

ADILSON BARBOZA LOPES

LPM E LCM : LINGUAGENS PARA PROGRAMAÇÃO E
CONFIGURAÇÃO DE APLICAÇÕES DE TEMPO-REAL

Dissertação apresentada ao Curso de MESTRADO EM SISTEMAS E COMPUTAÇÃO da Universidade Federal da Paraíba, em cumprimento às exigências para obtenção do Grau de Mestre.

ÁREA DE CONCENTRAÇÃO : CIÊNCIA DA COMPUTAÇÃO

MAURÍCIO FERREIRA MAGALHÃES

Orientador

GIUSEPPE MONGIOVI

Co-Orientador

CAMPINA GRANDE

AGOSTO-1986

À Iracy, minha esposa,

e à Marina, minha filha.





L8641 Lopes, Adilson Barboza
 LPM e LCM : linguagens para programacao configuracao de
 aplicacoes em tempo-real / Adilson Barboza Lopes. - Campina
 Grande, 1986.
 142 f.

 Dissertacao (Mestrado em Sistemas e Computacao) -
 Universidade Federal da Paraiba, Centro de Ciencias e
 Tecnologia.

 1. Linguagem para Programacao 2. LPM - 3. LCM - 4.
 Dissertacao I. Magalhaes, Mauricio Ferreira, Dr. II.
 Mongiovi, Giuseppe, Prof. III. Universidade Federal da
 Paraiba - Campina Grande (PB)

CDU 004.438(043)

LPM E LCM: LINGUAGENS PARA PROGRAMAÇÃO
E CONFIGURAÇÃO DE APLICAÇÕES DE TEMPO-REAL


ADILSON BARBOZA LOPES

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DO CURSO DE
PÓS-GRADUAÇÃO EM SISTEMAS E COMPUTAÇÃO DA UNIVERSIDADE FEDE
RAL DA PARAÍBA COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A
OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M.Sc.).

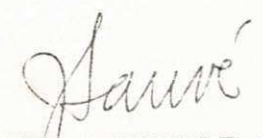
Aprovada por:


MAURÍCIO FERREIRA MAGALHÃES - Dr.

- Presidente -


GIUSEPPE MONGIOVI

-Co-Orientador -


JACQUES PHILIPPE SAUVÉ

- Examinador -

CAMPINA GRANDE
ESTADO DA PARAÍBA - BRASIL
AGOSTO - 1986

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

Aos Professores Maurício Ferreira Magalhães e Giusepe Mongiovi pela orientação, incentivo e dedicação com que acompanharam o desenvolvimento deste trabalho.

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

Ao colega Juan Manuel Adan Coello pela constante colaboração ao longo do trabalho e em particular pela implementação do núcleo tempo-real, sem o qual as linguagens não executariam.

As demais colegas da Divisão de Programas Básicos de Tempo-Real, Sérgio Donizetti Fischer e Aqueo Kamada, pelo incentivo e colaboração.

À Célia Maria Dorázio, pela dedicação na edição e diagramação deste texto.

Ao Roberto de Oliveira pelo esmero com que realizou os trabalhos de ilustração desta dissertação.

S U M Á R I O

1. INTRODUÇÃO.....	1
2. PROGRAMAÇÃO DE SISTEMAS DE TEMPO-REAL.....	4
2.1 Características de Sistemas de Tempo-Real.....	4
2.2 Concorrência em Sistemas de Tempo-Real.....	7
2.2.1 Comunicação e Sincronização Através de Memória Comparti- lhada.....	8
2.2.1.1 Semáforos.....	8
2.2.1.2 Monitores.....	11
2.2.2 Comunicação e Sincronização Através de Troca de Mensa- gens.....	14
2.2.2.1 Tipos de Sincronização.....	15
2.2.2.2 Comunicação Síncrona.....	15
2.2.2.3 Comunicação Assíncrona.....	20
3. LINGUAGENS PARA PROGRAMAÇÃO DE SISTEMAS DE TEMPO-REAL.....	22
3.1 Pascal Concorrente.....	23
3.2 Módulo.....	23
3.3 Módulo 2.....	24
3.4 Ada.....	25
3.5 PEARL.....	26
3.6 CONIC.....	26
3.7 Considerações Finais.....	27
4. AMBIENTE PARA DESENVOLVIMENTO DE APLICAÇÕES DE TEMPO-REAL.....	28

5.	ESPECIFICAÇÃO DAS LINGUAGENS DE PROGRAMAÇÃO E CONFIGURAÇÃO DE MÓDULOS.....	31
5.1	A Linguagem de Programação de Módulos - LPM.....	32
5.1.1	Vocabulário e Elementos Básicos da Linguagem.....	33
5.1.2	Definição de um Módulo.....	34
5.1.3	Importação de Definições.....	36
5.1.4	Declaração de Portas.....	36
5.1.5	Declaração de Mensagens.....	39
5.1.6	Comando para Especificação de um Ciclo Infinito.....	40
5.1.7	Comandos para Troca de Mensagens.....	40
5.1.7.1	Comando de Envio de Mensagens.....	41
5.1.7.2	Comando de Recepção de uma Mensagem.....	44
5.1.7.3	Comando de Desvio de uma Comunicação Síncrona.....	45
5.1.7.4	Comando de Recepção Seletiva.....	45
5.1.8	Unidades de Definição.....	48
5.1.9	Procedimentos e Funções Pré-Definidos.....	49
5.1.9.1	Operações Relacionadas à Troca de Mensagens.....	50
5.1.9.2	Mudança de Prioridade de um Módulo.....	51
5.1.9.3	Operações Relacionadas com o Tempo.....	51
5.1.9.4	Tratamento de Interrupções.....	52
5.1.9.5	Identificação de um Módulo em Tempo de Execução.....	53
5.2	A Linguagem de Configuração de Módulos - LCM.....	53
5.2.1	Vocabulário e Elementos Básicos da Linguagem.....	54
5.2.2	Estrutura de um Programa de Configuração.....	54
5.2.3	Declaração de Instâncias.....	55
5.2.4	Criação de Instâncias.....	56
5.2.5	Conexão de Instâncias.....	57
6.	CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO.....	59
6.1	Suporte de Tempo Real.....	60
6.1.1	Interface dos Serviços Oferecidos pelo Núcleo Tempo Real	61
6.1.1.1	Serviços de Troca de Mensagens.....	61

6.1.1.2	Serviços de Temporização.....	64
6.1.1.3	Serviço para a Inicialização dos Parâmetros Reais de um Módulo.....	65
6.1.1.4	Serviços para Tratamento Lógico de Interrupções.....	65
6.1.1.5	Serviço de Alteração de Prioridade de um Módulo.....	66
6.1.1.6	Serviço de Identificação de um Módulo.....	66
6.2	Aspectos Relacionados com a Portabilidade.....	66
6.3	O Tradutor LPM.....	67
6.3.1	O Processador de Unidades de Definição.....	69
6.3.2	O Pré-Compilador LPM.....	72
6.3.2.1	O Analisador Léxico.....	73
6.3.2.2	O Analisador Sintático.....	74
6.3.2.3	A Geração do Programa Pascal.....	76
6.3.2.3.1	Tradução do Comando SEND.....	77
6.3.2.3.2	Tradução do Comando LOOP.....	81
6.3.2.3.3	Tradução do Comando RECEIVE.....	81
6.3.2.3.4	Tradução do Comando SELECT.....	83
6.3.2.4	Recuperação de Erros.....	88
6.4	O Tradutor LCM.....	90
7.	CONCLUSÃO.....	94
	APÊNDICE A - DESCRIÇÃO BNF DAS LINGUAGENS LPM E LCM.....	97
	APÊNDICE B - SERVIÇOS OFERECIDOS PELO SUPORTE DE TEMPO-REAL (STR)..	103
	APÊNDICE C - UM EXEMPLO DO USO DAS LINGUAGENS.....	116
	BIBLIOGRAFIA CONSULTADA.....	138

L I S T A D E I L U S T R A Ç Õ E S

2.1 - Controle de processos por computador.....	5
2.2 - Solução do problema produtor/consumidor com semáforos.....	10
2.3 - Solução do problema produtor/consumidor com monitores.....	13
5.1 - Estrutura de um módulo LPM.....	34
5.2 - Parâmetros formais módulo.....	35
5.3 - Bloco LPM.....	35
5.4 - Importação de definições.....	36
5.5 - Declaração de portas.....	38
5.6 - Declaração de mensagens.....	39
5.7 - Comandos LOOP e EXIT.....	40
5.8 - Comando SEND.....	41
5.9 - Comando RECEIVE.....	44
5.10 - Comando REPLY.....	44
5.11 - Comando FORWARD.....	45
5.12 - Comando SELECT.....	46
5.13 - Estrutura de uma unidade de definição.....	49
5.14 - Estrutura de um programa LCM.....	55
5.15 - Declaração de instâncias.....	55
5.16 - Criação de instâncias.....	56
5.17 - Conexão de instâncias.....	57
6.1 - Estrutura funcional do tradutor LPM.....	67
6.2 - Estrutura e relação de dependência entre os componentes do tradutor LPM.....	68
6.3 - Esquema de implementação do processador de unidades de definição.....	70

6.4 - Dependência entre os passos envolvidos no processamento de uma unidade de definição.....	71
6.5 - Esquema de implementação do pré-compilador LPM.....	72
6.6 - Reconhecimento e tradução do comando SEND.....	80
6.7 - Reconhecimento e tradução do comando LOOP.....	81
6.8 - Reconhecimento e tradução do comando RECEIVE.....	82
6.9 - Reconhecimento e tradução do comando SELECT.....	87
6.10 - Estrutura funcional do tradutor LCM.....	90
6.11 - Mapa de configuração.....	91
C.1 - Código LPM do módulo tipo A.....	118
C.2 - A unidade de definição TipMen.....	119
C.3 - Código LPM do módulo tipo B.....	120
C.4 - Código LPM do módulo tipo C.....	121
C.5 - Código LPM do módulo tipo D.....	122
C.6 - Código LPM do módulo tipo TIMEMAN.....	123
C.7 - Diagrama de configuração do exemplo.....	124
C.8 - Programa em LCM para a configuração do exemplo.....	125

R E S U M O

Este trabalho apresenta a implementação de um ambiente orientado ao desenvolvimento de software de controle em tempo-real. O ambiente procura atender aos requisitos essenciais para a programação de sistemas distribuídos de Controle de Processos e consiste basicamente de uma metodologia e de duas linguagens: a Linguagem de Programação de Módulos - LPM e a Linguagem de Configuração de Módulos - LCM. No modelo adotado para o ambiente, o desenvolvimento de uma aplicação é constituído por duas etapas: a programação em LPM dos módulos que implementam as funções do sistema e a configuração em LCM da aplicação a partir dos módulos disponíveis. Esta característica possibilita a incorporação de mecanismos de reconfiguração dinâmica e tolerância a falhas. A comunicação entre módulos é feita através de troca de mensagens mediante uma interface constituída de portas lógicas de entrada e saída. A configuração de uma aplicação corresponde a um programa LCM que especifica os módulos componentes da aplicação e a interligação de suas portas. A implementação das linguagens foi realizada num computador PCS CADMUS-9200 através do uso das ferramentas YACC e LEX e está disponível para executar em ambiente compatível com IBM-PC. As linguagens são suportadas por um núcleo tempo-real cuja interface é apresentada na dissertação. Atualmente o ambiente suporta apenas configuração estática e processamento centralizado. Uma evolução consequente do trabalho é a extensão do ambiente de forma a possibilitar a execução distribuída de aplicações.

A B S T R A C T

This work presents the implementation of an environment appropriate for developing real time software. This environment is intended to cover the needs of distributed systems for process control. It integrates a design methodology and two languages: the Modules Programming Language - LPM, and the Modules Configuration Language - LCM. In this environment the development of applications involves two phases: the programming of LPM modules which implement the system functions, and the configuration of these modules using the LCM language. This facility allows the future incorporation of dynamic system reconfiguration and fault tolerance. Modules communicate by message passing through interfaces defined by entry and exit logical ports. The configuration program specifies the modules from which the application will be constructed and describes their ports interconnections. Both languages was implemented on PCS CADMUS-9200 computer using LEX and YACC tools and the generated code is available to run on IBM-PC compatible environment. The LPM and LCM languages are supported by a real time Kernel whose interface is herein presented. At present the environment developed supports only static configuration and centralized processing. A future goal of this work is to extend the proposed environment in order to handle distributed applications.

UNIVERSIDADE FEDERAL DA PARAIBA
Pró-Reitoria Para Assuntos do Interior
Coordenação Setorial de Pós-Graduação
Rua Aprígio Veloso, 882 - Tel (083) 321-7222-R 355
58.100 - Campina Grande - Paraíba

1. INTRODUÇÃO

Os recentes avanços tecnológicos possibilitam interligar diversos microcomputadores através de redes locais, formando sistemas distribuídos com desempenho comparável ao dos sistemas centralizados de maior porte, por um custo bastante inferior. Entretanto as ferramentas disponíveis para desenvolvimento de software estão orientadas a sistemas com memória compartilhada. Estas ferramentas são inadequadas ao desenvolvimento de programas constituídos por tarefas remotas que se comunicam através de troca de mensagens utilizando-se de um meio de comunicação sujeito a erros e atrasos. Por estas razões, as ferramentas orientadas para desenvolvimento de sistemas distribuídos vêm despertando grande interesse nos últimos anos.

Esta dissertação se insere neste contexto, onde são abordados os aspectos relacionados com o desenvolvimento de sistemas distribuídos de controle em tempo-real. Devido às características desses sistemas, a programação de uma aplicação envolve um alto grau de concorrência e paralelismo, tornando o processo de desenvolvimento bastante complexo. Além disso, a depuração e manutenção desses sistemas não é tarefa simples devido aos possíveis erros dependentes do tempo e da comunicação.

Neste sentido, procurou-se implementar um ambiente que atende aos requisitos essenciais para a programação de software distribuído de controle, tais como paralelismo real, contexto multi-tarefa com suporte tempo-real e capacidade de tolerância a falhas. O ambiente implementado é semelhante ao Sistema CONIC (KRAMER et alii, 1983), cujo projeto se encontra em andamento no Departamento de Computação do Imperial College (Londres) e consiste de um conjunto de ferramentas que facilita a estruturação, desenvolvimento, teste e configuração de aplicativos. 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, podendo interagir com outros módulos através do envio e recepção de mensagens, assíncronas e síncronas, mediante portas de comunicação.

Os módulos são programados usando-se 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 (ISO, 1983) que em adição às construções desta, permite:

- declarar portas de entrada e saída;
- declarar mensagens;
- enviar e receber mensagens através de portas;
- suspender a execução de um módulo por um período de tempo;
- mudar dinamicamente a prioridade de um módulo;
- tratar logicamente interrupções.

A execução das extensões do Pascal, bem como dos comandos de configuração da LCM é feita mediante o uso dos serviços oferecidos por um núcleo tempo-real (ADAN COELLO, 1986).

A implementação das linguagens LPM e LCM foi realizada numa estação PCS CADMUS-9200 através do uso das ferramentas YACC (JOHNSON, 1975) e LEX (LESK, 1979) e atualmente está disponível para ambiente compatível com IBM-PC.

No modelo adotado para o ambiente, o desenvolvimento de uma aplicação independe do fato da arquitetura ser centralizada ou distribuída, uma vez que o projetista configura a sua aplicação através da interconexão lógica dos módulos componentes, independentemente dos módulos residirem numa única estação ou estarem totalmente distribuídos.

Conseqüentemente uma aplicação pode ser desenvolvida e testada num ambiente centralizado e posteriormente ser executada em ambiente distribuído.

No decorrer desta dissertação, apresenta-se a descrição do ambiente adotado para desenvolvimento de aplicações de tempo-real e os principais aspectos relacionados com a sua implementação.

O Capítulo 2 descreve as principais características de um sistema de tempo-real, onde são discutidos os problemas clássicos que envolvem concorrência e apresentados alguns mecanismos que dão suporte a este tipo de programação.

No Capítulo 3 são apresentadas algumas linguagens orientadas ao desenvolvimento de programas concorrentes e as principais características de cada uma delas.

O Capítulo 4 apresenta o ambiente implementado neste trabalho, o qual procura atender aos requisitos essenciais à programação de software distribuído de controle.

No Capítulo 5 apresentam-se as especificações das linguagens LPM e LCM, as quais consistem no suporte ao ambiente adotado para desenvolvimento de aplicações de tempo-real, enquanto que o Capítulo 6 descreve as suas respectivas implementações.

Ao final, o Capítulo 7 apresenta as principais conclusões obtidas no desenvolvimento deste trabalho e faz algumas sugestões em termos de continuidade.

2. PROGRAMAÇÃO DE SISTEMAS DE TEMPO-REAL

A utilização de computadores nas mais variadas áreas tem sido ampliada a cada dia, em particular na área de controle de processos onde sistemas de tempo-real são usados com muita frequência. Este capítulo descreve as principais características de um sistema de tempo-real e apresenta alguns mecanismos que dão suporte a este tipo de programação.

2.1. Características de Sistemas de Tempo-Real

Um sistema de tempo-real é caracterizado por situações onde a atividade de processar uma dada informação, ou de responder a um estímulo de entrada externo, deve ocorrer num intervalo de tempo finito e determinado. No decorrer do texto, o termo tempo-real é utilizado no contexto de sistemas de controle e supervisão de processos, bem como a palavra tarefa é usada para representar o conceito de processo no sentido computacional.

A seguir, a Figura 2.1 apresenta um esquema típico para um sistema de controle de processos.

Neste exemplo o computador é usado para controlar a operação dos processos de uma planta industrial. A interface computador-processo é feita através de conversores AD/DA que obtém os dados do processo e exerce controle sobre os mesmos. Como os algoritmos de controle necessitam que o estado da planta seja analisado a intervalos de tempos regulares, o processamento normal deve ser interrompido pelo relógio de tempo-real a cada período de avaliação. O controle não é restrito a

interface com o processo, mas também com operadores, engenheiros de controle e gerentes, os quais poderão inicializar ou alterar as ações que determinam a configuração do sistema. Para isto é necessário que seja mantido um histórico das operações na planta em uma base de dados. Estas informações podem ser visualizadas por um conjunto de dispositivos, por exemplo, terminal gráfico, terminal de vídeo colorido, etc.

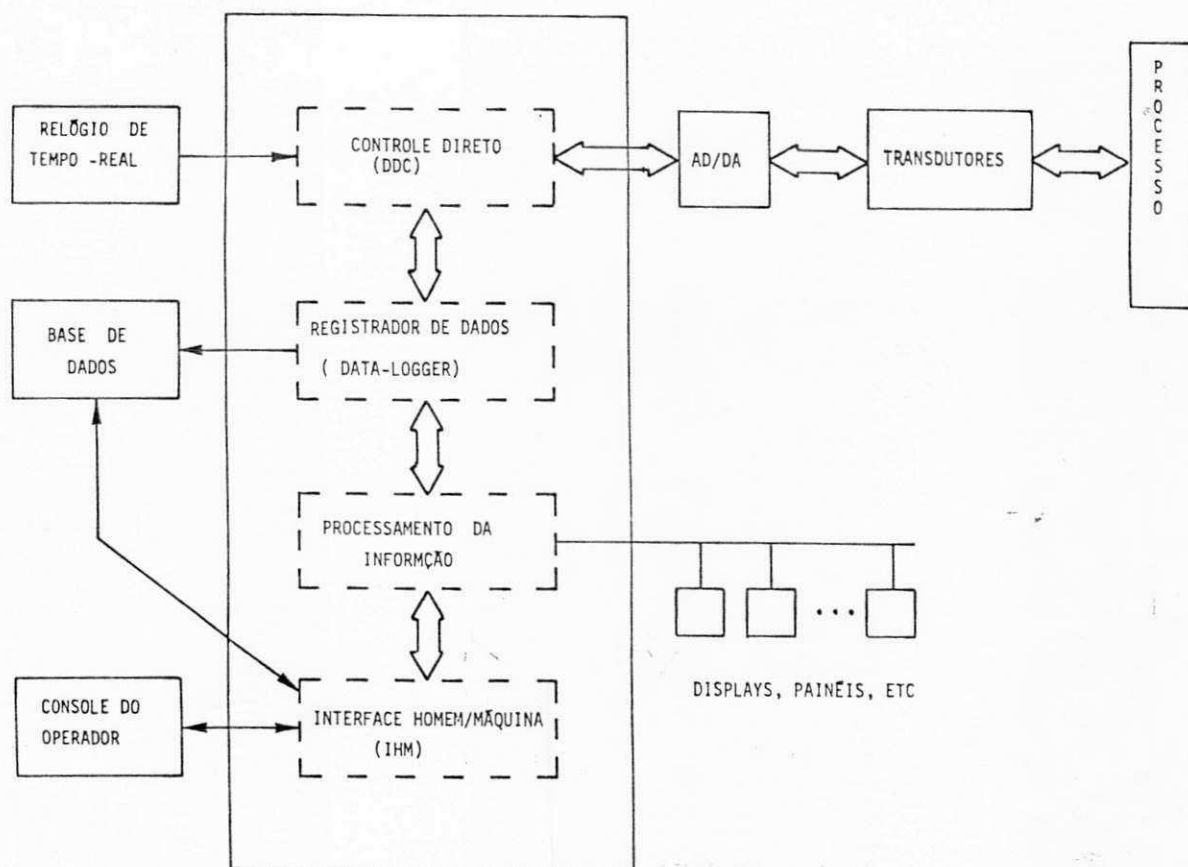


FIGURA 2.1 - Controle de processos por computador

O software do sistema correspondente ao exemplo apresentado é dividido em quatro funções básicas, as quais são identificadas a seguir:

- controle digital direto;
- registro de dados;
- gerenciamento de informação;
- interface com o operador.

Outras funções poderiam ser adicionadas ao sistema, como por exemplo, monitoramento de alarmes, otimização, supervisão, etc.

Cada uma dessas funções é representada em **software** como tarefas (processos no sentido computacional). As tarefas são executadas em "paralelo" e interagem quando necessário para sincronizar certos eventos e para trocar informações; a divisão do sistema em tarefas simplifica o seu projeto e desenvolvimento, bem como facilita o atendimento de suas requisições. Por exemplo, quando a tarefa DDC recebe uma solicitação de interrupção gerada pelo relógio de tempo-real, ela deve responder imediatamente avaliando o novo estado da planta e emitindo uma saída de controle conveniente. Esta tarefa de alta prioridade não deve ser ignorada pelo processamento de outra tarefa de prioridade mais baixa que eventualmente esteja sendo executada. Caso esse sistema fosse projetado como um único programa sequencial, a atribuição de tais prioridades às funções de processamento poderia ser muito mais complexa e difícil de se obter.

Uma vez fornecida esta introdução, apresenta-se a seguir as principais características dos sistemas de tempo-real:

- devem apresentar um alto nível de segurança; a sua falha pode provocar desde danos financeiros, no caso por exemplo, de controle de processos de produção numa linha de montagem de uma fábrica, chegando até a provocar catástrofes em termos de vida humana, como no caso de controle de estações de usinas nucleares;

- são geralmente grandes e complexos, o que implica num custo de desenvolvimento e manutenção bastante alto;
- devem responder a uma variedade de eventos externos, assegurando um tempo de resposta determinado;
- devem oferecer facilidades para interface com uma grande quantidade de dispositivos e periféricos;
- devem ser eficientes com relação à escolha de uma boa estratégia de uso dos recursos de **hardware** disponíveis.

2.2. Concorrência em Sistemas de Tempo-Real

Um sistema concorrente é caracterizado por permitir a existência de várias tarefas ativas simultaneamente. Em princípio todos os sistemas de tempo-real são concorrentes. A implementação da concorrência a nível de **hardware** é feita através das facilidades oferecidas pelo sistema de interrupção. Um exemplo clássico é uma operação de E/S, que envolve um procedimento de inicialização e é seguida por um procedimento de encerramento; durante esse período o dispositivo de E/S opera concorrentemente com o processador central e a sincronização é obtida através do uso do sistema de interrupção. A seguir apresentam-se alguns aspectos relativos à comunicação e sincronização entre tarefas.

2.2.1. Comunicação e Sincronização Através de Memória Compartilhada

Sistemas com memória compartilhada são caracterizados por utilizarem mecanismos de comunicação e sincronização através de dados compartilhados. Nestes sistemas a cooperação entre tarefas é feita através de variáveis comuns, onde o suporte operacional fornece procedimentos para garantir a exclusão mútua no acesso a essas variáveis, as quais podem ser usadas para representar o estado de recursos físicos compartilhados, podendo ser chamadas em geral de recursos.

