimentação do observador estão também disponíveis através da utilização das teclas de função **F1, F2, F7** e **F8** em lugar das opções do menu *para esquerda, para direita, para cima* e *para baixo*, respectivamente.

A figura 4.4 mostra uma rotação para a direita (**F2**), seguida de uma translação para cima (**F7**) e de um *zoom* (**F3**).

Neste capítulo apresentou-se o modelador gráfico incorporado ao ambiente UCI e alguns aspectos relacionados ao processo de modelagem. Concluí-se então a apresentação do ambiente integrado UCI. O próximo capítulo é dedicado a uma breve apresentação da implementação realizada para possibilitar a utilização do UCI na programação de tarefas do telemanipulador Kraft. Serão apresentados o sistema Kraft original e as alterações realizadas para tornar possível a implementação e validação do ambiente UCI apresentado.

Capítulo 5

Implementação e Validação

5.1 Introdução

Com o objetivo de dar ao telemanipulador Kraft a possibilidade de ser programado através da linguagem ULTRA C, algumas modificações no sistema original tiveram que ser realizadas. O sistema Kraft, originalmente não possui nenhuma *interface* capaz de enviar ou receber dados de um dispositivo externo pois foi projetado para funcionar como um sistema tipo *master-slave*, ou seja, todo movimento realizado pelo operador nas juntas do *master* é reproduzido pelo *slave*. Neste capítulo mostra-se o sistema original, as modificações realizadas e, ao final, mostra-se como o telemanipulador Kraft pode ser programado pela linguagem ULTRA C. Os apêndices A e B trazem, respectivamente, os parâmetros de Denavit-Hartemberg e as matrizes de transformação homogênea para o manipulador Kraft. O apêndice C traz dois exemplos de programas desenvolvidos para o manipulador Kraft.

5.2 O Sistema Kraft Original

O telemanipulador Kraft é um sistema *master-slave* projetado para a realização de tarefas submarinas ou em ambientes hostis. Através do controle de posição, movimentos realizados no *master* são reproduzidos no *slave*.

Os principais componentes do sistema são:

- *Slave*
- *Master*

- Sistema Eletrônico KMC 9100

Estes componentes são detalhados a seguir.

O Slave

O *slave* é um braço eletro-hidráulico com seis graus de liberdade mais o abrir e fechar do *gripper*. O movimento é realizado por sete atuadores hidráulicos. Cada uma das juntas possui como sensor de posição um potenciômetro de precisão.

O Master

O *master* é geometricamente similar ao *slave*, porém em escala reduzida. Também possui seis graus de liberdade. Potenciômetros em todas as seis juntas fornecem as informações de posição. Um potenciômetro a mais, colocado na pistola, fornece a informação de fechamento e abertura da garra. Chaves adicionais controlam algumas funções especiais do sistema: modo de rotação do punho (*wrist roll*) contínuo ou manual, manutenção da abertura corrente da garra, etc. Um *push button* localizado na base do *master* controla o fornecimento ou corte da pressão hidráulica.

O Sistema Eletrônico KMC 9100

O KMC 9100 (*Kraft Manipulator Controller 9100*) é um sistema baseado em microprocessador concebido para realizar o contrôle e as funções de telemetria necessárias à operação do manipulador. O KMC 9100 se subdivide em três componentes principais:

- *Master System Controller* (MSC) que contém o processador central
- *Hand-Terminal* (HT) que contém teclas para entrada de dados e um visor para *interface* com o usuário.
- *Remote Servo Driver* (RSD) localizado próximo ao *Slave*. O RSD interage diretamente com o *Slave*, enviando-lhe os comandos recebidos do MSC. A comunicação entre o MSC e o RSD pode ser realizada através de par trançado (versão utilizada pela PETROBRÁS), cabo coaxial ou fibra ótica.

