UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

"PROJETO E IMPLEMENTAÇÃO DE UM NÚCLEO DE SISTEMA OPERACIONAL DISTRIBUÍDO COM MECANISMOS PARA TEMPO REAL"

DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE FEDERAL DE SANTA CATARINA

PARA OBTENÇÃO DE GRAU DE MESTRE EM ENGENHARIA ELÉTRICA

LUIZ NACAMURA JÚNIOR

FLORIANÓPOLIS, JULHO - 1988

PROJETO E IMPLEMENTAÇÃO DE UM NÚCLEO DE SISTEMA OPERACIONAL DISTRIBUÍDO COM MECANISMOS PARA TEMPO REAL

LUIZ NACAMURA JÚNIOR

ESTA DISSERTAÇÃO FOI JULGADA PARA A OBTENÇÃO DO TÍTULO DE MESTRE EM ENGENHARIA - ESPECIALIDADE ENGENHARIA ELÉTRICA E APROVADA EM SUA FORMA FINAL PELO CURSO DE PÓS-GRADUAÇÃO.

Prof. Joni da Silva Fraga, Dr. Ing.

Orientador

Prof. Márcio Cherem Schneider, Dr. Sc.

Coordenador do Curso de Pós-Graduação em Engenharia Elétrica

Banca Examinadora

Prof. Claúdio Walter, Dr. Ing.

Prof. Jean-Marie Farines, Dr. Ing.

Prof. Rogério Drummond B. P. de Mello Filho, PhD.

AGRADECIMENTOS

Ao professor Joni da Silva Fraga pela orientação e apoio na realização deste trabalho.

Aos menbros da Banca Examinadora, pelos seus comentários e sugestões que enriqueceram o trabalho.

A todos os professores, colegas e funcionários, do Programa de Pós-Graduação em Engenharia Elétrica e do Laboratório de Controle e Microinformática (LCMI/DEEL), que de uma maneira de outra contribuiram para a realização deste trabalho. Em especial aos Professores Jean-Marie Farines e Vitório Bruno Mazzola, a Engenheira Claire Jallet e ao Bolsista Luiz Carlos Oenning Martins.

Aos meus pais, Luiz Nacamura e Zuleika Valverde Nacamura, e as minhas irmãs Luiza de Fátima Nacamura, Liliane Nacamura e Luciene de Cássia Nacamura pelo apoio e compreensão.

A UFSC e a CAPES pelo apoio financeiro que permitiu a realização deste trabalho.

SUMÁRIO

RESUMO

A presente dissertação apresenta o projeto e a implementação de um núcleo de Sistema Operacional Distribuído. O referido núcleo é responsável pela implantação de um ambiente multitarefas distribuído. As necessidades de tempo real são tratadas através de política de escalonamento e mecanismos de temporização.

O núcleo fornece um conjunto de primitivas de comunicação e sincronização entre tarefas (IPC) que permitem uma comunicação uniforme e independente da distribuição no sistema, com diferentes possibilidades de sincronismos (síncronas e assíncronas) e modos de endereçamento (um-para-um, um-para-vários, etc.).

O núcleo fornece ainda o suporte para a configuração estática e dinâmica (reconfiguração) de um sistema distribuído.

Este núcleo foi desenvolvido como parte do Ambiente de Desenvolvimento e Execução de Software (ADES) que corresponde a um conjunto de ferramentas cujo objetivo é facilitar a concepção de software distribuído. Este ambiente está centrado sobre uma Linguagen de Implementação de Sistema (LIS), sendo que o núcleo em questão fornece então suporte de tempo de execução para abstrações definidas por esta linguagem.

ABSTRACT

This work presents the design and implementation of the Distributed Operating System Kernel. The Kernel allows the generation of a Distributed Multitasking environment. The real time requitements are carry out by a scheduling policy and timing mechanims.

O Kernel provides an interporcess communication (IPC) that allows a uniform and system distribution independence, with different possibilities of synchronization (synchrous and asynchronous) and addressing ways (one-to-one, one-to-many, etc...).

The Kernel also provides a Static e Dynamic Configuration (Reconfiguration) support for Distributed Systems.

This Kernel was developed as part of a Software Development and Execution Environment (ADES) that corresponds to a set of tools whose aim is to make easy the distributed software conception. This environment is based in a System Implementation Language (LIS), that uses the Kernel as Run-Time support.

CAPÍTULO 1

INTRODUÇÃO

A evolução atual em automação industrial tem mostrado uma crescente utilização de Sistemas Distribuídos: isto tanto nas atividades de controle de processos com os chamados SDCDs (Sistemas Digitais de Controle Distribuído), como mais recentemente em automação da manufatura com as propostas de Sistemas Integrados de Manufatura (CIM - "Computer Integrated Manufacturing").

Ambos, controle de processos e automação de manufatura têm-se beneficiado dos Sistemas Distribuídos pela própria característica distribuída destas aplicações. São inúmeras as vantagens apontadas na literatura destes sistemas em relação a soluções mais clássicas. A disponibilidade de recursos, o desempenho e o crescimento incremental são fatores normalmente citados.

Sistemas com vários elementos de processamento, que se comunicam entre si exclusivamente através de uma rede de comunicação são normalmente chamados de Sistemas Distribuídos.

A utilização destes conduz a necessidade de um suporte básico para o gerenciamento e a integração em tempo de execução, dos recursos distribuídos na rede. Este suporte básico deve levar em consideração a natureza das aplicações em tempo real; os sistemas de computadores devem responder suficientemente rápido a estímulos do processo controlado de maneira a atender as necessidades deste último.

A motivação desta dissertação foi o estudo e a realização (projeto e implementação) de um Núcleo de Sistema Operacional Distribuído, que preenche em grande parte as necessidades de suporte em tempo de execução nestes sistemas. Este núcleo foi projetado para a utilização em aplicações industriais. É o responsável pela implantação de um ambiente multitarefas distribuído que executa a aplicação. As necessidades de tempo real são tratadas através de políticas de escalonamento e mecanismos de temporização.

O núcleo fornece um conjunto de primitivas de comunicação e sincronização entre tarefas (IPC) que permite comunicações uniformes e independentes da distribuição das tarefas no sistema. Também é fornecido suporte para configuração estática e dinâmica (reconfiguração) de um sistema distribuído; isto atende as necessidades em automação industrial de sistemas evolutivos, com fácil adaptação a mudanças no meio onde atuam.

Este núcleo foi desenvolvido como parte do **Ambiente de Desenvolvimento e Execução** (ADES) (FRAGA, 1988) que por sua vez corresponde a um conjunto de ferramentas cujo objetivo é facilitar a concepção de software distribuído. Este ambiente

segue a arquitetura CONIC (KRAMER, 1985) e está centrado sobre uma Linguagem de Implementação de Sistemas (LIS); sendo que o núcleo em questão fornece o suporte de tempo de execução para as abstrações definidas nesta linguagem.

Este trabalho está dividido em 6 capítulos. No segundo capítulo é apresentado um estudo introdutório sobre Sistemas Distribuídos e numa segunda etapa são descritas as particularidades destes sistemas quando utilizados em aplicações de tempo real.

No terceiro são discutidos os principais modelos de programação distribuídas, normalmente citados na literatura. Também é objeto deste capítulo a apresentação do paradigma de programação suportada pelo núcleo.

O quarto capítulo introduz as principais unidades funcionais do Núcleo, sendo também discutidos aspectos dos mecanismos que fazem deste um núcleo para aplicações em tempo real. O quinto capítulo descreve detalhes da implementação e os resultados obtidos. No sexto capítulo são apresentadas as conclusões finais deste trabalho.

CAPÍTULO 2

SISTEMAS DISTRIBUÍDOS PARA TEMPO REAL

2.1. Introdução

A utilização de processamento distribuído e de redes de computadores tem apresentado um crescimento significativo nos últimos anos. O advento dos chamados "sistemas distribuídos" tem exigido o desenvolvimento de novos modelos de programação e também de ferramentas apropriadas para o compartilhamento de recursos e de informações. Diante disto, estes sistemas tem se mostrado um campo promissor de pesquisa para especialistas de áreas tão diversas, tais como de linguagens, base de dados, sistemas operacionais, algorítmicas, etc...

Neste sentido, o objetivo deste capítulo é apresentar um estudo introdutório sobre sistemas distribuídos, procurando caracterizá-los através de conceitos e atributos, sendo também

finalidade deste capítulo explicitar os requisitos impostos a estes sistemas quando de aplicações em tempo real.

2.2. Caracterização de Sistemas Distribuídos

Existem ainda na literatura algumas confusões e certas contradições nos conceitos envolvidos com sistemas de processamentos distribuídos. Nesta dissertação será adotado a noção de sistemas distribuídos introduzido por Enslow (ENSLOW, 1981), e que tem como base as seguintes características:

- **Multiplicidade de elementos de processamento:** cada elemento de processamento corresponde a um processador operando sobre uma memória local, provendo os mais variados serviços;

- **Fraco acoplamento entre os componentes:** componentes lógicos e físicos encontram-se fisicamente distribuídos, apresentando-se interconectados através de uma rede de comunicação de dados. A comunicação entre componentes é realizada por troca de mensagens através desta rede. O compartilhamento de memória entre componentes não é permitido, o quê exclui desta classe os sistemas ditos fortemente acoplados (comunicação através de memória compartilhada);

- **Um sistema operacional de alto nível:** permite a integração dos diversos recursos distribuídos na rede;

- **Transparência de sistema:** permite requisitar serviços através de nomes, sem necessidades de se precisar ou manusear endereços (localização física);

- **Autonomia cooperativa:** os elementos de processamento são autônomos. Esta estratégia de controle implica na não existência de hierarquias entre nós (componentes). Desta forma, o gerenciamento em cada nó é feito segundo base local;

As potencialidades dos sistemas distribuídos e as vantagens sobre sistemas mais tradicionais são longamente apresentados na literatura. A seção seguinte tem por objetivo apresentar alguns dos atributos e requisitos de um sistema distribuído, que contribuem na apreciação positiva destes sistemas.

2.3. Requisitos e Atributos de um Sistema Distribuído

O desenvolvimento de novos modelos, ferramentas e mecanismos para sistemas distribuído tem como objetivo atingir:

- **Desempenho:** "os sistemas distribuídos, apresentam vários elementos de processamento que cooperam em uma simples atividade e utilizando se de técnicas descentralizada para coordenação deste processamento distribuído (controle descentralizado) e que não apresente um acréscimo significativo do "overhead" devido a esta coordenacão deve obter melhor desempenho e menores tempos de resposta que sistemas mais convencionais. A possibilidade de concentrar um número mínimo de funções por elementos de processamento

simplifica as necessidades de suporte e torna o processamento mais eficiente. Em sistemas tempo real o melhor desempenho está associado as estações especializadas (estações com funções definidas) (READY, 1986)";

- **Flexibilidade:** um sistema flexível é aquele que pode adaptar-se às mudanças do ambiente sem que ocorra ruptura de suas funções (LE LANN, 1981). Estas mudanças são devidas às exigências de desempenho ou funcionalidade do sistema. O requisito flexibilidade é plenamente satisfeito em sistema distribuído pela característica modular destes; isto permite então, sistemas evolutivos com facilidades para o crescimento incremental, a instalação e a manutenção;

- **Alta Disponibilidade e Grande Confiabilidade:** a separação física entre recursos e funções do sistema, reduzindo a correlação entre falhas, e a redundância inerente a estes sistemas permitem que operações contínuas sejam suportadas pelo mesmo em presença de faltas e perdas de elementos de processamento; oferecendo portanto, grande potencial para a construção de sistemas seguros de funcionamento ("dependability" (LAPRIE, 1986);

- **Compartilhamento de recurso:** o compartilhamento ótimo de recurso é uma das principais metas a ser alcançada em um sistema. Em sistemas distribuídos o compartilhamento requer o controle descentralizado de todas as atividades, obtendo uma alocação ótima e dinâmica dos recursos distribuídos na rede. Para o usuário, o compartilhamento

de recursos deve ser transparente, apresentando-se embutido em facilidades de utilização, independente da localização. Geralmente estas facilidades devem ser providas pelo Sistema Operacional.

2.4. Modelos de Referência para Sistemas Distribuídos

2.4.1. Motivações

A natureza distribuída e a autonomia dos componentes determinam algumas características nestes sistemas que são notórias:

- **Comunicação com atrasos variáveis:** a troca de informações entre componentes, utilizando uma rede de comunicação de dados, leva a seguinte constatação: existe um inevitável atraso na transmissão de mensagens. A latência entre a produção de um evento e a manifestação deste nos diversos componentes do sistemas, provocada por estes atrasos variáveis, determina uma inconsistência nas informações de estado global mantidas distribuídas nestes sistemas. Os algoritmos distribuídos, desta maneira, são executados ou sob uma visão local ou a partir de uma estimativa de estado global;

- **Heterogenidade de componentes:** as necessidades de sistemas evolutivos e atualizados, podem determinar a heterogeneidade dos componentes nestes sistemas. Esta heterogeneidade pode se refletir tanto a nível de hardware

como de software. Mecanismos (abstrações) devem ser introduzidos de forma que os componentes possam ser integrados mantendo a transparência em suas utilizações;

- **Descentralização no Controle:** permite uma dinâmica na alocação de recursos, visando a autonomia cooperativa e o balanceamento de carga. As dificuldades do controle descentralizados estão ligadas aos atrasos variáveis e nas inconsistência das informações mantidas em cada gerenciador; isto determina a complexibilidade dos algoritmos distribuídos, envolvendo normalmente uma troca significativa de informações para o controle descentralizado.

Diante dos fatos citados, chega-se a conclusão da complexidade e por vezes da ineficiência de ferramentas, modelos e técnicas que seguem abordagens clássicas quando aplicadas a estes sistemas. Nestes sistemas, o domínio da complexidade tem sido feito utilizando-se de modelos de referência que definem uma forma estruturada e consiste na decomposicão do sistema em camadas, criação de abstrações de serviços e o respectivos protocolos.

2.4.2. Técnica de camadas: Definições

O uso de **técnicas de camadas** em sistemas complexos é consagrada. A estruturação de sistemas em camadas é flexível e pode variar tanto em número de camadas como em complexidade destas. Camadas correspondem a níveis de abstrações "semi-

opaco", sendo que associadas a uma **camada (N)** estão duas interfaces (ver figura 2.1.): uma interface provendo serviços para a camada (N+1) e superiores, e uma interface acessada pela camada (N), de maneira a obter os serviços fornecidos pela camada (N-1) e inferiores. Cada camada deve prover novos serviços a partir dos fornecidos pela(s) camada(s) inferior(es), de tal forma que a camada superior tem o conjunto completo de serviços necessários para executar, aplicações distribuídas.

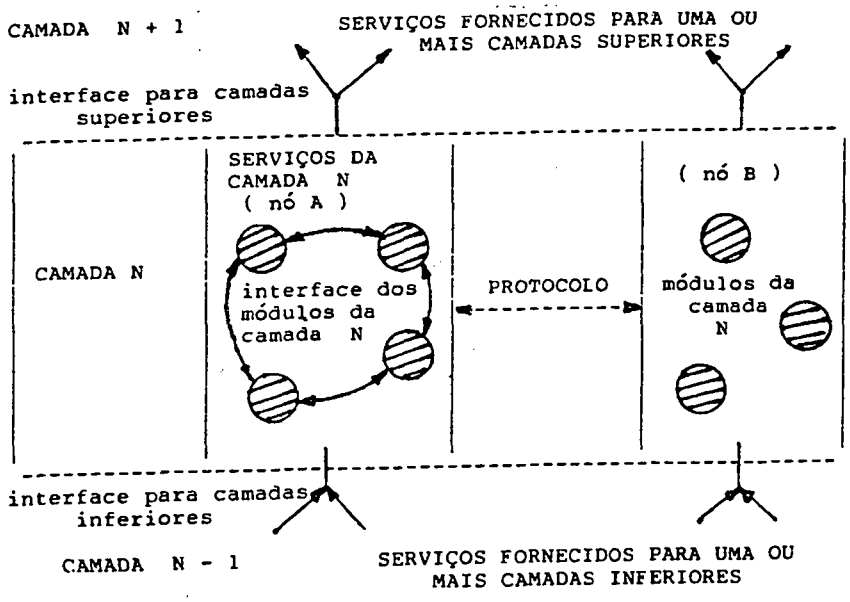


figura 2.1. - Estrutura de Camada

Os serviços fornecidos por uma camada N são futuramente decompostos e implementados através de **módulos**, podendo estes módulos serem distribuídos. Os módulos implementam os serviços de camada gerenciando um conjunto de recursos físicos ou lógicos. Os módulos que implementam a representação de um tipo de recurso e as operações nesta representação são chamados no modelo de gerenciadores ou de servidores de recursos ou simplesmente de servidores. Os detalhes de implementação de um tipo interessam apenas ao servidor. Esta característica é

importante quando o sistema distribuído é composto de componentes heterogêneos.

As cooperações entre módulos, da mesma forma que as solicitações dos serviços de uma camada, são feitas através das interfaces. Desta forma existem duas interfaces: uma entre camadas e outra entre módulos cooperantes de uma dada camada. Uma **interface** (1) é definida como um conjunto de convenções para troca de informações entre duas entidades, sendo constituída de 3 componentes (WATSON, 1981):

- Um conjunto de objetos abstratos visíveis e para cada um destes objetos um conjunto de operações e parâmetros associados;

- Um conjunto de regras que governam a sequência correta destas operações;

- Um conjunto de convenções sobre codificação e formatação exigidas para operações e parâmetros.

---

(1) Esforços tem sido realizados com a intenção de se utilizar o termo interface conotando apenas convenções para comunicação através de camadas adjacentes ou entre entidades de uma camada. O termo protocolo é utilizado para as comunicações entre módulos cooperantes distribuídos de uma mesma camada.

2.4.3. Um Modelo para Arquitetura Distribuída

O modelo **Arquitetura para Sistema Distribuído** apresentado em (WATSON, 1981), constitui-se num exemplo bastante representativo de estruturação para um sistema distribuído. A organização proposta neste caso, procura definir um visão integrada entre o suporte de interconexão (rede de comunicação), sistema operacional e a aplicação.

O referido modelo está centrado sobre um **Sistema Operacional Distribuído** que é o responsável pelo gerenciamento dos recursos distribuídos no sistema. Este Sistema Operacional deve suportar um paradigma de programação distribuída, criando a noção de tarefa (execução de programa sequencial independentes) como entidade elementar de concorrência e implementando os tipos comunicantes definidos pelo paradigma utilizado.

A arquitetura proposta em (WATSON, 1981) é composta de 4 camadas (ver figura 2.2):

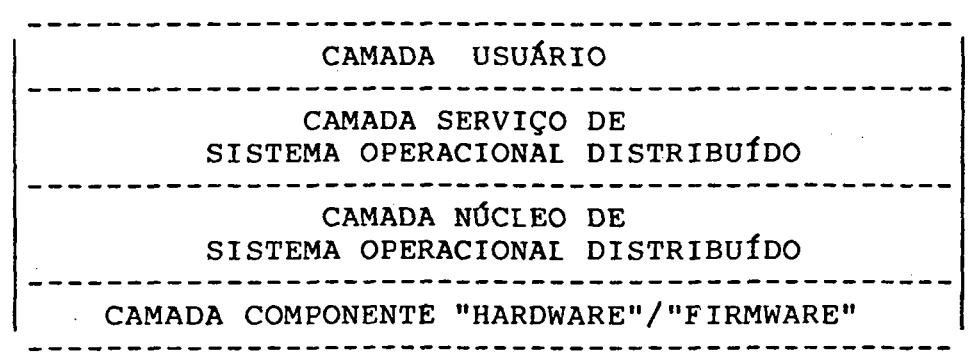


figura 2.2. - Estrutura de Camada do modelo

- **Camada Componente "Hardware"/"Firmware":** constituída de todos os componentes físicos do sistema, incluindo

processadores, memórias e o suporte de interconexão (parte física) que permite a transparência de mensagens de uma estação para outra;

- **Camada Núcleo de Sistema Operacional Distribuído ("Distributed Operating System Kernel"):** provê o suporte de tempo de execução de programas distribuídos, fornecendo os mecanismos de comunicação e sincronismo entre tarefas (IPC) e o ambiente "multi-tasking" distribuído. Fazem também parte desta camada um conjunto de "drivers" que controlam a utilização do suporte físico de E/S.
  Neste modelo é assumido que a interface dos serviços de IPC oferecidos às tarefas no sistema é uniforme, de modo que a comunicação e sincronismo entre tarefas se dê de maneira independente e transparente da distribuição das tarefas no sistema. Internamente o serviço de IPC identifica o destinatário de uma comunicação como local ou remoto, tratando-o com mecanismos adequados (mecanismo de liberação de mensagens com bases locais ou com protocolos fim-a-fim para comunicação remota);

- **Camada Serviço de Sistema Operacional Distribuído:** Esta camada fornece uma grande variedade de serviços úteis para a aplicação. Estes serviços são fornecidos, alocando recursos básicos distribuídos na rede, de maneira transparente ao cliente, levando em conta a descentralização do controle e política de utilização. Estes serviços são executados por tarefas que se comunicam e se sincronizam utilizando o serviço de IPC;

- **Camada Usuário:** Esta camada consiste de tarefas que executam atividades distribuídas ligadas a aplicação. As necessidades da aplicação definirão os serviços requeridos do Sistema Operacional Distribuído.

As tarefas de aplicação e as de serviços de sistema operacional são tratadas de maneira idêntica. A única distinção que pode se fazer entre as tarefas de aplicação e de sistema operacional, é que as últimas podem ser residentes em uma estação por razões de eficiência, de exigências de privilégios no acesso aos "driver" do núcleo, ou ainda de implementarem serviços utilizando-se daqueles providos por outros. No modelo proposto em (NATARAJAN, 1985), estas camadas correspondem a uma única camada, denominada simplesmente de aplicação.

O modelo de arquitetura distribuída apresentado neste item não é conflitante com as proposições dos modelos como os OSI/ISO (DAY, 1983) e o MAP (CROWDER, 1985), mas sim complementar. Estes últimos são descritos como uma abstração do IPC inter-máquina (DAY, 1983). Em (NATARAJAN, 1985) e (SLOMAN, 1986) estes aspectos são examinados e considerações são realizadas sobre a equivalência e a integração de serviços de camadas do modelo OSI/ISO com a subcamada de IPC do núcleo de Sistema Operacional Distribuído.

2.5. Sistema Operacional Distribuído

O Sistema Operacional Distribuído (SOD) é então o responsável pelo gerenciamento e integração dos recursos do

sistema, devendo prover uma visão uniforme e transparente dos mesmos. O requisito de flexibilidade necessário em sistemas distribuídos faz com que o SOD deva também possuir características modulares, permitindo sua estruturação parcial e posterior evolução de acordo com as necessidades da aplicação. Em (FLETCHER, 1980) é proposto uma subcamada de suporte de serviço que além de tratar com a heterogenidade dos componentes, fornece bases para a introdução de novos serviços.

As abordagens utilizadas na implementação de SODs são normalmentes utilizados na literatura para classifica-los. Destacam-se duas abordagens extremas de implementação de um SOD (WATSON, 1981):

- A primeira abordagem não utiliza nenhum software existente; isto corresponde a implementar todas as funções de SOD diretamente no "hardware" da estação. Esta abordagem exige grande esforço, mas apresenta a homogeneidade como principal vantagem. Os recursos são gerenciados com base global. Esta abordagem é apropriada para sistemas em que os computadores são similares ou onde o processamento está concentrado em computadores de médio porte. Esta abordagem é referenciada por **Nível-Base** (ENSLOW, 1981) ou por abordagem de **Sistema Operacional Distribuído** (DEITEL, 1984).

- Na segunda cada estação mantém seu Sistema Operacional Original (SO local). O SOD é implementado através de serviços que se utilizam das funções do SO local. Isto corresponde a introduzir características de distribuição ao

SO local. O gerenciamento neste caso é feito com base local. Esta abordagem simplifica a implementação, mas não é tão homogênea quanto a anterior. Uma desvantagem notória desta abordagem é a possibilidade da existência de superposição de funções entre a extensão introduzida e o SO local, além das modificações que necessitam ser realizadas neste último. Esta abordagem é referenciada como Meta-Sistema (ENSLOW, 1981) ou por abordagem de Sistema Operacional de Rede (DEITEL, 1984).

Como propósito de discussão pode-se afirmar que tal classificação é um pouco restritiva para a grande variedade e complexidade das proposições de SODs existentes. Na literatura são tidos como exemplos de "Sistema Operacional Distribuído" os sistemas: Chorus (ZIMMERMANN, 1984), Accent (RASHID, 1981), Locus (WALKER, 1983) e Roscoe (SOLO, 1979). Os sistemas NSW (FORSDICK, 1978), PNOS (ANANDA, 1984) e MACH (ACCETTA, 1986) são representativos dos ditos "Sistema Operacional de Rede".

No nosso caso, como será posteriormente discutido na seção 4.2., existirá uma estação onde o SO local será mantido (abordagem de Sistema Operacional de Rede) e outras estações em que por indisponibilidade de recursos ou simplicidade das operações as funções do SO local não será necessário (abordagem de Sistema Operacional Distribuído).

2.6. Linguagens de Programação Distribuídas

A programação de Sistemas Distribuídos se fez inicialmente através do uso de interface do SOD (linguagem de interface de SOD). Uma aplicação distribuída, nesta abordagem, é formada a partir de uma coleção de programas sequênciais (programas construídos com linguagem convencionais) que se comunicam utilizando chamadas às primitivas de interface do SOD. Nestas condições, pouco suporte é fornecido para a configuração inicial do sistema e a subsequente monitoração do conjunto de programas que formam a aplicação distribuída.

As dificuldades encontradas neste tipo de abordagem fez com que aparecessem proposições de **linguagens de programação distribuída.** Estas linguagens devem reduzir a complexidade na construção de programas distribuídos quando apresentam características como modularidade, forte verificação de tipos e ainda construções que encapsulem aspectos da arquitetura do software incluindo o suporte para concorrência e a distribuição de funções no sistema.

Linguagens de programação para sistemas distribuídos, em geral, tem sido construidas usando modificações **"ad hoc"** de linguagens sequenciais (Pascal, C, etc) ou de linguagens concorrentes (Modula 2, Pascal Concorrente, etc). Estas últimas já fornecem construções de alto nível para as especificações de tarefas e as interações entre as mesmas.

As linguagens que seguem o modelo **chamada de procedimento** são tidas como apropriadas para sistemas com memória

compartilhada. Em sistemas distribuídos fracamente acoplados devido a presença da rede de comunicação de dados, a interação por **troca de mensagem** é mais natural e em geral muitas das proposições de linguagens para sistemas distribuídos são baseadas, neste modelo. Contudo, o uso corrente de linguagem, tais como ADA e ARGUS, que seguem o modelo **chamada de procedimento**, fez com que surgissem as proposições das linguagens **chamadas de procedimentos remoto** (ou ainda **orientada a operação**), onde a comunicação remota mantém as características de uma chamada de procedimento. A discussão dos modelos de programação será retomada na próximo capítulo desta dissertação.

2.7. Sistema Distribuído para Tempo Real

2.7.1. Características

Os Sistemas Distribuídos para Tempo Real, com aplicações em automação industrial e Controle de Processos, correspondem a um caso particular dentro do universo compreendido por Sistemas Distribuídos. As decisões sobre arquitetura estão sujeitas ao entendimento das característica de um ambiente tempo real (KOPETZ, 1983).

Um Sistema Distribuído para Tempo Real (sistema de controle) é conectado a um processo (objeto controlado) via interfaces: sensores e atuadores. O sistema de controle então recebe dados de sensores de maneira a síncrono ou assíncrono (quando é "event driven"). Estes dados uma vez processados dão origem a saídas para o processo via atuadores. Diante disto, os

Sistemas Distribuídos para Tempo Real, também chamados "sistemas embutidos", apresentam características próprias:

- As interações com o meio ambiente (processo), sendo reguladores ou não, provocam ações no sistema de controle condicionados ao tempo (necessidade de tempo de resposta máximo). Isto corresponde a dizer que as ações além de estarem corretos, devem ser produzidas dentro de um determinado tempo;

- Os elementos de processamento não são de propósito geral mas, apresentam-se como estações especializadas ao longo da planta do processo. A especialização do elemento está ligada às características dos dispositivos de E/S que servem de interface com o processo (atuadores e sensores), permitindo o chamado controle direto;

- As estações são **embutidas**, isto é, apresentam a aplicação, o sistema operacional e o suporte de comunicação fortemente integrados;

- Os elementos componentes destes sistemas não se apresentam de maneira completamente autónoma, mas cooperam ativamente segundo a hierarquia de controle da aplicação;

- Alguns Sistemas para Tempo Real devem operar por longos períodos de tempo sem parar mesmo em casos de mudanças segundo necessidades funcionais ou por manutenção (processo contínuo). Neste caso, os requisitos de confiabilidade e de flexibilidade são muitos fortes.

2.7.2. Comunicação em Sistemas de Controle para Tempo Real

O desenvolvimento da informática nos últimos anos tem viabilizado a utilização de arquiteturas que correspondem a soluções mais naturais em aplicações de maior complexidade em automação industrial e controle de processo. Estas arquiteturas, visando uma maior integração operacional entre os vários níveis de funções relacionadas com a planta, apresentam-se estruturadas em um modelo hierárquico que compreende (PRINCE, 1981 e EMILIANO, 1987): Gerenciamento, Planejamento, Coordenação, Supervisão e Controle Digital Direto (ver figura 2.3.).

