UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA DE CAMPINAS

DEPARTAMENTO DE ENGENHARIA ELÉTRICA

Campinas, 20 de outubro de 1986

Este exemplar corresponde à redação final da tese defendida por: JUAN MANUEL ADÁN COELLO e aprovada pela Comissão Julgadora em 15/08/86.

Mauricio Ferreira Magalhães

SUPORTE DE TEMPO REAL PARA UM AMBIENTE DE PROGRAMAÇÃO CONCORRENTE

Por: JUAN MANUEL ADÁN COELLO

Orientador: Prof. Dr. MAURÍCIO FERREIRA MAGALHÃES

Dissertação apresentada à Faculdade de Engenharia de Campinas da Universidade Estadual de Campinas, em cumprimento às exigências para obtenção do Grau de Mestre.

AGOSTO-1986

A G R A D E C I M E N T O S

Ao Prof. Dr. Maurício Ferreira Magalhães, pelo incentivo, orientação, apoio e amizade que me dispensou durante o desenvolvimento deste trabalho.

Ao Prof. Dr. Luis Gimeno Latre, Diretor do Instituto de Automação do Centro Tecnológico para Informática e ao Prof. Dr. Jaime Szajner, Chefe do Departamento de Controle de Processos do Instituto de Automação, que com seu apoio possibilitaram a realização deste trabalho.

Ao colega Adilson Barbosa Lopes, responsável pela implementação das linguagens LPM e LCM, pela colaboração prestada ao longo de todo o trabalho.

À Célia Maria Dorázio pela eficiência, e paciência, demonstradas no trabalho de edição deste texto.

Ao Roberto de Oliveira ("Beto"), pelos desenhos dos diagramas de configuração dos exemplos.

SUMÁRIO

S U M Á R I O

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

R E S U M O

Este trabalho apresenta a implementação do suporte de tempo real de um ambiente orientado ao desenvolvimento de software para aplicações de controle de processos. A construção de um programa de aplicação neste ambiente divide-se em duas etapas: a programação dos módulos que implementam as funções do sistema e a configuração da aplicação a partir dos módulos disponíveis. Um módulo consiste de um programa sequencial que implementa uma ou mais funções e que pode interagir com outros módulos enviando e recebendo mensagens assíncronas e síncronas através de portas de comunicação. Os módulos são programados usando a Linguagem de Programação de Módulos (LPM) e a especificação de uma configuração corresponde a um programa em Linguagem de Configuração de Módulos (LCM). A LPM é uma extensão da Linguagem PASCAL que, em adição às construções desta, permite: declarar portas de entrada e saída; declarar mensagens; enviar e receber mensagens através de portas; suspender um módulo por um período de tempo e tratar logicamente interrupções. Nos programas escritos em LCM são especificados os módulos que comporão uma aplicação e a maneira como serão ligadas as suas portas de comunicação.

Na implementação da LPM, os comandos que não são parte do PASCAL padrão são traduzidos em chamadas a serviços oferecidos por um Suporte de Tempo Real (STR). A implementação do STR tem dois aspectos principais: o primeiro, independente do hardware, compreende as estruturas de dados e rotinas diretamente relacionadas a implementação dos serviços oferecidos para suportar as extensões da LPM ao PASCAL; o segundo, dependente do hardware, consiste basicamente dos mecanismos para: carga dos módulos da aplicação, interface entre os módulos e o STR e atendimento de interrupções.

Para implantar a primeira versão deste ambiente foi escolhido um microcomputador compatível com o IBM-PC. Esta escolha deve-se à grande popularidade desses equipamentos nas mais diversas áreas de aplicação e, em particular, à sua crescente aceitação nos ambientes industriais.

Na implantação do STR é empregada uma arquitetura que privilegia a independência dos módulos da aplicação entre si e destes e o STR. Esta independência implica basicamente na estrita separação entre os espaços de endereçamento dos componentes do sistema (módulos e STR).

ABSTRACT

ABSTRACT

This work presents the real time support implementation of an environment oriented to software development for process control applications. An application program construction in this environment is divided into two phases: the programming of modules that implement system functions, and, application configuration using available modules. A module is a sequential program that implements one or more functions and whose interaction with other modules is made sending and receiving, assynchronous and synchronous, messages by means of communication ports. Modules are programmed using the Modules Programming Language ("Linguagem de Programação de Módulos - LPM"). A configuration specification corresponds to a program in Modules Configuration Language ("Linguagem de Configuração de Módulos _ LCM"). The LPM is an extension of the PASCAL programming language that, in addition to PASCAL's constructions, allows: to declare entry and exit ports, to declare messages, to send and receive messages to/from ports, to delay a module by a period of time and to logically handle interrupts. LCM programs specify the modules that will compose an application and the way in that their ports will be connected.

In LPM implementation, comands that do not belong to standard PASCAL are translated into requests to services offered by a Real Time Support ("Suporte de Tempo Real - STR"). STR's implementation has two main aspects, the first one, hardware independent, comprehends the data structures and routines directly related to the implementation of the services offered to suport LPMs extensions to PASCAL; the second one, hardware dependent, consists basically of the mechanisms to: load application modules, interface the STR and the modules, and interrupt handling.

The first version of this environment was implanted in an IBM-PC compatible microcomputer. The choice for this computer was based on the big popularity of this kind of equipment in a wide range of applications, and particularly, on its increasing acceptance in industrial environments.

STR's implantation architecture guarantees application modules independence from each other and from the STR. This independence implys basically in a strict separation between system components addressing spaces (modules and STR).

CAPÍTULO I

INTRODUÇÃO

1. INTRODUÇÃO

A evolução da tecnologia de integração de componentes verificada nos últimos anos reduziu significativamente o custo do hardware, provocando um aumento contínuo dos campos de aplicação dos computadores. Neste período, a área de controle de processos em tempo real foi possivelmente aquela que apresentou o maior aumento no uso intensivo de computadores [YOU 82].

Paralelamente ao surgimento de microcomputadores de baixo custo, estão sendo introduzidas as redes locais de comunicação que possibilitam a construção de sistemas distribuídos com desempenho comparável ao dos sistemas centralizados tradicionais de maior porte, por um custo bem menor. Estes sistemas, em função da sua alta modularidade, praticamente eliminam as severas limitações de expansão encontradas nos sistemas centralizados, facilitando ainda a implementação de mecanismos de tolerância a falhas. Estas características são fundamentais em sistemas de controle de processos que devem apresentar grande confiabilidade e longa vida útil.

Embora tenhamos presenciado uma grande evolução tecnológica que barateou o custo do hardware, o mesmo não ocorreu com o software. Isto é particularmente verificável na área de controle de processos. O barateamento dos computadores aliado à necessidade de otimizar os processos produtivos motivou o desenvolvimento de sistemas aplicativos de controle cada vez mais sofisticados e confiáveis. A complexidade para o desenvolvimento desses sistemas usando um ferramental em geral obsoleto, especialmente para as novas arquiteturas distribuídas, aumentou sensivelmente. Conseqüentemente o custo de projeto, implementação e manutenção dessa geração de software cresceu significativamente.

O escasso ferramental disponível para a produção de software para controle em tempo real está orientado a sistemas centralizados de computação e conduz ao desenvolvimento de programas de estrutura muito rígida. Este tipo de programa não permite acompanhar a evolução natural da estrutura de um sistema de controle que geralmente é dinâmica. Isto porque, seja pela introdução de novas tecnologias, seja pela expansão do processo, o sistema está sempre em evolução motivando, em conseqüência, que o software aplicativo seja altamente reconfigurável. Na disponibilidade, por exemplo, de um módulo de software que implementa um novo algoritmo de controle este deveria poder substituir o módulo que implementa a versão antiga do algoritmo de forma que perturbasse o mínimo possível evitando, inclusive, a necessidade de parar o sistema de controle para recompilação do aplicativo.

Este trabalho apresenta a implementação do suporte de tempo real de um ambiente para programação concorrente. Este ambiente oferece um conjunto de ferramentas que simplificam o projeto, im-

plementação e teste de software para aplicações de controle em tempo real. As ferramentas oferecidas conduzem ao desenvolvimento de software de aplicação altamente modular, reconfigurável e independente da arquitetura do hardware. Consequentemente, uma aplicação pode ser desenvolvida e testada num ambiente centralizado podendo posteriormente ser executada em sistemas centralizados ou distribuídos. Esta primeira versão do suporte de tempo real do ambiente permite configuração estática das aplicações e a sua execução em arquiteturas centralizadas. Entretanto, a sua concepção facilita expandi-lo para sistemas distribuídos e possibilita a introdução de mecanismos para reconfiguração dinâmica.

O trabalho está organizado da seguinte maneira: o Capítulo 2 discute, brevemente, alguns conceitos importantes na computação em tempo real. No Capítulo 3 são apresentadas as ferramentas que este ambiente para programação concorrente oferece, consistindo basicamente de uma Linguagem para Programação de Módulos de software e de uma Linguagem para Configuração desses Módulos. No Capítulo 4 é descrita a implementação do suporte de tempo real nos seus aspectos independentes do hardware empregado. No Capítulo 5 é apresentada a implantação do suporte de tempo real num equipamento específico. No Capítulo 6 o uso das ferramentas oferecidas pelo ambiente é ilustrado através de dois exemplos. Finalmente, o Capítulo 7 faz algumas considerações relativas à versão do ambiente atualmente implantada e à continuidade do trabalho.

CAPÍTULO 2

COMPUTAÇÃO EM TEMPO REAL

2. COMPUTAÇÃO EM TEMPO REAL

A expressão "Computação em Tempo Real" comporta significados diferentes em contextos diferentes. A fim de caracterizar as aplicações que se tem em vista, convém esclarecer o seu significado neste texto. A computação em tempo real implica no acoplamento (ou interconexão) de um computador a um processo. O objetivo dessa interconexão é obter informações do processo (monitorar a sua operação) através da medição das variáveis importantes (pressão, temperatura, etc.) para possivelmente manipulá-las de alguma forma desejada (controlar a sua operação com base nas informações previamente adquiridas).

Um processo pode ser tão simples como um único instrumento de laboratório, ou tão complexo como toda uma refinaria de petróleo. Porém, independentemente da grande variedade de características presentes nesses processos, qualquer sistema tempo-real pode ser representado, muito simplificadamente, como na figura abaixo,

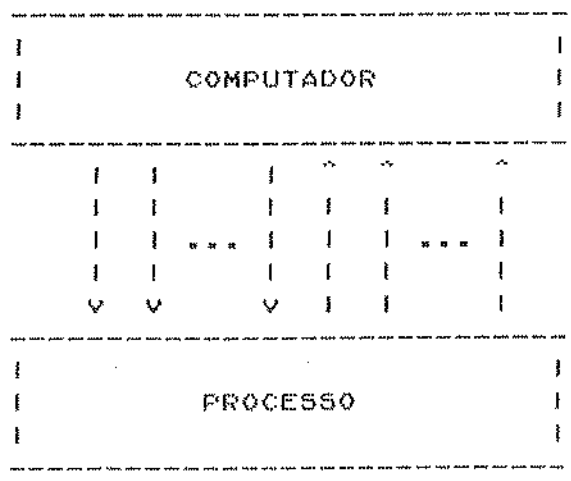

FIGURA 2.1 – Representação de um sistema de controle em tempo real

onde a interconexão (ou acoplamento) entre o computador e o processo é graficamene indicada por setas, que representam o fluxo de informações.

Qualquer que seja sua complexidade, todo processo opera com escalas de tempo próprias, que podem ser medidas em milesegundos, dias ou anos. É evidente que no acoplamento do computador ao pro-

cesso, as operações envolvendo o primeiro devem ser adequadas ao segundo e não o contrário,. É necessário, portanto, projetar e programar o computador para que ele possa "perceber" a passagem do "tempo real", independentemente de suas próprias operações internas.

O conceito de "tempo real" também implica na habilidade do computador responder suficientemente rápido aos estímulos do processo a fim de atender às suas necessidades. Por exemplo, caso uma situação de emergência ocorrida no processo seja sinalizada ao computador, este deverá reagir a tempo de poder tratá-la, evitando situações potencialmente danosas.

2.1. Programação Concorrente

A essência dos sistemas de aplicação em tempo real é a sua habilidade para tratar vários eventos que ocorrem, geralmente, a intervalos imprecisos. Esses eventos são ditos assíncronos porque podem ocorrer a qualquer instante, sendo potencialmente concorrentes, pois um evento pode ocorrer enquanto um outro ainda estiver sendo processado, implicando, portanto, numa disputa pelos recursos disponíveis no sistema (UCP, periféricos, etc.) para tratamento desses eventos.

Qualquer programa que se proponha a processar vários eventos assíncronos e concorrentes, tende a ser extremamente complexo. O programa deve processar os eventos, lembrar que eventos ocorreram, em que ordem e quais ainda não foram processados. À medida que o número de eventos aumenta, a complexidade do programa cresce substancialmente. Essa complexidade pode ser significativamente reduzida se, em vez de um único, forem projetados n programas, um para cada evento distinto. A seguir apresenta-se, sucintamente, uma série de conceitos e notações concebidas para permitir e facilitar a construção desses programas.

2.1.1. Processos

Um programa sequencial especifica a execução sequencial de uma lista de instruções; a sua execução é chamada um processo. Um programa concorrente especifica dois ou mais programas sequenciais que podem ser executados concorrentemente como "processos paralelos".

A execução de um programa concorrente num único processador caracteriza um ambiente denominado na literatura de multiprogramação. Caso o programa seja executado em dois ou mais processadores através do compartilhamento de uma memoria comum, diz-se que é feito multiprocessamento. Os sistemas com vários processadores nos quais estes têm memória própria e se comunicam entre si através de uma "rede de comunicações", são geralmente chamados "sistemas distribuídos".

A fim de que a execução de um processo possa influenciar na execução de um outro, deve ser possível que ambos se comuniquem. Por outro lado, como a execução dos processos de um programa concorrente ocorre geralmente a velocidades não previsíveis, é necessário sincronizá-los a fim de que a comunicação se dê de forma ordenada. A comunicação entre processos ocorre através do uso de variáveis compartilhadas ou através da troca de mensagens.

2.1.2. Comunicação e Sincronização entre Processos através de Variáveis Compartilhadas

Quando variáveis compartilhadas são usadas para a comunicação entre processos, dois tipos de sincronização são úteis: a exclusão mútua e a sincronização condicional. A exclusão mútua assegura que uma sequência de instruções (chamada de região crítica) seja tratada como uma operação indivisível.

A sincronização condicional ocorre quando um objeto de dados compartilhado está num estado impróprio para se efetuar determinada operação. Qualquer processo desejando efetuar essa operação deve ser suspenso até que o estado do objeto mude em consequência de operações sendo realizadas por outros processos.

Os dois principais mecanismos empregados na implementação desses dois tipos de sincronização são os semáforos e os monitores.

2.1.2.1. Semáforos

Um semáforo é uma variável inteira não negativa sobre a qual definem-se as operação P e V. Dado um semáforo s, P(s) suspende a execução do processo que efetuou a operação até que s seja maior que 0 e a seguir faz s:=s-1; o teste e o decremento de s são executados como uma única operação indivisível. V(s) faz s:=s+1 como uma operação indivisível, permitindo que processos suspensos em razão de uma operação P(s) possam prosseguir na sua execução.

Semáforos são mecanismos bastante gerais para resolver problemas de sincronismo. Para fazer uma exclusão mútua, cada região crítica é precedida de uma operação P e seguida de uma operação V no mesmo semáforo. Todas as regiões críticas mutuamente exclusivas devem usar o mesmo semáforo que deve ser iniciado com o valor 1. Esse tipo de semáforo que assume apenas os valores 0 e 1 é chamado de semáforo binário.

Um semáforo que pode assumir um valor qualquer é chamado um semáforo contador. Os semáforos contadores são geralmente usados para sincronização condicional quando se está controlando a alocação de recursos. Nesses casos o valor inicial do semáforo corresponde ao número inicial de unidades do recurso; a operação P é usada para retardar (ou suspender) um processo até que uma unidade do recurso seja liberada; a operação V é executada quando uma unidade do recurso é liberada.

2.1.2.2. Monitores

Embora as primitivas P e V possam ser usadas para implementar praticamente qualquer esquema de sincronização, elas não induzem a uma programação bem estruturada, sendo portanto fácil cometer erros no seu emprego. A principal alternativa ao uso de semáforos são os monitores. Um monitor é formado pelo encapsulamento da definição de um recurso e das operações que o manipulam. Consequentemente, um programador pode ignorar os detalhes da implementação do recurso quando o usa, e pode ignorar como ele será usado quando estiver programado o monitor que o implementa.

Um monitor consiste de uma coleção de variáveis permanentes usadas para armazenar o estado do recurso, do código de inicialização dessas variáveis e dos procedimentos que implementam as operações sobre o recurso. Os procedimentos do monitor podem ter parâmetros e variáveis locais. A estrutura de um monitor é a seguinte:

```
<id_monitor>: MONITOR

VAR <declaração das variáveis permanentes>

PROCEDURE <operação 1> (<parâmetros>);

VAR <declaração das variáveis locais do procedimento>;

BEGIN
     <código que implementa a operação 1>

END;
```

```
.
.
.

PROCEDURE <operação N> (<parâmetros>);

VAR <declaração das variáveis locais do procedimento>;

BEGIN
     <código que implementa a operação N>
END

BEGIN
     <código de inicialização das variáveis permanen-
      tes>
END
```

A chamada do procedimento do monitor que implementa a operação N é feita executando

```
CALL <id_monitor> . <operação N> (<argumentos>)
```

A execução dos procedimentos de um monitor é mutuamente exclusiva. Assim, as variáveis permanentes nunca são acessadas concorrentemente.

A sincronização condicional em monitores pode ser feita com a introdução de variáveis condicionais. Uma variável condicional é usada para suspender um processo executando um monitor. Sobre as variáveis condicionais são definidas as operações **SIGNAL** e **WAIT**.

Se cond for uma variável condicional,

```
cond.WAIT
```

faz com que o processo que executou a operação fique bloqueado em cond liberando o seu controle mutuamente exclusivo sobre o monitor. Quando da execução de

```
cond.SIGNAL
```

se nenhum processo estiver bloqueado em cond o processo que fez a operação continua normalmente a sua execução; em caso contrário, ele é suspenso e um processo bloqueado em cond é reativado. Um

processo suspenso devido à execução de uma opção **SIGNAL** prossegue a sua execução quando não houver outros processos no monitor. A introdução de variáveis condicionais permite que mais de um processo esteja num monitor, embora apenas um não fique bloqueado nas operações **WAIT** e **SIGNAL**.

2.1.3. Comunicação e Sincronização entre Processos através da Troca de Mensagens

Quando se usa a troca de mensagens para a comunicação e sincronização, os processos enviam e recebem mensagens em vez de ler e escrever em variáveis compartilhadas. Uma mensagem é enviada executando-se:

SEND ⟨expressão⟩ TO ⟨destino⟩

a mensagem conterá os valores da ⟨expressão⟩ no instante em que o **SEND** foi efetuado. O ⟨destino⟩ permite especificar quem receberá a mensagem. Uma mensagem é recebida executando-se:

RECEIVE ⟨lista de variáveis⟩ FROM ⟨origem⟩

a ⟨origem⟩ permite especificar de quem se deseja receber a mensagem. A recepção de uma mensagem faz com que os dados nela contidos sejam atribuídos à ⟨lista de variáveis⟩ após o que a mensagem será destruída.

A troca de mensagens apresenta dois aspectos principais: o endereçamento do ⟨destino⟩ e ⟨origem⟩ e a sincronização da comunicação.

2.1.3.1. Endereçamento numa Troca de Mensagens

O agrupamento de pares (⟨origem⟩,⟨destino⟩) define canais de comunicação. Vários esquemas foram propostos para nomear canais. O mais simples é conhecido por endereçamento direto e consiste em usar o nome dos processos envolvidos como ⟨origem⟩ e ⟨destino⟩. Quando o endereçamento direto é usado, a comunicação é chamada um para um, pois cada processo somente pode comunicar-se com o outro referenciado na primitiva de comunicação. O endereçamento direto é fácil de implementar e usar, sendo particularmente

apropriado à programação de "pipelines". Um "pipelines" é uma coleção de processos concorrentes nos quais a saída de um processo é usada como entrada de um outro.

Outro tipo importante de interação entre processos é a relação cliente/servidor. Nesta, algum processo servidor oferece um serviço a algum processo cliente. Um cliente pode solicitar que um serviço seja executado enviando uma mensagem a um servidor. Um servidor repetidamente recebe uma solicitação de serviço de um cliente, executa o serviço e (caso necessário) envia uma mensagem de término ao cliente. Um exemplo típico deste relacionamento é o existente entre um "driver" de entrada/saída e os processos que o usam.

Idealmente um RECEIVE num servidor deve permitir a recepção de mensagens de qualquer cliente. Se houver apenas um cliente o endereçamento direto funcionará muito bem; no entanto, caso haja mais clientes será necessário fazer um RECEIVE para cada cliente.

. Um esquema mais sofisticado para definir canais de comunicação é baseado no uso de nomes globais, comumente chamados de caixas-postais ("mail-boxes"). O uso de caixas-postais permite o estabelecimento de comunicações muitos para muitos, esquema especialmente apropriado para interações do tipo cliente/servidor. Os clientes enviam seus pedidos de serviço a uma caixa postal de onde os servidores retiram os pedidos que atenderão. Entretanto, a implementação de caixas postais em sistemas distribuídos pode ser muito cara. Quando uma mensagem é enviada a uma caixa postal, ela deve ser distribuída a todos os lugares onde houver processos recebendo dessa caixa. Após um dos processos efetuar um RECEIVE nessa caixa todos os lugares, para onde a mensagem foi enviada, devem ser notificados que ela já não está mais disponível.

Um esquema muito mais simples de implementação que permite comunicações muitos para um é a porta de comunicação. Uma porta de comunicação pode aparecer como identificador de destino em qualquer processo, porém somente poderá aparecer como identificador de origem num único processo, i.e., qualquer processo pode enviar uma mensagem a qualquer porta, porém, apenas um único processo pode receber mensagens de cada porta.

Os canais de comunicação podem ser estabelecidos em tempo de compilação ou em tempo de execução; no primeiro caso dizemos que o endereçamento é estático e no segundo dinâmico. O endereçamento estático apresenta dois problemas; o primeiro é que ele não permite que um programa se comunique através de canais não conhecidos em tempo de compilação, limitando as possibilidades de o programa existir num ambiente sujeito a mudanças de configuração. O segundo problema é que um processo deve manter o canal alocado permanentemente, ainda que ele somente o utilize muito esporadicamente.

2.1.3.2. Sincronização numa Troca de Mensagens

Outro aspecto importante numa troca de mensagens é a semântica dos comandos que a executam relativamente ao sincronismo que provocam nos processos envolvidos. Diz-se que é feito um envio não bloqueante ou envio assíncrono de uma mensagem quando a execução de um comando **SEND** não provoca a suspensão do processo emissor. Este tipo de envio pode ser implementado armazenando a mensagem entre o emissor e o receptor.

Se não houver um "buffer" onde o emissor possa depositar as mensagens que envia, ele será suspenso até que a mensagem chegue ao receptor. Este tipo de envio é chamado bloqueante ou síncrono. Caso o emissor além de esperar a chegada das mensagens que transmite ao receptor aguarde uma resposta, diz-se que é feita uma troca síncrona de mensagens.

O envio assíncrono de mensagens permite que o emissor fique arbitrariamente distante do receptor. Consequentemente, quando uma mensagem é recebida, ela contém informações sobre o estado do emissor, que não necessariamene correspondem ao seu estado atual. Quando uma troca síncrona de mensagens e empregada, ela estabelece um ponto de sincronização na execução do transmissor e do receptor. Logo a mensagem recebida corresponderá ao estado atual do emissor. Entre esses dois extremos está o envio "bufferizado" de mensagens, na qual a capacidade de armazenamento entre o emissor e o receptor é limitada. O envio "bufferizado" de mensagens permite que o emissor se distancie do receptor mas não de maneira arbitrária.

A recepção de mensagens pode ser explícita usando um comando **RECEIVE**, bloqueante ou não, ou implícita. A recepção implícita de uma mensagem provoca a execução de uma dada sequência de instruções de forma semelhante à execução do tratador associado a uma interrupção.

A combinação da troca síncrona de mensagens e da recepção implícita de mensagens é conhecida por chamada remota de procedimentos. A combinação da troca síncrona de mensagens e da recepção explícita de mensagens é conhecida como **rendez-vous**.

2.1.4. Políticas de Reescalonamento

Pode ocorrer durante a execução de um prosrama concorrente de num dado instante diversos processos estarem prontos (habilitados) para execução. Esta situação não pode ser resolvida apenas em termos de sincronização, sendo necessário adotar alguma "política de reescalonamento" para escolher o processo a executar. As

políticas mais comuns são o reescalonamento por prioridade (ou preemptivo) e o reescalonamento equânime (ou cíclico).

Na política de reescalonamento preemptivo sempre estará em execução o processo pronto de maior prioridade. Em conseqüência, sempre que algum evento provocar a passagem para o estado de pronto de um processo mais prioritário que aquele em execução haverá um reescalonamento. Analogamente, quando o processo em execução for suspenso em consequencia de uma operação interna (por exemplo devido à espera por uma mensagem) o processo pronto de maior prioridade passará para o estado de execução (receberá o controle sobre a UCP).

Na política de reescalonamento equânime ou cíclico a suspensão do processo em execução provocará a passagem do controle da UCP para o processo que está pronto há mais tempo.

Das políticas apresentadas, a preempção por prioridade é considerada a mais adequada à programação de sistemas de controle em tempo real. Isto se deve ao fato dela permitir o atendimento rápido de eventos que requerem tempo de resposta rígido, como por exemplo, a ocorrência de uma situação de emergência no processo sendo controlado.

2.2. Linguagens de Programação Concorrente

Do ponto de vista dos mecanismos empregados para a interação entre processos, as linguagens de programação concorrente atualmente disponíveis podem ser classificadas em três classes [AND 83]: orientadas a procedimentos, orientadas a mensagens e orientadas a operações.

Nas linguagens orientadas a procedimentos, a interação entre processos é baseada em variáveis compartilhadas. Os objetos nessas linguagens podem ser ativos (processos) ou passivos (monitores, semaforos, etc.). Os objetos passivos são representados por variáveis compartilhadas e por procedimentos que implementam as operações sobre os objetos. Os processos interagem acessando concorrentemente os objetos compartilhados. Os principais mecanismos usados para controlar o acesso compartilhado a variáveis nas linguagens orientadas a procedimentos são os semáforos e os monitores. Exemplos de linguagens nesta classe são o PASCAL concorrente [HAN 75] e MODULA [WIR 82].

As linguagens orientadas a mensagens oferecem os comandos SEND e RECEIVE como mecanismo basico de interação entre processos. Em consequencia, não há objetos passivos compartilhados, sendo cada objeto de propriedade de um unico processo que se encarrega de gerenciar o seu uso. Quando um processo desejar aces-

sar um objeto que não é de sua propriedade, ele envia uma mensagem de solicitação de acesso ao processo gerenciador do objeto. Este manipula o objeto conforme requisitado e, eventualmente, envia uma mensagem de resposta ao processo que solicitou o acesso ao objeto. Nesta classe de linguagens incluem-se OCCAM [WIL 83] e CONIC [KRA 84].

As linguagens orientadas a operações oferecem a chamada remota de procedimentos como mecanismo básico de interação entre processos. Como nas linguagens baseadas na troca de mensagens cada objeto pertence a um processo e como nas linguagens orientadas a procedimentos as operações sobre um objeto são feitas a partir da chamada de um procedimento. A linguagem mais importante nesta classe é ADA [ANS 83].

As linguagens dessas três classes são equivalentes em termos de poder de expressão computacional. Adicionalmente, todas elas podem ser usadas para escrever programas concorrentes em sistemas monoprocessados, multiprocessados e distribuídos. Entretanto, nem todas são igualmente apropriados para todas essas arquiteturas. As linguagens orientadas a procedimentos são as que permitem as implementações mais eficientes em sistemas com memória compartilhada (sistemas centralizados). Por outro lado, as linguagens orientadas a mensagens e a operações podem ser implementadas tanto em sistemas centralizados como distribuídos. Uma vantagem importante das linguagens baseadas em mensagens ou operações, sobre as linguagens baseadas em procedimentos, é a possibilidade de escrever programas independentes da arquitetura da máquina onde serão executados ser centralizada ou distribuída.

Um ponto particularmente importante na programação de sistemas distribuídos é a descrição de uma configuração. A descrição de uma configuração apresenta tres aspectos principais: a estrutura lógica, a estrutura física e o mapeamento da estrutura lógica na estrutura física. A estrutura lógica descreve os componentes de software do sistema e a sua interligação lógica. A estrutura física descreve os componentes de hardware do sistema e a sua interconexão física.

A inclusão da descrição da configuração lógica e dos componentes do sistema num mesmo programa foi o enfoque adotado por grande número de linguagens das quais ADA é um exemplo. Algumas linguagens de programação mais modernas oferecem uma linguagem para programar os componentes do sistema e uma outra para descrever a sua configuração. Este é o caso da linguagem CONIC.

Os itens a seguir descrevem sucintamente algumas linguagens de programação concorrente representativas das três classes de linguagens anteriormente mencionadas.

2.2.1. PASCAL Concorrente

A linguagem PASCAL concorrente estende o PASCAL sequencial incorporando processos concorrentes, monitores e classes, tendo sido a primeira linguagem de programação a suportar o conceito de monitor. Um dos maiores objetivos da linguagem foi permitir o desenvolvimento de programas que exibissem um comportamento reproduzível. O uso de monitores previne vários erros decorrentes da execução, em determinadas sequências, de rotinas concorrentes que compartilham dados, assegurando que as variáveis privadas de um processo sejam inacessíveis aos demais. Os processos podem comunicar-se exclusivamente através de monitores.

Uma classe define uma estrutura de dados e as operações sobre ela como num monitor. Entretanto, o acesso exclusivo de um processo às variáveis de uma classe é assegurado em tempo de compilação, tornando as chamadas a classes bem mais rápidas que as chamadas a monitores.

2.2.2. ADA

A linguagem ADA oferece como mecanismo básico para interação entre processos o rendez-vous. Os processos em ADA são chamados tarefas. As tarefas em ADA podem ser criadas dinamicamente, sendo que a execução do criador de uma tarefa não pode terminar antes que a tarefa criada também tenha terminado.

As tarefas definem procedimentos e entradas. As entradas são equivalentes a procedimentos que podem ser chamados apenas quando a tarefa estiver executando um comando ACCEPT correspondente. Entradas são chamadas executando um CALL remoto. As comunicações seletivas entre tarefas são possíveis mediante o uso do comando SELECT.

Os comandos CALL e ACCEPT são bloqueantes. O bloqueio de uma tarefa quando da execução de um CALL pode ser evitado usando um CALL condicional que efetiva a chamada se o rendez-vous for possível imediatamente.

O bloqueio num ACCEPT pode ser evitado usando um mecanismo que permite determinar o número de chamadas esperando por esse ACCEPT. O bloqueio num SELECT pode ser evitado através de uma cláusula ELSE que será executada quando não for possível efetuar nenhum rendez-vous naquele instante.

Embora nas chamadas remotas de procedimentos possam ser passados apontadores como argumentos, a linguagem ADA não oferece nenhum mecanismo para controlar o acesso concorrente à memória compartilhada.

2.2.3. OCCAM

A linguagem OCCAM permite que um sistema seja descrito como uma coleção de processos concorrentes, que se comunicam entre si e com os dispositivos periféricos através de canais.

Um programa em OCCAM é construído a partir de três processos primitivos:

v:=e atribui a expressão e à variável v

c! e envia a expressão e para o canal c

c? v recebe na variável v um valor do canal c.

Os processos primitivos são combinados através de construtores a fim de formar construções. Há três construtores principais que definem a maneira em que os processos componentes são executados:

SEQ = executa os processos componentes em sequência

PAR = executa os processos componentes em paralelo

ALT = executa o primeiro componente pronto.

As construções IF e WHILE também são fornecidas. Uma construção é por si só um processo e pode ser usada como componente de outra construção.

Os componentes de uma construção paralela constituem processos que podem comunicar-se unicamente através de canais. Um canal fornece um caminho de comunicação entre dois processos concorrentes. A comunicação é sincronizada e ocorre apenas quando ambos os processos estiverem prontos.

A linguagem OCCAM manipula basicamente palavras (words) do computador. As palavras podem ser usadas para representar números, caracteres, etc. A estrutura de dados mais complexa que a linguagem suporta é o vetor de palavras. OCCAM foi concebida para ser a menor e mais simples linguagem para a programação de sistemas compostos por um grande número de microcomputadores conectados.

2.2.4. CONIC

A linguagem CONIC está baseada no PASCAL e apresenta como mecanismo de comunicação e sincronização entre processos (chamados TASK MODULES) a troca de mensagens através de portas de comunicação. Em CONIC é feita uma clara separação entre a programação dos componentes de uma aplicação e a sua configuração. Isto visa basicamente possibilitar a reconfiguração dinâmica do sistema. Um sistema em CONIC consiste de instâncias de módulos, cuja interconexão é especificada numa descrição de configuração. A comunicação entre módulos é feita através da troca de mensagens. Um módulo envia e recebe mensagens através de portas locais de saída e entrada respectivamente.

As portas de um módulo têm um tipo associado e apenas mensagens desse tipo podem ser transmitidas por elas. O estabelecimento dos canais lógicos de comunicação entre os módulos é feito especificando, na descrição da configuração, a conexão de portas de saída a portas de entrada do mesmo tipo.

Neste capítulo foram introduzidos diversos conceitos e mecanismos importantes para a programação de sistemas concorrentes, bem como, algumas linguagens que incorporando esses mecanismos simplificam o desenvolvimento desses programas. No próximo capítulo será apresentado um ambiente de programação, baseado no sistema CONIC, que oferece ferramentas para a programação e configuração independente de sistemas concorrentes.

CAPÍTULO 3

O AMBIENTE PARA PROGRAMAÇÃO CONCORRENTE

3. O AMBIENTE PARA PROGRAMAÇÃO CONCORRENTE

O ambiente de desenvolvimento proposto fornece ao projetista de software em tempo real um conjunto de ferramentas que facilitam a estruturação, desenvolvimento, teste e configuração da aplicação. O ambiente está associado a uma metodologia de desenvolvimento de software [LOP 85] e procura retratar o projeto de uma placa de hardware onde os componentes ("chips") traduzem-se em módulos que encapsulam dados e funções e que interagem com o exterior por meio de uma interface precisamente definida através de portas de entrada e saída.