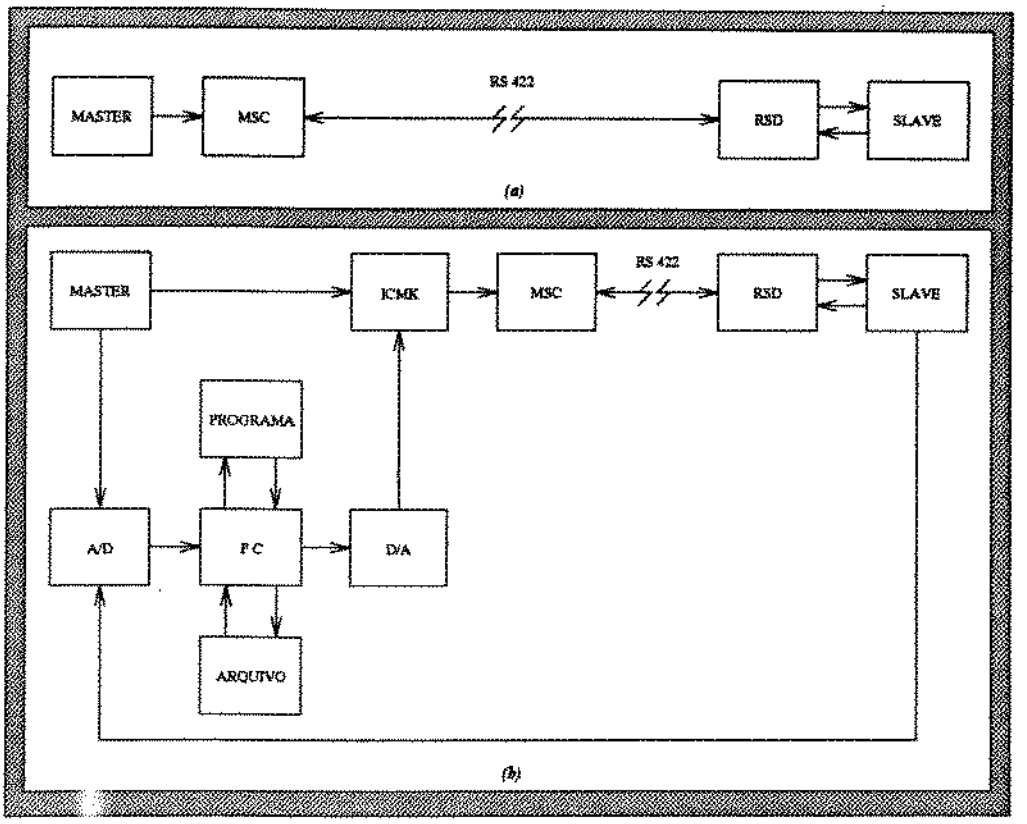


Figura 5.1: (a)*Configuração original.* (b) *Configuração Atual*

O diagrama mostrado na figura 5.1-a ilustra o fluxo de dados no sistema Kraft original.

Observando a figura, nota-se que a informação de posição é gerada no *master*, processada no MSC, transferida via *interface* serial RS 422 até o RSD onde ocorre a transferência para os controladores de junta do *slave*. Os potenciômetros do *slave* geram um sinal de tensão que percorre o caminho inverso, de volta ao MSC. Este retorno de sinal é necessário para testes de restrição mecânica das juntas e estabelecimento da calibração do manipulador [18].

5.3 O Sistema Kraft Alterado

Com o objetivo de tornar o manipulador programável pela linguagem ULTRA C, desenvolveu-se uma *interface* de *hardware* chamada ICMK (Interface para o Controlador do Manipulador Kraft). Esta *interface*, é responsável pela multiplexação dos sinais provenientes do *master* e do PC. Ver figura 5.1-b. O ICMK possui 16 entradas analógicas, 8 saídas analógicas, 16 entradas digitais e 16 saídas digitais. A comunicação com o PC é realizada através de duas placas:

- **CDA 12/08** - conversor digital/analógico de 12 bits e 8 canais;
- **CAD 12/36** - conversor analógico/digital de 12 bits e 16 canais com 16 entradas e 16 saídas digitais.

Maiores informações sobre ambas as placas podem ser obtidas em [21] e [22].