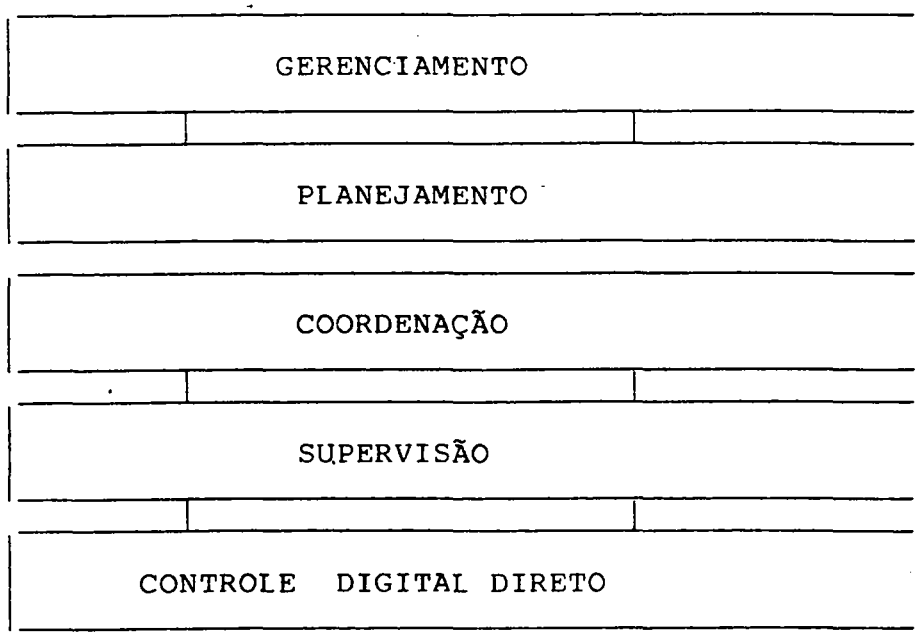


figura 2.3 - Modelo Hierárquico

Embora o sistema de controle esteja organizado hierarquicamente não existe nenhuma exigência quanto a estruturação do(s) sub-sistema(s) de comunicação. Os requisitos impostos ao(s) sub-sistema(s) de comunicação não são homogêneos

para todos os níveis da hierarquia. Em (EMILIANO, 1987) são apresentado vários parâmetros relacionados a comunicação nos diversos níveis de sistema industriais (ver fig 2.4.); constatando-se que, quando maior o nível de hierarquia:

- Menor o número de sistemas participantes;
- Menor é a frequência de comunicação entre estes sistemas;
- Maior é o comprimento das mensagens;
- Maior o tempo de validade da informação;
- Maior a necessidade de comunicação entre sistemas do mesmo nível; e
- Menor a necessidade de comunicação entre níveis adjacentes.

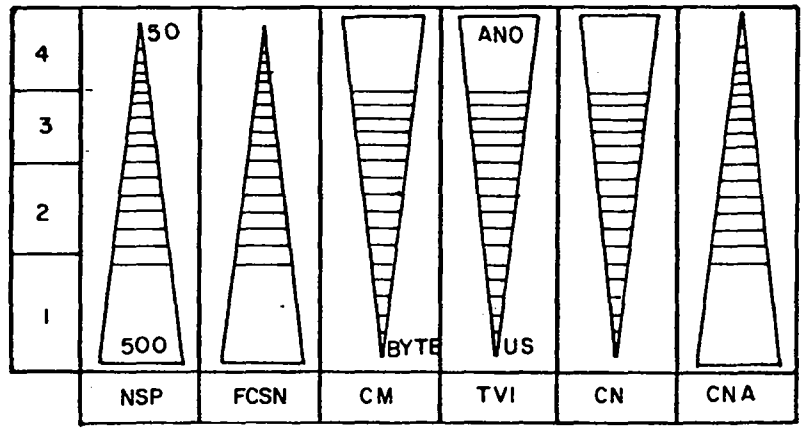


figura 2.4. - Parâmetros de Comunicação nos diversos níveis de Sistemas Industriais.

Um estudo semelhante é realizado em (PRINCE, 1981) onde são apresentadas os tipos e os parâmetros qualitativos (tamanho,

frequência, etc.) das mensagens normalmente trocada em Sistemas Industriais Tempo Real.

A diversidade de equipamentos envolvidos nestes sistemas aliada com a necessidade de integração levaram ao estabelecimento de padrões para os suportes de comunicação. O Modelo de Referência para Interconexão de Sistemas Abertos (OSI) proposto pela ISO ("International Stardards Organization) (DAY, 1983), corresponde a um exemplo típico de padrões desenvolvidos com o objetivo de interligar sistemas computacionais heterogêneos. Este modelo foi inicialmente utilizado no desenvolvimento de redes geográficas de longas distâncias e nas aplicações transacionais, onde os requisitos de tempo de respostas não são tão rígidos. Para aplicações industriais, houve relutância na utilização do modelo em virtude do "overhead" introduzido devido a multiplicidade e inadequação dos mecanismos envolvidos.

Protocolos alternativos baseados no modelo OSI/ISO foram desenvolvidos, visando satisfazer os requisitos de tempo de respostas quando da interconexão de equipamentos em automação industrial. Entre estes se encontra a proposta MAP/TOP (CROWDER 1985) introduzida pelas empresas General Motors e Boeing.

Em princípio esta proposta satisfaz os requisitos de tempo para os níveis superiores da hierarquia de controle de uma aplicação industrial (inferior a 400 milisegundos). Nos níveis mais baixos, onde os requisitos de tempo de respostas são mais severos e os sistemas encontram-se constantemente em cooperação o MAP/TOP mostra-se inadequado. Desta forma, visando

simplificações, abordagens alternativas para o sub-sistema de comunicação têm sido apresentado; nestas, as camadas intermediárias e algumas características (serviço de conexão, controle de erro, etc...) do modelo OSI/ISO são eliminadas ou simplificadas. A arquitetura MINI-MAP (GLAUBE, 1986) corresponde a um exemplo representativo destas alternativas. Esta arquitetura preenche em muito os aspectos levantados nas propostas de comunicação tempo real de (MACLEON, 1982) e (SLOMAN, 1986). Estas propostas basicamentes se resumem na definição de dois níveis para o suporte de comunicação:

- **Nível de transferência física de mensagem:** Neste nível, já existe consenso geral que o mecanismo de transmissão, apropriado para sistemas com requisitos de tempo real é o envio sem reconhecimento (datagrama), isto corresponde a dizer que os serviços de conexão, de controle de erros e de outros mecanismos de confiabilidade não são fornecidos neste nível;

- **Nível de transferência lógica de mensagens:** A semântica para transmissão lógica de mensagem de uma tarefa para outra, também basea-se no envio sem reconhecimento, associando a cada mensagem um tempo de validade. Esgotado este tempo, a mensagem é remotamente descartada e o manipulador local é ativado. Se algum reconhecimento é exigido, estes serão implementados a nível de tarefas aplicativas.

Embora há diminuição de confiabilidade, estas propostas permitem tempos de respostas menores para os sistemas de

controle; isto devido ao melhor desempenho do suporte de comunicação.

2.9. Conclusão

Neste capítulo foi apresentado um estudo introdutório sobre Sistemas Distribuídos para Tempo Real, caracterizando-os através de conceitos e atributos.

Baseado neste estudo podem ser feitas algumas constatações:

- De forma a integrar os recursos do sistema e prover uma visão uniforme e transparente do mesmo é verificado a necessidade de um Sistema Operacional Distribuído. Este sistema operacional pode também suportar a definição de uma Linguagem de Programação Distribuída. Esta linguagem tem por objetivo reduzir a complexidade na programação de aplicações distribuídas;

- As severas exigências de tempo impostas as comunicações em Sistemas Distribuídos para Tempo Real determina que o tratamento de exceções relativa a estas comunicações seja tratada a nível de aplicação.

No próximo capítulo será introduzido um paradigma de programação distribuído que juntamente com as características acima norteou este trabalho.

CAPÍTULO 3

PARADIGMAS DE PROGRAMAÇÃO DISTRIBUÍDA

3.1. Introdução

Uma aplicação distribuída consiste de várias ações se executando de forma paralela e eventualmente trocando informações (cooperando entre si). A complexidade na elaboração da aplicação distribuída pode ser facilitada pela definição de entidades que expressem modularidade, concorrência e distribuição, com características de alto grau de abstração, eficiência, confiabilidade e de mínima exigência de infraestrutura para a implementação. O conjunto destas entidades definem então o chamado **paradigma de programação distribuída.**

Este capítulo tem como principal objetivo introduzir o paradigma de programação distribuída adotado neste trabalho. Para tanto, são examinados inicialmente modelos de programação concorrente e os principais paradigmas de programação distribuída citados na literatura.

3.2. Programação Concorrente

Os conceitos de programação concorrente que há alguns anos atrás se aplicava quase que exclusivamemte a sistemas operacionais, tem sofrido mudanças. Tais conceitos tornaram-se importante também para programadores de diversos tipos de aplicações entre as quais se situam os sistemas tempo real.

A programação concorrente tem mudado substancialmente, principalmente devido ao surgimento na década passada de novos conceitos e notações para expressar concorrência. Uma consequência disto tudo é a aproximação sempre maior entre **sistemas operacionais** e **linguagens de programação**; isto pode ser verificado na medida em que linguagens são projetadas para programar sistemas e sistemas são estruturados de modo a se adequar as características de linguagens (HANSEN, 1978). Características referentes a programação concorrente estão incorporadas num grande número de linguagens de programação como construções de alto nível, permitindo então a estruturação e o suporte para programas concorrentes.

Existem uma grande variedade de linguagens disponíveis para programação concorrente, implementando vários tipos de cooperação (pedidos/respostas, "pipeline", etc.) entre as entidades de concorrência (tarefas, processos, etc). Algumas linguagens se apresentam mais apropriadas para determinados tipos de sistemas (multiprogramados, multiprocessadores, distribuídos, etc.) por apresentarem mecanismos de comunicação e meios de estruturação mais adequados às necessidades de desempenho destes. Dependendo da forma como definem suas

abstrações concorrência e comunicação, as linguagens podem ser agrupadas em 3 classes (ANDREWS, 1983):

- **linguagem orientada a procedimento** (ou **modelo monitor**);
- **linguagem orientada a mensagem;**
- **linguagem orientada a operação.**

3.2.1. Linguagem Orientada a Procedimento

A característica principal destas linguagens são as interações entre tarefas por variáveis compartilhadas. Existem dois tipos de objetos: ativos (processos) e passivos ("locks", semáforos, módulos, monitores, etc.). Os objetos passivos são representados por variáveis compartilhadas, sendo que cada objeto (variável compartilhada) é manipulado através de chamadas aos procedimentos. Desta forma, processos compartilham objetos passivos através de procedimentos. O compartilhamento de procedimentos por processos exige por parte da linguagem mecanismos de exclusão mútua. No modelo **chamada de procedimento** puro o grau de concorrência é controlado criando e destruindo processos. Os principais exemplos desta classe são Modula-2 (WIRTH, 1982), Pascal Concorrente (HANSEN, 1980) e MESA (NELSON, 1981).

3.2.2. Linguagem Orientada a Mensagem

A forma principal de interações entre processos destas linguagens é a **troca de mensagens.** Estas linguagens também são

composta de objetos ativos (processos) e passivos. Ao contrário da classe anterior, existe somente um processo responsável pelo gerenciamento de cada objeto passivo. Este processo é denominado processo servidor. Quando um processo deseja uma operação ou um serviço sobre um objeto, envia uma mensagem para o processo servidor responsável. Ao término da execução do serviço o processo servidor pode enviar uma mensagem de resposta.

No modelo dito **troca de mensagem** puro não há dados globais compartilhados e a concorrência é controlada usando operações independentes de **"SEND"** e **"RECEIVE"**. Portanto, esta classe enfatiza a concorrência explicitamente.

As linguagens desta classe podem ser utilizadas para sistemas com ou sem memória compartilhada. No último caso, a existência do suporte de comunicação (rede de comunicação de dados) é completamente transparente a aplicação. Como exemplo de linguagens com estas características temos CSP (HOARE, 1978), Plits (FELDMAN, 1979).

3.2.3. Linguagem Orientada a Operação

Estas linguagens combinam aspectos das duas classes anteriores; como em linguagens orientada a procedimento as invocações a um objeto passivo são feitas por chamada de procedimento, e semelhante às orientadas a mensagem existe um processo (objeto ativo) responsável pelo gerenciamento de cada objeto passivo. Durante a execução de um serviço o solicitante

e o processo servidor encontram-se sincronizados. Após o término da mesma, ambos continuam concorrentemente. O **"Rendez-Vous"** presente na linguagem ADA (BARNES, 1980) e a **chamada de procedimento remota** apresentam estas características.

3.2.4. Considerações Gerais

As comparações entre as abordagens troca de mensagens e chamadas de procedimento são antigas; Lauer e Needhan (LAUER, 1979) fizeram um estudo comparativo entre modelos canônicos das duas classes. As principais observações citadas neste artigo foram:

- As duas classes são duais;
- Os programas de uma classe são logicamente idênticos ao da outra classe;
- O desempenho de um programa considerando-se os tamanhos das filas, tempo de espera, velocidade dos serviços, etc, são idênticos ao seu programa dual desde que se considere a mesma estratégia de escalonamento.

A partir destas observações é concluído que as exigências determinantes para a escolha da classe de linguagem não se encontram ligadas à aplicação, e sim à estratégia a partir da qual o sistema será construído. Sendo assim, estes requisitos são funções do conjunto de operações e mecanismos primitivos que podem ser mais facilmente construído ou satisfatórios para as restrições imposta pela arquitetura e "hardware" da máquina.

Andrews e Schneider (ANDREWS, 1982) extenderam estas considerações para as 3 classes, chegando a concluir que a nível de abstração, pode-se transformar um programa escrito usando mecanismos encontrados em linguagens de uma classe no mesmo programa usando mecanismos de outra classe sem alteração de desempenho. Entretanto devido ao fato das classes enfatizarem estilos diferentes de programação, programas escritos em linguagens de diferentes classes são melhor estruturados usando caminhos diferentes, ou seja, na simples conversão de uma classe para outra, o código convertido não representa necessariamente um programa bem estruturado. Isto depende dos tipos de facilidades apresentados pelas linguagens.

Em relação a sistemas distribuídos, a comparação entre as classes **orientadas a mensagem** e **chamada de procedimento remoto** (orientada a operação) envolve aspectos ligados a confiabilidade:

- na classe orientada a mensagens se tem uma noção de falhas de "SEND" e de "RECEIVE" (o tratamento das exceções nestes dois serviços é definido explicitamente e em separado); isto é importante em ambientes distribuídos onde uma falha parcial deve ser encarada como um evento normal;

- na segunda classe (chamada de procedimento remoto), um problema semântico é fundamental: uma chamada de procedimento deve sempre retornar. Garantir que uma chamada de procedimento sempre retorna é extremamente difícil em um ambiente distribuído, onde os computadores e o sistema de comunicações estão sujeitas a falhas. A linha

normalmente seguida nesta classe é o tratamento implícito, envolvendo as chamadas **semânticas de confiabilidade** (NELSON, 1981) e (SPECTOR, 1982). Em (NATATAJAN, 1985) são propostos mecanismos de tratamento explícito.

3.3. Concorrência e Tempo Real

Umas das principais razões para o desenvolvimento de programação concorrente é o seu uso como metodologia para decomposição de programas em atividades separadas. Cada atividade está relacionada com a execução de uma ou mais tarefas sequências. A sequência de execução destas tarefas não necessita ser conhecida pelo programador. Portanto, para programação concorrente as únicas restrições à sequência de execução são aquelas relacionadas com a cooperação (sincronização e comunicação) entre as tarefas.

Os programas concorrentes que necessitam prover resultados ou operações dentro de certas restrições de tempo são denominados programas para tempo real. Para cumprir estas restrições de tempo, os programas tempo real estão sujeitos ao controle do sequenciamento das operações e possivelmente das tarefas.

Geralmente, as linguagens que se dizem adequadas a programação tempo real implementam apenas mecanismos para **espera temporizada** ("delay") e **Tempo de espera** ("timeout"). É impossível garantir que programas concorrentes utilizando apenas estes 2 mecanismos, cumpram satisfatoriamente as exigências

envolvidas em aplicações tempo real (LEE, 1985). As restrições de tempo derivadas do planejamento do controle e das escalas de tempo estão associadas ao tratamento de eventos que ocorrem nestes sistemas, devem ser impostas a programas concorrentes. Para isto existem duas abordagens:

- A primeira é um abordagem convencional que consiste em descrever todas as tarefas de um programa concorrente ignorando as restrições de tempo. Uma vez correta a lógica do programa, o programador introduz as restrições de tempo através de estratégia de escalonamento (define o níveis de prioridades) (LEE, 1985);

- Uma outra forma de abordar é prover estruturas para especificar restrições de tempo na própria linguagem de programação. Tais estruturas permitem ao programador especificar o tempo de resposta desejado e tempos envolvidos em todos os "loops" ou trechos do programa (GLIGOR, 1983) e (LEE, 1985). Estas informações possibilitam ao compilador (ou outras ferramentas) verificar se as tarefas programadas podem se executar dentro das necessidades de tempo definidas.

Enquanto na primeira abordagem o programador é o responsável pela definição e implementação da estratégia de escalonamento de tarefas, na segunda, esta responsabilidade é transferida para ferramentas ou ainda para o escalonador que ativaria tarefas, usando as informações de tempo especificadas.

3.4. Paradigmas de Programação Distribuída

A estruturação de aplicações distribuídas pode ser facilitada pela utilização de um modelo. A estruturação segundo um modelo consiste em definir as entidades e a forma de interações entre estas, permitindo deste modo, a visualização de forma abstrata das várias ações (ou atividades) executando concorrentemente e os pontos de cooperação entre elas no programa distribuído.

Os modelos mais representativos apresentados na literatura e que serviram de base a outros modelos podem ser agrupados em:

- **Modelo servidor/cliente;**
- **Modelo orientado a objeto;**
- **Modelo de aplicação transacional;**
- **Modelo processos sequenciais comunicantes (CSP);** e
- **Modelo de atores ("actor").**

3.4.1. Modelo Cliente/Servidor

É um dos primeiros modelos propostos para sistemas distribuídos. No modelo os processos dividem-se em clientes (processos que solicitam ou necessitam de recursos compartilhados) e servidores (processos que implementam as operações sobre o recurso e por conseguinte fornecem os serviços). Podem existir múltiplos servidores provendo o mesmo serviço no sistema. Um processo servidor pode em determinado instante solicitar serviços (ser cliente) de outros servidores.

A principal aplicabilidade do modelo está no compartilhamento de recursos, tais como arquivos, impressão, etc.

3.4.2. Modelo Orientado a Objeto

O termo orientado a objetos evoluiu dos trabalhos realizados por Anita Jones (JONES, 1978) e Liskov (LISKOV, 1977) para estruturação de sistemas e da experiência com as linguagens Simula 67 e Smaltalk (GOLBERG, 1984). Neste modelo toda entidade do sistema é um objeto. Sendo assim as aplicações distribuídas são composta de um conjunto de objetos. Um objeto corresponde a uma entidade que possue um estado interno e um comportamento. O estado interno corresponde a uma memória local que é inacessível e indevassável por outros objetos, e o comportamento é definido por um repertório de ações de que o objeto dispõe para responder a demandas externas de operações e que são internas ao próprio objeto. Vários objetos podem ter os mesmos comportamentos característicos, sendo portanto conveniente a definição um único conjunto de operações que é igualmente satisfatório para os vários objetos. Estes vários objetos que compartilham um único conjunto de operações formam então as chamadas **classes**. Neste modelo é introduzido também o conceito de **herança** que permitem a criação de classes de hierarquia, desta forma todos os objetos são uma instância de uma classe, que por sua vez é uma subclasse de outra classe e assim por diante (BOOCH, 1986), (BALTER, 1986).

3.4.3. Modelo de Aplicação Transacional

Neste modelo, a construção de aplicações distribuídas esta baseada nas noções de objetos, processos e transações. O conceito de objeto é o mesmo do modelo anterior, com exceção que a granularidade de objetos é geralmente maior e que os objetos podem ser compartilhados entre processos concorrentes (BALTER, 1986). Um objeto corresponde ao encapsulamento de um conjunto de recursos e operações compartilhadas.

Podem ser distinguidas duas abordagens neste modelo; isto dependendo se os objetos são ativos ou passivos (BALTER, 1986):

- Um objeto ativo inclue um ou mais processos que estão totalmente confinados dentro do mesmo, e que executam pedidos de outros objetos ou encontram-se executando independentemente no "background";

- No modelo passivo, processos são externos para a objetos. Durante a execução os procedimentos referentes às operações de objetos manipulados pelo processo devem fazer parte do espaço de endereçamento deste últimos.

Neste modelo, uma transação constitui uma sequência de operações executadas sobre o objeto (ou no objeto). A transação deve representar a unidade de sincronização e também de recobrimento nas atualizações dos objetos. Os objetos ditos recobríveis constituem-se num suporte para a realização de transações atômicas. Em (LAMPSON, 1981) é feita uma síntese destas técnicas e mecanismos propostos na literatura e que permitem a construção de objetos recobríveis e transações

atômicas. Um exemplo notório deste modelo é a programação à Guardiões introduzida por Liskov no Sistema Argus (LISKOV, 1983).

Este modelo é satisfatório para aplicações tempo real em que as restrições de tempo não são muito severas e onde a consistência das informações no sistema é muito importante.

3.4.4. Modelo Processos Sequênciais Comunicantes

O **modelo CSP** ("Communicating Sequential Processses") (HOARE, 1978) tem servido de fonte de inspiração para diversas propostas de estruturação de programas distribuídos. A entidade de concorrência do modelo é o processo. Todas as declarações de processos estão no início do programa, não existindo comandos que suportem à criação dinâmica de processos.

A execução concorrente de processos é expressa pelo comando **"parallel"**, baseado no comando "parbegin" (DISJKSTRA, 1968). Todos os processos incluídos num comando "parallel" iniciam-se simultaneamente, sendo que o comando só se encerra quando todos os processos estiverem terminados.

Não existe mecanismo de comunicação baseado em memória compartilhada (variáveis globais). A única forma de comunicação entre processos e através dos comandos sincronos **"output"** (correspondente a um envio) e **"input"** (correspondente a uma recepção).

A nomeação dos comandos "input" e "output" são diretas, isto é, o processo que executa o comando "input" especifica a origem. Da mesma forma, o processo que executa o comando "output" especifica o destino. Este tipo de nomeação restringe a flexibilidade. A comunicação ocorre quando ambos os processos encontram-se sincronizados (Rendez-vous). O comando "guard" é a única estrutura de controle condicional, e também o único meio de introduzir e controlar ações não determínistas.

3.4.5. Modelo de Atores

O modelo de Atores corresponde a abstração e mecanismos de comunicação implementados no sistema distribuído CHORUS (ZIMMERMANN, 1984).

Em Chorus, um sistema distribuído é composto de um conjunto de nós (estações) interconectados. Em cada nó, o processamento é executado por entidades ativas denominadas "Actores" (atores). O conceito de ator corresponde a noção de processo. Um ator é criado a partir de um **modelo de ator** (tipo).

A execução de uma aplicação distribuída consiste de passos de comunicação ("communication steps") e passos de processamentos ("processing steps"). Um **passo de processamento** é a unidade elementar de processamento executado por um ator a partir da recepção de uma mensagem concluída com a transmissões de "n" outras mensagens (n =0). Um ator é portanto composto de um ou mais passos de processamento que devem ser executados no mesmo nó.

Todas as comunicações entre atores são feitas por troca de mensagem. De maneira a comunicar-se, no mínimo 2 (dois) portos são necessários um para enviar a mensagem e um (ou mais) para recebe-la. É permitido também neste modelo, portos bidirecionais (o mesmo porto é usado para enviar e receber mensagens). Assim, uma mensagem é enviada por um porto do ator emissor (porto de origem) para um (ou vários) porto(s) do(s) ator(es) de recepção (porto de destino). A transmissão de mensagem é feita através de envios assíncronos. Sendo, a semântica independente da localização, isto é, comunicação local e remota são feitas da mesma forma. O sistema fornece facilidades para criação dinâmica de atores, portos e mensagens.

3.4.6. Considerações gerais

Os modelos citados acima dão a medida da profusão de modelos descritos na literatura e da dificuldade de se estabelecer limites entre estes modelos.

Embora alguns dos modelos sejam melhor adaptados a algumas classes de aplicação, nenhum é considerado especializado para uma classe de aplicação ou suficientemente geral para ser adotado como modelo de referência para construção de qualquer tipo de aplicação distribuída (DESWARTE, 1987).

3.5. Paradigma de Programação Suportado pelo Núcleo de Sistema Operacional Distribuído

O domínio da complexidade na programação distribuída depende em grande parte dos chamados **paradigmas de programação.** Estes modelos devem fornecer uma disciplina para a concepção e a estruturação de sistemas distribuídos. A eficiência neste objetivo é conseguida através de uma linguagem projetada para conter o paradigma e o suporte fornecido pelo ambiente de tempo de execução.

No caso de sistemas complexos a abordagem clássica para o projeto do software é o princípio da decomposição modular ("programming-in-the-small vs programming-in-the-large") apresentado em (DE REMER, 76). O paradigma de programação proposto incorpora este princípio e nestas condições, então, tem-se um sistema construído a partir de um conjunto de **módulos** (entidade elementar de configuração). O princípio de decomposição modular impõe então a separação da programação dos módulos de configuração do sistema (programação em larga escala); isto permite uma grande flexibilidade: um módulo pode participar de diferentes configurações (reutilização de software), uma configuração pode conter várias instâncias de um módulo e mudanças "on-line" de configurações são possíveis nos sistemas (sistemas evolutivos).

Módulos fornecem interfaces bem definidas (portos) para outros módulos e apresentam internamente apenas referências a objetos locais; isto garante a reutilização e permite que o projeto, a programação, a compilação e o teste do módulo sejam

independentes da configuração. Estas particularidades são importantes porque permitem separar o comportamento abstrato do módulo de sua implementação.

Em tempo de execução, a noção de modularidade é substituída pela noção de concorrência. Assim sendo um programa distribuído é considerado como **um conjunto de tarefas** cooperantes em execução competindo pelos (ou compartilhando dos) recursos do sistema. Uma tarefa é a entidade elementar de concorrência, correspondendo ao conceito de processo definidos em vários modelo ("Guardian","CSP", etc). As tarefas são criadas durante a instanciação de um módulo e o número de tarefas contidas em uma instância é definido quando da programação do módulo.

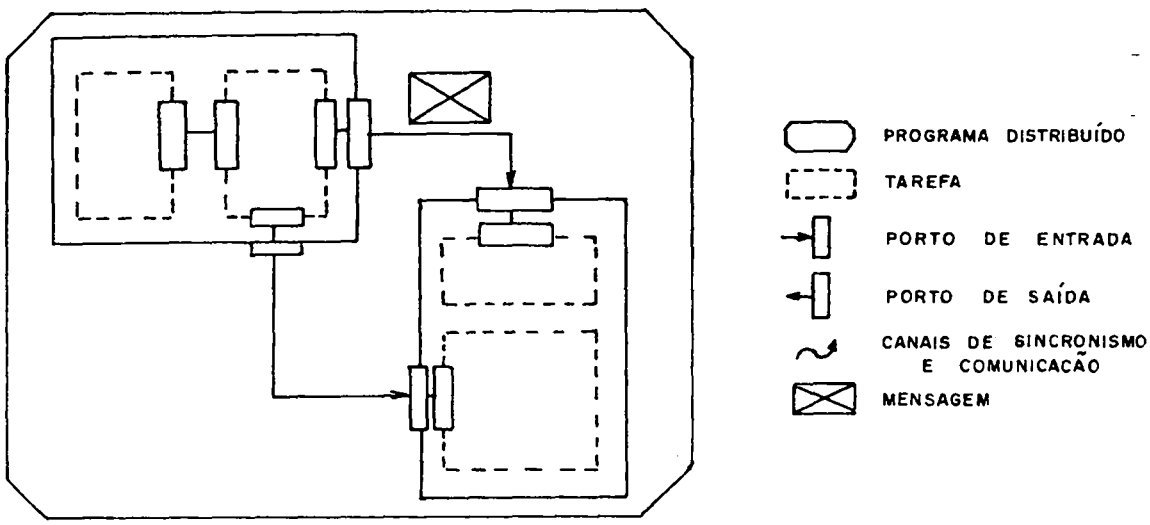


figura 3.1. - Paradigma de Programação Distribuída

A cooperação entre tarefas é feita por troca de mensagens, não existindo outra forma de comunicação e sincronização entre tarefas. Mensagens é então a entidade elementar para o transporte de informação. Tarefas comunicam-se por transmissões

síncronas e assíncronas através de interfaces bem definidas denominadas **porto.** A noção de porto é classicamente introduzida no modelo troca de mensagens como solução aos problemas de endereçamento e tratamento de mensagens associadas às tarefas.

No modelo utilizado, tarefas enviam mensagens através de portos de saída e recebem via porto de entrada. Os portos de saída são conectados aos portos de entrada em tempo de configuração, formando então, **canais de sincronismo e comunicação** entre tarefas.

O módulo, por sua vez, encapsula uma ou mais tarefas. A **composição multitarefa** de módulo não implica em compartilhamento de memória (acesso a objetos globais) por partes de suas tarefas, sendo no entanto possível o acesso rápido de objetos (tais como tabelas, listas, etc) através da passagem de ponteiros entre tarefas internas ao módulo. Uma outra característica importante para utilização de módulos multitarefas é a redução das informações armazenadas para a configuração do sistema.

Uma representação das entidades que compõem o paradigma de programação é mostrada na figura 3.1.

3.6. Linguagem de Implementação de Sistemas

A construção de um programa distribuído segundo o paradigma discutido no item anterior, necessita de ferramentas, que permitam a descrição dos vários elementos que compõem o programa e a devida instalação destes no sistema; para isto é definida a

Linguagem de Implementação de Sistemas (LIS). Segundo o princípio de decomposição modular, a LIS se apresenta composta de uma **Linguagem de Componentes Elementares (LINCE)** e uma **Linguagem de Configuração de Sistemas (LINCS).**

A linguagem LINCE é construída a partir de uma linguagem base (no caso o PASCAL), acrescida de extensões que permitem implementar o paradigma proposto. Desta forma, a linguagem é composta das seguintes declarações e instruções:

- definição de módulo;
- declarações de tarefas;
- declarações de portos;
- declaraçoes de importação de objetos;
- declarações de ligação de portos;
- instruções de comunicações;
- instruções de "loop";
- instruções de reconfiguração; e
- procedimentos-padrão do usuário.

A composição de um sistema (programa distribuído) é realizada através da linguagem LINCS que permite definir o conjunto de tipos módulo a partir do qual o sistema será constuído e oferece operações tais como: carregar/remover tipo módulo, criar/destruir instância e ligar/religar/desligar módulos. Estas operações são executadas separadamente (indèpendentes entre si), aumentando a flexibilidade na construção do sistema, permitindo carga e ligações incrementais que são úteis para a configuração dinâmica. A configuração pode ser hierárquica quando composta de subconfigurações. Neste caso

a linguagem de configuração apresenta construções de estruturação que permitem a construção de módulos a partir de módulos mais elementares. O nosso modelo de programação em larga escala segue a abordagem proposta no sistema CONIC (KRAMER, 1983).

As linguagens LINCS e LINCE são abordadas com mais detalhes em (SOUZA, 1988) e (SILVA, 1988).

3.7. Conclusão

Neste capítulo foi abordado alguns aspectos da programação concorrente e as restrições adicionais impostas quando em tempo real. Também foram apresentados alguns paradigma de programação distribuída usualmente citados na literatura. Baseado nestes estudos foram definidas então as entidades que compõem o paradigma de programação distribuída suportado pelo núcleo de tempo real, objeto desta dissertação.

