UNIVERSIDADE FEDERAL DE PARAíBA
CENTRO DE CIêNCIA E TECNOLOGIA
CURSO DE MESTRADO EM INFORMáTICA

DESENVOLVIMENTO DE UMA ESTAÇÃO DE REDE MAP PARA A AUTOMAÇÃO INDUSTRIAL

DAVID JOHN TURNELL

UFPb - CAMPINA GRANDE
SETEMBRO - 1990

DAVID JOHN TURNELL

DESENVOLVIMENTO DE UMA ESTAÇÃO DE REDE MAP PARA A AUTOMAÇÃO INDUSTRIAL

Dissertação apresentada ao Curso de Mestrado em Informática da Universidade Federal de Paraíba, em cumprimento às exigências para obtenção do Grau de Mestre.

AREA DE CONCENTRAÇÃO: CIêNCIA DA COMPUTAÇÃO

Prof. Dr. JOBERTO S.B.MARTINS

Orientador

UFPb - CAMPINA GRANDE

SETEMBRO - 1990

T944d Turnell, David John.
Desenvolvimento de uma estação de rede MAP para a automação industrial / David John Turnell. - Campina Grande, 1990.
137 f.

Dissertação (Mestrado em Informática) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, 1990.
"Orientação: Prof. Dr. Joberto Sérgio Barbosa Martins".
Referências.

1. Arquitetura de Redes. 2. Estação de Rede MAP. 3. Automação Industrial. 4. Informática - Dissertação. I. Martins, Joberto Sérgio Barbosa. II. Universidade Federal da Paraíba - Campina Grande (PB). III. Título

CDU 004.72(043)

DESENVOLVIMENTO DE UMA ESTAÇAO DE REDE MAP PARA A AUTOMAÇAO INDUSTRIAL

DAVID JOHN TURNELL

DISSERTAÇAO APROVADA EM 18.09.1990

ROBERTO SERGIO B. MARTINS, Dr.
Orientador

JOSÉ ANTAO BELTRAO MOURA, Ph.D
Componente da Banca

MANUEL DE JESUS MENDES, Dr.
Componente da Banca

JOSE HENRIQUE ZILBERBERG, M.Sc
Componente da Banca

Campina Grande, 18 de setembro de 1990

Meus agradecimentos ao meu orientador Joberto por seu suporte e profissionalismo; a minha esposa Fátima por sua paciência; as minhas crianças sem as quais nada teria sentido; e aos outros bons Brasileiros que encontrei - que eles possam levar os demais para um futuro melhor.

SUMáRIO

RESUMO

Este trabalho descreve o projeto e a implementação de um processador frontal de comunicação (PFC) para as redes tipo MAP. Na primeira parte do trabalho é apresentado o desenvolvimento do hardware na forma de uma placa para o super-microcomputador de propósito geral PP (Processador Preferencial) desenvolvido pela Telebrás. Esta placa dispõe de um microprocessador de 16 bits, um circuito integrado controlador de rede, e até 512 kbytes de memória. A segunda parte do trabalho descreve o núcleo multi-tarefa desenvolvido, chamado Executivo, que é projetado especificamente para este hardware e otimizado para operação em tempo-real e para o suporte da implantação de protocolos segundo o padrão RM OSI/ISO. O Executivo oferece serviços de gerenciamento de tarefa, de comunicação inter-tarefa, de temporização e de alocação dinâmica de memória. A terceira parte do trabalho descreve a implementação das camadas LLC ("Logic Link Control") e MAC ("Media Access Control") visando a implantação de protocolos segundo a arquitetura mini-MAP. Estas duas camadas são implementadas como uma única tarefa que é executada sob o Executivo.

SUMMARY

This work describes the design and implementation of a communications front-end processor (PFC) for MAP type networks. In the first part of this work is a presentation of the development of the hardware in the form of a board for the PP (Processador Preferêncial) general purpose super-microcomputer developed by Telebrás. This board contains a 16 bit microprocessor, a network controller IC, and up to 512 kbytes of memory. The second part of this work describes the developed multi-tasking nucleus, called the Executive, which is designed specifically for this hardware and optimized for real-time operation and the support of the implantation of the protocols which follow the RM OSI/ISO standard. The Executive offers task management services, inter-task communication services, timing services and dynamic memory allocation services. The third part of this work describes the implementation of the layers LLC (Logic Link Control) and MAC (Media Access Control) for the installation of protocols following a mini-MAP architecture. These two layers are implemented in the form of a single task which executes under the Executive.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

BD	Buffer Descriptor
CI	Circuito Integrado
CLNS	Connectionless Mode Network Service
DA	Destination Address
DMA	Direct Memory Access
DSAP	Destination Service Access Point
EPA	Enhanced Performance Architecture
EPROM	Erasable Programable Read Only Memory
FD	Frame Descriptor
IEEE	Institute of Electrical and Electronic Engineers
ISO	International Standards Organization
LAN	Local Area Network
LLC	Logic Link Control
LSAP	Link Service Access Point
LSB	Least Significant Bit
LSDU	Link Service Data Unit
MAC	Media Access Control

MAP	Manufacturing Automation Protocols
MMS	Manufacturing Message Standard
MSDU	Media access control Service Data Unit
OSI	Open Systems Interconnection
PAL	Programmable Array Logic
PDU	Protocol Data Unit
PFC	Processador Frontal de Comunicação
PP	Processador Preferêncial
RAM	Random Access Memory
ROM	Read Only Memory
RWR	Request With Response
SA	Source Address
SAP	Service Access Point
SSAP	Source Service Access Point
TBC	Token Bus Controller
UCP	Unidade Central de Processamento
VLSI	Very Large Scale Integration

1. INTRODUÇAO

Este capítulo apresenta uma visão geral da rede industrial MAP, com ênfase na arquitetura MAP/EPA, que é o tema central deste trabalho. Para concluir este capítulo há uma descrição dos objetivos deste trabalho e uma sinopse do conteúdo de cada capítulo restante.

1.1 A Automação Industrial

O computador se tornou uma ferramenta indispensável nas principais áreas de atuação do ser humano. Uma destas áreas é da produção industrial, onde a integração dos computadores no processo industrial é conhecida como a Automação Industrial. O aumento na produtividade que é o resultado da automação industrial é fundamental para a viabilidade das indústrias e o crescimento dos seus respectivos países.

Desde a década de sessenta o computador tem sido uma parte vital nos grandes centros de produção industrial. Ele é usado para uma variedade de funções - desde o controle dedicado de máquinas usando microcomputadores até a análise, otimização, e planejamento de processos inteiros usando minicomputadores e computadores de grande porte. Dentro do ambiente industrial estes computadores podem ser classificados numa hierarquia [ARA 86][MEN 88].

O nível inferior desta hierarquia é composto de computadores de controle. Estes computadores são ligados aos sensores e atuadores, executando um controle local limitado, coletando e filtrando dados para ser enviados aos níveis superiores da hierarquia.

O nível intermediário da hierarquia é dos computadores de operação. O operador, através de um vídeo gráfico colorido e um teclado, tem acesso ao estado do processo. Ele pode alterar pontos de verificação das variáveis do processo, parâmetros dos algorítmos de controle e aquisição de dados, além de outras funções.

O nível superior da hierarquia é dos computadores de supervisão, que são responsáveis por funções mais complexas como a otimização do desempenho global e o escalonamento de tarefas.

De alta importância para esta hierarquia é o meio da comunicação entre seus componentes. Com a proximidade física dos computadores e o elevado fluxo de dados a ser passado entre eles o meio de comunicação mais apropriado é a rede local. Uma maneira convencional de assegurar uma compatibilidade na interligação destes computadores é a de se usar o equipamento de um único fabricante numa arquitetura proprietária. Contudo, com um número e diversificação cada vez maior dos computadores utilizados no ambiente industrial surge a necessidade para um padrão projetado especificamente para definir esta interligação e que possa portanto acomodar a heterogeneidade existente.

A General Motors, que estava preocupada com o sucesso na automação industrial dos fabricantes Japoneses de automóveis, iniciou seu próprio programa de automação. Ela criou como base a especificação aberta de uma rede local projetada especificamente para formar a espinha dorsal em centros de produção de grande e médio porte. Esta especificação original e as suas versões posteriores são chamadas de MAP ("Manufacturing Automation Protocol") [GM 87][CRO 85]. A General Motors e as outras empresas e instituições envolvidas no projeto MAP estão tentando levar

sua arquitetura ao ponto de se tornar um padrão internacional.

1.2 A Rede MAP

A rede MAP é baseada no modelo OSI/ISO ("Open Systems Interconnection/International Standards Organization") [ISO 85]. Neste modelo a interface de rede é composta de sete camadas hierárquicas conforme a figura 1.1.