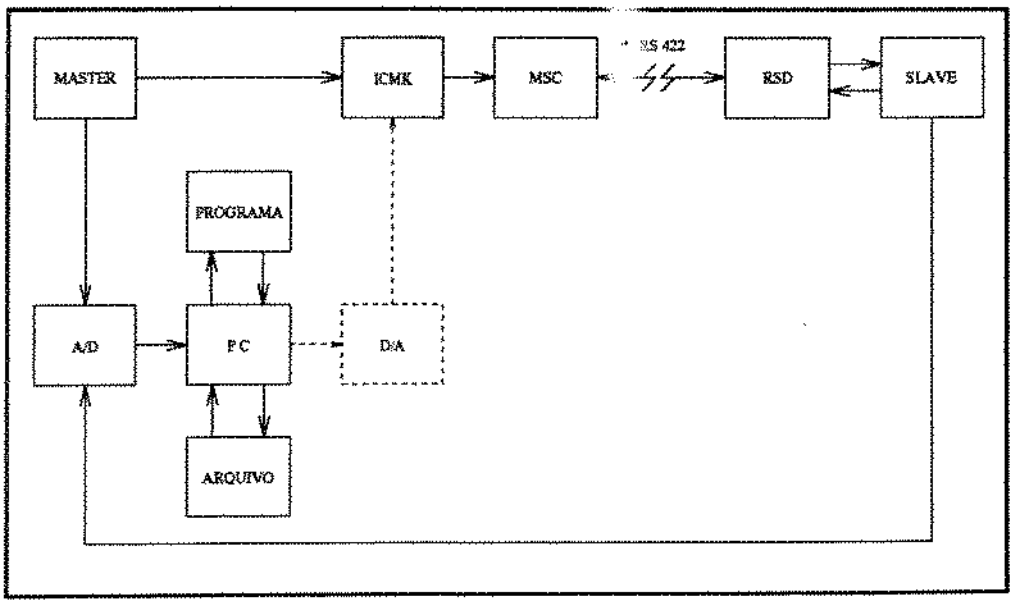


Figura 5.2: *Modo 1 - monitoramento*

A alteração realizada no sistema permite uma maior flexibilidade em sua utilização. A figura 5.2 mostra o PC apenas monitorando os sinais do sistema. A figura 5.3 mostra o PC assumindo o papel do *master*, ou seja, gerando trajetórias para o MSC. E, finalmente, a figura 5.4 mostra o PC adquirindo os sinais do *master* e processando-os antes de enviá-los para o MSC.

Além da alteração efetuada no *hardware*, desenvolveu-se todo o *software* necessário para o controle da *interface* ICMK, dos conversores A/D e D/A, acesso às entradas e saídas digitais, acesso à unidade de disco, aquisição e envio de trajetórias, chaveamento dos modos de operação, etc. O *software*, chamado de Controlador do Manipulador Kraft (CMK) foi implementado com sucesso. Maiores informações sobre o sistema Kraft e as alterações realizadas podem ser obtidas em [8].

5.4 Programando o Kraft em ULTRA C

Algumas rotinas criadas para a implementação do CMK foram migradas para a biblioteca UC.LIB de tal forma que elas pudessem ser utilizadas por um programa escrito em ULTRA