O conceito de exclusão mútua é fundamental para este tipo de sistemas, e pode ser melhor elucidado através do exemplo clássico do produtor/consumidor: suponha a existência de duas tarefas, onde a primeira gera uma sequência de dados (produtor) a ser transferida para a segunda (consumidor). Considerando que a produção dos dados ocorre concorrentemente com o consumo, a sequência de eventos é totalmente aleatória e deve apenas ser limitada pelo tamanho do buffer onde os dados são armazenados temporariamente. O problema pode ocorrer nos acessos concorrentes ao buffer de forma que os resultados podem se tornar totalmente inconsistentes. Os mecanismos de comunicação e sincronização apresentados nos itens seguintes resolvem este problema através da proibição de acesso simultâneo de tarefas a variáveis compartilhadas, o que é denominado de exclusão mútua.

2.2.1.1. Semáforos

Um semáforo (DIJKSTRA, 1968) é uma variável binária que guarda o estado do recurso protegido, ou seja, informa sua disponibilidade. Por exemplo, para um dado semáforo S, a atribuição do valor um ao semáforo indica que o recurso correspondente torna-se disponível, e assim perma-

nece até que seja solicitado por alguma tarefa, quando então deverá ocorrer uma transição para o estado não-disponível. Desta forma, se uma tarefa deseja utilizar o recurso protegido, ela deve esperar até que a condição recurso disponível seja obtida. Para evitar a constante utilização do processador na avaliação da condição de disponibilidade ($S=1$), a tarefa deverá ser suspensa e colocada numa fila de espera associada ao semáforo. A primitiva `SIGNAL` é usada neste caso pela tarefa produtora para informar a ocorrência do evento.

São fornecidas duas operações de sincronização na utilização de semáforos: `SIGNAL(e)` e `WAIT(e)`. A semântica das operações é descrita a seguir:

- `SIGNAL(e)`:

se fila (e) está vazia

então $e:=1$

senão retorna primeira tarefa da fila de e

- `WAIT(e)`:

se ($e=1$)

então $e:=0$

senão coloque a tarefa na fila de e.

A Figura 2.2 apresenta uma solução para o problema do produtor/consumidor, onde as operações de acesso ao `buffer` não são detalhadas no texto; assume-se que para colocar um dado no `buffer` dispõe-se da operação `PUT (x)` e para recuperá-lo, utiliza-se a operação `GET (x)`, e que existe as funções `CHEIO` e `VAZIO`, as quais retornam o valor `TRUE` se o `buffer` estiver cheio ou vazio, respectivamente. O controle do recurso é feito pelas seguintes funções:

```

PROGRAM produtor_X_consumidor1;
VAR   b: BUFFER;
      s: SEMÁFORO;
      não_cheio, não_vazio: SINAL;
PROCEDURE produtor;
  VAR x: DADO;
BEGIN
  REPEAT
    produzir (x);
    segurar (s);
    IF CHEIO
    THEN BEGIN
      liberar (s);
      WAIT (não_cheio);
      segurar (s);
    END;
    PUT (x);
    liberar (s);
    SIGNAL (não_vazio);
  FOREVER
END;
PROCEDURE consumidor,
  VAR x: DADO;
BEGIN
  REPEAT
    segurar (s);
    IF VAZIO
    THEN BEGIN
      liberar (s);
      WAIT (não_vazio);
      segurar (s);
    END;
    GET (x);
    liberar (s);
    SIGNAL (não_cheio)
  FOREVER
END;
BEGIN (* programa principal *)
  s:=1;
  COBEGIN
    produtor;
    consumidor;
  COEND
END.

```

FIGURA 2.2 - Solução do problema produtor/consumidor com semáforos

- Segurar (S):
 - se S=1
 - então S:=0
 - senão suspender a tarefa e colocá-la na fila de S;

- Liberar (S):
 - se fila (S) está vazia
 - então S:=1
 - senão ativar a primeira tarefa da fila de S.

A codificação do exemplo é feita em PASCAL concorrente onde as tarefas concorrentes são delimitadas no bloco por COBEGIN e COEND.

O uso de semáforos envolve primitivas de baixo nível, logo são passíveis de obscuridade, o que pode causar uma certa insegurança em sua utilização, e a verificação de correção no acesso a variáveis compartilhadas é bastante dificultada, uma vez que a compilação de cada tarefa é feita independentemente do conhecimento do ambiente de concorrência.

2.2.1.2. Monitores

O monitor (HOARE, 1978) é um módulo no qual os dados compartilhados são agrupados e associados a procedimentos que podem ser chamados pelas tarefas. Num determinado instante apenas uma tarefa poderá estar executando um procedimento do monitor.

Um monitor pode ser usado tanto para comunicação como para sincronização de tarefas, e funciona também como instrumento de estruturação de programas. Em sua definição são especificadas as declarações das variáveis, a seguir o conjunto de procedimentos que garantem a exclusão mútua no acesso a essas variáveis e por último, um corpo principal onde

são fornecidos valores iniciais às variáveis do monitor.

Em caráter de ilustração, apresenta-se através da Figura 2.3, uma nova solução para o problema do produtor/consumidor, usando desta vez a construção monitor.

A solução apresentada usando monitores é muito mais simples e legível do que a anteriormente mostrada. Tudo se resume na ativação de um procedimento do monitor, cujos acessos ao buffer compartilhado está sob a sua total responsabilidade.

A sincronização de tarefas no monitor é obtida através das primitivas **WAIT** e **SIGNAL**. Se uma tarefa obtém um acesso ao monitor e emite uma operação **WAIT**, a regra da exclusão mútua é renunciada e outra tarefa poderá entrar no monitor. A tarefa que está esperando, somente poderá ser ativada quando receber um sinal procedente da tarefa ativa; quando isto ocorre, as duas tarefas ficam ativas no monitor, portanto para garantir a exclusão mútua, deve-se adotar que o **SIGNAL** seja a última operação do monitor a ser executada.

As facilidades de manipulação de monitores decorrem de sua característica modular; desde que todos os acessos a dados compartilhados são executados por procedimentos do monitor, é possível construí-lo e testá-lo independentemente da tarefa que o utiliza. Portanto, um compilador pode verificar se os acessos aos dados compartilhados são legais, bem como algumas situações de **dead-lock** podem ser detectadas através da construção de grafos de acessos. Isto é possível uma vez que as primitivas **SIGNAL** e **WAIT** têm o seu escopo delimitado pelas fronteiras do monitor.

Em contrapartida às vantagens citadas, monitores apresentam algumas restrições. Por exemplo, o controle retornado na saída do monitor poderá ativar no máximo uma tarefa, entretanto, é comum em sistemas de controle a existência de uma tarefa que necessita sincronizar várias outras..


```

PROGRAM produtor_X_consumidor;
  MONITOR buffer_limitado;
  VAR b: BUFFER;
      não_cheio, não_vazio : SIGNAL;
  PROCEDURE armazenar (VAR x : DADO)
  BEGIN
    IF CHEIO
    THEN WAIT(não_cheio);
    PUT(x);
    SIGNAL(não_vazio);
  END;
  PROCEDURE retirar (VAR x : DADO)
  BEGIN
    IF VAZIO
    THEN WAIT(não_vazio);
    GET (x);
    SIGNAL(não_cheio);
  END;
  BEGIN (* corpo do monitor *)
    inicializa b;
  END;
  PROCEDURE produtor;
  VAR x : DADO;
  BEGIN
    REPEAT
      produzir (x);
      armazenar (x);
    FOREVER
  END;
  PROCEDURE consumidor;
  VAR x : DADO;
  BEGIN
    REPEAT
      retirar (x);
      consumir (x);
    FOREVER
  END;
  BEGIN (* programa principal *)
  COBEGIN
    produtor;
    consumidor
  COEND
  END.

```

FIGURA 2.3 - Solução do problema produtor/consumidor com monitores

O uso de monitores é bastante conveniente em sistemas com memória compartilhada, porém se mostra inadequado para sistemas com processamento distribuído; considerando que os custos de hardware têm decrescido nos últimos anos em função do constante avanço tecnológico da microeletrônica, os sistemas distribuídos estão se tornando bem mais atrativos, o que justifica a adoção de novos mecanismos para comunicação e sincronização de tarefas.

2.2.2. Comunicação e Sincronização Através de Troca de Mensagens

Os mecanismos de comunicação entre tarefas através de memória compartilhada impõem uma dependência muito grande no desenvolvimento de uma aplicação em relação à estrutura de hardware utilizada. Isto ocorre porque implicitamente o projeto das linguagens de alto nível refletem em suas construções as características do ambiente de multiprogramação. Ainda em relação a este fato, a experiência tem demonstrado que o enfoque mais natural para comunicação entre tarefas é o mecanismo de troca de mensagens, uma vez que transmissão de dados e sincronização constituem atividades inseparáveis (HANSEN, 1978).

Os mecanismos de comunicação e sincronização através de trocas de mensagens constituem-se no envio (SEND) de mensagens e na recepção (RECEIVE) de mensagens. A comunicação é caracterizada pela troca de informações entre tarefas, enquanto que a sincronização é identificada através da ordem em que os eventos ocorrem. Por exemplo, uma mensagem somente pode ser recebida após ter sido enviada.

O envio de uma mensagem ocorre através da execução de um comando

SEND mensagem TO destino

enquanto que o recebimento de uma mensagem dá-se pela execução do comando

RECEIVE mensagem FROM origem.

Os itens seguintes abordam os aspectos semânticos relacionados com os tipos de sincronização envolvidos nas primitivas para troca de mensagens.

2.2.2.1. Tipos de Sincronização

As primitivas para troca de mensagens podem ser implementadas de forma bloqueante ou não-bloqueante. Uma primitiva é dita não-bloqueante quando a tarefa que a executa continua a sua execução independentemente da ocorrência de qualquer evento. Diz-se então, neste caso, que a operação correspondente é assíncrona. No caso de uma primitiva síncrona, a tarefa que executa a operação correspondente poderá ficar bloqueada até a ocorrência do evento. Por exemplo, na execução de uma primitiva SEND síncrona, a tarefa que envia a mensagem fica suspensa até que a mensagem seja recebida pela tarefa destino.

2.2.2.2. Comunicação Síncrona

O mecanismo de troca de mensagens baseado no uso de primitivas síncronas pode ser classificado em três tipos, discutidos a seguir:

a) rendezvous simétrico

Neste tipo de comunicação, a tarefa emissora da mensagem é bloqueada até que a tarefa receptora esteja pronta para receber a mensagem. Caso a tarefa receptora execute a primitiva de recepção sem que a mensagem aguardada esteja disponível, a tarefa fica suspensa até a chegada da mensagem. Após a comunicação, as tarefas fonte e destino continuam a execução normalmente. Percebe-se neste caso que o mecanismo rendezvous é simétrico no sentido de que a tarefa fonte deve especificar a tarefa destino e vice-e-versa. Para casos práticos de linguagem de tempo-real isto é restritivo uma vez que é impossível escrever uma tarefa servidora que conheça os nomes de todas as tarefas clientes.

A caráter de ilustração, apresenta-se a seguir um exemplo que mostra a tarefa produtor enviando uma mensagem mens para a tarefa consumidor.

```

.
.
TASK produtor;
BEGIN
    VAR mens : MENSAGEM;
    WHILE true DO
        BEGIN
            .
            .
            SEND mens TO consumidor
            .
            .
        END
    END
END
TASK consumidor;
BEGIN
    VAR mens : MENSAGEM;
    WHILE true DO
        BEGIN
            .
            .
            RECEIVE mens FROM produtor;
            .
            .
        END
    END
END

```

b) rendezvous assimétrico

Esta forma de interação constitui uma alternativa para a comunicação simétrica fornecida pelo mecanismo anterior. Neste tipo de comunicação apenas uma tarefa (cliente) referencia a outra (servidora) na execução do rendezvous. Esta abordagem foi adotada no projeto da linguagem ADA.

O rendezvous assimétrico é representado na linguagem pela inclusão do comando ACCEPT nas tarefas servidoras. Este comando possui a forma de um procedimento de entrada que permite a tarefa cliente comunicar-se com a tarefa servidora através da execução do comando ACCEPT correspondente. O exemplo anterior pode ser reescrito com o mecanismo rendezvous assimétrico da seguinte forma:

```
TASK produtor;  
BEGIN  
    VAR x : MENSAGEM;  
    .  
    .  
    consumidor.SEND (x)  
    .  
    .  
END  
TASK consumidor  
BEGIN  
    VAR y : MENSAGEM;  
    .  
    .  
    ACCEPT SEND (IN item : MENSAGEM);  
        y:=item  
    END; (* fim do corpo do accept *)  
    .  
    .  
END
```

A transferência dos dados ocorre da mesma maneira que em uma chamada normal de procedimento, onde os parâmetros reais fornecidos na chamada do procedimento correspondem aos parâmetros formais especificados no comando `ACCEPT`.

A formulação do **rendezvous** assimétrico introduz alguns conceitos novos. Primeiramente, a transferência dos dados é especificada através dos parâmetros do procedimento. Os parâmetros podem ser especificados como `IN`, `OUT` ou `INOUT`, permitindo portanto um único **rendezvous** resultar numa transferência de dados bidirecional. Outro aspecto interessante é o fato que o corpo do comando `ACCEPT` é executado como uma região crítica. Entretanto o maior benefício obtido por este tipo de mecanismo é uma maior clareza no entendimento de programas concorrentes complexos, e uma maior facilidade no desenvolvimento de provas de correção de programas, devido ao fato de todas as tarefas necessitarem estabelecer uma sincronização em sua interação.

c) rendezvous seletivo

Esta forma de interação é uma extensão aos mecanismos anteriores, onde permite-se uma seleção não determinística para os comandos `ACCEPT`. Este mecanismo foi concebido para suportar a manipulação de operações assíncronas, o que possibilita a introdução de paralelismo em sua utilização. Um exemplo desta proposta pode ser visto a seguir, onde uma variável chamada `dado_comum` pode ser manipulada livremente por um conjunto de tarefas clientes, as quais devem ler ou escrever na variável com a garantia de que os acessos realizem-se sob a forma de exclusão mútua. Este problema é tratado no exemplo alocando-se a variável a uma tarefa cuja única função é protegê-la.

```

TASK variável_compartilhada;
BEGIN
  VAR dado_comum : DADO;
  BEGIN
    SELECT
      ACCEPT leitura (OUT x : DADO);
      x := dado_comum;
    END
    OR
      ACCEPT escrita (IN x : DADO);
      dado_comum :=x;
    END
  END SELECT
END
END

```

Esta tarefa aceita chamadas para ler ou escrever em variáveis compartilhadas em qualquer ordem. Num dado instante que o comando `SELECT` for executado, uma das quatro possibilidades pode ocorrer:

- existência de uma chamada pendente ao procedimento leitura;
- existência de uma chamada pendente ao procedimento escrita;
- existência de chamadas pendentes para os dois procedimentos;
- nenhuma chamada pendente para os dois procedimentos.

Nos dois primeiros casos, o `ACCEPT` correspondente é imediatamente executado. No terceiro caso, um comando `ACCEPT` é escolhido aleatoriamente e executado, já no último caso, a tarefa é suspensa até que seja efetuada uma chamada para a leitura ou escrita, implicando nesse instante na execução do `ACCEPT` associado. Em algumas situações uma condição extra pode ser necessária de modo a refletir o estado de estruturas de dados internas à tarefa servidora. Estas condições são introduzidas nos comandos `ACCEPT` internos a um comando `SELECT` tendo como prefixo uma *guarda*. O estado da *guarda* é representado pelo valor de uma expressão booleana de maneira que a *guarda* estará aberta para o valor

verdadeiro da expressão e estará fechada no caso de um valor falso. Quando uma tarefa servidora entra num comando **SELECT**, todas as **guardas** presentes são avaliadas, e somente os comandos **ACCEPT** precedidos por uma **guarda** aberta são candidatos a seleção.

O mecanismo **rendezvous** fornece uma solução clara para os problemas de multiprogramação em contrapartida às soluções anteriores com semáforos e monitores. Este mecanismo é também adequado a sistemas distribuídos, sistemas de multiprocessamento com memória compartilhada e a arquiteturas baseadas em monoprocessamento.

2.2.2.3. Comunicação Assíncrona

UNIVERSIDADE FEDERAL DA PARAÍBA
Pró-Reitoria Para Assuntos do Interior
Coordenação Setorial de Pós-Graduação
Rua Aprigio Veloso, 882 - Tel (083) 321-7222-R 355
58.100 - Campina Grande - Paraíba

Uma comunicação assíncrona é caracterizada pelo não bloqueio das tarefas que executam as primitivas. Assim, por exemplo, uma tarefa emissora ao executar uma primitiva **SEND** assíncrona, continua a sua execução imediatamente após a operação de envio de mensagem. Uma das vantagens no envio não bloqueante de mensagens é a implementação de mensagens do tipo difusão múltipla (**broadcasting**), onde esta é enviada a vários destinatários simultaneamente. Similarmente, a tarefa que executa a primitiva **RECEIVE** assíncrona, não é bloqueada caso naquele instante, a mensagem não se encontre disponível.

A grande vantagem da comunicação totalmente assíncrona é que o paralelismo na execução das tarefas é maximizado, contrariamente à comunicação síncrona, onde a necessidade de bloqueio das tarefas reduz o grau de paralelismo. Em contrapartida, este tipo de comunicação apresenta um inconveniente introduzido pelo tratamento de filas de mensagens. Isto pode ocorrer porque sendo as tarefas cooperantes totalmente assíncronas, é possível uma determinada tarefa enviar uma série de mensagens para uma outra, sem que esta esteja pronta para recebê-las; neste caso as mensagens devem ser guardadas em uma fila até que a tarefa

destinatária efetue as recepções pendentes.

O tratamento a ser dado às filas para armazenamento de mensagens é crítico pelas consequências que pode causar ao funcionamento de um sistema real. Não é admissível, por exemplo, que as filas cresçam arbitrariamente em função do número de mensagens, implicando portanto numa limitação do tamanho das filas de mensagens associadas às tarefas. A forma mais usual é dimensionar o tamanho das filas através de uma especificação do programador.

Em controle de processos, por exemplo, é muito comum a monitoração de alarmes ou leitura de sensores, onde normalmente a mensagem importante é a última recebida, uma vez que ela traduz o estado mais recente de uma determinada componente do processo controlado. Neste caso é suficiente dimensionar a área de mensagens com capacidade unitária.

3. LINGUAGENS PARA PROGRAMAÇÃO DE SISTEMAS DE TEMPO-REAL

As linguagens voltadas para a programação de aplicações de tempo-real devem suportar o projeto de programas bem estruturados e modulares, bem como fornecer mecanismos que ajudem ao desenvolvimento de programas corretos e confiáveis. Neste sentido, uma linguagem orientada a este tipo de problema deve atender aos requisitos do tipo modularidade, paralelismo real, contexto multi-tarefa com suporte tempo-real e capacidade de tolerância a falhas.

Muito embora a maioria das aplicações de tempo-real sejam programadas atualmente em linguagens com estrutura sequencial, o uso de linguagens com características mais avançadas, as quais refletem em sua definição os aspectos conceituais relativos a arquiteturas distribuídas, é bem mais adequado, além do que, favorece a implementação de verificação de correção de programas. Isto deve-se ao fato de que uma construção em uma linguagem constitui-se num elemento bem definido relativamente a sua sintaxe e semântica. Já o uso de linguagens convencionais em aplicações de tempo-real implica na criação de novas operações, por parte do programador, onde muitas delas são implementadas através de procedimentos, cuja semântica associada foge ao escopo do processador da linguagem.

Este capítulo apresenta algumas linguagens orientadas ao desenvolvimento de programas concorrentes e as principais características de cada uma delas.

3.1. Pascal Concorrente

O Pascal concorrente (HANSEN, 1977) é uma extensão do Pascal sequencial onde um programa consiste basicamente de três componentes básicos: tarefas, monitores e classes. As tarefas se comunicam através de monitores que representam estruturas de dados compartilhadas.

Classes são componentes passivos tal como monitores, correspondendo a sua definição à especificação de uma estrutura de dados e às operações sobre ela. O acesso exclusivo de uma tarefa aos componentes de uma classe pode ser garantido completamente em tempo de compilação, tornando conseqüentemente as chamadas aos procedimentos das classes bem mais rápidas que as chamadas de monitores.

3.2. Módulo

A linguagem Módulo (WIRTH, 1977) é baseada no Pascal, sendo adicionado à estrutura deste uma nova estrutura chamada módulo. Um módulo é um conjunto de procedimentos, tipos de dados e variáveis, onde o programador tem controle sobre os objetos que são importados e exportados.

Módulo é destinada principalmente à programação de sistemas de computação dedicados, incluindo controle de processos em máquinas pequenas. A linguagem possui três facilidades adicionais para multiprogramação: tarefas, módulos-interface e primitivas de sincronização ou sinais.

O conceito de módulo-interface é similar ao de monitor, sendo usado na cooperação entre tarefas para garantir a exclusão mútua nos acessos simultâneos, por parte das tarefas, aos objetos comuns.

Módulo oferece uma outra característica chamada módulo-dispositivo, a qual permite ao programador ter um controle mais efetivo sobre

os dispositivos periféricos do sistema.

As maiores críticas em relação a Módulo recaem nos aspectos relacionados às facilidades de contexto multi-tarefa. Uma vez que nenhuma tarefa pode referenciar outra, Módulo torna-se inviável para a implementação de supervisores, onde uma tarefa supervisora precisa manter estrito controle sobre as demais tarefas supervisionadas. Isto significa que Módulo não pode ser usado em sistemas onde possam ocorrer falhas, em particular, Módulo não pode ser usado para implementar sistemas operacionais de propósitos gerais.

3.3. Módulo 2

A linguagem Módulo 2 surgiu em consequência da necessidade de uma linguagem mais geral, flexível e eficaz para a programação de microcomputadores. As restrições de Módulo foram consideradas tão sérias que em Módulo 2 o conceito de tarefas foi substituído por corotinas (WIRTH, 1980), incorporando desta forma uma componente dinâmica, onde várias instâncias de um procedimento podem estar simultaneamente ativas.

Uma característica interessante de Módulo 2 é a possibilidade de construção de programas grandes a partir de módulos definidos e implementados separadamente. Entretanto, em função das modificações incorporadas na linguagem, Módulo 2 tornou-se muito mais adequada para implementação de sistemas do que para a programação de aplicações de tempo-real.

3.4. Ada

A linguagem Ada (BRAUN, 1981) foi projetada pelo Departamento de Defesa dos Estados Unidos com o objetivo de padronizar uma linguagem orientada para programas de sistemas de tempo-real.

Para o desenvolvimento de sistemas, Ada requer um ambiente de programação poderoso, o qual constitui-se de um núcleo e de um conjunto mínimo de ferramentas, incluindo compiladores, editores, gerenciadores, etc. Programas em Ada são estruturados na forma de pacotes (PACKGE) e podem expressar concorrência através de tarefas que interagem entre si usando o mecanismo rendezvous. Um pacote agrupa uma coleção de recursos que podem encapsular tipos de dados, objetos, procedimentos, funções, tarefas ou mesmo outros pacotes. Ele consiste de uma parte de especificação e uma parte de implementação, o que torna possível a abstração das estruturas, e a compilação separada de cada componente.

As tarefas definem procedimentos e entradas. Quando uma tarefa deseja se comunicar com outra, ela deve chamar o procedimento do comando ACCEPT correspondente, o qual deve se encontrar especificado na tarefa destino. Uma comunicação seletiva entre tarefas é possível mediante o uso do comando SELECT.

Ada apresenta algumas facilidades que podem suportar configuração dinâmica, uma vez que permite criação de tarefas em tempo de execução, e ainda mais, possibilita que o nome de uma tarefa seja comunicado para outras.

A alteração ou extensão de um sistema desenvolvido em Ada requer a reprogramação e reconstrução do sistema como um todo. Isto independe do fato da alteração ser provocada pela modificação de algum componente anteriormente existente, ou pela inclusão de um novo elemento.

3.5. PEARL

UNIVERSIDADE FEDERAL DA PARAÍBA
Pró-Reitoria Para Assuntos do Interior
Coordenação Setorial de Pós-Graduação
Rua Aprigio Veloso, 882 - Tel (083) 321-7222-R 355
58.100 - Campina Grande - Paraíba

PEARL ("Process and Experiment Automation Real-time Language") é uma linguagem orientada à programação de aplicações para controle de processos industriais (WERUN & WINDAUER, 1985). Um programa em Pearl é composto de um ou mais módulos, onde cada módulo constitui uma unidade independente de compilação.

Cada módulo inclui em sua especificação uma seção de programação de operações dependentes da instalação, denominada "SYSTEM-division" e uma seção de formulação e codificação da aplicação, envolvendo naturalmente os algoritmos de controle e os aspectos de concorrência relacionados, denominada "PROBLEM-division".

As tarefas constituem em Pearl os elementos concorrentes, as quais podem estar associadas a diferentes níveis de prioridade. A comunicação entre tarefas é feita através de variáveis compartilhadas e a sincronização é obtida através de semáforos e regiões críticas.

PEARL pode ser caracterizada como a primeira linguagem realmente dedicada a aplicações de tempo-real (STEUSLOFF, 1984). Suas construções refletem as características de um ambiente multi-tarefa, onde o programador dispõe de algum controle no escalonamento das tarefas através das primitivas de ativação, suspensão e encerramento de execução. Estas facilidades podem inclusive suportar configuração dinâmica.