O ambiente proposto é baseado no sistema CONIC [KRA 83] em desenvolvimento no Departamento de Computação do Imperial College (Londres) constituindo-se, basicamente, de duas linguagens: Linguagem para Programação de Módulos - LPM e Linguagem para Configuração de Módulos - LCM. Dentre as várias linguagens de alto nível PASCAL foi escolhida para servir de base à definição da LPM por ser uma linguagem poderosa, bastante difundida, por apresentar uma verificação rígida de tipos e por permitir o projeto de programas bem estruturados e modulares.

A LCM é a ferramenta oferecida ao usuário para configurar a sua aplicação. A especificação de uma configuração corresponde a um programa LCM que descreve a construção de um sistema a partir de um subconjunto de componentes funcionais (módulos LPM).

Este capítulo tem como objetivo a apresentação das linguagens LPM e LCM onde será usada uma extensão do formalismo BNF. Os símbolos não terminais aparecem entre os caracteres < e >. As construções opcionais são representadas entre os caracteres [e]. Caso se queira indicar zero ou mais repetições de uma construção opcional o símbolo] será seguido de um asterístico (*). Se um dos símbolos usados no formalismo de descrição fizer parte da construção sendo definida ele aparecerá entre aspas.

A fim de facilitar o entendimento da sintaxe e da semântica da LPM e da LCM, nos itens onde as construções das linguagens são apresentadas, referencia-se os exemplos do Capítulo 6, indicando entre parênteses uma ou mais páginas onde elas são empregadas.

3.1. Linguagem de Programação de Módulos - LPM

A Linguagem de Programação de Módulos (LPM) foi definida a partir da versão 2.4 da Linguagem de Programação do sistema CONIC [KRA 84], incorporando ao PASCAL [ISO 83] uma série de extensões que permitem:

- A declaração de tipos de módulos;
- A declaração de portas de entrada e saída;
- A declaração de mensagens;
- O envio e recepção de mensagens de comunicação e sincronização;
- A mudança da prioridade de um módulo;
- A suspensão de um módulo durante um período de tempo;
- O tratamento de interrupções.

A seguir, apresenta-se a descrição da sintaxe e da semântica das extensões ao PASCAL introduzidas na LPM.

3.1.1. Declaração de um Tipo de Módulo

Um tipo de módulo é declarado usando a seguinte sintaxe (v. ex. p. 100):

```
<módulo> ::=
     MODULE <identificador> [(<parâmetros formais>)];
     [<definição de contexto>;]*
     [<declaração de uma porta>;]*
     [<declaração de uma mensagem>;]*
     <bloco LPM>.
```

O módulo é a maior entidade de programa que pode ser construída na linguagem LPM. O identificador após a palavra reservada MODULE é o nome do tipo de módulo sendo definido, o qual será utilizado para criar as suas instâncias na fase de configuração de uma aplicação.

A cada parâmetro formal de um módulo será atribuído o valor de um parâmetro real durante a criação das instâncias do módulo. Os parâmetros de um módulo representam variáveis locais ao módulo e devem ser de tipos simples do PASCAL (BOOLEAN, CHAR, INTEGER e REAL).

O <bloco LPM> é uma extensão de um bloco PASCAL onde, em adição aos comandos, procedimentos e funções do PASCAL, podem ser usados os comandos SEND, RECEIVE, REPLY, FORWARD e SELECT, os procedimentos pré-declarados ABORT, SETPRIORITY, DELAY, TIME e

WAITIO, e as funções pré-declaradas REASON, LINKED, QLEN e INTALLOC.

3.1.2. Definição de Contexto

As definições de contexto servem para identificar os objetos (constantes, tipos, procedimentos e funções) comuns a vários módulos que foram declarados em unidades de definição. A definição de contexto procura evitar que os objetos comuns a diversos módulos sejam declarados em cada um deles, assegurando que a alteração de um objeto seja perceptível a todos os módulos que o usam. Uma definição de contexto deve seguir a seguinte sintaxe (v. ex. p. 100):

```
<definição de contexto>::=
     [FROM <caminho>] USE <contexto> [, <contexto>]*;
<contexto>::=
     <unidade de definição>: <objeto> [, objeto>]*
```

O <caminho> é uma informação dependendo do ambiente de implementação e especifica onde uma unidade de definição se encontra (diretório, base de dados, etc.). Uma unidade de definição geralmente representa um arquivo (v. ex. p. 101).

3.1.3. Declaração de Portas

De modo a permitir a realização de um trabalho cooperativo, os módulos de uma aplicação comunicam-se e sincronizam-se através de trocas de mensagens. Para permitir que o desenvolvimento de um módulo seja independente das aplicações onde será usado, nos comandos de envio e recepção de mensagens não são endereçados diretamente os parceiros da comunicação e sim portas de entrada e saída locais a cada um. A criação dos canais lógicos de comunicação e sincronização de uma aplicação é feita mediante o estabelecimento de conexões entre portas do mesmo tipo na fase de configuração da aplicação. A declaração de uma porta deve obedecer à seguinte sintaxe:

```
<porta>::=
     ENTRYPORT <porta de entrada> [; <porta de entra-
          da>]*;
     EXITPORT <porta de saída> [; <porta de saída>]*;
<porta de entrada>::=
     <lista de portas>: <tipo da mensagem de entrada>
          [REPLY <tipo da mensagem de resposta> | QUEUE
          <constante inteira>]
<porta de saída>::=
     <lista de portas>: <tipo da mensagem de saída>
          [REPLY <tipo da mensagem de resposta>]
<lista de portas>::= <identificador de porta> [, <iden-
     tificador de porta>]*
<identificador de porta>::= <identificador> ["[" <tipo
     do índice "]"]
```

Famílias de portas podem ser declaradas especificando um <tipo de índice> após o identificador de uma porta. Uma família de portas consiste de um vetor de portas do mesmo tipo. Quando se desejar fazer uma comunicação usando uma família de portas, nos comandos de envio e recepção de mensagens, o nome da família deve ser seguido de um índice que especifique uma porta da família (v. ex. p. 114).

A declaração de uma porta define o tipo de comunicação (ou transação) na qual será empregada, possibilitando os seguintes tipos de comunicação:

a) SÍNCRONA (v. ex. p. 100): Uma porta síncrona é definida usando uma cláusula REPLY na sua declaração. Várias portas de saída síncronas podem ser conectadas a uma mesma porta de entrada síncrona. As <mensagens de saída> enviadas pelas portas de saída síncronas são enfileiradas na porta de entrada de destino. Apenas uma mensagem oriunda de uma determinada porta de saída pode estar enfileirada numa dada porta de entrada num determinado instante. A chegada a uma porta de entrada de uma mensagem faz com que a mensagem antiga proveniente da mesma porta de saída (se ainda houver uma enfileirada na porta de entrada) seja descartada.

b) ASSÍNCRONA (v. ex. p. 100): Uma porta assíncrona não contém em sua declaração uma cláusula REPLY podendo no entanto, se for de entrada, conter uma cláusula QUEUE. A cláusula QUEUE define uma fila onde serão armazenadas as mensagens que chegarem à porta de entrada. Uma porta de entrada assíncrona sempre terá uma fila associada que no caso da sua declaração não conter uma cláusula QUEUE será

de tamanho 1, correspondendo ao espaço para armazenar uma mensagem do tipo da porta. Caso a fila de entrada de uma porta esteja cheia quando da chegada de uma mensagem, a mensagem mais antiga na fila será descartada para deixar espaço para a que acaba de chegar. As mensagens que chegam a uma porta de entrada assíncrona são enfileiradas em ordem de chegada, independentemente da porta de saída de onde provém. Uma porta de saída assíncrona pode ser conectada a mais de uma porta de entrada. Nesse caso cada mensagem enviada a essa porta de saída será copiada em todas as portas de entrada destinatárias.

3.1.4. Declaração de Mensagens

A declaração de mensagens na LPM é análoga à declaração de variáveis em PASCAL. A sua sintaxe é a seguinte (v. ex. p. 100):

```
<declaração de mensagens>::=
     MESSAGE <d_mensagens> [; <d_mensagens>]*
<d_mensagens>::=
     <lista de mensagens>: <tipo>
<lista de mensagens>::=
     <identificador> [, <identificador>]*
```

As mensagens são totalmente compatíveis com variáveis de mesmo tipo. Entretanto, nos comandos de envio e recepção devem ser especificadas mensagens e não variáveis comuns. O tipo de uma mensagem envolvida numa comunicação deve ser compatível com o tipo da porta empregada.

3.1.5. Envio e Recepção de Mensagens de Comunicação e Sincronização

Os comandos de troca de mensagens da LPM permitem estabelecer comunicações assíncronas e síncronas. Numa comunicação assíncrona o módulo emissor só será suspenso se a mensagem enviada estiver sendo esperada por um módulo de maior prioridade. Uma comunicação síncrona estabelece uma troca bidirecional de mensagens, que consiste no envio de uma mensagem num sentido e de uma resposta no outro. A operação de envio é síncrona e suspende o módulo emissor até a chegada de uma resposta. Em vez de enviar uma mensagem de resposta normal o módulo receptor de uma mensagem síncrona pode cancelar a comunicação enviando uma resposta indi-

cativa da ocorrência de uma falha (vide procedimento ABORT p.28) ou redirecionar a mensagem recebida para outra porta (vide comando FORWARD). A expressão abaixo relaciona os comandos de troca de mensagens da LPM.

```
<comando de troca de mensagens>::=
     <comando de envio> | <comando de recepção> |
          <comando de resposta> | <comando de recepção
               seletiva>
```

3.1.5.1. Comando de Envio de Mensagens

O envio assíncrono de uma mensagem deve ser feito mediante o uso de uma porta assíncrona, analogamente, uma troca síncrona de mensagens deve ser feita através de uma porta síncrona. As mensagens envolvidas em comunicações síncronas e assíncronas devem ter o mesmo tipo (ou formato) das portas através das quais são transmitidas. Os comandos de envio de uma mensagem suportam ambos tipos de comunicação. A sua sintaxe é a seguinte:

```
<comando de envio>::=
     SEND <mensagem> TO <porta de saída> [<cláusula de
          resposta>]
<cláusula de resposta>::=
     WAIT <mensagem> [<opções de resposta> END]
<opções de resposta>::=
     => <sequência de comandos> [<trata falha>] | <tra-
        ta falha>
<trata falha>::=
     FAIL [<tempo>] => <sequência de comandos>
<tempo>::=
     <expressão inteira>
```

O envio assíncrono de uma mensagem tem o seguinte formato (v. ex. p. 100):

```
SEND <mensagem> TO <porta de saída>
```

Este comando não produzirá nenhum efeito se a <porta de saída> não estiver conectada. A execução deste comando somente causará a suspensão do módulo emissor se a mensagem enviada estiver sendo aguardada por um módulo mais prioritário.

O comando de envio síncrono de uma mensagem tem o seguinte formato (v. ex. p. 100):

```
SEND <mensagem1> TO <porta de saída>
     WAIT <mensagem2>
```

Este comando envia a <mensagem1> à <porta de saída> e suspende a execução do módulo que o executou até que a <mensagem2> seja recebida. Se a <porta de saída> não tiver sido conectada antes do início da execução do comando ou for desconectada antes de sua conclusão, será gerado um erro de execução do comando ("run-time error"). Se o módulo receptor da <mensagem1> cancelar a transação, mediante o uso do procedimento **ABORT**, também será gerado um erro de execução.

O envio síncrono de uma mensagem com cláusula para tratamento de falha tem o seguinte formato:

```
SEND <mensagem1> TO <porta de saída>
     WAIT <mensagem2> => <seqüência de comandos 1>
     FAIL <tempo> => <seqüência de comandos 2>
END
```

A semântica deste comando é basicamente a mesma do comando de envio síncrono não temporizado. Adicionalmente a <seqüência de comandos 1> será executada se a <mensagem2> de resposta for recebida com sucesso. A <seqüência de comandos 2> será executada nos seguintes casos:

a) O período especificado por <tempo> expira sem que a mensagem de resposta tenha sido recebida;

b) A <porta de saída> não estava conectada no momento que se iniciou a execução do comando, ou ela é desconectada antes de sua conclusão;

c) A transação é cancelada, antes da recepção da resposta, pelo suporte da linguagem ou pelo módulo receptor.

Uma mensagem de resposta é descartada caso chegue após o <tempo> de espera ter expirado. Na seqüência de comandos da cláusula **FAIL** pode-se determinar o motivo que ocasionou o cancelamento da troca de mensagens através da função pré-declarada **REASON**, descrita na página 28.

3.1.5.2. Comandos de Recepção e Resposta

O comando de recepção de uma mensagem tem a seguinte sintaxe (v. ex. p. 102):

```
⟨comando de recepção⟩::=
     RECEIVE ⟨mensagem1⟩ FROM ⟨porta de entrada⟩
          [REPLY ⟨mensagem2⟩]
```

Quando no início da execução deste comando não houver uma mensagem disponível na ⟨porta de entrada⟩ o módulo será suspenso até a chegada de uma mensagem. A mensagem recebida será copiada para a variável especificada por ⟨mensagem1⟩. Caso o comando de recepção inclua uma cláusula REPLY, tão logo a ⟨mensagem1⟩ seja recebida será enviada, através da ⟨porta de entrada⟩, a ⟨mensagem2⟩ como resposta. Nesse caso a ⟨porta de entrada⟩ deve ser síncrona.

Em lugar de enviar uma resposta tão logo uma mensagem seja recebida é muitas vezes necessário efetuar alguma operação que inclua dados recebidos nessa mensagem. Nessas situações a mensagem de resposta é enviada através do comando REPLY que tem a seguinte sintaxe (v. ex. p. 103):

```
REPLY ⟨mensagem⟩ TO ⟨porta de entrada⟩
```

O envio de uma mensagem de resposta para uma porta de entrada deve ser precedido da recepção de uma mensagem nessa porta, caso contrário o comando REPLY não surtirá efeito algum. As mensagens de resposta transmitidas após haver expirado o período de espera, definido no comando que iniciou a troca síncrona, são descartadas. A execução do comando REPLY somente suspenderá o módulo que o executa caso o módulo destinatário tiver prioridade superior à sua.

3.1.5.3. Comando de Desvio de uma Comunicação Síncrona

Este comando pode ser usado no lugar do comando de resposta de uma comunicação síncrona. A sua sintaxe é a seguinte (v. ex. p. 102):

```
⟨comando de desvio⟩::=
     FORWARD ⟨porta de entrada⟩ TO ⟨porta de saída⟩
```

A mensagem anteriormente recebida na ‹porta de entrada› é desviada inalterada para a ‹porta de saída›. Ambas as portas devem ser do mesmo tipo. A resposta posteriormente transmitida será enviada diretamente ao emissor inicial da mensagem desviada.

3.1.5.4. Comando de Recepção Seletiva

Com o comando de recepção seletiva pode-se efetuar uma espera temporizada por uma mensagem num conjunto específico de portas. A sintaxe do comando de recepção seletiva é a seguinte (v. ex. pp. 103 e 114):

```
‹comando de recepção seletiva›::=
     ‹tipo de seleção› ‹alternativa de seleção›
          [OR ‹alternativa de seleção›]*
          [ELSE ‹sequência de comandos›]
     END
‹tipo de seleção›::= PSELECT | RSELECT
‹alternativa de seleção›::=
     [‹repetição›] [‹guarda›] ‹cláusula de seleção›
          [=› ‹sequência de comandos›]
‹repetição›::=
     FOR ‹variável de controle›:= ‹valor inicial›
          (TO | DOWTO) ‹valor final› DO
‹guarda›::=
     WHEN ‹expressão booleana›
‹cláusula de seleção›::=
     ‹comando de recepção› | ‹cláusula de tempo›
‹cláusula de tempo›::=
     TIMEOUT ‹unidades de tempo›
‹unidades de tempo›::=
     ‹expressão inteira›
```

A execução de um comando de recepção seletiva dá-se da seguinte forma:

a) A determinação da elegibilidade das cláusulas de seleção

São consideradas elegíveis as cláusulas de seleção não precedidas por guardas ou associadas a guardas cuja avaliação resulta no valor TRUE. O uso da cláusula de ‹repetição› equivale a repetir a avaliação da guarda associada para cada valor da ‹variável de controle› desde o ‹valor inicial› até o ‹valor final›.

b) A seleção de uma mensagem

Caso mais de uma das portas referenciadas, nos comandos de recepção das cláusulas elegíveis, tiver recebido uma ou mais mensagens, um desses comandos deve ser escolhido. A escolha é feita de acordo com o ‹tipo de seleção› especificado. Caso a seleção seja do tipo priorizado, i.e., caso o comando de seleção seja um PSELECT (v. ex. p. 114), é selecionada a mensagem associada à cláusula de maior prioridade. A prioridade num comando PSELECT decresce a partir da primeira cláusula, ou seja, a primeira cláusula é mais prioritária que a segunda, que por sua vez é mais prioritária que a terceira e assim por diante. Caso a seleção seja do tipo aleatório, i.e., caso o comando de seleção seja um RSELECT (v. ex. p. 114), a cláusula é escolhida de forma aleatória.
Se nenhuma das portas das alternativas de seleção tiver recebido uma mensagem, o módulo será suspenso até que uma mensagem chegue a uma dessas portas.
Após a execução do comando de recepção selecionado é executada a ‹sequência de comandos› a ele associada. No caso do comando ser precedido de uma cláusula de ‹repetição›, antes de executar a sua cláusula REPLY e a sequência de comandos associada, é atribuído à ‹variável de controle› do FOR o seu valor quando o comando de recepção tornou-se elegível.

c) A seleção de uma cláusula de tempo

O tempo durante o qual um módulo deve esperar pela chegada de uma mensagem pode ser limitado por uma ou mais ‹cláusulas de tempo›. Uma cláusula de tempo elegível é selecionada se após esperar o número de ‹unidades de tempo› especificado pela cláusula, o módulo não tiver recebido nenhuma mensagem. Se várias cláusulas de tempo forem elegíveis é selecionada aquela que especifica a menor espera. Se a cláusula de tempo selecionada for precedida por uma ‹repetição›, o tratamento a dar à ‹variável de controle› é o mesmo dado quando da seleção de uma mensagem.

d) A seleção de uma cláusula ELSE

Uma cláusula ELSE é equivalente a uma cláusula de tempo com espera igual a zero. Quando não houver mensagens disponíveis nas portas de entrada, especificadas pelos comandos de recepção, o uso desta cláusula impede que o módulo seja suspenso.

3.1.5.5. Procedimentos e Funções Relacionados à Troca de Mensagens

A LPM incorpora diversos procedimentos e funções pré-declarados que podem ser usados em conjunto com os comandos de troca de mensagens.

O procedimento **ABORT** pode ser usado no lugar de um comando REPLY para cancelar uma comunicação síncrona; ele está declarado da seguinte forma:

```
PROCEDURE ABORT (<porta de entrada>; <motivo falha>:
                 INTEGER);
```

onde <porta de entrada> especifica o nome de uma porta de entrada síncrona e <motivo falha> indica o motivo pelo qual a comunicação está sendo cancelada.

A função padrão **REASON** pode ser usada, na cláusula FAIL de um comando de envio síncrono, para saber o código da falha que ocasionou o cancelamento da comunicação. A função **REASON** está declarada da seguinte forma:

```
FUNCTION REASON: INTEGER;
```

O valor retornado por **REASON** corresponderá às constantes pré-declaradas **etimeout** e **elink** ou ao <motivo falha> indicado na execução do comando **ABORT** pelo módulo destinatário da mensagem síncrona. O código **etimeout** indica que a cláusula FAIL foi executada porque o tempo de espera por uma resposta expirou. O código **elink** indica que a porta de saída especificada no comando **SEND** não está conectada.

A função **LINKED** pode ser usada para saber se uma porta de saída está conectada ou não, ela está declarada da seguinte forma:

```
FUNCTION LINKED <porta de saída>: BOOLEAN;
```

onde <porta de saída> especifica o nome da porta de saída que se deseja saber se está conectada. A função retornará o valor **TRUE** se a porta estiver conectada e **FALSE** em caso contrário.

A determinação do número de mensagens disponíveis numa porta de entrada pode ser feita mediante a função **QLEN**, ela está declarada da seguinte forma:

```
FUNCTION QLEN (<porta de entrada>) : INTEGER;
```

3.1.6. Mudança da Prioridade de um Módulo

Um módulo tem a sua prioridade inicialmente determinada em tempo de configuração, entretanto, se ele desejar alterá-la deverá usar o procedimento SETPRIORITY, que está declarado da seguinte forma (v. ex. p. 100):

```
PROCEDURE SETPRIORITY (<nível de prioridade>: PRIORITY)
```

onde o tipo PRIORITY é pré-declarado como:

```
PRIORITY = (SYSTEMPR, HIGHPR, NORMALPR, LOWPR,
            LOWESTPR);
```

- Se a execução deste procedimento implicar numa diminuição da prioridade do módulo, ele poderá ser suspenso para permitir que outro módulo mais prioritário seja executado.

3.1.7. Suspensão de um Módulo durante um Período de Tempo

Um módulo pode ser suspenso por um determinado período de tempo usando o procedimento DELAY, declarado da seguinte forma (v. ex. p. 100):

```
PROCEDURE DELAY (<período de tempo>: LONGINTEGER)
```

onde <período de tempo> especifica o número de unidades de tempo durante o qual o módulo deve ser suspenso. A duração de uma unidade de tempo é dependente da implementação do suporte de execução da LPM. O tipo LONGINTEGER corresponde a um inteiro representado em 32 bits.

O relógio interno ao suporte da LPM pode ser lido através da função TIME, declarada da seguinte forma (v. ex. p. 100):

```
PROCEDURE TIME: LONGINTEGER
```

A função TIME retorna o valor, em unidades de tempo, marcado pelo relógio do sistema.

3.1.8. Tratamento de Interrupções

Como o processamento da ocorrência de eventos assíncronos é uma das atividades mais importantes nas aplicações de controle em tempo real, a LPM oferece a possibilidade de manipular a ocorrência de interrupções em um alto nível. Isto traduz-se basicamente pelo fato da programação dos tratadores de interrupções ser realizada como se estes fossem módulos de aplicação. A principal diferença entre os módulos de tratamento de interrupções e os demais é a utilização, por parte daqueles, da função INTALLOC e do procedimento WAITIO, descritos nos itens a seguir.

3.1.8.1. Mapeamento de um Elemento do Vetor Físico de Interrupções

A ocorrência de uma interrupção está associada a um determinado elemento do vetor físico de interrupções, elemento este que contém o endereço físico de uma sequência de instruções que será executada quando da ocorrência da interrupção. Para tratar uma interrupção o módulo deve mapear o elemento do vetor físico dessa interrupção, em um elemento de um vetor lógico de interrupções do ambiente. Este mapeameno é feito pela função INTALLOC, declarada da seguinte forma (v. ex. p. 105):

```
FUNCTION INTALLOC (<vetor físico>: ADDRESS): INTEGER
```

onde <vetor físico>, informação dependente do hardware onde o ambiente é implementado, identifica o endereço inicial da sequência de instruções que o processador executará em atendimento a uma dada interrupção. O tipo ADDRESS representa um endereço absoluto de memória. A função INTALLOC retorna um valor inteiro, correspondente a um elemento de um vetor lógico de interrupções, que deverá ser usado pelo módulo quando desejar aguardar a ocorrência da interrupção associada.

3.1.8.2. Espera pela Ocorrência de uma Interrupção

Um módulo aguarda a ocorrência de uma interrupção através do procedimento WAITIO, declarado da seguinte forma (v. ex. p. 105):

```
PROCEDURE WAITIO (<vetor lógico>: INTEGER)
```

onde ‹vetor lógico› é o número do vetor lógico da interrupção que o módulo deseja aguardar. A execução de **WAITIO** não suspende a execução do módulo se a interrupção especificada tiver ocorrido desde a última vez que **WAITIO** foi chamada para essa interrupção. Caso a interrupção ainda não tenha ocorrido, o módulo será suspenso à sua espera.

3.2. Linguagem de Configuração de Módulos - LCM

A Linguagem de Configuração de Módulos (LCM) é a ferramenta através da qual o usuário descreve uma configuração. Um programa de configuração é formado por declarações dos tipos de módulos que comporão uma determinada aplicação, pela criação das instâncias destes módulos (ou simplesmente módulos) e pela conexão das portas (das instâncias) dos módulos.

A sintaxe de um programa de configuração é a seguinte (v. ex. p. 107):

‹programa de configuração›::=
CONFIGURATION ‹identificador›;
‹declaração de instâncias›;
‹criação de instâncias›;
[‹conexão de portas›]*
END.

3.2.1. Declaração de Instâncias de Módulos

Na declaração de instâncias são definidas as cópias lógicas de cada tipo de módulo, atribuindo um nome lógico a cada uma.

A declaração de instâncias de módulos obedece à seguinte sintaxe (v. ex. p. 107):

‹declaração de instâncias›::=
INSTANCE ‹lista de identificadores› [;‹lista de identificadores›]*
‹lista de identificadores›::=
‹identificador› [,‹identificador›]* : ‹tipo de módulo›

3.2.2. Criação de Instâncias

Na criação de instâncias são determinados os valores dos parâmetros reais com que estas serão criadas. Opcionalmente, podem ser usadas diretivas para definir a prioridade inicial das instâncias e o tamanho das áreas de memória que cada uma deseja usar para pilha e "heap". Caso estas diretivas não sejam usadas prevalecem valores pré-declarados, característicos da implementação da LCM. O uso de diretivas será ilustrado nos exemplos do Capítulo 6.

A criação de instâncias de módulos obedece à seguinte sintaxe (v. ex. p. 107):

```
<criação de instâncias>::=
     CREATE <lista de instâncias>
<lista de instâncias>::=
     <instância> [, <instância>]*;
<instância>::=
     <identificador> [(<parâmetros reais da instân-
          cia>)]
<parâmetros reais da instância>::=
     <parâmetro> [, <parâmetro>]*
<parâmetro>::=
     <constante numérica> | <constante lógica> | <ca-
          racter>
```

3.2.3. Conexão de Portas

A conexão das portas dos módulos estabelece os canais lógicos de comunicação através dos quais eles se comunicarão. Num comando de conexão especifica-se sempre a ligação de portas de saída a portas de entrada. O estabelecimento de ligações do tipo um para muitos é possível somente entre portas assíncronas. As ligações muitos para um podem ser realizadas tanto entre portas assíncronas como entre portas síncronas.

A sintaxe do comando de conexão de portas é a seguinte (v. ex. p. 107):

```
<conexão de portas>::=
     LINK <conexão> [, <conexão>]*;
<conexão>::=
     <id. porta de saída> TO <lista de portas de entra-
          da>
```

```
<lista de portas de entrada>::=
     <id. porta de entrada> [, <id. porta de entra-
          da>]*;
<id. porta de saída>::=
     <id. porta>
<id. porta de entrada>::=
     <id. porta>
<id. porta>::=
     <módulo>.<porta>
```

Conforme se observa nas expressões acima, a identificação de uma porta requer a inclusão do nome da instância à qual pertence. Isto é necessário pois as portas de todas as instâncias de um mesmo tipo de módulo tem o mesmo nome.

É importante ressaltar que a conexão entre uma porta de saída e uma de entrada só é possível se ambas forem do mesmo tipo.

3.3. Implementação do Ambiente

A implementação do ambiente para programação concorrente foi dividida em dois trabalhos desenvolvidos em paralelo:

- Implementação de um pré-compilador para a linguagem LPM e um "tradutor" para a LCM;
- Implementação do suporte de tempo real para execução das aplicações escritas em LPM-LCM.

O desenvolvimento destes dois trabalhos está relacionado diretamente a três ferramentas: o YACC e o LEX [LES 82] disponíveis no ambiente UNIX e um compilador PASCAL disponível no mercado nacional que implementa o PASCAL segundo norma ISO. A implementação das linguagens é descrita em [LOP 86], sendo sucintamente discutida no próximo item. A implementação do suporte de tempo real será assunto dos próximos capítulos.

3.3.1. Implementação do Pré-Compilador LPM e do Configurador LCM

O pré-compilador LPM aceita como entrada um programa escrito em LPM e fornece como resultado um programa-fonte PASCAL e um conjunto de tabelas que serão utilizadas posteriormente na fase de configuração do sistema. As informações contidas nestas tabelas indicam, por exemplo, as portas de entrada e saída do módulo

pré-compilado, assim como, os tipos destas portas de modo a permitir um teste de consistência durante a fase de configuração.

Conforme mencionado anteriormente na implementação das linguagens utilizam-se duas ferramentas do ambiente UNIX [RIT 74]: LEX e YACC [LES 82]. O LEX é um gerador de programas que, baseado em uma tabela fornecida pelo usuário contendo expressões regulares e seções de programas associadas às expressões, gera um autômato determinístico traduzido em um programa-fonte em C [KER 78]. Quando da sua execução este programa particiona um conjunto de caracteres de entrada em cadeias de caracteres reconhecidas pelas expressões regulares da tabela fornecida ao LEX. Para cada expressão regular reconhecida no conjunto de caracteres de entrada a seção de programa associada é executada. A Figura 3.1 fornece uma visão do LEX onde fonte é a tabela de expressões regulares e ações associadas; YYlex é o programa da análise léxica gerado; entrada é o conjunto de caracteres a ser analisado e saída é um conjunto de ações realizadas pelo analisador em função das expressões regulares reconhecidas.

O YACC é um gerador de analisadores sintáticos a partir de uma especificação de entrada contendo a sintaxe desejada, baseado na teoria de analisadores LALR (1) implementando, ainda, regras de decisão no caso da presença de gramáticas ambíguas. Tal como no LEX, para cada regra definida o YACC possui uma seção de código em C que é executada quando a regra é reconhecida.

Em geral, uma combinação entre o YACC e o LEX é bastante apropriada pois o LEX é utilizado como um pré-processador para o analisador sintático. Este pré-processamento ocorre através do particionamento do conjunto de caracteres de entrada em "tokens" fornecidos ao analisador sintático gerado pelo YACC que valida as expressões de entrada e toma as ações associadas às regras gramaticais definidas.

A versão atual da linguagem de configuração LCM é bastante simplificada pelo aspecto estático associado, facilitando bastante a sua implementação através do YACC e LEX.

A implementação do pré-compilador LPM possiu como estratégia básica a cópia direta, para o arquivo de saída, dos comandos PASCAL encontrados no corpo do programa do módulo. Quando do reconhecimento dos comandos e declarações LPM (SELECT, SEND, RECEIVE, declarações de portas e mensagens, etc.) o pré-processador deve executar ações caracterizadas, basicamente, pela tradução dos comandos e declarações em chamadas de procedimentos do suporte de tempo real (STR) e na criação de estruturas de dados a serem utilizadas na fase de configuração.

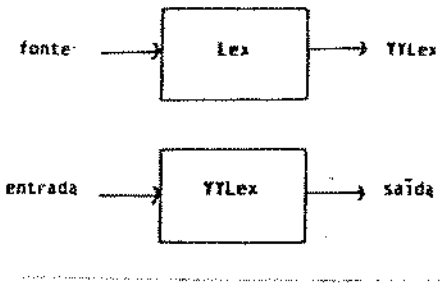

FIGURA 3.1 - Esquema básico do LEX

Neste capítulo foi descrito o ambiente de programação concorrente proposto, introduzindo-se as linguagens LPM e LCM. No próximo capítulo será apresentado o Suporte de Tempo Real (STR) que oferece os serviços necessários à execução das aplicações desenvolvidas usando a LPM e a LCM.

CAPÍTULO 4

SUPORTE DE TEMPO REAL (STR)

4. SUPORTE DE TEMPO REAL (STR)

O desenvolvimento do STR tem dois aspectos principais, o primeiro, independente de hardware, compreende a implementação dos serviços que suportam as extensões da LPM à linguagem PASCAL, o segundo diz respeito à implantação do STR numa arquitetura específica de hardware.

A implementação dos serviços do STR é independente do hardware, traduzindo-se nas estruturas de dados necessárias à representação das entidades de uma dada aplicação e nas rotinas que operando sobre essas estruturas de dados implementam os serviços.

A implantação do STR num hardware específico consiste basicamente dos mecanismos para carga de uma aplicação, interface entre a aplicação e o STR e atendimento de interrupções.

Este capítulo apresenta a implementação do STR de um ponto de vista independente do hardware. Na Seção 4.1 as extensões da LPM à linguagem PASCAL são associadas aos serviços do STR. Na Seção 4.2 descreve-se a implementação desses serviços. Finalmente, na Seção 4.3 são discutidas duas arquiteturas para a implantação do STR num equipamento compatível com o IBM-PC, uma delas será escolhida e descrita no Capítulo 5.

A descrição dos serviços do STR em duas seções procura contemplar a dois tipos diferentes de leitores. A seção 4.1 destina-se aos projetistas das linguagens LPM e LCM, e aos usuários desejando analisar os programas PASCAL resultantes da pré-compilação de módulos LPM. Essa seção apresenta apenas a interface dos serviços, não incluindo informações relativas à sua implementação, informações que, por não interessarem à classe de leitores citada, tornariam a descrição dos serviços desnecessariamente densa.

A seção 4.2 apresenta os principais aspectos da implementação dos serviços do STR. Esta seção destina-se a leitores que pretendam estudar o STR ao nível dos programas que o implementam.

4.1. Serviços Oferecidos pelo Suporte de Tempo Real

Nesta seção os comandos, procedimentos e funções pré-declaradas da LPM que não fazem parte do PASCAL padrão são associados aos serviços que lhes dão suporte em tempo de execução. Descreve-se basicamente a interface de cada serviço sem maiores detalhes relativos à sua semântica ou implementação já que o primeiro aspecto foi tratado no Capítulo 3 e o segundo será coberto no item 4.2.

Convem lembrar que a interface aqui descrita para cada serviço não é diretamente acessível ao usuário; este faz a programação dos seus módulos usando as construções da LPM apresentadas no Capítulo 3, as chamadas aos serviços do STR são geradas pelo pré-processador da LPM como resultado da tradução dos módulos fonte em LPM para os programas fonte em PASCAL.

A fim de facilitar o entendimento do processo de tradução das construções da LPM em chamadas aos serviços do STR, a exemplo do que foi feito no Capítulo 3, nos itens onde são apresentadas as interfaces dos serviços referencia-se os programas em PASCAL, dos Anexos A e B, resultantes da pré-compilação dos exemplos do Capítulo 6.

4.1.1. Troca de Mensagens

Os serviços de troca de mensagens suportam a execução das primitivas e funções relacionadas ao envio e recepção de mensagens através das quais os módulos se comunicam e sincronizam.