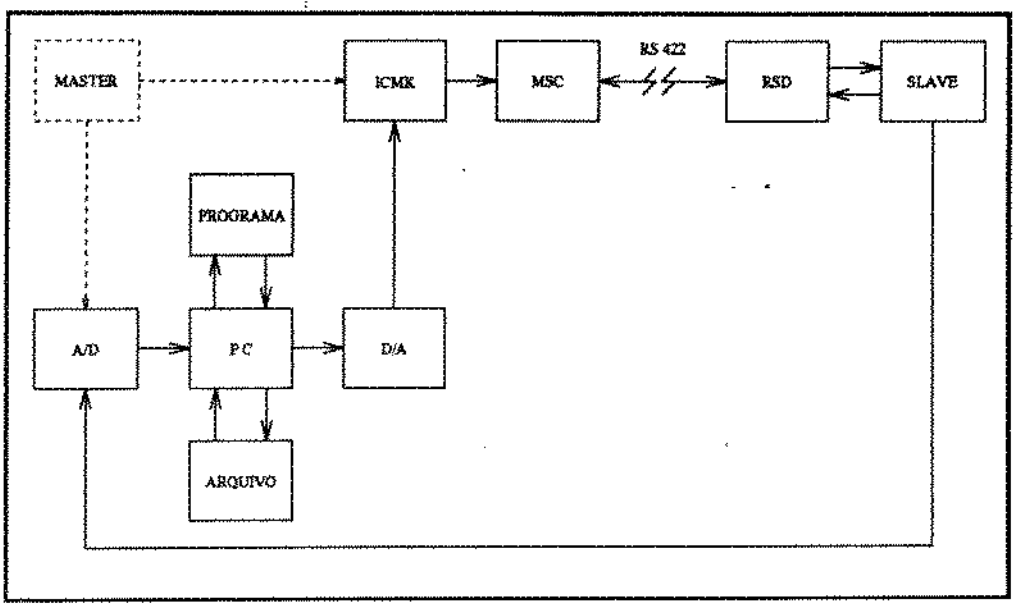


Figura 5.3: *Modo 2 - controle*

C. Desta forma, programas escritos em ULTRA C podem assumir o papel do *master*, ou seja, gerando trajetórias para o MSC enviar para o *slave*. Deve-se ressaltar que, para o programador, a utilização destas rotinas é totalmente transparente. Isto é, nenhuma rotina especial de acesso ao ICMK ou aos conversores A/D e D/A precisa ser utilizada diretamente pelo programador. A utilização destas rotinas é implícita. Este detalhe dá à linguagem, total independência em relação ao robô utilizado. Pode-se criar uma *biblioteca de robôs* contendo as rotinas de controle do *hardware* referente a cada robô. Para um usuário acostumado a utilizar ULTRA C para programar o manipulador Kraft, será bastante fácil passar a programar um robô da linha Puma, por exemplo.

O apêndice C traz dois exemplos de programação do manipulador Kraft. Observar a independência que os programas têm em relação às características específicas da geometria do manipulador e da interface ICMK.

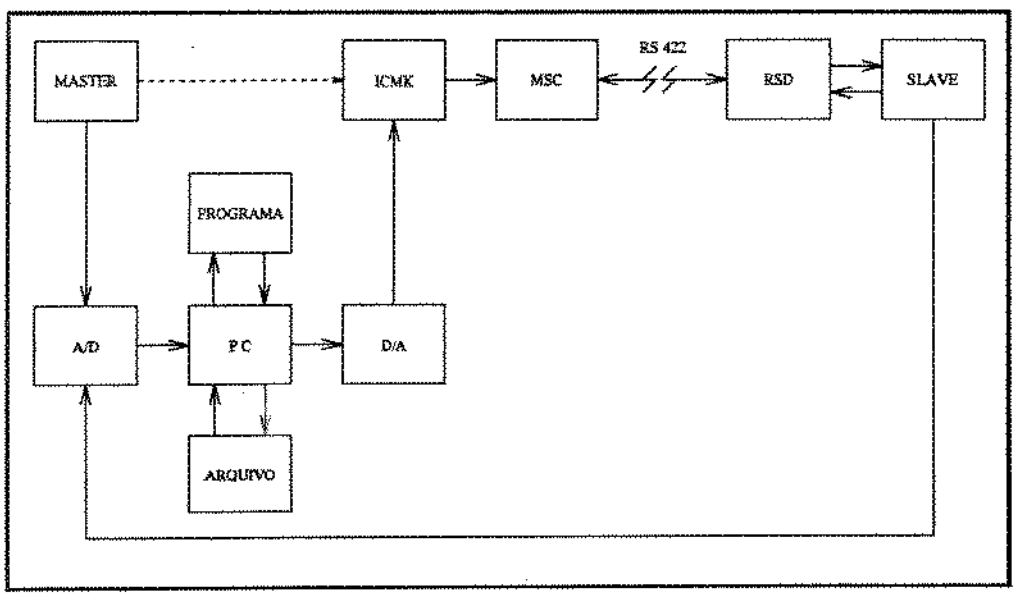


Figura 5.4: *Modo 3 - Simulação de Vôo*

Capítulo 6

Conclusões e Perspectivas

Ao longo deste trabalho desenvolveu-se um sistema capaz de aumentar o leque de aplicações possíveis para o manipulador Kraft com vistas a sua utilização em tarefas de automação submarina.

O sistema apresentado cumpre os objetivos inicialmente planejados de prover o telemanipulador Kraft da facilidade de programação *off-line*. Apesar de não ser o objetivo principal deste trabalho, manteve-se durante todo o projeto a preocupação de não se particularizar as soluções para o caso específico do manipulador Kraft. O resultado obtido foi uma ferramenta bastante poderosa e suficientemente genérica para que seja possível, com um mínimo de alterações, sua utilização na programação de diferentes tipos de robôs. Devido ao fato de estar baseado na linguagem ULTRA C, que é uma linguagem modular completamente aberta à implementação de novas funções, o ambiente UCI pode ser facilmente adaptado a diferentes tipos de *hardware*. Este fator torna-se o principal atrativo para sua implementação industrial.