3.6. CONIC

CONIC (KRAMER et alii, 1983), (Kramer et alii, 1984) é uma ferramenta dedicada a programação e configuração de sistemas distribuídos, cujo projeto encontra-se em desenvolvimento no Departamento de Computação do Imperial College (Londres).

A característica mais interessante de CONIC é a separação das etapas de programação e configuração de um aplicativo. Para isto CONIC fornece duas linguagens complementares: CONIC/P, usada para programar os módulos que implementam as funções componentes do aplicativo e a linguagem CONIC/C, a qual é usada para descrever o sistema a partir de interconexões de instâncias dos módulos programados. Esta facilidade possibilita a implementação de reconfiguração dinâmica do sistema.

A comunicação e sincronização entre tarefas é feita por troca de mensagens através de portas lógicas de comunicação. A cada porta é associado um tipo e apenas mensagens deste tipo podem ser transmitidas por seu intermédio. O estabelecimento de interconexões entre módulos é realizado na etapa de configuração através de ligações direcionadas de portas de saída a portas de entrada.

3.7. Considerações Finais

Dentre as várias linguagens citadas neste capítulo, vale a pena ressaltar a importância, em particular no contexto deste trabalho, das características das linguagens Ada e CONIC; Ada por ser uma linguagem bastante poderosa e por incorporar novos mecanismos especialmente adequados à programação de tempo-real e CONIC pela flexibilidade que oferece ao permitir que as etapas de programação e configuração de uma aplicação sejam realizadas de forma totalmente independente.

4. AMBIENTE PARA DESENVOLVIMENTO DE APLICAÇÕES DE TEMPO-REAL

Este capítulo apresenta o ambiente implementado no corrente trabalho, o qual procura atender aos requisitos essenciais para programação de software distribuído de controle. O ambiente consiste basicamente de uma metodologia orientada ao projeto de sistemas distribuídos de controle (LOPES & ADAN COELLO, 1986) e de duas linguagens: a Linguagem de Programação de Módulos (LPM) e a Linguagem de Configuração de Módulos (LCM).

A metodologia adotada explora o conceito de módulo, o qual mostra-se especialmente adequado aos sistemas de controle distribuído projetados para ter um longo período de vida útil, durante o qual estão sujeitos a mudanças.

As alterações de um sistema de controle podem ser provocadas pela expansão do processo que está sendo controlado, pela introdução de novas tecnologias ou pela ocorrência de falhas. Nessas situações pode ser extremamente oneroso ou mesmo perigoso parar completamente o sistema, sendo portanto desejável realizar as alterações com o sistema em operação, mesmo que de forma degradada.

Na metodologia incorporada ao ambiente os módulos contém interfaces bem definidas, permitindo portanto, que uma expansão ou reestruturação corresponda simplesmente a inclusão, exclusão ou reconexão de módulos através de mudanças nas ligações entre as suas interfaces.

Nesse sentido o desenvolvimento de cada módulo deve ser independente da estrutura do hardware, e somente na etapa de configuração tal estrutura deve ser considerada. A independência relativa ao hardware deve ir de um extremo, onde todas as funções de controle (módulos) residem numa única estação, até outro, onde cada função reside numa esta-

ção distinta. Esta flexibilidade deve ser obtida sem a necessidade de mudanças nos módulos de software que implementam cada uma das funções de controle, não sendo necessária, inclusive, a recompilação dos módulos. Esta opção oferece ao engenheiro de processo a possibilidade de ir de um extremo a outro, ou seja, de um processamento centralizado a um processamento totalmente distribuído, utilizando basicamente o mesmo software aplicativo.

A metodologia compõe-se basicamente das seguintes etapas:

a) Decomposição funcional do sistema em módulos

A decomposição do sistema em módulos está norteada pelo conceito de "independência de módulos" (PARNAS, 1972). A independência de módulos pode ser qualitativamente avaliada segundo os critérios de coesão e acoplamento. Coesão é a medida da unidade funcional de um módulo, isto é, um módulo altamente coeso deve desempenhar apenas (idealmente) uma função. Acoplamento é a medida da interdependência relativa entre módulos. Módulos com baixo índice de acoplamento trocam informações através de suas interfaces, enquanto módulos altamente acoplados compartilham dados. Módulos com alto grau de coesão requerem pouca interação com outros módulos para desempenhar sua função. Adicionalmente estes módulos podem ser vistos como componentes reusáveis na construção de diferentes sistemas onde a função seja necessária. É desejável portanto decompor o sistema em módulos com alta coesão e baixo acoplamento. Na metodologia adotada os módulos nunca compartilham área de dados, reduzindo portanto o grau de acoplamento das mesmas.

b) Definição das interfaces dos módulos

As interfaces dos módulos são constituídas por portas lógicas de entrada e saída, por onde fluem mensagens de formato bem definido. Uma porta lógica é de domínio exclusivo do módulo que a contém. Este conceito permite o desenvolvimento independente de cada módulo sem levar em conta a estrutura do hardware onde a aplicação será implementada.

c) Interligação dos módulos

A interligação de módulos constitui a etapa de configuração lógica da aplicação. Durante esta etapa as diversas subfunções do sistema (módulos) são compostas de forma a desempenhar a função global. A interligação de módulos é feita conectando-se portas de saída a portas de entrada do mesmo tipo. Portas do mesmo tipo são aquelas por onde fluem mensagens com o mesmo formato.

Os módulos constituem os elementos concorrentes do sistema e a sincronização entre eles é obtida através dos mecanismos de troca de mensagens. A implementação de uma aplicação no ambiente é realizada através do uso das linguagens LPM e LCM, as quais são suportadas em tempo de execução por um núcleo de tempo-real (ADAN COELLO, 1986).

5. ESPECIFICAÇÃO DAS LINGUAGENS DE PROGRAMAÇÃO E CONFIGURAÇÃO DE MÓDULOS

O ambiente para desenvolvimento de aplicações tempo-real apresentado no Capítulo 4 é constituído basicamente das Linguagens de Programação (LPM) e Configuração (LCM) de Módulos. A construção de um programa de aplicação divide-se em duas etapas: a programação em LPM dos módulos que implementam as funções do sistema e a configuração em LCM de aplicação a partir dos módulos disponíveis. Um módulo consiste de um programa sequencial que implementa uma ou mais funções. A comunicação entre módulos é feita através de troca de mensagens mediante uma interface constituída de portas lógicas de entrada e saída. A configuração de uma aplicação corresponde a um programa LCM, onde são especificados os módulos que compõem a aplicação e a interligação de suas portas. Esse modelo adotado, onde a comunicação entre módulos é feita através de suas interfaces, possibilita o desenvolvimento independente de cada módulo sem levar em conta a etapa de configuração.

Este capítulo apresenta uma visão geral das linguagens LPM e LCM, onde são abordados os principais aspectos sintáticos e semânticos. Para a representação da sintaxe das linguagens são utilizados diagramas de sintaxe, onde as figuras circulares envolvem os símbolos terminais, ou seja, palavras reservadas, identificadores, constantes e símbolos especiais. Os símbolos não-terminais, como param formais módulo e corpo módulo, são representados nas figuras retangulares. As semi-retas orientadas indicam a sequência na qual os componentes da linguagem estão dispostos. A descrição das linguagens LPM e LCM apresentadas nessa dissertação refletem o estágio atual de implementação e constituem juntamente com o Núcleo Tempo-Real (ADAN COELLO, 1986) o protótipo inicial

de validação do ambiente proposto no Capítulo 4. A definição BNF completa das linguagens é apresentada no Apêndice A.

5.1. A Linguagem de Programação de Módulos - LPM

A definição da LPM é similar à linguagem CONIC/P (KRAMER et alii, 1984a), ou seja, ela é baseada na linguagem Pascal e oferece ao usuário facilidades para expressar o conceito de módulo, sua interface e as primitivas de troca de mensagens.

Dentre as várias linguagens de alto nível existentes, Pascal foi escolhida para servir de base à definição da linguagem LPM, por ser uma linguagem poderosa, bem definida, bastante difundida, por apresentar uma verificação rígida de tipos, por permitir o projeto de programas bem estruturados e modulares e por algumas de suas versões suportarem a ligação de procedimentos externos escritos em outras linguagens.

Assim, a Linguagem de Programação de Módulos engloba todas as construções de Pascal destinadas a programação sequencial, sendo o conceito de programa em Pascal substituído pelo conceito de módulo na LPM. Para suportar as operações de troca de mensagens incorporou-se na LPM algumas extensões à linguagem Pascal.

As extensões à linguagem Pascal incorporadas possibilitam:

- a declaração de portas de entrada e saída;
- a comunicação entre módulos por intermédio de primitivas de troca de mensagens;
- a mudança dinâmica 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 linguagem LPM, sendo dada ênfase às construções correspondentes às extensões do Pascal. Com o objetivo de contrastar os comandos LPM dos comandos Pascal, a terminologia linguagem LPM ou simplesmente LPM, será usada no decorrer do texto para representar as extensões da linguagem Pascal.

5.1.1. Vocabulário e Elementos Básicos da Linguagem

A representação usada na descrição do vocabulário da linguagem é a BNF ("Bakus Normal Form"), onde apenas são apresentados os símbolos básicos usados na definição das construções LPM.

```
<VOCABULARIO> ::=
```

```
    <VOCABULARIO PASCAL> | <VOCABULARIO LPM>
```

```
<VOCABULARIO LPM> ::=
```

```
    <PALAVRAS RESERVADAS> | <SIMBOLO LPM ESPECIAL>
```

```
<PALAVRAS RESERVADAS> ::=
```

```
    ABORT | BEGIN_MODULE | DEFINE | DELAY |
    ELSE_SELECT | END_LOOP | END_MODULE | END_SELECT |
    END_SEND | ENTRYPORT | EXIT | EXITPORT | FAIL |
    FORWARD | FROM | INTALLOC | LINKED | LOOP |
    MAP_INT | MESSAGE | MODULE | MODULE_ID | OR_SELECT
    | PSELECT | QLEN | QUEUE | REASON | RECEIVE |
    REPLY | RSELECT | SEND | SETPRIORITY | TIME |
    TIMEOUT | USE | WAIT | WAITIO | WHEN
```

```
<SIMBOLO LPM ESPECIAL> ::= "=>"
```

5.1.2. Definição de um Módulo

O módulo é a maior entidade que pode ser construída na linguagem LPM, sendo a sua definição similar à especificação de um tipo, a partir do qual instâncias do módulo podem ser criadas durante a configuração da aplicação.

Os parâmetros formais especificados após o nome do módulo são opcionais e podem ser utilizados na configuração, mais precisamente, na criação das instâncias dos módulos para estabelecer, por exemplo, características inerentes àquela particular configuração. Estes parâmetros podem ser de qualquer um dos tipos básicos do Pascal, ou seja, inteiro, real, lógico ou caracter.

No módulo são declaradas as portas que compõem sua interface, as mensagens usadas em comandos de troca de mensagens e as estruturas de dados, procedimentos e funções usadas dentro dele. As Figuras 5.1, 5.2 e 5.3 apresentam, respectivamente, os diagramas sintáticos correspondentes à estrutura de um módulo, a definição de seus parâmetros formais, e a sequência de construções na especificação de um bloco LPM. Um

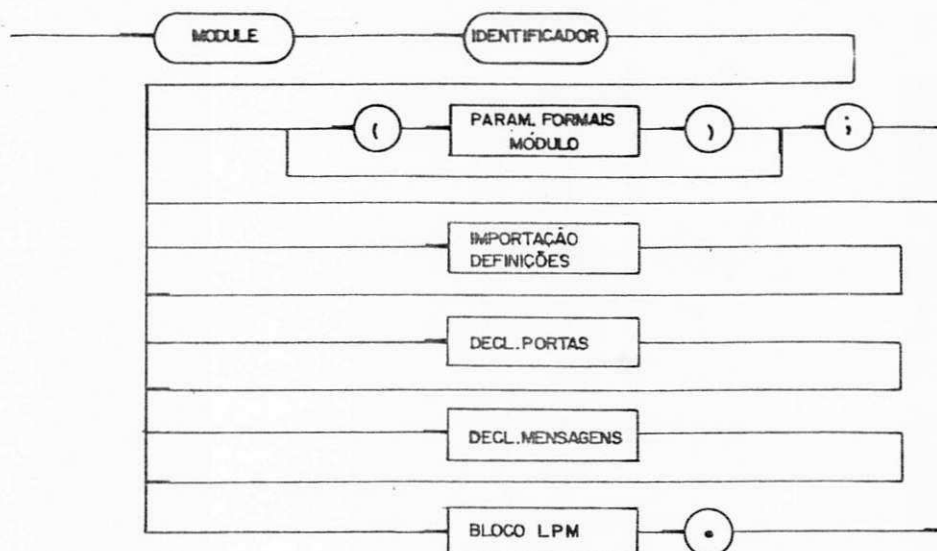


FIGURA 5.1 - Estrutura de um módulo LPM

bloco LPM é análogo a um bloco Pascal, onde o símbolo não-terminal comandos incorpora os comandos Pascal e LPM.

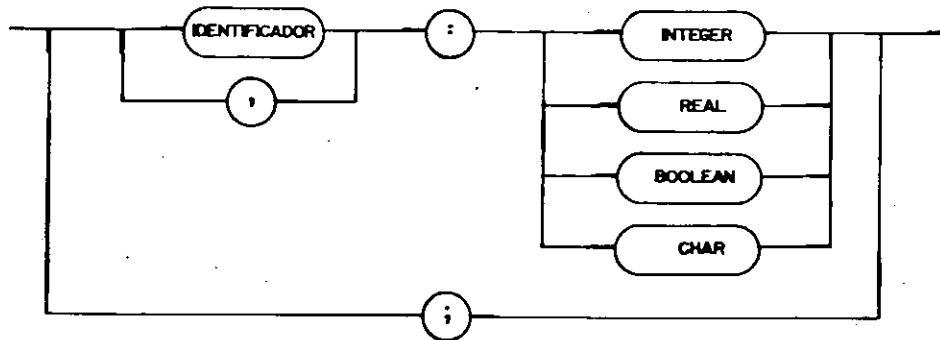


FIGURA 5.2 - Parâmetros formais módulo

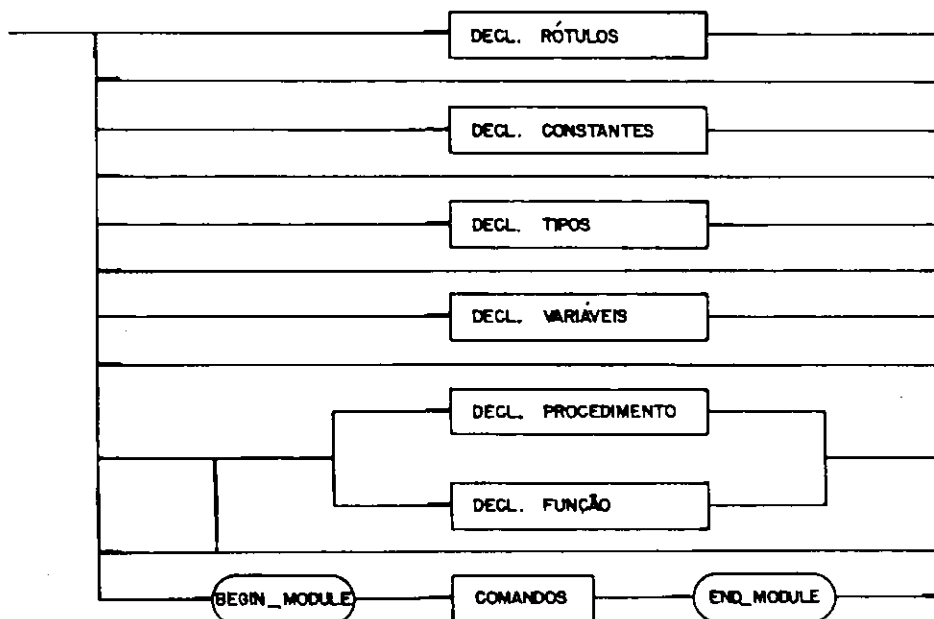


FIGURA 5.3 - Bloco LPM

5.1.3. Importação de Definições

Como uma aplicação normalmente é constituída por mais de um módulo, é necessário que alguns módulos usem as mesmas definições. Visando eliminar a tarefa de ter que declarar os tipos associados à interface em cada módulo e, objetivando a realização de consistências na fase de reconhecimento da linguagem, decidiu-se por utilizar um arquivo de definições (item 5.1.8), aonde o projetista da aplicação deve declarar os tipos associados à interface dos módulos; é permitido também a definição de constantes e tipos comuns aos vários módulos. A sintaxe do comando LPM para importação de objetos comuns é apresentada na Figura 5.4, onde o símbolo terminal nome arquivo é o nome do arquivo a ser incluído.



FIGURA 5.4 - Importação de definições

5.1.4. Declaração de Portas

Na linguagem de Programação de Módulos as portas estabelecem a interface para a comunicação entre módulos. A cada porta é associado um tipo. Essa associação de tipos possibilita a realização de consistências nos comandos de troca de mensagens e na conexão de portas na fase de configuração dos módulos.

Uma comunicação entre dois módulos é caracterizada por uma ligação entre portas do mesmo tipo; nos instantes em que um determinado módulo necessita enviar ou receber mensagens, ele deve fazer referência a

uma porta de saída ou de entrada, respectivamente, ou seja, o uso das portas de um determinado módulo é de domínio local ao módulo. Esse modelo de comunicação em que um módulo apenas se comunica com o exterior através de sua interface assegura um alto nível de independência da configuração em relação a programação, permitindo desta forma uma independência total na programação de cada módulo.

A declaração de uma porta (Figura 5.5) define os tipos de operações na qual ela será empregada. As operações primitivas em portas são: o envio de uma mensagem através de uma porta de saída e a recepção de uma mensagem a partir de uma porta de entrada. O tipo de comunicação envolvida, síncrona ou assíncrona, também é definida na declaração da porta.

Uma família de portas pode ser declarada através da especificação do intervalo de variação do índice após o identificador da porta. Esta construção é análoga à declaração de arrays em Pascal.

O tipo associado a uma porta ou família de portas pode ser elementar ou estruturado. Para que um tipo estruturado possa ser referenciado em uma declaração de portas é necessário que ele seja definido no arquivo de definições.

Uma porta síncrona é definida através da especificação da palavra reservada `REPLY` em sua declaração; o identificador seguinte ao `REPLY` indica o tipo de mensagem de resposta. Várias portas de saída síncronas podem ser conectadas a uma mesma porta de entrada síncrona. As mensagens 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, portanto a chegada de uma nova mensagem a uma porta de entrada procedente da mesma porta de saída, faz com que a mensagem antiga seja descartada, caso ela esteja ainda enfileirada na porta de entrada.

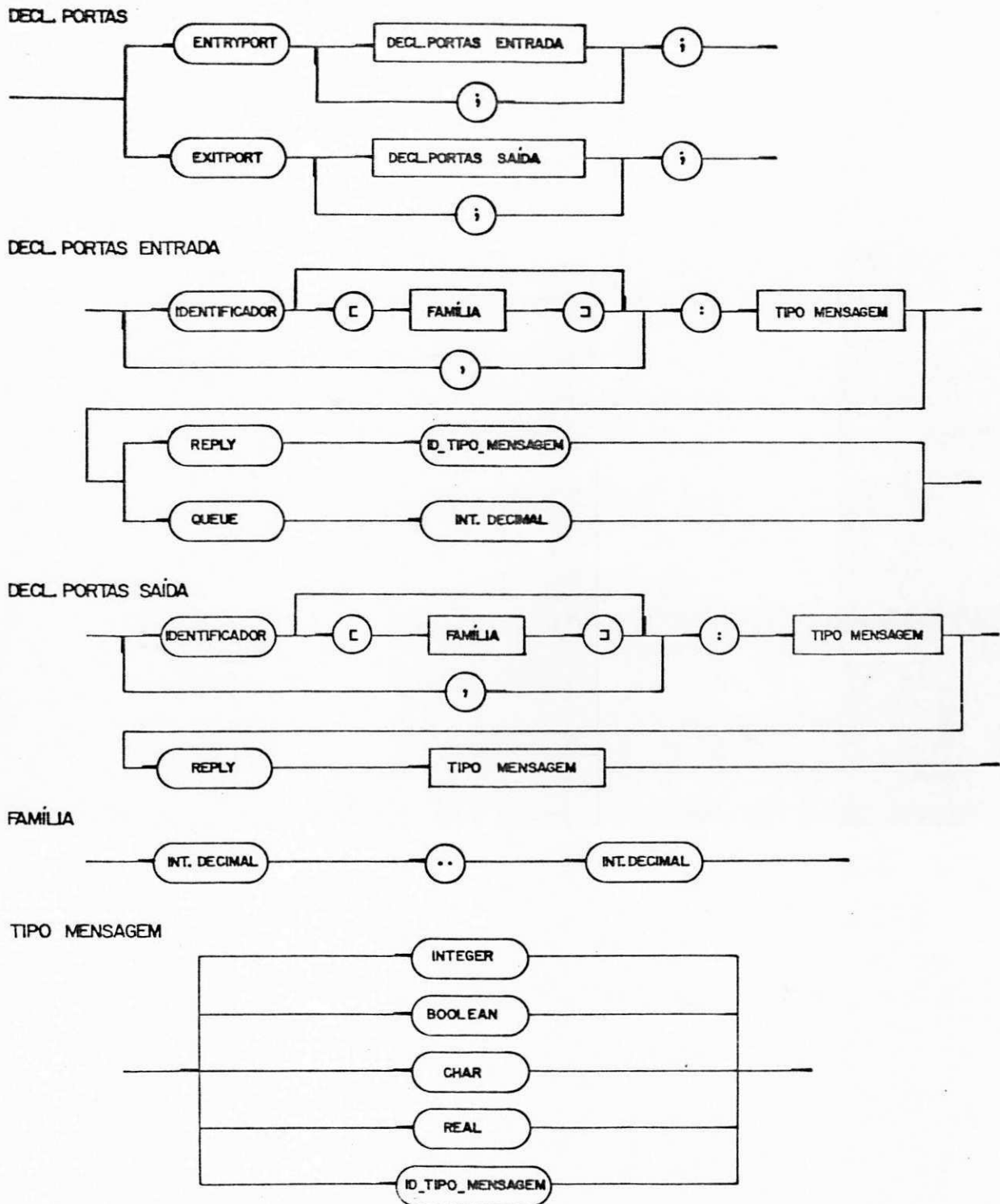


FIGURA 5.5 - Declaração de portas

Uma porta assíncrona não contém em sua definição a palavra reservada `REPLY`; entretanto se a porta declarada for uma porta de entrada, o programador pode especificar através da palavra reservada `QUEUE` a criação de uma fila onde serão armazenadas as mensagens que chegarem à porta de entrada. Na declaração deste tipo de porta, se o programador não estabelecer o número máximo de mensagens através da especificação de uma cláusula `QUEUE`, assumir-se-á que esta será de uma mensagem. Caso a fila de entrada de uma porta esteja cheia num dado instante em que ocorre a chegada de uma nova mensagem, a mensagem mais antiga da fila é descartada para que a mensagem recém-chegada possa ser armazenada. As mensagens que chegam a uma porta de entrada assíncrona são enfileiradas em ordem de chegada, independente da porta de saída de onde procede. Uma porta de saída assíncrona pode estar 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.

5.1.5. Declaração de Mensagens

UNIVERSIDADE FEDERAL DA PARAÍBA
Pró-Reitoria Para Assuntos do Interior
Coordenação Setorial de Pós-Graduação
Rua Aprígio Veloso, 882 - Tel (083) 321-7222-R 355
58.100 - Campina Grande - Paraíba

Para que sejam feitas verificações de compatibilidade de tipos entre as portas e as mensagens que por elas fluem, deve-se definir as estruturas das mensagens. A Figura 5.6 apresenta o diagrama de sintaxe para declarações de mensagens, onde o tipo a ser especificado na declaração deve ser um tipo elementar ou um identificador associado à declaração de um tipo estruturado no arquivo de definição.

DECL MENSAGENS

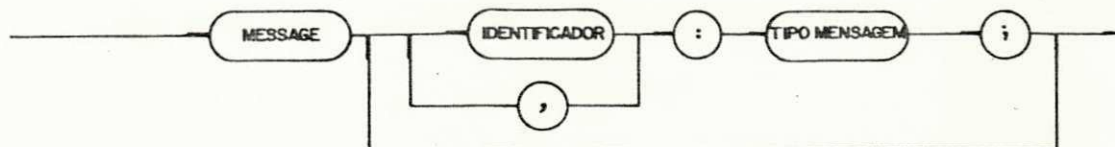


FIGURA 5.6 - Declaração de mensagens

5.1.6. Comando para Especificação de um Ciclo Infinito

A Figura 5.7 apresenta a sintaxe do comando LOOP, onde a sequência de comandos entre os símbolos terminais LOOP e END_LOOP será executada indefinidamente. A saída do ciclo pode ser obtida através de especificação em seu interior do comando EXIT, causando dessa forma o desvio de execução para o comando seguinte ao comando LOOP.