4.1.1.1. Envio Assincrono de uma Mensagem

O comando de envio assincrono de uma mensagem

SEND <mensagem> to <porta_de_saída>

é suportado pelo serviço **env_a** cuja interface é dada por (v. ex. p. 123):

PROCEDURE env_a (prt_saída, tam_men: INTEGER; end_men: ADDRESS);

onde **prt_saída** é o número correspondente a <porta_de_saída> através da qual a <mensagem> deve ser enviada; **tam_men** é o tamanho da mensagem a enviar e **end_men** é a sua posição no espaço de endereçamento do módulo emissor.

4.1.1.2. Envio Síncrono de uma Mensagem

O comando de envio síncrono de uma mensagem

```
SEND <mensagem> TO <porta_de_saída> WAIT <resposta>
```

é suportado pelo serviço **env_s** cuja interface é dada por (v. ex. p. 123):

```
PROCEDURE env_s (prt_saída, tam_men: INTEGER; end_mem,
                 end_resp: ADDRESS);
```

onde os parâmetros **prt_saída**, **tam_men** e **end_mem** são idênticos em significado aos seus correspondentes no serviço **env_a**; o parâmetro **end_resp** especifica o endereço onde deverá ser copiada a mensagem de <resposta>.

4.1.1.3. Envio Síncrono de uma Mensagem com Cláusula para Tratamento de Falha

O comando de envio síncrono de uma mensagem com cláusula para tratamento de falha

```
SEND <mensagem> TO <porta_de_saída>
     WAIT <resposta> => S1
     FAIL (tempo de espera) => S2
END
```

é suportado pelo serviço **env_s_temp** cuja interface é dada por:

```
FUNCTION env_s_temp (tempo: LONGINTEGER; prt_saída,
         tam_men: INTEGER; end_mem, end_resp: ADDRESS):
         BOOLEAN;
```

onde os parâmetros **prt_saída**, **tam_mem**, **end_mem** e **end_resp** são idênticos em significado aos seus correspondentes no serviço **env_s**; o parâmetro **tempo** especifica o período de tempo durante o qual deve ser aguardada uma mensagem de <resposta>, após o que passa-se a executar o comando S2, caso a cláusula **FAIL** do comando SEND não inclua um tempo de espera indicando um tempo de espera infinito. O parâmetro **tempo** deve ser um número negativo.

A função **env_s_temp** que implementa o envio síncrono temporizado retomará o valor **TRUE** caso receba uma mensagem dentro do período especificado e o valor **FALSE** em caso contrário. Em função do valor retomado por **env_s_temp** será executado o comando S1 ou S2, assim uma possível tradução para o comando de envio síncrono de mensagens é:

```
IF env_s_temp (...)
   THEN S1
   ELSE S2
```

4.1.1.4. Recepção Bloqueante de uma Mensagem

O comando de recepção bloqueante de uma mensagem

```
RECEIVE <mensagem> FROM <porta de entrada>
```

é suportado pelo serviço **rec** cuja interface é dada por (v. ex. p. 124).

```
PROCEDURE rec (end_mem: ADDRESS; prt_entrada: INTEGER);
```

onde o parâmetro **end_mem** especifica a posição, do espaço de endereçamento do módulo requisitante, onde a mensagem deve ser copiada e **prt_entrada** fornece o número da porta de entrada através da qual a mensagem deve ser recebida.

4.1.1.5. Envio de uma Mensagem de Resposta

O comando não bloqueante de envio de uma mensagem de resposta em uma comunicação síncrona

```
REPLY <mensagem> TO <porta de entrada>
```

é suportado pelo serviço **resp** cuja interface é dada por (v. ex. p. 127)

```
PROCEDURE resp (prt_entrada, tam_mem: INTEGER; end_mem:
          ADDRESS);
```

onde prt_entrada é o número da porta de entrada à qual a mensagem de resposta deve ser enviada; tam_mem é o tamanho em bytes dessa mensagem e end_mem é o seu endereço.

4.1.1.6. Desvio de uma Comunicação Síncrona

O comando de desvio de uma comunicação síncrona

```
FORWARD <porta de entrada> TO <porta de saída>
```

é suportado pelo serviço desvia cuja interface é dada por (v. ex. p. 125)

```
PROCEDURE desvia (prt_saía, prt_entrada: INTEGER);
```

onde os parâmetros prt_saída e prt_entrada correspondem, respectivamente, aos números das portas <porta de saída>, para onde a mensagem será desviada, e <porta de entrada>, de onde a mensagem será desviada.

4.1.1.7. Recepção Seletiva de uma Mensagem

O suporte dos comandos PSELECT e RSELECT requer o uso dos serviços ini_rec_sel, ini_temp_sel e sel. A execução de um comando SELECT compreende duas fases, uma de inicialização e outra de seleção. Na primeira fase são inicializadas as estruturas de dados do STR associadas a cada alternativa do comando SELECT. Na segunda fase escolhe-se a sequência de comandos associada a uma das alternativas para execução. Os serviços ini_rec_sel e ini_temp_sel são empregados na fase de inicialização e são comuns aos comandos PSELECT e RSELECT. O serviço SEL é usado durante a fase de seleção dos comandos PSELECT e RSELECT.

Para cada valor da variável de cotrole do FOR de uma alternativa de um comando SELECT que contém um RECEIVE, como por exemplo:

```
PSELECT
   .
   .
   .
   OR    FOR <variável de controle>:=...TO...
         RECEIVE <mensagem> FROM <porta de entrada>
         => S1
END
```

é gerada uma chamada ao serviço ini_rec_sel cuja interface é dada por (v. ex. p. 127):

```
PROCEDURE    ini_rec_sel    (índice_for, cláusula,
             prt_entrada: INTEGER; end_men: ADDRESS);
```

onde índice_for é o valor da <variável de controle> do FOR para a qual foi gerada esta requisição; cláusula é um inteiro que identifica a recepção sendo inicializada; prt_entrada é o número da <porta de entrada> e end_mem é o endereço onde a <mensagem> deverá ser recebida.

Para cada valor da variável de controle do FOR de uma alternativa de um comando SELECT que contiver um TIMEOUT como por exemplo:

```
PSELECT
   .
   .
   .
   OR FOR <variável de controle>:=...TO...
         TIMEOUT <tempo de espera>
         => S2
END
```

é gerada uma chamada ao serviço ini_temp_sel cuja interface é dada por (v. ex. p. 129):

```
PROCEDURE ini_temp_sel (índice_for, cláusula: INTEGER;
          tempo: LONGINTEGER);
```

onde os parâmetros índice_for e cláusula têm significado idêntico aos seus correspondentes no serviço ini_rec_sel; tempo específico o <tempo de espera> especificado pela cláusula TIMEOUT.

Caso uma alternativa de um comando SELECT não inclua uma repetição (FOR...) será feita uma única chamada ao serviço de inicialização correspondente à cláusula contida na alternativa RECEIVE ou TIMEOUT, devendo ser passado um valor qualquer em correspondência ao parâmetro indice_for.

A fase de seleção de um comando SELECT corresponde à requisição do serviço **sel** cuja interface é dada por (v. ex. pp. 127 e 133):

```
FUNCTION sel (tipo: CHAR; cláusula_else: INTEGER; VAR
        indice_for, prt_entrada: INTEGER): INTEGER;
```

onde tipo especifica o tipo de seleção desejada, podendo assumir os valores P ou R que indicam escolhas priorizadas ou randômicas respectivamente, clausula_else é um valor inteiro que identifica a cláusula ELSE do comado SELECT. Caso não haja uma cláusula ELSE o parâmetro clausula_else conterá o valor zero. A funçõo sel escolhe uma cláusula do comando SELECT e retorna o valor inteiro que lhe foi associado durante a fase de inicialização. Adicionalmente em indice_for e prt_entrada são retornados, respectivamente, os valores correspondentes à variável de controle do FOR da cláusula escolhida e ao número da porta de entrada onde foi recebida a mensagem (quando for o caso).

A diferença entre os serviços sel('P',...) (v. ex. pp. 127 e 133) e sel('R',...) (v. ex. p. 133) reside no processo de seleção da cláusula a executar. O serviço sel('P',...) considera como mais prioritária a primeira cláusula a ser inicializada (via ini_rec_sel) e como menos prioritária a última. No caso de um serviço sel('R',...) todas as cláusulas RECEIVE têm igual prioridade e a escolha é feita aleatoriamente.

4.1.1.8. Cancelamento de uma Comunicação Síncrona

O procedimento pré-declarado ABORT que permite cancelar uma comunicação síncrona e cuja interface é dada por:

```
PROCEDURE ABORT (<porta de entrada>, <motivo da falha>:
        INTEGER);
```

é suportado pelo serviço resp_falha cuja interface é dada por:

```
PROCEDURE res_falha (motivo, prt_entrada: INTEGER);
```

onde os parâmetros **motivo** e **prt_entrada** correspondem respectivamente ao <motivo da falha> e ao número da <porta de entrada>. O valor inteiro **motivo** deve ser maior que a constante pré-declarada **erro_max**, cujo valor atual é 128.

4.1.1.9. Determinação da Razão de uma Falha

A função pré-declarada **REASON** que permite saber o motivo pelo qual uma troca síncrona de mensagens falhou (foi cancelada) e cuja interface é dada por:

```
PROCEDURE REASON( INTEGER);
```

é suportada pelo serviço **motivo_falha** cuja interface é dada por:

```
FUNCTION motivo_falha: INTEGER;
```

O valor retornado por **motivo_falha** corresponderá às constantes pré-declaradas **etimeout** ou **elink** ou a um valor indicado pela execução do procedimento **ABORT** no módulo que enviou a resposta à comunicação síncrona. O código **etimeout** indica que a cláusula **FAIL** foi executada porque o tempo de espera por uma resposta expirou. O código **elink** indica que se tentou estabelecer uma comunicação através de uma porta de saída não conectada.

4.1.1.10. Teste de Conexão de uma Porta de Saída

A função pré-declarada **LINKED** que permite saber se uma dada porta de saída está conectada a alguma porta de entrada e cuja interface é dada por:

```
FUNCTION LINKED (<porta de saída>): BOOLEAN;
```

é suportada pelo serviço **conectada**, cuja interface é dada por:

```
FUNCTION conectada (prt_saída: INTEGER): BOOLEAN;
```

onde **prt_saída** é o número da <porta de saída> a ser testada.

4.1.1.11. Determinação da Quantidade de Mensagens Enfileiradas Numa Porta de Entrada

A função pré-declarada QLEN que permite saber quantas mensagens estão enfileiradas numa porta de entrada (bufereada ou não) e cuja interface é dada por:

```
FUNCTION QLEN (<porta de entrada>);
```

é suportada pelo serviço mem_prt_ent cuja interface é dada por:

```
FUNCTION mem_prt_ent (prt_entrada: INTEGER): INTEGER;
```

onde prt_entrada é o número da <porta de entrada>.

4.1.2. Mudança da Prioridade de um Módulo

O procedimento pré-declarado SETPRIORITY que permite alterar a prioridade de um módulo e cuja interface é dada por:

```
PROCEDURE SETPRIORITY (<nível de prioridade>:
          PRIORITY);
```

é suportado pelo serviço muda_prio, cuja interface é dada por:

```
PROCEDURE muda_prio (prio: PRIORITY);
```

onde prio indica o novo nível de prioridade do módulo.

4.1.3. Serviços de Temporização

São descritos aqui os serviços que suportam os procedimentos e funções pré-declaradas da LPM TIME, DELAY e TICK.

4.1.3.1. Leitura do Relógio do STR

A função pré-declarada TIME que permite saber o valor do relógio do STR em unidades de tempo e cuja interface é dada por:

```
FUNCTION TIME: LONGINTEGER;
```

é suportada pelo serviço relógio, cuja interface é dada por (v. ex. p. 123):

```
FUNCTION relógio: LONGINTEGER;
```

4.1.3.2. Retardamento de um Módulo

O procedimetno pré-declarado DELAY que suspende a execução do módulo requisitante por um período de tempo e cuja interface é dada por:

```
PROCEDURE DELAY (<período de tempo>: LONGINTEGER);
```

é suportado pelo serviço retarda cuja interface é dada por (v. ex. p. 123):

```
PROCEDURE retarda (tempo: LONGINTEGER);
```

onde tempo especifica o número de "ticks" correspondente ao <período de tempo> durante o qual o módulo requisitante deve ser retardado. A demanda de retarda com o parâmetro tempo igual a zero provoca apenas um reescalonamento.

4.1.3.3. Sinalização da Passagem de uma Unidade de Tempo ("TICK")

O STR não incorpora tratadores de interrupção para nenhum periférico, todos os tratadores de interrupção necessários devem ser implementados como módulos, inclusive o tratador das interrupções correspondentes aos pulsos do RTR. A fim de que este tratador possa sinalizar ao STR a passagem de uma unidade de tempo é

oferecido o serviço **tick** cuja interface é dada por:

```
PROCEDURE tick;
```

4.1.4. Interrupções

São descritos neste item os serviços que suportam a função **INTALLOC** e o procedimento **WAITIO**, ambos pré-declarados na LPM.

4.1.4.1. Mapeamento de um Elemento do Vetor Físico de Interrupções

A função pré-declarada **INTALLOC**, através da qual o núcleo mapeia um elemento do vetor físico de interrupções num elemento do vetor lógico de interrupções, cuja interface é dada por:

```
FUNCTION INTALLOC    (<endereço do vetor físico>:
                      ADDRESS): INTEGER;
```

é suportada pelo serviço **map_int** cuja interface é dada por:

```
FUNCTION map_int ( end_int : ADDRESS): INTEGER;
```

onde, **end_int** é o endereço de memória para onde será passado o controle quando a interrupção for atendida pelo processador; a função que implementa o serviço **map_int** retornará um número inteiro correspondente a um elemento do vetor lógico de interrupções mantido pelo STR. O módulo tratador de interrupções que requisitou o serviço deverá, sempre que quiser esperar pela ocorrência da interrupção envolvida no mapeamento, referenciar o elemento do valor lógico retornado pelo serviço **map_int**.

4.1.4.2. Espera pela Ocorrência de uma Interrupção

O procedimento pré-declarado **WAITIO**, através do qual um módulo tratador de interrupções aguarda a sua ocorrência, cuja interface é definida por:

```
PROCEDURE WAITIO (<vetor lógico>: INTEGER);
```

é suportado pelo serviço espera_int, cuja interface é dada por:

```
PROCEDURE espera_int (int_log: INTEGER);
```

onde int_log é o número do elemento do vetor lógico de interrupções associado à interrupção que o módulo requisitante quer asuardar.

4.2. Implementação dos Serviços do STR

Os serviços do STR são implementados por funções e procedimentos PASCAL, cuja interface foi definida na Seção 4.2, que compartilham as estruturas de dados que representam os objetos (módulos e portas) da aplicação. Além das rotinas que diretamente implementam os serviços, o STR incorpora outras que suportam as primeiras no serenciamento de memória e escalonamento preemptivo dos módulos.

4.2.1. Estruturas de Dados do STR

Neste item são apresentadas as principais estruturas de dados sobre os quais atuam os procedimentos do STR a fim de implementar os serviços oferecidos aos módulos de aplicação.

Conforme ilustra a Figura 4.1 cada módulo da aplicação é identificado por um número inteiro e descrito por um Bloco de controle de Módulo (BCM). A tabela de módulos reúne os apontadores aos BCMs dos módulos definidos.

Um BCM contém dados estáticos que descrevem o módulo que representa, bem como, dados dinâmicos que descrevem o seu estado num dado instante. Exemplos de dados estáticos relativos a um módulo são o seu nome, a quantidade de memória que é usada para heap e pilha, o apontador à sua base de dados, o valor dos seus parâmetros reais e a descrição de suas portas. Exemplos de dados dinâmicos sao o valor da prioridade do módulo e o tempo que ele deve ainda esperar para passar ao estado de pronto quando suspenso.

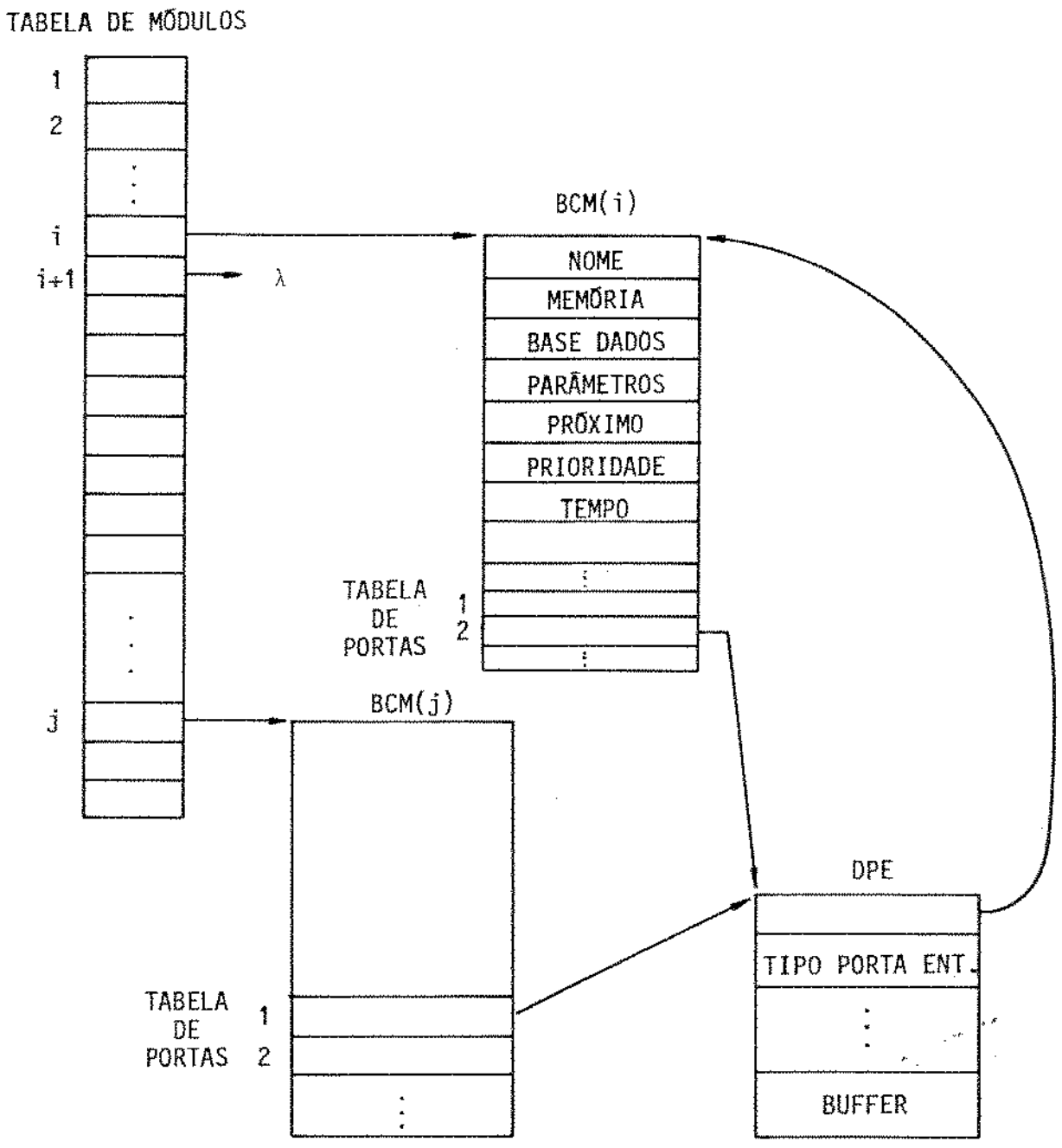

FIGURA 4.1 – A estrutura de dados básica do STR, ilustrando a conexão da porta de saída número 1, do módulo j, à porta de entrada número 2, do módulo i.

A tabela de portas de um BCM contém, em cada posição, um apontador ao descritor da porta de número correspondente à sua posição na tabela. Assim, na primeira posição da tabela de portas, há um apontador ao descritor da porta de número um, na segunda posição, há um apontador ao descritor da porta de número dois e assim por diante.

Se uma dada porta for de entrada, na posição que lhe corresponde na tabela de portas, haverá um apontador a um Descritor de Porta de Entrada (DPE) que representará a porta. Caso a porta seja de saída, na posição que lhe corresponde na tabela de portas, haverá um apontador ao DPE da porta de entrada à qual está conectada. Caso uma porta de entrada tenha um buffer, este será parte do seu DPE.

A Figura 4.1 mostra as estruturas descritas, ilustrando a conexão de uma porta de saída a uma porta de entrada. Cada DPE contém um apontador ao BCM do módulo a que pertence. Este fato será explorado pelos procedimentos que implementam os serviços de comunicação para determinar o estado de um módulo receptor. Na Figura 4.1 a porta de saída número 1, do módulo j, está conectada à porta de entrada número 2, do módulo i. Convém lembrar que os nomes lógicos dos módulos e de suas portas foram mapeados em identificadores inteiros durante a fase de compilação e configuração dos módulos da aplicação. Isto é feito para otimizar o acesso às estruturas de dados que representam essas entidades.

Caso uma porta de saída esteja conectada a várias de entrada (conexão multi-destino), não se aponta diretamente a um DPE e sim ao início de uma lista de apontadores a DPEs. A Figura 4.2 ilustra a conexão multi-destino da porta de saída número 2, do módulo K, às portas de entrada números 3 e 10, dos módulos j e l respectivamente.

Como mencionado anteriormente, as estruturas de dados do STR foram projetadas de maneira a minimizar o tempo envolvido na sua manipulação. Esta preocupação é muito importante pois favorece a implementação de serviços com tempos de resposta pequenos, característica muito importante em sistemas de tempo real. As listas do STR não fogem à regra, elas são definidas de maneira a permitir inserções e retiradas rápidas de nós nas suas extremidadas, o quê na maioria dos casos é suficiente já que as inserções no meio de uma lista são muito raras no STR. A Figura 4.3 apresenta a estrutura de uma lista onde Início e Fim são os apontadores às extremidades da lista.

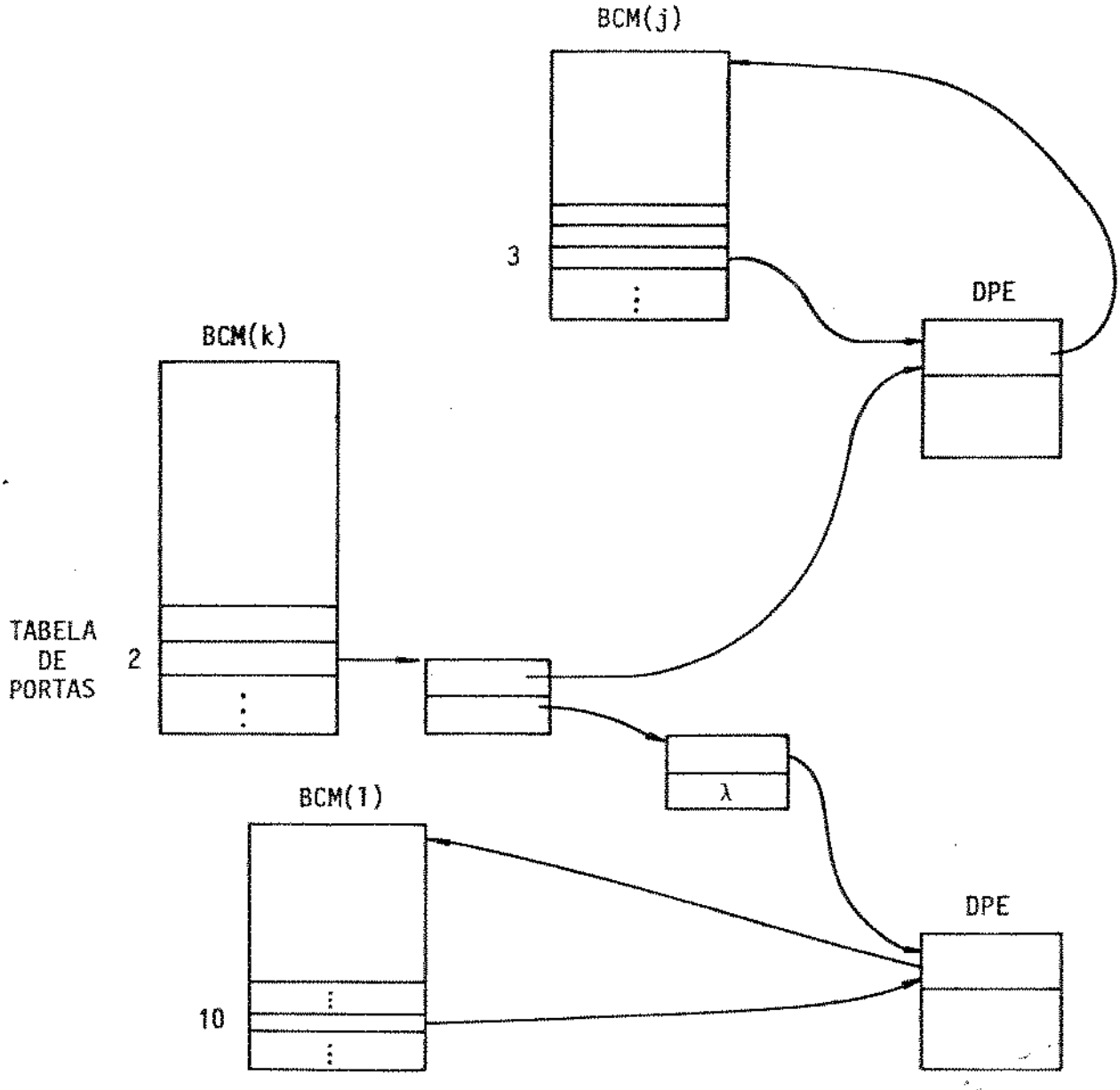

FIGURA 4.2 - Conexão multi-destino da porta de saída número 2, do módulo k, às portas de entrada de números 3 e 10, dos módulos j e l respectivamente

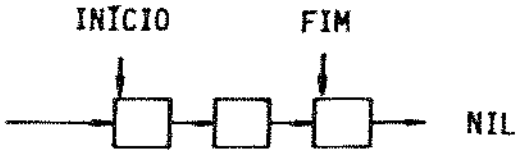

FIGURA 4.3 – Estrutura de uma lista ligada do STR

4.2.2. Reescalonamento de Módulos

A política de escalonamento de módulos empregada no STR é a preempção por prioridade. Nesta política o controle sobre o processador é dado sempre ao módulo de maior prioridade no estado pronto para executar. A preempção por prioridade é a política mais indicada às aplicações em tempo real, pois assegura um pronto atendimento dos módulos tratadores de situações urgentes.

O reescalonador atua basicamente sobre o apontador rodando e sobre a fila de tarefas prontas para executar (ou fila de pronto). O apontador rodando indica o BCM do módulo em execução. A fila de pronto é subdividida em diversas listas, uma para cada nível de prioridade existente. A Figura 4.4 mostra a fila de pronto com os cinco níveis de prioridade atulmente definidos no STR.

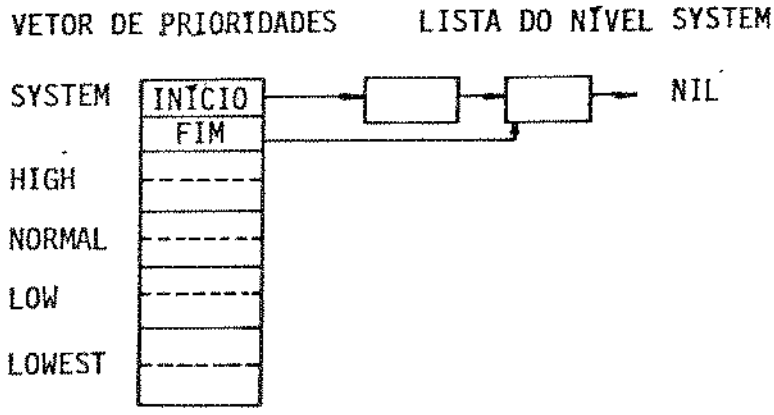

FIGURA 4.4 – A fila de pronto

Quando um módulo passa para o estado pronto seu BCM é inserido no final da lista que representa o seu nível de prioridade. O reescalonador quando tem que escolher um módulo para executar, sempre seleciona aquele cuja BCM está no início da lista de maior prioridade não vazia. Com esta organização da fila de pronto a inserção de um BCM é muito simples e rápida, já que é feita no final de uma lista de prioridade, cujo apontador fim é acessado diretamente no vetor de prioridades da fila de pronto, usando como índice a prioridade do módulo a inserir.

O reescalonador é composto dos procedimentos reescalona, executa, pronto e pronto_reesc.

O procedimento reescalona retira da fila de pronto o BCM no início da lista de maior prioridade não vazia e chama o procedimento executa. O procedimento executa salva o conteúdo dos registradores que apontam ao topo da pilha no BCM do módulo que vai perder o controle e recarrega esses registradores com os valores que apontam ao topo da pilha do módulo que vai receber o controle e que haviam anteriormente sido salvos no BCM deste.

O procedimento pronto insere um BCM no final da lista de pronto de sua prioridade. O procedimento pronto_reesc faz o mesmo e no caso do módulo recém inserido possuir uma prioridade superior à do módulo em execução chama o procedimento reescalona que lhe passará o controle.

A inserção de um BCM em outras listas é feita diretamente pelos procedimentos que implementam os serviços associados. Nesses casos o módulo requisitante do serviço é colocado num estado de espera pela ocorrência de um evento associado à lista onde seu BCM foi inserido.

4.2.3. Gerenciamento de Memória

Os BCMs e DPEs associados a uma dada aplicação são alocados dinamicamente durante o processo de inicialização das estruturas do STR. Essas estruturas de dados têm tamanhos variáveis em função do módulo ou porta que representam. Assim, um módulo com 10 portas tem um BCM de tamanho menor que um módulo com 50 portas, da mesma forma que o DPE de uma porta com um "buffer" de 50 bytes tem um tamanho menor que o DPE de uma porta com buffer de 200 bytes. Evidentemente poderiam ser definidos BCMs e DPEs de tamanho máximo que seriam alocados a todos os módulos e portas de entrada independentemente de suas características. Embora essa estratégia facilitasse bastante a alocação dinâmica de memória, que poderia inclusive ser feita mediante o uso do procedimento NEW do PASCAL, o seu emprego geraria um considerável desperdício de memória que se preferiu evitar. Implementou-se, em consequência, um alocador

de memória muito simples que usa a memória em função das necessidades do procedimento requisitante. A sua interface tem a forma:

```
FUNCTION aloc_mem (tamanho: INTEGER): ADDRESS;
```

O gerenciador de memória implementado não permite que se faça a dealocação de memória, o que por ora não é necessário. No caso de o STR vir a suportar a reconfiguração dinâmica da aplicação será necessário melhorar o gerenciador em uso de forma que este possa fazer a dealocação dinâmica de memória. Isto será possível mediante a incorporação ao gerenciador de algum dos algoritmos clássicos empregados para esse fim como por exemplo o best_fit ou o first_fit [MAD 74].

4.2.4. Troca de Mensagens

A implementação dos serviços para troca de mensagens será discutida na mesma sequência em que foram apresentadas no item 4.1.1.

4.2.4.1. Envio Assíncrono de uma Mensagem

O tratamento a dar à requisição do envio assíncrono de uma mensagem a uma porta de saída dependerá do estado do módulo que contém a porta de entrada à qual a mensagem se destina. O estado do módulo receptor é acessível através do apontador ao seu BCM existente no DPE da porta destino. Se o módulo receptor estiver esperando pela mensagem que está sendo enviada, a mesma será copiada diretamente no endereço onde é aguardada. Esse endereço encontra-se no DPE da porta receptora e corresponde ao parâmetro end_mem passado pelo módulo ao STR quando da requisição da recepção da mensagem. Uma vez copiada a mensagem, o módulo receptor será colocado no estado pronto através da rotina pronto_reesc. Observe-se que caso o módulo receptor tenha prioridade superior à do módulo emissor ele ganhará o controle como consequência do envio da mensagem. Por outro lado, se o módulo destinatário não estiver aguardando a mensagem esta será copiada no "buffer" da porta de entrada correspondente e o controle retornará ao módulo emissor. Numa comunicação assíncrona a porta de entrada tem sempre associado um "buffer" para no mínimo uma mensagem.

4.2.4.2. Envio de uma Mensagem Síncrona

A maneira de transferir uma mensagem síncrona sendo enviada depende do estado do módulo receptor. Caso ele esteja à espera da mensagem sendo enviada, esta será copiada diretamente para o endereço onde é aguardada, num processo análogo ao de um envio assíncrono. No caso de o módulo receptor não estar à espera da mensagem, o BCM do módulo emissor será colocado numa fila de espera, associada à porta destino, até que o módulo destino manifeste o seu interesse em receber uma mensagem dessa porta e todos os módulos que aguardavam na fila há mais tempo já tenham completado a sua transação. Seja qual for o caso, i.e., módulo destino à espera ou não, após o envio de uma mensagem síncrona sempre ocorre um reescalonamento pois o módulo emissor deve aguardar a resposta à mensagem que enviou.

4.2.4.3. Envio Síncrono de uma Mensagem com Cláusula para Tratamento de Falha

O envio síncrono e temporizado de uma mensagem é análogo ao envio síncrono simples; a única diferença é que o módulo emissor não ficará esperando indefinidamente a mensagem de resposta. Caso essa mensagem demore mais que um período de tempo especificado, ele será colocado no estado de pronto e receberá um código indicativo do término do período de espera pela mensagem.

Antes de chamar o reescalonador o procedimento que implementa o serviço de envio síncrono e temporizado indica, num campo específico do BCM do módulo emissor, o tempo durante o qual a mensagem deve ser aguardada. Este campo é testado periodocamente pelo serenciador de tempo do STR que colocará o módulo no estado de pronto assim que o período de espera tiver expirado. Caso uma mensagem de resposta seja enviada antes do fim do período de espera o próprio serviço encarregado do envio da resposta colocará o módulo destinatário no estado pronto.

4.2.4.4. Recepção Bloqueante de uma Mensagem

A implementação do serviço de recepção bloqueante de uma mensagem através de uma porta de entrada se dá da seguinte maneira: Se na porta de entrada não houver pelo menos uma mensagem disponível, o endereço onde uma mensagem deverá ser recebida será colocado no seu DPE, no BCM do módulo receptor será indicado que ele se encontra em estado de espera por uma mensagem, sendo a seguir chamado o reescalonador. Caso haja uma mensagem disponível

ela será copiada para o endereço desejado pelo módulo requisitante que receberá o controle em seguida.