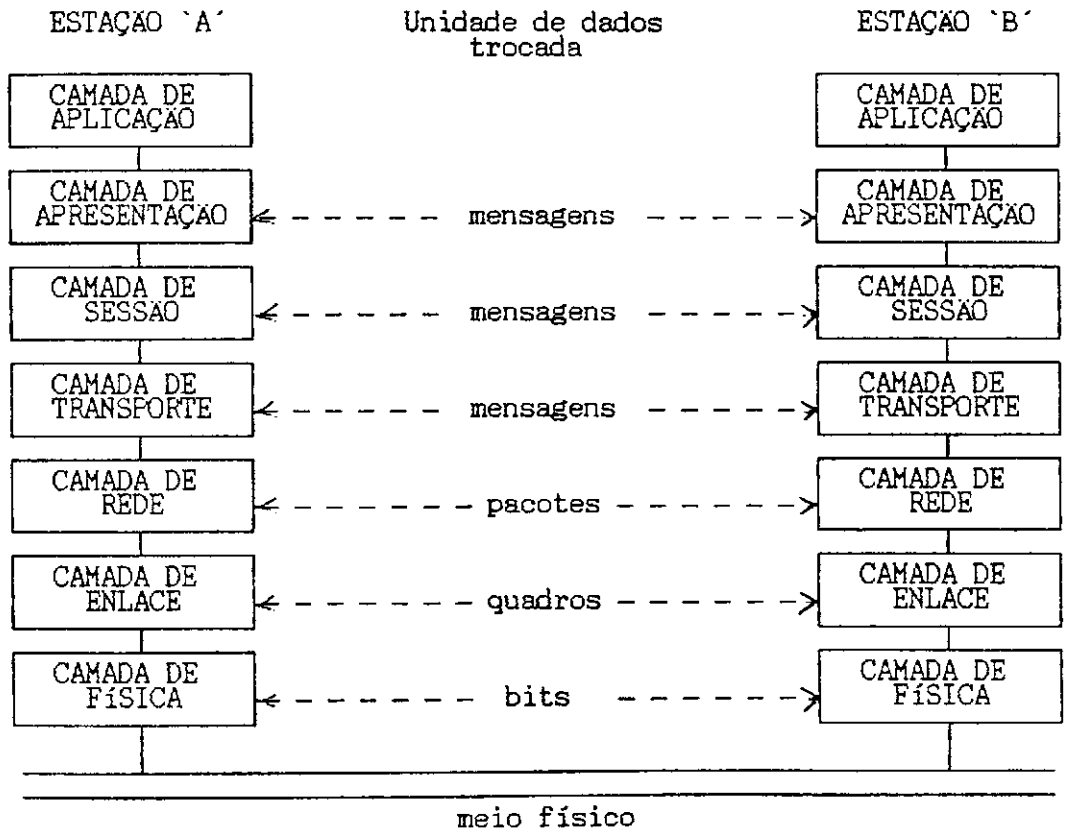


Figura 1.1 - O Modelo OSI/ISO

Cada camada fornece serviços para sua camada superior e faz o uso dos serviços da sua camada inferior. O modelo OSI/ISO define para cada

camada as primitivas que oferece em serviços para a sua camada superior e o formato dos quadros e mensagens que são trocados com a camada par em outras estações [ARA 86].

No lugar de tentar forçar a aceitação de mais um novo 'padrão', a General Motors adotou uma tecnologia já existente para as camadas física e enlace do MAP. Ela adotou o barramento com ficha ("Token Bus") [IEEE 88A]. Esta tecnologia é apropriada para o ambiente industrial devido ao uso de um protocolo determinístico, ou seja, os tempos máximos de entrega das mensagens de alta prioridade são calculáveis 'a priori'. Esta característica é importante para as aplicações do tipo tempo-real de controle e supervisão que impõem limitações no tempo de resposta da rede.

Para as camadas superiores a arquitetura MAP adota um sub-conjunto dos protocolos padrão do modelo OSI/ISO. A figura 1.2 mostra a arquitetura MAP.

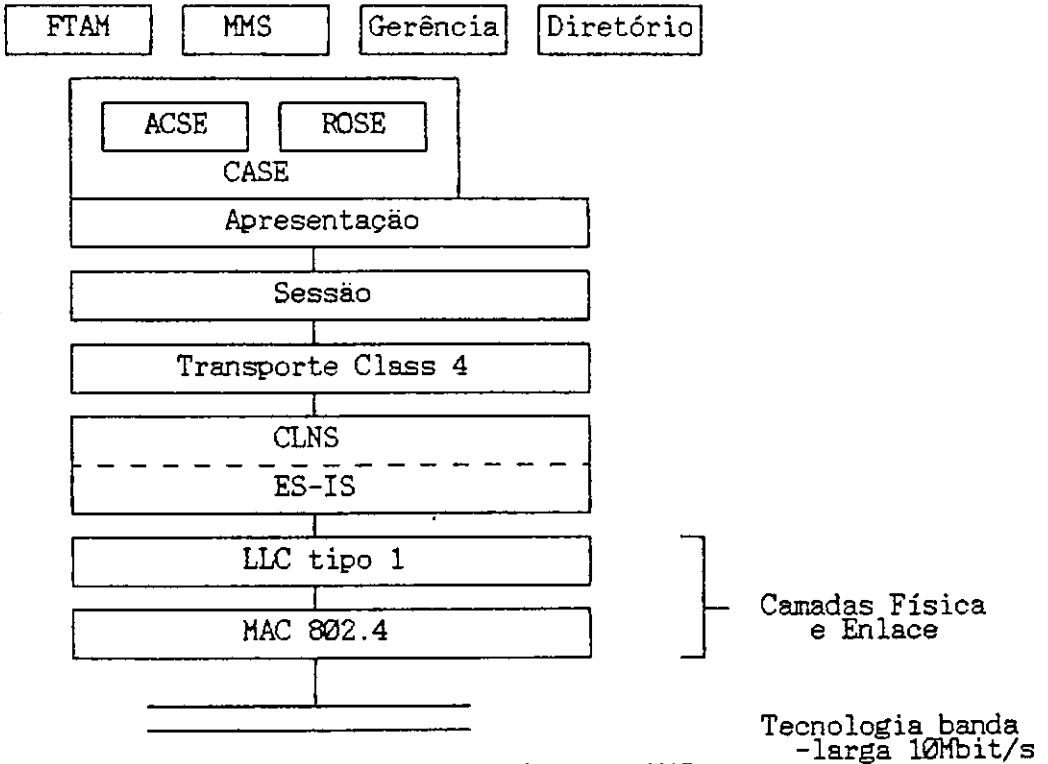
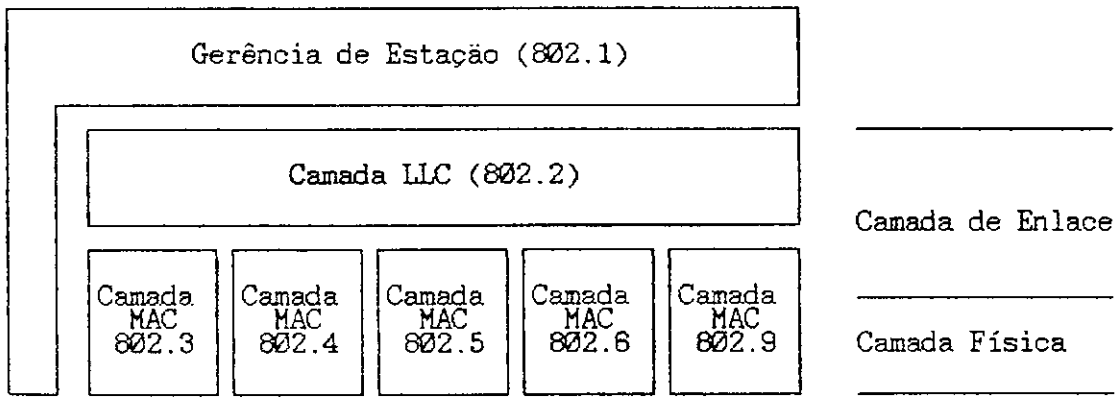


Figura 1.2 - A Arquitetura MAP

Na tecnologia de acesso ao meio usando barramento com ficha uma estação pode transmitir na rede somente quando tiver a ficha ("token"). Há somente uma ficha circulando na rede. A ficha é periodicamente passada de estação para estação, na ordem inversa de endereço lógico. A estação com endereço menor passa a ficha de volta para a estação com endereço maior. Esta circulação da ficha forma um anel lógico. Cada estação pode segurar a ficha por um tempo máximo predeterminado.

O barramento com ficha é definido pelo padrão 802.4 do IEEE [IEEE 88A], que é parte de um conjunto de padrões comumente chamado de arquitetura IEEE 802. Nesta arquitetura, mostrada na figura 1.3, as camadas física e enlace do modelo OSI/ISO são mapeadas para as camadas MAC ("Medium Access Control") e LLC ("Logic Link Control").



IEEE 802.3 - CSMA/CD da Ethernet
IEEE 802.4 - Barramento com Ficha
IEEE 802.5 - Anel com Ficha
IEEE 802.6 - "Metropolitan Area Network" (MAN)
IEEE 802.9 - Método de Acesso a LAN de Dados e Voz Integrados

Figura 1.3 - A Arquitetura IEEE

Na arquitetura IEEE 802 há uma especificação de camada MAC para cada tecnologia de rede - por exemplo, a MAC IEEE 802.4 é a do barramento com ficha e a MAC IEEE 802.3 é a tecnologia CSMA/CD das redes Ethernet. A especificação da camada LLC é comum a todas as especificações de MAC. Na figura 1.3 cinco MACs são citadas mas a arquitetura está sendo continuamente aumentada com a incorporação de novas tecnologias. Note-se que a camada MAC representa a camada física do modelo ISO/OSI junto com uma parte da camada de enlace.