CAPÍTULO 4

NÚCLEO TEMPO REAL

4.1. Introdução

Um programa distribuído é composto de vários módulos (ou tarefas) que se executam de forma concorrente, possivelmente em vários nós (estações), cooperando entre si para a realização de uma atividade. O responsável pela implantação do ambiente multitarefas que executa a aplicação distribuída é o Núcleo Tempo Real (NTR), objetivo principal desta dissertação. Este é constituído por um conjunto de primitivas acessadas em grande parte por tarefas em tempo de execução. Estas primitivas, portanto, devem estar disponíveis ao programador de aplicação através da Linguagem de Implementação de Sistema (LIS).

Cada nó do sistema, para suportar a execução de programas distribuídos deve conter uma cópia do NTR. O núcleo não toma decisões políticas quanto à alocação de recursos. As políticas são definidas a partir dos Serviços de Sistema Operacional Distribuído, mais precisamente do configurador.

Este núcleo foi concebido como parte do Ambiente de Desenvolvimento e Execução de Software (ADES) que corresponde a um conjunto de ferramentas cuja finalidade é a construção de software distribuído para aplicações em tempo real.

Neste capítulo será objeto de discussão as várias unidades funcionais do NTR, que fazem deste um Núcleo ("Kernel") de Sistema Operacional Distribuído.

4.2. Arquitetura do Sistema

O ambiente (ADES) consiste de **um conjunto de estações** (compatíveis IBM-PC) interligados por uma rede local. O conjunto de estações é composto de **estações de execução** e de uma **estação de trabalho** (ver figura 4.1.).

O desenvolvimento do software é realizado na estação de trabalho, utilizando-se de um conjunto de ferramentas que atuam em diferentes fases do ciclo de vida do software. Este, quando desenvolvido e testado, é então transferido às estações de execução para a operação do sistema total. Na estação de trabalho a abordagem utilizada para implementar o SOD é a de **Sistema Operacional de Rede** (item 2.5.), implicando na manutenção do sistema operacional hospedeiro. Portanto a estação é composta do sistema operacional hospedeiro, do Núcleo Tempo Real, e por último dos utilitários e ferramentas. Os serviços de Sistema Operacional Distribuídos (que por si só é um programa distribuído), são formados na estação de trabalho por módulos que compõem o gerenciamento local e no módulo

interfaceador com o sistema operacional hospedeiro. O gerenciador local da estação de trabalho é grandemente determinado pelo gerenciamento de configuração, envolvendo: transferência de arquivos de códigos, instanciação de componentes, monitoração do sistema, mudanças de configuração, etc. O interfaceador torna sequenciais as chamadas de funções do sistema operacional hospedeiro.

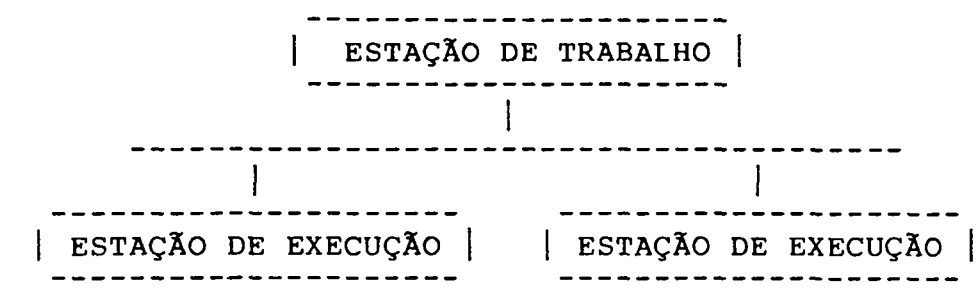


figura 4.1. - Arquitetura do Sistema

A estrutura da estação de execução é similar a da estação de trabalho, diferindo nas funções operacionais de forma a atender os serviços específicos de cada estação. Neste caso para atender a estes objetivos, o sistema operacional distribuído é configurado incorporando uma versão mais simples dos gererenciadores, cujas funções se restrigem as operações de controle de execução das instâncias, controle de erro e suporte de acesso remoto a memória. Portanto, nestas estações de execução, um sistema operacional hospedeiro não necessita estar presente, correspondendo assim a uma abordagem de estruturação de **sistema operacional distribuído** (item 2.5.).

4.3. Principais Funções do Núcleo Tempo Real

As principais funções do núcleo tempo real relacionam-se com:

- Suporte de Gerenciamento Local;
- Gerenciamento de Tarefa;
- Comunicação e Sincronização entre tarefas, inclusive comunicação remota;
- Gerenciamento de memória (para variáveis dinâmicas e blocos de memória);
- Tratamento de Interrupção; e
- Temporização.

Estas funções podem ser classificadas em 3 grupos distintos, de acordo com o suporte provido. Assim, entre as funções acima encontram-se funções fornecendo:

- suporte para as abstrações definidas na LINCE e na LINCS;
- suporte para serviços de sistema;
- suporte para comunicação e execução multitarefa.

Nas próximas seções serão descritos os serviços do núcleo relacionados com as unidades funcionais citadas acima.

4.4. Suporte de Gerenciamento Local

O suporte de gerenciamento local fornecido pelo núcleo está intimamente ligado às atividades de gerenciamento de configuração no sistema (serviços de sistema). As primitivas do

núcleo neste caso, estão associadas a operações envolvendo **tipo módulos, instâncias e estabelecimento de canais de comunicação.**

4.4.1. Operações Sobre o Tipo Módulo

O conjunto de primitivas relacionadas com tipo módulo são: **carregar, habilitar** e **remover** tipo módulo.

A operação **carregar tipo módulo** suporta o mapeamento da definição abstrata de tipo módulo para uma estrutura física e cria uma representação do tipo módulo no núcleo. O mapeamento para uma estrutura física corresponde a alocação de um segmento de memória ao código da estação. A representação no núcleo da definição abstrata de tipo módulo é feita por uma estrutura de dados, denominada Bloco de Controle de Tipo Módulo (BCTM).

A operação de carga de um tipo módulo é acompanhada pela criação de um **"lock"** que previne a criação de instâncias do tipo módulo antes de completada a carga. A operação **"habilita tipo módulo"** deve liberar este **"lock"** permitindo então, a instanciacão do tipo módulo.

A operação **remover tipo módulo** desaloca o segmento de código e destroi a representação (BCTM) na estrutura de dados do núcleo.

4.4.2. Operações sobre a Instância de Módulo

O núcleo fornece um conjunto de operações relacionadas com a instanciação de tipos módulo: **criar, iniciar, parar, suspender e destruir instância.**

A operação **criar instância** suporta a definição de um contexto de execução para o tipo módulo e também a criação de uma representação da instância no núcleo. A instância de um tipo de módulo é representada, por uma estrutura de dados, denominada de Blóco de Controle de Instância (BCI). A definição do contexto envolve também a alocação dos segmentos de dados e de "stack" para a instância. A operação **criar instância** define ainda toda a estrutura de dados de tempo de execução da instância: criações de tarefas (definição dos contextos e de descritores de tarefas) e de portos pertencentes a instância do tipo módulo.

Ao final da operação **criar instância**, todas as tarefas pertencentes a instância possuem os recursos necessários a execução alocados, porém ainda não estão concorrendo ao processador. A operação **iniciar instância** faz com que todas as tarefas da instância tornem-se **prontas** para concorrer ao processador. A separação das operações de **criar** e de **iniciar** instância é provida de maneira a distinguir claramente as operações envolvidas no processo de configuração.

A operação **parar instância** faz com que todas as tarefas da instância retornem a condição anterior à inicialização da instância. A execução da operação **parar instância**, não implica

em restrições a execução, novamente, da operação **iniciar instância.**

Suspender instância é a operação executada a partir de um gerenciador de erro ou de um "handler" de exceção, com o objetivo de suspender todas as tarefas de uma instância. Esta operação deve ser executada quando da ocorrência de erro ou exceções ("overflow" ou divisão por zero), na execução de uma das tarefas da instância. A estratégia de suspender todas as tarefas de uma instância foi adotada devido estas estarem fortemente relacionadas.

A operação **destruir instância** só pode ser executada após a operação **parar instância.** A destruição envolve a desalocação dos segmentos de dados e de "stack" e também a destruição de todas as estruturas relativas a instância (BCI e descritores de tarefa e porto).

4.4.3. Canal de Comunicação

O estabelecimento dos canais de comunicação e sincronismo (conexão lógica) entre módulos (na configuração) e entre tarefas (ligações intra-módulos realizadas quando da instanciação do tipo módulo) é suportada pelo conjunto de primitivas de ligação: **ligar, religar e desligar portos.**

A operação **ligar portos** estabelece a conexão lógica entre um porto de saída e um porto de entrada. Esta conexão é representada no NTR através de uma estrutura de ligação (EST_LIG).

A operação **religar portos**, altera uma conexão lógica estabelecida pela operação anterior. Esta alteração modifica os campos da EST_LIG. A operação **desligar portos** encerra a conexão lógica entre um porto de saída e um porto de entrada através da destruição da EST_LIG. A operação **religar portos** é provida para evitar que alterar uma conexão lógica envolva a execução consecutiva das operações de **desligar** e **ligar** portos.

4.5. Gerenciamento de Tarefa (Escalonamento)

O Núcleo Tempo Real apresenta um escalonador **"event driven"** que, portanto, segue uma política de **preempção** (**"preempt"**) prioridade para ocupação do processador. Normalmente, nesta política designam-se prioridades mais altas às tarefas cujas operações estão sujeitas a restrições severas de tempo. Embora este tipo de estratégia seja vista por muitos como mais apropriada para o processamento em tempo real, é também notório que esta estratégia não leva em consideração à importância momentânia da execução da tarefa, ou seja, não considera a importância de uma tarefa em relação ao conjunto de tarefas em determinado instante (LEE, 1985).

Outras estratégias que levam em consideração **"deadline"** (tempo de ativação) além de prioridades, se mostram mais adequadas diantes das necessidades de tempo real. O "deadline" corresponde ao valor limite exigido para que a tarefa (re)inicie a execução satisfazendo assim as restrições de tempo. Estas estratégias apresentam a vantagem de introduzir explicitamente restrições de tempo no escalonamento.

A adoção da estratégia de "preempt" por prioridade no núcleo em questão se deve à simplicidade da mesma e os resultados apontados na literatura como satisfatórios para sistemas relativamente simples.

4.6. Comunicação e Sincronização

A arquitetura de um Sistema Operacional Distribuído está centrada sobre mecanismo de comunicação e sincronização entre tarefas (IPC) . Um modelo bem definido de IPC (embutido num conjunto de construções de linguagem) fornece uma diciplina para o desenvolvimento de programas concorrentes e distribuídos que sejam eficientes e estruturados (WATSON, 1981).

O ambiente de execução fornecido pelo núcleo tem a comunicação e sincronização uniformes e independentes da distribuição das tarefas no sistema. O núcleo utiliza o modelo **troca de mensagens** na implementação das primitivas de comunicação e sincronização, e trata os problemas de endereçamento de tarefas através de portos.

A comunicação e sincronização entre tarefas é feita através de operações de envios e recepções de mensagens. A forma com que se dá o acoplamento comunicação/sincronismo determina diferentes operações de envios e recepções de mensagens.

4.6.1 Primitivas de Comunicação

Geralmente em sistemas tempo real, as razões que levam as tarefas a enviarem mensagens são (LEE, 1985):

- transferência de dados ou sinais a uma ou mais tarefas;
- necessidade de sincronização com uma ou mais tarefas;
- solicitação de serviço a uma tarefa.

Existem ainda muitas discussões com relação a semântica apropriada a ser utilizada em cada um dos casos acima. Porém, na literatura, destacam-se três tipos básicos de primitivas de envio, propostos com o intuito de englobar todas estas necessidades (LISKOV, 1979),(SCOTT, 1983),(RASHID, 1981):

- **Envio não bloqueante (SEND_NB);**
- **Envio Síncrono, aguardando recepção (SEND_B);**
- **Envio Síncrono, aguardando resposta (SEND_WAIT).**

Os envios SEND_NB e SEND_B caracterizam canais de comunicação unidirecionais. Por outro lado, a primitiva SEND_WAIT condiciona a conclusão da comunicação com a chegada de uma mensagem de resposta, determinando assim, um canal de comunicacão bidirecional.

As primitivas de recepção são normalmentes divididas em **bloqueantes e não bloqueantes.** Na **recepção bloqueante** a tarefa é colocada em espera até a chegada da mensagem. Na **recepção não bloqueante** a tarefa continua se executando mesmo que nenhuma mensagem tenha sido recebida. Alguns sistemas introduzem recepção em grupo de portos: recepções condicionais ou

seletivas. Estas recepções caracterizam uma certa forma de concorrência no recebimento de mensagens. Assim quando da execução de um **receive** seletivo (sobre um grupo de porto) somente uma mensagem é recebida pela tarefa, segundo certas condições.

Na próxima seção serão especificadas as operações de comunicação fornecidas pelo NTR.

4.6.1.1. Envio Não Bloqueante (SEND_NB)

Este tipo de envio não necessariamente sincroniza a tarefa emissora com a tarefa receptora, desta forma se no instante do envio de uma mensagem a tarefa receptora não se encontra em condição de recebe-la, a mensagem é colocada em um "buffer" até que ocorra a recepção. A consequencia disto é que a mensagem recebida não necessariamente contém informações atuais da tarefa emissora.

Desde que a disponibilidade de "buffer" é limitada, surgem duas questões fundamentais: a maneira de alocar (estaticamente ou dinamicamente) estes buffers e o tratamento a ser dado em casos de não disponibilidade de "buffer".

Quanto a primeira questão, optou-se pela alocação dinâmica destes buffers vizando uma maior flexibilidade e uma otimização na utilização de recursos (memória); embora o tempo de alocação possa contribuir com uma piora no desempenho global do envio.

Entre as opções para a segunda questão encontram-se: suspender a tarefa emissora até que o "buffer" esteja disponível ou ainda de sinalizar à tarefa de que não há "buffer" disponível. No núcleo considerado decidiu-se pela não suspensão da tarefa emissora e com isso são feitas várias considerações de maneira a diminuir a possibilidade de ocorrência de indisponibilidade de "buffers":

- O programador pode especificar o número máximo de mensagens relevantes à tarefa de recepção. Em outras palavras, o número máximo de "buffers" que os envios endereçados a uma tarefa (recepção) podem alocar;

- No caso do número máximo de mensagem ser atingido ou de não existir a disponibilidade de "buffer" para alocação a nova mensagem ocupará o "buffer" da mensagem mais antiga. A adoção desta estratégia faz que uma mensagem transmitida através do envio assíncrono em alguns casos não seja recebida;

- O programador (usuário) pode especificar na configuração, a área total disponível para alocação dinâmica de buffers, sendo desta forma, responsável pelo fator de aproveitamento desta área em tempo de execução.

As duas primeiras considerações satisfazem plenamente às exigências de tempo real, desde que as informações (mensagens) implicitamente estarão associadas a um tempo de validade e as mais relevantes serão as mensagens mais recentes.

4.6.1.2. Envio Síncrono aguardando recepção (SEND_B)

Este tipo de envio sincroniza a tarefa emissora com a recepção da mensagem por parte da tarefa receptora. As principais características deste tipo de envio são:

- Não existe necessidade de "buffer" entre o envio e a recepção da mensagem, desde que durante estas operações as tarefas encontram-se sincronizadas;

- Se no instante do envio a tarefa receptora não se encontra em condições de receber a mensagem, a tarefa emissora deverá ser bloqueada até o momento da recepção;

- O sincronismo está condicionado somente à recepção da mensagem. A confirmação da recepção da mensagem não envolve uma mensagem de resposta, isto é feito automaticamente pelo NTR, liberando a tarefa emissora.

Mecanismos devem ser providos para evitar que bloqueios indefinidos da tarefa emissora. As técnicas de "timeout" são clássicas nestes casos. Se até o esgotamento do "timeout" a mensagem não for recebida, o envio síncrono é então abortado e ma indicação de exceção (esgotamento de "timeout") é sinalizada na tarefa emissora. O tratamento correspondente à condição de exceção faz parte da programação da tarefa emissora.

4.6.1.3. Envio Síncrono aguardando resposta (SEND_WAIT)

Este envio além de sincronizar a tarefa emissora com a recepção da mensagem por parte da tarefa receptora, mantém o sincronismo entre as tarefas até a chegada de uma mensagem resposta à tarefa emissora. As principais características deste envio são:

- a combinação deste envio com as operações de recepção e resposta formam um canal de comunicação bidirecional, com características idênticas ao **chamada de procedimento** e **"Rendez-vous"**;

- Não existe necessidade de "buffer" entre os instantes de envio e recepção da mensagem e também entre a transmissão e recepção da resposta, porque em ambos instantes as tarefas estarão sincronizadas;

- Se no momento do envio a tarefa receptora não se encontra em condição de receber a mensagem a tarefa é bloqueada e permanece nesta condição até receber a mensagem resposta. Após a recepção da resposta ambas tarefas prosseguem em concorrência.

O mesmo mecanismo de "timeout" do envio síncrono é provido no SEND_WAIT para evitar que, em alguns casos a tarefa fique indefinidamente na espera da mensagem resposta. Também, neste caso, o envio é abortado quando do esgotamento do "timeout" e uma indicação de exceção é sinalizada à tarefa emissora. O tratamento a ser dado a condição de exceção também faz parte da programação da tarefa emissora.

4.6.1.4. Recepção Bloqueante

Este tipo de recepção, a princípio, condiciona o prosseguimento da execução da tarefa com o recebimento da mensagem. No caso da mensagem não estar disponível quando da execução da primitiva a tarefa é bloqueada até a chegada da mesma. Para evitar a espera indefinida, é provido mecanismo de "timeout". Desta forma, após a execução da recepção, a tarefa pode retornar do estado de espera ou pelo esgotamento do "timeout" ou pela chegada de uma mensagem. Um caso particular é a recepção com "timeout" igual a zero (0). Neste caso a mensagem somente é recebida, se no momento da execução de uma operação de recepção a mesma encontra-se disponível (**recepção não bloqueante**). O tratamento do esgotamento do "timeout" faz parte da programação da tarefa.

No caso oposto, em que no momento do envio a tarefa receptora não encontra-se aguardando pela mensagem, dependendo da forma de envio poderá ocorrer os seguintes casos:

- Se a mensagem foi enviada por uma primitiva não bloqueante; neste caso a mensagem deve ser colocada em um "buffer" até que se dê a sua recepção. Como foi discutido anteriormente no envio não bloqueante, pode ser limitada a capacidade de recepção de uma tarefa em uma comunicação (assíncrona); isto restringe o número máximo de mensagens colocadas em "buffers" endereçadas a uma tarefa;

- Se a mensagem foi enviada por umas das operações de envio bloqueante (SEND_B ou SEND_WAIT), a tarefa emissora nestas

condições ficará bloqueada, sem a necessidade que a mensagem seja colocada em um "buffer" entre os instantes de envio e de recepção. Por outro lado em comunicação VÁRIOS-PARA-UM o número máximo de mensagens endereçados a tarefa receptora corresponde ao número de emissores. Neste caso, estes emissores são bloqueados e organizados em uma fila até que ocorra a recepção de suas respectivas mensagens.

Portanto, pode existir uma fila contendo mensagens endereçadas à tarefa, ainda não recebidas. A organização das mensagens contidas em uma fila é por prioridade, sendo que as de mesma prioridade encontram-se na fila por ordem de chegada.

4.6.1.5. Recepção Seletiva

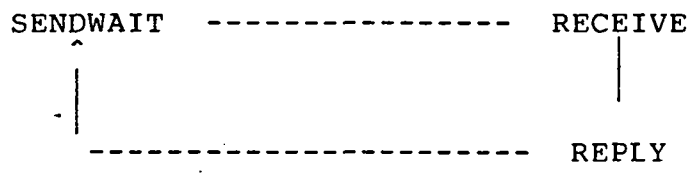
A recepção seletiva permite a uma tarefa aguardar concorrentemente várias mensagens, embora efetivamente somente uma mensagem seja recebida. A recepção de uma mensagem pode ser opcionalmente precedida de um comando GUARDA, correspodente a uma variável booleana que habilita ou não a recepção da mensagem.

A recepção seletiva também pode ser bloqueante ou não. No caso da recepção seletiva bloqueante, quando nenhuma das mensagens está disponível, a tarefa é colocada em espera, permanecendo neste estado até o esgotamento do "timeout" ou a chegada da primeira mensagem. No caso de recepção seletiva não bloqueante, somente é recebida uma mensagem, se no momento de execução desta operação houver mensagem disponível.

As considerações levantadas para as recepções simples, em relação a filas de mensagens também são válidas para a recepção seletiva.

4.1.1.6. Resposta (REPLY)

A operação resposta (REPLY) corresponde a um envio não bloqueante e deve ser utilizado em comunicações síncronas bidirecionais, como mostrado abaixo:



4.6.2. Porto de Saída e Porto de Entrada

As abstrações de Porto de Saída e Porto de Entrada são introduzidas; de maneira a separar a programação de componentes da programação da configuração do sistema, ou seja módulos (ou tarefas) só referenciam a objetos locais: seus portos. As definições de porto estão associadas a um tipo. Um tipo de porto corresponde ao tipo de mensagem (variáveis declaradas nas tarefas) enviada ou recebida através do porto. A verificação de compatibilidade de tipos dos portos em ligações inter-módulos estará a cargo da linguagem LINCS.

4.6.3. Tipos de Endereçamentos

A comunicação e sincronização entre tarefas são constituidas a partir de **conexões lógicas** de dois ou mais portos. As primitivas de comunicação definidas no item 4.6.1., quando aplicadas a conexões lógicas baseadas em portos, permitem os seguintes tipos de endereçamentos: **UM-para-UM, UM-para-VÁRIOS** e **VÁRIOS-para-UM.** As dificuldades encontradas em adequar a semântica da implementação de envios síncronos (SEND_B e SEND_WAIT), restringe o endereçamento UM-para-VÁRIOS somente para o envio assíncrono.

4.6.4. Mensagem

A nomeação de tarefas através de portos tipados permite a declaração de mensagens na programação da tarefa, da mesma forma que variáveis locais (espaço de endereçamento das tarefas). A **tipagem** e a estrutura de dados (descritores de portos), ambas associadas a noção de porto, garante uma compatibilidade na comunicação das variáveis de envio com as recepção e também a manutenção de informações como tamanho de mensagem, endereço, destino, etc. Porém uma das desvantagens da utilização de portos **tipados** é a restrição do envio (ou recepção) de apenas um tipo de mensagem.

Quando da execução de envio não bloqueante, em que variáveis necessitam ser alocadas dinamicamente de maneira a salvar temporariamente a mensagem, existe a necessidade de uma estrutura suplementar para manter as informações referentes à

variável alocada ("buffer"). Neste caso, mensagens ("buffers" alocados) passam a ser entidades diretamente manipuladas e gerenciadas pelo núcleo.

4.6.5. Comunicação Remota

O núcleo tempo real deve permitir a comunicação e o sincronismo entre tarefas de maneira uniforme, independente da distribuição destas no sistema. A extensão do IPC de forma transparente sobre a rede de computadores é feita classicamente usando "tarefas (ou módulos) servidoras de rede" (RASHID, 1981). Nestas condições, os requisitos colocados a um servidor de rede são: fornecer alguma forma de comunicação entre estações e acomodar as semânticas das primitivas de IPC (definidas sobre bases locais) quando da comunicação remota.

A vantagem da utilização de **tarefas servidoras de rede** é o encapsulamento das interfaces de acesso a rede em uma tarefa, tornando assim a portabilidade restrita a uma tarefa. Em outras palavras, a utilização do núcleo em ambientes diferentes, envolve a modificação de uma tarefa.

4.7. Gerenciamento de Memória

O núcleo provê o gerenciamento de memória em duas áreas distintas. A primeira área encontra-se no próprio segmento de dados do núcleo e é utilizado para alocação de variáveis representando os objetos do núcleo (blocos de controle, descri ores,

etc) e buffers de mensagens assíncronas. A segunda área encontra-se definida externamente ao núcleo e é disponível à alocação de segmentos (dados, código e "stack") em carregamentos e instanciações de tipos módulo.

Os gerenciamentos das áreas apresentam as seguintes características:

- As solicitações para alocações de segmentos (áreas externas) são feitas na construção do sistema (configuração estática); eventualmente serão solicitadas novas alocações em tempo de execução (configuração dinâmica);

- As variáveis alocadas pelo gerenciador de área interna ("heap") podem fazer parte da configuração (blocos de controle, descritores de tarefas, portos, etc) e geralmente permanecem alocadas durante o tempo de execução ou são temporárias (descritores e "buffer" de mensagens), sendo alocadas e desalocadas durante o tempo de execução.

Dentre as estratégias de alocação de memória ("best-fit", "first-fit" e "worst-fit") a "first-fit", que consiste em alocar o primeiro bloco de memória com disponibilidade igual ou superior ao solicitado é tida como a mais apropriada para o gerenciamento de memória tempo real. Portanto, ambos os gerenciamentos utilizam esta estratégia, adicionandas das seguintes particularidades:

- A alocação e compactação de blocos de memória adjacentes serão executadas por uma única função. Os blocos de

memória adjacentes livres serão compactados de acordo com as necessidades;

- A função de desalocação resumi-se em apenas uma indicação de bloco de memória livre.

As particularidades citadas visam simplificar o algoritmo de compactação restrigindo seu uso aos momentos de necessidade.

4.8. Tratamento de Interrupção

Várias linguagens de programação integram a manipulação de interrupções com a semântica do modelo **troca de mensagem**. Neste caso, a ocorrência de uma interrupção gera uma mensagem, que deve ser endereçada a um porto, que por sua vez ativa a tarefa que deve tratar o evento associado com a interrupção. Esta integração é consistente em relação aos princípios de troca de mensagem, onde o estado de uma tarefa só deve ser alterado através de mensagens.

Entretanto, por razões de eficiência preferiu-se sinalizar diretamente a tarefa quando da ocorrência de interrupção. Isto se justifica se considerado que as tarefas associadas com eventos externos (possivelmente não determinístas) têm prioridade atribuída ao nível sistema (nível mais alto); desta forma nada mais lógico que na ocorrência da interrupção a tarefa concorra ao processador em tempo mínimo.

4.9. Temporização

A função de temporização tem por objetivo prover uma base de tempo local, através da atualização periódica do relógio tempo real.

O gerenciamento de tarefas em espera temporizada ou comunicação síncronas temporizadas é feito segundo este relógio tempo real. Desta forma, periodicamente é verificada na fila a existência de tarefas com "timeout" esgotado. Nos casos que o esgotamento do "timeout" corresponde a uma situação de exceção (por exemplo: envio bloqueante) a operação é abortada é a tarefa é colocada na fila de pronta. Neste casos o tratamento dado a exceção faz parte da programação da tarefa.

4.10. Conclusão

Neste capítulo apresentaram se as funcionalidades do núcleo tempo real. Foram discutidos vários aspectos relacionados com os serviços fornecidos pelo núcleo que visam um suporte para a execução de software distribuído em tempo real. No próximo capítulo será discutida a implementação destes serviços e dados referentes a resultados desta implementação.

CAPÍTULO 5

FUNÇÕES DO NÚCLEO TEMPO REAL

5.1. Introdução

Este capítulo é dedicado inicialmente aos detalhes da implementação do núcleo, sendo portanto apresentadas as funções que executam os serviços introduzidos no capítulo anterior.

É objeto também deste capítulo uma discussão sobre os resultados obtidos. O desempenho do núcleo é medido através de valores conseguidos em testes que são descritos. Este capítulo se completa com o apêndice A, onde é descrito um exemplo de aplicação distribuída que foi concebida para testar a funcionalidade do núcleo, e também com o apêndice B, onde é descrito as principais funções do núcleo que estão disponíveis ao usuário, através da linguagem de LINCE.

5.2. Mapeamento de Memória do Núcleo Tempo Real

O mapeamento de memória do núcleo tempo real é composto dos **segmentos de códigos, de dados** e **de "stack"**, conforme representado na figura 5.1. O **segmento de código** contém os corpos das funções correspondentes aos serviços providos pelo núcleo e a função de iniciação do NTR. Esta última é responsável pela definição do mapeamento de memória, designando valores aos segmentos, e também pelas atribuições de valores iniciais as variáveis do núcleo. No **segmento de dados** são mantidas as variáveis estáticas e dinâmicas. A área reservada a alocação de variáveis dinâmicas é gerenciada por funções do núcleo, sendo que somente as variáveis criadas dinâmicamente durante a execução das funções do núcleo serão alocadas nesta área. O **segmento de "stack"** do NTR é utilizado somente pela função de iniciação pois, uma vez instalado o ambiente multi-tarefas, as funções do NTR utilizam o segmento de "stack" da tarefa ativa.

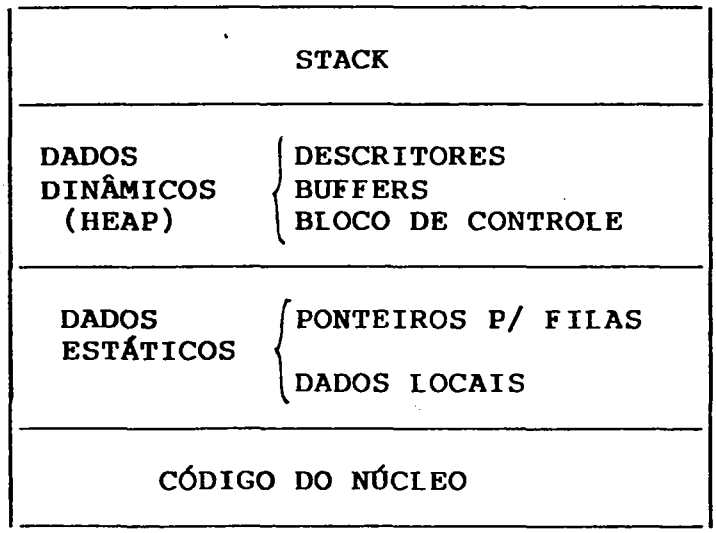


figura 5.1 - Mapeamento de memória do núcleo

5.3. Execução das funções do núcleo

As funções do núcleo tempo real estão implementadas na linguagem C. Desta forma, quando as tarefas invocam as primitivas do núcleo devem ser obedecidas as convenções de chamadas de funções da linguagem C; isto corresponde ao armazenamento do endereço de retorno e parâmetros a serem passados no "stack" da tarefa ativa. As funções do núcleo portanto, executam sobre o "stack" da tarefa. Com isto, os "stacks" das tarefas deverão ser definidos de maneira a suportar a criação das variáveis locais as funçoes do núcleo. Com isto, o programador pode encontrar dificuldades na estimativa das necessidades de "stack" da tarefas aplicativas; diante disto, o núcleo fornece a função:

void **overflow** ();

indicando que a tarefa deverá ser suspensa, no caso de "overflow" no "stack" da tarefa.