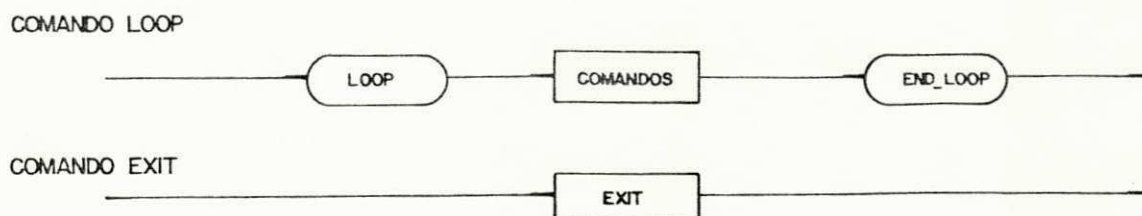


FIGURA 5.7 - Comandos LOOP e EXIT

Muito embora esta construção possa ser obtida na linguagem Pascal através da construção "REPEAT...UNTIL false" ou "WHILE true DO...", decidiu-se por incorporá-la diretamente na LPM uma vez que a ocorrência de ciclos infinitos em programação tempo-real é bastante frequente, tornando assim a sua especificação bem mais simples e legível.

5.1.7. Comandos para Troca de Mensagens

Os comandos LPM para troca de mensagens possibilitam a efetivação de comunicações assíncronas e síncronas. Uma comunicação assíncrona corresponde à criação de um canal lógico unidirecional, onde as mensagens fluem do módulo fonte para o módulo destino. Neste tipo de comunicação, o módulo fonte não fica suspenso após o envio da mensagem, a me-

nos que a mensagem despachada esteja sendo esperada por um módulo de maior prioridade.

Uma comunicação síncrona estabelece uma troca de mensagens bidirecional. Neste tipo de comunicação o módulo fonte envia uma mensagem para o módulo destino e fica bloqueado até que uma mensagem de resposta seja retornada. Em vez de retornar uma mensagem de resposta normal, o módulo receptor de uma mensagem síncrona pode cancelar a comunicação, enviando uma resposta indicativa de ocorrência de falha, através do procedimento ABORT (item 5.1.9.1) ou redirecionar a mensagem recebida para outra porta via comando FORWARD (item 5.1.7.3).

Os itens a seguir descrevem a sintaxe dos comandos LPM de troca de mensagens, a saber, comando para envio de mensagens, comando de recepção de mensagens, comando de resposta e comando de recepção seletiva.

5.1.7.1. Comando de Envio de Mensagens

Para o envio de uma mensagem através de uma porta de saída a linguagem LPM dispõe do comando SEND (Figura 5.8).

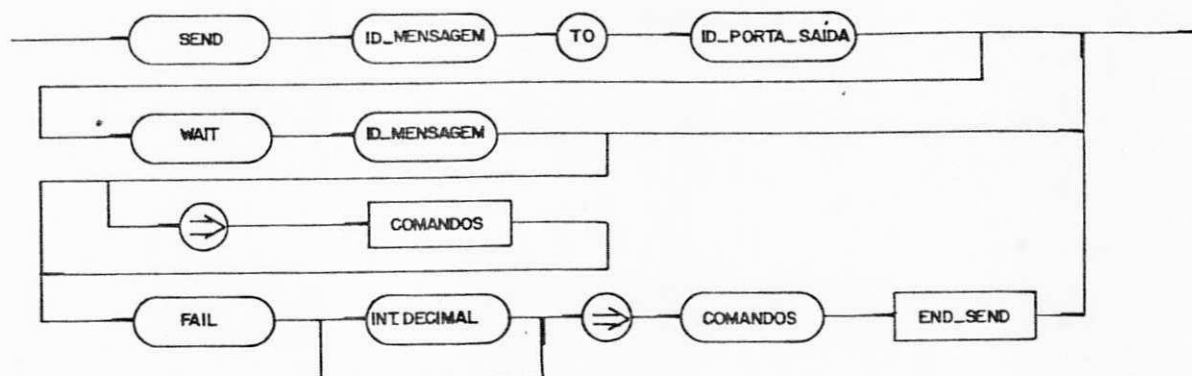


FIGURA 5.8 - Comando SEND

A porta de saída referenciada no comando após a palavra reservada TO define o tipo de comunicação; caso seja uma porta assíncrona, a sintaxe do comando SEND resume-se na seguinte estrutura:

```
SEND ident.mensagem TO porta saída.
```

A mensagem envolvida na comunicação deve ter o mesmo formato (tipo) da porta através da qual será transmitida. O comando não produz nenhum efeito se a porta de saída não estiver conectada.

Com relação ao envio síncrono de uma mensagem, a LPM fornece duas possibilidades:

a) Envio totalmente síncrono:

```
SEND ident. mensagem TO porta saída  
WAIT mensagem resposta
```

Este comando envia uma mensagem à porta de saída referenciada e bloqueia a execução do módulo correspondente até que a mensagem de resposta seja recebida. Os tipos referentes a mensagem a ser enviada e a resposta a ser recebida devem ser os mesmos usados na declaração da porta de saída. Se a porta estiver desconectada no instante da execução do comando, um erro será gerado em tempo de execução. Caso a transação seja cancelada antes da recepção da mensagem de resposta, uma mensagem de erro também será gerada em tempo de execução.

b) Envio parcialmente síncrono, cujo formato é o seguinte:

```
SEND ident. mensagem TO porta saída
      WAIT mensagem resposta =) comandos 1
      FAIL tempo =) comandos 2
END_SEND
```

A semântica deste comando é similar à do comando de envio totalmente síncrono, diferenciando apenas na adição de uma cláusula para tratamento de falhas. A sequência de comandos 1 será executada se a mensagem de resposta for recebida com sucesso; a sequência de comandos 2 será executada se ocorrer um dos seguintes eventos:

- o tempo de espera especificado na cláusula `FAIL` expira sem que a mensagem de resposta seja recebida;
- a porta de saída não está conectada no instante da execução do comando;
- a transação é cancelada pelo suporte da linguagem ou pelo módulo receptor.

Uma mensagem de resposta é descartada caso chegue após o período de tempo especificado. Na sequência de comandos 2 pode-se determinar o motivo que ocasionou o cancelamento da comunicação através da função `REASON` (item 5.1.9.1). A ausência da especificação do tempo de espera numa cláusula de falha, bem como a especificação de tempo de espera negativo causa a espera pela mensagem por um período de tempo indeterminado, a menos que ocorra um erro do tipo porta desconectada.

5.1.7.2. Comando de Recepção de uma Mensagem

O comando RECEIVE (Figura 5.9) permite a um módulo receber uma mensagem através de uma porta de entrada. Se no instante da execução do comando não houver nenhuma mensagem disponível na porta de entrada, o módulo fica bloqueado até a chegada de alguma mensagem, independentemente da comunicação ser síncrona ou assíncrona. A mensagem recebida é copiada na estrutura de dados correspondente à mensagem e caso o comando de recepção especifique uma cláusula REPLY, a mensagem de resposta é imediatamente enviada através da porta de entrada correspondente.

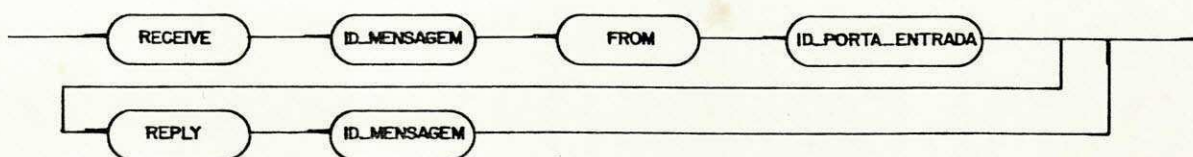


FIGURA 5.9 - Comando RECEIVE

Considerando que muitas vezes é necessário que algum processamento seja efetuado antes da resposta ser retornada, a LPM possibilita que a mensagem de resposta seja enviada através de um comando desvinculado da recepção (Figura 5.10).

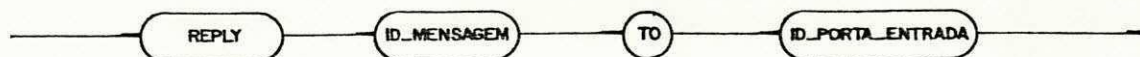


FIGURA 5.10 - Comando REPLY

A referência a uma porta de entrada num comando REPLY sem que haja ocorrido uma recepção de mensagem não causa efeito algum.

5.1.7.3. Comando de Desvio de uma Comunicação Síncrona

O comando **FORWARD** cuja sintaxe é apresentada na Figura 5.11 pode ser usado no lugar do comando de resposta de uma comunicação síncrona.



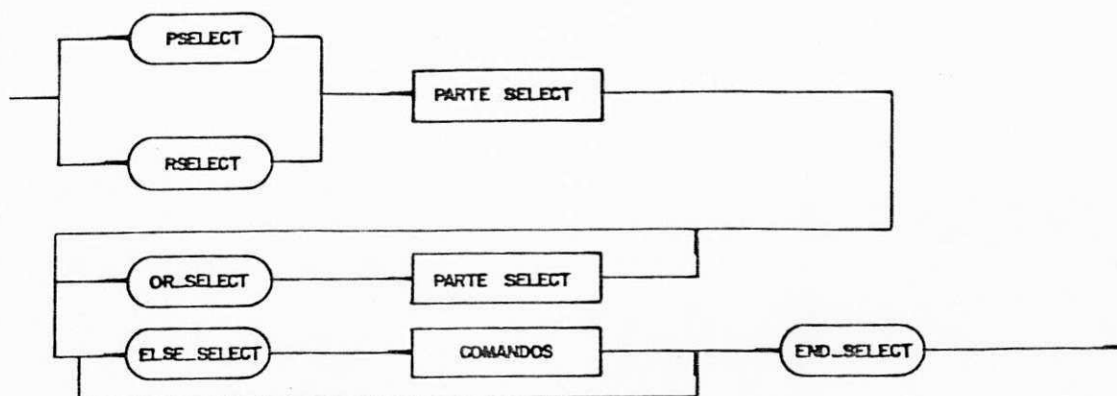
FIGURA 5.11 - Comando FORWARD

A mensagem recebida através da porta de entrada é imediatamente enviada para a porta de saída especificada sem qualquer processamento. Ambas as portas devem ser do mesmo tipo. A resposta eventualmente especificada pelo receptor final da comunicação será enviada diretamente ao emissor inicial da mensagem.

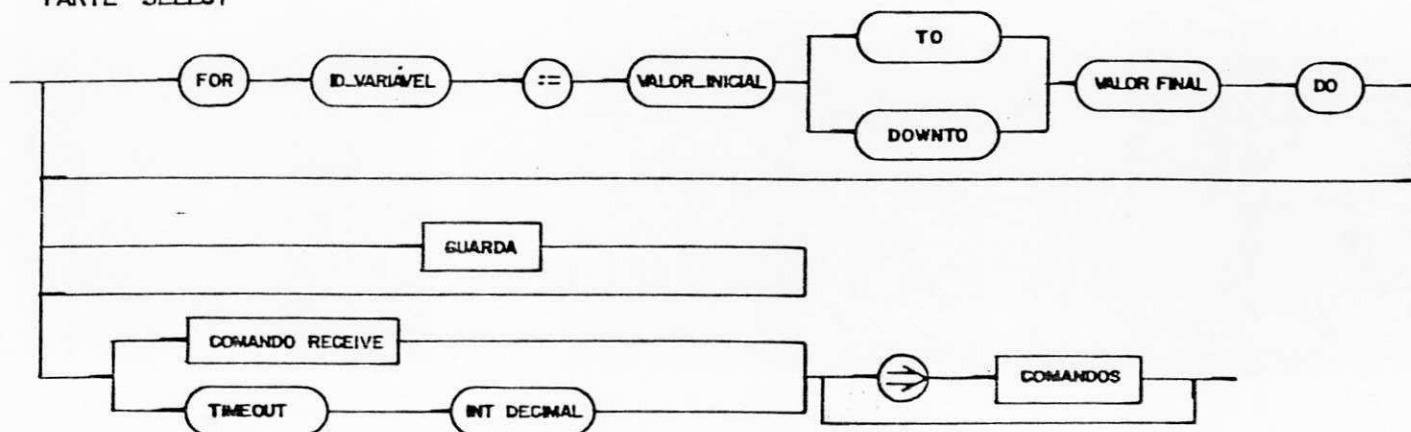
5.1.7.4. Comando de Recepção Seletiva

O comando **SELECT** (Figura 5.12) permite a um módulo esperar simultaneamente a recepção de uma mensagem a partir de um conjunto específico de portas. Neste tipo de recepção o programador pode especificar uma ou mais cláusulas de temporização (timeout), onde o tempo de espera por uma mensagem fica limitado a um período determinado. É permitido também o uso do comando de repetição **FOR** na especificação de uma cláusula de recepção ou de temporização, possibilitando-se desta forma que várias cláusulas sejam expressas numa única parte do comando **SELECT**.

COMANDO SELECT



PARTE SELECT



GUARDA

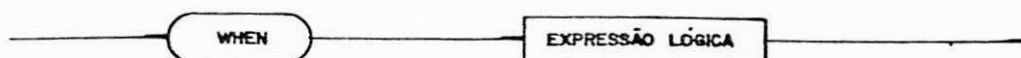


FIGURA 5.12 - Comando SELECT

A execução de um comando SELECT envolve os seguintes passos:

a) Determinação das cláusulas de seleção elegíveis à recepção
Todas as cláusulas de seleção não precedidas por guardas, ou associadas a guardas cuja avaliação resulta no valor TRUE, são consideradas elegíveis. O uso do comando de recepção equivale a repetir a avaliação da guarda para cada valor da variável de controle do FOR, desde o valor inicial até o valor final.

b) Seleção de uma cláusula de recepção

Se houver disponibilidade de mensagens em uma ou mais portas de entrada referenciadas em cláusulas elegíveis, uma dessas cláusulas deve ser escolhida. A escolha é feita de acordo com o tipo de seleção especificado. Caso a seleção seja do tipo priorizado, ou seja, caso o comando de seleção seja um PSELECT, a cláusula selecionada é a de maior prioridade. A prioridade num comando PSELECT decresce a partir da primeira cláusula; assim, a primeira cláusula é a prioritária, depois virá a segunda e assim por diante. Caso a seleção seja do tipo aleatório, ou seja, caso o comando de seleção seja um RSELECT, a cláusula é selecionada de forma aleatória.

Enquanto não houver disponibilidade de mensagens em pelo menos uma das portas elegíveis, o módulo receptor permanece bloqueado. Após a efetivação da recepção, a sequência de comandos associada à cláusula selecionada é executada. Quando a cláusula é precedida pelo comando de repetição FOR, a primeira ação a ser executada após a realização da recepção é restaurar a variável de controle do FOR com o valor correspondente ao instante em que o comando de recepção foi selecionado. Isto se faz necessário uma vez

que o programador pode usar esta variável no comando de resposta ou mesmo em algum controle interno.

c) Seleção de uma cláusula de tempo

O tempo durante o qual um módulo espera a chegada de uma mensagem pode ser limitado por uma ou mais cláusulas de temporização. Uma cláusula de temporização elegível é selecionada se após decorrido o período especificado em unidades de tempo, o módulo não tiver recebido nenhuma mensagem. Nesse instante a sequência de comandos associada à cláusula de temporização é executada. Se várias cláusulas de tempo forem elegíveis, a selecionada é aquela que especifica a menor espera. O tratamento dado na ocorrência do comando de repetição FOR é o mesmo dado na seleção de uma recepção.

d) Seleção da cláusula ELSE

Uma cláusula ELSE é equivalente a uma cláusula de temporização com tempo de espera zero, ou seja, caso não haja mensagens disponíveis nas portas de entrada elegíveis, a sequência de comandos associada à cláusula ELSE é executada. Assim a especificação da parte ELSE em um comando SELECT corresponde à implementação de uma recepção do tipo não-bloqueante.

5.1.8. Unidades de Definição

Considerando que a comunicação entre módulos através de troca de mensagens requer o uso de uma mesma definição em várias declarações de portas e mensagens, decidiu-se por utilizar um arquivo de definições,

denominado unidade de definição, onde os tipos referenciados em declarações de portas e mensagens devem ser declarados, exceto os tipos elementares do Pascal. Esses tipos são importados no módulo através do comando USE. A Figura 5.13 apresenta a estrutura de uma unidade de definição.

Na atual implementação uma unidade de definição deve ser previamente compilada, quando então é gerado o arquivo a ser referenciado pelo comando USE (Capítulo 6).

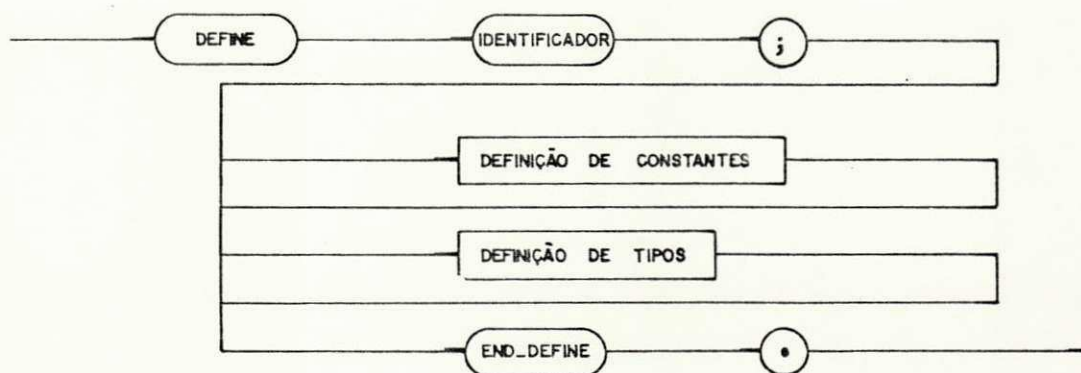


FIGURA 5.13 - Estrutura de uma unidade de definição

5.1.9. Procedimentos e Funções Pré-Definidos

A LPM fornece um conjunto de procedimentos e funções pré-declarados cuja descrição é apresentada nos itens seguintes.

5.1.9.1. Operações Relacionadas à Troca de Mensagens

Algumas operações relacionadas à troca de mensagens foram adicionadas à LPM na forma de procedimentos e funções, cujas descrições são apresentadas a seguir:

- PROCEDURE ABORT (porta de entrada, motivo falha)

O procedimento ABORT pode ser usado no lugar de um comando REPLY para cancelar uma comunicação síncrona; a porta de entrada referenciada no procedimento deve ser indicada através da especificação de um identificador da porta de entrada síncrona. O parâmetro motivo falha indica o motivo do cancelamento e corresponde a um valor inteiro definido pelo usuário. Este valor deve ser maior que a constante pré-declarada `erro_max`. Um cancelamento provoca a execução da cláusula FAIL do correspondente comando SEND.

- FUNCTION REASON

A função REASON pode ser usada na cláusula FAIL de um comando de envio síncrono para determinar a razão do cancelamento da comunicação. O valor retornado pela função REASON é um número inteiro correspondente às constantes pré-declaradas `etimeout` e `elink`, ou ao motivo falha especificado pelo comando ABORT do módulo destinatário. O código `etimeout` indica que o tempo de espera por uma resposta expirou e o código `elink` indica que a porta de saída especificada no comando não está conectada.

- FUNCTION LINKED (porta de saída)

A função LINKED pode ser usada para verificar se a porta de saída cujo nome é especificado na chamada de função está

conectada. O valor retornado é TRUE se a porta está conectada e FALSE em caso contrário.

- FUNCTION QLEN (porta de entrada)

A função QLEN determina o número de mensagens disponíveis em uma porta de entrada.

5.1.9.2. Mudança de Prioridade de um Módulo

A mudança da prioridade de um módulo em tempo de execução pode ser obtida através do seguinte procedimento:

PROCEDURE SETPRIORITY (nível de prioridade)

O nível de prioridade deve ser um valor do tipo pré-declarado PRIORITY, onde PRIORITY = (SYSTEMPR, HIGHPR, NORMALPR, LOWPR, LOWESTPR); caso a execução deste procedimento implique numa diminuição da prioridade do módulo, ele poderá ser suspenso para permitir que outro módulo mais prioritário seja executado.

5.1.9.3. Operações Relacionadas com o Tempo

As operações relacionadas com o tempo correspondem ao atraso da execução de um módulo e a leitura do relógio do sistema. Os procedimentos que implementam essas funções são apresentados a seguir:

- PROCEDURE DELAY (período de tempo)

O procedimento DELAY atrasa a execução do módulo por um período de tempo especificado em unidades de tempo (valor inteiro longo).

- FUNCTION TIME

A função TIME retorna o valor, em unidades de tempo (inteiro longo), assinalado pelo relógio do sistema.

5.1.9.4. Tratamento de Interrupções

Considerando que a maior parte dos eventos ocorridos em um sistema de controle tempo-real é de natureza assíncrona, e que muitos desses eventos necessitam de um tratamento prioritário, incorporou-se na LPM alguns mecanismos para tratamento de interrupções em alto nível. Os tratadores de interrupções são implementados como se eles fossem módulos de aplicação e os mecanismos oferecidos para a manipulação de interrupções constituem-se na função INTALLOC e no procedimento WAITIO, cuja descrição é apresentada a seguir:

- FUNCTION INTALLOC (vetor físico)

A função INTALLOC mapeia um elemento do vetor físico de interrupções conhecido pelo usuário, em um elemento do vetor lógico conhecido pelo núcleo. O vetor físico é uma informação dependente do hardware onde a LPM é implementada e identifica o endereço absoluto de memória do início da sequência de instruções que o processador executará em atendimento a uma dada interrupção. A função retorna um valor inteiro correspondente a um elemento do vetor lógico de in-

terrupções, que deverá ser usado pelo programador quando desejar aguardar a ocorrência da interrupção associada.

- PROCEDURE WAITIO (vetor lógico)

O procedimento WAITIO indica a espera pela ocorrência de uma interrupção lógica. O vetor lógico é um número inteiro que identifica a interrupção lógica aguardada. Se no instante da chamada do procedimento já tiver ocorrido a interrupção, o módulo correspondente não será suspenso.

5.1.9.5. Identificação de um Módulo em Tempo de Execução

A função inteira `MODULE_ID` permite que em tempo de execução o módulo correspondente obtenha o número usado pelo tradutor LCM para identificá-lo.

5.2. A Linguagem de Configuração de Módulos - LCM

A Linguagem de Configuração de Módulos é a ferramenta oferecida ao usuário para configurar sua aplicação. Na atual versão de implementação a LCM suporta apenas configuração estática em ambiente centralizado, correspondendo à especificação de uma configuração a um programa LCM que identifica os tipos dos módulos que compõem a aplicação, declara as instâncias desses módulos que serão criadas e descreve as conexões entre as várias instâncias.

5.2.1. Vocabulário e Elementos Básicos da Linguagem

A seguir apresenta-se o conjunto de elementos básicos que compõem a linguagem. Tal como na LPM, a representação usada na descrição do vocabulário é a BNF.

```
<PALAVRAS RESERVADAS> ::= CONFIGURATION | CREATE |
                           END_CONFIGURATION | INSTANCE
                           | LINK | TO
```

```
<SÍMBOLOS ESPECIAIS> ::= "[" | "]" | "(" | ")" | "." |
                           "," | ";" | ":"
```

A representação de um comentário em LCM é equivalente à representação usada no Pascal, ou seja, quaisquer sequências de caracteres delimitados por "{" e "}" ou "(*" e "*)".

Um identificador LCM é qualquer sequência de letras, dígitos ou "_" que comece por uma letra e que não seja uma palavra reservada.

5.2.2. Estrutura de um Programa de Configuração

A Figura 5.14 apresenta a estrutura de um programa de configuração. Uma especificação LCM consiste então de uma descrição onde as instâncias componentes da configuração são declaradas, criadas e interconectadas.