A arquitetura MAP é apropriada para os níveis superiores da hierarquia de sistemas de controle onde dominam os computadores de grande porte e os minicomputadores. Pela natureza da sua estrutura e a tecnologia adotada (banda-larga), os protocolos MAP apresentam três dificuldades em relação ao seu uso nos níveis inferiores desta hierarquia. Os computadores normalmente usados nestes níveis inferiores são microcomputadores de 8 e 16 bits[1] que tem uma capacidade limitada de processamento e que operam sob as mais rigorosas condições de tempo-real. Estas três dificuldades são:

- A interface MAP com suas sete camadas representa uma carga considerável para um microprocessador de 16 bits em termos de processamento e consumo de memória;
- Os tempos de processamento de mensagens são maiores do que os aceitáveis para condições tempo-real e
- O uso de uma tecnologia banda-larga no nível físico necessita de modems de alto custo e considerável tamanho físico.

Para enfrentar estes problemas, a versão 3.0 da especificação MAP introduziu a arquitetura MAP/EPA ("Enhanced Performance Architecture"), mostrada na figura 1.4. Os aplicativos críticos no tempo utilizam o lado

---

1. Por razões de confiabilidade, o uso de microprocessadores de 32 bits no ambiente industrial ainda está muito limitado.

EPA desta arquitetura. Associada à arquitetura MAP/EPA existe a arquitetura denominada mini-MAP, composta somente do lado EPA da arquitetura MAP/EPA.

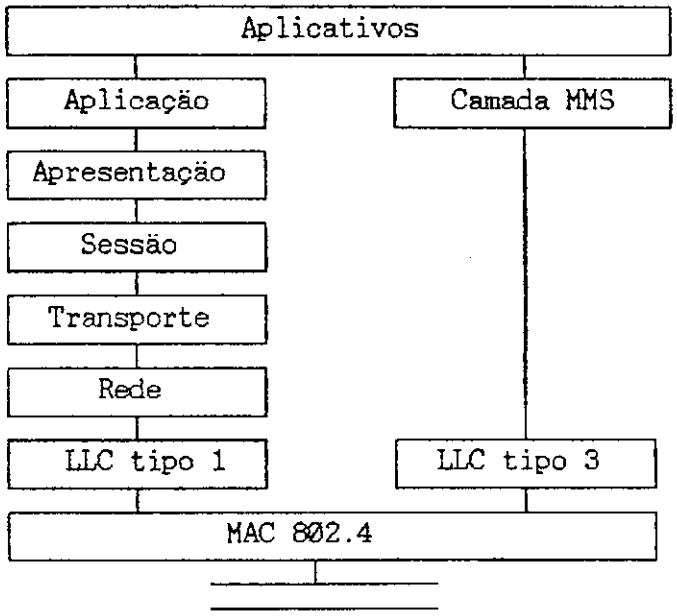


Figura 1.4 - A Arquitetura MAP/EPA

A arquitetura EPA de três camadas foi inspirada em parte pelos protocolos Proway desenvolvidos pela ISA ("Instrument Society of America"). Estes protocolos foram desenvolvidos independentemente dos protocolos MAP especificamente para servir as necessidades dos níveis inferiores das hierarquias de controle.

Os segmentos EPA são sub-redes de até 32 nós compostos de estações do tipo MAP/EPA e mini-MAP. No lugar da tecnologia banda-larga do MAP, os segmentos EPA utilizam a tecnologia banda-portadora para seus níveis físicos. Os modems desta tecnologia são disponíveis em forma de CIs por um custo muito inferior aquele da tecnologia banda-larga. Estas características resultam em interfaces mais compactas e econômicas em

relação aquelas do MAP. Os segmentos EPA e a simplicidade da arquitetura mini-MAP viabilizam a integração de equipamentos de pequeno porte do tipo sensor, atuador e controlador dedicado.

As arquiteturas MAP/EPA e mini-MAP usam a camada LLC tipo 3 que na sua operação faz reconhecimento sem conexão. Cada quadro enviado resulta no recebimento de um reconhecimento, este reconhecimento podendo incluir dados. Esta característica resulta numa camada LLC mais complicada em relação aquela do MAP (a LLC tipo 1) que é do tipo sem-conexão-nem-reconhecimento. A camada MMS ("Manufacturing Message Standard") [PAG 88] é usada ao nível da camada de aplicação. A figura 1.5 mostra a interligação de um segmento EPA com uma rede MAP 'espinha dorsal'.

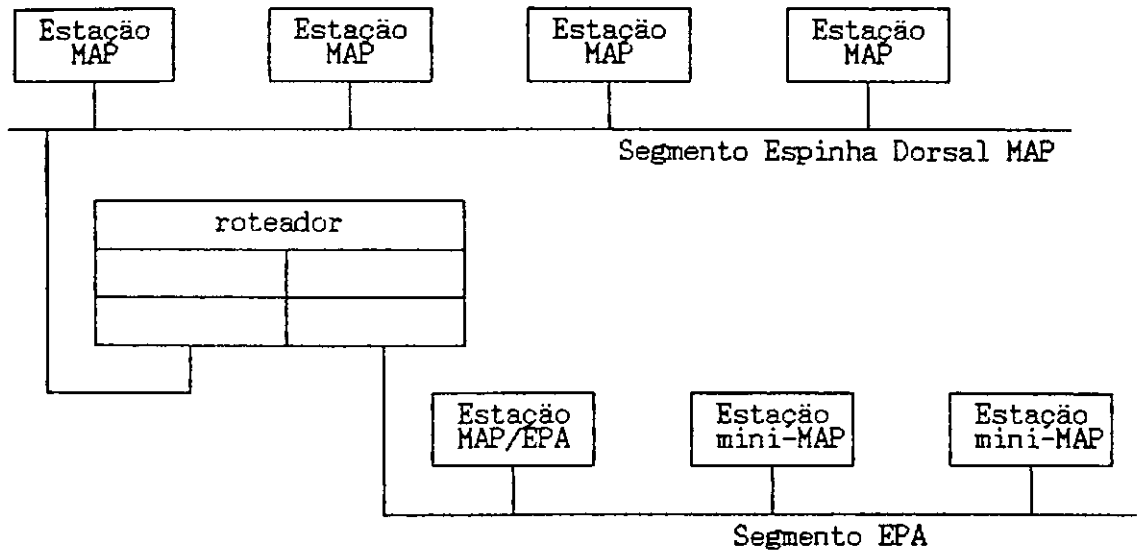


Figura 1.5 - Um Segmento MAP/EPA

Na interligação dos dois tipos de segmento tem que haver pelo menos um nó MAP/EPA que efetua a conversão entre as arquiteturas de 3 e 7 camadas. O roteador faz a interligação das duas redes, tendo hardware

banda-portador e banda-larga. Um roteador é usado em preferência a um 'bridge' porque ele faz segmentação de quadro.

1.3 O Objetivo deste Trabalho

O objetivo deste trabalho é o projeto e a implementação de uma interface nomeada PFC (Processador Frontal de Comunicação) para as redes locais industriais do tipo mini-MAP. Esta interface é 'inteligente', tendo um microprocessador de 16 bits e memória própria. Esta interface é capaz de executar a maior parte do software da arquitetura de rede nela instalada.

Na primeira parte do trabalho é descrito o desenvolvimento do hardware da interface na forma de uma placa de expansão para um super-microcomputador de propósito geral - o PP (Processador Preferêncial) desenvolvido pela Telebrás [TEL 87]. Este computador é típico daqueles sendo atualmente integrados no ambiente industrial. Esta placa de expansão dispõe de um microprocessador 80186, um controlador de rede MC68824 e até 512 kbytes de memória.

Na segunda parte do trabalho é descrito um núcleo multi-tarefa chamado de Executivo [TUR 89] que fornece serviços de gerenciamento de tarefa, de comunicação inter-tarefa, de temporização e de alocação dinâmica de memória. O Executivo é projetado especificamente para as necessidades de tempo-real das interfaces de rede.

Na terceira parte do trabalho é descrita a implementação das camadas LLC tipo 3 e MAC da arquitetura MAP/EPA. As duas camadas são implementadas como uma única tarefa que é executada sob o Executivo.

Estas três partes juntas formam uma plataforma básica para uma interface mini-MAP. Este trabalho inclui a implementação, integração e teste destas três partes. A seguir há um sinopse de cada capítulo:

- Capítulo 2: apresenta o hardware da interface PFC, começando com um resumo das diversas configurações de hardware usadas para interfaces de rede, seguida por uma avaliação da potencialidade destas configurações para este projeto de interface MAP, e terminando com uma descrição detalhada do hardware implementado.

- Capítulo 3: apresenta o Executivo, começando com uma identificação das suas características principais surgeridas pela interface mini-MAP, seguida por uma descrição dos seus módulos principais (os Gerentes) com detalhamento dos seus mecanismos internos e a definição das primitivas que são oferecidas às tarefas. Para concluir é feita uma abordagem do método de criação e instalação de tarefas que serão executadas sob o Executivo.

- Capítulo 4: apresenta a implementação da camada MAC, começando com uma descrição rápida das suas primitivas e de seu funcionamento, seguida de uma análise do método de implementação usado no PFC. Esta apresentação é concluída com o detalhamento da implementação em si.