Outra característica interessante do sistema apresentado, é que ele foi desenvolvido e implementado em uma plataforma de *hardware* simples e de uso bastante difundido (IBM PC e compatíveis).

Do ponto de vista acadêmico, tem-se uma ferramenta de apoio ao ensino da robótica. Através de sua utilização, estudantes poderão entrar em contato direto com as diferentes tecnologias relacionadas à automação e robótica, sem que haja a necessidade da presença física de um robô.

Estão previstas algumas melhorias nas funções já implementadas e a criação de novas funções que facilitem ainda mais sua utilização. Uma destas melhorias consiste em se substituir a representação gráfica em estrutura de arame (*wire-framming*) pela representação através da

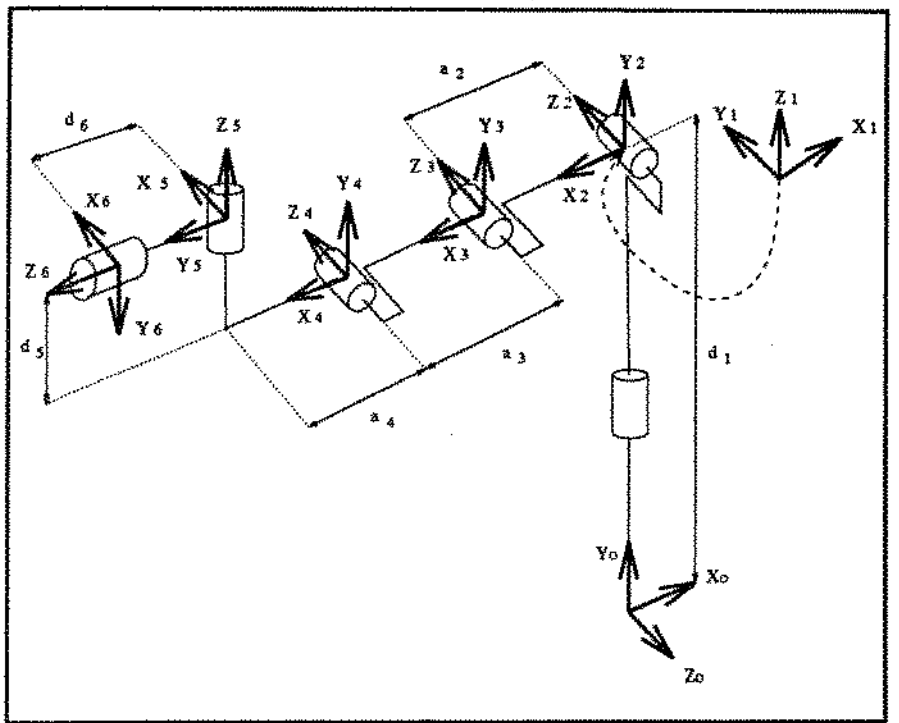
eliminação de arestas ocultas e texturização das faces dos sólidos. Para que este objetivo seja alcançado, todo o sistema deverá migrar para uma plataforma de *hardware* mais veloz.

Paralelamente, pretende-se criar uma ***biblioteca de robôs***, contendo os principais tipos utilizados no mercado. A criação desta biblioteca dará ao programador total independência em relação ao modelo do robô utilizado.

No futuro, serão desenvolvidos e implementados algoritmos de detecção de colisões, tanto *on-line* quanto *off-line*.

Apêndice A

Parâmetros de Denavit-Hartenberg para o Manipulador Kraft



JUNTA i	a_{i-1}	α_{i-1}	d_i	θ_i
1	0	-90°	d_1	θ_1
2	0	-90°	0	θ_2
3	a_2	0°	0	θ_3
4	a_3	0°	0	θ_4
5	a_4	-90°	d_5	θ_5
6	0	-90°	d_6	θ_6