Quando aos segmentos de dados, estes não são compartilhados. Então todas as primitivas do núcleo quando chamadas executam a função:

void **entrada** ();

que recupera o ponteiro do segmento de dados do núcleo no "stack" da tarefa. Da mesma forma, ao término das execuções destas primitivas executado a função:

```
void  saída (tarefa);
  ptr_dsc_tar    *tarefa;
```

que carrega o ponteiro do segmento de dados da tarefa ativa.

5.4. Gerenciamento de Memória

O núcleo tempo real fornece o suporte para o gerenciamento de alocação/desalocação de variáveis dinâmicas no segmento de dados e para alocação/desalocação de blocos de memória (externas ao núcleo).

5.4.1. Alocação/Desalocação de variáveis dinâmicas

As variáveis dinâmicas alocadas pelas funções do NTR encontram-se no próprio segmento de dados (ver figura 5.1.), acima das variáveis estáticas . A área disponível à alocação é definida na iniciação do núcleo pela função:

```
void  inicia_mem_dinâmica (inic_mem_din, tam_mem);
  char       *inic_mem_din;
  tam_mem    unsigned int;
```

que inicia as variáveis internas de controle, com o início e tamanho da área disponível a alocação dinâmica.

Inicialmente a área disponível à alocação de variáveis dinâmica é contínua, constituíndo assim num único segmento. Após várias alocações e desalocações este segmento encontra-se dividido em diversos fragmentos organizados em uma lista

(LIST_FRAG_MEM). As informações sobre o fragmento são mantidas em seu início, num cabeçalho. Um fragmento em determinado instante pode estar livre ou ocupado, partir de informações em seu cabeçalho.

A alocação da uma variável dinâmica é feita pela função:

```
char   aloca_mem_dinâmica (tam_mem);
  int tam_mem;
```

que percorre a LIST_FRAG_MEM a partir do primeiro fragmento livre até encontrar um fragmento ou fragmentos adjacentes livres com memória disponível. A função retorna o ponteiro para o início da variável alocada ou NULL (caso não exista memória disponível). Para evitar a proliferação de pequenos fragmentos é especificado um tamanho mínimo, assim a representação para um fragmento só é criada quando a disponibilidade de memória for maior do que o tamanho mínimo.

A desalocação de uma variável dinâmica é feita pela função:

```
void   restaura_mem_dinâmica (início_mem);
  char    *início_mem;
```

que simplesmente indica no cabeçalho que o fragmento está livre.

5.4.2. Alocação/Desalocação de blocos de memória

Os blocos de memória alocados para o carregamento de código do tipo módulo e criação de instância, embora sejam manipulados pelo núcleo são externos ao mapeamento de memória do NTR. A

área disponível à alocação de blocos é definida na iniciação do núcleo pela função:

```
void   inicia_mem_blc (início_livre, tam_mem);
  char      far       *início_livre;
  unsigned long       tam_mem;
```

A área de memória disponível para alocação de blocos também encontra-se particionada em diversos fragmentos, mantidos em uma lista (LIST_FRAG_BLC). A alocação de bloco de memória é realizada com a função:

```
char   aloca_blc (tam_mem);
  unsigned int  tam_mem;
```

que quando invocado, percorre a LIST_FRAG_BLC a partir do primeiro fragmento livre até encontrar fragmento ou fragmentos adjacentes que possam suprir às necessidades de memória solicitada. Se o fragmento ou a soma de fragmentos for maior do que o solicitado, ocorre uma nova fragmentação, de modo que o restante do bloco não utilizado continue disponível às futuras alocações. Para evitar a proliferação de pequenos fragmentos (blocos de memória), também é especificado um tamanho mínimo para a representação do fragmento. A função retorna um ponteiro para o início do bloco de memória alocado ou NULL (caso não exista bloco de memória disponível).

A desalocação de um bloco de memória simplesmente indica no cabeçalho que o fragmento está livre, sendo executada pela função:

```
void   restaura_blc (início_blc);
  char   *início_blc;
```

5.5. Suporte para Gerenciamento Local

O gerenciador local é composto basicamente dos gerenciadores de:

- **Módulo:** relacionado com as estrutura de tipo módulo e instância de módulo;
- **Ligação ("LINK"):** trata com a conexão/desconexão entre portos;
- **Erro:** processa erros de tempo de execução.

Nas próximas seções serão descritas as funções disponíveis ao Gerenciador Local a fim de suportar a configuração estática e/ou dinâmica de uma aplicação.

5.5.1. Tipo Módulo

O tipo módulo é representado e manipulado pelas funções do NTR através do Bloco de Controle de Tipo Módulo (BCTM). A representação do BCTM é mostrada na figura 5.2. Os BCTMs da estação encontram-se em uma lista (LIST_BCTM).

O gerenciador de módulo executa a função:

```
int   cria_BCTM (tipo, cs_tam, ds_tam, ss_tam, param_tam, inic);
  int             tipo;
  unsigned int    cs_tam,
                  ds_tam,
                  ss_tam,
                  param_tam,
  char            *inic;
```

para solicitar a alocação dinâmica do BCTM e coloca-lo na lista de BCTM da estação. O parâmetro retornado pela função indica:

- bloco de controle criado e colocado na fila;
- já existe representação para o BCTM na LIST_BCTM;
- não existe disponibilidade de memória para a alocação dinâmica do BCTM.

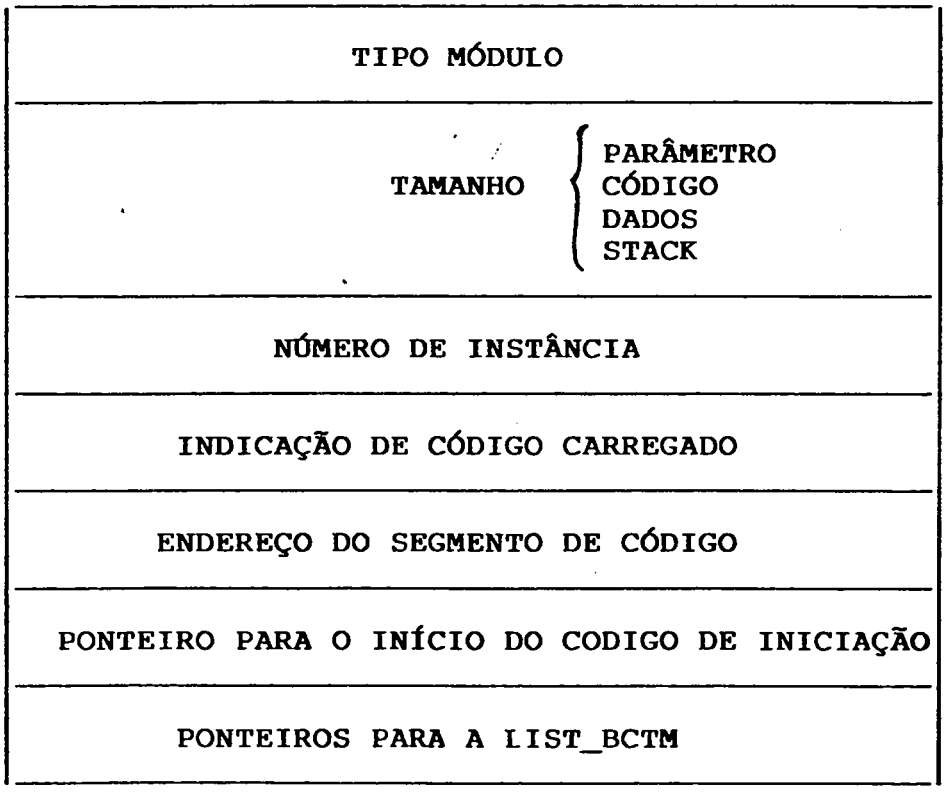


Figura 5.2. - Bloco de Controle de Tipo Módulo (BCTM)

O BCTM contém informações correspondente ao tipo módulo. A definição abstrata de um tipo módulo na LINCE é mapeada (carregada) num segmento de código. O tipo módulo pode estar ou não carregado no momento da criação do bloco de controle; os segmentos de códigos relativos a módulos de serviços do SOD (gerenciadores) encontram-se entre os que podem já estar carregados quando da criação de seus BCTMs no núcleo. Neste caso a função:

```
void  registra cs tip (tipo, base_cs);
  int   tipo;
```

deve ser executada no sentido de obter o BCTM na LIST_BCTM da estação, registrando em seguida o endereço do segmento de código, previamente alocado.

Quando de tipos módulo que necessitam da alocação do segmento (carregamento não realizado), o gerenciador deve executar:

```
char   aloca mem tip (tipo);
  int          tipo;
```

que obtem o BCTM na LIST_BCTM da estação e solicita a alocação do bloco de memória para o segmento de código do tipo módulo. A função retorna o ponteiro para o início do segmento alocado ou NULL. O carregamento do código neste segmento é feito através de um "LOADER", que se utiliza de serviço de acesso remoto à memória, provido pelo SOD.

O segmento de código é composto do(s) corpo(s) da(s) tarefa(s) e do código da função de iniciação do módulo. A

função de iniciação contém as chamadas às funções do núcleo que criam tarefas e portos, e também ligam portos internos do módulo. A figura 5.3. mostra a estrutura do segmento de código do tipo módulo.

No final do processo de carga do tipo módulo o gerenciador de módulo deve executar a função:

```
void   habilita_tipo (tipo);
  int   tipo;
```

que libera a criação de instâncias relativas ao tipo módulo.

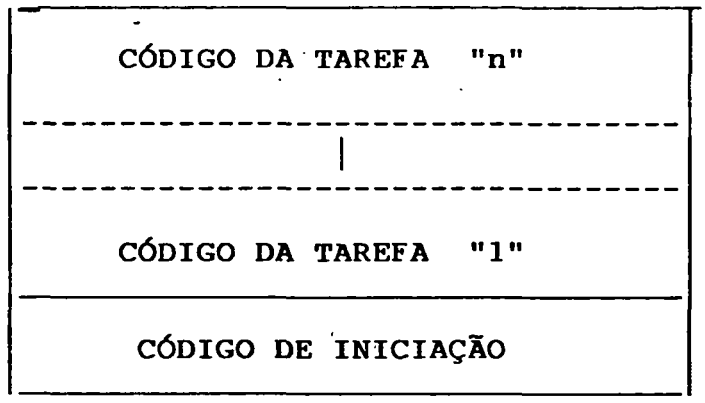


figura 5.3. - Segmento de Código do Tipo Módulo

O núcleo fornece também ao gerenciador, a desalocação de segmentos de código através da função:

```
int   desaloca_mem_tip (tipo);
  int   tipo;
```

que no caso de não existir instância deste tipo módulo solicita a liberação do segmento de código. O parâmetro retornado indica se o segmento foi liberado ou não. Com a liberação do segmento

deve também ser destruída a representação do tipo módulo na estrutura de dados do NTR; isto é feito através da função:

```
void   destroi_BCTM (tipo);
  int   tipo;
```

que retira da LIST_BCTM e desaloca a variável BCTM.

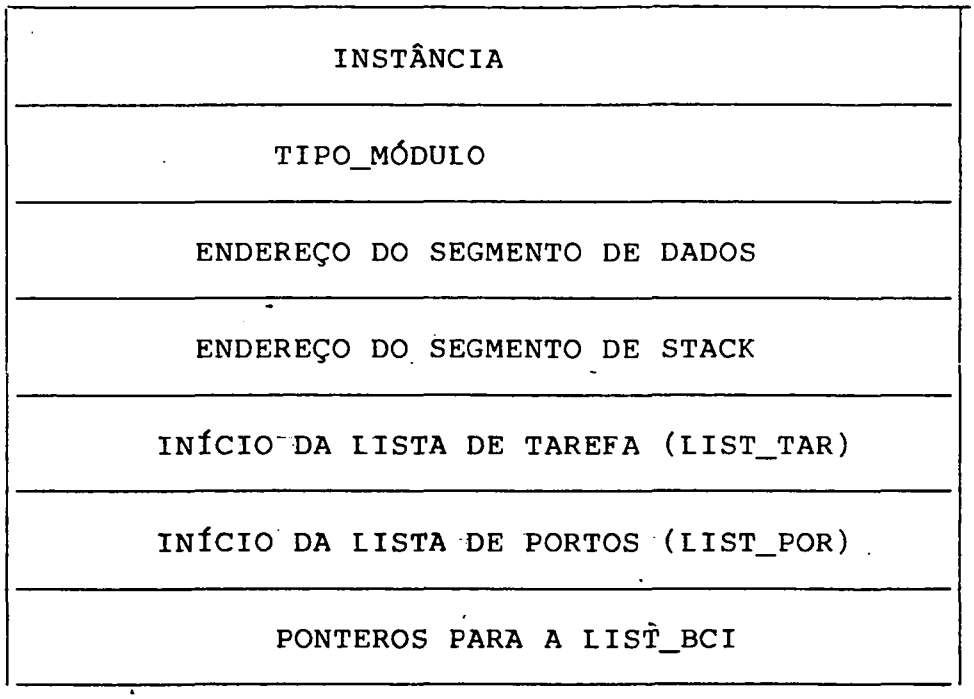


figura 5.4. - Bloco de Controle de Instância (BCI)

5.5.2. Instância

Uma instância é representada e manipulada pelas funções do núcleo através de um Bloco de Controle de Instância (BCI). A representação do BCI é mostrada na figura 5.4. Os BCIs da estação encontram-se em uma lista (LIST_BCI). A representação da instância envolve ainda a criação das representações das tarefas, de portos e de canais de comunicação internos à

instância. A tarefa é representada através de um descritor de tarefa (DSC_TAR, ver figura 5.5.), onde os DSC_TAR pertencentes a uma instância encontram-se em uma lista (LIST_TAR). Os portos são representados através de descritores de porto de saída (DSC_PS) ou de descritores de porto de entrada (DSC_PE),

PONTEIROS PARA O INÍCIO DA TAREFA (CS E OFFSET)
ENDEREÇO DO STACK DA TAREFA { SS, SP_INICIAL, SP_ATUAL }
ENDEREÇO DO SEGMENTO DE DADOS DA TAREFA
PRIORIDADE
ESTADO
TEMPO
CONDIÇÃO
PONTEIRO PARA LIGAÇÃO NA LIST_TAR
INÍCIO DA LIST_POR_TAR
INÍCIO DA LIST_RECEP_SEL
PONTEIRO P/ O ÚLTIMO PORTO DE COMUNICAÇÃO
PONTEIROS PARA LIGAÇÃO NAS FILAS

figura 5.5. - Descritor de Tarefa (DSC_TAR)

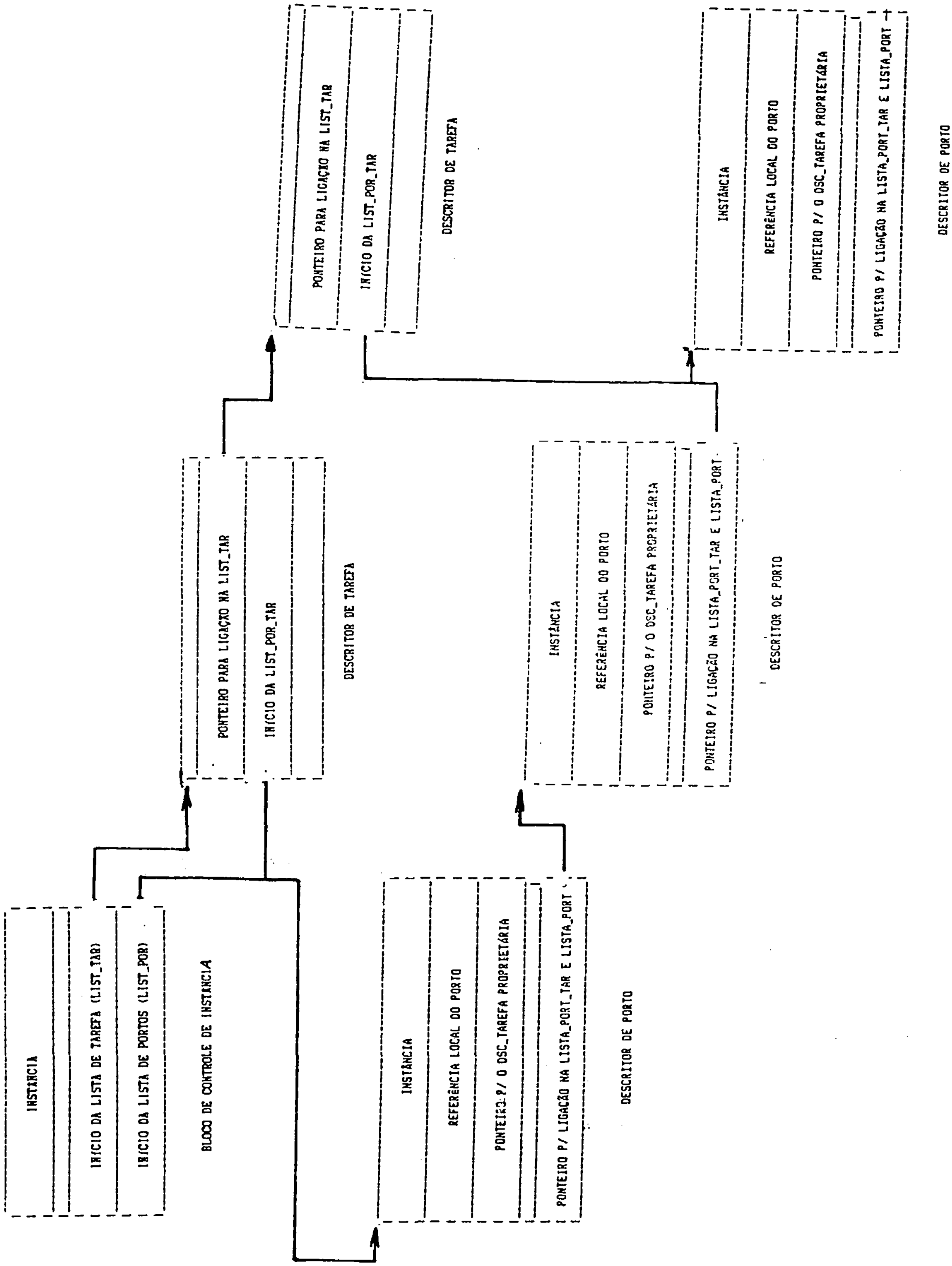


figura 5.6. - Estrutura de Dados de Uma Instância

mantidos na lista de portos da instância (LIST_PORT) e também na lista de porto da tarefa (LIST_PORT_TAR). A estrutura de dados completa correspondente a uma instância é mostrada na figura 5.6.

Uma instância define um novo contexto de execução para o tipo módulo. Esta definição de contexto corresponde a alocação de um **segmento de dados** e de "**stack**". A figura 5.7. mostra os segmentos de dados e de "stack de uma instância.

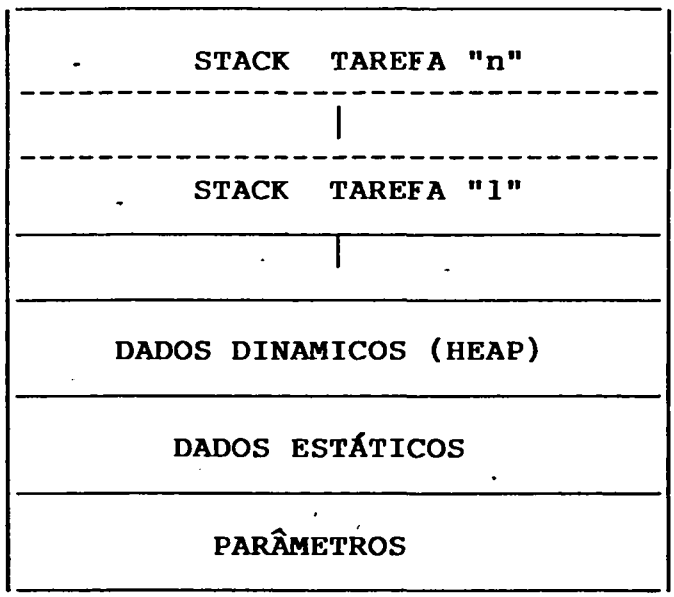


figura 5.7. - Segmento de dados e de "stack" de uma instancia

O **segmento de dados** de uma instância é comum a todas as tarefas pertecentes à instância e apresenta área de parâmetros e de dados estáticos e dinâmicos. Os parâmetos consistem em variáveis definidas estaticamente, mas que terão seus valores iniciados durante a instanciação do tipo módulo. O gerenciamento da área dinâmica é executado através de operações da linguagem base (no caso do PASCAL, "New" e "Dispose").

Durante a criação da tarefa o segmento de "stack" alocado para a instância é dividida em segmentos menores para as tarefas. Desta forma não existem compartilhamento de "stack" entre tarefas de uma mesma instância..

O gerenciador cria uma instância através da função:

```
int   cria instância (instância, tipo, base_param,off_param);
  int         instância,
              tipo;
  char        *base_param,
              *off_param;
```

Esta função obtem o BCTM na LIST_BCTM correspondente à instância, em seguida, solicita a criação do BCI, colocando-o na LIST_BCI e aloca os segmentos de dados e "stack" para a instância. Terminada esta etapa os parâmetros formais são inicializados na área de dados e é invocado o código de iniciação do módulo.

O código de iniciação executa chamadas às funções:

- int cria tarefa (tar_início, prioridade, ss_tam);

```
    char        *tar_início;
    int         prioridade,
                ss_tam;
```

que solicita a alocação dinâmica do descritor de tarefa (DSC_TAREFA, ver figura 5.5) coloca-o na LIST_TAR no BCI e especifica o segmento de "stack" para a tarefa no interior do segmento de "stack" anteriormente alocado à instância. O parâmetro retornado pela função indica se a tarefa foi

criada ou não. A tarefa não é criada se o DSC_TAREFA não puder ser alocado;

TIPO DE PORTO
INSTÂNCIA
PORTO
PONTEIRO P/ O DSC_TAREFA PROPRIETÁRIA
CONDIÇÃO
PRIORIDADE
INFORMAÇÃO DA MSG RECEPCÃO { TAMANHO, CS_MSG, OFFSET_MSG
INFORMAÇÃO DA MSG RESPOSTA { CS_RSP, OFFSET_RSP
PONTEIRO P/ LISTA DE EST_LIG
PONTEIRO P/ LIGAÇÃO NA LISTA_PORT_TAR E LISTA_PORT
PONTEIROS P/ DSC_MSG
PONTEIROS P/ DSC_PS

figura 5.8. - Descritor de Porto de Saida (DSC_PS)

- int cria_porto (ident, porto, num_max_msg, prioridade);

```
int     ident,
        porto,
        num_max_msg,
        prioridade;
```

que solicita a alocação dinâmica do descritor de porto de saída (DSC_PS, ver figura 5.8.) ou de porto de entrada (DSC_PE, ver fig 5.9.) colocando-o na LIST_PORT (BCI) e também na LIST_PORT_TAR (DSC_TAREFA). O parâmetro de retorno da função indica se o porto foi criado ou não. O porto não é criado se o DSC_PORTO não puder ser alocado.
Os portos de saídas são especializados segundo os tipos de comunicação que suportam:

- PS_SEND_NB para um envio não bloqueante;
- PS_SEND_B utilizado quando o envio é bloqueante; e
- PS_SEND_WAIT para envios com respostas.

Ao porto de saída também pode ser atribuídas prioridades: alta e normal; isto permite o tratamento diferenciado de mensagens enviadas através de vários portos de saída quando endereçadas ao mesmo porto de entrada.
Os portos de entradas também são especializados segundos os tipos de comunicações que suportam:

- PE_SEND_NB;
- PE_SEND_B; e
- PE_SEND_WAIT.

O porto de entrada tem associado uma fila de mensagens endereçadas ao porto porém ainda não recebidas pela tarefa.

TIPO DE PORTO
INSTÂNCIA
PORTO
PONTEIRO P/ O DSC_TAREFA PROPRIETÁRIA
CONDIÇÃO
NÚMERO MÁXIMO DE MENSAGEM
NÚMERO DE MENSAGEM
CLÁUSULA
INDÍCE DO FOR (RECEPÇÃO SELETIVA)
INFORMAÇÃO DA MSG RECEPCÃO { TAMANHO, CS_MSG, OFFSET_MSG }
PONTEIRO P/ LIGÀÇÃO NA LIST_RECEP_SELETIVA
PONTEIRO P/ LIGAÇÃO NA LISTA_PORT_TAR E LISTA_PORT
PONTEIROS P/ DSC_MSG
PONTEIROS P/ DSC_PS
EMISSOR DA MSG

figura 5.9. - Descritor de Porto de Entrada (DSC_PE)

Nos casos de PE_SEND_NB as mensagens são mantidas na fila, utilizando-se de um descritor mensagens (DSC_MSG, ver figura 5.10); sendo que o número máximo de mensagens é definido durante a criação do porto. Para PE_SEND_B e PE_SEND_WAIT as mensagens são mantidas na fila através dos descritores de portos de saída. Em todos os tipos de comunicação a fila está organizada por prioridade;

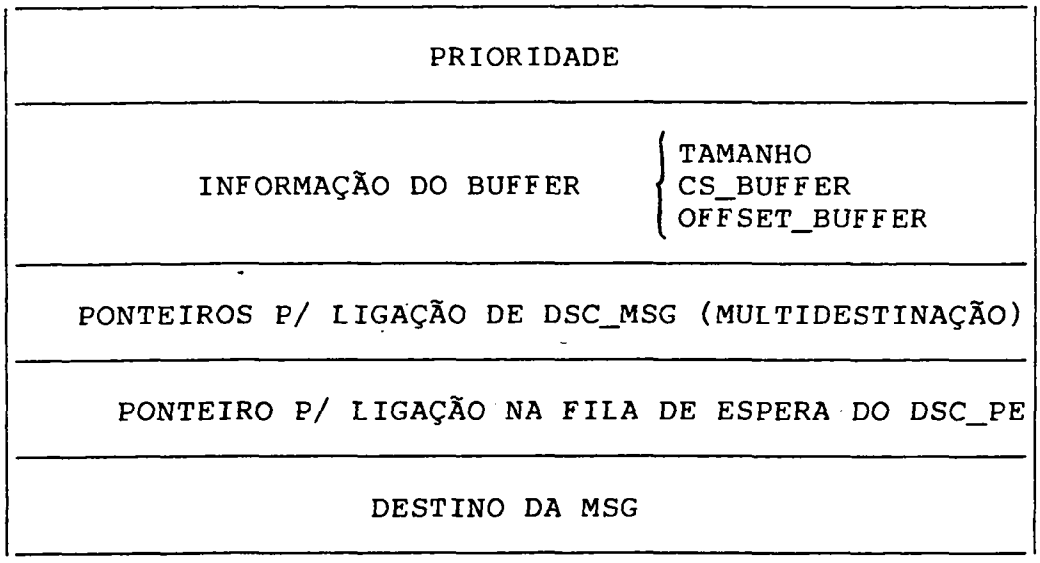


figura 5.10. - Descritor de Mensagem (DSC_MSG)

- int **liga_porto** (PS,PE);

  end_sist PS,

  PE;

Esta função tem sua descrição feita no item 5.5.4.

Quando uma instância é criada, a função incrementa o número de instâncias no BCTM correspondente. O parâmetro de retorno indica se a instância foi criada ou ocorreu uma das seguintes exceções:

- Não existe o tipo módulo, representado por seu BCTM na LIST_BCTM;
- Existe o tipo módulo, porém o código ainda não está carregado no segmento;
- Não existe disponibilidade de memória para variáveis dinâmicas;
- Não existe disponibilidade de blocos de memória para alocar o segmento de dados e/ou de "stack".
- Não existe disponibilidade de blocos de memória para alocar o segmento de dados e/ou de "stack".

As tarefas de uma instância começam a competir pelo processador, quando são **iniciadas** ("start") pelo gerenciador de módulo através da função:

```
void   inicia_instância (instância);
  int   instância;
```

que por sua vez, ajusta o "stack" de cada tarefa da instância, invocando a função:

- ```
  void   ajusta_stack (tarefa);
    dsc_tarefa   *tarefa;
  ```

inserindo, a seguir, as referidas tarefas na fila de pronta.