- Capítulo 5: apresenta a implementação da camada LLC tipo 3, começando com uma descrição das suas características mais importantes. Em seguida são destacados os aspectos da LLC que determinam a forma desta implementação. Em seguida há um detalhamento da implementação em si e, para concluir, há um esclarecimento da forma como cada primitiva da especificação é representada na implementação.

- Capítulo 6: serve como conclusão para este trabalho, onde nossas opiniões e resultados sobre este trabalho são expressas.

2. O HARDWARE

Este capítulo apresenta a implementaçäo do hardware do PFC. Ele começa com uma apresentaçäo das várias configuraçöes de hardware usadas para interfaces de rede, destacando para cada uma delas as suas vantagens e desvantagens. Em seguida há uma justificativa para a escolha de uma destas configuraçöes para formar a base deste projeto. Para concluir é feito o detalhamento do hardware implementado baseado no agrupamento de quatro circuitos distintos.

O componente principal do hardware de qualquer interface é o controlador de rede, que é um circuito integrado VLSI ("Very Large Scale Integration") projetado especificamente para gerenciar as operaçöes mais básicas da interface de rede. Com efeito, para as redes locais operando com taxas de transferências altas (por exemplo, 10 Mbit/s ou mais) as operaçöes de gerenciamento têm restriçöes de tempo täo críticas que impedem o uso de um microprocessador de propósito geral para executá-las.

Normalmente cada controlador é projetado para uma tecnologia específica de rede (Ethernet, Barramento com Ficha, etc), podendo operar

com uma faixa de taxas de transmissão. Os controladores necessitam de amplo acesso à memória onde normalmente estão mantidas as estruturas de controle e os buffers de dados utilizados.

A disponibilidade destes controladores por um custo relativamente baixo tem viabilizado a implementação do hardware das interfaces de rede do tipo PFC para os barramentos de diversos computadores como neste caso.

2.1 As Configurações Possíveis do Hardware

Dado o interesse inicial em projetar uma interface segundo o padrão MAP, a parte inicial deste trabalho consistiu na escolha da configuração do hardware. Existem três configurações básicas usadas em interfaces de rede. Elas são chamadas neste trabalho de interface simples, interface com memória e interface inteligente. As três variam bastante na sua complexidade e eficiência. A seguir há uma descrição destas configurações, destacando para cada uma delas as suas vantagens e desvantagens.

2.1.1 A Interface Simples

A primeira configuração analisada é a interface simples. Nesta interface o controlador faz acesso à memória através do barramento do hospedeiro, conforme a figura 2.1.

Para que o controlador possa ter acesso à memória ele primeiro tem que iniciar o processo de "tomada de barramento" usando sinais de pedido e de reconhecimento (no caso da família 8086 os sinais HOLD e HOLDA). Normalmente o controlador faz a transferência através de um canal DMA. Enquanto o controlador segura o barramento, nenhum outro elemento do sistema pode ter acesso ao mesmo. No caso do controlador utilizado neste

projeto a proporçäo do tempo em que o controlador está fazendo acesso a memória pode atingir até 5Ø% [MOT 87B].

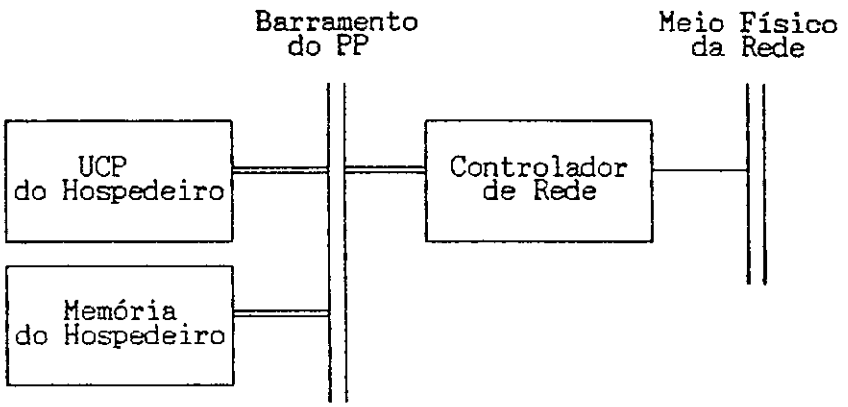


Figura 2.1 - A Interface Simples

Para aquele hospedeiro que faz acesso à memória local na placa UCP independentemente do barramento, não há uma perda considerável de desempenho do sistema, exceto para acesso ao disco. Para hospedeiros em que a UCP faz acesso à memória numa outra placa através do barramento este bloqueio pode prejudicar seriamente o desempenho do sistema. Há também a questäo do tempo de latência do barramento. Se o controlador não consegue `tomar´ o barramento em tempo, haverá quadros perdidos com a necessidade de retransmissöes resultando numa ineficiência para o sistema. Este fator é mais crítico em redes que operam com taxas elevadas de transmissäo (maior que 5 Mbit/s).

A vantagem deste tipo de interface é a sua simplicidade e o seu baixo custo - a interface sendo composto basicamente do CI controlador de rede, o modem e alguns buffers. Ela é mais usada nos equipamentos onde o custo é o fator mais importante. A desvantagem desta interface é a baixa eficiência do sistema devido ao acesso constante ao barramento e a queda resultante do desempenho da UCP principal.

Na medida que os controladores aumentem suas capacidades e o acesso à memória se torne mais crítica, este tipo de interface se tornará inviável.

2.1.2 A Interface com Memória

A segunda configuração de interface é aquela da figura 2.2, onde a interface dispõe de memória própria. O controlador faz acesso diretamente a esta memória sem bloquear o barramento do hospedeiro. Através de um circuito de arbitragem, a UCP do hospedeiro também pode fazer acesso a esta memória. Cada acesso pelo hospedeiro implica num atraso inicial em função da arbitragem.

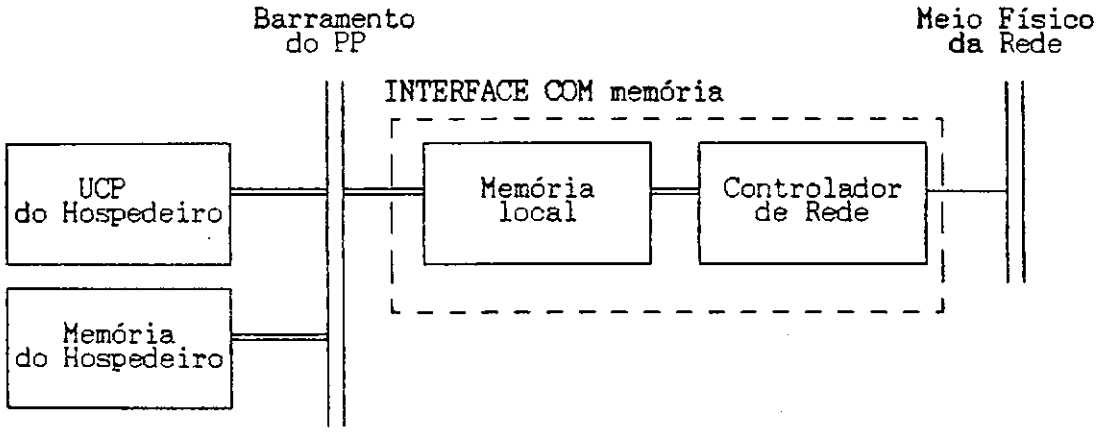


Figura 2.2 - A Interface com Memória Própria

Os controladores precisam de bastante memória para seus buffers e estruturas de controle. Por exemplo, quanto maior a quantidade de memória, maior o número e o tamanho dos buffers que podem ser alocados para a recepção, reduzindo assim o número de quadros perdidos por falta de buffers. Em nossas estimativas, 32 kbytes de memória seria um valor mínimo a ser utilizado. Este valor vêm dos cálculos seguintes:

- uma média de dezesseis buffers de 512 bytes (8 kbytes)

contendo quadros recebidos;

- uma média de dezesseis buffers de 512 bytes (8 kbytes) formando uma reserva para recepção;
- uma média de dezesseis buffers de 512 bytes (8 kbytes) contendo dados a ser transmitidos
- aproximadamente 8 kbytes para estruturas de controle (tabela de inicialização, descritores de quadro, etc).

Os hospedeiros alocam normalmente um espaço de memória reduzido para as suas interfaces e placas de expansão e, assim sendo, faz-se necessária a criação de um mecanismo de janelas ("windowing") para expandí-la.

Usando o PP como exemplo, é alocada 32 kbytes no mapa de memória do PP para cada placa de expansão. Se a interface dispuser de 128 kbytes de memória então o PP pode fazer acesso com base de quatro janelas de 32 kbytes. O acesso à memória pelo controlador é direito sem passar por este mecanismo de janelas. A figura 2.3 mostra este mecanismo.