A maneira como se dá a cópia de uma mensagem disponível na entrada de uma porta depende do tipo dessa porta. Se ela for síncrona, o seu DPE apontará a uma lista de BCMs de módulos que estão esperando que a mensagem que enviaram seja recebida e respondida. Nesse caso o BCM do módulo que esperou mais tempo será retirado da lista e a mensagem que enviou será copiada para o endereço especificado pelo módulo receptor. No DPE da porta receptora será armazenada a identidade do módulo emissor a fim de assegurar que a mensagem de resposta será corretamente endereçada. No caso de a porta de entrada receptora ser assíncrona, as mensagens recebidas estarão disponíveis no "buffer" de entrada da porta, de onde será copiada a mais antiga.

4.2.4.5. Envio de uma Mensagem de Resposta

Quando chamado, o procedimento que implementa o envio de uma mensagem de resposta verificará se anteriormente foi recebida uma mensagem pela porta de entrada especificada e se essa mensagem ainda não foi respondida. Essa verificação é feita consultando o identificador do módulo de onde foi recebida a última mensagem e o número da comunicação síncrona corrente, ambos no DPE da porta de entrada. Uma vez confirmado que a mensagem de resposta pode ser enviada, ela é copiada para o endereço onde é aguardada no módulo que iniciou a comunicação síncrona, que será nesse instante colocado no estado de pronto. O envio bem sucedido de uma mensagem de resposta encerra uma comunicação síncrona.

4.2.4.6. Desvio de uma Comunicação Síncrona

O desvio de uma comunicação síncrona é implementado reenviando, inalterada, a mensagem recebida por uma porta de entrada síncrona para uma porta de saída síncrona, do mesmo tipo, especificada no comando **FORWARD**. No DPE da porta de entrada conectada à porta de saída para onde foi feito o reenvio é identificado como módulo emissor da mensagem aquele que enviou a mensagem ao módulo que contém o **FORWARD**. Em consequência, do ponto de vista da comunicação lógica estabelecida entre o módulo que deu origem à mensagem e aquele que efetivamente a recebeu, tudo se passa como se o módulo, ou módulos, que efetuaram os desvios não existissem.

4.2.4.7. Recepção Seletiva de uma Mensagem

Como tratado anteriormente, a execução de um comando SELECT traduz-se na requisição dos serviços ini_rec_sel, ini_temp_sel e sel.

O serviço ini_rec_sel armazena os parâmetros indice_for, cláusula e end_mem no DPE da porta de entrada sendo preparada. Em seguida coloca esse DPE numa lista de espera por recepção seletiva associada ao BCM do módulo receptor.

O serviço ini_temp_sel armazena o parâmetro tempo num campo do BCM do módulo receptor. A exemplo do que é feito na implementação do envio síncrono e temporizado de uma mensagem, esse campo será testado periodicamente pelo gerenciador de tempo do STR, que colocará o módulo no estado de pronto se o tempo de espera expirar sem que nenhuma mensagem tenha sido recebida.

O serviço sel('P',...) verifica inicialmente se alguma das portas colocadas na lista de recepção seletiva já recebeu uma mensagem. A verificação é feita sequencialmene começando pelo DPE da porta correspondente à primeira cláusula de recepção do PSELECT e que consequentemene foi traduzida no primeiro ini_rec_sel do comando. A primeira porta encontrada com uma mensagem será a escolhida, sendo a mensagem copiada usando um procedimento análogo ao empregado na recepção bloqueante. Além de copiar a mensagem recebida ini_rec_sel deve informar ao módulo o número da porta por onde foi recebida a mensagem, o valor da variável de controle do FOR no qual a cláusula de recepção seletiva foi feita e as demais informações especificadas na interface do serviço. Essas informações são retiradas do DPE da porta escolhida, onde tinham sido armazenadas pelo serviço ini_rec_sel correspondente.

Caso não haja nenhuma mensagem disponível sel_pri verificará se o parâmetro cláusula_else de sua interface é diferente de zero; se for será retornado o inteiro associado à clausula ELSE do comando a fim de que esta seja executada.

Se nenhuma das portas colocadas na lista de recepção seletiva tiver recebido uma mensagem e o comando PSELECT não incluir uma cláusula ELSE, o módulo será suspenso até a chegada de uma mensagem a alguma das portas, ou até que o tempo de espera especificado através do serviço ini_temp_sel expire.

A execução do serviço sel('R',...) diferencia-se da execução de sel('P',...) apenas quando no momento de sua execução já houver mensagens disponíveis nas portas de entrada. Nesse caso, sel('R',...) seleciona aleatoriamente uma das portas com mensagens disponíveis. Esta escolha é um pouco mais onerosa do que aquela feita por sel('P',...) pois, para fazê-la sel('R',...) deve varrer toda a lista de portas em espera seletiva contando quantas já receberam uma mensagem, a fim de serar um número alea-

tório que dirá qual delas será selecionada. O número aleatório é obtido usando o serador aditivo:

$$X_n = (X_{n-24} + X_{n-55}) \bmod m, \quad n >= 55 \qquad [KNU\ 81]$$

4.2.4.8. Cancelamento de uma Comunicação Síncrona

A implementação deste serviço é semelhante ao do envio de uma mensagem de resposta normal, porém neste caso em vez de enviar uma mensagem de resposta é enviado um código indicativo do motivo do cancelamento. Esse código será colocado no BCM do módulo que iniciou a comunicação síncrona e corresponderá ao parâmetro recebido na interface do serviço **resp_falha**.

4.2.4.9. Determinação da Razão de uma Falha

O serviço **motivo_falha** retorna o código do erro ocorrido na comunicação síncrona cujo comando **FAIL** está sendo executado. Esse valor foi armazenado pelos procedimentos que implementam a comunicação síncrona no BCM do módulo em execução.

4.2.4.10. Teste de Conexão de uma Porta de Saída

O serviço **conectado** verifica na tabela de portas do módulo se o apontador correspondente à porta de saída que se deseja saber se está conectada é igual a **NIL** ou não. Caso seja, a porta não está conectada e é retornado um valor **FALSE**.

4.2.4.11. Determinação da Quantidade de Mensagens Enfileiradas numa Porta de Entrada

Se a porta de entrada for "bufereada", o serviço **men_prt_ent** retorna o contador de mensagens armazenadas nesse "buffer". Caso a porta de entrada não seja "bufereada", i.e., seja uma porta para comunicações síncronas, **men_prt_ent** conta o número de nós na

lista que contém os BCMs dos módulos que iniciaram uma comunicação síncrona com essa porta.

4.2.5. Mudança da Prioridade de um Módulo

O serviço muda_prio armazena o valor do novo nível de prioridade no BCM do módulo requisitante, que é o módulo em execução, e chama o reescalonador. O reescalonador insere o módulo em execução na fila de pronto e passa o controle ao módulo de maior prioridade nessa fila, que pode ou não ser aquele que solicitou a mudança de prioridade.

4.2.6. Serviços de Temporização

Será discutida aqui a implementação dos serviços de temporização apresentados na seção 4.1.2.

4.2.6.1. Leitura do Relógio do STR

O STR mantém um contador de tempo usado para detectar o fim dos períodos de espera dos módulos decorrentes do uso do procedimento DELAY ou da execução de comunicações com cláusulas de temporização. O valor desse contador pode ser acessado pelos módulos da aplicação através do serviço relógio.

4.2.6.2. Retardamento de um Módulo

O serviço retarda coloca o BCM do módulo requisitante numa fila de espera até a passagem do período de tempo especificado. A Figura 4.5 ilustra um possível estado da fila de retardo onde o módulo i deve esperar 10 unidades de tempo e os módulos j, k e l 20 antes de passarem para o estado de pronto. Como se pode observar na figura, o tempo de espera de um modulo corresponde à soma do campo tempo do seu BCM aos campos tempo dos BCMs que o antecedem na fila.

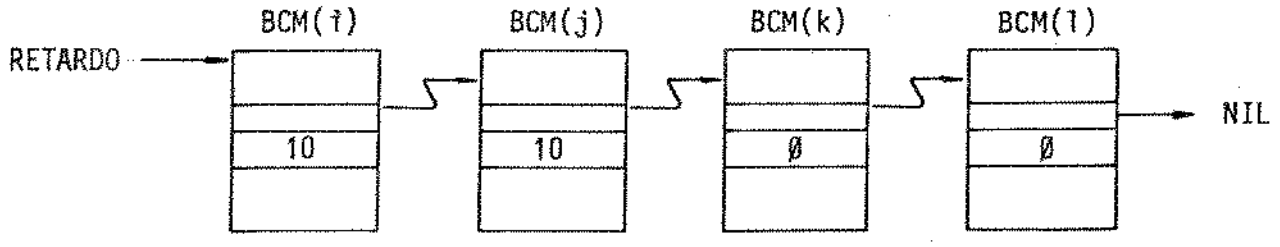

FIGURA 4.5 - A fila de retardo. O módulo i deve esperar 10 unidades de tempo e os módulos j, k e l 20 unidades antes de passarem para o estado de pronto

A inserção de um BCM na fila de retardo não é tão simples como em outras filas do STR onde as inserções são sempre feitas numa das extremidades. Entretanto, com esta organização a cada unidade de tempo decorrida é necessário decrementar apenas o campo **tempo** do BCM no início da lista, que ao chegar a zero idicará que o módulo que representa, bem como os demais à sua diretia na fila cujo campo **tempo** for zero, podem passar para o estado de pronto.

Se o serviço **retarda** for requisitado com o parâmetro **tempo** igual a zero, o BCM do módulo requisitante será colocado no fim da lista de pronto correspondente à sua prioridade e será feito um reescalonamento.

4.2.6.3. Sinalização da Passagem de uma Unidade de Tempo

O serviço **tick** é chamado pelo tratador das interrupções do relógio de tempo real cada vez que decorrer uma unidade de tempo (ou tick) do sistema. Observe-se que a duração da unidade de tempo do sistema é determinada pelo tratador das interrupções do STR e não necessariamente corresponde ao intervalo de tempo existente entre dois pulsos do relógio.

O serviço **tick** se encarrega de verificar se já expiraram os tempos de espera dos módulos envolvidos em comunicações temporizadas e de decrementar o campo **tempo** do BCM no início da fila de retardo. Os BCMs dos módulos cujos períodos de espera por mensagens tiverem terminado serão colocados na fila de pronto. Da mesma forma, se o campo **tempo** do BCM no início da fila de retardo tornar-se zero ele e todos os seus sucessores com esse campo igual a zero também serão colocados na fila de pronto.

4.2.7. Interrupções

Discute-se neste item a implementação dos serviços que suportam o tratamento lógico das interrupções apresentadas no item 4.1.3.

4.2.7.1. Mapeamento de um Elemento do Vetor Físico de Interrupções

O mapeamento de uma interrupção física em uma interrupção lógica feito pelo serviço map_int é mostrado na Figura 4.6.

Na figura supõe-se que o módulo n cuja estrutura é a seguinte:

```
MODULE n
VAR int:INTEGER;
BEGIN
      int:=INTALLOC(ICH)
      .
      .
      .
      WAITIO(int)
      .
      .
      .
END;
```

solicita um mapeamento num instante em que nenhuma interrupção lógica foi ainda alocada. Nesse caso a interrupção lógica número um será alocada ao módulo. Isto é feito colocando no descritor desta interrupção lógica um apontador ao BCM do módulo n e chamando o procedimento inivet que coloca, no endereço para onde é desviado o controle quando do atendimento da interrupção física correspondente (ICH), uma sequência de instruções que sinaliza a ocorrência da interrupção lógica de número 1. Esta sequência de instruções deve chamar o procedimento INTLOG passando como parâmetro o número da interrupção lógica associada à interrupção física ocorrida. INTLOG registrará a ocorrência da interrupção no descritor da interrupção lógica caso o módulo tratador não esteja à sua espera ou colocará o módulo no estado de pronto em caso contrário. O procedimento inivet é bastante dependente da arquitetura do hardware onde o núcleo é implementado.

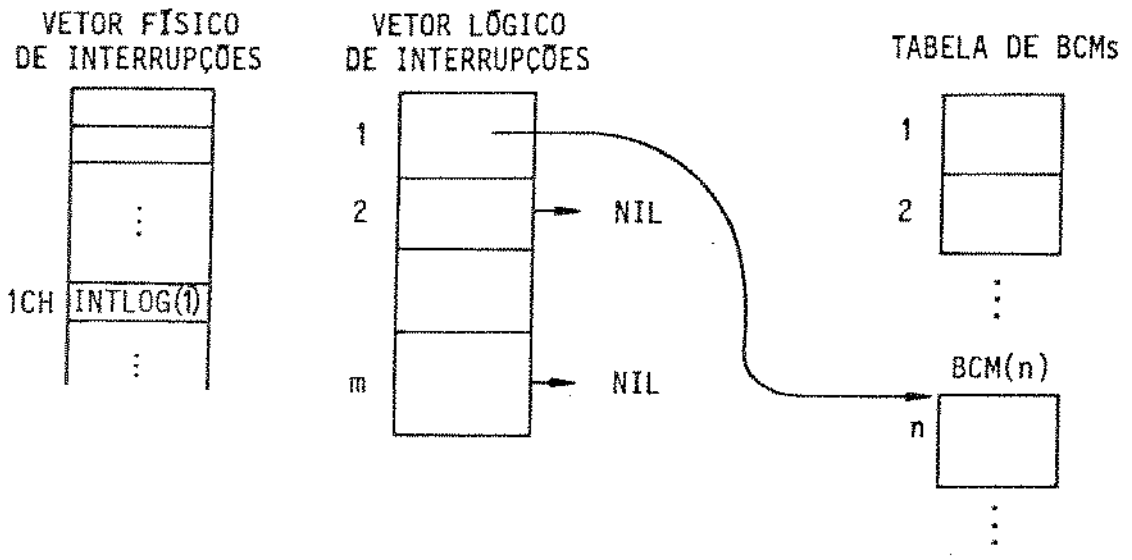

FIGURA 4.6 - O tratamento lógico de interrupções. A interrupção lógica numero 1 foi alocada ao módulo n

4.2.7.2. Espera pela Ocorrência de uma Interrupção

O serviço **espera_int** verifica no descritor da interrupção lógica especificada se ela já ocorreu; em caso afirmativo o controle retorna ao módulo requisitante. Caso a interrupção não tenha ocorrido, o módulo será colocado em estado de espera pela interrupção, o que provocará um reescalonamento.

4.3. Duas Arquiteturas para a Implantação do STR

Esta seção discute as duas arquiteturas analisadas para a implantação do STR num equipamento compatível com o microcomputador IBM-PC usando como ferramentas básicas de desenvolvimento um compilador PASCAL, um macroassembler para o 8086/8088, um serenciador de biblioteca e um ligador.

Na primeira arquitetura de implantação considerada o processo de compilação e configuração dos módulos de uma aplicação resulta num único programa PASCAL. Nesse programa cada módulo é representado por um procedimento. Esses módulos/procedimentos interagem através da chamada de outros procedimentos e funções que implementam os serviços do STR. As variáveis globais desse programa correspondem as estruturas de dados do STR, sobre os quais

operam os procedimentos e funções que implementam os serviços. O código de inicialização do STR corresponde ao programa principal que cria o contexto multi-tarefa dos módulos da aplicação. A criação desse contexto inclui a alocação de área para uma pilha independente para cada módulo. O principal requisito a ser atendido pelo compilador PASCAL empregado na geração do código deste programa é que as rotinas da sua biblioteca sejam reentrantes, já que elas serão compartilhadas pelos diversos módulos/procedimentos do programa concorrente. Do ponto de vista dos procedimentos e funções que implementam os serviços do STR esse requisito é automaticamente satisfeito pois, os procedimentos e funções do PASCAL padrão são recursivos e em conseqüência reentrantes.

A principal característica da segunda arquitetura considerada é a independência dos espaços de endereçamento dos módulos entre si e destes e do STR.

Nesta arquitetura cada módulo é traduzido num programa PASCAL que será gerado e carregado separadamente dos demais. Nesta arquitetura há uma distinção bem clara entre as fases de geração do código executável dos módulos e a fase de configuração de uma aplicação. Da primeira fase resulta um código independente das características da aplicação onde cada módulo será executado.

Da fase de configuração da aplicação resulta uma tabela que define o relacionamento existente entre os módulos componentes bem como as características que particularizam cada um deles.

A partir de uma tabela de configuração que lhe é fornecida o STR inicializa suas estruturas de dados após o que carrega e inicia a execução da aplicação.

Um estudo comparativo das duas arquiteturas apresentadas mostrará que a primeira é superior à segunda na economia de memória, já que as rotinas da biblioteca PASCAL são compartilhadas entre os diversos módulos/procedimentos da aplicação e do STR. Entretanto a existência de um único espaço de endereçamento para todos os módulos da aplicação e para o STR pode conduzir a resultados inesperados devidos à interferência, acidental ou não, de um módulo/procedimento sobre o espaço de endereçamento dos demais. Adicionalmente a existencia de um único espaço de endereçamento pode inviabilizar a futura incorporação ao STR de mecanismos para reconfiguração dinâmica.

Pelo exposto e pela maior coerencia da segunda arquitetura com o ambiente proposto onde a modularidade (e em conseqüência a independencia entre componentes) é o aspecto fundamental, a segunda arquitetura apresentada foi a escolhida para a implantação desta primeira versão do STR. Esta implantação será descrita no capitulo que se segue.

CAPÍTULO 5

ARQUITETURA DA IMPLANTAÇÃO DO SUPORTE DE TEMPO REAL

5. ARQUITETURA DA IMPLANTAÇÃO DO SUPORTE DE TEMPO REAL

O equipamento escolhido para implantação da primeira versão do ambiente apresentado é uma máquina compatível com o IBM-PC. A razão dessa escolha deveu-se à grande popularidade dos compatíveis PC de uma maneira geral, como também, a sua aceitação em larga escala nos ambientes industriais. Como para o IBM-PC é oferecido apenas um conjunto limitado de ferramentas de software orientadas às aplicações industriais com características de tempo real, a implementação das linguagens LPM e LCM neste equipamento fornece uma opção para o desenvolvimento de aplicativos voltados ao controle/supervisão de processos.

A arquitetura definida para a implantação do suporte de tempo real procura privilegiar uma situação na qual os módulos do usuário sejam independentes entre si da mesma forma que estes também sejam independentes do STR. Esta independência implica fundamentalmente na separação explícita dos espaços de endereçamento dos módulos entre si e entre o STR, não permitindo que nenhum módulo interfira diretamente com um outro ou com o STR. Esta estrutura é a mais coerente com um ambiente no qual os módulos comunicam-se somente através da troca de mensagens, assim como, adequa-se mais claramente às possibilidades futuras de reconfiguração dinâmica do sistema. Definida a arquitetura explicitou-se ainda como requisito de implantação a possibilidade de utilização por parte dos módulos do usuário dos serviços disponíveis no MS-DOS.

Este capítulo está estruturado da seguinte maneira: A Seção 5.1 descreve a geração de um programa de aplicação que será suportado pelo STR. A Seção 5.2 apresenta de maneira sucinta a arquitetura do microprocessador 8088, no qual é baseado o equipamento usado na implementação do STR. A Seção 5.3 apresenta o mapa de memória do computador quando da execução de uma aplicação discutindo os seus componentes. A Seção 5.4 descreve a carga do STR e a inicialização das suas estruturas de dados a partir da tabela de configuração da aplicação. A Seção 5.5 descreve o procedimento empregado para a carga e início da execução da aplicação. A Seção 5.6 discute o casamento entre os sistemas lógico e físico de interrupções. Finalmente na Seção 5.7 são apresentados alguns dados relativos ao desempenho desta versão do STR.

5.1. Geração de um Programa de Aplicação

A geração de um programa de aplicação compreende duas atividades principais: a primeira consiste na compilação independente dos módulos e a segunda na configuração do programa de aplicação (ou simplesmente aplicação).

5.1.1. Compilação de um Módulo

O processo de compilação de um módulo é descrito a seguir e ilustrado na Figura 5.1. O programa fonte do módulo escrito pelo usuário (M1.LPM) é inicialmente submetido ao pré-processador LPM (ou simplesmente LPM). O pré-processador converte o programa fonte em LPM num programa fonte em linguagem PASCAL (M1.PAS) e numa estrutura de dados chamada qualificador do módulo (M1.QLF). As extensões à linguagem PASCAL tratadas pelo pré-processador LPM podem ser divididas em dois grupos: executáveis e não executáveis.

As extensões ditas executáveis são compostas basicamente pelas primitivas de comunicação (SEND, RECEIVE, SELECT...), pela função de temporização (DELAY) e pelos procedimentos para o tratamento de interrupções (INTALLOC e WAITIO). As extensões executáveis são traduzidas pelo pré-processador LPM em chamadas a serviços do STR, representados como procedimentos no programa PASCAL resultante da pré-compilação. As extensões não executáveis são compostas pelas construções de definição de portas de comunicação e de mensagens. Estas construções são usadas para fazer testes de consistência durante o pré-processamento e para a geração do qualificador do módulo. Os testes de consistência visam assegurar o uso correto das portas de comunicação, verificando se as mensagens e as portas referenciadas nas primitivas de comunicação são do mesmo tipo.

O qualificador de um módulo descreve as suas portas e será usado durante a configuração das aplicações que utilizem esse módulo. O código objeto resultante da compilação do programa PASCAL gerado pelo pré-processador LPM será submetido ao ligador que gerará o código objeto executável do módulo. Conforme ilustrado na Figura 5.1 ao código objeto do módulo são ligados vários outros arquivos objeto a fim de produzir o código executável do módulo. O arquivo *.OBJ na figura representa um número qualquer de rotinas escritas em outras linguagens (FORTRAN, linguagem de montagem, etc.) e declaradas como externas no módulo em LPM (M1.LPM). PASCAL.LIB é a biblioteca PASCAL, de onde o ligador carregará as rotinas necessárias à resolução das referências existentes no código objeto como consequência da compilação PASCAL. INIM.OBJ contém a rotina responsável por uma série de inicializações que antecedem à execução do módulo e ISTR.OBJ contém as rotinas de interfaceamento do módulo ao STR. INIM e ISTR serão melhor descritos nos itens posteriores. Do processo de ligação resulta o código objeto executável e relocável (M1.EXE) do módulo LPM (M1.LPM).

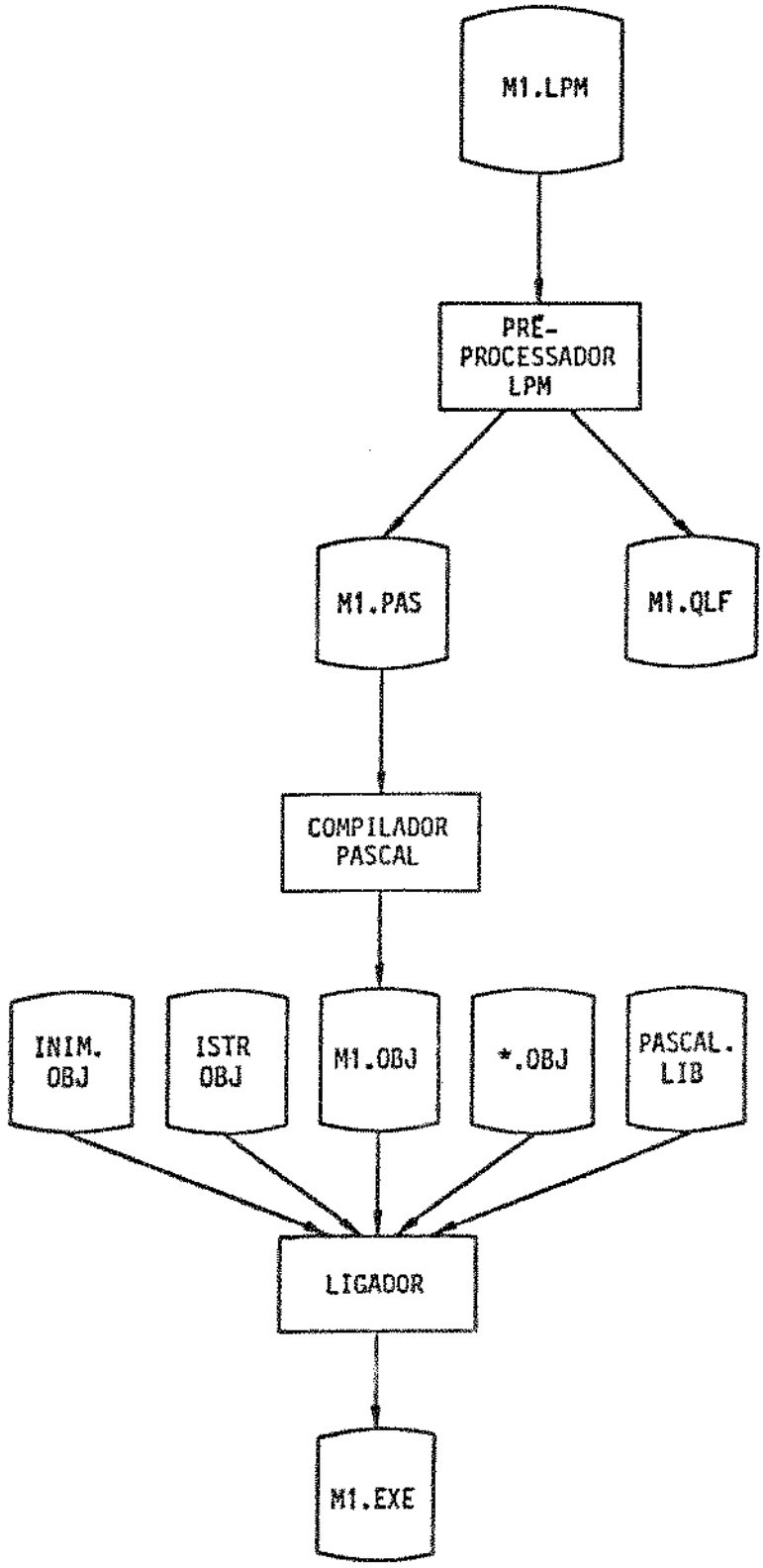

FIGURA 5.1 - A compilação de um módulo

5.1.2. Configuração de uma Aplicação

A configuração de uma aplicação é descrita a seguir e ilustrada na Figura 5.2. A etapa de configuração de uma aplicação pressupõe que todos os módulos envolvidos já foram previamente compilados conforme descrito anteriormente.

Uma configuração é especificada através de um programa fonte em linguagem LCM (APLICAÇÃO.LCM na Figura 5.2). A partir desse programa fonte e dos qualificadores dos módulos envolvidos o processador da LCM cria uma tabela (APLICAÇÃO.CNF) que representa a configuração especificada. No programa de configuração são relacionados os módulos que deverão compor a aplicação, descritas as características particulares que cada um apresentará e estabelecido o relacionamento existente entre eles via conexão de suas portas.

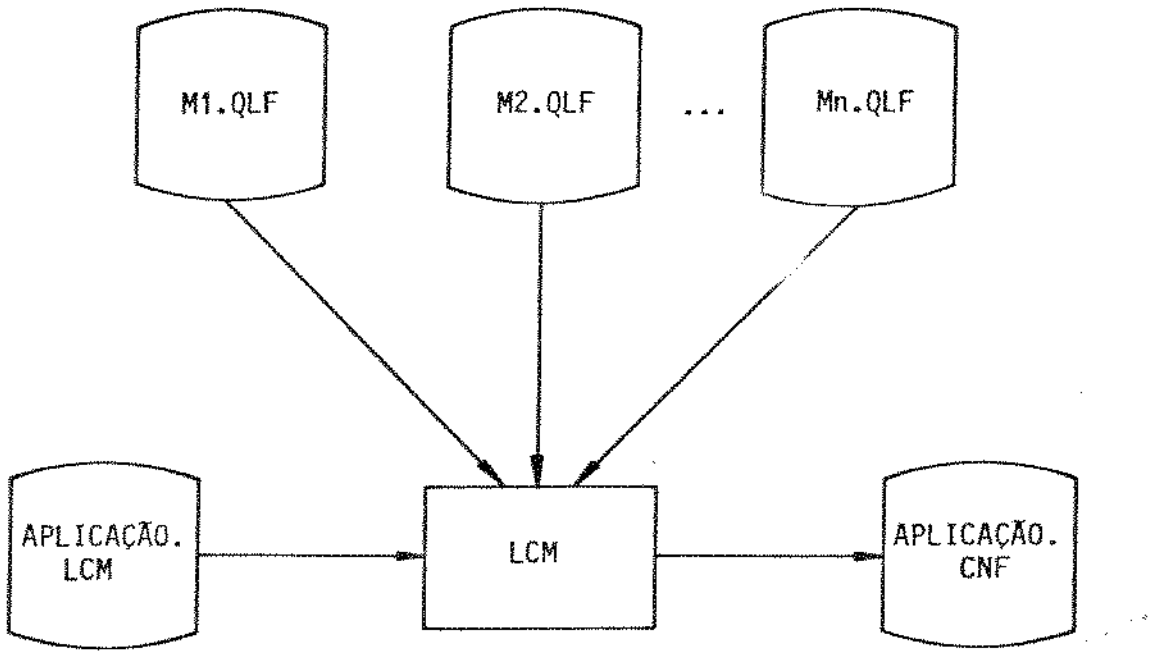

FIGURA 5.2 - A configuração de uma aplicação

A particularização de cada módulo compreende a determinação do valor de seus parâmetros reais, da quantidade de memória que lhe deverá ser alocado (para heap e pilha) e a sua prioridade inicial.

A efetivação das conexões especificadas pressupõe a sua validação pela LCM. Uma conexão será válida quando feita entre uma porta de saída e uma porta de entrada do mesmo tipo. Para verificar se uma dada conexão é válida a LCM utiliza as informações relativas às portas envolvidas, disponíveis nos qualificadores dos módulos.

A tabela de configuração gerada pelo processador LCM será passada ao STR quando se desejar executar a aplicação. A partir dessa tabela o STR inicializará as suas estruturas de dados e carregará a aplicação a fim de executá-la, processo este descrito nas seções 5.4 e 5.5.

5.2. Arquitetura do 8088

O microprocessador INTEL 8088 é o coração do hardware usado na implantação do sistema descrito neste trabalho, sendo por esse motivo necessário conhecê-lo um pouco mais.

O 8088 é uma versão do 8086 com barramento de 8 bits, tendo ambos conjuntos compatíveis de instruções. Embora o 8088 não tenha a elegância ou a potência de outros microprocessadores como o MOTOROLA 68000, ele faz parte de uma família numerosa e bastante poderosa.

Um dos membros mais importantes da família do 8088 é o processador aritmético 8087. Usando o 8087 as operações sobre números representados em ponto flutuante ou em BCD podem ser feitas de 10 a 100 vezes mais rapidamente do que quando efetuadas por software. Por exemplo, operando a uma frequência de 5 MHZ o 8087 calcula uma raiz quadrada de um número em ponto flutuante de 80 bits em aproximadamente 37 microsegundos, num desempenho comparável ao de máquinas muito maiores.

5.2.1. Microprocessador 8088

Para os propósitos deste trabalho o 8088 pode ser visto como um simples conjunto de registradores, cujos conteúdos podem ser modificados de diversas maneiras pelo conjunto de instruções do processador. Conforme pode ser observado na Figura 5.3, o 8088 possui catorze registradores de 16 bits. Como ilustra a figura, eles são divididos em quatro diferentes grupos funcionais:

a/ De dados;

b/ Apontadores e indexadores;

c) De sesmentos;

d) De controle.

Vários resistradores têm funções especiais quando usados com certas instruções, embora eles possam também ser usados como registradores de propósito geral.

Os registradores do grupo de dados são os mais flexíveis e geralmente os mais usados. Cada um dos quatro registradores desse grupo pode ser usado como um único registrador de 16 bits ou como um par de registradores de 8 bits. Por exemplo, pode-se carregar o registrador de 16 bits AX com o número 0008H usando a instrução MOV AX, 0008H ou usando as instruções MOV AH, 00H e MOV AL, 08H. O "H" e o "L" nas instruções MOV indicam que se está fazendo uma referência à parte alta ("High") ou à parte baixa ("Low") do registrador de 16 bits AX. O registrador AX ou acumulador diferencia-se dos demais pelo fato de que muitas instruções são executadas com mais rapidez ou tem um código em linguasem de máquina menor quando ele contém o operando da instrução.

O registrador BX é também chamado de registrador base por ser o único registrador de dados que pode ser usado com o modo indireto-indexado de endereçamento. O registrador CX é geralmente usado como contador em determinadas instruções repetitivas. O único uso especial do registrador DX é poder conter o endereço de uma porta (não confundir com porta lógica de comunicacões) em instruções de entrada e saída.

O segundo grupo é composto de 4 registradores de 16 bits: BP, SP, SI e DI. Esses registradores não podem ser acessados como pares de 8 bits, podendo porém ser utilizados em operações aritméticas e lógicas. Os dois registradores apontadores (SP e BP) são normalmente usados para manipular estruturas de dados do tipo pilha. Os registradores indexadores SI e DI são usados principalmente com instruções que manipulam cadeias de bytes.

Os registradores do grupo de controle são essenciais; este grupo é o formado pelo apontador de instruções (IP) e pelo registrador de condições ou flags. O registrador IP aponta sempre para a próxima instrução a ser executada.

O registrador de flags é bastante particular, ele é uma coleção de bits de condição (ou flags). Os flags refletem certas condições resultantes da execução de instruções lógicas e aritméticas, bem como, indicam modos particulares de execução do 8088. Por exemplo, o flag zero (ZF) será ligado, isto é, receberá o valor 1, se uma operação lógica ou aritmética produzir como resultado o valor zero, caso contrário ele será desligado, ou seja, receberá o valor 0.