As tarefas de uma instância deixam de competir pelo processador quando é executado a função:
  ```

```
void   para instância (instância);
  int  instância;
```

que retira a(s) tarefa(s) da instância da(s) filas de execução (pronta, espera e suspensa), abortando a operação atual de cada uma destas tarefas. Depois da execução desta função as estruturas de dados (DSC_TAREFA, DSC_PE, etc) encontram-se nas mesmas condições que quando da criação instância.

Nestas condições, a instância pode ser iniciada outra vez ou ainda destruída. O gerenciador local destroi uma instância através da função:

```
void    destroi instância (instância);
  int   instância;
```

que por sua vez chama as funções:

- ```
  void   destroi tarefa (tarefa);
    dsc_tarefa   *tarefa;
  ```

  que retira a tarefa da LIST_TAR e solicita a desalocação do DSC_TAREFA;

- ```
  void destroi porto (porto);
    dsc_porto   *porto;
  ```

  que retira o porto da LIST_PORT e solicita a desalocação do DSC_PORTO . Portos de saída que no instante da destruição encontram-se ligados, são desligados, antes de serem destruídos.

Depois de executadas estas funções é solicitado a desalocação do BCI e dos segmentos de dados e de "stack" da instância e também é decremento o número de instância no BCTM.

5.5.3. Suporte para Controle de Erro

O núcleo provê a função:

void <u>suspende instância</u> ()

que deve ser executada pela tarefa **gerenciadora de erro** quando da ocorrência de um erro ou exceções especiais (p.e. divisão por zero, indicação de "overflow") em tempo de execução. Esta função retira todas as tarefas da instância das filas de execução, inserindo-as posteriormente na fila de suspensa.

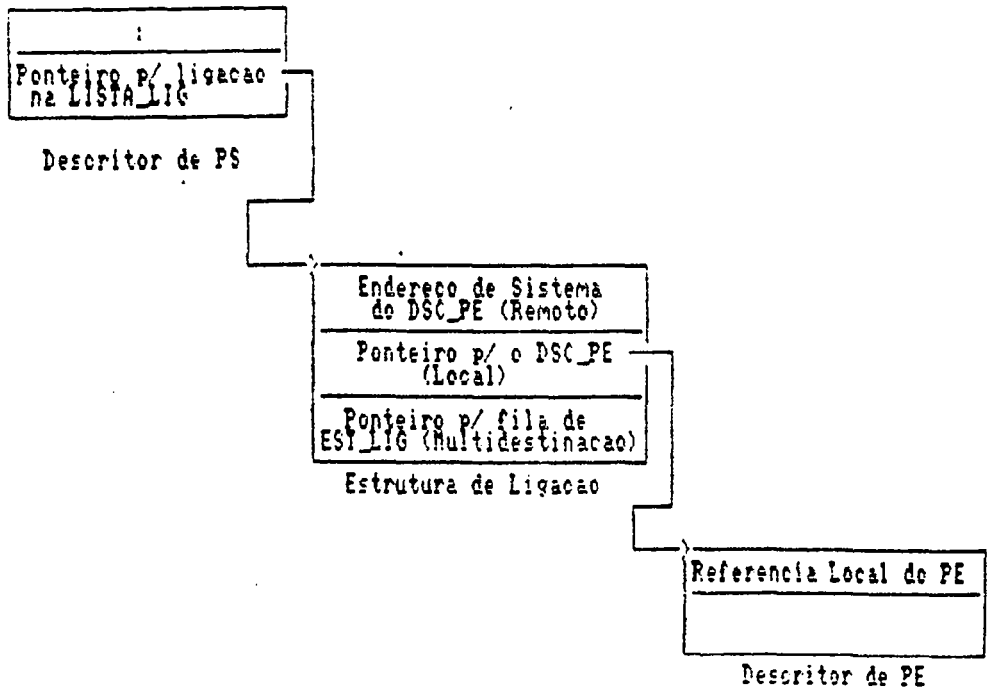


figura 5.11. - Ligação de entre portos

5.5.4. Suporte para Canal de Comunicação

Para que mensagens endereçadas a um porto de saída sejam recebidas num porto de entrada é necessário que estes estejam interconectados logicamente. Esta conexão define um **canal de comunicação** que é representado e manipulado pelas funções do núcleo através de uma estrutura de ligação. A EST_LIG quando do estabelecimento de um canal de comunicação é colocada na lista de ligação do descritor de porto de saída (DSC_PS) deste canal (LIST_EST_LIG, ver figura 5.11); esta estrutura indica informações do porto de entrada (DES_PE) que compõe o canal de comunicação.

Um porto de saída é ligado a um porto de entrada através da função:

```
int liga_porto (PS, PE);
  end_sist  PS,
            PE;
```

que obtém os endereços locais dos DSC_PS e DSC_PE a partir das LIST_BCI da estação e LIST_PORTO da instância; em seguida solicita a alocação dinâmica de uma EST_LIG onde são registrados as informações de ligação (ponteiros para o DSC_PE) e posteriormente, coloca esta estrutura de ligação (EST_LIG) na LIST_LIG do descritor de porto de saída. A função retorna um parâmetro indicando se a ligação foi feita ou não. Uma ligação não é estabelecida quando a EST_LIG não puder ser alocada.

Para portos de entradas remotos o DSC_PS é conectado ao DSC_PE da tarefa servidora de rede (ver comunicação remota, item 5.10.) responsável exclusivamente pela comunicação remota.

A ligação entre portos pode ser alterada através da função:

```
int   religa_porto (PS, PE);
  end_sist  PS,
            PE;
```

que executa as mesmas operações do procedimento anterior com a resalva que a EST_LIG já está alocada, implicando apenas na modificação de seus campos. O parâmetro de retorno indica se a ligação foi alterada ou não. O religamento não é feito se a ligação não é encontrado.

Uma ligação entre portos é destruída usando a função:

```
int   desliga_porto (PS, PE);
  end_sist  PS,
            PE;
```

que obtém o endereço do DSC_PS e retira a EST_LIG correspondente ao porto de entrada da LIST_LIGAÇÃO e em seguida solicita a desalocação da EST_LIG. O parâmetro de retorno indica se o desligamento foi realizado ou não. O desligamento não é feito se a ligação não é encontrado.

5.6. Suporte Multitarefa

O gerenciamento de tarefas em tempo de execução, pelo NTR, é feito através de seus descritores. As informações relevantes como: prioridade, ponteiro do contexto de execução, estado, etc, da tarefa são mantidas no respectivo descritor.

5.6.1. Estados da tarefa

Os estados possíveis de uma tarefa durante a execução são:

- **ATIVA:** a tarefa em estado ativa é a mais prioritária entre as tarefas em estado de pronta. É a tarefa que detém o processador;

- **PRONTA:** as tarefas em estado de pronta possuem todos os recursos necessários a sua execução, com exceção do processador;

- **ESPERA:** as tarefa em estado de espera necessitam de algum recurso para retornar à execução. Normalmente, a disponibilidades deste recursos é sinalizada através de uma mensagem ou de uma interrupção. As tarefas também podem retornar para o estado de pronta em condições de exceção, tal como esgotamento de "timeout";

- **SUSPENSA:** as tarefas em estado suspenso não concorrem mais ao processador. Encontram-se neste estado tarefas que se suspenderam ou foram suspensas através do gerenciador de

erro, devido a ocorrência de algum erro de tempo de execução (overflow e divisão por zero);

- **DEFINIDA:** Neste estado as tarefas possuem todos as estruturas de dados necessárias a sua execução (descritores de portos, tarefas, etc...), mas ainda não estão concorrendo ao processador. O estado **de tarefa definida** é usado normalmente para tarefas que não participam ainda da configuração do sistema. O gerenciador local é responsável pela mudança do estado da tarefa de "definida" para pronta.

O estados da tarefas e as transições possíveis entre estes estados são mostrado na figura 5.12.

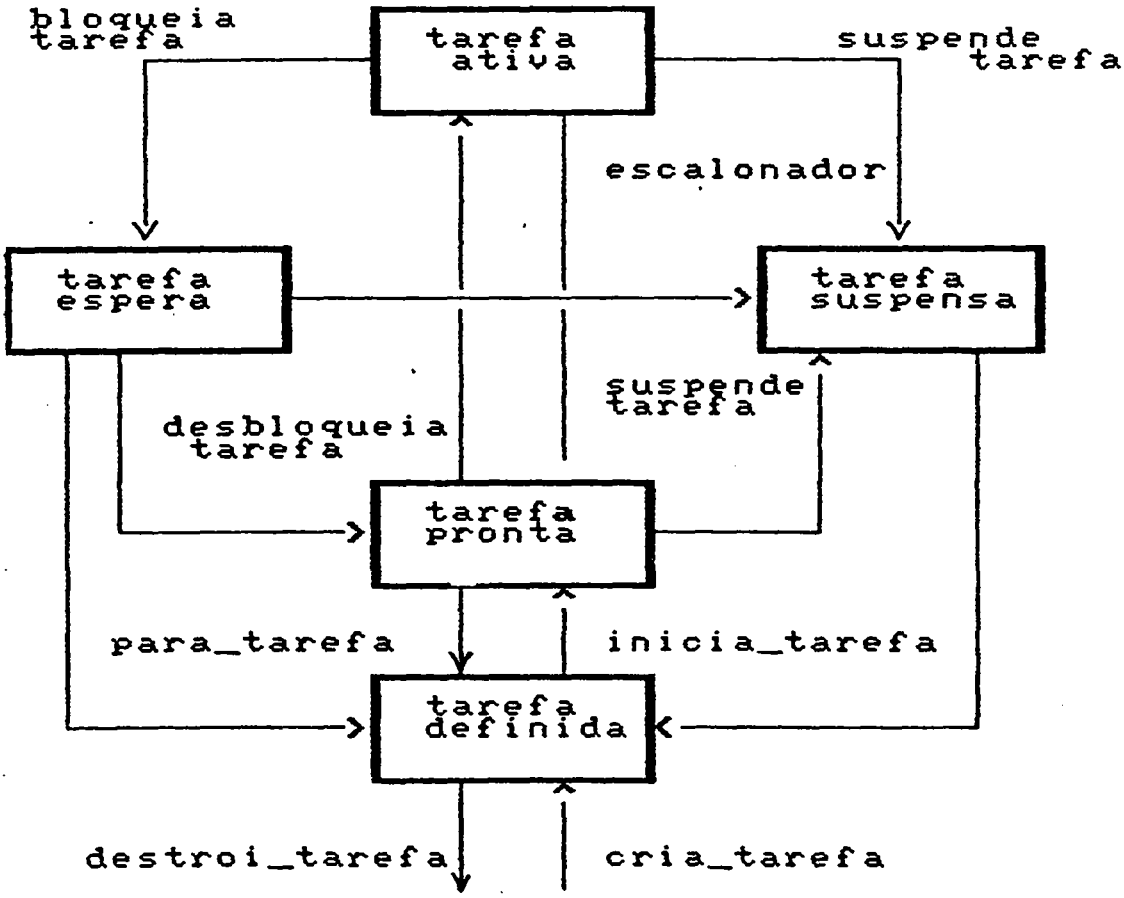


figura 5.12. - Estados e Transições de uma tarefa

Excetuando-se os estados ativa e definida, as tarefas nos outros estados encontram-se em filas organizadas de acordo com algum critério. As filas de estados suportado pelo núcleo são:

- **FILA PRONTA:** A fila de tarefas prontas está organizada segundo critérios de prioridade, sendo que tarefas com mesmas prioridades encontram-se na fila em ordem de chegada. A tarefa mais prioritária encontra-se no topo da fila. A inserção de tarefas na fila é feita por um dos 2 extremos; as tarefas com prioridades maiores são inseridas pelo topo da fila, caso contrário a inserção é feita pelo final da fila. Está facilidade é fornecida de maneira diminuir o tempo de inserção da tarefa na fila de prontas. O núcleo fornece a funções:

```
void  ins_tar_pronta (tarefa);
  dsc_tarefa  *tarefa;

void  ret_tar_pronta (tarefa);
  dsc_tarefa  *tarefa;
```

para inserir e remover respectivamente uma tarefa da fila de pronta.

A ordem de ocupação do processador é estabelecida através da fila de pronta. Portanto, tarefas executando operações (funções) sujeitas a restrições severas de tempo deverão possuir maior prioridade. A prioridade é definida na criação da tarefa e pode ser alterada durante a execução da tarefa através da função:

```
void   muda_prioridade (prioridade);
  int   prioridade;
```

São definidos 16 níveis de prioridades, representado por valores de 0 a 15, para as tarefas. Sendo que a maior prioridade corresponde ao de nível **"zero"** (0). Os níveis de prioridades encontram-se ainda divididos em classes de prioridade:

CLASSE	FAIXA DE PRIORIDADE
sistema	0 - 3
alta	4 - 7
normal	8 - 11
baixa	12 - 15

O agrupamento de níveis de prioridades em classes, tem um sentido prático de facilitar ao usuário a escolha da prioridade atribuída a uma tarefa. As classes disponíveis as tarefas aplicativas programadas pelo usuário são: **alta, normal** e **baixa.**

- **FILA DE ESPERA:** As tarefas na fila de espera estão ordenadas segundo seus **tempos limites de espera.** Este limite de espera corresponde ao valor do relógio tempo real no momento da inserção na fila acrescido do decurso de prazo ("timeout") que, por sua vez, determina o período máximo que a tarefa permanecerá na fila. Para simplificar o mecanismo de controle da fila de espera o "timeout" máximo corresponde a metade do valor assumido pelo relógio tempo real. Tarefas em espera não sujeitas à restrições de tempo são especificadas com "timeout" igual a menos um

(-1) e são inseridas no final da fila com tempo limite de espera indeterminado. As vantagens da organização da fila de espera segundo os tempos limites de espera está nas facilidades de inserir e remover tarefas, bem como de verificar o esgotamento do decurso de prazo testando apenas a primeira tarefa da fila.

O núcleo provê a função:

```
void   ins_tar_espera (tarefa);
  dsc_tarefa   *tarefa;
```

para inserir uma tarefa na fila de espera, e também a função:

```
void   ret_tar_espera (tarefa);
  dsc_tarefa   *tarefa;
```

para remover uma tarefa da fila de espera.

- **FILA DE SUSPENSA:** A fila de tarefas suspensa está organizada de acordo com a ordem inversa de inserção na fila.

O núcleo provê a função:

```
void  ins_tar_suspensa (tarefa);
  dsc_tarefa   *tarefa;
```

para inserir uma tarefa na fila de suspensa e também a função:

```
void   ret_tar_suspensa (tarefa);
  dsc_tarefa   *tarefa:
```

para remover uma tarefa da fila de suspensa.

5.6.2. Escalonamento

Conforme já foi citado anteriormente, o escalonamento segue a estratégia de prioridade com "preempt". A função:

```
void   escalonador ();
```

é acionada diante da alteração de estado da tarefa ativa a partir de funções de comunicação, temporização, interrupção, etc. Quando da chamada ao escalonador, é verificado se a tarefa ativa continua em estado de pronta (isto é, não mudou de estado) e ainda se continua sendo a tarefa mais prioritária entre as prontas; nestes casos, a execução retorna normalmente a função chamadora. Em caso contrário, quando da alteração de tarefa ativa para suspensa ou espera, ou ainda ocorrência de uma tarefa mais prioritária na fila de prontas, o escalonador invoca a função:

```
void   troca_contexto (tarefa);
  dsc_tarefa   *tarefa;
```

que salva o "stack pointer" da tarefa ativa no seu descritor e carrega o contexto da tarefa mais prioritária da fila de prontas. Estabelecer um novo contexto corresponde a carregar nos registros correspondentes o segmento de "stack" e o "stack pointer" da tarefa ativa. Estes valores encontram-se registrados no respectivo descritor da tarefa.

5.7. Comunicação e Sincronização

O suporte para as primitivas de comunicação citadas anteriormente serão descritas neste item.

5.7.1. Envio de mensagem

Todas as primitivas de envio disponíveis na linguagem LINCE são pré-processadas na função:

```
void envio (porto, timeout, tam_msg, base_msg,
                                off_msg, base_rsp, off_rsp);
  int       porto,
            timeout
            tam_msg;
  char      *base_msg,
            *off_msg,
            *base_rsp,
            *off_rsp;
```

que obtem o DSC_PS na FILA_PORT_TAR da tarefa emissora com indíce "porto", em seguida registra os parâmetros base_msg, off_msg, tam_msg, base_rsp e off_rsp neste descritor. O DSC_PE da tarefa receptora é localizado a partir da LIST_LIG (ver figura 5.11.). Se o envio é:

- **SENB_NB:** se a tarefa proprietária (tarefa receptora) do DSC_PE encontra-se aguardando mensagem neste porto, a mensagem é transferida diretamente do espaço de dados da tarefa emissora (registrado no DSC_PS) para o espaço de

dados da tarefa receptora (registrado no DSC_PE); neste caso a tarefa receptora é retirada da fila de espera é inserida na fila de pronta. No caso em que a tarefa receptora não aguarda a mensagem, neste porto, esta é transferida para um "buffer" alocado dinâmicamente na área de dados do núcleo. As informações referentes ao "buffer" são registradas no DSC_MSG (ver figura 5.10.) que também é alocado dinamicamente. O DSC_MSG é então colocado na fila de mensagens do DSC_PE na posição relativa a sua prioridade. Se a fila de mensagem do DSC_PE está completa (número de mensagem igual ao número máximo suportado pela fila de mensagem do DSC_PE), primeira mensagem da fila é descartada (retirada da fila e desalocada). Terminado o envio de uma mensagem é verificado se existe outra EST_LIG (apontado para outro DSC_PE) na LIST_LIG do DSC_PS (envio vários-para-um, ver figura 5.11), neste caso a função só termina quando a mensagem for enviada a todos os portos de entradas que estiverem ligados ao porto de saída;

- **SEND_B:** se a tarefa proprietária do DSC_PE encontra-se aguardando mensagem neste porto, a mensagem é transferida diretamente do espaço de dados da tarefa emissora (registrado no DSC_PS) para o espaço de dados da tarefa receptora (registrado no DSC_PE); neste caso a tarefa é retirada da fila de espera e colocada na fila de pronta. Se a tarefa receptora não está aguardando por mensagem neste porto o DSC_PS é colocado na fila de mensagens do DSC_PE. Diante disto, a tarefa emissora é retirada da fila de tarefas prontas e colocada na fila de espera,

permanecendo neste estado até que ocorra a recepção da mensagem ou se esgote o "timeout". Se o "timeout" é igual a -1 (menos um) a tarefa permanece em espera até a recepção da mensagem (decurso de prazo ilimitado);

- **SEND_WAIT:** se a tarefa proprietária do DSC_PE encontra-se aguardando mensagem neste porto, a mensagem é transferida diretamente do espaço de dados da tarefa emissora (registrado no DSC_PS) para o espaço de dados da tarefa receptora (registrado no DSC_PE); neste caso a tarefa receptora é retirada da fila de espera e colocada na fila de pronta. O DSC_PS é colocado na fila de mensagens do DSC_PE de maneira que a tarefa receptora possa enviar a mensagem de resposta. A tarefa emissora é retirada do estado ativa e é colocada em estado em espera. Caso contrário, quando a tarefa receptora não se encontra aguardando mensagem neste porto, o DSC_PS é colocado na fila de mensagens do DSC_PE da tarefa receptora, permitindo que a mensagem seja recebida posteriormente e também que se produza a resposta. Desta forma em ambas condições a tarefa emissora é colocada na fila de espera, permanecendo nesta até a chegada da resposta ou o esgotamento do "timeout". Se o "timeout" é igual a -1 (menos um) a tarefa permanece em espera até a recepção da mensagem ("timeout" ilimitado).

A função de **escalonamento** é chamada pela função de **envio**, caso o estado de alguma tarefa (emissora ou receptora) é alterado.

Após a execução da função envio a tarefa emissora pode obter a razão pela qual retornou ao estado ativa, através da função:

int **falha** ();

que retorna o parâmetro indicando uma das seguintes condições:

- Esgotamento de "timeout" (envio bloqueante);
- Porto não ligado, indicando que o porto de saída não está ligado a nenhum porto de entrada;
- Envio normal.

Após o teste do parâmetro retornado pelo função **falha** a tarefa pode continuar executando normalmente ou ainda executar em caso de exceção o tratamento específico. O código correspondente a exceção faz parte da programação da tarefa.

5.7.2. Recepção de mensagem

A LINCE provê recepções não bloqueante, bloqueantes e seletiva. A recepção bloqueante e não bloqueante é executada sobre apenas um porto de entrada, já a recepção seletiva é executada em varíos portos de entrada. Um exemplo da comando de recepção seletiva é mostrada a seguir:

```
SELECT
     RECEIVE  msg1  FROM  pe1
  OR
    WHEN  guarda = TRUE
        RECEIVE  msg2  FROM  pe2
  OR
    FOR  i=3   TO  10
      RECEIVE  msg (i)  FROM  pe (i)
  OR
    TIMEOUT  = -1;
END_select
```

No comando cada primitiva de recepção (RECEIVE) caracteriza uma cláusula. Porém uma cláusula pode conter recepções em famílias de portos; o terceiro RECEIVE no acima , mostra uma família de portos embutida num comando **FOR**.

Todos os tipos de comandos de recepção são pré-processados no caso mais geral, que é a recepção seletiva em famílias de portos (caso geral). Por exemplo, um comando único de recepção em um porto é pré-processado em uma recepção seletiva com uma única cláusula em uma família de porto com apenas um componente. O núcleo suporta todos os tipos de recepções acima, através das funções:

- void **inicia_recepção** (porto, ind_for, cláusula,
  base_msg, off_msg);

```
int     porto,
        ind_for,
        cláusula;
char    *base_msg,
        *off_msg;
```

que quando chamada, obtém o DSC_PE de indíce "porto" na LIST_PORT_TAR, registrando em seguida os parâmetros base_msg e off_msg neste descritor. O DSC_PE é então inserido na lista de recepção seletiva (LIST_RECEP_SEL) do descritor da tarefa receptora. Esta função em uma recepção seletiva deve ser executada para todos os portos de entrada. No caso de recepção bloqueante ou não, em um único porto de entrada está função será invocada apenas uma vez;

- void **recepção** (timeout);

```
int   timeout;
```

que deve ser chamada, após a execução, da função inicia_recepção em todos os portos da entradas que compõem a recepção. A função percorre a LIST_RECEP_SEL da tarefa ativa (tarefa receptora) a partir de seu início até encontrar um porto de entrada com mensagem ou o final da fila. Se existe porto de entrada com mensagem na fila, esta é transferida do endereço de origem (registrado no DSC_MSG ou DSC_PS) para o espaço de dados da tarefa receptora (endereço registrado no DSC_PE). Por outro lado, se não existe mensagem tem-se:

- **"timeout" igual a zero:** a tarefa continua em estado de ativa, mas é indicado em seu descritor que a mensagem não foi recebida;

- **"timeout" maior que zero:** a tarefa é retirada da fila de pronta e inserida na fila de espera, permanecendo neste último estado até a chegada da mensagem ou esgotamento do "timeout";

- **"timeout" igual menos um (-1):** as ações equivalem ao caso anterior, com a tarefa permanecendo em espera até a chegada de uma mensagem.

A função **escalonador** é invocada pela função recepção, no caso do estado de alguma tarefa (emissora ou receptora) ser alterado durante sua execução.

Após a execução da recepção a tarefa pode obter o motivo pelo qual retornou ao estado de pronta através da função:

```
void   condição recepção (ind_for, cláusula);
  char far        *ind_for,
                  *cláusula;
```

As possíveis razões são:

- esgotamento de "timeout" (recepção bloqueante) ou inexistência de mensagem (recepção não bloqueante);

- mensagem recebida. Neste caso a cláusula a qual pertence o porto de entrada em que foi recebida é retornada, bem como o indíce do FOR em recepções sobre familias de portos.

Esta função permite a tarefa, após o teste na variável de retorno executar operações compatíveis com a mensagem recebida ou ainda, em casos de exceções realizar o tratamento específico.

5.7.3. Envio da mensagem resposta

Afim de prover um canal de comunicação bidirecional a LINCE permite que se possa, após a recepção de uma mensagem em um porto de entrada, enviar uma via o mesmo porto de entrada.

O envio de mensagem resposta é realizado através da função:

```
void__responde (porto, tam_rsp, base_rsp, off_rsp);
  int           porto,
                tam_rsp;
  char          *base_rsp,
                *off_rsp;
```

que quando executado obtem o DSC_PE na LIST_PORT_TAR de indíce "porto", registrando os parâmetros base_rsp e off_rsp neste descritor. Se a tarefa proprietária do primeiro DSC_PS da fila de mensagem encontra-se aguardando pela mensagem resposta, esta é transferida diretamente do espaço de dados da tarefa que executa a função responde (registrado no DSC_PE) para o endereço registrado no DSC_PS; com isto a tarefa proprietária deste último descritor é retirada da fila de espera e inserida na fila de tarefas prontas. A mensagem resposta é descartada caso da tarefa não esteja aguardando pela mesma. A função escalonador é invocada pela função resposta se durante sua execução o estado de alguma tarefa for alterado.

5.8. Temporização

O núcleo tempo real mantém um contador de 16 bits (variável do tipo "unsigned int") que é incrementado periodicamente pela função:

void **tempo** ();

que deve ser chamada por uma tarefa (**tarefa tempo**) que é ativada através de uma interrupção. Além de incrementar o relógio tempo real, a função verifica na fila de espera se existe(m) tarefa(s) com "timeout" esgotado, e em caso positivo:

- retira a(s) tarefa(s) da fila de espera;
- indica no(s) DSC_TAREFA(s) o esgotamento do "timeout";
- coloca(s) as tarefa(s) na fila de pronta; se o esgotamento do timeout" corresponde a uma exceção, a função que originou o bloqueio (envios e recepção bloqueantes) é abortado.

A verificação de esgotamento de "timeout" é facilitada pela organização da fila de espera. Para verificar se existe(m) tarefa(s) com "timeout" esgotado basta comparar o tempo de espera da primeira tarefa da fila com o relógio tempo real.

A **tarefa tempo** deve ter prioridade máxima e quando de sua ativação é desnecessária uma chamada explícita da função **escalonador**.

O valor do relógio tempo real pode ser lido pelas tarefas atráves da função:

unsigned int **le_relógio** ();

que retorna o valor do relógio.

5.9. Tratamento de Interrupção

A estrutura do suporte para o atendimento de interrupção esta centrada em uma tabela de interrupção interna ao núcleo. A TAB_INTR relaciona os vetores físicos de interrupção (vetores de interrupção do processador) com os vetores lógicos de interrupção de níveis lógicos gerenciados pelo núcleo. A noção de vetor ou nível lógico de interrupção tem por objetivo permitir a reentrância de interrupção física. Desta forma, esta estrutura possibilita que mesmo que a tarefa não esteja aguardando uma interrupção de um nível lógico, uma vez que ela encontra-se relacionada com uma de nível lógico, as informações quanto ao número de ocorrência desta é mantida na TAB_INTR. Cada posição da referida tabela esta relacionada com uma função **de entrada** que chama o **manipulador de interrupção.** Por exemplo, a posição da tabela 5 esta relacionada com a função:

void **entr_manipulador_5** ();

que quando ativada, chama a função:

void **manipulador** (ind_intr);
  int ind_intr;

passando como parâmetro o **ind_intr** do nível lógico 5.

Estas funções são internas ao núcleo, fazendo parte do gerenciamento de interrupção. A nível de aplicação (linguagem LINCE) uma tarefa torna-se apta à atender interrupções através das funções:

- int **inicia_interrupção** (vtr_intr);
  unsigned int vtr_intr;

  que obtem o primeiro vetor lógico de interrupção disponível (a primeira posição livre da TAB_INTR). Uma vez alocada o vetor lógico, esta função registra na posição referente ao vetor físico (vtr_int), o endereço inicial da função entrada correspondente ao vetor lógico. A função **inicia_interrupção** retorna o vetor lógico de interrupção. Esta funcão deve somente ser invocada uma vez durante a execução da tarefa;

- void **espera_interrupção** (ind_intr);
  int ind_intr;

  verifica na TAB_INTR se a interrupção de nível lógico (ind_intr) já ocorreu. Em caso afirmativo, o número de ocorrências da interrupção na TAB_INTR é decrementado e a tarefa continua em estado de ativa. No caso da interrupção não ter ocorrido, a tarefa é retirada da fila de pronta e inserida na fila de espera com decurso de prazo indeterminado. Neste caso a função **escalonador** deve ser invocada.

Quando ocorre uma interrupção, a execução é desviada para a função **entrada do manipulador**, que por sua vez é ativado. O manipulador é implementado pela função:

```
void   manipulador ();        /* programamada em Assembly */
```

que é responsável:

- pelo salvamento do contexto da tarefa (registradores) da tarefa ativa;

- pela chamada à função:

  ```
  void   reconhece_interrupção (ind_intr);
    int   ind_intr;
  ```

  que sinaliza a ocorrência da interrupção. A indicação de ocorrência pode ser feita através do incremento do número de interrupção da TAB_INT, neste caso a tarefa não aguarda pela interrupção, ou pela retirada da tarefa da fila de espera e colocação na fila de pronta. Neste último caso a função **escalonador** é invocada.

- pela restauração do contexto da tarefa; e

- indicação do final de atendimento de interrupção (instrução iRET).

A estrutura de manipulação de dado para tratamento de interrupção é similar a implementada em (COELLO, 1986).

TIPO
TIMEOUT
IND_ESTAÇÃO_DEST
IND_INS_MOD_DEST
IND_PE
IND_ESTAÇÃO_OR
IND_INS_MOD_OR
IND_PS

figura 5.13. - Descritor de Mensagem Remota

5.10. Comunicação Remota

A comunicação remota foi projetada de maneira a permitir a comunicação inter-estação tranparente para as tarefas aplicativas, de tal forma que não implique em distinções entre comunicação local e remota. Isto é feito através do módulo Servidor de Rede composto de **uma tarefa Servidora de Rede de Envio** (SR_ENVIO) e de **uma ou várias tarefas Servidora de Rede de**

Recepção (SR_RECEPÇÃO). O SR_ENVIO contém uma única interface, porto de entrada do tipo SENDWAIT (PE_SEND_WAIT), que recebe todas as mensagens, sejam as comunicações bloqueantes ou não bloqueantes. O SR_RECEPÇÃO contém como interface 4 portos: um porto de entrada (PE_SEND_WAIT) e de três portos de saída de tipos distintos: PS_SEND_NB, PS_SEND_B e PS_SEND_WAIT.

Uma comunicação remota resume-se basicamente na seguinte descrição: " uma tarefa envia mensagem através de primitivas bloqueantes ou não para um PS, este porto mantém a EST_LIG contendo o endereço local do PE e o endereço de sistema do PE remoto. Quando a tarefa SR_ENVIO recebe a mensagem através deste porto, formata o descritor de mensagem remota (DSC_MSG_REMOTA, ver figura 5.13) e acessa a interface da camada transporte. A tarefa SR_RECEPÇÃO ao receber a mensagem, executa a operação de envio correspondente (SEND_NB ou SEND_B ou ainda SEND_WAIT). Nos casos de envios bloqueantes SEND_B ou SEND_WAIT a tarefa SR_RECEPÇÃO será bloqueada respectivamente até a recepção da mensagem ou chegada da mensagem de resposta. Isto implica na necessidade de várias tarefa SR_RECEPÇÃO em uma configuração, para que durante o instante que uma tarefa SR_RECEPÇÃO esteja bloqueada nas próximas mensagens endereçadas a estação sejam recebidas.

A configuração das tarefas servidoras de rede (de envio e de recepção) é mostrada na figura 5.14.

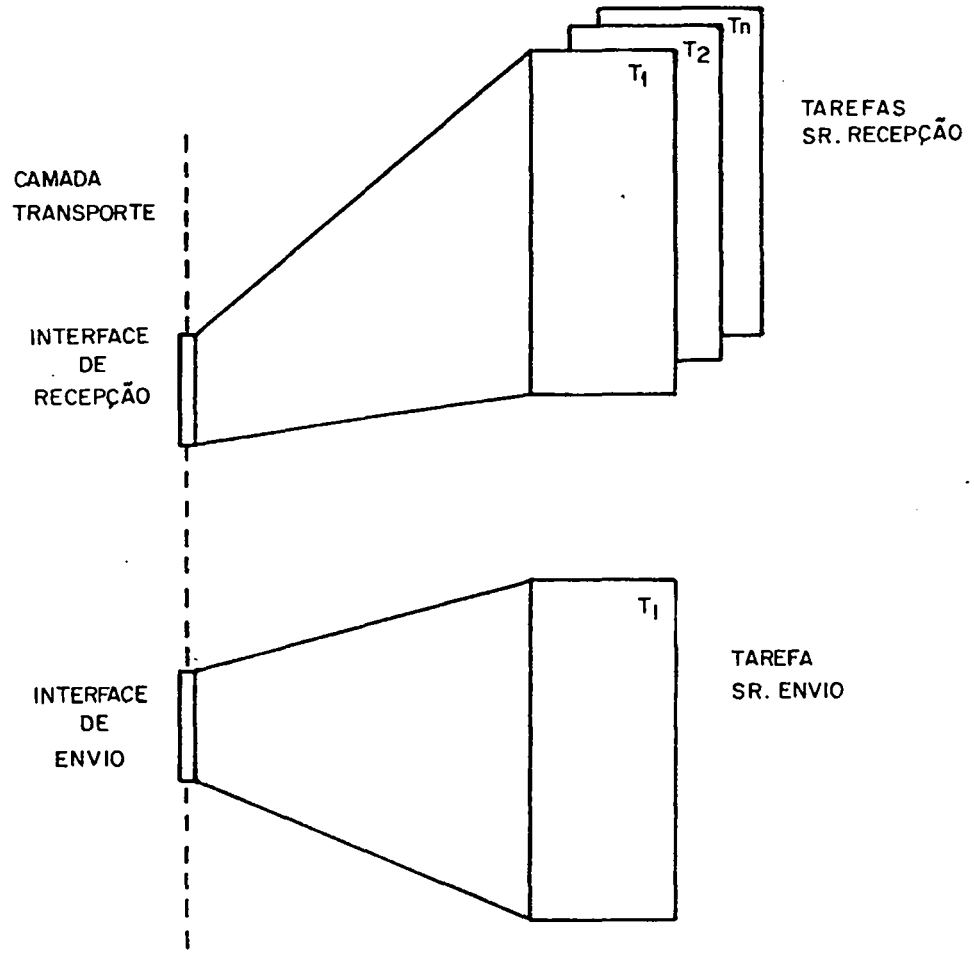


figura 5.14. - Configuração das Tarefas Servidoras de Rede

5.10.1. Descrição da Interface da Camada transporte

O suporte de comunicação utilizado na atual versão é uma rede local comercial CETUS (CETUS INFORMATICA SA.), com topologia de barramento com protocolo de acesso CSMA-CD ("Carrier Sense Multiple Acess With Collision Detection").

O "software" utilizado para acessar este suporte de comunicação foi também desenvolvido pela própria CETUS. A atual versão deste "software" é denominada "driver de rede".

O "driver de rede" acessa e utiliza as "interfaces" de mais baixo nível de forma a prover os seguintes serviços:

- Número da placa;
- Status;
- Enviar pacote;
- Cancelar envio de pacote;
- Habilitar recepção de pacote;
- Desabilitar recepção de pacote;
- Número da versão do "driver de rede";
- Endereço da placa.

Não é nosso objetivo descrever todos os serviços e suas interfaces. Para implemetar a versão corrente do módulo Servidor de Rede, nós utilizamos somente os serviços enviar pacote e habilitar recepção de pacote.

Um detalhe interessante de ser frisado é a limitação em 576 bytes do tamanho máximo de pacotes.

5.10.2. Tarefa SR_ENVIO

Quando um porto de saída é conectado a um porto de entrada, as informações referentes ao porto de entrada são registrada em uma EST_LIG mantida no DSC_PS. Estas informações do porto de entrada correspondem ao endereço local do porto (ponteiro para o DSC_PE) e ao endereço de sistema (estação/instância/porto). Quando o porto de entrada é remoto, o endereço local mantido na EST_LIG corresponde ao ponteiro para o DSC_PE da tarefa SR_ENVIO. Quando do envio de mensagem, a função envio utiliza-se sempre do endereço local para endereçar o DSC_PE, desta forma quando o porto é remoto a mensagem é destinada a tarefa SR_ENVIO.

A tarefa SR_ENVIO recebe mensagem sincronamente (Recepção bloqueante) no porto de entrada, em um "buffer" de tamanho fixo equivalente ao tamanho máximo da mensagem (560 bytes). Para cada mensagem recebida deve ser formatado o DSC_MSG_REMOA através da função:

```
int   formata_dsc_msg_remota (base_dsc_msg_remota,
                                    offeset_dsc_msg_remota);
  char      *base_dsc_msg_remota,
            *off_dsc_msg_remota;
```

que preenche os campos relativos a mensagem a ser enviada. As particularidades apresentadas na formatação do DSC_MSG_REMOTA são:

- **SEND_NB:** o "timeout" é setado em zero e os campos relativos a origem (Estação/Instância/PS) não são preenchidos;

- **SEND_B:** o "timeout" correspondente ao envio é decrementado de um valor constante, correspondente ao tempo médio da transmissão de pacote pelo suporte de comunicação. Também neste caso os campos destinados à origem não são preenchidos;

- **SEND_WAIT:** o "timeout" correspondente ao envio é decrementado de duas vezes o tempo médio da transmissão de pacote. Os campos relativos a origem são preenchidos de maneira que remotamente possa ser enviada uma mensagem resposta.

Posteriormente a formatação do DSC_MSG_REMOTA, a tarefa SR_ENVIO utiliza-se do **serviço enviar pacote**, disponível na interface do driver de rede (item 5.10.1).

5.10.3. Tarefa SR_RECEPÇÃO

O tarefa SR_RECEPÇÃO é ativada por interrupção, assim toda vez que um pacote chega na estação é gerada uma interrupção de maneira a este ser recebido mais brevemente possível. Esta preocupação é perfeitamente justificável, pois libera a recepção de novos pacotes e por outro lado evita que pacotes sejam perdidos por superposição em "buffer".

Antes de esperar pela chegada da interrupção a tarefa deve habilitar a recepção de pacote através do serviço provido pelo **driver de rede**. Quando chega um pacote, a tarefa é despertada e identifica o tipo de mensagem remota, a seguir executa, conforme o tipo de comunicação a operação correspondente:

- **SEND_NB:** a tarefa religa seu PS_SEND_NB com o porto de entrada de destino antes de enviar a mensagem. A partir daí está habilitada a receber novos pacotes;

- **SEND_B:** a tarefa religa seu PS_SEND_B com o porto de entrada de destino antes de enviar a mensagem. A tarefa estará apta a receber novos pacotes após a conclusão do envio (por recepção da mensagem ou esgotamento de "timeout").

- **SEND_WAIT:** a tarefa religa seu PS_SEND_WAIT com o porto de entrada de destino antes de enviar a mensagem. A tarefa permanece em espera até receber a mensagem de resposta ou esgotamento do "timeout". Quando a mensagem de resposta é recebida no PS_SEND_WAIT, a tarefa transmite um pacote a estação origem da mensagem. Para isto, necessita da formatação do DSC_MSG_REMOTA que é executada pela função:

```
void  formata_msg_rsp (base_dsc_msg_remota,
                                    off_dsc_msg_remota);
  char     *base_dsc_msg_remota,
           *off_dsc_msg_remota;
```

Em seguida utilizando-se do serviço provido pelo "driver de rede" envia o pacote. Após a execução da transmissão de pacote ou esgotamento de "timeout" a tarefa está apta a receber novos pacotes;

- **REPLY:** a tarefa executa a função:

```
void  prepara_para_responder (PS,PE);
  end_sist  PS,
            PE;
```

que estrutura o DSC_PE do SR_RECEPÇÃO e o DSC_PS da tarefa que espera pela resposta,de forma que a tarefa posteriormente possa executar o envio da resposta. A partir do envio de resposta a tarefa está apta a receber novos pacotes.

Como foi dito anteriormente em uma estação pode coexistir várias tarefas SR_RECEPÇÃO, neste caso elas encontram-se organizadas por ordem de habilitação de recepção de pacotes, sendo que a manipulação desta fila é de responsabilidade do "driver de rede".

5.11. Característica e Desempenho

Os requisitos de tempo real, estão intimamente ligados ao suporte de tempo de execução. Portanto, aos aspectos referentes às necessidades de memória, de desempenho na execução das primitivas de comunicação e no tratamento de interrupção são fatores determinantes para a obtenção de tempos de respostas compatíveis com o processo a ser controlado.

Neste sentido, o objetivo deste item é levantar alguns parâmetros quantitativos referentes aos requisitos de memória para a implantação do núcleo tempo real e também dos tempos envolvidos na execução das primitivas de comunicação e no tratamento de interrupção.

5.11.1. Requisitos de Memória

A área de memória do núcleo encontra-se dividida nas áreas de código, dados e stack. Com relação a área de código, o núcleo requer aproximadamente 11.9 Kbytes de memória, assim divididas:

- gerenciamento local..... 3.6 Kbytes
- gerenciamento de tarefa..... 1.3 Kbytes
- gerenciamento de tempo e interrupção..... 1.1 Kbytes
- comunicação: local..... 3.9 Kbytes
  remota..... 0.5 Kbytes
- gerenciamento de memória..... 1.5 Kbytes

Estes valores referem-se às funções de cada bloco funcional, não incluindo códigos de tarefas (por exemplo: tarefas servidoras de rede) que se utilizam deste suporte. Do total acima (11.9 Kbytes) apenas 4% destas funções foi implementada em "assembly", facilitando assim a portabilidade do núcleo para outros processadores.

A versão utilizada do compilador C (Microsoft 4.0) fornece um conjunto de funções em biblioteca de tempo de execução (Run-Time Reference); fatores como grande necessidade de memória, utilização das funções do sistema operacional MS-DOS e execução destas funções segundo mapeamento de memória especificos pesaram para que estas funções não fossem utilizadas pelo núcleo.

A área de dados encontra-se dividida em estática e dinâmica. As exigências de memória estática são extremamente baixa (inferiores a 0.2 Kbytes) pelo fato das funções do núcleo

manipularem ponteiros para filas de estruturas de dados alocadas na área de dados dinâmica. Desta forma, com relação a área dinâmica que deve suportar todas as estruturas de dados manipuladas pelo núcleo as exigências são variáveis de acordo com a aplicação. Esta área pode ser estimada de acordo com o número módulos, tarefas, portos, etc, suportadas pelo núcleo. O tamanho das estruturas é:

- Bloco de Controle de Tipo Módulo (BCTM).....21 bytes
- Bloco de controle de Instância (BCI).....18 bytes
- Descritor de Tarefa (DSC_TAREFA).....32 bytes
- Descritor de Porto (DSC_PE ou DSC_PS).....44 bytes
- Estrutura de Ligação (EST_LIG).....10 bytes
- Descritor de Mensagem (DSC_MSG).....22 bytes

Com relação às exigências de área de "stack"; estas são extremamentes baixas pelo fato desta área ser utilizada apenas na iniciação do núcleo tempo real.

5.11.2. Requisitos de Tempo

A viabilidade da utilização de um suporte para aplicações de tempo real é principalmente verificada pela avaliação dos tempos de execução das primitivas fornecidas; particular atenção dada as primitivas de comunicação, tratamento de interrupção e de escalonamento de tarefas (troca de contexto). São apresentados neste sub-item valores que possibilitam analisar o comportamento do núcleo com relação a estes requisitos.

No sentido de qualificar os tempos dos serviços de IPC foram elaborados os seguintes testes:

a) envio não bloqueante (SEND_NB):

- Tarefa receptora não aguardando Mensagem (alocação de "buffer") o tempo conseguido neste caso é de aproximadamente 0.526 mseg (milisegundos) para uma mensagem de 10 bytes;

- Tarefa receptora com maior prioridade e aguardando Mensagem (sem alocação de "buffer"): o tempo conseguido neste caso é de aproximadamente 0.43 mseg (milisegundos) para uma mensagem de 10 bytes;

- Tarefa receptora com menor prioridade e aguardando Mensagem (sem alocação de "buffer"): o tempo conseguido neste caso é de aproximadamente 0.35 mseg (milisegundos) para uma mensagem de 10 bytes;

b) envio bloqueante aguardando recepção (SEND_B):

- Tarefa receptora maior prioridade e aguardando Mensagem: o tempo conseguido neste caso é de aproximadamente 0.43 mseg (milisegundos) para uma mensagem de 10 bytes;

- Tarefa receptora menor prioridade e aguardando Mensagem: o tempo conseguido neste caso é de aproximadamente 0.350 mseg (milisegundos) para uma mensagem de 10 bytes;

c) envio bloqueante aguardando resposta (SEND_WAIT)

- o tempo conseguido neste caso foi de 1.9 mseg para uma comunicação envolvendo mensagem de 10 bytes.

Os valores obtidos referem-se a um equipamento com processador 8088 de frequência de "clock" de 4.77 MHz. Os tempos foram obtidos através da execução das respectivas primitivas, simulando determinadas situações; por exemplo, para obtenção do tempo de um envio assíncrono as tarefas envolvidas inicialmente executam operações pré-determinadas. Quando da execução da primitiva, o contador de "timer" (8253) é setado no início e lido no final da operação correspondente. Este contador possui uma frequência de entrada de 1.8432 MHz possibilitando assim medidas de até 542.53 nseg (nanosegundos). As interrupções de hardware utilizadas no sistema (compatíveis IBM-PC) quando da realização dos testes estão habilitadas, logo os dados acima estão acrescidos dos valores correspondentes aos tempos de tratamento destas interrupções. Outro tipo de estimativa destes tempos podem ser feitas utilizando a contagem dos tempos médios de cada intrução.

As diferenças do tempo envolvido na execução do envio SEND_B, com a tarefa receptora sendo mais prioritária, está na troca de contexto envolvida (0.08 mseg).

Com relação a comunicação remota valores de tempo correspondentes à execução das primitivas são irrelevantes se comparados com o transporte da mensagem na rede, este tempo está estimado em 28 mseg. Isto se deve ao fato que a rede utilizada

não segue a proposta **datagrama** apresentada no item 2.8. para redes locais de suportes para Tempo Real.

O tempos referentes as interrupções também foram obtidos, porém levando-se em consideração que a tarefa a ser ativada é a mais prioritária (0.11 mseg). Para de operação normal, com várias tarefas no sistema, este tempo deve ser superior.

5.12.Conclusões

Neste capítulo foram apresentados detalhes de implementação do núcleo. São apresentados também valores quantitativos que indicam o desempenho do núcleo. Os resultados obtidos foram considerados plenamente satisfatórios.

CAPÍTULO 6

CONCLUSÃO

O ambiente ADES (Ambiente de Desenvolvimento e Execução de Software) que está sendo desenvolvido, tem o propósito de investigar uma variedade de problemas relacionados com a construção de programas distribuídos para aplicações em tempo real. Este trabalho tratou do suporte de tempo de execução deste ambiente: o núcleo de tempo real. Este núcleo é destinado a suportar programas distribuídos construídos segundo o princípio de decomposição modular, com características que favorecem a reutilização de componentes e a configuração dinâmica (programas evolutivos).

O núcleo na sua versão atual foi projetado para se executar em uma rede local com equipamentos compatíveis com IBM-PC; sendo porém facilmente transportável para hardware similares de outros equipamentos que dispõe de compilador C. Este núcleo é bastante compacto e apresenta características de modularidade.

Em relação aos mecanismos de tempo real, embora a estratégia de "preempt" por prioridade e os mecanismos de

temporização definidos sejam plenamente satisfatórios para sistemas relativamente simples, a noção de "deadline", uma vez introduzida na fila de prontas deve conduzir a uma melhora no atendimento das necessidades de tempo de resposta.

A funcionalidade apresentada por este núcleo abriu perspectivas de novos trabalhos. Entre estes trabalhos encontram-se a Linguagem de Implementação de Sistemas (LIS) e ferramentas de especificação/validação que estão sendo desenvolvidos paralelamente. O modelo de programação proposto para isto deverá se utilizar do suporte para reconfiguração existente no núcleo.

BIBLIOGRAFIA

(ACCETTA, 1986) M. Accetta, R. Baron, W. Boloshy, D. Golub, R. Rashid, A. Tevaniam and M. Young.

MACH: a New Kernel Foudation for Unix Development. Computer Science Department. Carnegie-Mellon University, Pittsburg. DRAFT. MAI 1986.

(ANANDA, 1984) A.A. Ananda and B.W. Marsden

A Network Operating System For Microcomputers. Computer Communications. Vol. 7 n.2. ABR 1984.pg 65-72

(ANDREWS, 1982) G.R. Andrews, F.B Schneider

Concepts and Notations for Concurrent Programming. Technical Report 82-520. Department of Computer Science, Cornell University. SET 1982.

(BALTER, 1986) R. Balter

Locally Distributed Operating Systems Analysis and Classification. Project Comands DSG/CRG/OSI BULL c/ o IMAG. DEZ 1986.

(BARNES, 1980) J.G.P. BARNES

An Overview Of ADA. Software-Pratice and Experience, Vol.10. JUL 1986. pg. 851-887

(BOOCH, 1986)

Object-Oriented Development. IEEE Transaction on Software Engineering. Vol SE-12, n.2. FEV 1986.

COELLO, 1986) J.M.A. COELLO

Suporte de Tempo Real para um Ambiente de Programção Concorrente. Dissertação de Mestrado. UNICAMP. AGO 1986.

(CROWDER, 1985) R. Crowder

The MAP Specification. Control Enginneering. OUT 1985.

(COX, 1986) B.J. Cox

Object Oriented Programming, An Evolutionary Approach. Addison-Wesley Publishing Company.1986.

(DAY, 1983) J.D. Day, H. Zimmermann

The OSI Reference Model. Proceedings of the IEEE. Vol. 71, No 2. DEZ 1983. pg 1334-1345.

(DEITEL, 1984) H.M. Deitel

An Introduction to Operating Systems. Addison-Wesley Publishing Company. JUL 1984.

(DE REMER, 1976) F. DeRemmer, H.H. Kron

Programming-in-the-Large versus Programming-in-the-small. IEEE Trans. on Software Engineering, Vol. SE-2, No 2.JUN 1876.

(DESWARTE, 1987) Y. Deswarte

Tendências da Pesquisa em Sistemas Distribuídos. Seminário de Automação Industrial, Florianópolis, Brasil. JUL 1987. pg 1-18.

(DIJKSTRA, 1968) E.W. Dijkstra

Co-operating Sequential Processes in Programming. Programming Languages, Academic Press, New York. 1986. pg 43-112

(EMILIANO, 1987) J.R.L. Emiliano e M.J. Mendes

Protocolos de Aplicação Industrial em Redes Locais de Computadores na Automação Industrial (Protocolos MAP-TOP). 5 Simpósio Brasiléiro de Redes de Computadores, São Paulo. ABR 1987

(ENSLOW, 1981) P.H. Enslow

"Distributed And Decentralized Control in Fully Distributed Control Systems". Agard Avionics Panel Symposium on "Tactical. Airborne Distributed Computing an Networks", Normay. JUN 1981. pg 17-1-17-18.

(FELDMAM, 1979) J.A. Feldman

High Level Programming of Distributed Computing. Commu. of the ACM, Vol. 22, No 6. JUN 1979. pg 363-368.

(FLETCHER, 1980) J.G. Fletcher, R.W. Watson

Service Suport for Network Operating System. COMPCON 80. 1980. pg 415-424.

(FORSDICK, 1978) H.C. Forsdick

Operating Systems for Computer Network. Computer 11(1). JAN 1987.pg 48-57

(FRAGA, 1988) J.S. Fraga, E.S. Silva, J.M. Farines, L.E. Souza e Luiz Nacamura J.

A Software Environment For Distributed Aplications. 15 IFAC/IFIC. Valencia SPAIN. 1988.

(GLAUBER, 1986) M. Glauber

The carrier Band Network And MINI-MAP: Low-Cost Solutions Control Engineering. OUT 1986. pg 30-34

(GLIGOR, 1983) V.D. Gligor, G.L. Luckenbaugh

An Assessment of the Real-Time Requirements for Programming Environment and Languages. IEEE Real-Time Systems Symposium, Arlindo, USA. DEZ 1983. pg 3-19.

(GOLBERG, 1984) A. Goldberg

Smalltalk -80- The Interactive Prograamming Environment. Addison-Wesley. 1984.

(HANSEN, 1978) B. Hansen

The Programming Language Concurrent Pascal. IEEE Transactions on Software on Software Enginnering, SE-1 No2. JUN 1975.

(HANSEN, 1978) B. Hansen

Distributed Process: A concurrent Programming Concept. Commu. of the ACM, Vol. 21, No 11. NOV 1978.

(HOARE, 1978) C.A.R Hoare

Communication Sequential Process. Commu. of the ACM, Vol. 21, N. 8. AGO 1978. pg 667-677.

(JONES, 1978b) A.K. Jones

The Object Model: A Conceptual toll for structuring Software. Lecture Notes in Computer Science, Vol. 60, Springer-Verlag. 1978. pg 7-16.

(KOPETZ, 1983) H. Kopetz and W. Merker

A Maintainable Real Time System. IEEE Computer Architecture Technical Committee. JUN 1985. pg 70-83.

(KRAMER, 1983) J. Kramer, J. Magge, M. Sloman, A. Lister

CONIC: An Integrated Aproach to Distributed Computer Control Systems. IEE Proc., Vol. 130, Pt_E, No 1. JAN 1983. pg 1-10.

(KRAMER, 1985) J. Kramer and J. Magge

"Dynamic Configuration of the Distributede Systems". IEEE Transaction on Software Enginnering SE-11(4). ABR 1985. pg. 425-436.

(LAPRIE, 1986) J.C Laprie

"DEPENDABILITY: Anifyng Concept For Realiable Computing and Fault Tolerance". LAAS REPORT No 86.357. DEZ 1986.

(LAMPSON, 1981) B.W. Lampson

Atomic Transactions. Lecture Notes in Computer Science, Distributed Systems - Architecture and Implementation, Edited by B.W. Lampson, M. Paul and H.J Siegert, Springer-Verlag, Berlin Heidelberg New York. 1981. pg 246-264.

(LAUER, 1978) H.C. Lauer, R.M. Needham

On the Duality of Operating System Structures. 2nd International Symposium on Operating Systems, IRIA. OTU 1978.

(LEE, 1985) I. Lee and Gehlot

Language Contructs for Distributed Real-Time Programming. IEEE Real-Time Systems Sysposium, San Diego, California. 1985. pg 5-66.

(LE LANN, 1981) G. Le Lann

Motivations, Objetives and Characterization of Distributed Systems. Lecture Notes in Computer Science, Distributed Systems - Architecture and Implementation, Edited by B.W. Lampson, M. Paul and H.J Siegert, Springer-Verlag, Berlin Heidelberg New York. 1981. pg 1-8.

(LISKOV, 1977) B. Liskov, A.S. Nyder, R. Atkinson and C. Schaffert

Abstraction Mechanisms in CLU, Communications of the ACM, V.20 No 8. AGO 1977. pg 564-576

(LISKOV, 1983) B. Liskov and M. Herlihy

Issue in Process and Communication Structure for Distributed Programs. III Symp. IEEE on Realibility in Distributed Software and Database System. 1983.pg 123-132.

(MACLEOD, 1982) I.M. MacLeod, M.G. Rood

Interprocess Communication Primitivas for Distributed for Distributed Process Control. IFAC Software for Computer Control, Madrid, Spain. 1982. pg 51-59.

(MAGGE, 1984) J.N. Magge

Provision of Flexibility in Distributed Systems. Ph. D. Thesis, Department of Computing, Imperial College of Science & Technology, London. ABR 1984.

(MAGGE, 1987) J.N. Magge, J. Kramer, M. Sloman

Construction Distributed Systems in CONIC. Research Report Doc. 87/4, Department of Computing, Imperial College, London. MAR 1987.

(NATARAJAN, 1985) N. Natarajan

Communication and Synchronization Primitives for Distributed Programs. IEEE Trans. on Software Enginnering, Vol. SE-11, No 4. ABR 1985. pg 396-416.

(NELSON, 1981) B.J. Nelson

"Remote Procedure Call". Ph. D. Thesis Carnegie-Mellon University n. 81-119. 1981.

(PRINCE, 1981) S.M. Prince and M.S. Sloman

Communication Requirements of a Distributed Computer Control System. IEE Proc., Vol 128, Pt_E, No 1. JAN 81. pg 21-34.

(RASHID 81) R.F Rashid and G.G. Robertson

Accent: A Communication Oriented Network Operating System Kernel. VIII Symp. on Operacional Systems Principles, California, USA. DEZ 1981

(READY, 1986) J.F. Ready

VRTX: A Real_Time Operating System for Embedded Microprocessor Applications. IEEE MICRO. 1986.pg 8-17.

(SCOTT, 1983) M.L. Scott

Message vs. Procedures in False Dictomy. Singplan Notices, V18. MAI 1983.

(SLOMAN, 1986) M. Sloman, J. Kramer, J. Magge, K. Twidle

Flexible Communication Structure for Distributed Embedded Systems. IEE Proc., Vol. 133, Pr_E, Nº 4. JUL 1986. pg 201-211.

(SILVA, 1988) E.S. Silva

Projeto e Implementação de Uma Linguagem de Componentes de Software para Sistema Distribuídos de Controle. Dissertação de Mestrado. Em desenvolvimento.

(SOLOMON, 1979) M.H. SOLOMON and R.A. FINKEL

The Roscoe Distributed Operating System. Procceding of the 7th Symp on Operating Systems Principles, California. DEZ 1979. pg. 108-144.

(SOUZA, 1988) L.E. de Souza

Projeto e Implementação de um Configurador para Sistema Distribuídos. Dissertação de Mestrado. Em desenvolvimento

(SPECTOR, 1982) A.F. Spector

Perfoming Remote Operation Efficiently on a Local Computer Network. Communications of The ACM, 25(4). ABR 1982. pg 246-260.

(WALKER, 1983) B. Walker, G. Popek, R. English, C. Kline and G. Thiel.

The LOCUS distributed Operating System. Proc 9 th SOPS, Operating Systems Review, 17. 1983. pg 49-70

(WATSON, 1981) R.W. Watson

Distributed System Architecture Model. Lecture Notes in Computer Science, Distributed Systems - Architecture and Implementation, Edited by B.W. Lampson, M. Paul and H.J Siegert, Springer-Verlag, Berlin Heidelberg New York. 1981. pg 10-43.

(WIRTH, 1985) N. Wirth

Programming in Module-2. Springer-Verlag Berlin Heidelberg Network Tokio.

(ZIMMERMANN, 1984) H. Zirmmermann, M. Guillemont, G. Morisset, J.S. Banino

Chorus: A Communication and Processing Architecture for Distributed Systems. INRIA Report Nº 328. France. SET 1984.

APÊNDICE 1

EXEMPLO - CÉLULA FLEXÍVEL

1.1 Descrição do Exemplo

De forma a verificar a funcionalidade do núcleo tempo real, quando envolvido com aplicações distribuídas tempo real, foi testado uma simulação de sistema de monitoração e supervisão. O sistema proposto consiste de uma célula flexível (ver figura A.1.) formada das seguintes componentes:

- uma esteira;
- uma camera;
- dois robôs: Rp e Ra;
- um torno; e
- um armazém.

A esteira possui duas posições de parada:

- P1: para que a peça possa ser visualizada pela camera; e
- P2: para que a peça possa ser retirada da esteira pelo robô Rp;

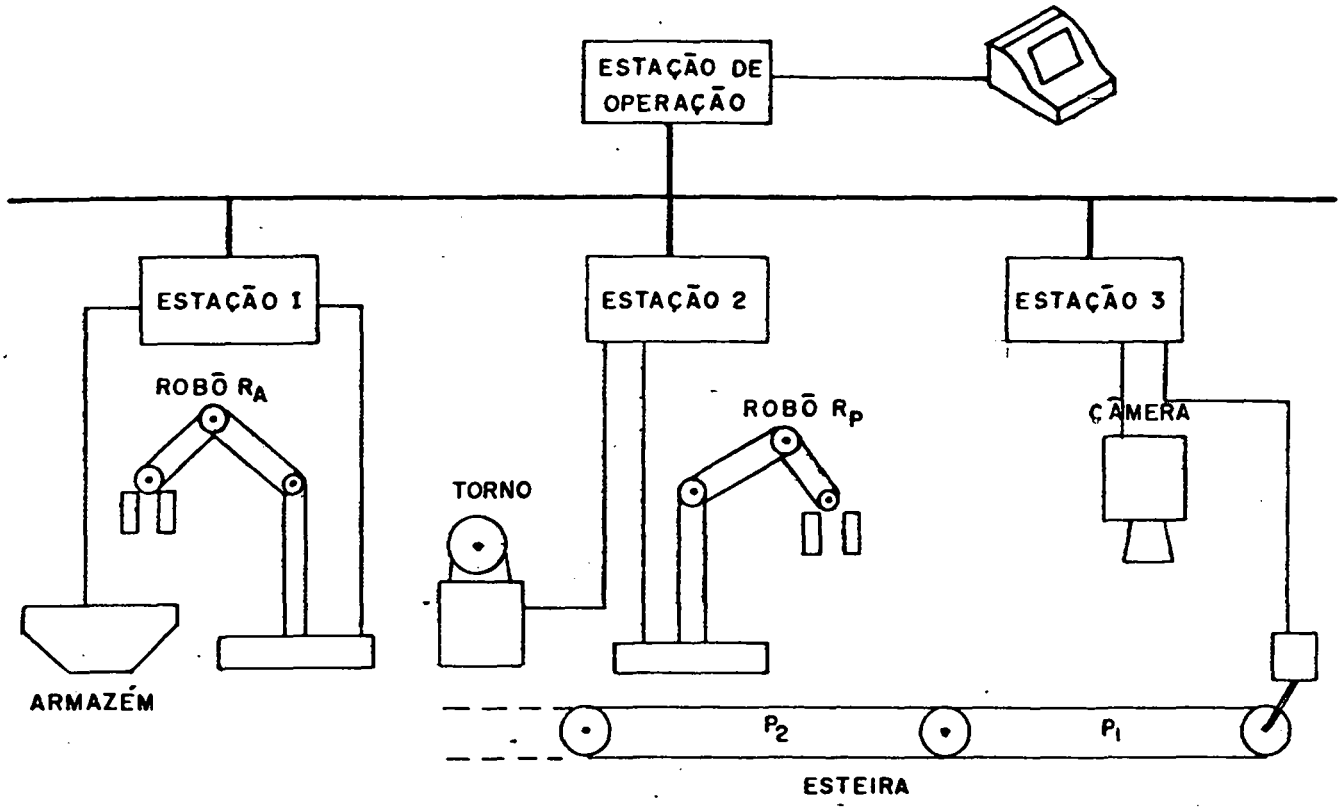


figura A.1 - Célula Flexível

As posições S_Rp corresponde a posição de repouso do robô Rp, sendo que a posição S_Ra é posição de repouso do robô Ra.

O sistema consiste de uma esteira que transporta peças de diversos tipos. A distribuição de peças sobre a esteira é aleatória. Quando a esteira encontra-se na posição P1, o movimento da esteira é interrompido para que seja feita a visualizaçào da peça, pela camera. Uma vez terminada a visualização a esteira é novamente colocada em movimento. A informação obtida por esta visualização é então classificada através de parâmetros pré-estabelecidos. Se as características visualizadas não se enquadra em nenhum destes parâmetros a peça é rejeitada. Quando a peça encontra-se na posição P2, o movimento da esteira é novamente interrompido para que a peça

seja retirada pelo robô Rp. Várias peças podem se encontrar entre P1 e P2. Quando a esteira encontra-se em repouso, para que o robô Rp retire a peça, é necessário que a informação da localização (direita, esquerda e centro) da peça tenha sido recebida. Entretanto se a peça foi rejeitada pela visualização esta continua na esteira (não é pega pelo robô). Um vez retirada de P2 (peça válida), a esteira é posta em movimento.

Para que a peça possa ser colocada no torno é necessário que o torno esteja disponível e o robô Ra não se encontre nas proximidades do torno. Após a colocação das peça no torno o robô Rp retorna a posição de repouso, estando assim disponível para retirar outra peça da esteira. Ao final do torneamento a peça pode ser retirada do torno pelo robô Ra, desde que o robô Rp não se encontre nas proximidades do torno. Uma vez retirada a peça o robô Ra coloca a peça no armazém. No final da colocação o robô Ra retorna a posição de repouso.

1.2. Decomposição do Sistema em Módulos

Baseado na descrição do item anterior a aplicação (célula flexível) foi decomposta em vários módulos, sendo que cada um destes representa operações executadas por componentes ou em componentes do sistema (robô, esteira, etc). De forma a simular funcionamento real de cada componente foram incorporadas também na aplicação módulos simuladores. A configuração de módulos (tarefas) em 4 estações é mostrada na figura .2.

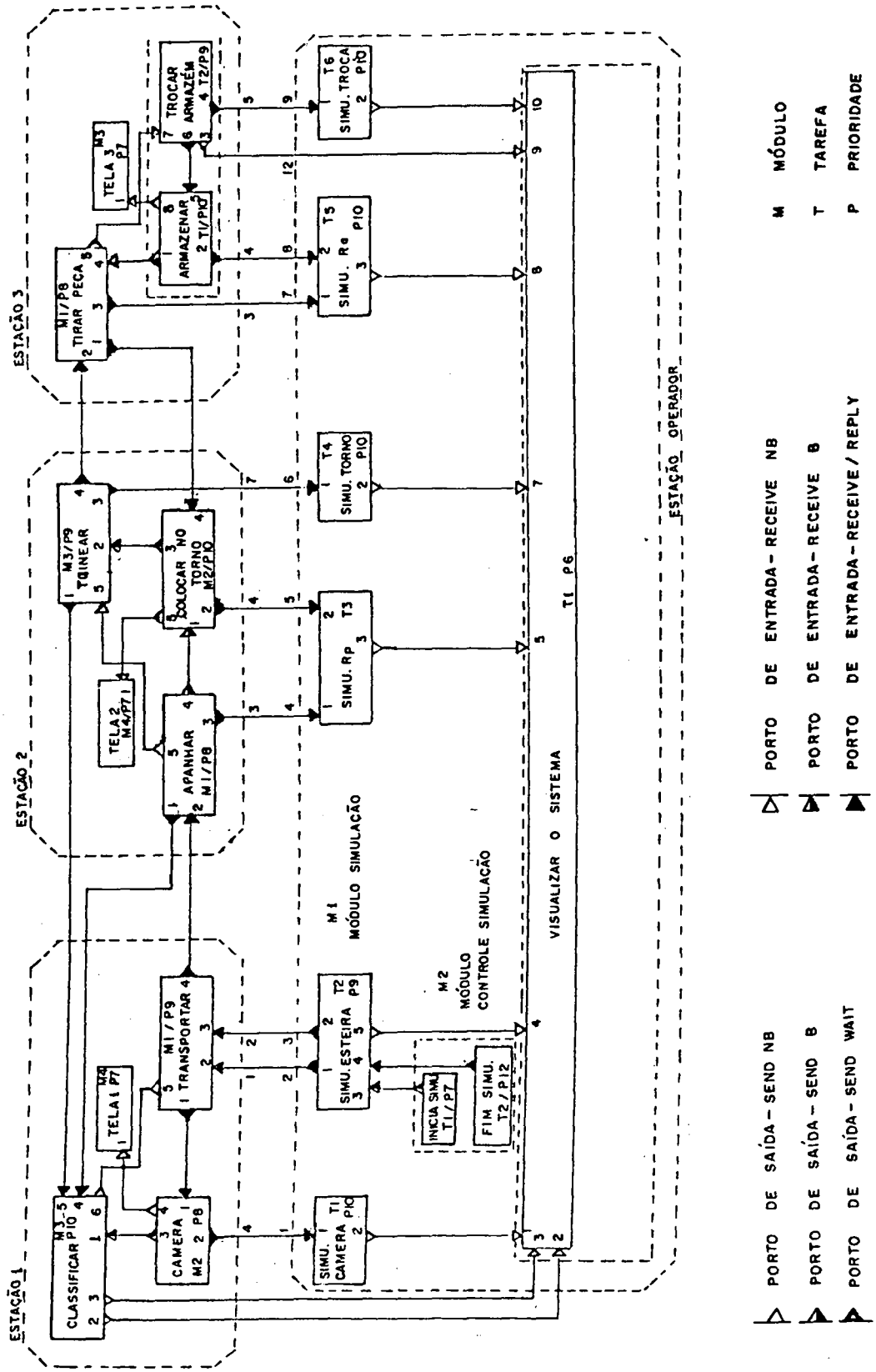


figura A.2. - Decomposição de Aplicação Célula Flexível em Módulos

Para representação dos módulos, tarefas, portos e operações de comunicação será utilizada uma Pseudo-Linguagem que utiliza como base a linguagem C. A estrutura de um módulo é mostrada a seguir:

```
MODULO  nome_do_módulo
    PORTO_MÓDULO
      nome: tipo (msg:tipo / rsp:tipo)
         parâmetros:  num_msg  portos de entrada SEND_NB;
      /* declaração dos portos do módulo */
TAREFA nome da tarefa_1;
    PORTO_TAREFA  nome da tarefa
     /*  declaração dos portos da tarefa1 */
(
/* declaração das variáveis */
|
/* corpo da tarefa */
  REPEAT
    |
  UNTIL FALSE
) /*END_tarefa_1 */
TAREFA nome da tarefa_2;
    PORTO_TAREFA
     /*  declaração dos portos da tarefa 2 */
(
/* declaração das variáveis */
|
/* corpo da tarefa 2 */
  REPEAT
    :
  UNTIL FALSE
) /* END_tarefa_2 */
LINK internos ao módulo;
END_MÓDULO.
```

As extensões propostas para representar chamadas de funções são:

- **RECEBE:** recebe uma mensagem, em um porto. O parâmetro TIMEOUT determina se a recepção é bloqueante (igual a -1 ou maior que zero) ou não (0). A recepção seletiva será apresentada através de vários comandos RECEBE;

- **ENVIA:** envia uma mensagem, para um porto. O parâmetro TIMEOUT determina se o envio é bloqueante ou não. O comando ESPERA após o envio indica que a tarefa aguarda resposta (SEND_WAIT);

- **RESPONDE:** envia uma mensagem de resposta para um porto de entrada, que anteriormente recebeu uma mensagem;

- **INICIA_INTERRUPÇÃO:** aloca um nível de interrupçào lógica;

- **ESPERA_INTERRUPÇÃO:** para depois de iniciar uma interrupção de nível lógico aguardar pela interrupção;

- **ATRASO:** coloca a tarefa em espera por um tempo determinado.

```
MODULO TRANSPORTAR
  PORTO_MÓDULO
    MTP1: PS "SEND_WAIT" (msg1:float       / rsp1:signal_type);
    MTP2: PE "SEND_WAIT" (msg2:float       / rsp2:sigal1_type);
    MTP3: PE "SEND_WAIT" (msg3:signal_type / rsp3:signal_type);
    MTP4: PS "SEND_WAIT" (msg4:signal_type / rsp4:signal_type);
    MTP5: PS " SEND_NB " (msg5:string_type);   /* visualização
TAREFA transportar (P=9, SP=1024);                      local */
  PORTO_TAREFA
    /*  corresponde aos mesmos do módulo */
(
float  msg1,
       msg2;     /* msg1 = msg2 = informação da peça em P1 */
char   msg3;
char   msg5 (80);      /* string_type */
  REPEAT
    RECEBE msg2 no MTP2  ou
      RECEBE msg3 no MTP3 com TIMEOUT = -1;
    IF recebeu msg1  THEN       /* presença de peça em P1 */
                        (
                          msg5= "presença de peça em P1";
                          ENVIA mg5 para MTP5 com TIMEOUT = 0;
                          ENVIA msg2 para MTP1 e
                            ESPERA signal_type com TIMEOUT = -1;
                          RESPONDE signal_type para MTP2;
                          msg5= "fim de visualização";
                          ENVIA msg5 para MTP5 com TIMEOUT = 0;
                        )
```

```
    ELSE    /* presença de peça em P2 */
      (
        msg5= "presença de peça em P2";
        ENVIA msg5 para MTP5 com TIMEOUT = 0;
        ENVIA signal_type para MTP4 e
          ESPERA signal_type com TIMEOUT = -1;
        RESPONDE signal_type para TP3;
        msg5= "fim de apanhar"
        ENVIA msg5 para MTP5 com TIMEOUT = 0;
      END_(
  ULTIL FALSE
) /*END_tarefa_transportar */
END_MÓDULO_TRANSPORTAR.
```

```
*************************************************************
MODULO  CAMERA      /* Declaração de módulo */
  PORTO_MÓDULO
   MTP1: PE "SEND_WAIT" (msg1:float          / rsp1:signal_type);
   MTP2: PS "SEND_WAIT" (msg2:signal_type    / rsp2:int);
   MTP3: PS " SEND_B  " (msg3:info_type);
   MTP4: PS " SEND_NB " (msg4:string_type);      /* visualização
TAREFA   camera;                                          local */
  PORTO_TAREFA
   /*  corresponde ao mesmos do módulo */
(
/* Declaração de variáveis */
real           msg1;     /* informação sobre a peça em P1 */
int            rsp2;     /* informação sobre a posição da peça */
struct info_type
(  int   loc;              /* posição da peça */
   float p1;               /* informação sobre a peça */
)msg3;                     /* informação sobre a peça */
string_type    msg4 (80);  /* string_type */
/* inicializações de variáveis */
  REPEAT
    msg4 = "camera ativa";
    ENVIA  msg4  para  MTP4  com TIMEOUT = 0;
    RECEBE  msg1  no MTP1  com  TIMEOUT = -1;
    msg4 = "CAMERA : Presença de peça em P1";
    ENVIA  msg4  para  MTP4  com  TIMEOUT = 0;
    ENVIA  signal_type para  MTP2  e
      ESPERA   rsp2  TIMEOUT = -1;
    /* preenche os campos da msg3 */
    ENVIA  msg3  para  MTP3  com  TIMEOUT = -1;
    RESPONDE  signal_type  para  MTP1;
  UTIL FALSE;
)  /*END_tarefa_camera*/
END_MÓDULO_CAMERA.
```

```
************************************************************
MODULO  CLASSIFICAR    /* declaração de módulo */;
  PORTO_MÓDULO
   MTP1: PE " SEND_NB " (msg1:info_type)  10 ;
   MTP2: PS " SEND_NB " (msg2:char);
   MTP3: PS " SEND_NB " (msg3:int);
   MTP4: PE "SEND_WAIT" (msg4:signal_type  /  rsp4:clas_type);
   MTP5: PE "SEND_WAIT" (msg5:signal_type  /  rsp5:char);
   MTP6: PS " SEND_NB " (msg6:string_type);
TAREFA  classificar;
  PORTO_TAREFA
   /*  corresponde ao mesmos do módulo */
(
/* declarações de variáveis */
struct info_type
( int    loc;       /* informação sobre a posição da peça */
  float  pl ;       /* informação sobre a peça */
) msg1;
struct clas_type
( char   posição,   /* posição da peça */
         validade; /* validade da peça */
) rsp4;
char   rsp5,         /*  rsp 5 é igual ao tipo de peça  */
       msg2,         /* estado da tarefa (visualização global)*/
       msg5 (80);    /* string type */
int    msg3 (4);     /* número de peça de cada tipo */
  REPEAT
    RECEBE msg1 no MTP1 com TIMEOUT = -1;
    msg6 = "CLASSIFICAR : sinal da camera chegou";
    ENVIA msg6 para MTP6 com TIMEOUT = 0;
    ATRASO = 1;
    msg2 = "1";           /*  classificando  */
    ENVIA msg2 para MTP2 com TIMEOUT = 0;
    /* posiciona a peça esteira */
```

```
    msg6 = "CLASSIFICAR : espera sinal de apanhar";
    ENVIA msg6 para MTP6 com TIMEOUT = 0;
    RECEBE signal_type no MTP4 com TIMEOUT = -1;
    msg6 = "CLASSIFICAR : sinal de Rp chegou";
    ENVIA msg6 para MTP6 com TIMEOUT = 0;
    RESPONDE rsp4 para MTP4;
    msg6 = "CLASSIFICAR : respondeu a Rp";
    ENVIA msg6 para MTP6 com TIMEOUT = 0;
    /*  classificação da peça (rsp5) */
    RECEBE signal_type no MTP5 com TIMEOUT = -1;
        RESPONDE rsp5 para MTP5;
    msg3 = tipo de peça;
    ENVIA msg3 para MTP3 com TIMEOUT = 0;
    msg6 = "CLASSIFICAR : mandou mensagem para visualizar";
    ENVIA msg6 para MTP6 com TIMEOUT = 0;
    msg2 = "0";      /*  repouso   */
    ENVIA msg2 para MTP2 com TIMEOUT = 0;
  UNTIL FALSE
) /* END_tarefa_classificar */
END_MÓDULO_CLASSIFICAR.
```

```
**********************************************************

MODULO  APANHAR   /*  Declaração de módulo  */

  PORTO_MÓDULO

   MTP1: PS "SEND_WAIT" (msg1:signal_type  /  rsp1:clas_type);

   MTP2: PE "SEND_WAIT" (msg2:signal_type  /  rsp2:signal_type);

   MTP3: PS "SEND_WAIT" (msg3:char         /  rsp3:signal_type);

   MTP4: PS " SEND_B  " (msg4:signal_type);

   MTP5: PS " SEND_NB " (msg5:string_type);   /*  visualização

TAREFA   apanhar;    /* Declaração de tarefa  */        local */

  PORTO_TAREFA  _/* declarações de portos da tarefa */
   /*  corresponde aos mesmos do módulo */

(

/* declarações de variáveis */

|struct clas_type

( char   posição,

         validade;

) rsp1;          */* informação da peça em P2 */

char   msg3,          /* tipo de peça */

       msg5 (80);     /* string_type */

  REPEAT

    msg5 = "APANHAR : espera sinal transportar";

    ENVIA msg5 para MTP5 com TIMEOUT = 0;

    RECEBE signal_type no MTP2 com TIMEOUT = -1;

    msg5 = "APANHAR : presença de peça em P2";

    ENVIA msg5 para MTP5 com TIMEOUT = 0;

    ENVIA signal_type para MTP1 e

    ESPERA rsp1 com TIMEOUT = -1;

    msg5 = "APANHAR : recebeu informação sobre a peça";

    ENVIA msg5 para MTP5 com TIMEOUT = 0;
```

```
    IF peça válida THEN
                        (
                          msg3 = posição
                          ENVIA msg3 para MTP3 e
                            ESPERA signal_type com TIMEOUT = -1;
                        )
    RESPONDE signal_type para MTP2;  /* peça retiada */
    IF peça válida THEN
       ENVIA signal_type para MTP4 com TIMEOUT = -1;
  UNTIL FALSE
) /* END_tarefa_apanhar */
END_MÓDULO_APANHAR.
```

```
*****************************************************************
MODULO  TIRAR_PEÇA
  PORTO_MÓDULO
   MTP1: PS "SEND_WAIT" (msg1:signal_type  /  rsp1: ignal_type);
   MTP2: PE "SEND_WAIT" (msg2:signal_type  /  rsp2:signal_type);
   MTP3: PS "SEND_WAIT" (msg3:signal_type  /  rsp3:signal_type);
   MTP4: PS " SEND_B  " (msg4:signal_type);
   MTP5: PS " SEND_NB " (msg5:string_type);  /*visualização
TAREFA   tirar_peça;                                        local */
  PORTO_TAREFA
  /*  corresponde aos mesmos do módulo */
(
char  msg5 (80);   /* string_type */
   REPEAT
     ENVIA signal_type para MTP1 e
       ESPERA signal_type com TIMEOUT = -1;
     msg5= "TIRAR: espera peça.";
     ENVIA msg5 para MTP5 com TIMEOUT = 0;
     RECEIVE signal_type no MTP2 com TIMEOUT = -1;
     msg5= "TIRAR: recebeu sinal para retirar peça.";
     ENVIA msg5 para MTP5 com TIMEOUT = 0;
     ENVIA signal_type para MTP3 e
       ESPERA signal_type com TIMEOUT = -1;
     msg5= "TIRAR: espera novo sinal de torneamento";
     ENVIA msg5 para MTP5 com TIMEOUT = 0;
     RESPONDE signal_type para MTP2;
   UNTIL FALSE
) /* END_tarefa_tirar_peça */
END_MÓDULO_TIRAR_PEÇA.
```

```
*********************************************************************
MÓDULO  SIMU_RP
  PORTO_MÓDULO
   MTP1: PE "SEND_WAIT" (msg1:char        /  rsp1:signal_type);
   MTP2: PE "SEND_WAIT" (msg2:signal_type /  rsp2:signal_type);
   MTP3: PS " SEND_NB " (msg3:char);
TAREFA   simu_Rp;

  PORTO_TAREFA
   /* corresponde aos mesmos do módulo */
(
char msg3;
  REPEAT
    RECEBE  signal_type no MTP1 com TIMEOUT = -1
    msg3= "2";     /* robô Rp apanhando */
    ENVIA msg3 para MTP3 com TIMEOUT = 0;
    ATRASO (10);
    RESPONDE signal_type para MTP1;
    RECEBE  signal_type no MTP1 com TIMEOUT = -1
    msg3= "1";     /* robô Rp colocando peça no torno */
    ENVIA msg3 para MTP3 com TIMEOUT = 0;
    ATRASO (20);
    msg3= "1";     /* robô Rp em repouso */
    ENVIA msg3 para MTP3 com TIMEOUT = 0;
    RESPONDE signal_type para MTP1;
  UNTIL_FALSE
) /* END_tarefa_simu_Rp */
END_MÓDULO_SIMU_RP;
```

```
*********************************************************************
MÓDULO SIMU_RA
  PORTO_MÓDULO
   MTP1: PE "SEND_WAIT" (msg1:signal_type  /  rsp2:signal_type);
   MTP2: PE "SEND_WAIT" (msg2:signal_type  /  rsp3:signal_type);
   MTP3: PS " SEND_NB " (msg3:char);
TAREFA   simu_Ra;   /* Declaração de tarefa */
   PORTO_TAREFA
    /* corresponde ao mesmos do módulo */
(
char  msg3;   /* estado da tarefa */
   REPEAT
     msg3= "0";  /* em repouso */
     ENVIA msg3 para MTP3 com TIMEOUT = -1;
     RECEBE signal_type no MTP1 com TIMEOUT = -1;
     msg3= "1";  /* retirando peça do torno */
     ENVIA msg3 para MTP3 com TIMEOUT = -1;
     ATRASO (6);
     RESPONDE signal_type para MTP1;
     RECEBE signal_type na MTP2 com TIMEOUT = -1;
     msg3= "2";  /* armazenando peça */
     ENVIA msg3 para MTP3 com TIMEOUT = -1;
     ATRASO (12);
     RESPONDE signal_type para MTP2;
   UNTIL_FALSE
) /* END_tarefa_simu_Ra */
END_MÓDULO_SIMU_RA.
```

```
*****************************************************************
MÓDULO  SIMU_TORNO
  PORTO_MÓDULO
   MTP1: PE "SEND_WAIT" (msg1:char  /  rsp1:signal_type);
   MTP2: PS " SEND_NB " (msg2:char);
TAREFA   simu_torno;
  PORTO_TAREFA
   /* corresponde aos mesmos do módulo */
(
char  msg1,    /* tipo de peça */
      msg2;
   REPEAT
     RECEBE msg1 no MTP1 com TIMEOUT = -1;
     msg2= "1";   /* torneando */
     ENVIA msg2 para MTP2 com TIMEOUT = 0;
     SWITCH  (msg1)
     (
       case (peça_1) ( ATRASO (10);
                       break;)
       case (peça_2) ( ATRASO (20);
                       break;)
       case (peça_3) ( ATRASO (30);
                       break;)
     )
     msg2= "0";   /* torno em repouso */
     ENVIA msg2 para MTP2 com TIMEOUT = 0;
     RESPONDE signal_type para MTP1;
   UNTIL FALSE
) /* END_tarefa_simu_torno */
END_MÓDULO_SIMU_TORNO.
```

```
**********************************************************************
MÓDULO SIMULA ESTEIRA
   PORTO_MÓDULO
     MTP1: PS "SEND_WAIT" (msg1: float / rsp1:signal_type);
     MTP2: PS "SEND_WAIT" (msg2: float / rsp2:signal_type);
     MTP3: PE " SEND_B  " (msg3: signal_type);
     MTP4: PE "SEND_WAIT" (msg4: signal_type);
     MTP5: PS " SEND_NB " (msg5: char);
TAREFA   simu_esteira;
  PORTO_TAREFA
    /* corresponde aos mesmos do módulo */
(
float  msg1;
char   msg5;   /* representação da esteira */
RECEIVE signal_type no MTP1 com TIMEOUT = -1;  /* início de
/*  inicia representação da esteira */              simulação */
  REPEAT
    /* simula o movimento da esteira */
    RECEBE signal_type no MTP4 com TIMEOUT = -1;
    IF signal_type recebido THEN
                              (
                               /* fim de simulação */
                              )
    ELSE
      /* distribuição aleatória */
      /* atualiza a distribuição na esteira */
      IF peça em P1  THEN
                       ENVIA msg1 para MTP1 e
                         ESPERA signal_type com TIMEOUT = -1;
      IF peça em P2  THEN
                        ENVIA msg1 para MTP1 e
                          ESPERA signal_type com TIMEOUT = -1;
      ENVIA msg5 para MTP5 com TIMEOUT = 0;
   UNTIL FALSE
) /* END_tarefa_simula_esteira */
END_MÓDULO_SIMULA_ESTEIRA
```

```
* ***************************************************************
MÓDULO  SIMU_CAMERA
     PORTO_MÓDULO
      TP1: PE "SEND_WAIT" (msg1:signal_type / rsp1: int);
      TP2: PS " SEND_NB " (msg2:char);
TAREFA   simu_camera;
  PORTO_TAREFA
   /* corresponde aos mesmos do módulo */
(
int   rsp1;
char  msg2;    /* estado da tarefa */
   REPEAT
     RECEIVE signal_type no MTP1 com TIMEOUT = -1;
     msg2= "1";  /* tarefa visualizando */
     ENVIA msg2 para MTP2 com TIMEOUT = 0;
     ATRASO (20);
     /* geração de número aleatório */
     /* rsp1 = informação para posicionar a peça */
     RESPONDE rsp1 para MTP1;
     msg2= "0";  /* tarefa em repouso */
     ENVIA msg2 para MTP2 com TIMEOUT = 0;
   UNTIL_FALSE
) /* END_tarefa_simu_camera */
END_MÓDULO_SIMU_CAMERA
```

```
*************************************************************
MODULO  TORNEAR   /* Declaração de módulo */
  PORTO_MÓDULO
   MTP1: PS "SEND_WAIT" (msg1:signal_type / rsp1:char);
   MTP2: PE " SEND_B  " (msg2:signal_type);
   MTP4: PS "SEND_WAIT" (msg3:char        / rsp3:signal_type);
   MTP5: PS "SEND_WAIT" (msg4:signal_type/  rsp4:signal_type);
   MTP7: PS " SEND_NB " (msg5:string_type); /*visualização
TAREFA   tornear;                                    local */
(
char  rsp1,   /* tipo de peça */
      msg3,
      msg5 (80); /* string_type */
  REPEAT
    RECEBE signal_type no MTP2 com TIMEOUT = -1;
    msg5= "TORNEAR: presença de peça.";
    ENVIA msg5 para MTP5 com TIMEOUT = 0;
    ENVIA signal_type para MTP1 e
      ESPERA rsp1 com TIMEOUT = -1;
    msg5= "TORNEAR: informação sobre a peça foi recebida.";
    ENVIA msg5 para MTP5 com TIMEOUT = 0;
    ENVIA msg3 para MTP3 e
      ESPERA signal_type com TIMEOUT = -1;
    msg5= "TORNEAR: fim de torneamento.";
    ENVIA msg5 para MTP5 com TIMEOUT = 0;
    ENVIA signal_type para MTP5 e
      ESPERA signal_type com TIMEOUT = -1;
    msg5= "TORNEAR: fim de torneamento.";
    ENVIA msg5 para MTP5 com TIMEOUT = 0;
  UNTIL FALSE
) /* END_tarefa_tornear */
END_MÓDULO_TORNEAR.
```

```
*****************************************************************
MODULO  COLOCAR_NO_TORNO
  PORTO_MÓDULO
   MTP1: PE " SEND_B  " (msg1:signal_type);
   MTP2: PS "SEND_WAIT" (msg2:signal_type  /  rsp2:signal_type);
   MTP3: PS " SEND_NB " (msg3:signal_type);
   MTP4: PE "SEND_WAIT" (msg4:signal_type  /  rsp4:signal_type);
   MTP5: PS " SEND_NB " (msg5:string_type);  /* visualização
 TAREFA   colocar_no_torno;                                  local */
  PORTO_TAREFA
   /*  corresponde aos mesmos do módulo */
(
char  msg (80);   /*  string_type */
(
  REPEAT
    RECEBE signal_type no porto MTP4 com TIMEOUT = -1;
    msg5 = "COLOCAR: redor do torno livre",
    ENVIA msg5 para MTP5 para TIMEOUT = 0;
    RECEBE signal_type no MTP1 com TIMEOUT = -1;
    msg5 = "COLOCAR: robô Rp disponível para colocar";
    ENVIA msg5 para MTP5 para TIMEOUT = 0;
    ENVIA signal_type para MTP3 com TIMEOUT = -1;
    RESPONDE signal_type MTP4;
    RESPONDE signal_type MTP1;
  UNTIL FALSE
) /* END_tarefa_colocar_no_torno */
END_MÓDULO_COLOCAR_NO_TORNO.
```

```
*************************************************************
MODULO  VISUALIZAR_o_SISTEMA   /* Declaração de módulo */
  PORTO_MÓDULO
   MTP1: PE  " SEND_NB "  (msg1:char)  5 ;
   MTP2: PE  " SEND_NB "  (msg2:char)  5 ;
   MTP3: PE  " SEND_NB "  (msg3:char)  5 ;
   MTP4: PE  " SEND_NB "  (msg4:char)  5 ;
   MTP5: PE  " SEND_NB "  (msg5:char)  5 ;
   MTP6: PE  " SEND_NB "  (msg6:char)  5 ;
   MTP7: PE  " SEND_NB "  (msg7:char)  5 ;
   MTP8: PE  " SEND_NB "  (msg8:char)  5 ;
   MTP9: PE  " SEND_NB "  (msg9:char)  5 ;
TAREFA   visualizar_o_sistema;
  PORTO_TAREFA
    /* corresponde aos mesmos do módulos */
(
char msg1, msg2, msg3, msg4, msg5, msg6, msg7,
     msg8, msg9;
/* inicializa a tela com a estrutura */
   REPEAT
     RECEBE msg1 no MTP1 ou
       RECEBE msg2 no MTP2 ou
         RECEBE msg3 no MTP3 ou
           RECEBE msg4 no MTP4 ou
             RECEBE msg5 no MTP5 ou
               RECEBE msg6 no MTP6 ou
                 RECEBE msg7 no MTP7 ou
                   RECEBE msg8 no MTP8 ou
                     RECEBE msg9 no MTP9 com TIMEOUT = -1;
    SWITCH (msg)
      CASE msg1: ( /* camera em repouso ou visualizando */)
                 break;
      CASE msg2: ( /* classificar em repouso ou ativa   */)
                 break;
```

```
      CASE msg3: ( /* número de peças */)
                 break;
      CASE msg4: ( /* representação do movivemto da esteira */)
                 break;
      CASE msg5: ( /* robô Rp em respouso ou apanhando ou
                      colocando */)
                 break;
      CASE msg6: ( /* torno torneando ou em repouso */)
                 break;
      CASE msg7: ( /* Robô Ra em repouso, retirando
                      ou armazenando */);
                 break;
      CASE msg8: ( /* múmero de peças no armazém */)
                 break;
      CASE msg9: ( /* armazém disponível ou em troca */)
                 break;
   REPEAT_FALSE
) /* END_tarefa_visualizar_o_sistema */
END_MÓDULO_VISUALIZAR_O_SISTEMA.
```

```
*************************************************************
MODULO  ARMAZENAR     /* Declaração de módulo */
  PORTO_MÓDULO
   MTP1: PE "SEND_WAIT" (signal_type);
   MTP2: PS "SEND_WAIT" (signal_type/signal_type);
   MTP3: PS " SEND_NB " (int);
   MTP4: PS "SEND_WAIT" (signal_type/signal_type);
   MTP7: PS " SEND_NB " (string_type);
   MTP8: PS " SEND_NB " (string_type);
TAREFA   armazenar;
  PORTO_TAREFA
   TP1: PE " SEND_NB " (msg1:signal_type);
   TP2: PS "SEND_WAIT" (msg2:signal_type  /  rsp2:signal_type);
   TP5: PE "SEND_WAIT" (msg5:signal_type  /  rsp5:signal_type);
   TP8: PS " SEND_NB " (msg8:string_type);
(
char  msg8 (80);
   REPEAT
     msg8= "ARMAZENAR: espera sinal de tirar.";
     ENVIA msg8 para TP8 com TIMEOUT = 0;
     RECEBE signal_type no TP1 com TIMEOUT = -1;
     msg8= "ARMAZENAR: espera sinal de trocar.";
     ENVIA msg8 para TP8 com TIMEOUT = 0;
     RECEBE signal_type no TP5 com TIMEOUT = -1;
     msg8= "ARMAZENAR: armazém disponivel.";
     ENVIA msg8 para TP8 com TIMEOUT = 0;
     ENVIA signal_type para TP2;
       ESPERA signal_type com TIMEOUT = -1;
     msg8= "ARMAZENAR: fim de armazenagem.";
     ENVIA msg8 para TP8 com TIMEOUT = 0;
     RESPONDE signal_type para TP5;
   UNTIL FALSE
) /* END_TAREFA_ARMAZENAR */
```

```
TAREFA   trocar_armazém;
  PORTO_TAREFA
   TP3: PS " SEND_NB " (msg1:int);
   TP4: PS "SEND_WAIT" (msg2:signal_type  /  rsp2:signal_type);
   TP6: PS "SEND_WAIT" (msg3:signal_type  /  rsp3:signal_type);
   TP7: PS " SEND_NB " (msg4:string_type);
(
int    msg3;            /* número de peça */
char   msg7 (80);       /* string_type */
   REPEAT
     ENVIA signal_type para TP6 e
       ESPERA signal_type com TIMEOUT = -1;
     /* incrementa o número de peça */
     msg7= "TROCAR ARM: mais uma peça";
     ENVIA msg7 para TP7 com TIMEOUT = 0;
     IF armazém completo
        THEN
          (
             ENVIA signal_type para TMP4 e
               ESPERA signal_type com TIMEOUT =0;
             msg7= "TROCAR ARM: troca armazém";
             ENVIA msg7 para TP7 com TIMEOUT = 0;
          )
      UNTIL FALSE
) /* END_tarefa_trocar_armazém */
LINK  MT5 TO MT6
END_MÓDULO_ARMAZENAR
```