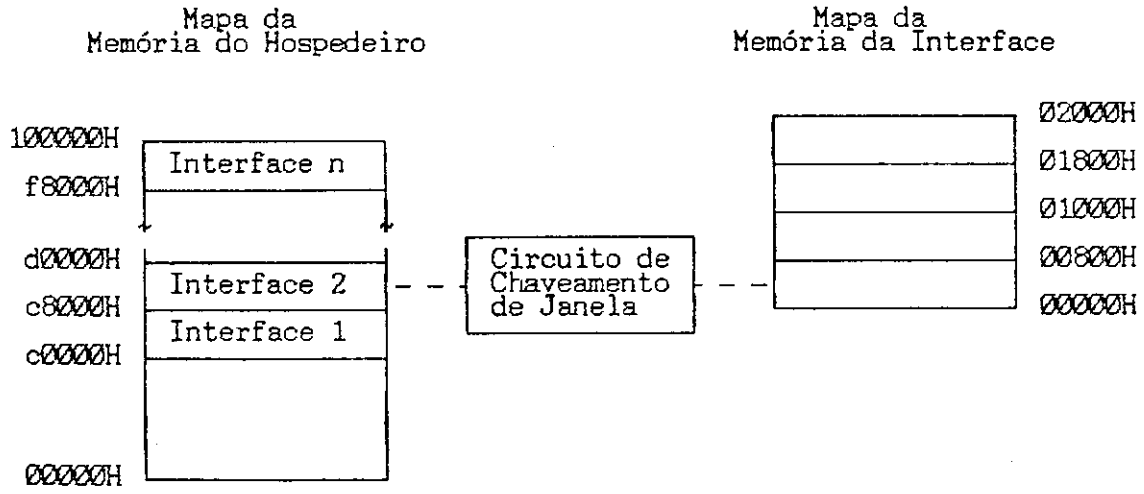


Figura 2.3 - Um Mecanismo de Janela para a Memória da Interface

Uma vantagem deste tipo de interface é que a sua eficiência é mais alta que a interface anterior, especialmente para hospedeiros que fazem acesso a memória em placas separadas daquela da UCP. Uma segunda vantagem é que o controlador não consome memória principal do hospedeiro.

Esta interface é um pouco mais complexa do que a interface anterior, tendo a mais a memória local e o circuito de arbitragem.

2.1.3 A Interface Inteligente

A terceira configuração de interface é aquela da figura 2.3. Esta interface contém um microprocessador local à interface e uma quantidade elevada de memória. Nesta interface uma parte do software da interface é deslocada para ser executada na própria interface, permitindo várias configurações de software.

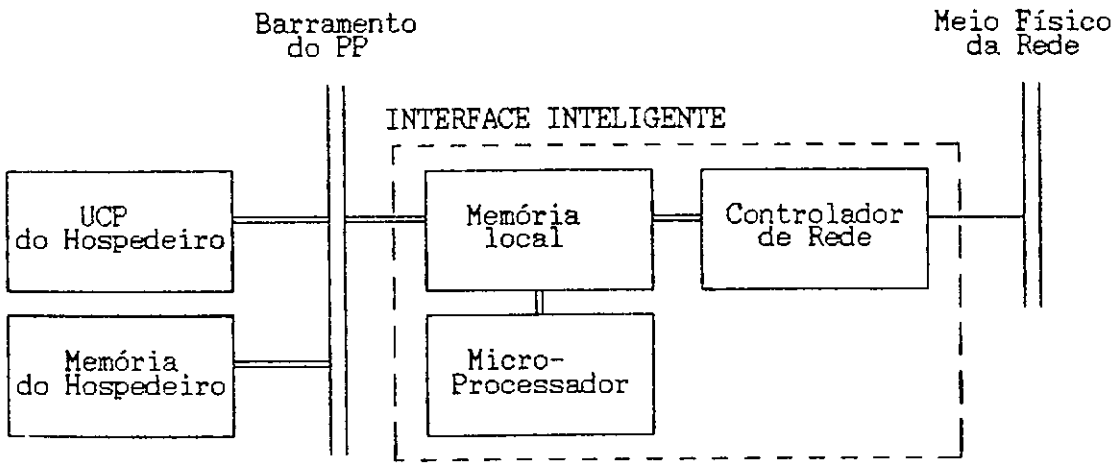


Figura 2.4 - A Interface Inteligente

A memória da interface é ligada diretamente ao microprocessador, permitindo o seu acesso sem passar por um processo de arbitragem.

Através de um circuito de arbitragem, o controlador e o hospedeiro também fazem acesso a memória, e durante o acesso o microprocessador fica num estado de "HOLD" com suas linhas de endereço, dados e controle em alta impedância. O acesso pelo controlador e pelo hospedeiro não é tão eficiente devido ao processo de arbitragem que resulta num atraso no começo de cada acesso. Na realidade os controladores de rede normalmente utilizam um canal DMA e, assim sendo, o atraso no acesso à memória acontece apenas no início de uma transferência de bloco.

Como no caso da interface anterior, a memória da interface pode dispor de um mecanismo de janelas. Neste caso a parte da memória tendo acesso pelo hospedeiro e pelo microprocessador é chamado de memória comum.

Com uma quantidade de memória adequada, a maioria do software da interface de rede pode se executar na própria interface, reduzindo assim a carga no hospedeiro. A comunicação entre o software residente na placa e no hospedeiro é feita através de buffers na memória comum.

A vantagem deste tipo de interface é a sua eficiência. Primeiro, em função do microprocessador ser dedicado, o tempo médio entre transmissão e recepção de mensagens é menor. Segundo, o hospedeiro permanece disponível para sua função principal que, no ambiente industrial, seria de monitoração e controle. Terceiro, a interface não consome uma grande quantidade de memória do hospedeiro. É importante notar que cada camada de software que é executada na própria interface libera dezenas de kilobytes no hospedeiro para o caso de protocolos de comunicação.

Uma desvantagem desta interface é sua complexidade. Ela tem um número significativamente maior de componentes em relação às outras duas interfaces. Uma outra desvantagem que se aplica durante a fase de implementação é que a depuração do software é mais difícil pois os depuradores disponíveis para MS-DOS não podem funcionar com código

instalado no PFC.

2.1.4 A Configuração do PFC

Todas as três configuraçöes analisadas são usadas em interfaces de rede. Qual delas seria mais apropriada depende muito de fatores e características da implantaçäo. Em alguns casos o custo pode ser de maior importância do que o desempenho e em outros casos pode ser o contrário.

Consideramos que no ambiente industrial o desempenho é o mais importante dos dois. Normalmente, os sistemas implantados no ambiente industrial têm que ser conformes com uma série de especificaçöes que determinam níveis mínimos de desempenho. Alguns exemplos säo:

- tempo de resposta a um evento;
- número de eventos processados por unidade tempo e
- tempo de recuperaçäo depois de um evento extraordinário (reset de sistema, erro de protocolo da rede, etc).

Assim, uma estaçäo que tem uma UCP potente pode utilizar uma interface com memória sem falhar nas suas especificaçöes. Por outro lado, uma estaçäo com uma UCP menos potente necessita de uma interface inteligente.

Säo três as razöes para a escolha da interface inteligente para este projeto:

- uma interface inteligente evita sobrecarga da estaçäo;
- o processamento exigido pela rede é alto (uma alta velocidade de transmissäo);
- a modularizaçäo e hierarquia do software säo melhor exploradas com uma interface inteligente.

Por estas razöes os PFCs säo uma tendência internacional [SPA 86].

2.2 A Escolha dos Componentes

Depois de definir a configuração do hardware faz-se necessária a escolha dos componentes principais tais como o controlador, o microprocessador e a memória. Sempre que possível, optou-se pelo uso de componentes de fácil aquisição e utilização, um exemplo sendo do uso de lógica convencional (família 74) ao invés de PALs ("Programmable Array Logic") devido à falta de recursos locais para a programação destes PALs. Tal estratégia implica que o circuito possa ser 'enxugado' numa fase posterior.

2.2.1 O Controlador de Rede

Atualmente existem dois controladores de rede usados nas interfaces MAP: o WD2840 e o MC68824.

O WD2840 da Western Digital não é totalmente compatível com o padrão 802.4, que é o barramento com ficha usado na MAP. Também, ele é um controlador simples que deixa a maior parte da carga da MAC para o microprocessador.

No projeto do PFC optou-se pelo MC68824 da Motorola que é 100% compatível com o 802.4, além de qual ele é parcialmente inteligente, sendo capaz de receber e transmitir cadeias completas de quadros sem a intervenção do microprocessador. Ele executa também um conjunto de funções auxiliares que simplificam a implementação da camada MAC e aumentam a eficiência da interface.

Na sua documentação técnica o MC68824 é chamado de TBC ("Token Bus Controller") e este será o nome usado no restante deste texto.

2.2.2 O Microprocessador

Devido ao uso de um controlador de rede da Motorola, a escolha de um microprocessador teria sido naturalmente da família 68000 que é também da Motorola. Neste caso haveria um casamento perfeito em termos de sinais elétricos, de protocolos (por exemplo, o protocolo de arbitragem de barramento e de reconhecimento de interrupção) e das convenções (por exemplo, a ordem dos bytes e a representação de endereços pelos apontadores) entre os dois CI's.

Entretanto, em função de uma disponibilidade muito maior de equipamento e ferramentas de apoio ao projeto baseados em microprocessadores da Intel, optou-se por utilizar um microprocessador desta família como UCP do processador frontal de comunicação. Fator determinante nesta escolha foi o fato da possibilidade de utilizar o PP (baseado no 80286) para desenvolvimento de software.