Apêndice B

Matrizes de Transformação Homogênea para o Manipulador Kraft

$$
{}^{0}_{1}T = \begin{bmatrix} c\theta_1 & -s\theta_1 & 0 & 0 \\ 0 & 0 & 1 & d_1 \\ -s\theta_1 & -c\theta_1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad
{}^{1}_{2}T = \begin{bmatrix} c\theta_2 & -s\theta_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -s\theta_2 & -c\theta_2 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{2}_{3}T = \begin{bmatrix} c\theta_3 & -s\theta_3 & 0 & 0 \\ s\theta_3 & c\theta_3 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad
{}^{3}_{4}T = \begin{bmatrix} c\theta_4 & -s\theta_4 & 0 & 0 \\ s\theta_4 & c\theta_4 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

$$
{}^{4}_{5}T = \begin{bmatrix} c\theta_5 & -s\theta_5 & 0 & a_4 \\ 0 & 0 & 1 & d_5 \\ -s\theta_5 & -c\theta_5 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad
{}^{5}_{6}T = \begin{bmatrix} c\theta_6 & -s\theta_6 & 0 & 0 \\ 0 & 0 & 1 & d_6 \\ -s\theta_6 & -c\theta_6 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
$$

Onde $s\theta_i$ significa $\mathrm{sen}(\theta_i)$ e $c\theta_i$ significa $\cos(\theta_i)$.

Apêndice C

Exemplos de Programação Usando ULTRA C

Exemplo 1

```
/* Incluindo o arquivo cabecalho uc.h que contem as definicoes de */
/* funcoes e variaveis globais utilizadas pela linguagem ULTRA C  */
#include <uc.h>

main()
{
/* Definicao de variaveis locais                                  */

   POINT POS_OBS,VISAO;
   JOINT ANGULOS={0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

/* Inicio do programa                                             */
   start_uc();

/* Inicializando o modo grafico                                   */
   start_gr();

/* Definindo o centro do desenho                                  */
   XCENTER=MAXX/2;
   YCENTER=MAXY/2;

/* Definindo o espaco de trabalho                                 */
   defworkspace(10000,10000,10000);

/* Atribuindo valores ao ponto POS_OBS (posicao do observador)    */
   POS_OBS.x=100;
   POS_OBS.y=200;
   POS_OBS.z=-300;

/* Atribuindo valores ao ponto VISAO (direcao de observacao)      */
```

```
   VISAO.x=0;
   VISAO.y=200;
   VISAO.z=0;

/* Definindo a posicao do observador                          */
   defobserver(POS_OBS,VISAO);

/* Carregando os dados referentes ao manipulador Kraft        */
   loadrobot(KRAFT);

/* Definindo a posicao de home                               */
   defhome(ANGULOS);

/* Carregando a garra tipo 1                                  */
   loadgripper(GARRA1);

/* Desenhando o manipulador + garra na posicao de home        */
   drawrobot();

/* Definindo a velocidade maxima para os movimentos           */
   speed(100);

/* Definindo orientacao livre                                 */
   orientation(FREE);

/* Definindo trajetoria ponto a ponto                         */
   interpol(PTP);

/* Fazendo a garra do manipulador se aproximar 20mm do        */
/* observador no sentido negativo do eixo Z                   */
   approach(POS_OBS,20,-DZ);
   drawrobot();

/* Fechando completamente a garra                             */
   closegripper(0);
```

```
    drawrobot();

/* Fazendo o robo voltar para a posicao de home                    */
    home();
    drawrobot();

/* Abrindo completamente a garra                                   */
    opengripper(TOTAL);
    drawrobot();

/* Voltando ao modo texto                                          */
    end_gr();

/* Encerrando o programa                                           */
    end_uc();
}
```