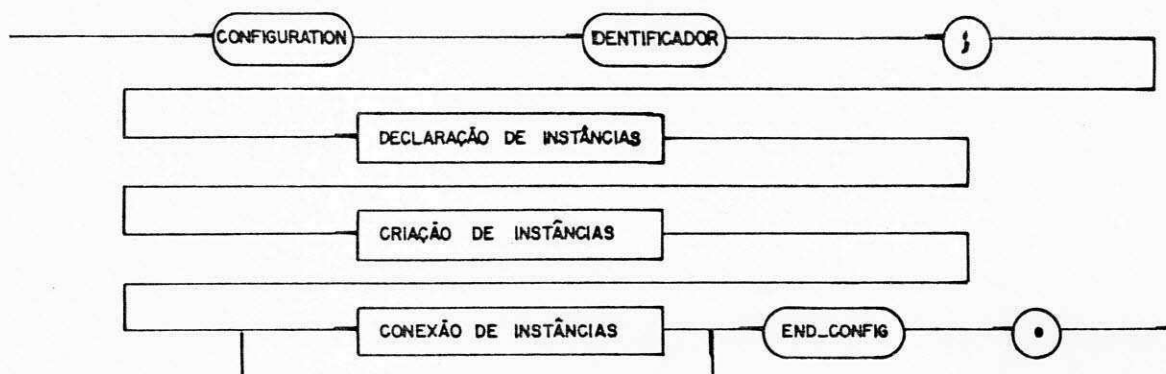


FIGURA 5.14 - Estrutura de um programa LCM

5.2.3. Declaração de Instâncias

Nesta declaração são associados nomes às várias instâncias dos módulos da aplicação (Figura 5.15). O nome associado a cada instância deve ser único e corresponde ao identificador que sucede a palavra reservada `INSTANCE`. Quanto ao identificador que indica o tipo do módulo, ele deve corresponder ao nome de um módulo da linguagem LPM previamente compilado.

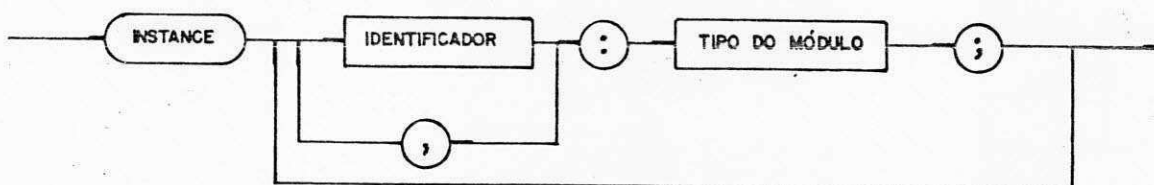
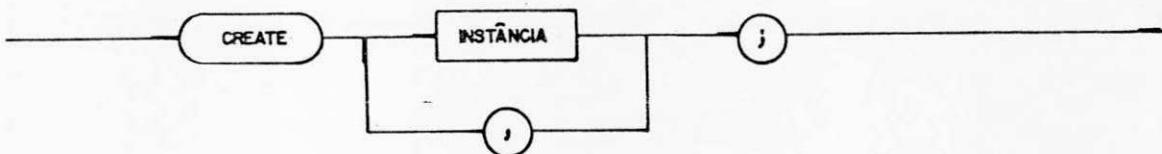


FIGURA 5.15 - Declaração de instâncias

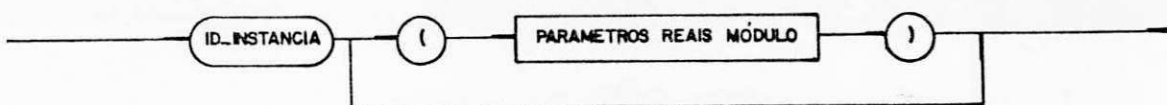
5.2.4. Criação de Instâncias

A construção LCM apresentada na Figura 5.16 é usada para criar as instâncias dos módulos que compõem a aplicação. Esta operação é análoga à declaração VAR do Pascal, onde o programador cria uma instância de um tipo abstrato de dados. Na LCM os módulos são vistos como tipos a partir dos quais várias instâncias podem ser criadas. Caso a especificação LPM de um módulo possua parâmetros formais, os parâmetros reais correspondentes devem então ser especificados na criação das instâncias. Os parâmetros podem ser de qualquer um dos tipos básicos do Pascal e devem aparecer na mesma ordem em que foram declarados no módulo LPM.

CRIAÇÃO DE INSTÂNCIAS



INSTÂNCIA



PARAMETROS REAIS MÓDULO

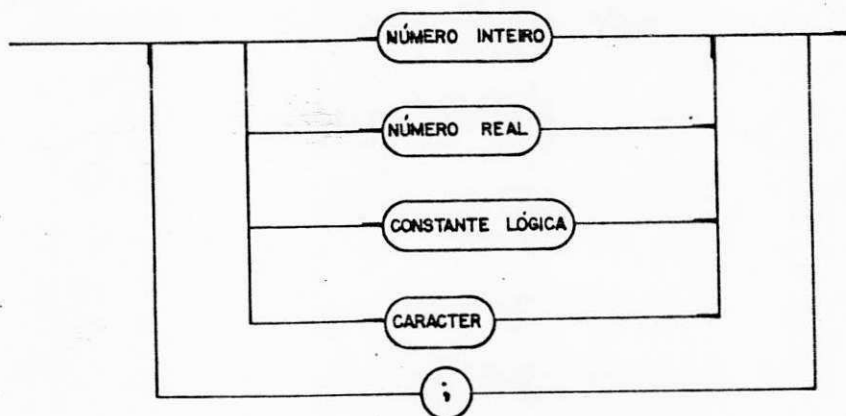


FIGURA 5.16 - Criação de instâncias

Na criação de instâncias também é permitido ao usuário a especificação de algumas diretivas de configuração para definição de prioridade inicial da instância e a área de memória reservada para pilha e heap. Caso estas diretivas não sejam especificadas, os valores padrão característicos da versão de implementação serão assumidos. Muito embora as diretivas não estejam definidas no diagrama sintático da Figura 5.16, elas podem ser vistas no exemplo do Apêndice C, onde /P é usada para definir uma prioridade inicial de um módulo, /S para definir a área reservada para pilha e /H para definir a área reservada para heap.

5.2.5. Conexão de Instâncias

Uma aplicação é geralmente constituída por vários módulos, sendo a comunicação entre eles feita através de troca de mensagens. Para que um módulo possa se comunicar com outro, é necessário que exista uma conexão lógica entre eles. O diagrama sintático do comando LCM que faz conexões lógicas entre portas de módulos distintos é apresentado na Figura 5.17.

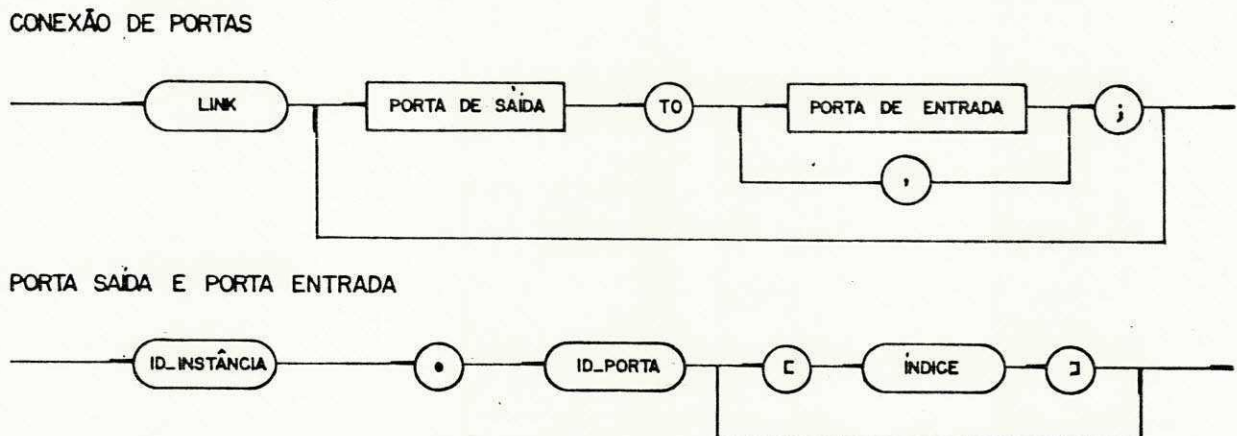


FIGURA 5.17 - Conexão de instâncias

Uma conexão é especificada através de uma ligação direcionada de uma porta de saída para uma porta de entrada. Os seguintes tipos de conexão são permitidos:

- uma porta de saída a uma porta de entrada: permitido para portas assíncronas e síncronas;
- uma porta de saída a várias portas de entrada: permitido somente para portas assíncronas;
- várias portas de saída a uma porta de entrada: permitido para portas assíncronas e síncronas.

A identificação de uma porta deve ser precedida pelo nome da instância à qual ela pertence. Isto é necessário pois as portas das instâncias de um mesmo tipo de módulo tem o mesmo nome. Somente são permitidas conexões que envolvem portas do mesmo tipo.

6. CONSIDERAÇÕES SOBRE A IMPLEMENTAÇÃO

Este capítulo descreve a estrutura e organização da implementação das linguagens LPM e LCM, cujas especificações foram apresentadas no Capítulo 5 e discute alguns aspectos relacionados com o Suporte de Tempo-Real. Uma visão global do esquema adotado é fornecida, ocorrendo apenas o aprofundamento do nível de detalhamento nas questões relativas às principais decisões de projeto.

As linguagens LPM e LCM foram implementadas num computador PCS CADMUS-9200 com Sistema Operacional MUNIX e atualmente estão disponíveis para executar em ambiente compatível com IBM-PC.

As decisões tomadas na fase de desenvolvimento foram orientadas pelos seguintes princípios:

- o código gerado deve executar corretamente em total acordo com a semântica definida para a linguagem;
- as linguagens devem apresentar uma interface bastante "amigável", sinalizando corretamente os erros, localizando-os precisamente com mensagens claras e objetivas;
- os tradutores devem ser facilmente alterados de forma a atender modificações futuras nas definições das linguagens, motivadas por exemplo, pela necessidade de incorporação de novos conceitos, ou mesmo pela própria evolução natural do projeto;
- o código gerado deve ser modular e transportável;

- a implementação deve ser simplificada sempre que possível através do uso de ferramentas.

Considerando que o Sistema Operacional MUNIX dispõe das ferramentas de geração de programas YACC e LEX, decidiu-se por utilizá-las na implementação das linguagens LPM e LCM, uma vez que elas atendem os requisitos explicitados anteriormente e simplificam a tarefa árdua de programação e codificação.

6.1. Suporte de Tempo Real

As Linguagens de Programação e Configuração de Módulos são suportadas em tempo de execução por um Núcleo de Tempo-Real (ADAN COELLO, 1986). O núcleo oferece os serviços necessários a:

- configuração física dos módulos componentes da aplicação;
- troca de mensagens entre módulos através das portas que fornecem a sua interface;
- controle de execução;
- tratamento lógico de interrupções.

O suporte à configuração física da aplicação consiste nos serviços de carga e interconexão de módulos.

O suporte à troca de mensagens entre módulos traduz-se nas primitivas de envio de uma mensagem a uma porta e recepção de uma mensagem de uma porta.

O núcleo utiliza um esquema de escalonamento preemptivo de módulos, ou seja, o módulo de maior prioridade num dado instante que possui todos os recursos para executar, deverá estar no estado de execução.

O controle dinâmico de execução é suportado por serviços que

permitem mudança de prioridade e suspensão temporária de módulos.

O suporte à manipulação e tratamento de interrupções consiste em serviços que possibilitam um mapeamento do vetor físico de interrupções, conhecido pelo usuário, em um elemento do vetor lógico de interrupções, conhecido pelo núcleo.

O núcleo oferece ainda um conjunto de serviços necessários às aplicações de tempo-real, tais como: leitura do relógio de tempo real, cancelamento de comunicação síncrona, etc.

6.1.1. Interface dos Serviços Oferecidos pelo Núcleo de Tempo-Real

Esta seção apresenta a interface dos serviços do núcleo. Convém lembrar que essa interface não é diretamente acessível ao usuário, uma vez que todos os serviços do núcleo estão disponíveis através das construções de alto nível da LPM.

A descrição é apresentada na forma de cabeçalhos de procedimentos Pascal, onde cada elemento que compõe a interface é caracterizado funcionalmente através da escolha de identificadores, cujos nomes são auto-sugestivos. Maiores detalhes podem ser vistos no Apêndice A, onde uma abordagem mais detalhada sobre a interface é apresentada.

6.1.1.1. Serviços de Troca de Mensagens

Esta classe de serviços oferece as funções necessárias à sincronização e comunicação entre módulos. Os serviços de troca de mensagens oferecidos pelo núcleo são:

- Envio assíncrono de uma mensagem:

```
PROCEDURE env_a
    (porta_saída      : INTEGER;
     tamanho_mensagem : INTEGER;
     endereço_mensagem : ADSMEN);
```

- Envio síncrono de uma mensagem:

```
PROCEDURE env_s
    (porta_saída      : INTEGER;
     tamanho_mensagem : INTEGER;
     endereço_mensagem : ADSMEN;
     endereço_resposta : ADSMEN);
```

- Envio síncrono e temporizado de uma mensagem:

```
FUNCTION env_s_temp
    (tempo          : INTEGER4;
     porta_saída    : INTEGER;
     tamanho_mensagem : INTEGER;
     endereço_mensagem : ADSMEN;
     endereço_resposta : ADSMEN): BOOLEAN;
```

- Inicialização de uma cláusula de recepção seletiva de uma mensagem:

```
PROCEDURE irri_rec_sel
    (índice_for      : INTEGER;
     cláusula         : INTEGER;
     porta_entrada   : INTEGER;
     endereço_mensagem : ADSMEN);
```

- Inicialização de uma cláusula de temporização num comando de recepção seletiva:

```

PROCEDURE ini_temp_sel
    (índice_for      : INTEGER;
     cláusula        : INTEGER;
     tempo           : INTEGER4);

```

- Seleção de uma cláusula de recepção seletiva de uma mensagem:

```

FUNCTION sel
    (tipo_do_select  : CHAR;
     cláusula_else   : INTEGER;
     VARS índice_for : INTEGER;
     VARS porta_entrada: INTEGER): INTEGER;

```

- Recepção bloqueante de uma mensagem:

```

PROCEDURE rec
    (endereço_mensagem : ADSMEN;
     porta_entrada     : INTEGER);

```

- Envio de uma mensagem de resposta:

```

PROCEDURE resp
    (porta_entrada      : INTEGER;
     tamanho_mensagem  : INTEGER;
     endereço_mensagem : ADSMEN);

```

- Cancelamento de uma comunicação síncrona:

```

PROCEDURE resp_falha
    (motivo           : INTEGER;
     porta_entrada    : INTEGER);

```

- Desvio de uma comunicação síncrona:

```
PROCEDURE desvia
    (porta_saída      : INTEGER;
     porta_entrada    : INTEGER);
```

- Determinação da razão de uma falha:

```
FUNCTION motivo_falha : INTEGER;
```

- Determinação da quantidade de mensagens enfileiradas numa porta de entrada:

```
FUNCTION men_prt_ent
    (porta_entrada : INTEGER): INTEGER;
```

- Teste de conexão de uma porta de saída:

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

6.1.1.2. Serviços de Temporização

Os serviços de temporização especificados a seguir permitem a utilização do relógio tempo-real na aplicação e a suspensão temporária da execução de um módulo por um período de tempo. Os serviços oferecidos são:

- Leitura do relógio do núcleo:

```
FUNCTION relógio : INTEGER4;
```

- Retardamento de um módulo:

```
PROCEDURE retarda
    (período : INTEGER4);
```

6.1.1.3. Serviço para a Inicialização dos Parâmetros Reais de um Módulo

Esse serviço é usado pelo pré-compilador para informar ao núcleo, no início da execução do aplicativo, o endereço da área de parâmetros. Obtida essa informação, o núcleo pode proceder a carga dos parâmetros reais do módulo. A seguir, apresenta-se a interface correspondente:

```
PROCEDURE ini_par_mod
    (endereço_primeiro_parâmetro : ADSMEN);
```

6.1.1.4. Serviços para Tratamento Lógico de Interrupções

Os serviços para tratamento de interrupções dão suporte à função INTALOC e ao procedimento WAITIO, cujas interfaces são apresentadas a seguir:

- Mapeamento de um elemento do vetor físico de interrupções em um elemento do vetor lógico de interrupções do núcleo:

```
FUNCTION map_int
    (endereço_interrupção : BYTE) : INTEGER;
```

- Espera pela ocorrência de uma interrupção lógica:

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

6.1.1.5. Serviço de Alteração de Prioridade de um Módulo

O serviço cuja interface é apresentada a seguir, permite que a prioridade de um módulo seja alterada em tempo de execução.

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

6.1.1.6. Serviço de Identificação de um Módulo

O serviço de identificação de um módulo permite que, em tempo de execução, o módulo obtenha o número usado pelo tradutor LCM para identificá-lo. A interface correspondente a este serviço é apresentada a seguir.

```
FUNCTION num_mod : INTEGER;
```

6.2. Aspectos Relacionados com a Portabilidade

Alguns cuidados foram tomados na implementação com relação à portabilidade, em particular no que diz respeito às características dependentes de instalação. No caso específico das linguagens, procurou-se isolar a parte do código dependente de instalação, de forma que uma nova configuração em outro ambiente pode ser facilmente obtida através de ações bem definidas e rapidamente localizadas. As maiores dificuldades de transporte surgem em razão das características do Suporte de Tempo-Real, onde a dependência relativa ao hardware específico do am-

biente é inevitável. Visando minimizar estes efeitos, a implementação do Núcleo de Tempo-Real (ADAN COELLO, 1986) foi realizada usando-se a linguagem Pascal, ocorrendo apenas a programação em assembly quando a interação com o hardware e o Sistema Operacional hospedeiro não é possível realizar-se em alto nível. Em termos globais as linhas de código fonte Pascal do Núcleo correspondem a um percentual de 85% enquanto que as linhas de código assembly correspondem a 15%.

6.3. O Tradutor LPM

Como mostram as Figuras 6.1 e 6.2, o tradutor gera duas saídas para cada módulo LPM submetido: o módulo em código executável (módulo.EXE) e uma estrutura de dados chamada qualificador do módulo (módulo.QLF) que descreve as informações do módulo a serem utilizadas na configuração.

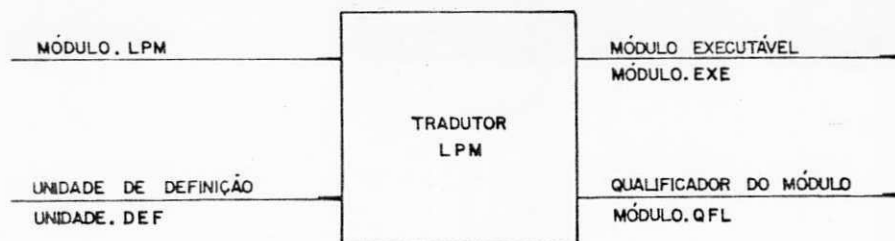


FIGURA 6.1 - Estrutura funcional do tradutor LPM

A unidade de definição fornecida ao tradutor (unidade.DEF) deverá conter as definições dos tipos usados na declaração de portas e mensagens do módulo. O Capítulo 3 descreve a estrutura de uma unidade de definição e faz algumas considerações sobre sua utilização.

As informações do módulo armazenadas no arquivo qualificador são as seguintes:

- número de parâmetros do módulo e tipos correspondentes;
- nomes das portas, tipos, número de referência, tamanho das mensagens associadas, tamanho da fila, quando trata-se de uma porta de entrada assíncrona, e limites de variação de índice, quando trata-se de uma família de portas.

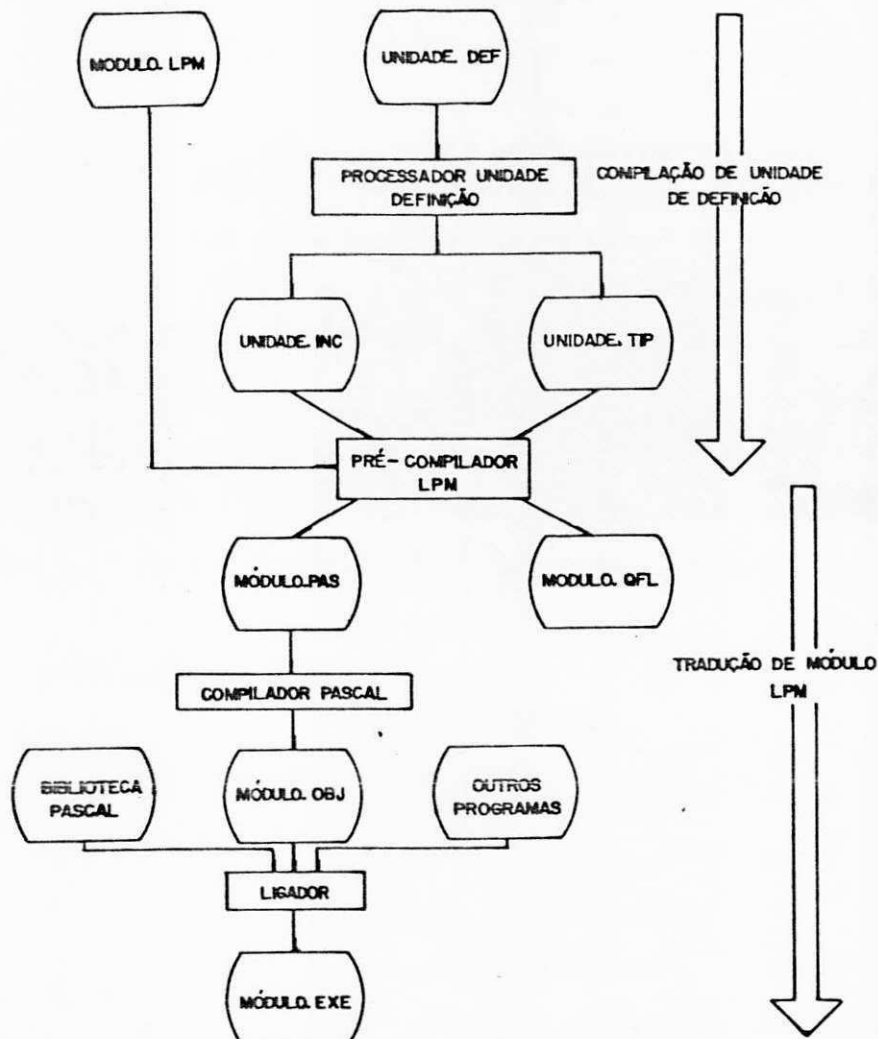


FIGURA 6.2 - Estrutura e relação de dependência entre os componentes do tradutor LPM

A implementação do tradutor LPM consiste basicamente de dois componentes: um processador de unidades de definição e um pré-compilador LPM.

Antes de ser referenciada por um comando LPM, uma unidade de definição deve ser previamente compilada pelo processador de unidade de definição, cuja função é verificar a especificação fornecida, obter os tamanhos correspondentes aos tipos das mensagens e gerar os arquivos: `unidade.Type` com os tamanhos dos tipos correspondentes às mensagens e `unidade.INC` com os tipos a serem incluídos por um comando LPM de importação de tipos.

6.3.1. O Processador de Unidades de Definição

Esta seção descreve o utilitário de processamento de unidades de definição a nível funcional. Os detalhes de implementação são abordados superficialmente, pois trata-se de um processo de natureza simples e escopo reduzido. Os passos envolvidos nesta etapa são os mesmos utilizados na implementação do pré-compilador LPM, cuja descrição detalhada é apresentada na seção seguinte (item 6.3.2).

A incorporação deste passo no processo de tradução, deve-se ao fato de que o núcleo precisa das informações relativas aos tamanhos das mensagens no instante da configuração, uma vez que as estruturas de dados associadas às portas são criadas naquele instante.

O Processador de Unidades de Definição compõe-se basicamente de um analisador léxico gerado pelo LEX e de um analisador sintático gerado pelo YACC (Figuras 6.3 e 6.4). Sua implementação consiste na geração de um programa Pascal que obtém os tamanhos correspondentes aos tipos especificados e cria um arquivo com as informações obtidas (`unidade.Type`). Tão logo seja gerado, o programa Pascal é compilado e executado. O arquivo que contém as definições a serem incluídas

(unidade.INC) é obtido através de cópia direta da unidade de definição, eliminando-se apenas as diretivas de controle de processador.