Dos microprocessadores membros da família Intel o considerado mais apropriado para o PFC foi o 80186 de 10 MHz [INT 83A] [INT 83B] pelas seguintes razões:

- ele dispõe da capacidade mínima de processamento considerada adequada para uma interface do tipo MAP [MOT 87C];
- ele permite um hardware compacto pois tem circuitos embutidos de DMA, relógio, temporização e controle de interrupção;
- ele não exige um projeto super-avançado de hardware como é necessário para os microprocessadores mais avançados e que seria além das capacidades deste trabalho.

2.2.3 A Memória

Para determinar a quantidade e o tipo de memória a serem utilizadas no PFC havia a necessidade de um estimativa do tamanho do

Executivo, das tarefas e dos buffers de recepção e transmissão. Os tamanhos iniciais tomados como guias para este fim foram os seguintes:

Executivo	16k
Tarefa (Camada de protocolo)	32 - 128k

Os valores para o Executivo e a tarefa incluem a memória alocada dinamicamente para eles. As alternativas de configuração inicialmente previstas para o PFC foram duas:

- suportando uma arquitetura mini-MAP e
- suportando uma arquitetura MAP completa.

Para a arquitetura mini-MAP a estimativa de demanda de memória foi a seguinte:

Executivo	**16k**
Tarefa (LLC)	**32k**
Tarefa (MMS)	128k
----------	----
Total	176k

Para a arquitetura MAP a estimativa de demanda de memória foi a seguinte (presumindo que a camada de aplicação e os aplicativos residem no hospedeiro):

Executivo	**16k**
Tarefa (LLC)	**32k**
Tarefa (Rede)	**64k**
Tarefa (Transporte)	**64k**
Tarefa (Sessão)	64k
Tarefa (Apresentação)	128k
----------	----
Total	368k

Arredondando as estimativas acima, conclui-se que a memória do PFC deve ter algo entre 256 kbytes para a arquitetura mini-MAP e 512 kbytes para a arquitetura MAP.

Estas estimativas são importantes porque elas influenciam a escolha entre RAM estática ou RAM dinâmica. Para as quantidades de memória calculadas os dois tipos de RAM ocupariam o mesmo espaço físico e teriam aproximadamente o mesmo custo (a RAM dinâmica precisa de vários CIs de suporte).

Como a filosofia de implementação do hardware é de manter os circuitos o mais simples possível, a decisão foi de usar RAM estática. Por razões de disponibilidade, pastilhas de 32k por 8 bits foram usadas, mas a decisão foi tomada com base da disponibilidade de pastilhas de 128k por 8 bits. Assim, elimina-se a complexidade implícita no projeto de memórias dinâmicas (controlador, consumo elevado de corrente, estados de espera, etc).

2.3 A Modularização do Hardware

Na sua estruturação o projeto de hardware é dividido em quatro circuitos principais:

- circuito do controlador de rede;
- circuito do microprocessador;
- circuito de memória e
- circuito da interface PP/PFC.

A seguir faz-se uma descrição destes quatro circuitos. Os diagramas de circuito correspondentes encontram-se no apêndice A.

2.3.1 O Circuito do Controlador de Rede

Este circuito é composto do TBC e de alguns circuitos lógicos

usados principalmente para adaptar os sinais do TBC aos sinais correspondentes do microprocessador 80186.

O TBC pode trabalhar com um relógio de até 12,5 MHz, mas nesta implementação ele utiliza o relógio do microprocessador 80186 que é de 10 MHz. Note-se que a freqüência de operação do TBC pode ser diferente da freqüência de operação da rede.

Seria possível, num trabalho futuro, incluir um circuito gerador de relógio de 12,5 MHz especificamente para o TBC, aumentando ligeiramente o desempenho do PFC. O TBC não utiliza um sinal de reset - esta função está sendo feita por software por um comando de reset lançado pelo microprocessador.

Os sinais SYMREQ, TXCLK, TXSYM0, TXSYM1, TXSYM2, SMIND, RXCLK, RXSYM0, RXSYM1, e RXSYM2 compõem a interface com o circuito integrado que faz a modulação e demodulação dos sinais da rede (o modem).

O TBC opera em dois modos. No modo passivo ele age como periférico do 80186, ocupando espaço no mapa de entrada/saída com seus registradores de comando e de estado. Quando ocorrem eventos na rede, o TBC entra no modo ativo, pedindo e esperando a arbitragem da memória. Quando recebe a memória o TBC ativa seus sinais de endereço, dados e controle para fazer acesso.

Enquanto no modo passivo o sinal R/-W fornecido ao TBC tem que ser ativado desde o início do ciclo de acesso. Porém o sinal R/-W do 80186 é gerado depois do início do ciclo. Para solucionar este problema o CI34 usa os sinais de estado do 80186 para gerar um sinal R/-W adiantado. No início do ciclo atual este R/-W adiantado é passado para o TBC através do CI35.

Os sinais -UDS ("Upper Data Select") e -LDS ("Lower Data Select") do TBC são quase equivalentes aos sinais -BHE ("Byte High Enable") e A0 ("Address 0") do 80186, porém eles nunca entram em alta impedância ("tri-state"), sendo necessária portanto o isolamento deles com o buffer

CI37.

Quando o TBC precisa fazer acesso a memória, ele ativa o sinal -BR ("Bus Request"). Este sinal é memorizado ("latched") pelo CI52b para gerar o sinal TBCREQ (TBC Request) que vai ao circuito de arbitração de memória no circuito do microprocessador. Depois do TBC ter terminado seu acesso à memória a desativação dos sinais -BR e -BRACK ("Bus Request ACKnowledge") força a desativação do sinal TBCHOLD.

O sinal TBCREQA ("TBC REQuest Acknowledge") é gerado pelo circuito de arbitração para indicar que o TBC pode ter acesso a memória. Através do "latch" CI52a este sinal é convertido para o sinal -BG ("Bus Grant") do TBC. O tempo de latência do processo de arbitragem é de 8-12 ciclos dependendo da instrução corrente em execução no 80186. Uma vez que o TBC tenha a memória ele a mantém até completar uma seqüência DMA.

O sinal -IRQ ("Interrupt Request") do TBC é invertido e passado para o módulo do microprocessador onde gera uma interrupção.

2.3.2 O Circuito do Microprocessador

O circuito do microprocessador é composto do microprocessador 80186, buffers de dados, "latches" de endereço e o circuito de arbitragem de memória.

O 80186 do PFC trabalha com um relógio de 10 MHz. A saída de relógio do 80186 (o sinal CLKOUT) fornece o relógio para os demais circuitos, inclusive o TBC.

Os barramentos de dados e endereço do 80186 são multiplexados. Os CI's CI32 e CI33 são "latches" que demultiplexam as linhas A0 - A15 de endereço. O CI31 é o "latch" para as linhas A16 - A19 de endereço.

O CI CI55a desabilita a saída dos buffers e "latches" enquanto o PFC está sob "reset". Sem esta desabilitação haveria conflito com os buffers de dados do circuito de interface PP/PFC quando o PP está

inicializando a memória do PFC.

O 80186 não dispõe de um sinal de M/IO (Memória/Entrada Saída) mas o sinal S2 efetivamente faz a mesma função. Os CIs CI55d, CI56a e CI54b geram o sinal de habilitação de memória sob as seguintes condições:

- "reset" desabilitado;
- a memória é arbitrada ao 80186;
- o sinal ALE ("Address Logic Enable") do 80186 está ativo e
- o sinal S2 indica uma operação de acesso à memória.

O circuito de arbitração é baseado num circuito seqüencial, sendo composto dos "flip-flops" CI50a e CI50b. Estes usam fases alternadas do relógio de 10 MHz para gerar os sinais de pedido de memória PPREQ ("PP REQuest") e TBCREQ ("TBC REQuest"). Quando um destes sinais estiver ativo a porta CI55b habilita o sinal HOLD do 80186. O uso de fases opostas do relógio e a interligação dos dois "latches" garante que somente um dos sinais PPREQ e TBCREQ podem estar ativos por vez.

2.3.3 O Circuito da Memória

O circuito de memória é composto da memória em si mais os decodificadores de endereço. A memória usada é a 62256 (RAM estática - 32k por 8 bits). Como pode ser visto no diagrama de circuito, o uso desta RAM estática resultou num circuito de memória bastante simples.

A memória do PFC é organizada em até oito filas de 64 kbytes. Cada fila é composto de duas memórias 62256 - uma para os bits 0 a 7 e a outra para os bits 8 - 15. As oito memórias para os bits 0 - 7 formam o banco par e as outras formam o banco ímpar.. A Figura 2.4 mostra o organização da memória.

Cada banco tem seu decodificador 3:8 (CI38 e CI39) que decodifica as linhas de endereço A16, A17, e A18 para selecionar a fila. As chaves S3,S4 e S5 selecionam o número de circuitos de memória instalados. As

configurações possíveis são mostradas na tabela 2.1.