Exemplo 2

```
/* Incluindo o arquivo cabecalho uc.h que contem as definicoes de */
/* funcoes e variaveis globais utilizadas pela linguagem ULTRA C  */
#include <uc.h>

/* Definindo as variaveis globais.                                */
/* As variaveis globais podem ser utilizadas por qualquer funcao  */
/* do programa                                                    */
STRUCTURE PECA,PECA_AUX;
FRAME     DEPOSITO,ALIMENTADOR;

main()
{
/* Definicao de variaveis locais                                  */
/* As variaveis locais so podem ser utilizadas dentro das funcoes */
/* onde forem definidas                                           */

   JOINT ANGULOS={pi/3.0,-pi/6.0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

/* Inicio das instrucoes do programa                              */
   start_uc();

/* Definindo o espaco de trabalho                                 */
   defworkspace(5000,1000,1000);

/* Definindo a posicao e a orientacao do alimentador de pecas     */
   ALIMENTADOR.x=250;    /*              */
   ALIMENTADOR.y=810;    /*  posicao     */
   ALIMENTADOR.z=-235;   /*              */

   ALIMENTADOR.a=pi/4;   /*              */
   ALIMENTADOR.b=0;      /*  orientacao  */
   ALIMENTADOR.c=-pi;    /*              */

/* Carregando os dados referentes ao manipulador Kraft            */
```

```
   loadrobot(KRAFT);

/* Definindo a posicao de home                                    */
   defhome(ANGULOS);

/* Carregando a garra tipo 1                                      */
   loadgripper(GARRA1);

/* Carregando os dados referentes a peca                          */
   loadstr(PECA);

/* Definindo a velocidade maxima para os movimentos               */
   speed(100);

/* Definindo orientacao livre                                     */
   orientation(FREE);

/* Definindo trajetoria linear                                    */
   interpol(LINEAR);

/* Fazendo o manipulador ir para a posicao de home                */
   home();

/* Iniciando o loop de transferencia de peca                      */
   while(pega_peca()==0){
      coloca_peca();
   }

/* De volta para a posicao de home                                */
   home();

/* Encerrando o programa                                          */
   end_uc();
}
```

```
/* Inicio da definicao da funcao pega_peca()                          */
int pega_peca(void){

/* Verificando se existe peca no alimentador atraves de uma           */
/* porta paralela do PC                                               */
   if(inportb(0)==0){
      printf("Nao existe peca no alimentador !!!");
      /* retornando o valor 1 para a funcao main() (fim de peca)      */
      return 1;
   }

/* Atribuindo a PECA_AUX os dados referentes a PECA                   */
   PECA_AUX=PECA;

/* Redefinindo o referencial de pega da peca                          */
/* Equivale a girar a peca em torno da origem com a rotacao           */
/* definida pelo frame ALIMENTADOR e depois translada-la ate a        */
/* a posicao do alimentador                                           */
   PECA_AUX=rotstr(PECA_AUX,ALIMENTADOR.a,
    ALIMENTADOR.b,
    ALIMENTADOR.c,
    XYZ);
   PECA_AUX=movstr(PECA_AUX,ALIMENTADOR.x,
    ALIMENTADOR.y,
    ALIMENTADOR.z);

/* Movimentando a garra ate a peca e acoplando-a                      */
   takestr(PECA_AUX);

/* Fechando a garra                                                   */
   closegripper(10);

/* Retornando para a funcao main() o codigo 0                         */
```

```
   return 0;
}
/* Fim da definicao da funcao pega_peca()                    */



/* Inicio da definicao da funcao coloca_peca()               */
int coloca_peca(){

/* Movendo a garra para a o deposito de pecas                */
   moveto(DEPOSITO);

/* Soltando a peca                                           */
   dropstr();
   opengripper(100);

/* Retornando para a funcao main()                           */
   return;

}
/* Fim da definicao da funcao coloca_peca()                  */
```

BIBLIOGRAFIA

[1] Acevedo, O. M., "Simulador Gráfico para um Robô Industrial," Tese de Mestrado, Unicamp, 1993.

[2] Ahmad, S. I., "Programming, Task and Motion," in *International Encyclopedia of Robotics Applications and Automation*, vol. 2, pp. 1260-1268, 1988.

[3] Amaral, P. F. S., "Concepção e Implementação de um Laboratório para Desenvolvimento de Técnicas e Programação de Robôs Industriais," Tese de Doutorado, EPUSP, 1985.

[4] Aust, E., Boke, M., Gustmann, M., Hahn, G., Niemann, H. R., Schultheiss, G. F., "Development and Testing of a Freely Programmable Robot for Work in Deep Water," in *SPE 1990 - Latin American Petroleum Conference - IV Congresso Brasileiro de Petroleo*, Rio Centro, Rio de Janeiro, Brasil, Petrobrás,IBP,SPE, pp.01-18, 1990.

[5] Brown, M. K.,Buntschuh,B. M., e Wilpon, J. G., "SAM: A Perceptive Spoken Language Understanding Robot," in *IEEE Trans. Systems, Man, and Cybernetics*, vol. 22, pp. 1390-1402, Dez, 1992.

[6] Brown, M. K.,Wilpon, J. G.,Wei, N. C., e Smith, D. R., "Controlling a Robot with Natural Connected Speech," in *Proc. 1986 Conf. Intelligent Systems and Machines*, pp. 122-127, Abr, 1986.