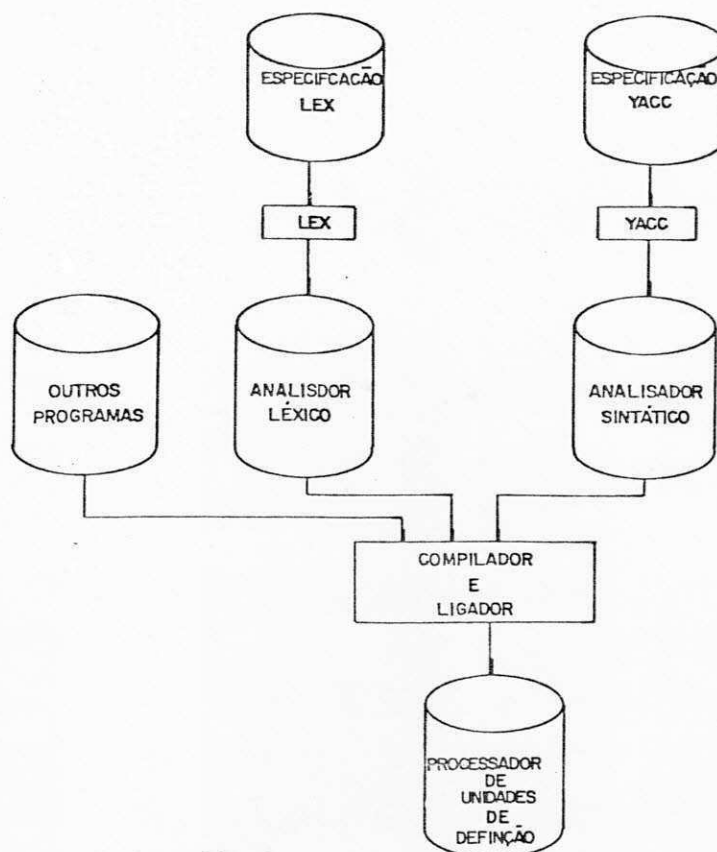


FIGURA 6.3 - Esquema de implementação do processador de unidades de definição

Muito embora atualmente não sejam feitas inclusões seletivas, onde apenas alguns objetos de uma unidade de definição são incluídos, a estruturação e uso do arquivo de definições (unidade.INC) favorece a implementação dessa facilidade, uma vez que o pré-processador conhece a organização e o formato do arquivo, o que pode proporcionar o desenvolvimento de técnicas otimizadas de acesso e manipulação dessas informações.

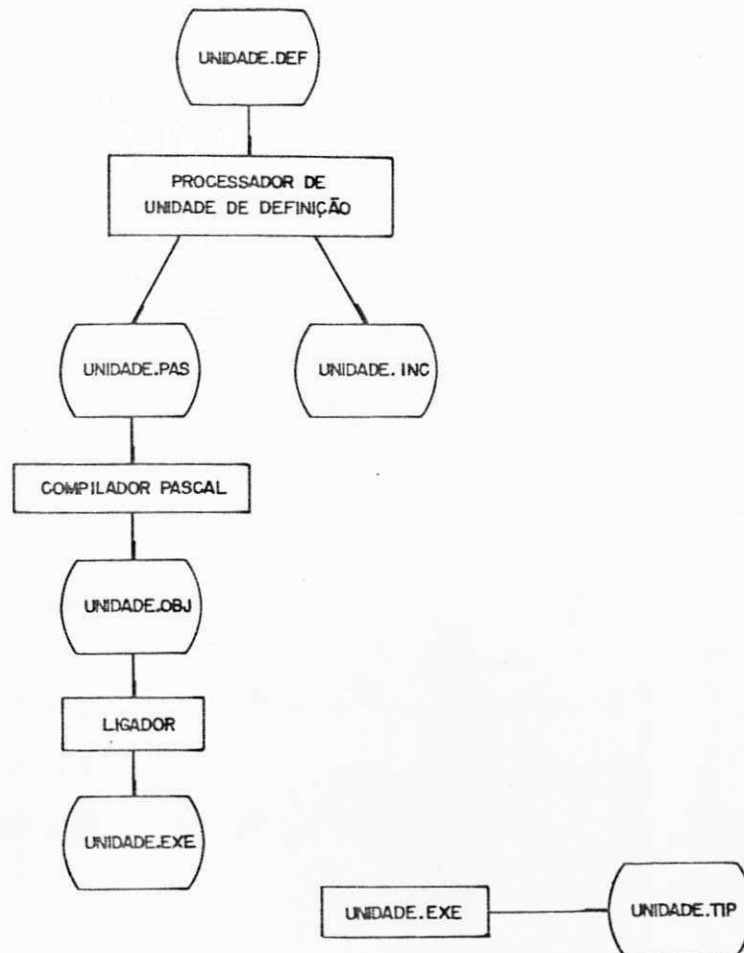


FIGURA 6.4 - Dependência entre os passos envolvidos no processamento de uma unidade de definição

Decidiu-se pela obtenção dos tamanhos dos tipos de mensagens através do Processador de Unidades de Definição ao invés de proceder uma única varredura no trecho do programa Pascal, por questões de simplicidade. Entendeu-se que isto não chegaria a onerar o processo de desenvolvimento pois, mesmo podendo uma unidade de definição ser usada por vários módulos LPM, a sua compilação é requerida apenas uma vez. Dessa forma o passo correspondente na tradução de módulos LPM é eliminado durante o ciclo de vida daquela unidade de definição.

6.3.2. O Pré-Compilador LPM

No modelo adotado para o núcleo tempo-real cada módulo LPM gera um programa Pascal, propiciando desta forma uma independência dos módulos entre si, e dos módulos em relação ao núcleo.

O atendimento às solicitações de serviços do núcleo é feito via uma interface constituída por procedimentos (seção 6.1.1); quando o pré-compilador encontra algum comando ou função LPM, ele gera uma chamada a um procedimento da interface, o qual suporta a execução da correspondente construção LPM.

O pré-compilador LPM consiste de um analisador léxico gerado pelo LEX e de um analisador sintático gerado pelo YACC, integrados segundo o esquema da Figura 6.5.

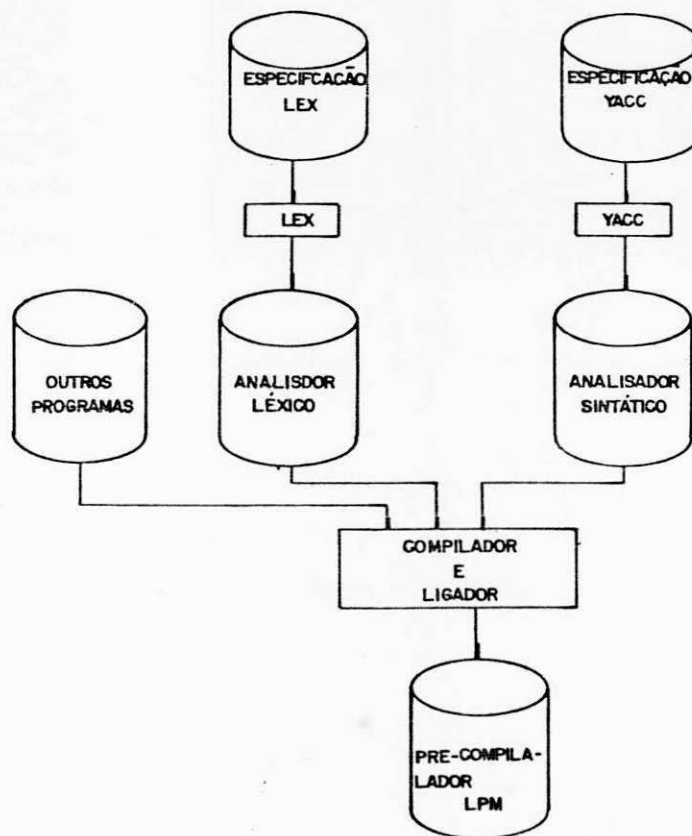


FIGURA 6.5 - Esquema de implementação do pré-compilador LPM

6.3.2.1. O Analisador Léxico

Na implementação do analisador léxico adotou-se o seguinte procedimento: Copiar no arquivo de saída, sem nenhum processamento, todos os símbolos que pertencem a comandos da linguagem Pascal e retornar para o analisador sintático apenas símbolos que compõem uma estrutura LPM.

As classes de itens léxicos reconhecidos pelo analisador léxico são as seguintes:

- Identificadores: qualquer sequência de letras, dígitos ou "_" que comece por letra e que não seja uma palavra reservada;
- Palavras reservadas: sequência de letras que pertence à tabela de palavras reservadas;
- Constantes numéricas: números inteiros decimais e reais decimais;
- Literais: uma sequência de caracteres delimitados pelo caracter """;
- Símbolos especiais: operadores e delimitadores de linguagem.

Quando o analisador léxico reconhece um identificador no arquivo de entrada, uma função de pesquisa binária é ativada para verificar se a cadeia de caracteres correspondente não constitui um elemento da tabela de palavras reservadas. Outra opção, para reconhecimento de palavras reservadas, seria considerar na especificação de entrada LEX uma

expressão regular para cada palavra. Esta solução não foi escolhida devido ao grande número de palavras reservadas da linguagem, o que faz aumentar de forma exponencial, o número de estados do autômato de reconhecimento, gerando conseqüentemente, um programa bem maior em termos de espaço de memória.

O tratamento de símbolos que compõem um comando LPM de forma diferenciada é possível devido ao uso de um campo adicional de informação à tabela de palavras reservadas. Esta informação consiste num valor lógico que identifica se a palavra-chave pertence ao vocabulário LPM ou ao vocabulário Pascal.

Uma vez obtido um símbolo, o analisador léxico verifica se o processamento corrente refere-se ao reconhecimento de um comando LPM; em caso afirmativo, são retornados para o analisador sintático as seguintes informações: o código inteiro do símbolo, a cadeia de caracteres correspondente e o número da linha em que ele aparece no arquivo fonte. Não se tratando de reconhecimento de um comando LPM, o analisador apenas copia na saída o símbolo recentemente obtido.

Esta estratégia de analisar somente parte do arquivo de entrada, usada em função do próprio conceito de um pré-processador, é dificultada em alguns casos, como por exemplo, em comandos LPM que permitem trechos do código Pascal como parte de suas especificações. A solução desses problemas pode ser vista nos itens 6.3.2.2 e 6.3.2.3 onde são descritos alguns aspectos relacionados com a implementação do analisador sintático e a geração do programa Pascal.

6.3.2.2. O Analisador Sintático

A implementação do analisador sintático consiste basicamente na descrição de uma especificação YACC, onde as fases de reconhecimento e tradução estão fortemente relacionadas através das regras de definição

da sintaxe. Apresentar-se-ão, a seguir, alguns aspectos estratégicos relacionados com a implementação do analisador sintático. A especificação YACC referente ao reconhecimento de algum comando LPM, juntamente com ações correspondentes às suas traduções são apresentadas no item 6.3.2.3.

Durante a fase de reconhecimento e tradução, várias informações a respeito das portas, mensagens e parâmetros que aparecem no módulo devem ser armazenadas e manipuladas pelas ações, tanto para verificação semântica como para tradução. As principais estruturas de dados utilizadas na implementação do analisador sintático são descritas a seguir:

- `blc_par`: é um registro descritor dos parâmetros do módulo; contém o número de parâmetros, a área ocupada em bytes, e o nome e tipo de cada parâmetro do módulo.
- `tab_portas`: tabela que mantém as informações relativas às portas declaradas no módulo; para cada porta são armazenados o nome, o número, a classe, o tipo de mensagem anunciada e tamanho, o tipo de sua porta, quando especificado, o tamanho da família e limites de variação do índice, quando trata-se de uma família, e o tamanho da fila, quando trata-se de uma porta de entrada assíncrona.
- `tab_mensagens`: tabela que mantém as informações relativas às mensagens declaradas; para cada mensagem são armazenados o tamanho e o tipo de mensagem.

A impressão de mensagens de erros é localizada num procedimento. Alguns mecanismos de recuperação foram implementados de forma que o reconhecimento é processado até o final do arquivo de entrada.

6.3.2.3. A Geração do Programa Pascal

A tradução de um módulo LPM para um programa Pascal é feita num único passo. A geração de código é realizada através da técnica de tradução dirigida pela sintaxe (AHO and ULLMAN, 1979). Neste esquema, as ações de tratamento semântico são colocadas em pontos adequados nas regras da gramática e, quando ativadas pelo analisador sintático, geram o código correspondente. Esse método é bastante conveniente, em particular quando o analisador sintático é gerado pelo YACC, porque possibilita a tradução em termos de especificação da sintaxe da linguagem.

Durante a fase de pré-compilação são realizados testes de consistência com o objetivo de detectar erros na programação dos módulos. Um dos testes que ocorre com bastante frequência consiste em verificar se a aplicação está usando corretamente uma determinada porta na troca de mensagens. A recepção de uma mensagem numa porta de saída, o envio de uma mensagem a uma porta de entrada, o uso de uma mensagem num comando de troca de mensagens cujo tipo é incompatível com o tipo da porta, a tentativa de realização de uma comunicação síncrona numa porta assíncrona e vice-versa, são exemplos de alguns erros detectados na fase de tradução.

A seguir, discute-se o modo como é realizada a tradução de alguns comandos LPM, a saber, os comandos SEND, RECEIVE, LOOP e SELECT. A descrição da tradução desses comandos consiste basicamente na apresentação das correspondentes especificações YACC utilizadas na implementação. A tradução dos outros comandos é bastante simples e facilmente dedutível a partir da definição da linguagem e da especificação da interface dos serviços do núcleo. Os símbolos terminais referenciados na especificação são constituídos por caracteres maiúsculos, enquanto que os símbolos não terminais por caracteres minúsculos. Com o objetivo de facilitar o entendimento, as ações não são apresentadas em forma de código na linguagem de programação C, mas sim em comandos equivalentes des-

critos numa pseudo-linguagem, cuja estrutura é bem mais legível. O Apêndice C traz alguns programas Pascal gerados pelo pré-compilador, os quais correspondem aos módulos LPM componentes do exemplo também apresentado no Apêndice C.

6.3.2.3.1. Tradução do Comando SEND

A tradução do comando SEND para a linguagem Pascal é bastante extensa e relativamente complexa devido à existência de três tipos de comunicação: assíncrona, síncrona e síncrona com cláusula de temporização.

Como mostra a Figura 6.6, o primeiro caso apresentado corresponde a um envio assíncrono de mensagens do tipo "SEND mensagem TO porta", onde simplesmente é feita uma tradução direta para a chamada de um procedimento da interface do núcleo. As informações relativas a número_porta, tamanho_mensagem e endereço_mensagem são obtidas à medida que o reconhecimento vai sendo processado.

A segunda alternativa apresentada corresponde a uma comunicação síncrona, onde nenhuma ação é explícita no caso da recepção ser confirmada, sendo a tradução feita também através da chamada de um único procedimento.

Nos demais casos aparece a componente IMPLICA que condiciona a execução de trechos de código à ocorrência de sucesso ou falha na comunicação. A maior complexidade surge em função da manipulação de vários arquivos de saída na geração do programa Pascal, provocada principalmente pela permissão de uso aninhado de comandos LPM. A parte correspondente aos comandos relacionados à ocorrência de falha é transcrita diretamente para saída corrente, a menos que exista algum comando LPM de troca de mensagens que necessite da criação de um novo arquivo temporário.

```

comando_send : prefixo_send
{
  se o tipo da porta é compatível com o tipo da
    mensagem
  então gerar ("env_a (número_porta, tamanho_mensa-
    gem, endereço_mensagem);")
  senão erro ("incompatibilidade de tipos em
    SEND");
}
| prefixo_send parte_wait
{
  se o tipo da porta é compatível com o tipo da
    mensagem e o tipo da parte reply da porta é
    compatível com o tipo da mensagem resposta.
  então gerar ("env_s (número_porta, tamanho_mensa-
    gem, endereço_mensagem, endereço_respos-
    ta);");
  senão erro ("incompatibilidade de tipos em
    SEND");
}
| prefixo_send parte_wait implica_w END_SEND
{
  assinalar o redirecionamento da saída para o an-
    tigo arquivo corrente;
  se o tipo_da_porta é compatível com o tipo da
    mensagem e o tipo da parte reply da porta é
    compatível com o tipo da mensagem resposta
  então
    início
      gerar ("env_s (número_porta, tamanho_
        mensagem, endereço_mensagem, en-
        dereço_resposta);");
      copiar na saída o arquivo temporário
        gerado;
    fim
  senão erro ("incompatibilidade de tipos em
    SEND");
}
| prefixo_send parte_wait implica_w parte_fail
{
  se o tipo_da_porta é compatível com o tipo da
    mensagem e o tipo da parte reply da porta é
    compatível com o tipo da mensagem resposta
  então
    início
      gerar ("IF (env_s_temp (tempo, número_
        porta, tamanho_mensagem, endere-
        ço_mensagem, endereço_resposta)
        THEN");

```

```

        /* início dos comandos da parte wait */
        gerar ("BEGIN");
        copiar o arquivo temporário gerado na
            saída;
        fim
        senão erro ("incompatibilidade de tipos em
            SEND");
    }
END_SEND
{
    /* encerramento dos comandos especificados na
        parte wait */
    gerar ("END");
    /* início dos comandos da parte fail */
    gerar ("ELSE BEGIN");
    copiar os símbolos da entrada no arquivo de saí-
        da;
    gerar ("END;");
}
;

prefixo_send : SEND ID
    {
        se ID é identificador de mensagem
        então guardar atributos da mensagem para usar
            na tradução
        senão erro ("mensagem não declarada em
            SEND");
    }
TO porta
    {
        se a classe da porta é entryport
        então erro ("envio de mensagem em porta de
            entrada")
        senão guardar atributos de porta para usar na
            tradução;
    }
;

parte_wait : WAIT ID
    {
        se ID é identificador de mensagem
        então guardar atributos da mensagem para usar
            na tradução
        senão erro ("mensagem não declarada na parte
            wait de um comando SEND");
    }

```

```

porta      : ID
            {
            se   ID não é identificador de porta
            então erro ("porta não declarada em comando
            de troca de mensagem")
            else obter os atributos da porta;
            }
índice_família
            {
            quando trata-se de família de portas com-
            putar o deslocamento na obtenção do núme-
            ro da porta;
            }

implica_w  : {
            criar      arquivo temporário para armazenar os
            comandos correspondentes a parte_wait
            do SEND;
            assinalar  o redirecionamento da saída para o ar-
            quivo temporário gerado;
            atualizar  controle para que o analisador léxico
            copie os símbolos seguintes diretamente
            na saída, ou seja, no arquivo tempore-
            rio recentemente criado;
            }

parte_fail  : FAIL expressão_opcional
            {
            guardar o valor da expressão para uso
            na tradução;
            }
IMPLICA
            {
            assinalar controle para o analisador léxico
            copiar os símbolos seguintes dire-
            tamente na saída;
            }

```

FIGURA 6.6 - Reconhecimento e tradução do comando SEND

6.3.2.3.2. Tradução do comando LOOP

A tradução do comando LOOP (Figura 6.7) também é simples e consiste basicamente em gerar um ciclo infinito. O encerramento da execução do ciclo pode ocorrer através da execução de um comando EXIT, cuja tradução corresponde à geração de um desvio para o fim do ciclo.

```

comando_loop : {
    assinalar controle para o analisador lé-
    xico copiar os símbolos seguin-
    tes diretamente na saída;
}
LOOP
{
    obter_rótulo;
    gerar ("WHILE true DO BEGIN");
}
comandos
END_LOOP
{
    gerar ("END;");
    gerar ("rótulo : ");
}

```

FIGURA 6.7 - Reconhecimento e tradução do comando LOOP

6.3.2.3.3. Tradução do Comando RECEIVE

A Figura 6.8 mostra como fica a tradução do comando RECEIVE, a qual não exhibe maiores dificuldades. A primeira regra especificada gera uma chamada ao serviço de recepção de mensagens, independentemente da comunicação ser síncrona ou assíncrona. Caso a especificação opcional do REPLY seja encontrada, quer na forma de "REPLY mensagem" ou "ABORT (porta, razão)", o procedimento de tradução é o mesmo, ou seja, simplesmente gera-se uma chamada ao serviço do núcleo correspondente.

```

comando_receive : RECEIVE ID
    {
        se ID é identificador de mensagem
        então guardar atributos de mensagem para
            uso na tradução
        senão erro ("mensagem não declarada em
            RECEIVE");
    }
FROM porta
    {
        se a classe da porta é exitport
        então erro ("recepção de mensagem em porta
            de saída")
        senão se o tipo da porta é compatível
            com o tipo da mensagem
            então gerar ("rec (endereço_mensagem,
                número_porta);")
            senão erro ("incompatibilidade de ti-
                pos em RECEIVE");
    }
reply_opcional
;
reply_opcional : /* VAZIO */
| comando_abort
| REPLY ID
    {
        se ID é identificador de mensagem
        então se o tipo da parte reply da porta é
            compatível com o tipo da mensagem
            então gerar ("resp (número_porta, tama-
                nho_mensagem, endereço_mensa-
                gem);")
            senão erro ("incompatibilidade de tipos
                na parte REPLY de um RECEIVE");
    }
;
comando_abort : ABORT ABRE_PARÊNTESES porta
    {
        se a classe da porta é exitport
        então erro ("tentativa de uso de abort em
            porta de saída")
    }
VÍRGULA
NÚMERO_INTEIRO FECHA_PARÊNTESES
    {
        gerar ("resp_falha (número_inteiro, número_
            porta);")
    }

```

FIGURA 6.8 - Reconhecimento e tradução do comando RECEIVE

6.3.2.3.4. Tradução do comando SELECT

A geração de código Pascal para o comando `SELECT` (Figura 6.9) envolve a combinação de várias alternativas de recepção de mensagens, através de portas de entrada, e tratamento de cláusulas de temporização (timeout). O esquema de tradução adotado consiste basicamente em gerar um comando `CASE` da linguagem Pascal, cuja expressão a ser avaliada em tempo de execução, corresponde a uma chamada ao serviço do núcleo (`sel`) responsável por selecionar uma recepção, ou encerrar a transação após a ocorrência de um período de tempo.

No instante da programação, o usuário pode especificar trechos de código, comandos Pascal ou LPM, cuja execução está condicionada à seleção de uma cláusula de recepção ou de temporização. A ocorrência de uma cláusula `else` é tratada pelo núcleo de forma equivalente ao reconhecimento de um "timeout 0", ou seja, se no instante da seleção não existe nenhuma mensagem disponível, o núcleo decide pela seleção da cláusula `else`, devendo então ser ativada a execução dos comandos correspondentes. Se na programação não é especificada a parte `else` do `SELECT` e nenhuma outra cláusula de temporização aparece, o núcleo associa um tempo de espera infinito, ou seja, a comunicação definida é totalmente síncrona.

Considerando a complexidade do próprio comando, adicionada à permissão de uso aninhado de estruturas LPM, a tradução do comando `select` é a que apresenta maior nível de dificuldade. Visando facilitar ainda mais o entendimento da especificação correspondente à Figura 6.9, utilizou-se na descrição de algumas ações o uso da pseudo-linguagem numa forma bastante verbosa e extensa.

Ao início do reconhecimento do comando, um arquivo temporário é criado para que sejam armazenados os vários comandos que compõem o `CASE` e, algumas variáveis de controle como por exemplo, `tipo_do_select`, `número_cláusula` e `rótulo_else` são inicializadas. A cada

ocorrência da palavra chave `or_select` na especificação, a variável `número_cláusula` é incrementada de forma que os comandos cuja execução é condicionada à seleção daquela cláusula possam ser identificados e agrupados, constituindo-se então no rótulo associado ao comando composto a ser ativado pelo CASE.

Quando a última cláusula do SELECT é encontrada (regra "`partes_select : parte_select`"), o rótulo `_else` é assinalado com o valor de `número_cláusula` adicionado de uma unidade, para que, em caso da parte `_else` do SELECT estar presente na especificação, o pré-compilador possa associar os comandos do `_else` ao trecho de código a ser ativado pelo CASE.

Como nos demais comandos onde aparece a componente **IMPLICA**, a dificuldade surge em decorrência das constantes permutas do arquivo de saída. Considerando que esta implementação permite a especificação de comandos aninhados até cinco níveis de profundidade, pode-se existir num dado instante cinco arquivos temporários abertos. A tradução de um comando `_recepção_seletiva` gera a chamada ao serviço `ini_rec_sel` no arquivo de saída do nível corrente, enquanto que os comandos condicionados à seleção da correspondente cláusula, são gerados na saída de nível subsequente ao nível corrente.