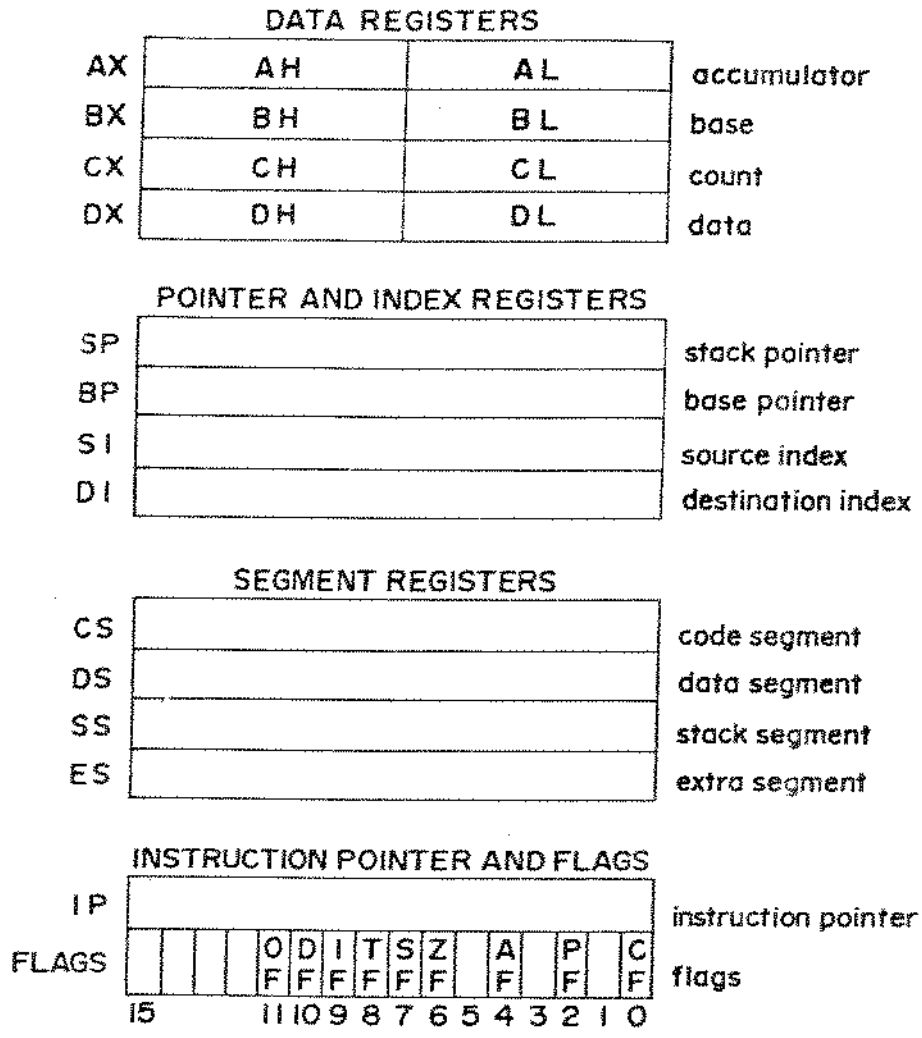

FIGURA 5.3 - O conjunto de registradores do 8088

Os flags IF, DF e TF indicam modos particulares de operação do processador. O flag de interrupções (IF) indica se as interrupções estão ou não habilitadas. O flag de direção (DF) é usado em conjunto com instruções para manipulação de cadeias de bytes, indicando em que direção uma cadeia será movida. Quando o flag de trap (TF) está ligado é serada uma interrupção após a execução de cada instrução, permitindo a execução passo a passo de um programa.

O grupo de registradores de segmento está relacionado ao esquema de endereçamento do 8088 e através deles pode-se endereçar um megabyte de memória. O endereçamento de um megabyte de memória requer o uso de 20 bits, como todos os registradores do 8088 são de 16 bits forma-se um endereço somando dois registradores, um

dos quais é sempre um registrador de segmento. A soma é feita deslocando o conteúdo do registrador de segmeno quatro bits a esquerda e somando esse valor ao segundo registrador. O processo de formação de endereços no 8088 é mostrado na Figura 5.4.

O endereço de onde será lida uma instrução para execução é formado somando o conteúdo do registrador IP ao conteúdo do registrador CS. Por exemplo, se IP contiver 0100H e CS contiver 0200H a próxima instrução a executar será aquela que está no endereço 02100H.

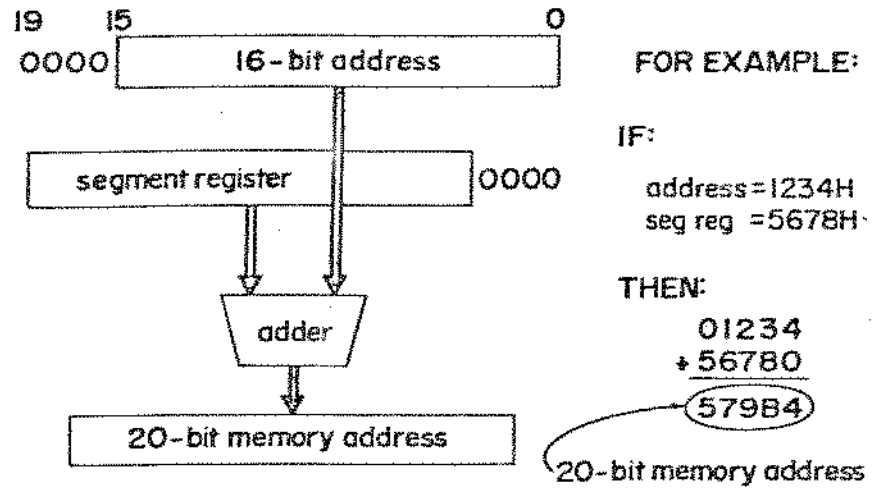

FIGURA 5.4 - A formação de um endereço pelo 8088

A soma do conteúdo de um registrador base de um segmento a um deslocamento relativo a essa base, a fim de formar um endereço de 20 bits, é feita automaticamente pelo 8088. Para fazer referência a um endereço explicitando a base e o deslocamento relativo a ela usa-se a notação ⟨segmento⟩:⟨deslocamento⟩, logo poderíamos representar o endereço do exemplo anterior escrevendo CS:IP = 200H:100H.

Todos os acessos feitos pelo 8088 à memória são formados por endereços de 20 bits. Quando um dado é movido para a memória ou da memória usando a instrução MOV um endereço de 20 bits é formado usando o registrador do segmento de dados (DS). Por exemplo, a instrução MOV AL, [100] move para AL o dado da posição cujo endereço é formado somando 100 ao conteúdo do registrador DS deslocado de quatro bits à esquerda. O registrador do segmento de pilha (SS) é somado ao registrador SP ou ao registrador BP para obter um endereço. Finalmente, o registrador do segmento extra (ES) é geralmente usado juntamente com o registrador DI para formar um endereço em instruções de manipulação de cadeias de bytes.

Em resumo, no esquema de endereçamento do 8088 cada registrador de segmento aponta para o endereço inicial de um bloco de memória de 64K bytes (chamado um segmento), localizado num espaço de endereçamento de um megabyte. Este segmento de 64K bytes pode começar em qualquer endereço que seja múltiplo de 16 (devido ao deslocamento de quatro bits à esquerda usado na formação de um endereço de 20 bits). A Figura 5.5 mostra um possível mapa de memória resultante de um dado conjunto de valores dos registradores de segmentos. Não há restrições quanto às posições dos segmentos de memória apontados por CS, DS, SS e ES; eles podem ocupar regiões totalmente distintas de memória ou podem sobrepor-se de forma parcial ou mesmo total.

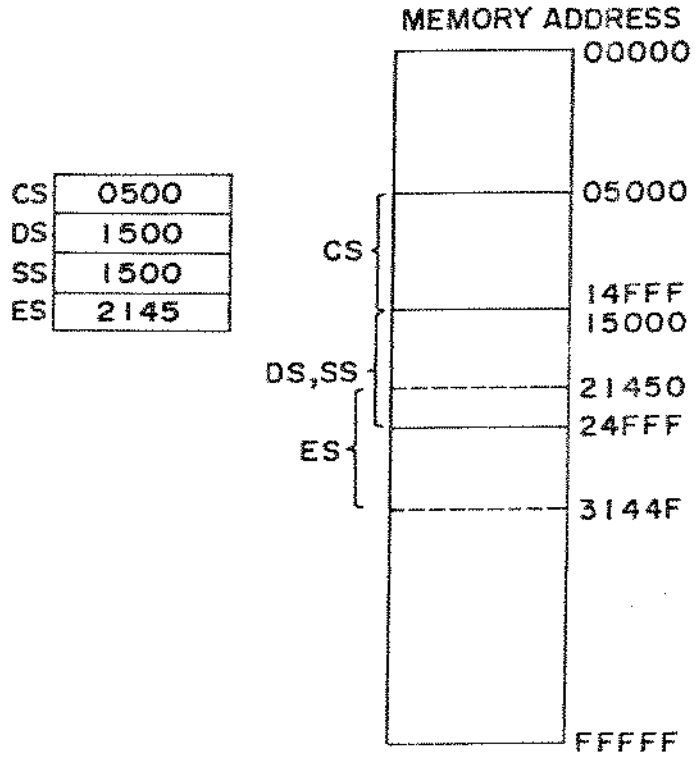

FIGURA 5.5 - Um exemplo do endereçamento dos segmentos de memória usando os registradores de segmento

5.2.2. Tratamento de Interrupções no 8088

O 8088 apresenta dois tipos de interrupções. As interrupções por software e as interrupções por hardware. Uma interrupção por software é gerada como consequência da execução da instrução INT. A execução de uma instrução INT n provoca os seguintes eventos:

a) Os registradores de Flags, CS e IP são empilhados;

b) As interrupções são desabilitadas;

c) O controle do processador é transferido para a instrução que se encontra no endereço contido na posição de memória 0:n*4.

O n da instrução INT n representa um valor inteiro entre 0 e 255, em consequência caso se queira utilizar as 256 possíveis interrupções os primeiros 1024 bytes de memória devem ser reservados para o vetor de interrupções. Cada grupo de quatro bytes dessa região conterá o endereço da rotina de tratamento da interrupção correspondente, conforme ilustrado na Figura 5.6.

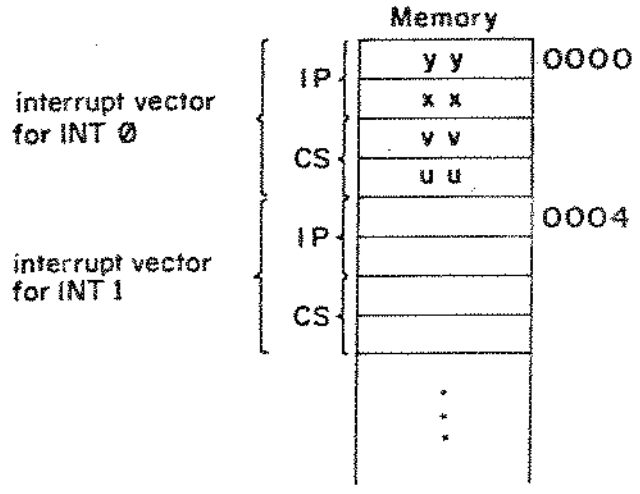

FIGURA 5.6 - Estrutura da área do vetor de interrupções. O endereço inicial do tratador da interrupção zero será CS:IP=uuvv:xxyy

Uma subrotina chamada por uma instrução INT difere de uma subrotina normal, acessível por um CALL, apenas pelo fato de ela ter que terminar com uma instrução IRET (retorno de uma interrupção) em lugar de uma instrução normal de retorno (RET). A instrução IRET restaura CS, IP e os flass, enquanto uma instrução RET restaura apenas IP ou IP e CS no caso de a instrução CALL que acessou a rotina não estar no mesmo segmento desta.

Uma interrupção por hardware ocorre quando um dispositivo externo envia um sinal ao 8088 através da linha de requisição de interrupções. O 8088 responde à requisição interrompoendo o que estava fazendo e desviando para uma subrotina que trata a interrupção. A rotina executada é determinada por um número n (de 8 bits) que é passado ao 8088 no barramento de dados. O número n é usado para localizar a subrotina a executar do mesmo modo como foi descrito para a instrução INT n.

5.2.3. Tempo de Execução das Instruções no 8088

O 8086 possui uma unidade de execução (EU) e uma unidade de interface ao barramento (BIU). A EU contém os resistradores de dados, os registradores de endereços, a Unidade Lógica e Aritmética e a Unidade de Controle. A BIU contém a lógica de acesso ao barramento, os registradores de segmentos, a lógica de endereçamento da memória e uma fila de instruções de 6 bytes.

A EU e a BIU operam assincronamente. Quando a EU está pronta para executar uma nova instrução ela retira o código no início da fila da BIU, cujo tempo de acesso é muito menor do que o da memória. Se a fila estiver vazia a EU asuarda que a BIU faça um acesso à memória para buscar uma instrução. Entretanto a fila raramente estará vazia, pois a BIU é independente da EU e procura mantê-la sempre cheia. Se dois ou mais bytes da fila estiverem vazios a BIU faz um acesso à memória (desde que a EU não tenha uma requisição pendente) a fim de preenchê-la.

A Figura 5.7 ilustra a EU e BIU do 8088. NO 8088 a fila da BIU é de apenas 4 bytes e a BIU começará a acessar a memória a fim de preenchê-la tão logo um ou mais bytes da fila estejam vazios. Como a fila do 8088 é menor que a do 8086, instruções que neste caberiam totalmente na fila podem requerer acessos adicionais à memória naquele. Por exemplo,

```
SUB TABELA [BX],300
```

representa uma instrução de 6 bytes, onde 2 bytes correspondem ao código da instrução, 2 bytes correspondem ao deslocamento a par-

tir de BX (TABELA) e os últimos 2 bytes correspondem ao dado a mover (0300D). Assumindo que os primeiros quatro bytes estejam contidos na fila do 8088, os dois últimos bytes devem ser buscados na memória, o que representa 8 pulsos do relógio a mais sobre o tempo de execução da mesma instrução no 8086. Ou seja, para cada byte a mais a buscar na memória devem ser adicionados 4 pulsos de relógio ao tempo de execução da mesma instrução pelo 8086. Adicionalmente o fato de o barramento de dados do 8088 ser de 8 bits implica em que ele deve fazer dois acessos à memoria para buscar um dado de 16 bits enquanto que o 8086 precisa fazer apenas um acesso.

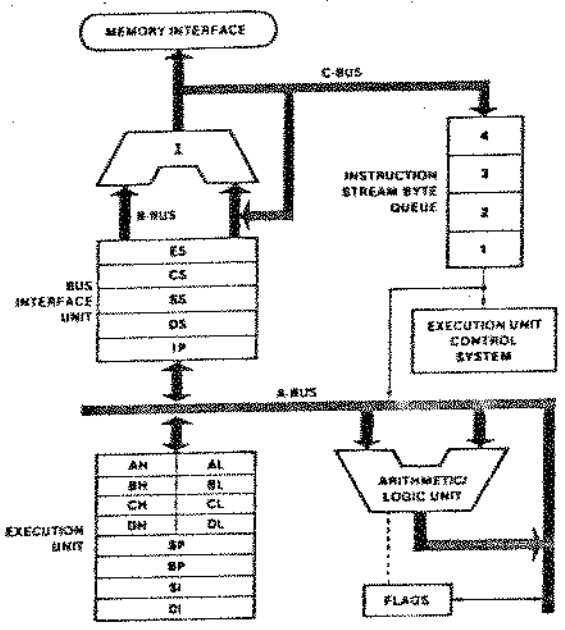

FIGURA 5.7 - As unidades de execução e interface ao barramento no 8088

Esta seção introduziu a arquitetura do microprocessador 8088. A próxima seção discute a utilização do espaço de endereçamento de um microcomputador baseado no 8088, para a execução dos programas de aplicação desenvolvidos com o auxílio das linguagens LPM e LCM.

5.3. Mapa de Memória do Sistema em Operação

A Figura 5.8 apresenta o mapa de memória do computador quando um programa configurado pela LCM estiver rodando sob o suporte do STR. Os primeiros 1024 são reservados para o vetor de interrupções, o qual pode conter até 256 elementos. A seguir encontra-se carregado o MS-DOS seguido do STR e dos módulos da aplicação.

Os itens a seguir discutirão com mais detalhes os componentes do mapa de memória da Figura 5.8.

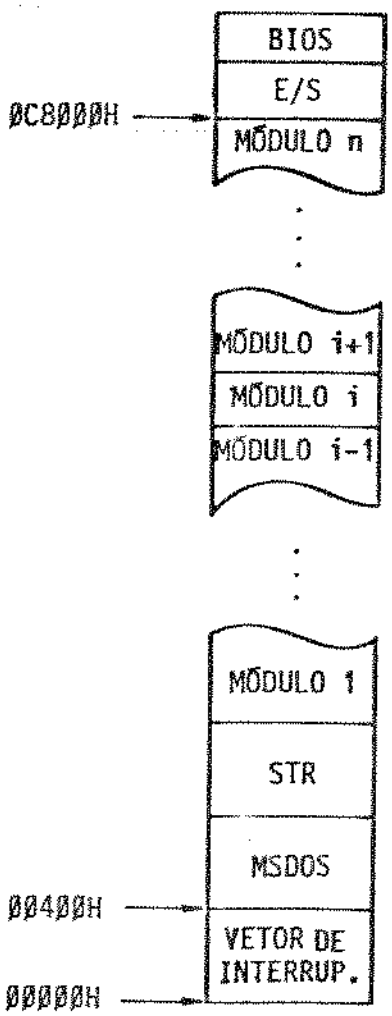

FIGURA 5.8 - Mapa de memória de uma estação em operação

5.3.1. Vetor de Interrupções

Os primeiros 400H bytes de memória em sistemas baseados nos microprocessadores INTEL 8086 ou 8088 estão geralmente reservados para o vetor de interrupções. O vetor de interrupção contém endereços de rotinas para o tratamento de interrupções gerados por hardware ou por software. As interrupções geradas por hardware destinam-se a atender aos periféricos do computador tais como teclado, impressoras, canais seriais, controladores de disco, interfaces análogo-digitais e o relógio de tempo real. A Figura 5.9 apresenta o uso do vetor de interrupções no IBM-PC.

São de particular interesse as interrupções 8 e 1CH que estão associadas ao relógio de tempo real (RTR). A interrupção número 8 é gerada 18,2 vezes por segundo pelo RTR. O tratador da interrupção número 8 mantém o horário e a data do MS-DOS e após gera, por software, uma interrupção de número 1CH. A interrupção 1CH está associada ao gerenciador de tempo do STR.

As interrupções de números 60H e 61H são usadas pelo STR e serão chamadas de VSTR e DSSTR respectivamente. As posições de memória correspondentes às interrupções VSTR e DSSTR serão carregadas durante a fase de inicialização do STR, descrita adiante. VSTR será carregada com o endereço do decodificador de comandos do STR. O decodificador de comandos acessa os procedimentos do STR que implementam os serviços requisitados pelo usuário. DSSTR conterá o valor do registrador do segmento de dados (DS) do STR. O uso de VSTR e DSSTR será explicado mais adiante.

5.3.2. MS-DOS

O MS-DOS ("Microsoft Operating System") ou apenas DOS é o mais popular sistema operacional para microcomputadores baseados no 8086/8088. O MS-DOS não é certamente o mais elegante dos sistemas operacionais disponíveis para microcomputadores, inclusive para o IBM-PC, porém, ele é pequeno, rápido e oferece os recursos necessários ao desenvolvimento da maioria das aplicações convencionais. O MS-DOS tornou- se um padrão "de fato" para microcomputadores baseados no 8086/8088, assim como, o mais antigo e menos poderoso CP/M-80 já se tinha tornado para micros baseados no 8080/8085/Z80. Embora os dois estejam longe de serem S.O. ideais, o fato de tantos usuários terem seguido suas convenções e desenvolvido uma infinidade de programas para eles tornou-os muito mais úteis que outros sistemas operacionais "bem melhores".

INTERRUPT	FUNCTION
0	Divide by zero
1	Single step
2	Non-maskable interrupt (NMI)
3	Break point instruction
4	Overflow
5	Print screen
6,7	Reserved
8	Time of day hardware interrupt (18.2/sec.)
9	Keyboard hardware interrupt
A	Reserved
B,C	Serial communications hardware interrupts
D	Fixed disk hardware interrupt
E	Diskette hardware interrupt
F	Printer hardware interrupt
10	Video I/O call
11	Equipment check call
12	Memory check call
13	Diskette I/O call
14	RS232 I/O call
15	Cassette I/O call
16	Keyboard I/O call
17	Printer I/O call
18	ROM basic entry code
19	Boot strap loader
1A	Time of day call
1B	Get control on keyboard break
1C	Get control on timer interrupt
1D	Pointer to video initialization table
1E	Pointer to diskette parameter table
1F	Pointer to graphics character generator
20	DOS program terminate
21	DOS function call
22	DOS terminate address
23	DOS CTRL-BRK exit address
24	DOS fatal error vector
25	DOS absolute disk read
26	DOS absolute disk write
27	DOS terminate, fix in storage
28-3F	Reserved for DOS
40-5F	Reserved
60-67	Reserved for user software interrupts
68-7F	[not used]
80-85	Reserved by BASIC
86-F0	Used by BASIC interpreter while running
F1-FF	[not used]

FIGURA 5.9 – O uso do vetor de interrupção no IBM-PC

O DOS é um programa em disco que é carregado automaticamente quando o computador é ligado ou reinicializado ("resetado"). O MS-DOS 2.0 possui 26 comandos internos (ou embutidos) e 19 comandos externos. Quando um comando interno como ERASE (apaga um arquivo) é digitado, o MS-DOS executa-o instantaneamente pois o código necessário já está na memória. Quando um comando externo como o FORMAT é digitado o MS-DOS procura no disco um arquivo com esse nome e com a extensão COM ou EXE, e caso o encontre carrega-o na memória e executa-o.

O MS-DOS 2.0 inclui diversos mecanismos muito úteis existentes no UNIX tais como "pipes", "caminhos de busca" ("search paths"), redirecionamento de E/S e diretório com estrutura em árvore. O conceito de redirecionamento de E/S permite que qualquer dispositivo seja tratado como um arquivo em disco, e vice-versa. Um "pipe" é um mecanismo que permite que a saída padrão de um programa seja usada como entrada padrão de outro programa.

O DOS consiste de uma série de funções facilmente acessíveis pelos programas do usuário. Quando essas funções (ou serviços) são chamadas pelos programas do usuário elas recebem parâmetros através dos resistradores do processador e de blocos de controle criados na memória. Quando o atendimento de uma função necessitar acessar um periférico o DOS gera uma ou mais chamadas ao BIOS para fazer o acesso físico ao periférico.

O DOS oferece aos programas do usuário um número significativo de serviços como leitura do teclado, saída ao vídeo e à impressora, construção de blocos de controle de arquivos, gerenciamento de memória, manutenção de data e hora e uma variedade de serviços para manipulação de arquivos e diretórios. A maioria dos serviços requerem que os parâmetros necessários lhes sejam passados nos resistradores. A forma padrão de chamar o MS-DOS (a partir da versão 2.0) consiste em gerar uma interrupção 21H através da instrução INT 21H.

5.3.3. BIOS

O BIOS reside em memórias do tipo "apenas-leitura" (Read Only Memory-ROM) e oferece um conjunto de serviços que permitem o acesso aos dispositivos de entrada e saída do computador. O BIOS oferece uma interface que torna eventuais mudanças no hardware transparentes aos programas do usuário. As rotinas do BIOS permitem ao programador "assembly" efetuar operações de entrada e saída ao nível de transferência de blocos (leitura e escrita de setores de discos) e de caracteres (leitura e escrita de caractéres em terminais, linhas seriais, etc.) sem a necessidade de conhecer as características de operação dos dispositivos.

O acesso às rotinas do BIOS é feito através de interrupções de software. Cada ponto de entrada ao BIOS tem associado um elemento do vetor de interrupções do 8088. A passagem de parâmetros entre os programas do usuário e as rotinas do BIOS é feita mediante o uso dos resistradores do 8088. Normalmente um programador de aplicação faz E/S solicitando os serviços de um sistema operacional, como o MS-DOS, que roda sob o BIOS, e não acessando diretamente as rotinas do BIOS. O acesso direto aos serviços do BIOS está, geralmente, restrito a programadores de software básico.

5.3.4. Estrutura do STR

Conforme ilustrado na Figura 5.10 o STR é dividido basicamente em uma área de código e uma área de dados. A área de código é de tamanho fixo e ocupa aproximadamente 7 KB. Já a área de dados terá seu tamanho definido pelas características das aplicações que suportará, pois para cada módulo e para cada porta da aplicação é necessário alocar uma estrutura de dados que a represente.

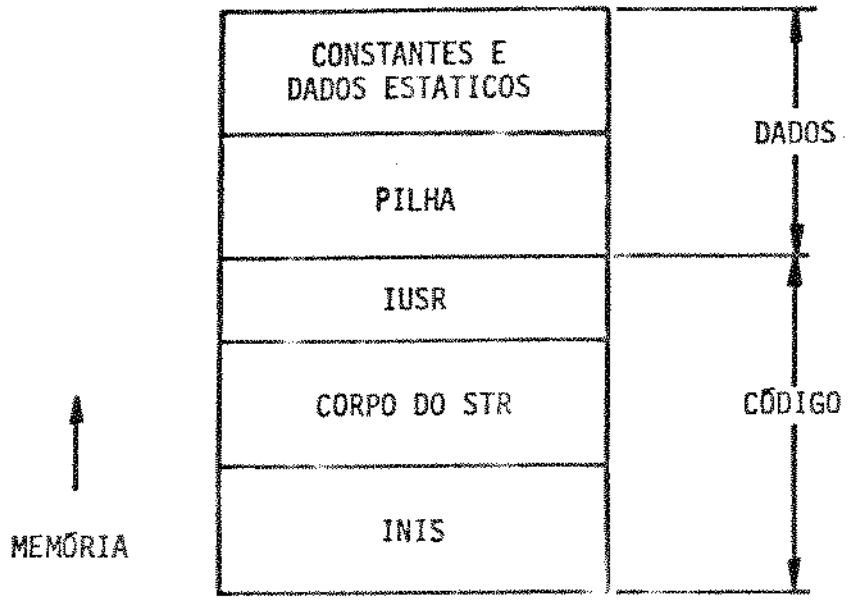

FIGURA 5.10 - Mapa de memória do STR

A área de código é composta de três elementos principais: INIS, o corpo do STR e IUSR. INIS é a rotina que recebe o controle após a carga do STR, sendo responsável pelas inicializações que antecedem a carga da aplicação. A descrição de sua operação será feita no item que tratará da carga do STR. O corpo do STR contém os procedimentos e funções, discutidos no Capítulo 4, que implementam os serviços oferecidos aos módulos da aplicação. IUSR é composto de rotinas que formam a interface entre o STR e os módulos do usuário (aplicação). IUSR será melhor discutido no item que trata da interface entre os módulos e o STR.

A área de dados é dividida entre uma região destinada a constantes e dados estáticos e outra destinada à pilha. Na região de dados estáticos as rotinas que carregam a aplicação alocarão os BCMS e descritores de portas necessários à execução da aplicação. A pilha do STR será utilizada apenas durante a fase de ini-

cialização do STR já que para o atendimento de requisição de serviços o STR trabalha na pilha dos módulos requisitantes.

Embora o STR seja um programa PASCAL a Figura 5.10 não apresenta uma área destinada ao "heap", isto é devido ao fato de a alocação dinâmica de memória do STR ser realizado pelo gerenciador de memória e não através da função NEW do PASCAL.

5.3.5. Estrutura de um Módulo

A Figura 5.11 mostra a estrutura geral de um módulo do usuário. A visível semelhança entre a sua estrutura e a do STR deve-se ao fato de ambos serem programas PASCAL.

A área de código de um módulo é composta basicamente da rotina de inicialização INIM, do corpo do módulo e das rotinas de interface ao STR reunidas em ISTR. A rotina INIM recebe o controle imediatamente após a carga do módulo, efetuando as inicializações que antecedem a sua execução. Sua operação será descrita no item que tratará da carga da aplicação. O corpo do módulo é composto pelo programa principal e pelos procedimentos resultantes da compilação do programa PASCAL, gerado a partir do módulo LPM do usuário. As rotinas de ISTR serão discutidas no item que trata da interface entre os módulos e o STR.

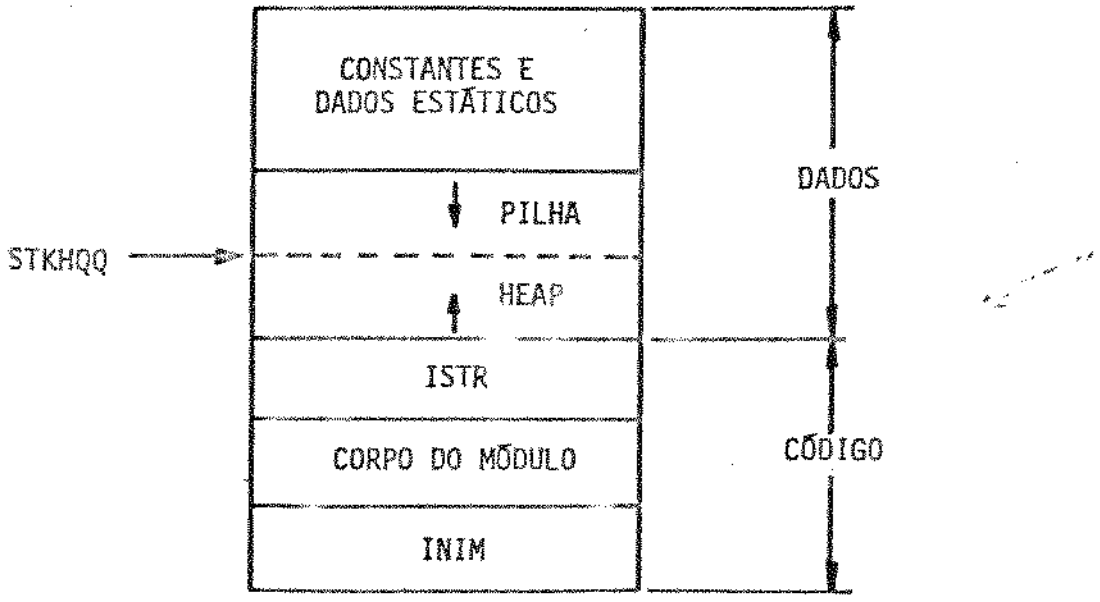

FIGURA 5.11 - Mapa de memória de um módulo

A área de dados de um módulo está dividida entre três regiões principais; a primeira é destinada a constantes e dados estáticos, a segunda destina-se à pilha e a terceira será usada como "heap".

Os dados estáticos correspondem às variáveis globais declaradas no corpo do módulo. A pilha do módulo será usada no salvamento de endereços de retorno, passagem de parâmetros e alocação de variáveis locais durante a chamada de procedimentos. Quando chamado para oferecer algum serviço, o STR recebe os parâmetros necessários na pilha do módulo requisitante, onde trabalhará e na qual desenvolverá, quando for o caso, os resultados de sua operação. O apontador STKHQQ delimita a fronteira entre as áreas de pilha e "heap" e será usado, quando desejado, para verificar se está havendo estouro da pilha ou do "heap" durante chamadas de procedimentos ou alocação dinâmica de variáveis.

5.3.6. Interface entre os Módulos e o STR

De modo a atender ao requisito de separação dos mapas de memória dos módulos entre si e destes e do STR, definiu-se uma interface que permite aos módulos acessar os serviços do STR sem o conhecimento explícito dos endereços dos procedimentos que os implementam. A Figura 5.12 apresenta o esquema de interface utilizado.

Durante a fase de inicialização do STR a rotina INIS carrega VSTR, no vetor de interrupções, com o endereço do decodificador de pedidos de serviço (IUSR). Dessa forma, independentemente do lugar da memória onde os módulos e o STR sejam carregados a comunicação entre ambos é estabelecida através do endereço carregado nessa posição fixa de memória.

A LPM traduz as extensões à linguagem PASCAL em chamadas a procedimentos do STR que implementam os serviços. Como há uma estrita separação entre os espaços de endereçamento dos módulos entre si e destes e do STR, obtida mediante a compilação e ligação separada de cada módulo, o pré-processador da LPM gera a declaração da interface de cada serviço utilizado por um módulo como um procedimento externo.

A declaração dos serviços do STR como procedimentos externos adia a resolução dessas referências para a fase de ligação dos módulos, durante a qual ISTR é incorporado ao código do módulo. ISTR contém um ponto de entrada para cada serviço do STR usado pelo módulo. Em cada ponto de entrada de ISTR é carregado no registrador BX o código do serviço requisitado e feito um desvio indireto para VSTR. Através desse desvio entramos em IUSR, que usa o código passado em BX como o deslocamento numa tabela que

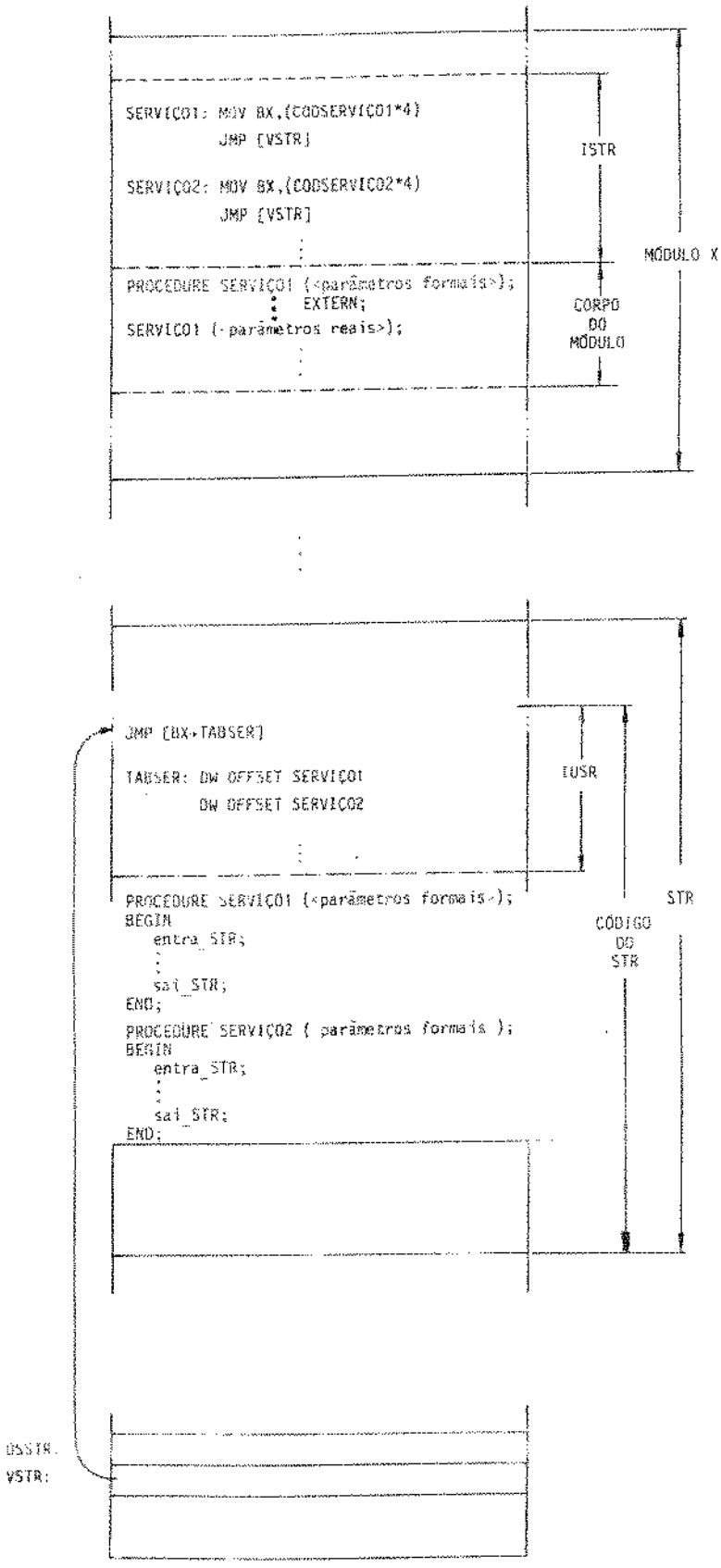

FIGURA 5.12 – A interface entre os módulos e o STR

contém o endereço de todos os procedimentos do STR que implementam serviços. Fazendo um desvio ao endereço cuja posição na tabela foi calculado usando BX como deslocamento entramos no procedimento que implementa o serviço requisitado pelo usuário.