[7] Craig, J. J., " Introduction to Robotics : Mechanics and Control," Addison-Wesley, 2ed., 1989.

[8] Cruz, J. M. C., "Projeto e Desenvolvimento de um Sistema de Geração Automática de Trajetórias para Manipuladores Industriais," Tese de Mestrado, Unicamp, 1993.

[9] Deiseroth, M. P.,Wilpon, "Robot Teaching," in *Handbook of Industrial Robotics*, pp. 352-365, 1985.

[10] Denavit, J., Hartemberg, R. S., "A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices," in *Journal of Applied Mechanics*, pp. 215-221, Jun,1955.

[11] Ferreira, E. P., "Robótica," II Escola Brasileiro-Argentina de Informática, Buenos Aires, Argentina, 1987.

[12] Gorla, B.,Renaud, M., "Modéles des Robots Manipulateurs: Application à Leur Command," Cepadues Editions, Toulouse, França, 1984.

[13] Gruver, W. A., "Industrial Robot Programming Languages: A Comparative Evaluation," in *IEEE Transactions on System Man and Cybernetics*, Vol. SMC-14, n. 4, pp. 565-570, 1984.

[14] Fu, K. S., Gonzalez, R. C., Lee, C. S. G. "Robotics Control, Sensing, Vision, and Intelligence," McGraw-Hill,inc., 1987.

[15] Gomaa,H., "Programming of Multiple Robot Systems," in *International Encyclopedia of Robotics Applications and Automation*, vol. 2, pp. 1250-1260, 1988.

[16] Kelly-Bootle, S., "Mastering Turbo C," Sybex, Inc., 1989.

[17] Kernighan, B., Ritchie, D., "The C Programming Language," Prentice-Hall, 1978.

[18] Kraft, "Underwater Manipulator System," 1985.

[19] Lee, C. S. G., and Ziegler, M., "A Geometric Approach in Solving the Inverse Kinematics of PUMA Robots," in *IEEE Trans. Aerospace ann Eletronic Systems*, vol. AES-20, No. 6, pp. 695-706, 1984.

[20] Lozano-Pérez,T., "Robot Programming," in *Proceedings of IEEE*, vol. 71, pp. 821-841, Jul,1983.

[21] Lynx Tecnologia Eletrônica Ltda, "CAD 12/36 Manual do Usuário e de Referência," 1989.

[22] Lynx Tecnologia Eletrônica Ltda, "CDA 12/08 Manual do Usuário e de Referência," 1991.

[23] Mendeleck, A., "Uma Linguagem para Programação Off-line de Robôs," Tese de Mestrado, 1991.

[24] Miss, R. W., Schultheiss, G. F., "The Design of an Algorithm For the Control of Redundant Kinematic Chains," 5th ICAR, Pisa, Itália, 1992.

[25] Niemann, H. R., Aust, E., Gustmann, M., Hahn, G.: "A Six DOF Robot Allows Diverless Intervention," GKSS 92/E/44, Alemanha, 1992.

[26] Paul, R. P., "Robot Manipulator Mathematics, Programming, and Control," MIT Press series in Artificial Intelligence, 1981.

[27] Rembold, U., "Programming of Industrial Robots. Today and in the Future," in *Languages for Sensors-based Control in Robotics*, pp. 3-23, 1987.

[28] Rogers, D., Adams, J., "Mathematical Elements for Computer Graphics," McGraw-Hill, Inc., 1990.

[29] Rosario, J.M., Weber, H.I., Morooka, C., "Development and Control of Advanced Robots for Underwater Tasks," 5th ICAR, Pisa, Itátia, pp. 1792–1795, 1991.

[30] Schildt, H., "Advanced C," McGraw-Hill, Inc., 1988.

[31] Schildt, H., "C The Complete Reference," McGraw-Hill, Inc., 1990.

[32] Snyder, W. E., "Industrial Robots: Computer Interfacing and Control," Prentice Hall, Inc., 1985.

[33] Spur,G., et al, "An Integrated Approach Towards Off-Line Robot Programming," in *Off-Line Programming of Industrial Robots*, pp. 181–191, 1987.

[34] Storr,A.,Schumacher,H., "Programming Methods for Industrial Robots," in *Off-Line Programming of Industrial Robots*, pp. 1–4, 1987.