As cláusulas de recepção e temporização podem ser precedidas por um comando de repetição, permitindo dessa forma a especificação de múltiplas recepções ou classes de timeout numa única cláusula do `select`. Portanto, não basta apenas referenciar o número da cláusula para gerar código de ativação dos comandos associados à cláusula. É preciso também conhecer o valor da variável de controle correspondente à cláusula do SELECT selecionada. Como o usuário pode fazer uso dessa variável na sequência de comandos referente a cláusula, a primeira instrução Pascal gerada no comando composto do CASE corresponde à restauração desse valor, cuja descrição encontra-se na ação especificada para definição do símbolo não terminal `opção_sucesso`.


```

comando_select : tipo_select
{
    criar arquivo intermediário para armazenar os co-
    mandos relacionados após a ocorrência do
    símbolo IMPLICA("="), cuja execução está
    condicionada a satisfação de eventos do ti-
    po: a recepção foi efetivada, o tempo espe-
    cificado expirou ou nenhuma mensagem foi
    recebida;
    número_cláusula := 0;
    rótulo_else := 0;
}
partes_select
parte_else_select
END_SELECT
{
    /* var_for_ret é uma variável pré-definida
    pelo pré-compilador para retornar o valor
    do for correspondente a porta selecionada
    */
    gerar ("CASE sel(tipo_do_select, rótulo_el-
    se, var_for_ret, número_porta_selecio-
    nada) OF");
    copiar arquivo temporário gerado na saída
    (comandos associados as várias cláusu-
    las do select, cujo trecho a ser exe-
    cutado é determinado pelo CASE);
}
;

tipo_select : RSELECT
{
    assinalar tipo_do_select como randômico;
}
| PSELECT
{
    assinalar tipo_do_select como priorizado;
}
;

partes_select : parte_select
{
    rótulo_else := número_cláusula + 1;
    restaurar o nível corrente de saída com o valor
    do nível antecedente;
}

```

```

| partes_select OR_SELECT
  {
    número_cláusula := número_cláusula + 1;
  }
  parte_select
;

parte_select : repetição guarda cláusula_select opção_sucesso
;

repetição : /* VAZIO */
  {
    /* a variável nulo é definida previamente pelo
    pré-compilador */
    assinalar a variável nulo como variável_do_for;
  }
| FOR ID
  {
    assinalar a variável ID como variável_do_
      for;
    copiar na saída "FOR" e ID;
  }
  ATRIBUIÇÃO expressão_int
  {
    copiar na saída ":@" e expressão_int
  }
  variação_passo
  {
    copiar na saída o símbolo "TO" ou
      "DOWNTO"
  }
  expressão_int
  {
    copiar na saída expressão_int;
  }
  DO
  {
    copiar na saída o símbolo "DO";
  }
;

guarda : /* vazio */
| WHEN expressão_lógica
  {
    gerar ("IF expressão_lógica THEN");
  }
;

```

```

cláusula_select : comando_recepção_seletiva
| TIMEOUT expressão_int
{
    gerar ("ini_temp_sel (variável_do_for, número_
           cláusula, expressão_int);");
}
;

comando_recepção_seletiva : /* o comando de recepção seletiva é de-
                             finido com a mesma sintaxe do comando_receive
                             apresentado no item 6.3.2.3.2, apenas mudando a
                             ação semântica de geração de código correspondente
                             ao reconhecimento da estrutura "RECEIVE ID
                             from porta". A ação equivalente no comando select
                             é:
                             gerar ("ini_rec_sel (variável_do_for, número_
                             cláusula, número_porta, endereço_mensagem);");
                             */
;

opção_sucesso : /* VAZIO */
|
{
    se a variável_do_for não é a variável nulo
    então gera_no_arquivo_temporário ("variável_
                                       do_for := var_for_ret);
    assinalar o redirecionamento da saída para o ar-
    quivo temporário e atualizar controle
    para que o analisador léxico copie os
    símbolos seguintes diretamente na saída;
}
IMPLICA comandos
;

parte_else_select : /* VAZIO */
{
    rótulo_else := 0;
}
|
{
    assinalar controle para o analisador lé-
    xico copiar os símbolos seguin-
    tes diretamente na saída, ou
    seja, no arquivo temporário,
}
ELSE_SELECT
;

```

FIGURA 6.9 - Reconhecimento e tradução do comando SELECT

Caso a parte `else` do `SELECT` não seja especificada no comando, a variável `rótulo_else` é assinalada com o valor zero; essa convenção é usada para o núcleo tomar conhecimento da não existência da cláusula `_else`. Se o símbolo `ELSE_SELECT` é reconhecido, o analisador léxico passa a copiar os símbolos da entrada na saída (arquivo temporário) até que seja encontrado o símbolo `END_SELECT`. Nesse instante já foram geradas no arquivo corrente de saída todas as chamadas aos respectivos serviços do núcleo responsáveis pelas inicializações das cláusulas especificadas. O tradutor gera então o comando `CASE`, onde a avaliação da expressão associada, em tempo de execução, é obtida através da ativação da função `sel`, a qual retorna o número da cláusula selecionada e o valor da variável de controle do comando de repetição. Os comandos referentes ao `CASE` são gerados através de simples cópia do arquivo temporário (nível subsequente ao arquivo corrente de saída) para o arquivo de saída. O exemplo apresentado no Apêndice C mostra a tradução completa de alguns módulos LPM onde pode-se ver o código Pascal gerado para um comando `SELECT`.

6.3.2.4. Recuperação de Erros

Considerando que a maioria dos programas apresenta algum grau de incorreção, procurou-se dotar o pré-compilador de meios para tratar os erros detectados durante o processamento dos módulos LPM.

Devido à sua natureza não determinística, o tratamento de erros em reconhecimento de linguagens é um processo muito complexo envolvendo tratamento semântico na maioria dos casos. Com relação a LPM, o pré-compilador ao encontrar um erro indica as proximidades em que ele ocorreu, emite uma mensagem com as suas características e continua o reconhecimento até o final do arquivo fonte. Isto é implementado usando-se os mecanismos do YACC orientados à recuperação de erros, os

quais consistem basicamente em referenciar o símbolo terminal **error** nas regras da gramática, indicando desta forma os locais potenciais onde erros podem acontecer. A cada ocorrência de erro no processo de pré-compilação, uma ação de recuperação é ativada.

Basicamente os erros são classificados em três classes: erros léxicos, erros sintáticos e erros semânticos. Os erros léxicos são aqueles decorrentes da codificação incorreta de um símbolo terminal ou do uso de caracteres inválidos para a linguagem. Os erros sintáticos estão relacionados com o desrespeito às regras gramaticais da linguagem na elaboração de um programa enquanto que, os erros semânticos ocorrem quando o programador não obedece as definições semânticas da linguagem, como por exemplo, o uso de uma porta não declarada num comando de troca de mensagens.

Considerando que a diretriz básica do analisador léxico é reconhecer apenas símbolos que compõem estruturas LPM, nenhum erro léxico é detectado pelo pré-compilador. Os demais símbolos são copiados diretamente da entrada para a saída sem qualquer processamento. Os eventuais erros léxicos que possam existir na especificação de um módulo LPM, são detectados no passo seguinte, ou seja, durante a compilação do programa Pascal gerado.

Os erros sintáticos são tratados usando-se o símbolo **error**, reservado pelo YACC para este fim, nas regras da definição da linguagem. Estratégias tipo descartar os símbolos seguintes ao erro até encontrar um determinado delimitador, são usadas para posicionamento da continuação da tradução. Adotou-se como princípio básico na recuperação de um erro sintático, que o número de símbolos descartados na entrada deve ser o menor possível. O símbolo **error** foi então posicionado em todas as definições recursivas e em partes localizadas onde erros podem ocorrer.

O tratamento de um erro semântico no pré-compilador LPM consiste basicamente na sinalização do erro, e apesar do reconhecimento continuar sendo processado após seu diagnóstico sem que nenhum símbolo da entrada seja descartada, o tradutor não gera código algum para a estru-

tura reconhecida.

6.4. O Tradutor LCM

O tradutor LCM processa um programa de configuração que especifica os módulos componentes da aplicação e a interligação de suas portas, produzindo um mapa de configuração. Este mapa de configuração é basicamente uma tabela estruturada que contém as informações requisitadas pelo núcleo para efetivar a configuração física da aplicação. A Figura 6.10 apresenta um bloco funcional do tradutor implementado onde são mostradas as informações usadas e produzidas.

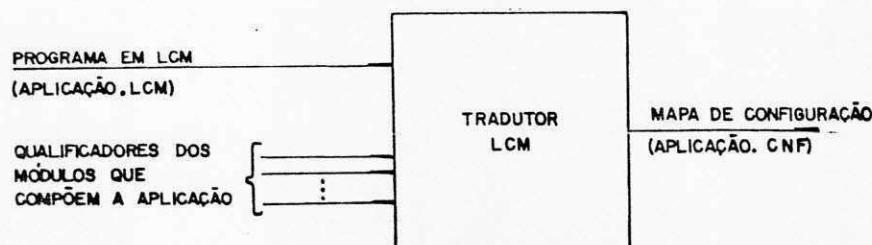


FIGURA 6.10 - Estrutura funcional do tradutor LCM

O mapa de configuração, gerado num arquivo com extensão.CNF, define a imagem da estação a ser configurada. A execução de uma configuração é precedida por um processo de inicialização onde o núcleo temporal varre o mapa de configuração, obtém as informações relativas aos módulos componentes, carrega-os na memória da estação e faz as interconexões especificadas, criando desta forma a imagem executável correspondente à configuração física da aplicação submetida.

O esquema utilizado na implementação do tradutor LCM é similar ao apresentado na implementação do pré-compilador LPM, cuja descrição

pode ser vista no item 6.3.2. A implementação compõe-se basicamente de um analisador léxico e um analisador sintático, gerados também pelo LEX e YACC, respectivamente. Considerando que esta implementação está limitada à efetivação de configurações estáticas, o processo de tradução é relativamente simples, consistindo apenas na geração do mapa de configuração, onde várias informações a respeito dos módulos e de suas respectivas conexões devem ser armazenadas durante o processo de tradução. A Figura 6.11 mostra a organização lógica das informações que compõem o mapa de configuração.

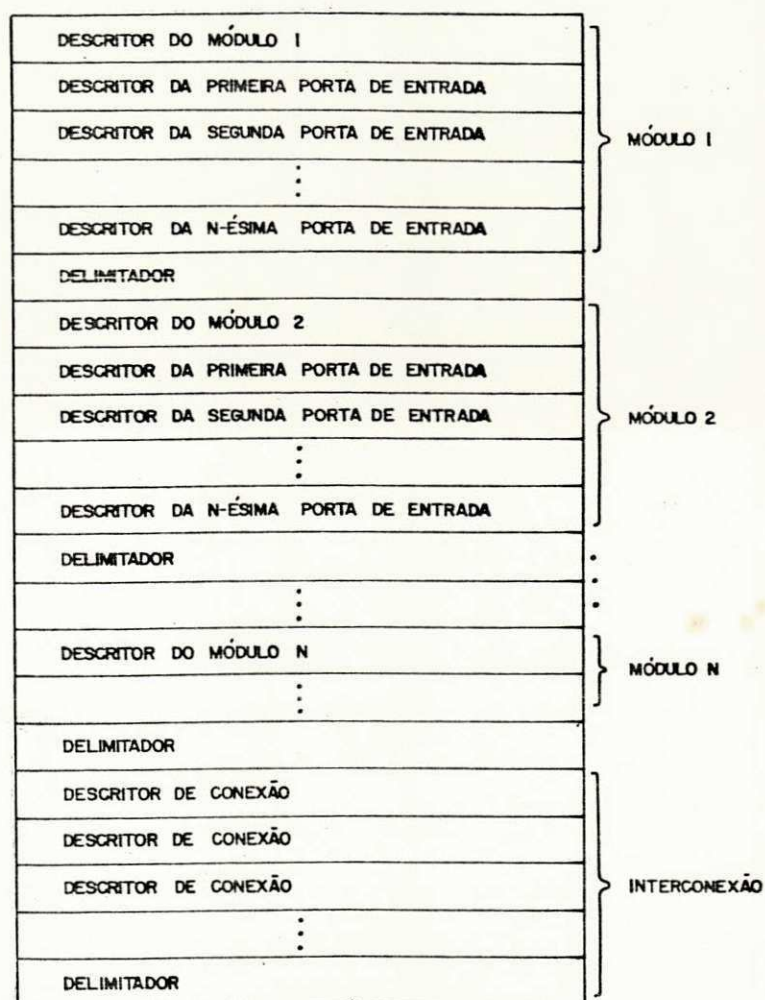


FIGURA 6.11 - Mapa de configuração

As descrições das estruturas de dados usadas na implementação para a geração do mapa de configuração, são apresentadas a seguir:

- descritor de módulo: apontador para um registro que descreve o módulo. Essa descrição contém o número do módulo, o nome do arquivo correspondente ao código executável, o número de parágrafos reservado para pilha e heap, a prioridade, o número de portas (entrada e saída), o tamanho da área de parâmetros e os respectivos parâmetros.
- descritor de porta de entrada: registro que contém o número da primeira porta da família (portas elementares são vistas como famílias de um único elemento), o número de portas da família, o número de mensagens associadas à fila de entrada, o tamanho dos elementos da fila em bytes.
- descritor de conexão: registro que contém o número do módulo fonte, o número da porta de saída do módulo fonte, o número do módulo destino e o número da porta de entrada do módulo destino.

A tradução de um programa LCM consiste em reconhecer uma especificação e obter as informações necessárias à geração do mapa de configuração. À medida que o reconhecimento vai sendo processado, alguns testes de consistência são realizados. Esses testes consistem em verificar se os parâmetros reais especificados no programa LCM são compatíveis com os parâmetros formais definidos no módulo LPM, e se as conexões especificadas correspondem a portas de mesmo tipo e classes complementares, ou seja, uma conexão somente é válida se for orientada de uma porta de saída para uma porta de entrada.

Ao término do reconhecimento, caso não haja ocorrência de erros, o tradutor LCM descarrega o conteúdo do mapa de configuração num arqui-

vo binário de saída (Aplicação.CNF). A execução de uma aplicação é ativada pelo núcleo quando recebe o mapa de configuração correspondente. Nesse instante o núcleo inicializa as suas estruturas de dados e carrega os módulos componentes da aplicação a fim de proceder efetivamente a sua execução. O Apêndice C apresenta um exemplo completo envolvendo o programa de alguns módulos e uma possível configuração desses módulos.

7. CONCLUSÃO

Muito embora os sistemas distribuídos apresentem características bastante atraentes, em particular para aplicações de tempo-real, não encontram-se disponíveis ferramentas capazes de explorar adequadamente essas características. Este trabalho procurou explorar as potencialidades dos sistemas distribuídos implementando um ambiente que integra uma metodologia apropriada e ferramentas de desenvolvimento.

O ambiente implementado é caracterizado por permitir a separação das etapas de programação e configuração no desenvolvimento de uma aplicação. Para isto são fornecidas duas linguagens: a Linguagem de Programação de Módulos e a Linguagem de Configuração de Módulos. Esta característica possibilita a incorporação de mecanismos de reconfiguração dinâmica, e conseqüentemente tolerância a falhas.

A implementação das linguagens foi realizada em um computador PCS CADMUS-9200 através do uso das ferramentas YACC e LEX e atualmente está disponível para executar em ambiente compatível com IBM-PC. Esta opção deveu-se à grande utilização de microcomputadores IBM-PC nas mais diversas áreas de aplicação, e em particular à sua crescente aceitação nos meios industriais.

A linguagem LPM foi implementada na forma de um pré-compilador, onde as extensões da linguagem Pascal são traduzidas para chamadas de procedimentos correspondentes aos serviços do núcleo tempo real. O tradutor LCM gera um mapa de configuração a partir da especificação LCM fornecida, o qual é utilizado pelo núcleo para efetivar a configuração física da aplicação, e conseqüentemente poder colocá-la em operação.

A atual versão das linguagens suporta apenas configuração estática e processamento centralizado, e juntamente com o núcleo tempo-real

desenvolvido por (ADAN COELLO, 1986) constitui o protótipo inicial de validação das idéias propostas. Este fato não chega a comprometer os objetivos do projeto, uma vez que a programação de uma aplicação no ambiente independe do fato da arquitetura ser centralizada ou distribuída.

* Uma evolução consequente do trabalho é o interfaceamento do ambiente a uma rede local, possibilitando desta forma a execução distribuída de aplicações. A nível das linguagens, a incorporação de novos mecanismos é feita de forma bastante simples, bastando para tanto que o núcleo ofereça os serviços correspondentes.

Como haveria de se esperar, a opção de implementar a LPM através de um pré-compilador impõe algumas restrições no sentido de que ela implica em um nível considerável de redundância na tradução, bem como força a tomada de algumas decisões que afetam em alguns casos a homogeneidade da definição da linguagem. Isto ocorre devido a necessidade de inserir alguns mecanismos de identificação de trechos particulares de código. Considerando que o objetivo do corrente trabalho foi criar mecanismos de validação das idéias apresentadas na dissertação, esta decisão não chega a ser comprometedora e constituiu numa forma factível de resolver o problema no decorrer de um tempo consideravelmente curto.

A nível de continuidade do trabalho prevê-se uma fase de testes exaustivos, incluindo a possibilidade de integrar o ambiente implementado a nível de alguns projetos em desenvolvimento no CTI. Existe também uma expectativa com relação ao uso das ferramentas por parte de algumas empresas, as quais se ressentem da ausência de um ferramental adequado às suas aplicações. A curto prazo, espera-se fazer alguns experimentos num ambiente constituído por um conjunto de microcomputadores IBM-PC ligados serialmente, os quais poderão eventualmente estar conectados a um **hardware** de controle de processos, desempenhando, por exemplo, funções de interface Homem/Máquina.

Paralelamente a esta implementação existe em andamento no Instituto de Computação do Centro Tecnológico para Informática (CTI), o de-

envolvimento de um compilador Pascal para microcomputadores IBM-PC. Nesse sentido propõe-se que sejam feitos estudos visando desenvolver um compilador para a linguagem LPM a partir da incorporação das construções LPM à definição da linguagem Pascal usada na implementação do compilador.

Esta integração poderá ser feita de forma relativamente simples, uma vez que a característica fundamental do projeto do compilador Pascal consiste na utilização de uma linguagem intermediária, estruturada na forma de árvore decorada, onde são armazenadas as informações semânticas relativas aos operandos e operadores, a partir da qual procede-se a análise semântica e a geração de código. A integração poderia ser feita através da incorporação das novas construções na gramática, da implementação dos reconhecedores sintático e semântico correspondentes às novas construções, e da modificação do gerador de código no sentido de ampliar o seu universo de construções aceitas.

Como conclusão final do trabalho vale a pena enfatizar os aspectos relacionados com a experiência e os conhecimentos adquiridos em sua realização.

A P Ê N D I C E A

DESCRIÇÃO BNF DAS LINGUAGENS LPM E LCM

Neste apêndice são apresentadas as descrições sintáticas das linguagens LPM e LCM. Para representação da sintaxe utiliza-se uma notação BNF estendida onde:

- [<construção>] : representa uma construção opcional
- [<construção>]* : representa zero ou mais repetições da construção

Na descrição correspondente a LPM apenas serão apresentadas as construções das extensões à linguagem PASCAL. As construções não definidas neste apêndice podem ser obtidas em (ISO, 1983).

A.1 - Linguagem de Programação de Módulos

DEFINIÇÃO DE UM MÓDULO

```
<definição de um módulo> ::=  
  MODULE <identificador> [(<parâmetros formais>)];  
    [<importação definições>]  
    [<declaração de portas>]  
    [<declaração de mensagens>]  
  <bloco LPM>.
```

```

<parâmetros formais> ::=
    <grupo de parâmetros> [; <grupo de parâmetros>]
<grupo de parâmetros> ::=
    <lista de identificadores> : <tipo básico PASCAL>
<bloco LPM> ::=
    [<declaração de rótulos>]
    [<declaração de constantes>]
    [<declaração de tipos>]
    [<declaração de variáveis>]
    [<declaração de funções e procedimentos>]
    BEGIN_MODULE <comandos> END_MODULE

```

IMPORTAÇÃO DE DEFINIÇÕES

```

<importação definições> ::=
    USE <nome arquivo definições>;

```

DECLARAÇÃO DE PORTAS

```

<declaração de portas> ::=
    ENTRYPORT <decl. porta entrada>
        [; <decl. porta entrada>]*; |
    EXITPORT <decl. porta saída> [; <decl. porta saída>]*;

```

DECLARAÇÃO PORTA ENTRADA

```

<decl. porta entrada> ::=
    <declarador> [, <declarador>] : <tipo mensagem>
        [<parte reply> | <parte queue>]
<declarador> ::=
    <identificador> [ "[" <família> "]" ]
<família> ::=
    <inteiro decimal> .. <inteiro decimal>
<tipo mensagem> ::=
    <tipo básico PASCAL> | <identificador tipo mensagem>

```

```

<parte reply> ::=
    REPLY <tipo mensagem>
<parte queue> ::=
    QUEUE <inteiro decimal>

```

DECLARAÇÃO PORTA SAÍDA

```

<decl. porta saída> ::=
    <declarador> [, <declarador>] : <tipo mensagem>
    [<parte reply>]

```

DECLARAÇÃO DE MENSAGENS

```

<declaração de mensagens> ::=
    MESSAGE <declarador de mensagens>
        [; <declarador de mensagens>];
<declarador de mensagens> ::=
    <identificador> [, <identificador>] : <tipo mensagem>

```

COMANDOS

```

<comandos> ::=
    <comando PASCAL> | <comando LPM>
<comando LPM> ::=
    <comando LOOP> | <comando SEND> |
    <comando RECEIVE> | <comando REPLY>
    <comando SELECT> | <comando FORWARD>
    <comando EXIT>

```

COMANDO LOOP

```

<comando LOOP> ::=
    LOOP <comandos> END_LOOP
<comando EXIT> ::= EXIT

```

COMANDO SEND

```

<comando SEND> ::=
    SEND <ident. mensagem> TO <ident. porta saída>
        [<cláusula resposta>]
<cláusula resposta> ::=
    WAIT <ident. mensagem> [<opções de resposta> END_SEND]
<opções de resposta> ::=
    => <comandos> [<tratamento falha>] | <tratamento falha>
<tratamento falha> ::=
    FAIL [<tempo>] = <comandos>
<tempo> ::= <inteiro decimal>

```

COMANDO RECEIVE

```

<comando RECEIVE> ::=
    RECEIVE <ident. mensagem> FROM <ident. porta entrada>
        [REPLY <ident. mensagem>]

```

COMANDO REPLY

```

<comando REPLY> ::=
    REPLY <ident. mensagem> TO <ident. porta entrada>

```

COMANDO SELECT

```

<comando SELECT> ::=
    <tipo de seleção> <parte select>
        [OR_SELECT <parte select>]*
        [else_select <comandos>]
    END_SELECT
<tipo de seleção> ::=
    PSELECT | RSELECT
<parte select> ::=
    [<repetição>] [<guarda>] <cláusula de seleção>
        [= > <comandos>]

```



```

<repetição> ::=
    FOR <variável de controle> := <valor inicial>
        <TO | DOWNTO> <valor final> DO
<guarda> ::=
    WHEN <expressão booleana>
<cláusula de seleção> ::=
    <comando RECEIVE> | <cláusula de tempo>
<cláusula de tempo> ::=
    TIMEOUT <unidades de tempo>
<unidades de tempo> ::=
    <expressão inteira>

```

COMANDO FORWARD

```

<comando FORWARD> ::=
    FORWARD <ident. porta entrada> TO <ident. porta saída>

```

A.2 - Linguagem de Configuração de Módulos

PROGRAMA DE CONFIGURAÇÃO

```

<programa de configuração> ::=
    CONFIGURATION <identificador>;
    <declaração de instâncias>
    <criação de instâncias>
    [<conexão de instâncias>]
    END_CONFIG.

```

DECLARAÇÃO DE INSTÂNCIAS

```

<declaração de instâncias> ::=
    INSTANCE <declarador instâncias> [; <declarador instâncias>]*;

```

UNIVERSIDADE FEDERAL DA PARAÍBA
 Pró-Reitoria Para Assuntos do Interior
 Coordenação Setorial de Pós-Graduação
 Rua Aprígio Veloso, 882 - Tel (083) 321-7222-R 355
 58.100 - Campina Grande - Paraíba

```

<declarador instâncias> ::=
    <identificador> [, <identificador>]* : <tipo do módulo>

```

CRIAÇÃO DE INSTÂNCIAS

```

<criação de instâncias> ::=
    CREATE <instância> [, <instância>]*;
<instância> ::=
    <ident. instância> [( <parâmetros reais módulo> )]
<parâmetros reais módulo> ::=
    <constante inteira> | <constante real>
    <constante lógica> | <caracter>

```

CONEXÃO DE INSTÂNCIAS

```

<conexão de instâncias> ::=
    LINK <porta saída> TO <porta entrada>
    [, <porta entrada>]*;

```

PORTA SAÍDA E PORTA ENTRADA

```

<porta saída> ::=
    <ident. instância> . <ident. porta saída>
    [ "[" <índice> "]" ]
<porta entrada> ::=
    <ident. instância> . <ident. porta entrada>
    [ "[" <índice> "]" ]

```

A P Ê N D I C E B

SERVIÇOS OFERECIDOS PELO SUPORTE DE TEMPO-REAL (STR)

Este apêndice apresenta uma descrição da interface dos serviços do núcleo Tempo-Real, a qual constitui a secção 6.1.1 da dissertação de mestrado desenvolvida por (ADAN COELLO, 1986).

B.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.

B.1.1 - Envio Assíncrono de uma Mensagem

O comando de envio assíncrono de uma mensagem

```
SEND <mensagem> to <porta_de_saída>
```

é suportado pelo serviço `env_a` cuja interface é dada por:

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

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

B.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:

```
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>.

B.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_mem:  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 quê deve-se executar o comando `S2`. Caso a cláusula `FAIL` do comando `SEND` não inclua um tempo de espera, indicando um tempo de espera ilimitado, o parâmetro `tempo` deve ser um número negativo.

A função `env_s_temp` que implementa o envio síncrono temporizado retornará o valor `TRUE` caso receba uma mensagem dentro do período especificado e o valor `FALSE` em caso contrário. Em função do valor retornado 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
```

B.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:

```
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` dá o número da porta de entrada através da qual a mensagem deve ser recebida.

B.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:

```
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.