É importante observar que durante o desvio para um serviço do STR, através do mecanismo de interface descrito acima, a pilha é preservada no estado deixado pelo módulo requisitante no instante da chamada ao serviço. Portanto, quando se inicia a execução de um procedimento respônsável por alsum serviço, tudo se passa como se o procedimetno tivesse sido acessado diretamente pelo usuário, através do processo normal de chamada de procedimentos.

Antes de realizar o serviço requisitado o STR executa um protocolo de entrada, comum a todos os serviços, que consiste na mudança do registrador base da área de dados do módulo para a área de dados do STR e na desabilitação de interrupções, já que os serviços do STR devem ser executados em exclusão mútua. O registrador base da área de dados do STR é carregado da posição DSSTR do vetor de interrupções, sendo armazenado nessa posição pela rotina de inicialização do STR INIS.

Embora exista uma separação explícita entre as áreas de dados estáticos do STR e dos módulos, implementada através da troca dos registros de base das áreas de dados, o mesmo não ocorre com a pilha. O STR trabalha na pilha do módulo que requisita o serviço e através dela recebe e retoma os parâmetros especificados na sua interface. Uma vez realizado o serviço o STR retorna ao módulo via um protocolo de saída, comum a todos os serviços, que consiste na recarga do registrador base da área de dados do móculo (armazenado no BCM deste) e na habilitação de interrupções.

5.4. Carga e Inicialização do STR

Para o MS-DOS o STR é um programa como outro qualquer, logo, para que ele seja carregado basta digitar o seu nome no teclado. O MS-DOS permite que se passe uma linha de comando ao programa sendo carregado, essa linha deve aparecer logo após o nome do programa, separada deste por um espaço. No caso do STR, a linha de comando esperada deve especificar o nome do arquivo que contém a tabela de configuração da aplicação que se pretende executar. Assim, para executarmos uma aplicação cuja configuração esteja descrita na tabela contida no arquivo CONFIG.EXE deve-se disitar:

```
C> STR CONFIG.EXE <CR>
```

Em resposta a este comando o MS-DOS carrega o STR na primeira posição livre de memória, passa-lhe o controle e lhe deixa disponível a linha de comando CONFIG.EXE. A execução do STR começa na rotina INIS, responsável pelas inicializações que antecedem a carga da aplicação.

Primeiramente, INIS reloca a área de constantes e dados do STR a fim de deixar espaço para a pilha a qual, como descrito anteriormente, será usada apenas durante a fase que antecede à execução da aplicação. A seguir INIS salva o registrador base da área de dados estáticos do STR (DS) na posição DSSTR do vetor de interrupção (Figura 5.9). O valor de DS depende da posição em que o STR foi carregado e da quantidade de memória alocada para a pilha.

Após a inicialização de DSSTR, INIS libera toda a memória acima do STR e solicita ao MS-DOS a carga da tabela de configuração especificada pelo usuário. O MS-DOS carregará a tabela na primeira posição livre após o STR. A tabela de configuração embora só contenha dados é vista pelo MS-DOS como um programa. Por esse motivo a liberação explicita de memória é necessária já que o MS-DOS aloca toda a memória disponível ao programa recem-carregado, não sendo possível a solicitação da carga de novos programas a menos que memória para esse fim seja liberada.

O STR dará sequência ao processo de inicialização usando as informações contidas na tabela de configuração. O processo de inicialização culmina com a carga dos módulos de aplicação e com a transferência do controle ao módulo de maior prioridade.

A estrutura da tabela de configuração é mostrada na Figura 5.13. Cada descritor de módulo dará origem a um Bloco de Controle de Módulo (BCM) e a um número de Descritores de Portas de Entrada (DPE) igual à quantidade dessas portas no módulo. Os BCMs e DPEs são alocados dinamicamente e inicializados a partir dos dados contidos nos descritores de módulos que lhes deram origem.

Cada descritor de conexão, da tabela de configuração, representa a ligação de uma porta de saída a uma porta de entrada. Um descritor de conexão tem a forma [(Módulo, Porta), (Módulo, Porta)] onde o primeiro par (Módulo, Porta) identifica uma porta de saída e o segundo par (Módulo, Porta) identifica uma porta de entrada à qual a porta de saída está ligada. A partir de um descritor de conexão será colocado na posição correspondente à porta de saída do módulo origem da conexão um apontador ao DPE da porta de entrada de destino.

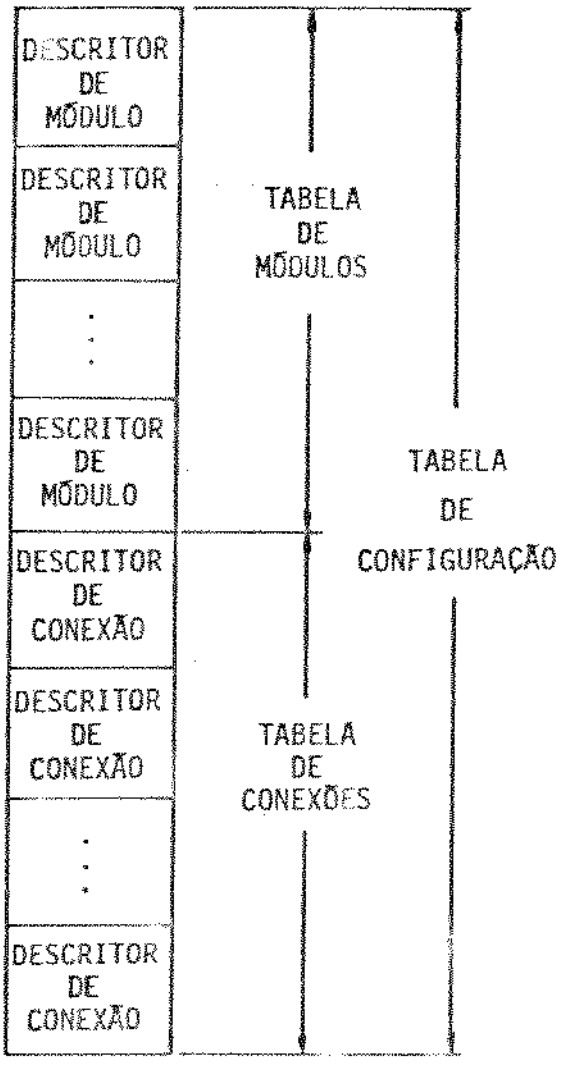

FIGURA 5.13 - A tabela de configuração

5.5. Carga e Execução da Aplicação

Após a inicialização das suas estruturas de dados, a partir da tabela de configuração, o STR procede à carga e posterior "disparo" da aplicação conforme será descrito nos itens a seguir.

5.5.1. Carga da Aplicação

A carga da aplicação compreende duas atividades principais, a primeira consiste na carga do código objeto dos módulos e a segunda na criação do contexto inicial de cada módulo.

A carga do código objeto de cada módulo é feita pelo MS-DOS em atendimento a uma solicitação feita pelo STR. A carga de um módulo é totalmente independente da carga dos demais, sendo cada uma vista pelo MS-DOS como a carga de um novo programa. O STR obtém os nomes dos arquivos que contêm os programas (módulos) a carregar nos BCMs criados a partir da tabela de configuração.

O contexto inicial de um módulo consiste do registrador base da sua área de dados estáticos (DS), do seu ponto de entrada (CS:IP) e do apontador do topo da sua pilha (SS:SP). A criação do contexto inicial de um módulo dá-se através da interação do STR com o módulo, durante o processo de carga deste. A interação entre o STR e o módulo é necessária pois a geração de um módulo é independente da geração dos demais módulos e da geração do STR, não havendo, em consequência, uma fase de ligação global de todos os componentes da aplicação onde esse contexto seja previamente determinado. Essa fase de ligação não é de fato desejável pois dificultaria bastante a futura implementação de mecanismos para reconfiguração dinâmica do sistema.

Antes de dar início à carga dos módulos o STR inicializa a posição VSTR do vetor de interrupções assim, a partir desse instante, os módulos poderão acessar os serviços do STR conforme descrito no item 5.3.4.

O processo de carga de um módulo dá-se nos seguintes passos:

- O STR inicializa um apontador de sua estrutura de dados, denominado running, com o endereço BCM do módulo e solicita ao MS-DOS a carga e execução do programa (módulo) cujo nome obteve nesse BCM;

- O MS-DOS carrega o programa na primeira posição livre de memória e lhe passa o controle;

- O ponto de entrada do programa recém-carregado é a sua rotina INIM. A rotina INIM interage com o STR a fim de inicializar o mapa de memória do módulo;

- Primeiramente, INIM solicita ao STR, através do serviço denominado memória, que lhe seja informada a quantidade de memória que deverá alocar para "heap" e pilha do módulo que está sendo carregado;

- O STR responde à solicitação acessando o BCM apontado por running;

- INIM recebe o valor de memória requisitado e reloca a área de constantes e de variáveis estáticas do módulo a fim de deixar o espaço especificado para "heap" e pilha. Após a relocação dos dados INIM acessa o serviço inires do STR informando ao STR a quantidade total de memória ocupada pelo módulo e o conteúdo dos registradores DS, SS e SP. Este acesso é feito na instrução de INIM imediatamente anterior àquela em que este transfere o controle ao corpo do módulo.

- O serviço inires inicializa os campos basedados e pilha no BCM do módulo requisitante. O campo basedados é inicializado com o valor recebido no registrador DS e o campo pilha é inicializado com os valores contidos nos registradores SS e SP. Dessa forma, o contexto inicial do módulo requisitante está totalmente definido já que o endereço da instrução que transfere o controle ao corpo do módulo encontra-se salvo na pilha. Havendo sido carregado o módulo, feita a alocação da memória necessária à sua operação e conhecido o seu contexto inicial o STR pode dar como encerrada a carga deste módulo.

Convém lembrar que neste instante para o MS-DOS o programa que se encontra em execução é o programa (módulo) que foi carregado e executado a pedido do STR. O procedimento do STR que implementa o serviço inires é visto, neste momento, como um procedimento do programa que acabou de ser carregado. Valendo-se deste fato o procedimento que implementa inires solicita ao MS-DOS que seja encerrada a execução do programa corrente e que o controle retorne ao programa que solicitou a sua carga e execução. Nesta requisição ao MS-DOS o procedimento inires informa o tamanho do programa corrente e solicita que ele seja mantido na memória. Como consequência dessa solicitação, o módulo que foi carregado é mantido na memória e o controle retorna à instrução do STR imediatamente posterior àquela através da qual o MS-DOS foi chamado para carregar e executar o módulo. Caso seja requisitada a carga de um novo programa o MS-DOS considerará como primeira posição livre, a partir de onde a carga se iniciará, aquela que segue ao programa anteriormente carregado.

O processo de carga acima descrito é repetido para todos os módulos componentes da aplicação, após o que, a aplicação estará totalmente carregada na memória e pronta para ser executada. Observe-se que durante o processo de carga da aplicação foram executadas as rotinas INIM de cada módulo e não o corpo dos mesmos. Ou seja, do ponto de vista do programa de aplicação a execução ainda não começou.

O item a seguir apresentará o disparo do programa concorrente cuja carga foi descrita acima.

5.5.2. Execução da Aplicação

Durante a fase de configuração cada módulo recebeu uma prioridade inicial. A fim de obedecer a essa hierarquia de prioridades, os BCMs dos módulos do programa de aplicação são colocados nas filas associadas ao estado pronto para executar, de acordo com a prioridade de cada um.

A dinâmica do STR exige que sempre haja um módulo em execução; a fim de atender a este requisito, antes de passar o controle à aplicação, o núcleo cria um módulo de nome idle e coloca-o no estado rodando. O módulo idle é criado com a mais baixa prioridade do sistema, prioridade esta sempre inferior à menor prioridade de um módulo da aplicação. O módulo idle será executado somente quando não houver nenhum módulo da aplicação em condições de ser executado. Após a criação do módulo idle o STR faz um reescalonamento, provocando a transferência do controle deste para o módulo de maior prioridade do usuário.

- O primeiro serviço requisitado por um módulo, caso ele seja parametrizado, é inipar cuja interface é dada por:

```
PROCEDURE inipar (endpar: ADDRESS);
```

onde endpar é o endereço inicial a partir de onde foram alocadas as variáveis correspondentes aos parâmetros do módulo. A chamada a esse serviço é transparente ao usuário pois ela é gerada pela LPM sempre que um módulo for parametrizado. Este serviço é necessário porque em decorrência da independência dos espaços de endereçamento dos módulos e do STR, este não sabe em que posição da área de dados daqueles o compilador PASCAL alocou as variáveis correspondentes aos seus parâmetros. A partir desse ponto o comportamento do sistema dependerá exclusivamente do programa de aplicação do usuário. O STR intervirá apenas quando chamado pelos módulos do usuário ou pela ocorrência de alguma interrupção. Mesmo neste último caso o atendimento a uma interrupção está condicionado à sua inicialização por parte de um módulo tratador de interrupção do usuário.

5.6. Tratamento de Interrupções

No Capítulo 4 os mecanismos para tratamento de interrupções foram apresentados de um ponto de vista estritamente lógico, sem maiores preocupações com detalhes específicos da arquitetura do hardware onde o sistema foi implementado. Neste item será feito o casamento entre os sistemas lógico e físico de interrupções.

O mapeamento de uma interrupção física em uma interrupção lógica é feito pelo procedimento **inivet**. Quando chamado pelo sistema lógico de interrupções, inivet coloca na posição de memória correspondente ao elemento de um vetor físico especificado pelo usuário o endereço de uma sequência de instruções associada ao número da interrupção lógica alocada pelo STR. A Figura 5.14 ilustra o resultado da instrução inivet(1CH,1); pode-se observar que na posição correspondente à interrupção física 1CH foi colocado o endereço de uma sequência de instruções que carrega o registrador AX com o valor 1, que corresponde ao número da interrupção lógica que o STR associa à interrupção física 1CH.

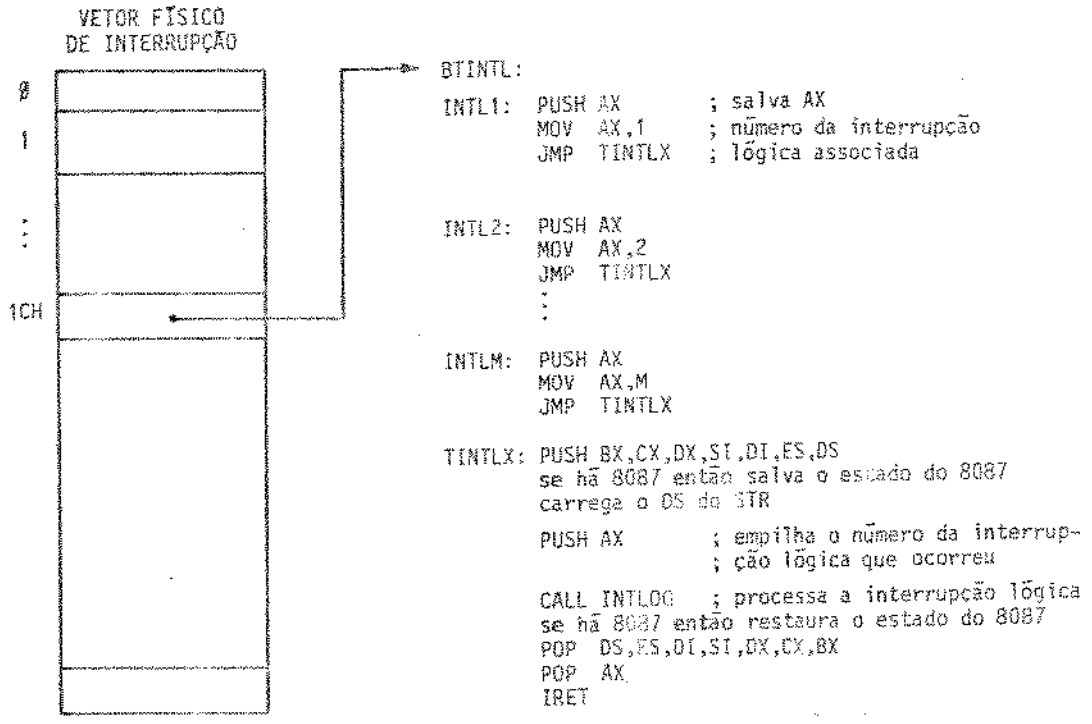

FIGURA 5.14 - A resposta à ocorrência de uma interrupção

Como as interrupções podem ocorrer de maneira totalmente assíncrona e consequentemente fora do controle do protocolo normal de acesso aos serviços do STR é necessário, quanto de seu atendimento, salvar todos os registradores do 8088 e o estado do processador aritmético 8087, já que eles podem estar em uso no momento da ocorrência da interrupção devendo, portanto, serem preservados até que o módulo interrompido retome o controle da máquina. A chamada ao procedimento **INTLOG** pode provocar ou não a transferência do controle para outro módulo. Haverá uma transferência de controle caso o tratador de interrupções esteja à espera da interrupção ocorrida e tenha uma prioridade superior àquela do módulo interrompido.

5.7. Considerações sobre o Desempenho do STR

No projeto de sistemas de software para aplicações em tempo real é muito importante assegurar um tempo de resposta compatível com o processo a monitorar ou controlar.

Nesta sentido, esta seção apresenta os tempos aproximados, em microsegundos, necessários ao atendimento dos principais serviços envolvidos na interação entre os módulos aplicativos. Outro parâmetro também importante no dimensionamento de um sistema é o consumo de memória, para facilitar a sua determinação apresenta-se ainda dados relativos às necessidades de memória em função das características da aplicação.

Os dados a seguir prestam-se tanto para dimensionar um sistema de aplicação quanto para identificar trechos menos eficientes do STR, a fim de otimizá-los caso eles estejam comprometendo o desempenho global do sistema. A otimização de algum dos procedimentos do STR pode ser feita mediante uma nova escolha do algoritmo empregado ou simplesmente recodificando o procedimento em linguagem de montagem.

5.7.1. Código do STR

A Tabela 5.1 apresenta alguns dados relativos aos códigos fonte e objeto do STR. Através dela pode-se ter uma idéia do esforço empregado no seu desenvolvimento bem como do trabalho envolvido no seu transporte para outros equipamentos.

O código PASCAL corresponde aos procedimentos que implementam os serviços oferecidos pelo STR sendo virtualmente independente do hardware empregado. Na sua programação empregam-se algumas extensões ao PASCAL-ISO, visando contornar a sua rígida tipificação. Isto se faz necessário em alguns poucos pontos do STR para manipular endereços físicos através do uso de apontadores. As extensões citadas são específicas do compilador empregado, porém, a sua inexistência não comprometeria a implementação do STR pois elas podem ser definidas no PASCAL padrão usando registros variantes. Esta opção só não foi utilizada porque o uso das extensões presentes no compilador empregado permitia uma programação mais eficiente.

O código em linguagem de montagem (Assembly) corresponde às rotinas dependentes da arquitetura do hardware e àquelas rotinas mais simples e críticas cuja implementação em Assembly apresenta um considerável ganho de velocidade com menor custo de memória, relativamente às suas equivalentes em PASCAL. Um exemplo de rotina dependente do hardware é a executa, que chaveia o processador

de um módulo para outro. Um exemplo de rotina que poderia ser programada em PASCAL mas que por razões de eficiência o foi em linguagem de montagem é a copia, que transfere uma sequência de bytes (geralmente uma mensagem) de uma região da memória para outra.

DISCRIMINAÇÃO	LINHAS DE CÓDIGO FONTE	% DO CÓDIGO FONTE	CÓDIGO OBJETO GERADO (EM BYTES)	% DO CÓDIGO OBJETO
Código em PASCAL	1500	84	5400	77
Código em Linguagem de Montagem	300	16	530	8
Código da Biblioteca PASCAL	-	-	1070	15
TOTAL	1800	100	7000	100

TABELA 5.1 - O código do STR

5.7.2. Área de Dados do STR

No dimensionamento dos requisitos de memória para dados do STR devem ser consideradas três quantidades: a primeira corresponde a uma área de tamanho fixo de 700 bytes; a segunda à tabela de módulos e a terceira à área para alocação dinâmica de estruturas de dados. As áreas variáveis devem ser alocadas em função das características das aplicações e o seu dimensionamento é descrito a seguir, onde todos os valores são expressos em bytes.

A tabela de módulos requer uma área de duas vezes o número máximo de módulos que se deseja que o STR suporte.

Na área para alocação dinâmica de estruturas de dados são criados os blocos descritores de módulos (BCM) e os descritores de portas de entrada (DPE). A quantidade de memória necessária para cada BCM é dada pela expressão

70 + max_par + (2*num_port)

onde, max_par é o tamanho máximo dos parâmetros do módulo e max_port é o número total de portas de entrada e saída do módulo. Na versão atual max_par deve ser igual para todos os módulos. Para cada porta de entrada síncrona é alocado um DPE de 24 bytes. Para cada porta de entrada assíncrona é alocado um DPE de 34 bytes mais a área necessária para o seu "buffer". Adicionalmente nas ligações multi-destino devem se alocados os elos da lista de apontadores às portas de entrada destino, o que demanda mais 6 bytes para cada porta destinatária.

5.7.3. Tempo de Resposta dos Serviços do STR

Os dados aqui apresentados permitem estimar o tempo de resposta de um módulo a um evento associado a uma troca de mensasens ou interrupção. Através deles pode-se formar uma idéia inicial da possibilidade ou não de utilizar esta versão do STR numa dada aplicação.

Os tempos de resposta correspondem a um 8088 operando a uma freqüência de 4.77 MHZ e foram calculados a partir do código objeto dos procedimentos do núcleo. Utilizou-se o tempo médio aproximado para a execução de cada tipo de instrução (movimento de dados, operações lógicas e aritméticas, desvio etc.) considerando o modo de endereçamento empregado em cada uma delas.

Os tempos de resposta dos procedimentos associados ao reescalonador, ao tratamento de interrupções e aos serviços de troca de mensasens são apresentados nos próximos itens, principalmente sob a forma de tabelas. A unidade empregada nas tabelas é o microsegundo (us) e o tempo de execução de uma rotina inclui a execução das rotinas por ela chamadas, ainda que os tempos destas apareçem separadamente na tabela. Por exemplo, na Tabela 5.2 os 327us necessários à execução de pronto_reesc, caso seja feito um reescalonamento, incluem os 105us de execução do procedimento pronto e os 175us do procedimento reescalona.

5.7.3.1. Reescalonamento

A Tabela 5.2 apresenta os tempos de execução dos procedimentos associados ao reescalonador. Na tabela, o tempo de execução de reescalona inclui a execução do procedimento executa, ou seja, desde a entrada de reescalona até o início da execução da primeira instrução do módulo que vai receber o controle, decorrem 175us.

ROTINA	TEMPO DE EXECUÇÃO (us)	OBSERVAÇÕES
reescalona	175	Até o início do módulo que ganha o controle
pronto	105	
pronto_reesc	152	Se não for feito um reescalonamento
	327	Se for feito um reescalonamento

TABELA 5.2 - Tempos de resposta associados ao reescalonador

5.7.3.2. Atendimento de Interrupções

Se ocorrer uma interrupção e o módulo tratador associado não estiver à sua espera, são gastos 133us (ou 221us caso se esteja usando um 8087) para salvar o contexto do módulo em execução, registrar a ocorrência da interrupção lógica associada, restaurar o contexto do módulo interrompido e retornar-lhe o controle.

Caso haja um módulo à espera por uma interrupção no momento de sua ocorrência, o tempo necessária para executar a sua primeira instrução, desde o instante do atendimento da interrupção, é de 423us (ou 467us com 8087).

5.7.3.3. Serviços de Troca de Mensagens

A Tabela 5.3 apresenta os tempos de resposta dos serviços de comunicação mais importantes. Para um perfeito entendimento dos dados constantes dessa tabela faz-se necessário o conhecimento anterior da semântica e implementação de cada serviço, assuntos já tratados em capítulos anteriores. Os parágrafos a seguir fazem algumas considerações adicionais visando simplificar a interpretação dos dados da tabela.

Os valores expressos na tabela dão o tempo de execução dos serviços a partir do início dos procedimentos que os implementam; eles não incluem o tempo gasto pela interface entre os módulos e o núcleo que é de 22us.

As quatro primeiras linhas da tabela referem-se ao serviço env_a. Essas linhas cobrem o tempo de execução de todas as possíveis situações encontradas durante um envio assíncrono. A primeira linha dá o tempo de execução de env_a quando o módulo destinatário não está à espera da mensagem enviada. Nesse caso a mensagem é copiada para o "buffer" de entrada da porta de entrada e o controle retorna ao módulo emissor; toda essa operação requer 492 us mais 4us para cada byte da mensagem enviada. Na segunda linha dá-se o tempo de execução do serviço quando o módulo destinatário está à espera pela mensagem. O tempo dado compreende a cópia da mensagem para a região da memória do módulo destinatário onde a mensagem é aguardada. A terceira e quarta linhas referem-se à situação na qual o módulo destinatário está à espera pela mensagem e tem prioridade superior à do módulo emissor, razão pela qual ganhará o controle do processador. O tempo expresso na terceira linha compreende a cópia da mensagem para a região da memória do módulo destinatário e todo o processo de reescalonamento que termina quando se inicia a execução da primeira instrução do módulo receptor da mensagem. Quando o módulo emissor recuperar o controle, a sua execução se iniciará dentro do procedimento do núcleo que implementa o envio assíncrono, no comando que sucede à chamada do reescalonador. Desse ponto até o início da execução da primeira instrução após aquela que requisitou o serviço de envio assíncrono são necessários outros 38us. Estas considerações feitas para o serviço env_a aplicam-se igualmente a situações correspondentes nos demais serviços.

O cálculo do tempo de execução do serviço sel requer a soma de duas parcelas. A primeira corresponde ao tempo de escolha de uma das alternativas a executar. Se alguma das portas associadas às alternativas do SELECT tiver recebido uma mensagem, a segunda parcela corresponde ao tempo para copiar a mensagem para a área de dados do módulo receptor. O valor dessa parcela dependerá do tipo da porta escolhida. Se nenhuma das portas fazendo recepção seletiva tiver recebido uma mensagem e houver uma alternativa ELSE no SELECT, a segunda parcela corresponderá a 170us. Se nenhuma das portas tiver recebido uma mensagem e o SELECT não in-

cluir uma alternativa ELSE, o tempo necessário para executar a primeira instrução do módulo do início da fila do pronto será a soma da primeira parcela a uma segunda dada por 308us mais 76us vezes o número total de portas de entrada à espera por recepção no comando SELECT. Neste caso, após a recuperação do controle por parte do módulo receptor são necessários ainda 99us para iniciar a execução da instrução após aquela que requisitou o serviço sel.

SERVIÇO	TEMPO DE EXECUÇÃO (us)	OBSERVAÇÕES
env_a	492+tcopia	Caso não haja um módulo à espera da mensagem
	557+tcopia	Caso o módulo à espera não ganhe o controle
	767+tcopia	Até o inicio do módulo que recebe o controle
	38	Da recuperação do controle até a instruçao após aquela que requisitou o serviço
env_s	718+tcopia	Se o módulo destinatário estiver esperando uma mensagem, tempo até o início do módulo que recebe o controle
	545+tcopia	Se o módulo destinatário não estiver esperando uma mensagem, tempo até o início do módulo que recebe o controle
	40	Da recuperação do controle até a instrução após aquela que requisitou o serviço
rec	378+tcopia	Se já houver chegado uma mensagem síncrona
	479+tcopia	Se já houver chegado uma mensagem assíncrona
	336	Se não houver uma mensagem disponível, tempo até o início do módulo que recebe o controle
	38	Da recuperação do controle até a instrução após aquela que requisitou o serviço

Continuação

	660+tcopia	Caso o módulo que receber a mensagem desviada não sanhe o controle (i.e., ele não tem prioridade superior ao que fez o desvio)
desvia	824+tcopia	Caso o módulo que receber a mensagem desviada sanhe o controle, tempo até o início de sua execução
	38	Da recuperação do controle até a instrução após aquela que requisitou o serviço
ini_rec_sel	206	
ini_temp_sel	160	
	49+(36*prt_ante)	Para escolha priorizada da alternativa a executar
	((20+(prt_rec*11))*tot_prt)+75	Para escolha aleatória da alternativa a executar
	356+tcopia	Para copiar uma mensagem síncrona
	456+tcopia	Para copiar uma mensagem assíncrona
sel	170	Se não houver uma mensagem disponível e o SELECT tiver uma alternativa ELSE (somar ao tempo máximo de escolha priorizada)
	308+ 76*tot_prt)	Se não houver uma mensagem disponível e o SELECT não tiver uma alternativa ELSE, tempo até o início do módulo que recebe o controle
	99	Da recuperação do controle até a instrução após aquela que requisitou o serviço

Continuação

	546	Até o início do módulo que recebe o controle
resp	38	Da recuperação do controle até a instrução após aquela que requisitou o serviço.

tcopia = (tam_mem*4)

tam_mem = tamanho da mensagem copiada em bytes

prt_antes = número de portas de maior prioridade que aquela que recebeu a mensagem

prt_rec = número de portas que receberam mensagens

tot_prt = número total de portas em espera seletiva

TABELA 5.3 - Tempos de resposta dos serviços de troca de mensagens

CAPÍTULO 6

EXEMPLOS

6. EXEMPLOS

Este capítulo ilustra, através de dois exemplos, o processo de programação, configuração e execução de aplicações em LPM e LCM. Os módulos componentes dos exemplos ilustram o uso da maioria das construções da LPM. Na sua configuração são empregados esquemas de conexão um para um, um para vários e vários para um entre portas assíncronas e síncronas. Os programas PASCAL resultantes da pré-compilação dos módulos destes exemplos constituem os Anexos A e B do trabalho.

Os exemplos introduzem o modo pelo qual os componentes de um programa concorrente de aplicação têm acesso aos serviços do MS-DOS, que por não ser reentrante requer a adoção de uma estratégia para sequencializar tentativas de acesso concorrente. Isto é obtido estabelecendo um relacionamento do tipo cliente-servidor entre os módulos das aplicações, onde os módulos servidores recebem concorrentemente pedidos de E/S dos módulos clientes e seriadamente interage com o MS-DOS para atendê-los. Os módulos servidores destes exemplos são bastante específicos, porém, usando este esquema básico deverão ser construídos módulos mais gerais oferecendo serviços de E/S aos diversos periféricos do computador. O mesmo esquema deve ser empregado à criação de servidores que implementem serviços sem a intermediação de um sistema operacional hospedeiro, no caso o MS-DOS.

Os módulos dos exemplos têm como único propósito ilustrar o tipo de programação obtido com a LPM e LCM não implementando nenhum algoritmo específico. Os exemplos serão referenciados como exemplo 1 e exemplo 2, sendo apresentados nas seções 6.1 e 6.2 respectivamente.

6.1. Exemplo 1

O exemplo 1 é formado por 5 tipos de módulos: A, B, C, D e TIMEMAN. Estes módulos serão descritos no item 6.1.1, a configuração do exemplo a partir destes 5 tipos básicos é apresentada no item 6.1.2, finalmente, o item 6.1.3. discute a execução da aplicação resultante.

6.1.1. Módulos Componentes

O código LPM do módulo do tipo A, referenciado a seguir apenas por módulo A, é ilustrado na Figura 6.1. A definição de contexto do módulo especifica objetos de duas unidades de definição: TipSis e TipMen. A unidade de definição TipSis contém os objetos pré-declarados do sistema que não são tratados diretamente pela LPM, entre eles o tipo PRIORITY. A unidade de definição TipMem, mostrada na Figura 6.2, define os tipos das mensagens que serão usadas para comunicação e sincronização entre os módulos da aplicação.

```
MODULE a(pausa:INTEGER);

   USE  TipSis:PRIORITY;
        TipMen:ident,pausa_prio;

   EXITPORT
        ps1:ident REPLY pausa_prio;
        ps2:ident;

   MESSAGE
        men_ident:ident;
        men_pausa_prio:pausa_prio;

   VAR
        i:INTEGER;

($PAGE+)
   BEGIN
        men_ident.modulo:=MOD1;
        men_ident.num:=0;
        REPEAT
           FOR i:=1 TO 4 DO
                BEGIN
                   men_ident.instante:=TIME;
                   men_ident.num:=men_ident.num+1;
                   SEND men_ident TO ps2;
                   DELAY(pausa);
                END;
           men_ident.instante:=TIME;
           men_ident.num:=mem.ident.num+1;
           SEND men_ident TO ps1 WAIT men_pausa_prio;
           pausa:=men_pausa_prio.pausa;
           SETPRIORITY(men_pausa_prio.prio);
        UNTIL FALSE;
   END.
```

FIGURA 6.1 - Código LPM do módulo tipo A

```
DEFINE TipMen:ident,pausa_prio;

   TYPE
        ident=RECORD
                modulo:INTEGER;
                instante:INTEGER4;
                num:INTEGER;
              END;

        pausa_prio=RECORD
                        pausa:INTEGER;
                        prio:PRIORITY;
                   END;

   END.
```

FIGURA 6.2 - A unidade de definição TipMen

Os tipos definidos na unidade TipMen servem de base para a declaração das portas e mensagens do módulo, feitas após as palavras reservadas EXITPORT e MESSAGE respectivamente. O ciclo de execução do módulo é o seguinte: são enviadas quatro mensagens do tipo ident à porta de saída assíncrona ps2, havendo entre o envio de cada mensagem um intervalo especificado pelo parâmetro pausa. Essas mensagens são numeradas e contém o identificador pelo qual o STR conhece a instância do módulo sendo executada, bem como o instante em que a mensagem foi enviada. O identificador do módulo para o STR é um inteiro definido em tempo de configuração da aplicação pela LCM e acessível através da função MODID. O instante em que a mensagem é enviada é obtido lendo o relógio do STR através da função TIME. Após o envio dessas quatro mensagens assíncronas o módulo A envia uma mensagem síncrona à porta de saída ps1. Ao enviar essa mensagem o módulo será suspenso até a chegada de uma resposta que conterá novos valores para a pausa entre o envio de cada mensagem assíncrona, bem como para a prioridade da instância. A mudança de prioridade é obtida mediante o procedimento SETPRIORITY.