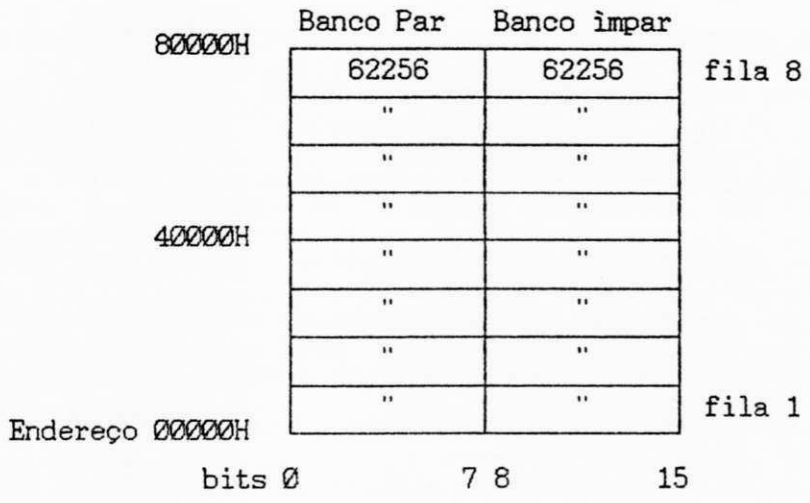


Figura 2.5 - A Organização da Memória

S3	S4	S5	Número de filas	Quantidade Total de Memória
A	A	A	1	64k
A	A	F	2	128k
A	F	F	4	256k
F	F	F	8	512k

A - Aberta
F - Fechada

Tabela 2.1 - Configuração das Chaves de Memória

A ativação do banco par é feita com o sinal AØ, e do banco ímpar com o sinal BHE ("Byte High Enable").

O microprocessador 80186 utiliza duas áreas de memória para fins específicos. A primeira área é o primeiro kilobyte a partir do endereço 00000H da memória que é usada para os vetores de interrupção. A segunda área são 16 bytes no fim da memória (0FFFF0H) que são usados para o código de partida que normalmente contém uma instrução de salto para o código de "bootstrap". Estas duas áreas têm que estar presentes do ponto de vista do 80186.

Como o PFC dispõe de no máximo 512 kbytes, que não são em si o suficiente para abranger estas duas áreas, foi adotada uma decodificação de memória que resulta em várias imagens da memória, ou seja, onde são criadas múltiplas imagens da memória realmente presente. Estas imagens preenchem completamente o mapa de memória do 80186. Por exemplo, no caso onde há 256 kbytes de memória instalada no PFC o mapa de memória é composto de quatro imagens destes 256 kbytes. Assim, quando o 80186 faz acesso ao endereço lógico de 0FFFF0H para obter as instruções de partida na realidade ele faz acesso ao endereço físico 03FFF0H.

Com a chegada iminente da RAM estática de 64K por 8 bits, o PFC poderia ser equipado com o dobro da quantidade de memória atual, ou seja, com 128, 256, 512 ou 1024 kbytes. Neste caso as linhas A17, A18, e A19 seriam usadas como decodificadoras de endereço.

2.3.4 O Circuito da Interface PP/PFC

O circuito da interface PP/PFC permite ao PP ter acesso à memória e à porta de controle do PFC. Em termos de componentes este circuito contém o maior número de CIs, sendo composto dos buffers de barramento e do circuito de decodificação de endereço.

A especificação do barramento do PP impõe limites no carregamento de cada sinal do barramento. Para não ultrapassar estes limites todos os sinais do barramento utilizados são desacoplados através de buffers. Os

buffers CI24 e CI25 são usados para os sinais de dados DØ - D15. Os "latches" CIØØ, CI21, e CI22 são utilizados para os sinais de endereço. O "latch" CI23 é usado para os sinais de controle.

A parte central deste circuito é o decodificador de endereço. O PP suporta até oito placas de expansão, cada placa tendo alocada uma faixa de endereços de memória e de entrada/saída. As chaves S1, S2, e S3 do PFC determinam o endereço do PFC. As chaves selecionam a faixa de endereço de acordo com a tabela 2.2.

O circuito de decodificação deste módulo gera o sinal -PPIO quando o PP faz acesso a um endereço dentro da faixa de entrada/saída selecionada pelas chaves. Da mesma forma ele gera o sinal -PPMEM quando o endereço do PP se situa dentro da faixa selecionada de memória. Quando um destes sinais é ativado o sinal PPREQ é ativado indicando ao circuito de arbitração que o PP quer ter acesso à placa.

S1	S2	S3	Faixa de Memória	Faixa de E/S
A	A	A	cØØØØ-c7fff	ff4Ø-ff47
A	A	F	c8ØØØ-cffff	ff48-ff4f
A	F	A	dØØØØ-d7fff	ff5Ø-ff57
A	F	F	d8ØØØ-dffff	ff58-ff5f
F	A	A	eØØØØ-e7fff	ff6Ø-ff67
F	A	F	e8ØØØ-e8fff	ff68-ff6f
F	F	A	fØØØØ-f7fff	ff7Ø-ff77
F	F	F	f8ØØØ-f8fff	ff78-ff7f

A - Aberta
F - Fechada

Tabela 2.2 - Configuração das Chaves de Endereço

O sinal -PPHOLDA ("PP HOLD Acknowledge") é ativado quando o circuito de arbitragem passa a memória para o PP. Este sinal habilita os buffers de endereço e sinais de controle da memória -BHE, R/-W e -MEM. O "latch" CI40b garante um ciclo do relógio (100ns) depois da ativação de -PPHOLDA antes que os buffers de dados CI24 e CI25 e o sinal -NBRY (Bus ReadY) estejam ativados. Estes ciclo é necessário para permitir que o endereço de acesso do PP seja passado para a memória. O sinal -NBRY indica ao PP que ele pode continuar seu ciclo de acesso.

Como o circuito de decodificação tem que estar permanentemente ativo, os buffers CI29 e CI30 isolam os sinais de endereço usados por este circuito dos sinais de endereço usados no restante da placa.

Os CI28 e CI40a formam a porta de controle do PFC usada para controlar o sinal RESET e a seleção de janela. Qualquer uma das quatro palavras dentro da faixa de entrada/saída faz acesso a esta porta, que é de escrita apenas. O estado do sinal RESET depende do endereço de acesso desta porta de controle - um acesso a um endereço par ativa o sinal RESET enquanto um acesso a um endereço ímpar o desativa.

Os bits 0 - 4 da porta de controle selecionam a janela de memória a qual o PP quer ter acesso. Cada janela é de 32 kbytes. O CI28 é um "latch" que contém o valor da janela atual. Este valor pode ser alterado através da escrita dos bits 0-4 da porta de controle.

Cada acesso à memória pelo PP sofre de um atraso devido a arbitragem. Este atraso é da ordem de 6-8 ciclos do relógio (ou seja, com relógio de 10 MHz o atraso é de até 1 microsegundo).

3. O EXECUTIVO

Este capítulo descreve o núcleo multi-tarefa do PFC que é chamado de Executivo. Este capítulo começa com uma identificaçäo dos serviços que o Executivo deve oferecer e as características que ele deve ter para satisfazer as necessidades desta interface de rede mini-MAP. Em seguida é feita uma análise dos módulos principais do Executivo com uma abordagem dos seus mecanismos internos. Para cada módulo é apresentada uma declaraçäo das primitivas que säo usadas pelas tarefas para a invocaçäo dos seus serviços. No final deste capítulo há uma descriçäo do procedimento usado para a criaçäo e instalaçäo das tarefas no PFC.

3.1 O Perfil do Executivo

O Executivo foi criado para servir às necessidades de interfaces de rede. Os serviços que ele fornece säo eficientes e apropriados neste respeito. Há um grau de generalidade nestes serviços para acomodar as necessidades diferentes das tarefas que compöem uma interface de rede. Esta generalizaçäo permite o emprego do Executivo em outros projetos. A

seguir há uma discussão das características principais do Executivo.

3.1.1 A Criação Dinâmica de Tarefas

Uma facilidade normalmente utilizada em núcleos multi-tarefa é a de criação dinâmica de tarefas, ou seja, a possibilidade de criar novas tarefas depois de concluído o processo de inicialização.

Quando se analisa uma interface de rede, verifica-se que a configuração das tarefas pode ser estática [JOB 90]. Por exemplo, no caso de uma interface mini-MAP, poderia haver duas tarefas executando no PFC - uma que representa as camadas MAC e LLC e a outra que representa a camada MMS. Não é estritamente necessário ter-se a criação dinâmica de tarefa para implementar uma interface de rede.

Sem criação dinâmica de tarefas a memória do PFC pode ser reservada na fase de inicialização para fins específicos como, por exemplo, para as próprias tarefas ou para a alocação dinâmica de blocos de memória. Isso ajuda a manter o Executivo simples. A introdução da criação dinâmica de tarefas implica na inclusão de mecanismos de gerenciamento de memória do tipo compactação e coletor de lixo ("garbage collection") [HOR 82] o que significa um grande aumento na complexidade do Executivo.

Na nossa implementação o Executivo não dispõe do mecanismo de criação dinâmica de tarefa. Porém, é possível que numa futura versão dele seja criado um mecanismo de criação de processos leves ("threads") [LAC 88], onde poderia existir múltiplos "threads" de execução para cada tarefa.

Por exemplo, no caso da camada LLC poderia haver um "thread" para cada componente de recepção e de transmissão. Como a eficiência disso é questionável a inclusão do mecanismo de "threads" dependerá de um estudo mais profundo sobre seu aptidão para as tarefas que representam as

camadas de protocolos.

3.1.2 A Mudança de Contexto

Sendo o Executivo um núcleo multi-tarefa ele necessita de um mecanismo de mudança de contexto. Isso normalmente significa:

- a salvação dos registradores e o endereço de execução da UCP referentes a uma tarefa; e
- a restauração destes itens referentes a uma outra tarefa.