```
*****************************************************************
MODULO  CONTROLE_SIMULAÇÃO   /* Declaração de módulo */
  PORTO_MÓDULO
   MTP1: PS " SEND_NB " (signal_type);
   MTP2: PS " SEND_NB " (signal_type);
TAREFA   inicia_simu;
  PORTO_TAREFA
   TP1: PS " SEND_NB " (signal_type);
(
  /* espera F1 ser pressionada
  ENVIA signal_type para MTP1 com TIMEOUT =0;
  SUSPENDE_TAREFA;
) /* END_tarefa_controle_simulação */
TAREFA   fim_simu;
  PORTO_TAREFA
   TP2: PS " SEND_NB " (signal_type);
(
  ind_intr= INICIA_INTERRUPÇÃO (0X1b)
  ESPERA_INTERRUPÇÃO (ind_intr);
  SUSPENDE_TAREFA;
) /* END_tarefa_fim_simu */
END_MÓDULO__FIM_SIMU
```

APENDICE 2

FUNÇÕES DO NÚCLEO

As funções do núcleo, disponíveis as tarefas de aplicação estão relacionadas com:

- TIPO MÓDULO;
- INSTÂNCIA DE MÓDULO;
- TAREFA;
- PORTO;
- LIGAÇÃO ENTRE PORTOS;
- TEMPORIZAÇÃO; e
- ESCALONAMENTO.

1. Funções relacionadas ao tipo módulo

1.1. Criação de um bloco de controle de tipo módulo

A função "CRIA_BC_TIP_MOD", gera dinamicamente, um bloco de controle de tipo módulo.

FORMATO DE CHAMADA:

int **CRIA_BC_TIP_MOD** (ind_tip_mod, cs_tam, ds_tam, ss_tam, param_tam, iniciação);

Parâmetros de entrada:

- Ind_tip_mod: Valor do tipo inteiro, correspodente ao indíce na fila de blocos de controle de tipo;

- Cs_tam: Valor do tipo unsigned int, indicando o tamanho do bloco de memória necessário ao carregamento do módulo. Este valor deve ser múltiplo de 16;

- Ds_tam: Valor do tipo unsigned int, que indica o tamanho do bloco de memória necessáio para instanciar o tipo módulo, este valor corresponde a soma das áreas de dados estáticos e dinâmicos e deve ser múltiplo de 16;

- Ss_tam: Valor do tipo unsigned int, correspondente ao tamanho do bloco de memória necessário aos "stacks" das tarefas pertencentes a intância do tipo módulo e deve ser múltiplo de 16;

- Param_tam: Valor do tipo unsigned int, correspondente ao tamanho de memória ocupado pelo parâmetros da instância;

- Iniciação: Valor do tipo ponteiro para caracter, correspondente ao offset da função que cria os descritores das tarefas e dos portos pertencentes a instância do módulo. Este offset deve estar em relação ao início do bloco de memória em que o módulo foi (ou será) carregado;

Parâmetros de retorno:

- Status: Valor do tipo inteiro, indicando:
  0 - toda operação foi um sucesso;
  1 - não existe memória disponível no núcleo para criar o bloco de controle de tipo;
  2 - existe bloco de controle de tipo com mesmo indíce.

1.2. Registra o endereço do segmento de código no bloco de controle de tipo.

A função "REGISTRA_END_CS_TIP", registra o endereço da base do segmento de código no bloco de controle de tipo módulo. Esta função só deve ser utilizada para registrar o endereço do segmento de código de módulos carregados na inicialização da estação.

FORMATO DE CHAMADA:

void **REGISTRA_END_CS_TIP** (ind_tip_mód, cs_base);

Parâmetros de entrada:

- Ind_tip_mod: Valor do tipo inteiro, correspondente ao bloco de controle de tipo;

- Cs_base: Valor do tipo ponteiro para caracter, correspondente a base do segmento de código.

Parâmetros de retorno:

- Não há.

1.3. Alocação de memória para o carregamento de código do módulo

A função "ALOCA_MEM_TIP", aloca o bloco de memória necessário ao carregamento do código de um tipo módulo. O tamano do bloco de memória é obtido no bloco de controle de tipo.

FORMATO DE CHAMADA:

void **ALOCA_MEM_TIP** (ind_tip_mod, cs_mod);

Parâmetros de entrada:

- Ind_tip_mod: Valor do tipo inteiro, correspondente ao bloco de controle de tipo;

Parâmetro de retorno:

- Cs_mod: Valor do tipo ponteiro para caracter, correspondente a base do segmento do bloco de controle alocado para carrregamento do código.

1.4. Habilitação de um tipo módulo

A função "HABILITA_TIPO", indica no bloco de controle de tipo que o módulo pode ser instanciado. Esta função deve ser utilizada para sinalizar que o código correspondente ao módulo já está carregado.

FORMATO DE CHAMADA:

void **HABILITA_TIPO** (ind_tip_mod);

Parâmetro de entrada:

- Ind_tip_mod: Valor do tipo inteiro, correspondente ao indíce do bloco de controle de tipo de módulo.

Parâmetro de retorno:

- Não Há

1.5. Destruição de um bloco de controle de tipo módulo

A função "DESTROI_BC_TIP_MOD (ind_tip_mod), destroi o bloco de controle de tipo módulo.

FORMATO DE CHAMADA:

void **DESTROI_BC_TIP_MOD** (ind_tip_mod);

Parâmetro de entrada:

- Ind_tip_mod: Valor do tipo inteiro, correspondente ao indíce do bloco de controle de tipo de módulo.

Parâmetro de retorno:

- Não Há.

1.6. Desalocação de um bloco de memória reservado ao código de módulo (tipo módulo)

A função "DESALOCA_MEM_TIP", dealoca o bloco de memória, correspondente ao código do módulo; caso não exista instância do tipo módulo.

FORMATO DE CHAMADA:

int **DESALOCA_MEM_TIP** (ind_tip_mod);

Parâmetro de entrada:

- Ind_tip_mod: Valor do tipo inteiro, correspondente ao indíce do bloco de controle de tipo de módulo.

Parâmetro de retorno:

- status: Valor do tipo inteiro, indicando:
  0 - a memória foi dealocada;
  1 - existe instância do módulo.

2. Funções relacionadas a instânciação de módulos.

2.1.Criação de uma instância de módulo

A função "CRIA_INS_MOD", gera dinamicamente um bloco de controle de uma instância (BCI) e aloca os blocos de memória para dados e "stack" da instância. Os valores do tamanho da área e do "stack" são obtidos no BCTM. Esta função, também cria os descritores de tarefas e de portos pertencentes a instância e inicializa os parâmetros.

FORMATO DE CHAMADA:

int **CRIA_INS_MOD** (ind_ins_mod, ind_tip_mod, base_param, ofs_parm);

Parâmetro de entrada:

- Ind_ins_mod: Valor do tipo inteiro, correspondente ao indíce na fila de BCI;

- Ind_tip_mod: Valor do tipo inteiro, correspondente ao indíce do tipo módulo que será instanciado;

- Base_param: Valor do tipo ponteiro para caracter, indicando a base do segmento em que se encontra os valores iniciais dos parâmetros correspondentes a instância;

- Ofs_param: Valor do tipo ponteiro para caracter, indicando offset em que se encontra os valores iniciais dos parâmetros correspondentes a instância.

Parâmetro de retorno:

- Status: Valor do tipo inteiro, indicando:
  0 - O módulo foi instanciado;
  1 - Não existe memória disponível no núcleo para criar o BCI;
  2 - Não existe memória disnponível na estação para alocar a area referente ao segmento de dados ;
  3 - Já existe o BCI;
  4 - O tipo módulo ainda não foi carregado;
  5 - Não existe o BCTM.

2.2. Registra o endereço do segmento de dados no BCI.

A função "REGISTRA_END_DS", registra o endereço da base do segmento de dados, correspondente a inicialização da estação, no BCI. Esta função somente deve ser utilizada para instância carregadas com o núcleo.

FORMATO DE CHAMADA:

void **REGISTRA_END_DS** (ind_ins_mod, ds_base_seg);

Parâmetro de entrada:

- Ind_ins_mod: Valor do tipo inteiro, correspondente ao indíce do BCI;

- Ds_base_seg: Valor do tipo ponteiro para caracter, correspondente ao segmento de dados da instância.

Parâmetro de retorno:

- Não Há.

2.3. Iniciação de uma instância

A função "INICIA_MÓDULO" coloca todos os descritores de tarefas pertencentes a instância na fila de tarefas prontas. A partir da execução desta função todas as tarefas pertencentes a instância estarão concorrendo ao processador.

FORMATO DE CHAMADA:

void **INICIA_MÓDULO** (ind_ins_mod);

Parâmetro de entrada:

- Ind_ins_mod: Valor do tipo inteiro, correspondente ao indíce do BCI.

Parâmetro de retorno:

- Não Há.

2.4. Retira todas as tarefas pertencentes a instância, das filas de tempo de execução.

A função "PARA_MÓDULO" retira todos descritores de tarefas, pertencentes a instância das filas de tempo de execução. Depois da execução desta função a instância pode ser reiniciada (função inicia_módulo) ou destruída.

FORMATO DE CHAMADA:

void **PARA_MÓDULO** (ind_ins_mod);

Parâmetro de entrada:

- Ind_ins_mod: Valor do tipo inteiro, correspondente ao indíce do BCI.

Parâmetro de retorno:

- Não Há.

2.5) Suspensão de uma instância

A função "SUSPENDE_MÓDULO coloca todas as tarefas pertencentes a instância em estado suspenso. Esta função deve ser chamada quando da execução de erros em tempo de execução (overflow e divisão por zero). As tarefas em estado suspensas podem retornar ao estado de pronta somente depois da execução da função PARA_MÓDULO, seguida da função INICIA_MÓDULO.

FORMATO DE CHAMADA:

void **SUSPENDE_MÓDULO** ();

Parâmetro de entrada:

- Não Há.

Parâmetro de retorno:

- Não Há.

2.6. Destruição de instância de módulo

A função "DESTROI_INS_MOD", desaloca o bloco de memória correspondente ao segmento de dados da instância, destroi os descritores de tarefas e de portos pertencentes a instância, decrementa o número de instância relacionada no BCTM e destroi o BCI.

FORMATO DE CHAMADA:

void **DESTROI_INS_MOD** (ind_ins_mod);

Parâmetro de entrada:

- Ind_ins_mod: Valor do tipo inteiro, correspondente ao indíce do BCI.

Parâmetro de retorno:

- Não Há.

3. Funções relacionadas a tarefa

3.1. Criação de tarefa

A função "CRIA_TAREFA" gera dinamicamente um descritor de tarefa e aloca o "stack" da tarefa na área anteriormente alocada a instância.

FORMATO DE CHAMADO:

int **CRIA_TAREFA** (tar_início, prioridade, ss_tam);

Parâmetro de entrada:

- Tar_início: Valor do tipo ponteiro para caracter, correspondente ao endereço de início do código da tarefa. Este endereço deve estar em relação a base do segmento de código do módulo;

- Prioridade: Valor do tipo inteiro que indica a prioridade da tarefa. São definidos 16 níveis de prioridade, sendo que a maior prioridade é de nível "zero". Os níveis de prioridade possuem as seguintes:

CLASSE	FAIXA_DE_PRIORIDADE
sistema	0 - 3
alta	4 - 7
normal	8 - 11
baixa	12 - 15

O usuário deve utilizar apenas as prioridades das 3 últimas classes;

- Ss_tam: Valor do tipo unsigned int, indicando o tamanho da área de "stack" da tarefa. Este valor deve ser múltiplo de 16.

Parâmetro de retorno:

- Status: Valor do tipo inteiro, indicando:
  0 - Sucesso da criação;
  1 - O descritor de tarefa não foi criado.

3.2. Mudança de prioridade de uma tarefa

A função "MUDA_PRIORIDADE" permite a uma tarefa alterar a própria prioridade em tempo de execução.

FORMATO DE CHAMADO:

void **MUDA_PRIORIDADE** (prioridade);

Parâmetro de entrada:

- Prioridade: Valor do tipo inteiro, correspondente a "nova" prioridade.

Parâmetro de retorno:

- Não Há.

3.3. Coloca uma tarefa em estado de suspensa

A função "SUSPENDE_TAREFA" permite a uma tarefa suspender-se. Uma tarefa suspensa por esta função só retornará ao estado de pronta, se for reinicializada.

FORMATO DE CHAMADO:

void **SUSPENDE_TAREFA** ();

Parâmetro de entrada:

- Não Há.

Parâmetro de retorno:

- Não Há.

4. Funções relacionadas ao porto

4.1. Criação de descritor de porto

A função "CRIA_PORTO", gera dinamicamente um descritor de porto.

FORMATO DE CHAMADO:

int **CRIA_PORTO** (identificador, ind_porto, num_max_msg, prioridade);

Parâmetro de entrada:

- Identificador: Valor do tipo inteiro, indicando o tipo de porto. São definidos os seguintes tipos

  1 - Porto de entrada SEND_NB;

  2 - Porto de entrada SEND_B;

  3 - Porto de entrada SEND_WAIT;

  4 - Porto de saída SEND_NB;

  5 - Porto de saída SEND_B;

  6 - Porto de saída SEND_WAIT.

- Ind_porto: Valor do tipo inteiro, correspondente ao indíce do porto na fila de portos pertencentes a instância. Os indíces dos portos pertencentes a mesma instância devem ser diferentes;

- Num_max_msg: Valor do tipo inteiro, indicando o número máximo de mensagens suportado pela fila de espera de um porto de entrada SEND_NB. Nos outros tipos de portos este campo deve ser igual a zero;

- Prioridade: Valor do tipo inteiro, correspondente a prioridade das mensagens enviadas através do porto de saída São definidos 2 níveis de prioridade:

  0 - Prioridade ALTA;

  1 - Prioridade NORMAL.

  Este campo somente é utilizado pelo núcleo: em comunicações remotas e envios vários-para-um;

Parâmetro de retorno:

- Status: Valor do tipo inteiro, indicando:
  0 - Sucesso da criação;
  1 - O descritor de porto não foi criado.

5. Funções relacionadas a Ligação entre portos

5.1. Ligação entre porto de saída e porto de entrada

A função "LIGA_PORTO". gera dinamicamente uma estrutura de ligação e registra as informações relacionadas ao descritor de porto de entrada. O endereço desta estrutura de ligação e então armazenado no descritor de porto de saída.

FORMATO DE CHAMADO:

```
int  LIGA_PORTO (ind_estação_pe, ind_ins_mod_pe, ind_pe,
                 ind_estação_ps, ind_ins_mod_ps, ind_ps);
```

Parâmetro de entrada:

- Ind_estação_pe: Valor do tipo inteiro, indicando o indíce da estação do descritor de porto de entrada;
- Ind_ins_mod_pe: Valor do tipo inteiro, indicando o indíce da instância que contém o descritor de porto de entrada;
- Ind_pe: Valor do tipo inteiro, indicando o indíce do porto de entrada;
- Ind_estação_ps: Valor do tipo inteiro, indicando o indíce da estação do descritor de porto de saída;

- Ind_ins_mod_ps: Valor do tipo inteiro, indicando o indíce da instância que contém o descritor de porto de saída;

- Ind_estação_ps: Valor do tipo inteiro, indicando a estação do descritor de porto de saída;

Para ligação entre portos de uma mesma estação:

ind_estação_pe = ind_estação_ps

Se a condição acima é satisfeita e:

ind_ins_mod_pe = ind_ins_mod_ps

a ligação corresponde a portos internos a instância.

Parâmetro de retorno:

- Status: Valor do tipo inteiro, indicando:

0 - Sucesso da ligação;

1 - Fracasso da ligação.

5.2. Religação entre portos

A função "RELIGA_PORTO", altera os valores contidos na estrutura de ligação de descritor de porto de saída, com as informações do "novo" descritor de porto de entrada.

FORMATO DE CHAMADO:

```
int  RELIGA_PORTO (ind_estação_pe, ind_ins_mod_pe, ind_pe,
                   ind_estação_ps, ind_ins_mod_ps, ind_ps);
```

Parâmetro de entrada:

- Ind_estação_pe: Valor do tipo inteiro, indicando o indíce da estação do descritor de porto de entrada;

- Ind_ins_mod_pe: Valor do tipo inteiro, indicando o indíce da instância que contém o descritor de porto de entrada;

- Ind_pe: Valor do tipo inteiro, indicando o indíce do porto de entrada;

- Ind_estação_ps: Valor do tipo inteiro, indicando o indíce da estação do descritor de porto de saída;

- Ind_ins_mod_ps: Valor do tipo inteiro, indicando o indíce da instância que contém o descritor de porto de saída;

- Ind_estação_ps: Valor do tipo inteiro, indicando a estação do descritor de porto de saída;

  Para ligação entre portos de uma mesma estação:

  ind_estação_pe = ind_estação_ps

  Se a condição acima é satisfeita e:

  ind_ins_mod_pe = ind_ins_mod_ps

  a ligação corresponde a portos internos a instância.

Parâmetro de retorno:

- Status: Valor do tipo inteiro, indicando:
  0 - Sucesso da religação;
  1 - Fracasso da religação.

5.3. Desligamento de portos

A função "DESLIGA_PORTO", destroi a estrutura de ligação correspondente a uma ligação.

FORMATO DE CHAMADO:

int **DESLIGA_PORTO** (ind_estação_pe, ind_ins_mod_pe, ind_pe,
ind_estação_ps, ind_ins_mod_ps, ind_ps);

Parâmetro de entrada:

- Ind_estação_pe: Valor do tipo inteiro, indicando o indíce da estação do descritor de porto de entrada;

- Ind_ins_mod_pe: Valor do tipo inteiro, indicando o indíce da instância que contém o descritor de porto de entrada;

- Ind_pe: Valor do tipo inteiro, indicando o indíce do porto de entrada;

- Ind_estação_ps: Valor do tipo inteiro, indicando o indíce da estação do descritor de porto de saída;

- Ind_ins_mod_ps: Valor do tipo inteiro, indicando o indíce da instância que contém o descritor de porto de saída;

- Ind_estação_ps: Valor do tipo inteiro, indicando a estação do descritor de porto de saída;

6. Funções relacionadas com a temporização

6.1. Fornece o valor atual do relógio tempo real.

A função "LE_RELOGIO", retorna o valor atual do relógio tempo real.

FORMATO DE CHAMADA:

int **LE_RELOGIO** ();

Parâmetro de entrada:

- Não Há.

Parâmetro de retorno:

- Relógio: Valor do tipo unsigneint, correspondente ao valor do relógio tempo real.

6.2. Temporização

A função "TEMPO" cada vez que é invocada:

- incrementa o relógio tempo real;
- Verificana fila de espera se existe tarefa com timeout esgotado. Caso houver insere na fila de pronta.

Esta função é periodicamente chamada por uma tarefa.

FORMATO DE CHAMADO:

void **TEMPO** ();

Parâmetro de entrada:

- Não Há.

Parâmetro de retorno:

- Não Há.

6.3. Espera Temporizada

A função "ESPERA_TEMPORIZADA", coloca a tarefa ativa na fila de espera por um tempo definido. Ao término deste tempo a tarefa retorna ao estado de pronta.

FORMATO DE CHAMADO:

void **ESPERA_TEMPORIZADA** (timeout);

Parâmetro de entrada:

- Timeout: Valor do tipo inteiro, indicando o tempo que a tarefa permanecerá em espera.

Parâmetro de retorno:

- Não Há.

7. Funções relacionadas com interrupção

7.1. Iniciação de um nível de interrupção

A função "INICIA_INTERRUPÇÃO" associa um vetor de interrupção (físico), com um indíce retornado (lógico). Este indíce corresponde a posição na tabela de interrupção, que faz parte da estrutura de dados do núcleo. Esta função deve ser invocada na inicialização da tarefa, não devendo estar dentro do loop.

FORMATO DE CHAMADO:

int **INICIA_INTERRUPÇÃO** (vetor_interrupção);

Parâmetro de entrada:

- Vetor_interrupção: Valor do tipo inteiro, correspondente ao vetor de interrupção. Os vetores entre os níveis 60h e 67h estão disponíveis a tarefas de sistema e de usuário.

Parâmetro de retorno:

- Ind_int: Valor do tipo inteiro, correspondente a uma posição na tabela de interrupção. Este valor deve ser utilizado pela função ESPERA_INTERRUPÇÃO.

7.2. Espera de interrupção

A função "ESPERA_INTERRUÇÃO" possibilita a uma tarefa esperar por um determinado nível de interrupção. Se no momento da execução da função a interrupção já ocorreu, a tarefa contínua em execção.

FORMATO DE CHAMADO:

void **ESPERA_INTERRUPÇÃO** (ind_int);

Parâmetro de entrada:

- Ind_int: Valor do tipo inteiro, correspondente a uma posição da tabela de interrupção (interna ao núcleo). Este valor é retornado pela função INICIA_INTERRUPÇÃO.

Parâmetro de retorno:

- Não Há.

8. Funções Relacionadas com a comunicação

8.1. Envio de uma mensagem

A função "ENVIO" permite a uma tarefa enviar uma mensagem através de um descritor de porto de saída. A transferência da mensagem enviada através desta função, se dá durante a execução da função (quando a função de RECEPÇÃO já ter sido executada pela tarefa receptora) ou durante a função RECEPÇÃO (neste caso a função RECEPÇÃO é executada após a função de envio).

A função de "ENVIO" possui 3 semânticas distintas:

- Ela pode ser "não bloqueante" (SEND_NB), neste caso se a tarefa receptora não estiver a espera da mensgem é alocado dinamicamente um "buffer" e um descritor de mensagem;
- Ela pode ser "bloqueante, aguardando recepção" (SEND_B), neste caso a tarefa fica no estado de espera até que a tarefa receptora receba a mensagem. Caso a tarefa receptora esteja a espera da mensagem a tarefa emissora continua normalmente.
- Ela também pode ser "bloqueante, aguardando resposta (SEND_WAIT), neste caso a tarefa emissora fica em espera até que a tarefa reptora envie uma mensagem de resposta.

Para os envios bloqueantes é definido um tempo máximo que a tarefa pode ficar em espera. Ao término deste tempo a tarefa retorna ao estado de pronta. Desta forma uma tarefa retorna ao estado de pronta, após a execução da função, ou pela complementação do envio ou pelo esgotamento do "timeout". O núcleo provê a função "FALHA" que executada após a função

"ENVIO" retorna a razão pela qual a tarefa retornou ao estado de pronta.

FORMATO DE CHAMADO:

void **ENVIO** (ind_porto, timeout, tam_msg, base_msg, ofs_msg, base_rsp, ofs_rsp);

Parâmetro de entrada:

- Ind_porto: Valor do tipo inteiro, indicando o indíce do porto de saída;

- Timeout: Valor do tipo inteiro, indicando o tempo máximo que a tarefa pode ficar em estado de espera, em caso dos envios bloqueantes. Para o caso da tarefa desejar esperar indefinidamente este parâmetro deve ser setado para (-1);

- Tam_msg: Valor do tipo inteiro, indicando o tamanho da mensagem de envio;

- Base_msg: Valor do tipo ponteiro para caracter, indicando a base do segmento em que se encontra a mensagem de envio;

- Ofs_msg: Valor do tipo ponteiro para caracter, indicando o offset da mensagem de envio;

- Base_rsp: Valor do tipo ponteiro para caracter, indicando a base do segmento em que se encontra a mensagem de reposta;

- Ofs_msg: Valor do tipo ponteiro para caracter, indicando o offset da mensagem de resposta;

Parâmetro de retorno:

- Não Há

8.2. Condição após um envio

A função "FALHA" permite a uma tarefa que executou um envio, obter a razão pela qual retornou ao estado de pronta.

FORMATO DE CHAMADO:

int **FALHA** ();

Parâmetro de entrada:

- Não Há.

Parâmetro de retorno:

- Status: valor do tipo inteiro, indicando:
  0 - Timeout esgotado;
  1 - Envio normal;
  2 - Porto de saída não ligado.

8.3. Inicialização de uma recepção

A função "INICIA_RECEPÇÃO" é executada para habilitar a tarefa à receber mensagem em um porto de entrada. Esta função deve ser executada antes da função de recepção.

FORMATO DE CHAMADA:

void **INICIA_RECEPÇÃO** (ind_porto, base_msg, ofs_msg,
ind_for; cláusula);

Parâmetro de entrada:

- Ind_porto: Valor do tipo inteiro, indicando o indíce do porto de entrada;

- Base_msg: Valor do tipo ponteiro para caracter, indicando a base do segmento em que se encontra a mensagem a ser recebida;

- Ofs_msg: Valor do tipo ponteiro para caracter, indicando o offset da mensagem a ser recebida;

- Ind_for: Valor do típo inteiro, indicando o indíce da variável de um comando FOR (Pascal, C, etc...). Este parâmetro só necessita ser preenchido quando a função inicia_recepção estiver interno ao comando FOR.

- Cláusula: Valor do tipo inteiro, indicando a cláusula de um "RECEIVE" em um contruto "SELECT". Para recepções única (caso geral), este parâmetro deve ser setado para 1.

Parâmetro de retorno:

- Não Há.

8.4. Recepção de uma mensagem

A função de "RECEPÇÃO" possibilita a uma tarefa receber mensagem de um descritor de porto de entrada. Antes da execução desta função deve ser executada a função "INICIA_RECEPÇÃO" para todos os descritores de portos de entrada que estão habilitado à receber mensagens.

A função pode ser bloqueante ou não. No caso de uma recepção bloqueante é definido um tempo máximo (timeout) que a tarefa pode ficar esperando pela mensagem. No final deste tempo a tarefa retorna ao estado de pronta. Portanto a tarefa pode retornar ao estado de pronta: quando receber a mensagem ou quando do esgotamento do timeout. Para obter a condição pela qual a tarefa retornou ao estado de pronta dever ser executada a função CONDIÇÃO_RECEPÇÃO.

FORMATO DE CHAMADA:

void **RECEPÇÃO** (timeout);

Parâmetro de entrada:

- Timeout: Valor do tipo inteiro, indicando o tempo máximo que a tarefa pode ficar esperando pela mensagem.

Parâmetro de retorno:

- Não Há.

8.5. Condição após uma recepção

A função CONDIÇÃO_RECEPÇÃO" executada após a função "RECEPÇÃO", indica a razão pela qual a tarefa retornou ao estado de pronta.

FORMATO DE CHAMADA:

void **CONDIÇÃO_RECEPÇÃO** (ind_for_ptr, cláusula);

Parâmetro de entrada:

- Não Há.

Parâmetro de retorno:

- Ind_for_ptr: Valor do tipo ponteiro para inteiro "FAR", indicando o endereço da variável na qual é retornado o ind_for do porto de entrada em que a tarefa recebeu a mensagem;

- Cláusula_ptr: Valor do tipo ponteiro para inteiro "FAR", indicando o endereço da variável na qual é retornado a cláusula do porto de entrada em que a tarefa recebeu mensagem. Se cláusula for igual a zero (0), indica que o timeout esgotou-se.

8.6. Envio de uma mensagem de resposta

A função "RESPONDE" permite a uma tarefa enviar um mensagem de resposta, após a recepção de mensagem. A função responde é não bloqueante.

FORMATO DE CHAMADA:

```
void  RESPONDE (ind_porto, tam_rsp, base_rsp, ofs_rsp);
```

Parâmetro de entrada:

- Ind_porto: Valor do tipo inteiro, indicando o indíce do descritor de porto de entrada;

- Tam_rsp: Valor do tipo inteiro, indicando o tamanho da mensagem de resposta;

- Base_rsp: Valor do tipo ponteiro para caracter, indicando a base do segmento da mensagem de resposta;

- Ofs_rsp: Valor do tipo ponteiro para caracter, indicando o offset da mensagem de resposta.

Parâmetro de retorno:

- Não Há.

9. Funções relacionadas com a comunicação remota

9.1. Formatação de um descritor de mensagem remota para envio da mensagem

A função "FORMATA_MSG_ENVIO_REMOTA", preenche os campos do descritor de mensagem remota, com as informações correspondente a mensagem, destino da mensagem e a primitiva utilizada no envio da mensagem.

FORMATO DE CHAMADA:

int **FORMATA_MSG_ENVIO_REMOTA** (base_dsc, ofs_dsc);

Parâmetro de entrada:

- Base_dsc: Valor do tipo ponteiro para caracter, indicando a base do segmento em que encontra-se o descritor de mensagem remoto;

- Ofs_dsc: Valor do tipo ponteiro para caracter, indicando a offset do descritor de mensagem remoto;

Parâmetro de retorno:

- Status: Valor do tipo inteiro, indicando:
  * igual 0 - final de envio;

* diferente 0 - o envio deve ser executado outra vez, isto indica que a mensagem é multidestinação, implicando em uma nova formatação da mensagem.

9.2. Formatação de um descritor de mensagem remota para envio da mensagem de resposta

A função "FORMATA_MSG_RESPOSTA_REMOTA", preenche os campos do descritor de mensagem remota, com as informações correspondente a mensagem de resposta.

FORMATO DE CHAMADA:

int **FORMATA_MSG_ENVIO_REMOTA** (base_dsc, ofs_dsc);

Parâmetro de entrada:

- Base_dsc: Valor do tipo ponteiro para caracter, indicando a base do segmento em que encontra-se o descritor de mensagem remoto;

- Ofs_dsc: Valor do tipo ponteiro para caracter, indicando a offset do descritor de mensagem remoto;

Parâmetro de retorno:

- Não Há.

9.3. Preparação para o servidor de rede de recepção executar a função responde.

A função "PREPARA_PARA_RESPONDER", coloca o descritor de porto de saída na fila de espera do descritor de porto de

entrada do servidor de rede de receçpão para que possa ser executada a função responde.

FORMATO DE CHAMADA:

void **PREPARA_PARA_RESPONDER** (ind_pe, ind_ins_mod, ind_ps);

Parâmetro de entrada:

- Ind_pe: Valor do tipo inteiro, correspondente ao indíce do porto de entrada do servidor de rede de recepção;
- Ind_ins_mod: Valor do tipo inteiro, correspondente ao indíce da instância, a qual pertence o descritor de porto de saída que está aguardando pela resposta;
- Ind_ps: Valor do tipo inteiro, correspondente ao descritor de porto de saída.

Parâmetro de retorno:

- Não Há.