O módulo tipo B, mostrado na Figura 6.3, usa as mesmas unidades de definição do módulo tipo A e tem o seguinte ciclo de execução: o módulo faz uma recepção bloqueante através de porta síncrona de entrada ps1. Recebida uma mensagem o módulo registra o momento em que isso ocorreu no campo instante da mensagem mem_ident e em vez de enviar uma mensagem em resposta à mensagem síncrona recebida o módulo redireciona a comunicação para a porta síncrona de saída ps1, deixando a cargo do módulo que contém a porta a ela conectada o envio da resposta. Após repetir essa sequência por duas vezes o módulo solicita a pausa e a prioridade através da porta síncrona de saída ps1 e procede de modo análogo ao módulo A. Deve ser observado que essa solicitação foi feita através da mesma porta de saída, ps1, usada para desviar a mensagem recebida pela porta de entrada ps1.

```
MODULE b;

   USE  TipSis:PRIORITY;
        TipMen:ident,pausa_prio;

   ENTRYPORT
        pe1:ident REPLY pausa_prio;

   EXITPORT
        ps1:ident REPLY pausa_prio;
        ps2:ident;

   MESSAGE
        men_identA,men_identB:ident;
        men_pausa_prio:pausa_prio;

   VAR
        i:INTEGER;

   BEGIN
        men_identB.modulo:=MODID;
        men_identB.num:=0;
        REPEAT
           FOR i:=1 TO 2 DO
                BEGIN
                   RECEIVE men_identA FROM pe1;
                   men_identB.instante:=TIME;
                   men_identB.num:=men_identB.num+1;
                   FORWARD pe1 TO ps1;
                   SEND men_identB TO ps2;
                END;
           men_identB.instante:=TIME;
           men_identB.num:=men_identB.num+1;
           SEND men_identB TO ps1 WAIT men_pausa_prio;
           SETPRIORITY(men_pausa_prio.prio);
        UNTIL FALSE;
   END.
```

FIGURA 6.3 - Código LYN do módulo tipo B

O módulo tipo C, mostrado na Figura 6.4, desempenhará o papel de servidor de E/S para os demais módulos da aplicação. O ciclo de execução do módulo tipo C é o seguinte: Após efetuar 40 recepções seletivas o módulo solicita novos valores para o tempo de espera do comando PSELECT e para a sua prioridade. A solicita-

```
MODULE c(espera_sel:INTEGER);

  USE   TipSis:PRIORITY;
        TipMen:ident,pausa_prio;

  ENTRYPORT
        pe1:ident REPLY pausa_prio;
        pe2:ident QUEUE 6;

  MESSAGE
        men_req:men_ident:ident;
        men_pausa_prio:pausa_prio;

  VAR
        i,num_modulo:INTEGER;
        inst:INTEGER4;
        prioridade:PRIORITY;

  PROCEDURE LePausaPrio(mo:INTEGER; inst:INTEGER4; num_mem:INTEGER;
                        VAR pausa:INTEGER; VAR pri:PRIORITY);

  BEGIN
      WRITE('? Em ',inst:4,' o modulo ',mo:1,' pela mensagem ',
            num_men:3,' pediu a pausa e a  prioridade? ');
      READLN(pausa,pri);
  END;

  BEGIN
        num_modulo:=MODID;
        REPEAT
           FOR i:=1 TO 40 DO
              PSELECT
                   RECEIVE men_req FROM pe1
                   =>   LePausaPrio(men_req.modulo,men_req.instante,
                                    men_req.num,men_pausa_prio.pausa,
                                    men_pausa_prio.prio);
                        REPLY men_pausa_prio TO pe1;
                OR
                   RECEIVE men_ident FROM pe2
                   =>   inst:=TIME;
                        WRITELN('* Em ',inst:4,' foi recebida a mensagem ',
                                men_ident.num:3,' que o modulo ',
                                men_ident.modulo:1,' enviou em ',
                                men_ident.instante:4);
                OR
                   TIMEOUT espera_sel
                   =>   inst:=TIME;
                        WRITELN('Timeout em ',inst:4);
                END;
           inst:=TIME;
           LePausaPrio(num_modulo,inst,0,espera_sel,prioridade);
           SETPRIORITY(prioridade);
        UNTIL FALSE;
  END.
```

FIGURA 6.4 - Código LPM de módulo tipo C

ção é feita mediante comandos WRITE e READ do PASCAL que por "default" escrevem na tela . lêem do teclado. A execução desses comandos implicará na requisição de serviços, não reentrantes, do MS-DOS, razão pela qual devem ser concentrados num único módulo servidor ou usados após o estabelecimento de sincronismo, mediante o uso de mensagens, entre os módulos envolvidos. A solução do servidor por ser mais geral e elegante é a mais indicada para estas situações. Neste exemplo o servidor é implementado empregando o comando de recepção PSELECT, que privilegia o atendimento de requisições de novos valores para pausa e prioridade dos módulos, feitos por intermédio da porta de entrada pe1. Caso não haja mensagens à espera na pe1 são retiradas as mensagens armazenadas no "buffer" da porta de entrada pe2 e mostradas na tela juntamente com o instante em que foram recebidas. Caso não haja nenhuma mensagem disponível em nenhuma dessas portas de entrada o módulo aguardará um período de tempo, dado pelo conteúdo de espera_sel, antes de dar uma mensagem indicativa de "timeout".

O módulo tipo D, mostrado na Figura 6.5, tem um ciclo de execução muito simples que consiste em efetuar uma recepção bloqueante na porta de entrada pe1 e enviar a mensagem recebida para a porta de saída ps1.

```
MODULE d;

   USE  TipSis:PRIORITY;
        TipMen:ident,pausa_prio

   ENTRYPORT
        pe1:ident;

   EXITPORT
        ps1:ident;

   MESSAGE
        men_ident:ident;

   BEGIN
        REPEAT
           RECEIVE men_ident FROM pe1;
           SEND men_ident TO ps1;
        UNTIL FALSE;
   END.
```

FIGURA 6.5 - Código LPM do módulo tipo D

O módulo tipo TIMEMAN, mostrado na Figura 6.6, ilustra a construção de um módulo tratador de interrupções; neste caso as interrupções tratadas são aquelas geradas pelo relógio de tempo real (RTR). O módulo inicialmente mapeia o elemento do vetor físico de interrupções associado ao RTR num elemento do vetor lógico de interrupções usando a função INTALLOC. A seguir o módulo entra no seu ciclo de operação que consiste na espera da ocorrência do número pulsos do RTR correspondente a uma unidade de tempo do sistema e na sinalização da sua ocorrência ao STR através do procedimento TICK. O número de pulsos do RTR correspondente a um "tick" é definido durante a fase de configuração da aplicação. A frequência padrão da ocorrência de pulsos do RTR no IBM-PC é de 18,2 vezes por segundo (1 pulso a cada aproximadamente 55ms), podendo no entanto ser aumentada através da sua reprogramação.

```
MODULE timeman(pulsos:INTEGER);

    CONST
        vrtr=16#1C;

    VAR
        i,int_log_rtr:INTEGER;

    BEGIN
        int_log_rtr:=INTALLOC(vrtr);
        REPEAT
           FOR i:=1 TO pulsos DO WAITIO(int_log_rtr);
           TICK;
        UNTIL FALSE;
    END.
```

FIGURA 6.6 - Código LPM do módulo tipo TIMEMAN

A definição da configuração do exemplo, usando os módulos dos tipos A, B, C, D e TIMEMAN, é apresentada na próxima seção.

6.1.2. Configuração

A Figura 6.7 apresenta o diagrama de configuração do exemplo, onde são representadas as conexões entre as portas das instâncias dos módulos de tipos A, B, C, D e TIMEMAN. O programa em LCM para descrever a configuração ilustrada no diagrama é mostrado na Figura 6.8.

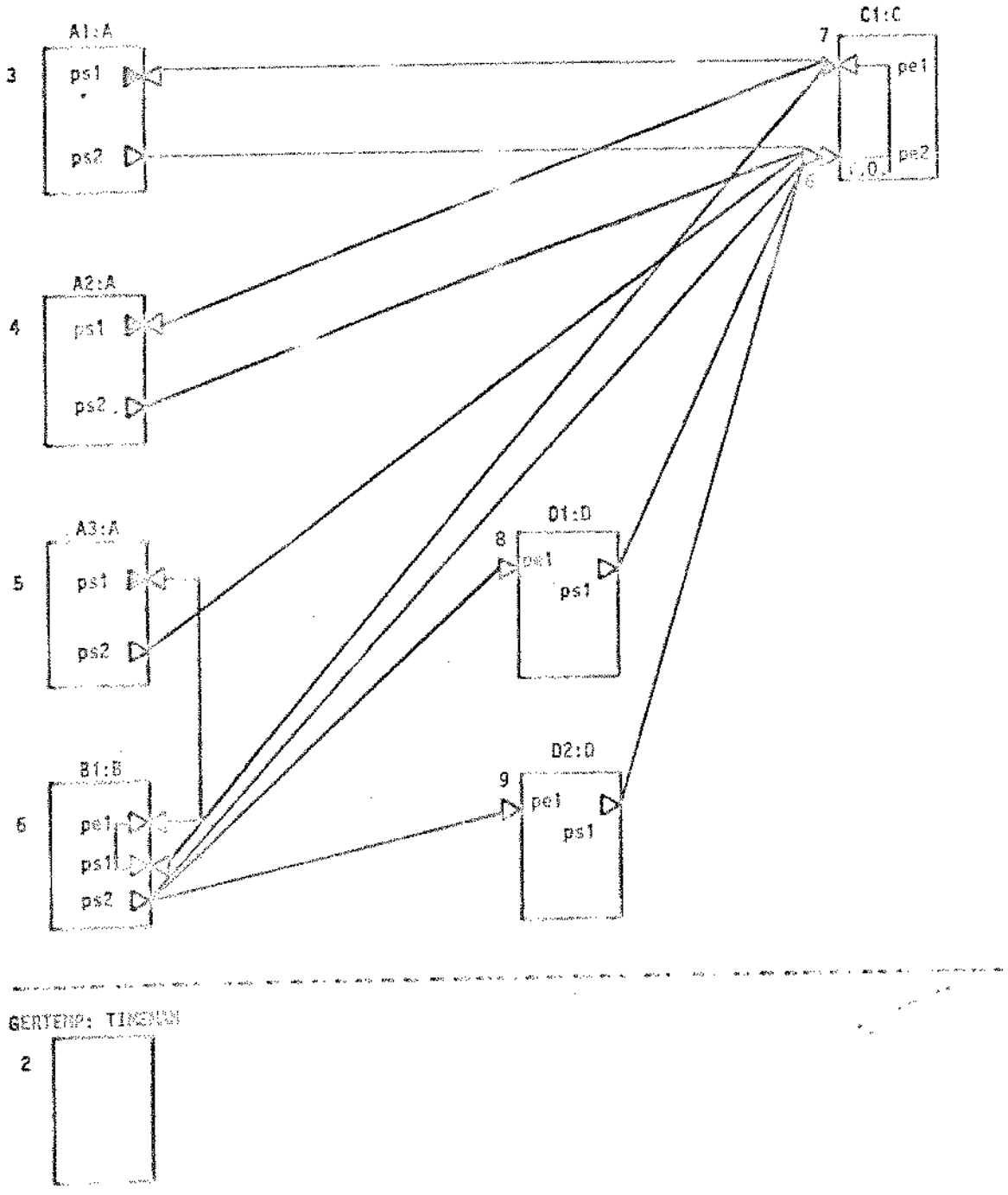

FIGURA 4.7 – Diagrama de configuração do exemplo 1

```
CONFIGURATION exemplo1;
     INSTANCE   GERTEMP:TIMEMAN;
                A1,A2,A3:A;
                B1:B;
                C1:C;
                D1,D2:D;
     CREATE     GERTEMP(1)/P=Systempr,
                A1(100), A2(100), A3(100),
                B1,
                D1,D2,
                C1(100)/P=Highpr/S=64/H=2;
     LINK       A1.ps1 TO C1.pe1;
                A1.ps2 TO C1.pe2;
                A2.ps1 TO C1.pe1;
                A2.ps2 TO C1.pe2;
                A3.ps1 TO B1.pe1;
                A3.ps2 TO C1.pe2;
                B1.ps1 TO C1.pe1;
                B1.ps2 TO C1.pe2, D1.PE1, D2.PE1;
                D1.ps1 TO C1.pe2;
                D2.ps1 TO C1.pe2;
END
```

FIGURA 6.8 - Programa em LCM para a configuração do exemplo 1

A partir do tipo de módulo A são criadas as instâncias A1, A2 e A3; a partir do tipo B é criada a instância B1; a partir do tipo C é criada a instância C1; a partir do tipo D são criadas as instâncias D1 e D2 e a partir do tipo TIMEMAN é criada a instância GERTEMP.

A conexão das portas de saída A1.ps1 (porta ps1 da instância A1), A2.ps1, B1.ps1 à porta de entrada C1.pe1 ilustra uma comunicação do tipo muitas para uma (ou n para 1) entre portas síncronas. A conexão das portas de saída A1.ps2, A2.ps2, A3.ps2, B1.ps2, D1.ps1 e D2.ps1 à porta de entrada C1.pe2 ilustram uma comunicação muitas para uma entre portas assíncronas. As comunicações muitas para uma envolvendo portas síncronas são características de relacionamentos do tipo cliente-servidor entre módulos, onde o servidor recebe numa porta associada a um dado serviço as solicitações dos clientes.

Na criação da instância GERTEMP o valor 1 é associado ao parâmetro formal pulsos e a prioridade inicial da instância, definida por intermédio da chave P, tem o valor systempr. A atribuição da prioridade systempr a GERTEMP visa assegurar um pronto processamento da interrupção do STR, dessa maneira evita-se que seja "perdido" algum pulso pois todos os demais módulos da aplicação serão executados com prioridades menores e, em consequência, perderão o controle do processador sempre que, durante a sua execução, uma interrupção aguardada por GERTEMP ocorra.

As instâncias A1, A2 e A3 são criadas associando o valor 100 ao parâmetro formal pausa. A não explicitação de uma prioridade inicial indica que elas serão executadas tendo como nível de prioridade normalpr.

Na criação da instância C1 o valor 100 é associado ao parâmetro formal espera_sel, a prioridade inicial da instância, definida pela chave P, tem o valor hishpr, a memória para pilha do módulo, definida por intermédio da chave S, é de 64 parágrafos (um parágrafo corresponde a 16 bytes) e a memória para heap, definida pela chave H, é de 2 parágrafos. A execução da instância C1 com o nível de prioprídade superior ao dos outros módulos não tratadores de interrupção da aplicação, visa oferecer um rápido atendimento aos serviços por eles requisitados. A manutenção de sua prioridade ao mesmo nível dos demais módulos implicaria no atendimetno de uma requisição de serviço apenas após a execução de todas as instâncias na frente de C1 na fila de pronto. Por outro lado, a manutenção de C1 em um nível de prioridade inferior ao das outras instâncias poderia provocar a perda de mensagens assíncronas enviadas a sua porta de entrada pe2. Isto ocorreria em razão de a maior prioridade das outras instâncias permitir que elas fossem executadas várias vezes antes de passar o controle a C1, nesses vários ciclos de execução o "buffer" de entrada de porta pe2 poderia ficar cheio e, em conseqüência, as mensagens mais antigas passariam a ser substituídas por aquelas mais recentes. Essas situações poderão ser observadas na próxima seção, onde se discute a execução do exemplo.

6.1.3. Execução

Nesta seção serão discutidas algumas situações resultantes da execução do exemplo cuja configuração foi descrita na seção precedente. Em razão da ordem com que foram criadas as instâncias no programa LCM da Figura 6.8 a instância GERTEMP será identificada pelo STR através do número 2 e as instâncias A1, A2, A3, B1, C1, D1 e D2 pelos números 3, 4, 5, 6, 7, 8 e 9 respectivamente. Estes números serão empregados nas figuras a seguir que ilustram a execução do exemplo.

Conforme se pode apreciar na Figura 6.9, que mostra o início da execução do exemplo, até o instante 40 as instâncias A1, A2 e A3 enviam mensagens assíncronas à instância C1 a intervalos de 10 "ticks". Em razão da maior prioridade de C1 as mensagens recebidas são imediatamente mostradas na tela. O atraso de uma mensagem ser recebida (mostrada na tela) no "mesmo" instante (mesma unidade de tempo) em que foi enviada deve-se à duração da unidade de tempo ser, neste exemplo, de aproximadamente 55ms, período suficiente para a ocorrência de vários escalonamentos.

```
Em    1 foi recebida a mensagem  1 que o modulo 3 enviou em    1
Em    6 foi recebida a mensagem  1 que o modulo 4 enviou em    6
Em   11 foi recebida a mensagem  1 que o modulo 5 enviou em   11
Em  101 foi recebida a mensagem  2 que o modulo 3 enviou em  101
Em  106 foi recebida a mensagem  2 que o modulo 4 enviou em  106
Em  111 foi recebida a mensagem  2 que o modulo 5 enviou em  111
Em  201 foi recebida a mensagem  3 que o modulo 3 enviou em  201
Em  206 foi recebida a mensagem  3 que o modulo 4 enviou em  206
Em  211 foi recebida a mensagem  3 que o modulo 5 enviou em  211
Em  301 foi recebida a mensagem  4 que o modulo 3 enviou em  301
Em  306 foi recebida a mensagem  4 que o modulo 4 enviou em  306
Em  311 foi recebida a mensagem  4 que o modulo 5 enviou em  311
Em  401 o modulo 3 pela mensagem  5 pediu a pausa e a prioridade? 1 2
Em  406 o modulo 4 pela mensagem  5 pediu a pausa e a prioridade? 10 2
Em  413 foi recebida a mensagem  6 que o modulo 3 enviou em  413
Em  418 o modulo 5 pela mensagem  5 pediu a pausa e a prioridade? 100 2
Em  429 foi recebida a mensagem  1 que o modulo 6 enviou em  418
Em  429 foi recebida a mensagem  6 que o modulo 4 enviou em  429
Em  434 foi recebida a mensagem  7 que o modulo 3 enviou em  434
Em  439 foi recebida a mensagem  1 que o modulo 6 enviou em  418
Em  448 foi recebida a mensagem  1 que o modulo 6 enviou em  418
Em  458 foi recebida a mensagem  6 que o modulo 3 enviou em  458
Em  455 foi recebida a mensagem  7 que o modulo 4 enviou em  455
Em  460 foi recebida a mensagem  6 que o modulo 5 enviou em  460
Em  466 foi recebida a mensagem  9 que o modulo 3 enviou em  466
Em  471 foi recebida a mensagem  8 que o modulo 4 enviou em  471
Em  477 o modulo 3 pela mensagem  1F pediu a pausa e a prioridade? 102 2
Em  491 foi recebida a mensagem  11 que o modulo 3 enviou em  491
Em  501 foi recebida a mensagem  9 que o modulo 4 enviou em  501
Em  511 o modulo 4 pela mensagem  10 pediu a pausa e a prioridade? 10 2
Em  529 foi recebida a mensagem  11 que o modulo 4 enviou em  529
Em  539 foi recebida a mensagem  12 que o modulo 4 enviou em  539
Em  549 foi recebida a mensagem  13 que o modulo 4 enviou em  549
Em  559 foi recebida a mensagem  14 que o modulo 4 enviou em  559
Em  564 foi recebida a mensagem  7 que o modulo 5 enviou em  564
Em  569 o modulo 5 pela mensagem  15 pediu a pausa e a prioridade? 1 2
Em  576 foi recebida a mensagem  16 que o modulo 4 enviou em  576
Em  581 foi recebida a mensagem  17 que o modulo 4 enviou em  581
Em  587 foi recebida a mensagem  18 que o modulo 4 enviou em  587
Em  592 foi recebida a mensagem  19 que o modulo 4 enviou em  592
Em  598 o modulo 7 pela mensagem  6 pediu a pausa e a prioridade? 100 2
Em  615 foi recebida a mensagem  1A que o modulo 3 enviou em  615
Em  681 o modulo 4 pela mensagem  2B pediu a pausa e a prioridade? 100 2
Em  654 foi recebida a mensagem  2J que o modulo 4 enviou em  654
Em  664 foi recebida a mensagem  8 que o modulo 5 enviou em  664
Em  715 foi recebida a mensagem  1B que o modulo 3 enviou em  715
Em  754 foi recebida a mensagem  2K que o modulo 4 enviou em  754
Em  764 foi recebida a mensagem  9 que o modulo 5 enviou em  764
Em  815 foi recebida a mensagem  1C que o modulo 3 enviou em  815
Em  854 foi recebida a mensagem  2S que o modulo 4 enviou em  854
Em  864 o modulo 5 pela mensagem  11 pediu a pausa e a prioridade? 100 2
Em  864 o modulo 6 pela mensagem  3 pediu a pausa e a prioridade? 100 2
Em  872 foi recebida a mensagem  2 que o modulo 6 enviou em  864
Em  882 foi recebida a mensagem  2 que o modulo 6 enviou em  864
Em  882 foi recebida a mensagem  2 que o modulo 6 enviou em  864
```

109

FIGURA 5.9 - Exemplo de Execução 1

Nos instantes 401, 407 e 413 é alterado o período durante o qual as instâncias A1, A2 e A3 se retardarão. Como fica evidente nos instantes seguintes, isso altera a frequência com que são enviadas as mensagens por essas instâncias. Em consequência já no instante 427 a instância A1 solicita nova pausa e prioridade enquanto que as instâncias A2 e A3 somente o farão nos instantes 511 e 844 respectivamente. Em 424 é recebida a primeira cópia da mensagem enviada por B1 diretamente a C1 após o desvio da mensagem síncrona que A3 mandou à sua porta pe1. As outras duas cópias enviadas indiretamente, por intermédio de D1 e D2, chegam nos instantes 439 e 444 respectivamente.

A instância B1 envia a mensagem número 3 para requisição de pausa e prioridade no instante 864, após haver recebido e desviado duas mensagens síncronas de A3. Observe que as cópias da mensagem número 2 são mostradas (recebidas) após a mensagem número 3, isto se deve à maior prioridade da porta síncrona pe1 de C1 por onde foi recebida a mensagem 3.

Na Figura 6.10 mostra-se um período da execução do exemplo durante o qual o tempo de espera por uma mensagem, no comando de recepção seletiva da instância C1, é reduzido para 10, enquanto o intervalo entre envio de mensagens nas outras instâncias é mantido em 100. Em consequência, se após 10 unidades de tempo de espera C1 não receber nenhuma mensagem, ele executará a sequência de comandos associada à expiração do tempo do SELECT, que consiste em dar uma mensagem de "time out".

O efeito da manutenção da prioridade do módulo servidor C1 ao mesmo nível e em nível inferior ao dos módulos clientes é ilustrado na Figura 6.11. No instante em que a instância C1 (módulo 7) solicita a pausa e a prioridade as demais instâncias operam com pausa 10 e prioridade 2. A partir do instante 10400, em que o módulo 3 solicita a pausa e a prioridade, C1 não mais consegue mostrar as mensagens síncronas enviadas pelos demais módulos. As mensagens síncronas são mostradas pois num determinado instante todos os clientes estarão bloqueados à espera de resposta. Pode-se ver claramente que antes de enviar a mensagem número 270, na qual pedira pausa e a prioridade, a instância A1 (módulo 3) enviou 4 mensagens assíncronas que C1 não conseguiu mostrar. Após haver colocado todos os clientes num nível de prioridade inferior ao do servidor, no instante 10443, este passa a receber prontamente as mensagens que recebe descarregando, inicialmente, as mensagens armazenadas no "buffer" da porta de entrada pe2. Durante o período em que C1 rodou com prioridade menor ou igual à das demais instâncias ele pode (não pode mostrar) as mensagens assíncronas da instância A1 (módulo 3) compreendidas entre 264 e 274; as mensagens assíncronas da instância A2 (módulo 4) compreendidas entre 274 e 284; as mensagens assíncronas da instância A3 (módulo 5) compreendidas entre 259 e 269 e todas as cópias das mensagens enviadas pela instância B1 (módulo 6), após desviar os pedidos de pausa e prioridade de A3, entre os instantes 10400 e 10800.

```
* Em 1881 foi recebida a mensagem  21 que o modulo 3 enviou em 1831
? Em 1890 o modulo 4 pela mensagem  30 pediu a pausa e a prioridade? 100 2
* Em 1926 foi recebida a mensagem  18 que o modulo 5 enviou em 1926
* Em 1932 foi recebida a mensagem  22 que o modulo 3 enviou em 1932
? Em 1937 o modulo 7 pela mensagem   0 pediu a pausa e a prioridade? 10 1
* Em 2010 foi recebida a mensagem  31 que o modulo 4 enviou em 2010
! Timeout em 2025
* Em 2026 foi recebida a mensagem  19 que o modulo 5 enviou em 2026
* Em 2032 foi recebida a mensagem  23 que o modulo 3 enviou em 2032
! Timeout em 2047
! Timeout em 2058
! Timeout em 2069
! Timeout em 2080
! Timeout em 2091
! Timeout em 2102
* Em 2110 foi recebida a mensagem  32 que o modulo 4 enviou em 2110
! Timeout em 2121
? Em 2126 o modulo 1 pela mensagem  21 pediu a pausa e a prioridade? 10 2
? Em 2126 o modulo 6 pela mensagem   6 pediu a pausa e a prioridade? 10 2
* Em 2211 foi recebida a mensagem   5 que o modulo 6 enviou em 2126
* Em 2217 foi recebida a mensagem   5 que o modulo 6 enviou em 2126
* Em 2223 foi recebida a mensagem   5 que o modulo 6 enviou em 2126
* Em 2228 foi recebida a mensagem  24 que o modulo 3 enviou em 2228
* Em 2233 foi recebida a mensagem  33 que o modulo 4 enviou em 2233
* Em 2239 foi recebida a mensagem  21 que o modulo 5 enviou em 2239
! Timeout em 2254
! Timeout em 2265
! Timeout em 2276
! Timeout em 2287
! Timeout em 2298
! Timeout em 2309
! Timeout em 2320
? Em 2328 o modulo 3 pela mensagem  26 pediu a pausa e a prioridade? 10 2
```

FIGURA 4.10 - Execução do Exemplo 1 (continuação)

```
* Em 13419 foi recebida a mensagem 264 que o modulo 3 enviou em 13418
* Em 13424 foi recebida a mensagem 274 que o modulo 4 enviou em 13424
* Em 13430 foi recebida a mensagem 259 que o modulo 5 enviou em 13430
? Em 13435 o modulo 7 pela mensagem   0 pediu a pausa e a prioridade? 10 3
? Em 13439 o modulo 3 pela mensagem 265 pediu a pausa e a prioridade? 10 2
? Em 13443 o modulo 4 pela mensagem 275 pediu a pausa e a prioridade? 10 2
? Em 13448 o modulo 5 pela mensagem 260 pediu a pausa e a prioridade? 10 2
? Em 13488 o modulo 6 pela mensagem  78 pediu a pausa e a prioridade? 10 3
? Em 13483 o modulo 3 pela mensagem 270 pediu a pausa e a prioridade? 10 3
? Em 13485 o modulo 4 pela mensagem 280 pediu a pausa e a prioridade? 10 3
? Em 13758 o modulo 5 pela mensagem 265 pediu a pausa e a prioridade? 10 4
? Em 13796 o modulo 3 pela mensagem 275 pediu a pausa e a prioridade? 10 4
? Em 13832 o modulo 4 pela mensagem 285 pediu a pausa e a prioridade? 10 4
? Em 13863 o modulo 5 pela mensagem 270 pediu a pausa e a prioridade? 10 4
? Em 13867 o modulo 6 pela mensagem  81 pediu a pausa e a prioridade? 10 4
* Em 13997 foi recebida a mensagem 267 que o modulo 5 enviou em 13838
* Em 14002 foi recebida a mensagem 268 que o modulo 5 enviou em 13841
* Em 14008 foi recebida a mensagem 269 que o modulo 5 enviou em 13852
* Em 14013 foi recebida a mensagem  80 que o modulo 6 enviou em 13863
* Em 14019 foi recebida a mensagem  80 que o modulo 6 enviou em 13863
* Em 14025 foi recebida a mensagem  80 que o modulo 6 enviou em 13863
* Em 14031 foi recebida a mensagem 276 que o modulo 3 enviou em 14031
* Em 14036 foi recebida a mensagem 286 que o modulo 4 enviou em 14036
```

FIGURA 6.11 - Execução do Exemplo 1 (continuação)

6.2. Exemplo 2

O exemplo 2 é formado por módulos dos tipos AIR e CIL. Os módulos são descritos no item 6.2.1, a configuração do exemplo é apresentada no item 6.2.2, finalmente no item 6.2.3 discute-se a execução da aplicação resultante.

6.2.1. Módulos Componentes

O módulo do tipo AIR, cujo código LPT é mostrado na Figura 6.12, possui uma única porta síncrona de saída (psi), através da qual envia mensagens do tipo ident e recebe mensagens do tipo BOOLEAN. As mensagens do tipo ident contêm o número de identificação do módulo emissor (campo módulo) o instante em que foram geradas (campo instante) e um número que identifica univocamente cada mensagem (campo num).

```
MODULE air;

    USE TipMen:ident;

    EXPORT
          ps1:ident REPLY BOOLEAN;


    MESSAGE
          men_ident:ident;
          men_resp:BOOLEAN;

    VAR
          i:INTEGER;

    BEGIN
          men_ident.modulo:=MODID;
          men_ident.num:=0;
          REPEAT
              men_ident.instante:=TIME;
              men_ident.num:=men_ident.num+1;
              SEND men_ident TO ps1 WAIT men_resp;
          UNTIL FALSE;
    END.
```

FIGURA 4.12 - Código LPN do módulo tipo AIR

O módulo do tipo AIR, cujo código LPN é mostrado na Figura 4.13, possui uma família de portas de entrada (pe1..pn) síncronas, através das quais recebe mensagens do tipo ident e envia mensagens de resposta do tipo BOOLEAN.

O ciclo de execução do módulo AIR é dividido em duas sequências de recepções seletivas. Na primeira sequência são recebidas k mensagens usando comandos de recepção seletiva priorizada, na segunda sequência são recebidas k mensagens usando comandos de recepção seletiva aleatória. Em ambas sequências, para cada mensagem recebida, é mostrado o instante em que ela foi recebida, bem como, o conteúdo dos seus campos, que contém a identificação do módulo emissor, o número da mensagem e o instante em que a mensagem foi gerada.

Na execução deste exemplo, as funções desempenhadas pelo módulo GERENTE do exemplo 1 foram incorporadas ao SIK. Esse incorporação procura dispensar o usuário de explicitamente ter que incluir o módulo GERENTE na configuração de suas aplicações. O módulo este que implementa funções inerentes ao SIK, adicionalmente, a incorporação do tratador das interrupções do kit ao SIK ajustante o desempenho deste em função da eliminação de mudanças de contexto necessárias à execução do GERENTE.

```
USE cir;

CONST
     n=10; { tamanho da familia de portas de entrada pe }
     k=30; { numero de recepcoes seletivas priorizadas/aleatorias }
     ff=12;{ 'form feed' }

se Tipos.idents;

NTRYPORT
     pe[1..n]:ident OF REPLY BOOLEAN;

ESSAGE
     men_ident:ident;
     men_resp:BOOLEAN;

AR
     i,j,num_modulo:INTEGER;
     inst:INTEGER;

EGIN
     num_modulo:=MOD.0;
     men_resp:=TRUE;
     REPEAT
        WRITELN(CHR(ff),'A partir deste instante a selecao sera priorizada ');
        FOR i:=1 TO k DO
          PSELECT
             FOR j:=1 TO n DO
                RECEIVE men_ident FROM pe[j]
                     => inst:=TIME;
                        WRITELN('* Em ',inst:4,' foi recebida a mensagem ',
                                men_ident.num:3,' que o modulo ',
                                men_ident.modulo:1,' enviou em ',
                                men_ident.instant:4);
                        REPLY men_resp TO pe[j]
          END;

        WRITELN(CHR(ff),'A partir deste instante a selecao sera aleatoria ');
        FOR i:=1 TO k DO
          RSELECT
             FOR j:=1 TO n DO
                RECEIVE men_ident FROM pe[j]
                     => inst:=TIME;
                        WRITELN('* Em ',inst:4,' foi recebida a mensagem ',
                                men_ident.num:3,' que o modulo ',
                                men_ident.modulo:1,' enviou em ',
                                men_ident.instant:4);
                        REPLY men_resp TO pe[j];
          END;
     UNTIL FALSE;
END.
```

FIGURA 4.13 - Código IPL do módulo tipo CIR

4.2.2. Configuração

O programa de configuração do exemplo 2, cujo diagrama de configuração é apresentado na Figura 4.14, é mostrado na Figura 4.15.