B.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:

```
PROCEDURE desvia (prt_saída, 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.

B.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 controle do FOR de uma alternativa de um comando SELECT que contém um RECEIVE, como por exemplo:

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

é gerada uma chamada ao serviço ini_rec_sel cuja interface é dada por:

```
PROCEDURE ini_rec_sel (índice_for, cláusula, prt_entrada:
        INTEGER; end_mem: 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_SELECT FOR <variável de controle>:=...TO...
        TIMEOUT <tempo de espera>
        => S2
END_SELECT
```


é gerada uma chamada ao serviço `ini_temp_sel` cuja interface é dada por:

```
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 `índice_for`.

A fase de seleção de um comando `SELECT` corresponde à requisição do serviço `sel` cuja interface é dada por:

```
FUNCTION sel (tipo: CHAR; cláusula_else: INTEGER;
              VAR índice_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 comando `SELECT`. Caso não haja uma cláusula `ELSE` o parâmetro `clausula_else` conterà o valor zero. A função `sel` escolhe uma cláusula do comando `SELECT` e retoma o valor inteiro que lhe foi associado durante a fase de inicialização. Adicionalmente em `índice_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',...)` e `sel('R',...)` 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.

B.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 resp_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`.

B.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.

B.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.

B.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>.

B.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.

B.3 - Serviços de Temporização

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

B.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:

```
FUNCTION relógio: LONGINTEGER;
```

B.3.2 - Retardamento de um Módulo

O procedimento 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:

```
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.

B.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;
```

B.4 - Interrupções

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

B.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`.

B.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 aguardar.

A P Ê N D I C E C

UM EXEMPLO DO USO DAS LINGUAGENS

Este apêndice ilustra, através de um exemplo de (ADAN COELLO, 1986). O processo de programação, configuração e execução de aplicações em LPM e LCM. Os módulos componentes do exemplo 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 são também apresentados.

O exemplo introduz 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 sequenciar 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 na criação de servidores que implementem serviços sem a intermediação de um sistema operacional hospedeiro, no caso o MS-DOS.

Os módulos do exemplo têm como único propósito ilustrar o tipo de programação obtido com a LPM e LCM não implementando nenhum algoritmo específico.

O exemplo é formado por 5 tipos de módulos: A, B, C, D e TIMEMAN. Estes módulos serão descritos em C.1, a configuração do exemplo a partir destes 5 tipos básicos é apresentada em C.2.

C.1 - Módulos Componentes

O código LPM do módulo do tipo A, referenciado a seguir apenas por módulo A, é ilustrado na Figura C.1. A definição de contexto do módulo especifica objetos da unidade de definição TipMen. A unidade de definição TipMem, mostrada na Figura C.2, define os tipos das mensagens que serão usadas para comunicação e sincronização entre os módulos da aplicação.

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 MODULE_ID. O instante em que a mensagem é enviada é obtido lendo o relógio do núcleo 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.

```

MODULE a(pausa:INTEGER);

  USE TipMen.INC;

  EXITPORT
    ps1:ident REPLY pausa_prio;
    ps2:ident;

  MESSAGE
    men_ident:ident;
    men_pausa_prio:pausa_prio;

  VAR
    i:INTEGER;

($PAGE+)
  BEGIN_MODULE
    men_ident.modulo:=MODULE_ID;
    men_ident.num:=0;
    LOOP
      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:=men_ident.num+1;
        SEND men_ident TO ps1 WAIT men_pausa_prio;
        pausa:=men_pausa_prio.pausa;
        SETPRIORITY(men_pausa_prio.prio);
      END_LOOP;
  END_MODULE.

```

FIGURA C.1 - Código LPM do módulo tipo A

```

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

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

```

FIGURA C.2 – A unidade de definição TipMen

O módulo tipo B, mostrado na Figura C.3, usa a mesma unidade de definição do módulo tipo A e tem o seguinte ciclo de execução: O módulo faz uma recepção bloqueante através da porta síncrona de entrada *pe1*, recebida uma mensagem o módulo registra o momento em que isso ocorreu no campo instante da mensagem *mem_identB* 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 *pe1*.

```

MODULE b;

USE TipMen.INC;

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_MODULE
    men_identB.modulo:=MODULE_ID;
    men_identB.num:=0;
    LOOP
        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);
        END_LOOP;
    END_MODULE.

```

FIGURA C.3 - Código LPM do módulo tipo B

O módulo tipo C, mostrado na Figura C.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 solicitação é feita mediante comandos WRITE e READ do Pascal que por "default" escrevem na tela e lêem do teclado. A execução desses comandos implicará na requisição de serviços

```

MODULE c(espera_sel:integer);

USE TipMen.INC;

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_mem:3,' pediu a pausa e a prioridade? ');
    READLN(pausa,pri);
END;

BEGIN_MODULE
    num_modulo:=MODULE_ID;
    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_SELECT
                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_SELECT
                TIMEOUT espera_sel
                => inst:=TIME;
                WRITELN('Timeout em ',inst:4);
            END_SELECT;
            inst:=TIME;
            LePausaPrio(num_modulo,inst,0,espera_sel,prioridade);
            SETPRIORITY(prioridade);
        UNTIL FALSE;
    END_MODULE.

```

FIGURA C.4 - Código LPM do módulo tipo C

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 C.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 TipMen.INC;

    ENTRYPORT
        pe1:ident;

    EXITPORT
        ps1:ident;

    MESSAGE
        men_ident:ident;

    BEGIN_MODULE
        REPEAT
            RECEIVE men_ident FROM pe1;
            SEND men_ident TO ps1;
        UNTIL FALSE;
    END_MODULE.

```

FIGURA C.5 - Código LPM do módulo tipo D

O módulo tipo TIMEMAN, mostrado na Figura C.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_MODULE
    int_log_rtr:=INTALLOC(vrtr);
    REPEAT
      FOR i:=1 TO pulsos DO WAITIO(int_log_rtr);
      TICK;
    UNTIL FALSE;
  END_MODULE.

```

FIGURA C.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 em C.2.

C.2 - Configuração

A Figura C.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 C.8.

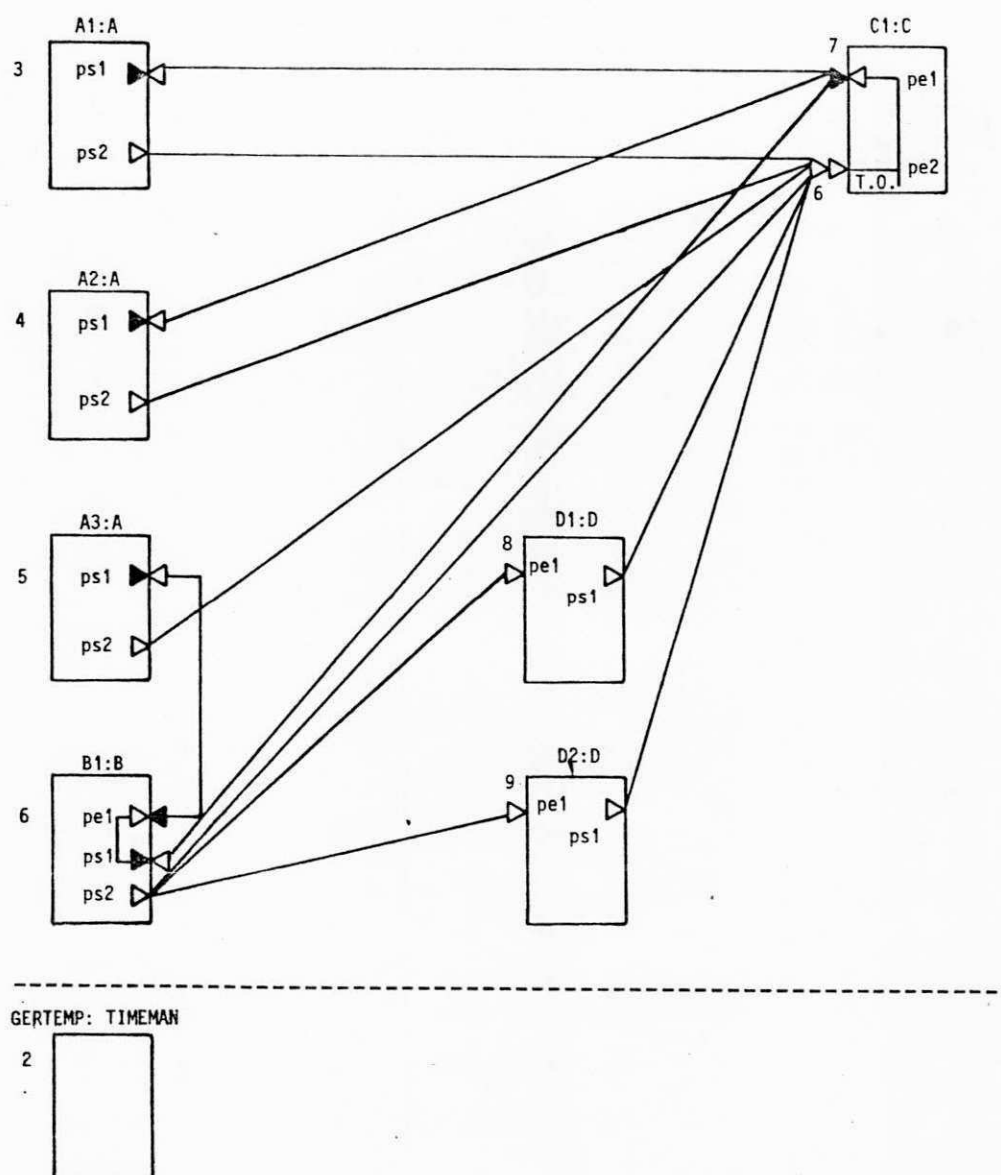


FIGURA C.7 - Diagrama de configuração do exemplo


```

CONFIGURATION exemplo;
  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_CONFIG

```

FIGURA C.8 - Programa em LCM para a configuração do exemplo

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 de 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 `highpr`, 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 prioridade 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 atendimento 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 da porta `pe2` poderia ficar cheio e, em consequência, as mensagens mais antigas passariam a ser substituídas por aquelas mais recentes.

C.3 - Programas Pascal Resultantes da Pré-Compilação

O processo de pré-compilação é ilustrado através da apresentação dos programas Pascal resultantes da tradução dos módulos componentes do exemplo apresentado.

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.

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 do serviço ini_par_mod. A definição de contexto (USE...) no módulo a, provoca a cópia dos objetos referenciados da unidade de definição TipMen, para o programa a. Nesse instante o tipo pré-declarado PRIORITY também é introduzido no programa Pascal. 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 reciprocidade não é verdadeira, i.e., variáveis normais não podem ser usadas no lugar de variáveis correspondentes a mensagens nos comandos de comuni-

cações e sincronização. Todos os procedimentos e funções correspondentes a serviços do núcleo são declarados como externos. Na chamada aos serviços de comunicação do núcleo são empregadas as funções `SIZEOF` e `ADS`, as quais retornam o tamanho e o endereço, respectivamente, das mensagens a enviar ou receber.

Conforme pode ser observado na listagem do programa C a construção da LPM que apresenta a tradução mais complexa é o comando de recepção seletiva. Ele requer inclusive a criação das variáveis auxiliares `var_for_ret` e `número_porta_selecionada`, as quais guardam, respectivamente, o valor da variável de controle do comando de repetição correspondente ao instante da seleção e o número da porta que recebeu a mensagem.

```

{# MODULE a(pausa:INTEGER); #}
PROGRAM a;
  {# Label gerado pelo comando LOOP #}
  LABEL 1;

  {# inclusao do tipo pre-definido PRIORITY #}
  TYPE
    PRIORITY=(SYSTEMPR,HIGHPR,NORMALPR,LOWPR,LOWESTPR);

  {# USE tipmen.INC; #}
  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
    {# Variavel correspondente ao parametro do modulo #}
    pausa:integer;

    {# MESSAGE ... #}
    men_ident : ident;
    men_pausa_prio : pausa_prio;

    {# Variaveis pre-declaradas #}
    nulo : integer;
    numero_porta_selecionada : integer;
    var_for_ret : integer;

    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 relógio:INTEGER4; EXTERN;
  PROCEDURE retarda(period:INTEGER4); EXTERN;
  PROCEDURE ini_par_mod(end_par:ADSMEM); EXTERN;
  PROCEDURE muda_prio(prio:PRIORITY); EXTERN;
  FUNCTION num_mod:INTEGER; EXTERN;

  {#PAGE+}
  BEGIN
    {# inicializacao area parametros do modulo #}
    ini_par_mod(ADS(pausa));

```

```

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

men_ident.num:=0;

(# LOOP ... #)
WHILE true DO
BEGIN
  FOR i:=1 TO 4 DO
  BEGIN
    (# men_ident.instante := TIME; #)
    men_ident.instante:= relógio ;

    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:= relógio ;

  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);
  (# END_LOOP; #)
END;

```

ND.

```

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

  (## Label gerado pelo comando LOOP ##)
  LABEL i;

  (## Inclusao do tipo pre-definido PRIORITY ##)
  TYPE
    PRIORITY=(SYSTEMPR,HIGHPR,NORMALPR,LOWPR,LOWESTPR);

  (## USE tipmen.INC; ##)
  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 : ident;
    men_identB : ident;
    men_pausa_prio : pausa_prio;

    (## Variaveis criadas pelo pre_compilador ##)
    nulo : integer;
    numero_porta_selecionada : integer;
    var_for_ret : integer;

    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;
  PROCEDURE rec(end_men:ADSMEM; prt_entrada:INTEGER); EXTERN;
  PROCEDURE desvia(prt_saida,prt_entrada:INTEGER); EXTERN;
  FUNCTION relógio:INTEGER4; EXTERN;
  PROCEDURE muda_prio(prio:PRIORITY); EXTERN;
  FUNCTION num_mod:INTEGER; EXTERN;

  BEGIN

```

```

men_identB.modulo:= num_mod;
men_identB.num:=0;
WHILE true DO
BEGIN
  FOR i:=1 TO 2 DO
  BEGIN
    {# RECEIVE men_identA FROM pe1; #}
    rec(ADS(men_identA),1);

    {# men_identB.instante:= TIME; #}
    men_identB.instante:= relógio ;

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

    {# SEND men_identB TO ps2; #}
    env_a(3,ORD(sizeof(men_identB)),ADS(men_identB));
  END;

  {# men_identB.instante:= TIME; #}
  men_identB.instante:= relógio ;

  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);

{# END_LOOP #}
END;

```



```

{# MODULE c(espera_sel:INTEGER); #}
PROGRAM c;

  {# Inclusao do tipo pre-definido PRIORITY #}
  TYPE
    PRIORITY=(SYSTEMPR,HIGHPR,NORMALPR,LOWPR,LOWESTPR);

  {# USE tipmen.INC; #}
  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

    {# Variavel correspondente ao parametro do modulo #}
    espera_sel:integer;

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

    {# Variaveis criadas pelo pre-compilador #}
    nulo : integer;
    numero_porta_selecionada : integer;
    var_for_ret : integer;

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

  {# Interface dos servicos oferecidos pelo STR #}
  PROCEDURE ini_rec_sel(indice_for,clausula,prt_entrada:INTEGER;
    end_men:ADSMEM); EXTERN;
  PROCEDURE ini_temp_sel(indice_for,clausula:INTEGER; tempo:INTEGER4); EXTERN;
  FUNCTION sel(tipo:CHAR;clausula_else:INTEGER;
    VARS indice_for,prt_entrada:INTEGER):INTEGER; EXTERN;
  PROCEDURE resp(prt_entrada,tam_men:INTEGER; end_men:ADSMEM); EXTERN;
  FUNCTION relógio:INTEGER4; EXTERN;
  PROCEDURE ini_par_mod(end_par:ADSMEM); EXTERN;
  PROCEDURE muda_prio(prio:PRIORITY); EXTERN;
  FUNCTION num_mod:INTEGER; EXTERN;

```

```

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_mem:3,' pediu a pausa e a prioridade? ');
  READLN(pausa,pri);
END;

3IN
  {# Variavel correspondente ao parametro do modulo #}
  ini_par_mod(ADS(espera_sel));
  {# num_modulo:= MODULE_ID; #}
  num_modulo:= num_mod;

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

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

        {# OR_SELECT #}
        {# RECEIVE men_ident FROM pe2 #}
        ini_rec_sel(nulo,1,2,ADS(men_ident));

        {# OR_SELECT #}

        {# TIMEOUT espera_sel #}
        ini_temp_sel(nulo,2,espera_sel);

        CASE sel('P',0,var_for_ret,numero_porta_selecionada) OF
          0: 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(numero_porta_selecionada,
                  ORD(sizeof(men_pausa_prio)),
                  ADS(men_pausa_prio));
            END;
          1: BEGIN
              {# inst := TIME; #}
              inst:= relógio ;

              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;
          2: BEGIN

```

```
        (# inst := TIME #)
        inst:= relógio ;

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

(# inst := TIME; #)
inst:= relógio ;

LePausaPrio(num_modulo,inst,0,espera_sel,prioridade);

(# SETPRIORITY(prioridade); #)
muda_prio(prioridade);
UNTIL FALSE;
END.
```

```

{# MODULE d; #}
PROGRAM d;

{# Inclusao do tipo pre_definido PRIORITY #}
TYPE
  PRIORITY=(SYSTEMPR,HIGHPR,NORMALPR,LOWPR,LOWESTPR);

  {# USE tipmen.INC #}
  Ident=RECORD
    modulo:INTEGER;
    instante:INTEGER4;
    num:INTEGER;
  END;

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

{# ENTRYPORT
  pe1:ident; #}

{# EXITPORT
  ps1:ident;#}

VAR

  {# MESSAGE #}
  men_ident : ident;

  {# Variaveis pre-declaradas #}
  nulo : integer;
  numero_porta_selecionada : integer;
  var_for_ret : integer;

{# 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;

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.

```

```

( # MODULE timeman(pulsos: INTEGER); # )
PROGRAM timeman;

  CONST

    vrtr=16#1C;

  VAR

    ( # Variavel correspondente ao parametro do modulo # )
    pulsos:integer;

    ( # Variaveis pre-declaradas # )
    nulo : integer;
    numero_porta_selecionada : integer;
    var_for_ret : integer;

    i,int_log_rtr:INTEGER;

( # Interface dos servicos oferecidos pelo STR # )
FUNCTION map_int(end_int:BYTE):INTEGER; EXTERN;
PROCEDURE espera_int(int_log:INTEGER); EXTERN;

BEGIN

  ( # Inicializacao area parametros do modulo # )
  ini_par_mod(ADS(pulsos));

  int_log_rtr:= map_int(vrtr);

  REPEAT
    FOR i:=1 TO pulsos DO

      ( # WAITIO(int_log_rtr); # )
      espera_int(int_log_rtr);

      TICK;
    UNTIL FALSE;
  END.

```

BIBLIOGRAFIA CONSULTADA

ADAN COELLO, J.M. Suporte de Tempo-Real para um Ambiente de Programação Concorrente. Dissertação de Mestrado em Engenharia Elétrica. Campinas, UNICAMP, 1986.

AHO, A.V. & ULLMAN, J.D. Principles of Compiler Design. 3.ed. Reading, MA, Addison-Wesley, 1979.

ANDERSON, T. & LEE, P.A. Fault Tolerancy Terminology Proposals. In: FTCS 12th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society Press, 1982.

ANDREWS, G.R. The Distributed Programming Language SR - Mechanisms, Design and Implementation. Software - Practice & Experience, 12(8): 719-53, 1982.

_____. & SCHNEIDER, F.B. Concepts and Notations for Concurrent Programming. Computing Surveys, 15(1): 3-43, Mar. 1983.

BARIGAZZI, G. et alii. Reconfiguration Procedure in a Distributed Multiprocessor System. In: FTCS 12th Annual International Symposium on Fault-Tolerant Computing. IEEE Computer Society Press, 1982.

BRAUN, C.L. Ada: Programming in the 80's. Computer, 14(6): 11-12, June 1981.

- BRENDER, R.F. & NASSI, J.R. What is Ada? Computer, 14(6): 17-24, June 1983.
- CARLSON, W.E. Ada: A Promising Beginning. Computer, 14(6): 13-15, June 1981.
- DIJKSTRA, E.W. Cooperation Sequential Process. In Programming Languages. Ed. F. Genuys Academic Press, 1968.
- DIJKSTRA, E.W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Communications of the ACM, 18(8): 433-57, Aug. 1975.
- FELDMAN, J.A. High Level Programming for Distributed Computing. Communications of the ACM, 22(6): 353-68, June 1979.
- GRIES, D. Compiler Construction for Digital Computers. New York, NY, John Wiley, 1971.
- HANSEN, P.B. The Architecture of Concurrent Programs. Englewood Cliffs, Prentice-Hall, 1977.
- _____. Distributed Process: A Concurrent Programming Concept. Communications of the ACM, 21(11): 934-41, Nov. 1978.
- _____. Edison - A Multiprocessor Language. Software: Practice and Experience, 11(4): 325-61, Oct. 1981.
- HARLAND D.M. Concurrency in a Language Employing Messages. Information Processing Letters. 12(2): 59-62, Apr. 1981.
- HOARE, C.A.R. Monitors: An Operation System Structuring Concept. Communications of the ACM, 21(8): 666-77, Aug. 1978.

JOHNSON, S.C. YACC: Yet Another Compiler-Compiler. Bell Laboratories, July 1978.

_____. Language Development Tools on the Unix System. Computer, 16-21, Aug. 1980.

IBM-PC Pascal Compiler Language Reference Version 2.00. Personal Computer Language Series, IBM Corp., 1984.

IBM-PC Pascal Compiler Fundamentals Version 2.00. Personal Computer Languages Series, IBM Corp. 1984.

ISO 7185. Specification for Computer Programming Language Pascal. International Standards Organization, 1983.

KERNIGHAN, B.W. & RITCHIE. The C Programming Language. Prentice-Hall, Englewood Cliffs, 1978.

KLEINROCK, L. Distributed Systems. Communications of the ACM, 28(11): 1200-213, Nov, 1985.

KRAMER, J. et. alii. CONIC: An Integrated Approach to Distributed Computer Control Systems. IEEE Proceedings, 130(1): 1-10, Jan. 1983.

_____. et alii. The CONIC Programming Language Version 2.4. Research Report. DOC 84/19, Department of Computing, Imperial College, London,. Oct. 1984.

_____. et alii. The CONIC Configuration Language Version 2.3. Research Report DOC 84/20, Department of Computing, Imperial College, London, Nov. 1984.

- _____. & MAGEE, J. Dynamic Configuration for Distributed Systems. IEEE Transactions on Software Engineering, SE-11(4): 424-36, Apr. 1985.
- LESK, M.E. & SCHMIDT, E. LEX - A Lexical Analyzer Generator. Bell Laboratories, Aug. 1982.
- LOPES, A.B. & COELLO, J.M.A. 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, 1985, p. 467-75.
- MAGEE, J.N. Provision of Flexibility in Distributed Systems. Ph.D. Thesis, Department of Computing - Imperial College of Science & Technology, London, Apr. 1984.
- MAGALHÃES, M.F. Software para Tempo-Real. Editora da Universidade Estadual de Campinas, 1986.
- PARNAS, D.L. On the Criteria to be Used in Decomposing Systems into Modules. Communications of the ACM 15(12), Dec. 1972.
- PRESSMAN, R.S. Software Engineering. MacGraw-Hill, Inc. 1982.
- PRINCE, S.N. & SLOMAN, M.S. Communication Requirements of a Distributed Computer Control System. IEEE Proceedings, 128(1): 21-34, Jan. 1981.
- SANTOS, S.M. Programação Concorrente: Mecanismos de Sincronização e Comunicação de Processos. Quarta Escola de Computação. São Paulo, 1984.

SCHREINER, A.T. & FRIEDMAN, H.G. Jr. Introduction to Compiler Construction with Unix. Englewood Cliffs, Prentice-Hall, 1985.

STEUSLOFF, H.U. Structures of Automatic Control Systems and the Consequences for Programming Languages. Process Automation, 1979. p. 3-11.

_____. Advanced Real-Time Languages for Distributed Industrial Process Control. IEEE Computer, Feb. 1984, p. 37-46.

WERUM, W. & WINDAUER, H. Introduction to PEARL: Process and Experiment Automation Realtime Language. 3 ed. - Braunschweig; Wiesbaden: Vieweg, 1985.

WIRTH, N. Modula: A Language for Modular Multiprogramming. Software: Practice and experience, 7(1): 3-35, Jan./Feb. 1977.

_____. Modula 2. Report 36, Institut für Informatik, ETH, Zurich, 1980.

_____. History and Goals of Modula-2. BYTE, Aug. 1984, p. 145-152.

YOUNG, S.J. Real Time Languages: Design and Development. Ellis Horwood Ltd., 1982.