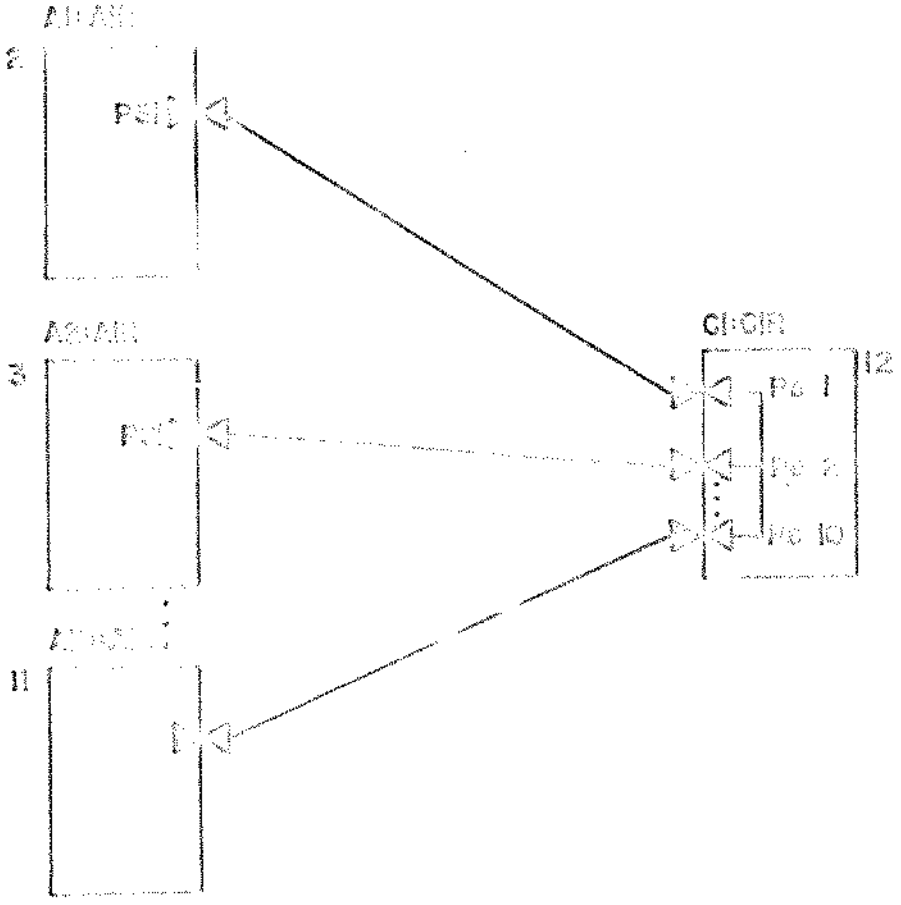

FIGURA 4.14 - Diagrama de configuração do Exemplo 2

A partir do tipo de módulo A1P são criadas as instâncias A1,A2,...,A10; a partir do tipo C1P é criada a instância C1. As portas de saída ps1, das instâncias A1,A2,...,A10, são conectadas as portas 1,2,...,10 da família pe, da instância C1.

```
CONFIGURATION.xcf exemplo2.cfg
        INSTANCE    A1,A2,A3,A4,A5,
                    A6,A7,A8,A9,A10:A10,
                    C1:C10,
        GROUP       A1,A2,A3,A4,A5,
                    A6,A7,A8,A9,A10,
                    C1
         LINK       A1.psl  TO C1.psE13
                    A2.psl  TO C1.psE23
                    A3.psl  TO C1.psE33
                    A4.psl  TO C1.psE43
                    A5.psl  TO C1.psE53
                    A6.psl  TO C1.psE63
                    A7.psl  TO C1.psE73
                    A8.psl  TO C1.psE83
                    A9.psl  TO C1.psE93
                    A10.psl TO C1.psE103
END.
```

Figura 6.15 - Procedure em LDM para a configuração do Exemplo 2

6.2.3. Execução

As Figuras 6.16 e 6.17 ilustram duas seqüências de processos distintas durante a execução do exemplo 2. No configuração do exemplo todas as instâncias com funcionalidade comum pertencem ao mesmo grupo. As instâncias A1,A2,...,A10 e C1 são do mesmo grupo.

No Figura 6.16 [illegible] instâncias de [illegible] bem como as [illegible] através da [illegible] Na Figura 6.17 [illegible] plo 2 [illegible] dos módulos [illegible]

```
A partir deste instante a selecao sera priorizada
* Em   26 foi recebida a mensagem  1 que o modulo 1 enviou em     1
* Em   57 foi recebida a mensagem  1 que o modulo 2 enviou em     1
* Em   8[illegible] foi recebida a mensagem  1 que o modulo 3 enviou em     1
* Em  119 foi recebida a mensagem  1 que o modulo 4 enviou em     1
* Em  150 foi recebida a mensagem  1 que o modulo 5 enviou em     2
* Em  181 foi recebida a mensagem  1 que o modulo 6 enviou em     2
* Em  212 foi recebida a mensagem  1 que o modulo 7 enviou em     2
* Em  2[illegible] foi recebida a mensagem  1 que o modulo 8 enviou em     2
* Em  27[illegible] foi recebida a mensagem  1 que o modulo 9 enviou em     3
* Em  3[illegible] foi recebida a mensagem  1 que o modulo 10 enviou em     3
* Em  33[illegible] foi recebida a mensagem  2 que o modulo 1 enviou em   33[illegible]
* Em  3[illegible] foi recebida a mensagem  2 que o modulo 2 enviou em   33[illegible]
* Em  [illegible] foi recebida a mensagem  2 que o modulo 3 enviou em   33[illegible]
* Em  [illegible] foi recebida a mensagem  2 que o modulo 4 enviou em   3[illegible]
* Em  [illegible] foi recebida a mensagem  2 que o modulo 5 enviou em   3[illegible]
* Em  [illegible] foi recebida a mensagem  2 que o modulo 6 enviou em   3[illegible]
* Em  5[illegible] foi recebida a mensagem  2 que o modulo 7 enviou em   3[illegible]
* Em  5[illegible] foi recebida a mensagem  2 que o modulo 8 enviou em   3[illegible]
* Em  5[illegible] foi recebida a mensagem  2 que o modulo 9 enviou em   33[illegible]
* Em  6[illegible] foi recebida a mensagem  2 que o modulo 10 enviou em   3[illegible]
* Em  6[illegible] foi recebida a mensagem  3 que o modulo 1 enviou em   [illegible]
* Em  6[illegible] foi recebida a mensagem  3 que o modulo 2 enviou em   6[illegible]
* Em  716 foi recebida a mensagem  3 que o modulo 3 enviou em   6[illegible]
* Em  747 foi recebida a mensagem  3 que o modulo 4 enviou em   6[illegible]
* Em  778 foi recebida a mensagem  3 que o modulo 5 enviou em   [illegible]
* Em  [illegible] foi recebida a mensagem  3 que o modulo 6 enviou em   [illegible]
* Em  8[illegible] foi recebida a mensagem  3 que o modulo 7 enviou em   [illegible]
* Em  872 foi recebida a mensagem  3 que o modulo 8 enviou em   [illegible]
* Em  903 foi recebida a mensagem  3 que o modulo 9 enviou em   [illegible]
* Em  93[illegible] foi recebida a mensagem  3 que o modulo 10 enviou em   [illegible]
```

[illegible] - A execução de uma seqüência de recepções seletivos priorizados no [illegible] 2

```
A partir deste instante a selecao sera aleatoria
* Em  986 foi recebida a mensagem  4 que o modulo 1 enviou em  987
* Em 1125 foi recebida a mensagem  4 que o modulo 5 enviou em  988
* Em 1158 foi recebida a mensagem  4 que o modulo 7 enviou em  989
* Em 1167 foi recebida a mensagem  4 que o modulo 6 enviou em  990
* Em 1219 foi recebida a mensagem  4 que o modulo 2 enviou em  987
* Em 1256 foi recebida a mensagem  4 que o modulo 10 enviou em  988
* Em 1288 foi recebida a mensagem  4 que o modulo 3 enviou em  987
* Em 1414 foi recebida a mensagem  4 que o modulo 4 enviou em  988
* Em 1445 foi recebida a mensagem  4 que o modulo 8 enviou em  989
* Em 1467 foi recebida a mensagem  4 que o modulo 9 enviou em  989
* Em 1510 foi recebida a mensagem  5 que o modulo 1 enviou em 1510
* Em 1508 foi recebida a mensagem  5 que o modulo 3 enviou em 1511
* Em 1508 foi recebida a mensagem  5 que o modulo 8 enviou em 1511
* Em 1508 foi recebida a mensagem  5 que o modulo 4 enviou em 1511
* Em 1508 foi recebida a mensagem  5 que o modulo 10 enviou em 1511
* Em 1508 foi recebida a mensagem  5 que o modulo 7 enviou em 1511
* Em 1511 foi recebida a mensagem  5 que o modulo 9 enviou em 1511
* Em 1833 foi recebida a mensagem  5 que o modulo 2 enviou em 1833
* Em 2024 foi recebida a mensagem  5 que o modulo 6 enviou em 1511
* Em 2107 foi recebida a mensagem  5 que o modulo 5 enviou em 1510
* Em 2130 foi recebida a mensagem  6 que o modulo 1 enviou em 2130
* Em 2176 foi recebida a mensagem  6 que o modulo 6 enviou em 2176
* Em 2215 foi recebida a mensagem  6 que o modulo 5 enviou em 2215
* Em 2248 foi recebida a mensagem  6 que o modulo 7 enviou em 2248
* Em 2310 foi recebida a mensagem  6 que o modulo 10 enviou em 2310
* Em 2324 foi recebida a mensagem  6 que o modulo 8 enviou em 2324
* Em 2404 foi recebida a mensagem  6 que o modulo 4 enviou em 2404
* Em 2437 foi recebida a mensagem  6 que o modulo 3 enviou em 2437
* Em 2459 foi recebida a mensagem  6 que o modulo 6 enviou em 2459
* Em 2491 foi recebida a mensagem  6 que o modulo 9 enviou em 2490
* Em 2522 foi recebida a mensagem  6 que o modulo 9 enviou em 2520
```

Figura 4.12 - A execução com a seqüência de numeração aleatória ativa alocação dinâmica

7. CONCLUSÃO

Este trabalho apresentou a implementação do suporte de tempo real (STR) de um ambiente para o desenvolvimento de sistemas de controle em tempo real. O software desenvolvido usando as ferramentas oferecidas pelo ambiente é altamente modular, reconfigurável e independente da arquitetura do hardware onde será executado. A versão atualmente implementada do STR suporta a configuração estática das aplicações e a sua execução em sistemas centralizados. Entretanto, a sua concepção permite a incorporação de mecanismos para reconfiguração dinâmica, bem como, a expansão para sistemas distribuídos.

A presente versão do ambiente aqui descrito está em fase de testes no Instituto de Automação do Centro Tecnológico para Informática (CTI) e será a seguir transportada para um hardware distribuído em fase de projeto. Esse hardware consistirá de estações de controle baseadas no barramento VME tendo como UCP o microprocessador M68000. A conexão física entre as estações será obtida mediante o uso de uma rede local de comunicações que implementará um protocolo de comunicações do tipo "passagem de permissão" ("token-passing"). O transporte do STR deverá ser simples já que cerca de 85% de suas linhas de código fonte são escritas em PASCAL e as restantes 15% compreendem rotinas bastante simples escritas em linguagem de montagem.

Paralelamente à implementação desta versão do ambiente estão sendo realizados estudos visando o desenvolvimento de um compilador para a LPM. Este compilador propiciaria grande flexibilidade na introdução de novos elementos na linguagem, bem como, facilitaria o transporte do ambiente para outros equipamentos. Outro aspecto em estudo refere-se à introdução de mecanismos para reconfiguração dinâmica da aplicação o que, no caso deste ambiente, apresenta perspectivas bastante interessantes em função das características modulares das aplicações e pelas possibilidades que a estrutura de interconexão de portas de entrada e saída oferece para a introdução de estratégias de reconfiguração. A implementação de construções na LCM e de serviços no STR para suportar a reconfiguração dinâmica é relativamente simples, entretanto, ainda é tema de pesquisa a validação do estado de sistemas distribuídos para a realização de tais reconfigurações, sendo necessário aguardar novos resultados dessas pesquisas oferecendo um maior suporte teórico à implementação desses mecanismos.

CAPÍTULO 7

CONCLUSÃO

ANEXO A

ANEXO A

PROGRAMAS PASCAL RESULTANTES DA PRÉ-COMPILAÇÃO DOS MÓDULOS DO EXEMPLO 1

Este anexo ilustra o processo de pré-compilação, apresentando os programas PASCAL resultantes da tradução dos módulos do exemplo 1, descrito na seção 6.1.

Nos programas PASCAL listados a seguir, as construções da LPM, traduzidas pelo pré-compilador, aparecem entre as cadeias de caracteres "(*" e "*)" seguidas das construções PASCAL resultantes da sua tradução. Da mesma maneira todas as linhas de comentários, introduzidas pelo pré-compilador, são delimitadas por essas sequências de caracteres.

O programa a (PROGRAM a) corresponde à tradução do módulo do tipo A (MODULE a), o programa b corresponde à tradução do módulo b e assim sucessivamente.

No programa a pode-se observar o tratamento dado pelo pré-compilador a diversas construções da LPM, por exemplo, o parâmetro formal pausa, do módulo a, é traduzido no programa a, pela variável global pausa e pela chamada ao serviço inipar. A definição de contexto (USE...), do módulo (USE...) provoca a cópia dos objetos referenciados, das unidades de definição especificadas (TipSis e TipMen) para o programa a. A declaração das portas de saída do módulo (EXITPORT...) servirá para o mapeamento dos nomes lógicos das portas em identificadores inteiros e para a criação, no qualificador do módulo, dos descritores das portas. As portas, de entrada ou saída, são mapeadas em ordem crescente na sequência em que são declaradas, em consequência, no programa a a porta ps1 será referenciada pelo valor inteiro 1 e a porta ps2 pelo valor inteiro 2. As mensagens declaradas no módulo a são traduzidas em variáveis do programa a, podendo em consequência ser usadas como qualquer outra variável do seu tipo. Embora o pré-compilador permita que as variáveis correspondentes às mensagens sejam usadas como variáveis normais a recíproca não é verdadeira, i.e., variáveis normais não podem ser usadas no lugar de variáveis correspondentes a mensagens nos comandos de comunicação e sincronização. Todos os procedimentos e funções correspondentes a serviços do STR usados pelo programa são declarados como externos. Na chamada aos serviços de comunicação do STR são empregadas as funções SIZEOF e ADS, que retornam o tamanho e o endereço, respectivamente, das mensagens/variáveis a enviar ou receber.

Conforme pode facilmente ser observado na listagem do programa c a construção da LCM que apresenta a tradução mais complexa é o comando de recepção seletiva. Ele requer inclusive a criação de duas variáveis auxiliares, for_sel e prt_sel, para receber do STR a valor da variável de controle do FOR (se houver) correspondente à alternativa do SELECT escolhida e o número da porta que recebeu a mensagem (se alguma tiver recebido).

```
(#MODULE a(pausa:INTEGER);#)
PROGRAM a;

  (#USE         Tipsis:PRIORITY;
        TipMen:ident,pausa_prio;#)
  TYPE
        PRIORITY=(SYSTEMPR,HIGHPR,NORMALPR,LOWPR,LOWESTPR);

        ident=RECORD
                modulo:INTEGER;
                instante:INTEGER4;
                num:INTEGER;

              END;

        pausa_prio=RECORD
                        pausa:INTEGER;
                        prio:PRIORITY;
                   END;

  (#EXITPORT
      ps1:ident REPLY pausa_prio;
        ps2:ident;#)


  VAR
        (# variaveis correspondentes aos parametros do modulo#)
        pausa:INTEGER;

  (#MESSAGE#)
        men_ident:ident;
        men_pausa_prio:pausa_prio;

        i:INTEGER;

#       interface dos servicos oferecidos pelo STR #)
PROCEDURE env_a(prt_saida,tam_men:INTEGER;end_men:ADSMEM); EXTERN;
PROCEDURE env_s(prt_saida,tam_men:INTEGER; end_men,end_resp:ADSMEM); EXTERN;
FUNCTION relogio:INTEGER4; EXTERN;
PROCEDURE retarda(period:INTEGER4); EXTERN;
PROCEDURE inipar(endpar:ADSMEM); EXTERN;
FUNCTION num_mod:INTEGER; EXTERN;
PROCEDURE muda_prio(prio:PRIORITY); EXTERN;

$PAGE+)
  BEGIN
        (# inicializacao dos parametros do modulo #)
        inipar(ADS(pausa));

        (#men_ident.modulo:=MODID;#)
        men_ident.modulo:=num_mod;

        men_ident.num:=0;
```

```
    REPEAT
       FOR i:=1 TO 4 DO
            BEGIN
               (#men_ident.instante:=TIME;#)
               men_ident.instante:=relogio;

               men_ident.num:=men_ident.num+1;
               (#SEND men_ident TO ps2;#)
               env_a(2,ORD(SIZEOF(men_ident)),ADS(men_ident));

               (#DELAY(pausa);#)
               retarda(pausa)

            END;
       (#men_ident.instante:=TIME;#)
       men_ident.instante:=relogio;

       men_ident.num:=men_ident.num+1;
       (#SEND men_ident TO ps1 WAIT men_pausa_prio;#)
       env_s(1,ORD(SIZEOF(men_ident)),ADS(men_ident),ADS(men_pausa_prio));

       pausa:=men_pausa_prio.pausa;
       (#SETPRIORITY(men_pausa_prio.prio);#)
       muda_prio(men_pausa_prio.prio);

    UNTIL FALSE;
ND.
```

```
(#MODULE b;#)
PROGRAM b;

   (#USE  TipSis:PRIORITY;
        TipMen:ident,pausa_prio;#)
   TYPE
        PRIORITY=(SYSTEMPR,HIGHPR,NORMALPR,LOWPR,LOWESTPR);

        ident=RECORD
                modulo:INTEGER;
                instante:INTEGER4;
                num:INTEGER;
              END;

        pausa_prio=RECORD
                        pausa:INTEGER;
                        prio:PRIORITY;
                   END;


   (#ENTRYPORT
        pe1:ident REPLY pausa_prio;#)

   (#EXITPORT
        ps1:ident REPLY pausa_prio;
        ps2:ident;#)

   VAR

   (#MESSAGE#)
        men_identA,men_identB:ident;
        men_pausa_prio:pausa_prio;

        i:INTEGER;

:#      interface dos servicos oferecidos pelo STR #)
'ROCEDURE env_a(prt_saida,tam_men:INTEGER;end_men:ADSMEM); EXTERN;
'ROCEDURE env_s(prt_saida,tam_men:INTEGER; end_men,end_resp:ADSMEM); EXTERN;
'ROCEDURE rec(end_men:ADSMEM; prt_entrada:INTEGER); EXTERN;
'ROCEDURE desvia(prt_saida,prt_entrada:INTEGER); EXTERN;
'UNCTION relogio:INTEGER4; EXTERN;
'UNCTION num_mod:INTEGER; EXTERN;
'ROCEDURE muda_prio(prio:PRIORITY); EXTERN;

   BEGIN
        (#men_identB.modulo:=MODID;#)
        men_identB.modulo:=num_mod;

        men_identB.num:=0;
        REPEAT
           FOR i:=1 TO 2 DO
                BEGIN
```

```
                (#RECEIVE men_identA FROM pe1;#)
                rec(ADS(men_identA),1);

                (#men_identB.instante:=TIME;#)
                men_identB.instante:=relogio;

                men_identB.num:=men_identB.num+1;
                (#FORWARD pe1 TO ps1;#)
                desvia(2,1);

                (#SEND men_identB TO ps2;#)
                env_a(3,ORD(SIZEOF(men_identB)),ADS(men_identB));

            END;
        (#men_identB.instante:=TIME;#)
        men_identB.instante:=relogio;

        men_identB.num:=men_identB.num+1;
        (#SEND men_identB TO ps1 WAIT men_pausa_prio;#)
        env_s(2,ORD(SIZEOF(men_identB)),ADS(men_identB),ADS(men_pausa_prio));

        (#SETPRIORITY(men_pausa_prio.prio);#)
        muda_prio(men_pausa_prio.prio);

    UNTIL FALSE;
END.
```

```
{#MODULE c(espera_sel:INTEGER);#}
PROGRAM c(INPUT,OUTPUT);

   {#USE         TipSis:PRIORITY;
         TipMen:ident,pausa_prio;#}
   TYPE
         PRIORITY=(SYSTEMPR,HIGHPR,NORMALPR,LOWPR,LOWESTPR);


         ident=RECORD
               modulo:INTEGER;
               instante:INTEGER4;
               num:INTEGER;
             END;

         pausa_prio=RECORD
                         pausa:INTEGER;
                         prio:PRIORITY;
                    END;

   {#ENTRYPORT
         pe1:ident REPLY pausa_prio;
         pe2:ident QUEUE 6;#}

   VAR

   {# variaveis correspondentes aos parametros do modulo #}
   espera_sel:INTEGER;

   {# variaveis auxiliares para os comandos PSELECT e RSELECT #}
   for_sel,prt_sel:INTEGER;

   {#MESSAGE#}
         men_req,men_ident:ident;
         men_pausa_prio:pausa_prio;

         i,num_modulo:INTEGER;
         inst:INTEGER4;
         prioridade:PRIORITY;

#        interface dos servicos oferecidos pelo STR #)
ROCEDURE ini_rec_sel(indice_for,clausula,prt_entrada:INTEGER;
                     end_men:ADSMEM); EXTERN;
ROCEDURE ini_temp_sel(indice_for,clausula:INTEGER; tempo:INTEGER4); EXTERN;
UNCTION sel(tipo:CHAR; clausula_else:INTEGER;
            VARS indice_for,prt_entrada:INTEGER):INTEGER; EXTERN;
ROCEDURE resp(prt_entrada,tam_men:INTEGER; end_men:ADSMEM); EXTERN;
UNCTION relogio:INTEGER4; EXTERN;
ROCEDURE inipar(endpar:ADSMEM); EXTERN;
UNCTION num_mod:INTEGER; EXTERN;
```

```
PROCEDURE muda_prio(prio:PRIORITY); EXTERN;

   PROCEDURE LePausaPrio(mo:INTEGER; inst:INTEGER4; num_men:INTEGER;
                         VAR pausa:INTEGER; VAR pri:PRIORITY);

   BEGIN
        WRITE('? Em ',inst:4,' o modulo ',mo:1,' pela mensagem ',
              num_men:3,' pediu a pausa e a prioridade? ');
        READLN(pausa,pri);
   END;

   BEGIN
        {# inicializacao dos parametros do modulo #}
        inipar(ADS(espera_sel));

        {#num_modulo:=MODID;#}
        num_modulo:=num_mod;

        REPEAT
           FOR i:=1 TO 40 DO
             {#PSELECT#}
             BEGIN

                 {#RECEIVE men_req FROM pe1#}
                 ini_rec_sel(0,1,1,ADS(men_req));

               {#OR#}

                 {#RECEIVE men_ident FROM pe2#}
                 ini_rec_sel(0,2,2,ADS(men_ident));

               {#OR#}

                 {#TIMEOUT espera_sel#}
                 ini_temp_sel(0,3,espera_sel);

               CASE sel('P',0,for_sel,prt_sel) OF
                 1:BEGIN
                      LePausaPrio(men_req.modulo,men_req.instante,
                                  men_req.num,men_pausa_prio.pausa,
                                  men_pausa_prio.prio);
                      {#REPLY men_pausa_prio TO pe1;#}
                      resp(prt_sel,ORD(SIZEOF(men_pausa_prio)),
                           ADS(men_pausa_prio));
                   END;
                 2:BEGIN
                      {#inst:=TIME;#}
                      inst:=relogio;

                      WRITELN('* Em ',inst:4,' foi recebida a mensagem ',
                              men_ident.num:3,' que o modulo ',
                              men_ident.modulo:1,' enviou em ',
                              men_ident.instante:4);
                   END;
```

```
                3:BEGIN
                     {#inst:=TIME;#}
                     inst:=relogio;

                     WRITELN('! Timeout em ',inst:4);
                   END;
              END;
            END;

          {#inst:=TIME;#}
          inst:=relogio;

          LePausaPrio(num_modulo,inst,0,espera_sel,prioridade);
          {#SETPRIORITY(prioridade);#}
          muda_prio(prioridade);

      UNTIL FALSE;

END.
```

```
(#MODULE d;#)
PROGRAM d;

   (#USE TipSis:PRIORITY;
         TipMen:ident,pausa_prio;#)

   TYPE
        PRIORITY=(SYSTEMPR,HIGHPR,NORMALPR,LOWPR,LOWESTPR);
        ident=RECORD
                 modulo:INTEGER;
                 instante:INTEGER4;
                 num:INTEGER;
               END;

        pausa_prio=RECORD
                        pausa:INTEGER;
                        prio:PRIORITY;
                   END;

   (#ENTRYPORT
        pe1:ident;#)

   (#EXITPORT
        ps1:ident;#)

(#      interface dos servicos oferecidos pelo STR #)
PROCEDURE env_a(prt_saida,tam_men:INTEGER;end_men:ADSMEM); EXTERN;
PROCEDURE rec(end_men:ADSMEM; prt_entrada:INTEGER); EXTERN;

   VAR

   (#MESSAGE#)
        men_ident:ident;

   BEGIN
        REPEAT
           (#RECEIVE men_ident FROM pe1;#)
           rec(ADS(men_ident),1);

           (#SEND men_ident TO ps1;#)
           env_a(2,ORD(SIZEOF(men_ident)),ADS(men_ident));

        UNTIL FALSE;
   END.
```

ANEXO B

A N E X O B

PROGRAMAS PASCAL RESULTANTES DA PRÉ-COMPILAÇÃO DOS MÓDULOS DO EXEMPLO 2

Este anexo apresenta os programas PASCAL resultantes da tradução dos módulos do exemplo 2, descrito na seção 6.2. Na análise dos programas apresentados aplicam-se as mesmas considerações, relativas ao processo de pré-compilação, feitas no Anexo A para os programas do exemplo 1.

O programa a1r corresponde à tradução do módulo do tipo A1R e o programa c1r corresponde à tradução do módulo do tipo C1R.

```
(#MODULE a1r;#)
PROGRAM a1r;

   (#USE TipMen:ident;#)
   TYPE

        ident=RECORD
                modulo:INTEGER;
                instante:INTEGER4;
                num:INTEGER;

              END;

   (#EXITPORT
        ps1:ident REPLY BOOLEAN;#)


   VAR

   (#MESSAGE#)
        men_ident:ident;
        men_resp:BOOLEAN;

        i:INTEGER;

#       interface dos servicos oferecidos pelo STR #)
ROCEDURE env_s(prt_saida,tam_men:INTEGER; end_men,end_resp:ADSMEM); EXTERN;
UNCTION relogio:INTEGER4; EXTERN;
UNCTION num_mod:INTEGER; EXTERN;

  BEGIN
        (#men_ident.modulo:=MODID;#)
        men_ident.modulo:=num_mod;

        men_ident.num:=0;
        REPEAT
           (#men_ident.instante:=TIME;#)
           men_ident.instante:=relogio;

           men_ident.num:=men_ident.num+1;
           (#SEND men_ident TO ps1 WAIT men_resp;#)
           env_s(1,ORD(SIZEOF(men_ident)),ADS(men_ident),ADS(men_resp));

        UNTIL FALSE;
  END.
```

```
LE clr;#)
M clr(INPUT,OUTPUT);

ST
  n=10; ( tamanho da familia de portas de entrada pe )
  k=30; ( numero de recepcoes seletivas priorizadas/aleatorias )
  ff=12;( 'form feed' )

SE TipMen:ident;#)
E
  ident=RECORD
          modulo:INTEGER;
          instante:INTEGER4;
          num:INTEGER;
        END;


NTRYPORT
  pe[1..n]:ident REPLY BOOLEAN;#)

      .

variaveis auxiliares para os comandos PSELECT e RSELECT #)
_sel:INTEGER;

ESSAGE#)
  men_ident:ident;
  men_resp:BOOLEAN;

  i,j,num_modulo:INTEGER;
  inst:INTEGER4;

  interface dos servicos oferecidos pelo STR #)
URE ini_rec_sel(indice_for,clausula,prt_entrada:INTEGER;
                end_men:ADSMEM); EXTERN;
URE ini_temp_sel(indice_for,clausula:INTEGER; tempo:INTEGER4); EXTERN;
ON sel(tipo:CHAR;clausula_else:INTEGER;
       VARS indice_for,prt_entrada:INTEGER):INTEGER; EXTERN;
URE resp(prt_entrada,tam_men:INTEGER; end_men:ADSMEM); EXTERN;
ON relogio:INTEGER4; EXTERN;
ON num_mod:INTEGER; EXTERN;

IN
  (#num_modulo:=MODID;#)
  num_modulo:=num_mod;

  men_resp:=TRUE;
  REPEAT
       WRITELN(CHR(ff),'A partir deste instante a selecao sera priorizada');
       FOR i:=1 TO k DO
       (#PSELECT#)
       BEGIN
          FOR j:=1 TO n DO
```

```
        (#RECEIVE men_ident FROM pe[j]#)

           ini_rec_sel(j,1,0+j,ADS(men_ident));

        CASE sel('P',0,j,prt_sel) OF
           1:BEGIN
                (#inst:=TIME;#)
                inst:=relogio;

                WRITELN('* Em ',inst:4,' foi recebida a mensagem ',
                        men_ident.num:3,' que o modulo ',
                        men_ident.modulo:1,' enviou em ',
                        men_ident.instante:4);
                (#REPLY men_resp TO pe[j];#)
                resp(prt_sel,ORD(SIZEOF(men_resp)),
                     ADS(men_resp));
             END;
        END;
      END;

      WRITELN(CHR(ff),'A partir deste instante a selecao sera aleatoria');
      FOR i:=1 TO k DO
      (#RSELECT#)
      BEGIN
         FOR j:=1 TO n DO
         (#RECEIVE men_ident FROM pe[j]#)

            ini_rec_sel(j,1,0+j,ADS(men_ident));

         CASE sel('R',0,j,prt_sel) OF
            1:BEGIN
                 (#inst:=TIME;#)
                 inst:=relogio;

                 WRITELN('* Em ',inst:4,' foi recebida a mensagem ',
                         men_ident.num:3,' que o modulo ',
                         men_ident.modulo:1,' enviou em ',
                         men_ident.instante:4);
                 (#REPLY men_resp TO pe[j];#)
                 resp(prt_sel,ORD(SIZEOF(men_resp)),
                      ADS(men_resp));
              END;
         END;
      END;
  UNTIL FALSE;
```

BIBLIOGRAFIA CONSULTADA

BIBLIOGRAFIA CONSULTADA

[AND 82] ANDREWS, G.R.: "The Distributed Programming Language SR: Mechanisms, Design and Implementation". Software - Pract. Exp., Vol. 12, 1982.

[AND 83] ANDREWS, G.R.; SCHNEIDER, F.B.: "Concepts and Notations for Concurrent Programming". Computing Surveys, Vol. 15, Nº 1, Mar. 1983.

[ANS 83] ANSI/MIL - STD 1815A: "The Ada Programming Language". American National Standards Institute, 1983.

[BOR 84] BORNAT, R.: "Imperative Languages in Distributed Computing". Distributed Computing Systems Program. Brighton, Sussex, England, Sept. 1984.

[FEL 79] FELDMAN, J.A.: "High Level Programming for Distributed Computing". Comm. ACM, Jun. 1979.

[HAN 75] HANSEN, P.B.: "The Programming Language Concurrent PASCAL". IEEE Trans. Software Eng., Jun. 1975.

[HAN 78] HANSEN, P.B.: "Distributed Processes: A Concurrent Programming Concept". Comm. ACM, Nov. 1978.

[HAN 81] HANSEN, P.B.: "Edison - a Multiprocessor Language". Software - Practice and Experience, Ud. 11, 1981.

[HOA 78] HOARE, C.A.R.: "Communicating Sequential Processes". Comm. ACM, Aug. 1978.

[HON 84] HONNEL, G.: "Language Constructs for Distributed Programs". Advanced Course on Distributed Systems". Institut fuer Informatik der Technischen Universitaet Munich, Germany, Apr. 1984.

[IBM 84a] "IBM-PC PASCAL Compiler Language Reference Version 2.00". Personal Computer Language Series, IBM Corp., 1984.

[IBM 84b] "IBM-PC PASCAL Compiler Fundamentals Version 2.00". Personal Computer Language Series, IBM Corp., 1984.

[IBM 84c] "IBM-PC Technical Reference". Personal Computer Hardware Reference Library, IBM Corp., 1984.

[IBM 84d] "IBM-PC Disk Operating System (DOS), Version 2.00". IBM Corp., 1984.

[ISO 83] ISO 7185: "Specification for Computer Programming Language PASCAL". International Standards Organization, 1983.

[KER 78] KERNIGHAN, B.W.: "The C Programming Language".Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1978.

[KNU 81] KNUTH, D.E.: "The Art of Computer Programming - Volume 2 Seminumerical Algorithms, 2d. ed.". Addison-Wesley, 1981.

[KRA 83] KRAMER, J.; MAGEE, J.; SLOMAN, N. & LISTER, A.: "CONIC: An Integrated Approach to Distributed Computer control Systems". IEEE Proc., Vol. 130, Pt. E, Nº 1, Jan. 1983.

[KRA 84] KRAMER, J.; MAGEE, J.; SLOMAN, M. et alii: "The CONIC Programming Language Version 2.4". Research Report. DOC 84/19, Department of Computing, Imperial College, London, Oct. 1984.

[KRA 85] KRAMER, J. & MAGEE, J.: "Dynamic Configuration for Distributed Systems". IEEE Trans. Software Eng., Apr. 1985.

[LES 82] LESK, M.E. & SCHMIDT, E.: "Lex - A Lexical Analyzer Generator". Bell Laboratories, Aug. 1982.

[LOP 85] LOPES, A.B. & ADÁN COELLO, J.M.: "Um Ambiente para o Projeto e Implementação de Software para Sistemas Distribuídos de Controle em Tempo Real". Anais do 2º Congresso Nacional de Automação Industrial (CONAI), São Paulo, Nov. 1985.

[LOP 86] LOPES, A.B.: "LPM E LCM: Linguagens para Programação e Configuração de Aplicações de Tempo-Real". Dissertação de Mestrado, Curso de Mestrado em Sistemas e Computação, Centro de Ciências e Tecnologia, Universidade Federal na Paraíba; Ago. 1986.

[MAD 74] MADNICK, J.E. & DONOVAN, J.J.: "Operating Systems". McGraw-Hill, 1974.

[MAG 84] MAGGE, J.N.: "Provision of Flexibility in Distributed Systems". Ph.D. Thesis, Department of Computing - Imperial College of Science & Technology, London, Apr. 1984.

[MAG 86] MAGALHÃES, M.F.: "Software para Tempo Real", Campinas, Editora da UNICAMP, 1986.

[MIC 83a] "Macro Assembler for MS-DOS Operations System". Microsoft Corp., 1983.

[MIC 83b] "Microsoft Link for 8086 and 8088 Microprocessors". Microsoft Corp., 1983.

[REC 80] RECTOR, R. & ALEXY, G.: "The 8086 Book". Berkeley, CA, Osborne, 1980.

[RIT 74] RITCHIE, D.M.& THOMPSON, K.: "The Unix time-sharing system". IEEE Transactions on Software Engineering, vol. SE-8, Nº 2, p137-146, 1982.

[SAR 84] SARGENT III, M. & SHOEMAKER, L.: "The IBM Personal Computer from the Inside Out". Addison-Wesley, 1984.

[STE 84] STEUSLOFF, H.U.: "Advanced Real Time Languages for Distributed Industrial Process Control". IEEE Computer, Feb. 1984.

[WIL 83] WILSON, P.: "OCCAM Architecture Eases System Design". Computer Design, Nov. 1983.

[WIR 77] WIRTH, N.: "Design and Implementation of Modula". Software - Pract. Exp., Vol. 7, 1977.

[WIR 82] WIRTH, N.: "Programming in Modula-2". New York, Heidelbers, Berlim: Springer-Verlag, 1982.

[YOU 82] YOUNG, S.J.: "Real Time Languages: Design and Development". ELLIS HORWOOD Ltd., 1982.