Mas como as camadas de protocolos podem ser consideradas como maquinas de estado, surge-se uma outra possibilidade para a mudança de contexto, uma em que são salvos e restaurados os estados destas maquinas de estado.

A primeira vista, esta possibilidade parece apropriada para as camadas inferiores da pilha de protocolos. Porém, seu aptidão para as camadas superiores não é claro e esta possibilidade não seria perseguida mais neste trabalho por falta de um estudo mais profundo.

Em termos de mudança de contexto convencional, existem dois mecanismos alternativos:

- a mudança preemptiva; e
- a mudança cooperativa.

Para decidir qual seria a melhor para o Executivo faz-se necessário um exame destas alternativas [COM 84] [DEI 84] [LAC 88]. Na mudança cooperativa o programador de cada tarefa decide por quanto tempo sua tarefa segura a UCP, enquanto na mudança preemptiva um relógio tempo-real força a mudança periodicamente.

Sistemas operacionais multi-usuários como o OS/2 [LET 88] [LAC 88] e o UNIX [GRO 86] utilizam a mudança preemptiva [GLA 88]. Normalmente, a mudança cooperativa é usada em sistemas de pequeno porte como, por exemplo, o MS-DOS com Windows [JAM 87], onde o próprio fluxo na execução

do código pode ser definido `a priori'. A mudança preemptiva é considerada mais satisfactória tecnicamente porque ela implica num escalonamento `determinístico' onde o acesso à UCP é garantida para cada tarefa. A mudança preemptiva facilita a implementação das tarefas porque o programador não precisa se preocupar em como sua tarefa `segura' a UCP.

Para ambas as técnicas a maior parte da implementação da mudança de contexto corresponde aos mecanismos de salvamento e recuperação de contexto. A parte da mudança preemptiva que lida com o relógio tempo-real é de pequena complexidade. Assim, a escolha entre as duas técnicas não depende da dificuldade da implementação.

Pode-se argumentar que a mudança cooperativa seria mais eficiente para as tarefas que representam as camadas inferiores da pilha de protocolos porque nestas camadas a execução é do tipo orientada a transação, ou seja, em que o processamento é composto de eventos independentes de comando e resposta. Neste caso, por razões de eficiência, a tarefa deve segurar a UCP até terminar todas as transações pendentes, depois de qual ela pode passar controle para uma outra tarefa.

Em conseqüência desta considerações, optou-se pela implantação de um mecanismo default de mudança de contexto preemptiva mas com a inclusão de primitivas para sua desabilitação em favor de mudança cooperativa. Assim, é o projetista da interface quem escolha o mecanismo mais apropriado.

Em relação à mudança preemptiva, o período do relógio tempo-real deve ser suficientemente curto para permitir as tarefas executarem `suavemente', Um período curto também permite uma resolução razoável para os mecanismos de temporização a serem usados. Por outro lado, o período não deve ser tão curto que a sobrecarga da mudança de contexto comece a ser significativa (consideramos uma sobrecarga `significativa'

como sendo superior a 5%).

3.1.3 O Escalonamento

O processo de escalonamento é o mecanismo pelo qual o núcleo determina qual tarefa seria a próxima a receber a UCP. Existem várias formas de escalonamento. A mais simples é a de seleção circular ("round robin" [TAN 87]), onde a UCP é passada sempre na mesma seqüência num anel de tarefas. Para a maioria dos sistemas este método de escalonamento é ineficiente porque ele não incorpora nenhuma maneira para o escalonador alocar tempo na UCP de acordo com as necessidades estáticas ou dinâmicas das tarefas. Porém, pode ser que para uma interface de rede esta forma seja adequada, com execução passando sucessivamente para cada camada da interface.

Um outro mecanismo utilizado é baseado no uso de prioridades para cada tarefa. Quando a tarefa recebe a UCP sua prioridade é diminuída. Na medida que ela não recebe a UCP sua prioridade é progressivamente aumentada. O escalonador escolhe a tarefa com a prioridade mais alta naquele instante. Uma extensão deste método é para aquelas tarefas que consomem muito a UCP (sem fazer operações de entrada/saída) terem suas prioridades diminuídas em maior grau.

Um outro mecanismo é aquele onde cada tarefa tem uma prioridade inicial que serve como sua prioridade "default". Esta prioridade pode ser alterada dinamicamente pela própria tarefa conforme a carga atual da tarefa. No instante de selecionar a próxima tarefa o núcleo leva em conta esta prioridade e também o tempo transcorrido desde a última vez que a tarefa executou. Usando este mecanismo o configurador do PFC poderia manipular as prioridades iniciais das tarefas numa tentativa de balancear ou otimizar o desempenho global.

Devido ao ambiente tempo-real do PFC faz-se necessário um

escalonamento que garanta a maior eficácia nas diversas condições. Foi definido então para o Executivo um mecanismo de escalonamento de tarefa baseado em prioridade. A cada tarefa é dada uma prioridade "default" pelo configurador do PFC. Através de primitivas a própria tarefa pode alterar dinamicamente sua prioridade de acordo com as suas necessidades.

Note-se que se as prioridades forem iguais então se obtem, efetivamente, escalonamento "round-robin".

3.1.4 O Gerenciamento de Memória

Todos os sistemas operacionais, sejam eles de grande ou pequeno porte, mono ou multi-tarefa, têm mecanismos internos de gerenciamento de memória [HYD 88]. Para os sistemas em que não existe um hardware especial para esta função o gerenciamento de memória consiste basicamente na alocação dinâmica de memória e os mecanismos associados de compactação e recuperação de memória.

Em termos do PFC a memória alocada dinamicamente seria usada principalmente para as mensagens sendo processados e as primitivas de serviço intercambiadas entre as tarefas. O mecanismo de alocação tem que ser rápido para que uma tarefa com um alto fluxo de dados possa pedir mais memória em tempo suficiente para não perder mensagens.

Para facilitar a implementação das tarefas o Executivo deve permitir a alocação de memória em blocos de comprimento arbitrário. Porém, por razões de eficiência, o Executivo deve alocar memória internamente em múltiplos fixos de bytes. Esta 'granularização' permite o uso de inteiros para representar os endereços e tamanhos dos blocos livres, aumentando assim a rapidez do processo de alocação.

Uma granularização fina reduz o desperdício de memória, enquanto que uma granularização grossa reduz a fragmentação da memória e torna o processo de alocação mais rápido. Através de um parâmetro de

inicialização o configurador do PFC deve ser capaz de especificar esta granularização, de acordo com a quantidade de memória instalada no PFC, assim como o número e tamanho das tarefas nele instaladas.

Se houver um hardware de suporte para gerenciamento de memória então existe a possibilidade de proteção e paginação da memória [COM 84] [GUI 80][HEN 88][LAC 88]. Estes dois mecanismos são normalmente vantajosos em ambientes multi-tarefa porque a configuração das tarefas está constantemente mudando e também, as vezes, é executado software que não está totalmente depurado. Porém, num ambiente estático com software já totalmente depurado a proteção e paginação de memória não oferecem grandes vantagens e implicam numa sobrecarga de processamento.

Do ponto de vista do hardware, o microprocessador mais indicado, o 80186, não dispõe dos circuitos de suporte para este fim. Por estas razões o Executivo oferece alocação dinâmica de memória mas não oferece proteção ou paginação de memória.

3.1.5 A Comunicação Inter-Tarefa

São vários os mecanismos existentes para a comunicação inter-tarefa [GLA 88][LAC 88][LET 88]:

- tubos ("pipes")
- tubos nomeados ("named pipes")
- filas
- mensagens
- memória comum
- "mailboxes"

Para determinar o mecanismo mais adequada para o PFC faz-se necessário um exame da maneira como as tarefas de uma interface do tipo MAP se comunicam.

As especificações das camadas na arquitetura MAP introduzem o

conceito de `primitiva de serviço' que representa um grupo de informações que é passado entre as camadas. Nestas especificações não existe uma definição da forma de implementação destas primitivas. Na prática uma primitiva seria uma estrutura de dados passada de uma tarefa para outra. Do ponto de vista do Executivo isto implica numa passagem de mensagens entre as tarefas.

A passagem de uma mensagem de uma tarefa para outra tem que ser feita de maneira eficiente devido às características de `tempo-real' da interface de rede.

O mecanismo de tubos ("pipes") usado no UNIX [GRO 86] copia uma mensagem byte por byte num buffer interno para depois copiar byte por byte para a tarefa final. Este é um processo ineficiente demais para o PFC.

O mecanismo de "mailbox" pode ser usado para passar um apontador para a mensagem que representa uma aumento considerável em eficiência em relação aos tubos. Porém, com "mailboxes" a tarefa tem que se responsabilizar pelo processo de enfileiramento das mensagens.

Uma maneira mais eficiente, que foi adotada para o Executivo, consiste em criar a mensagem, pela tarefa, já com um cabeçalho para uso interno do Executivo. Usando um apontador neste cabeçalho o Executivo pode encadear as mensagens em filas de recepção sem ter que copiá-las em buffers internos ou intermediários.

Consideremos um exemplo explicando o funcionamento deste mecanismo numa interface de rede. Uma tarefa mantem um conjunto de buffers como reserva. Estes buffers vêm da alocação dinâmica de memória. Para mandar uma mensagem, a tarefa preenche uma estrutura de dados em um destes buffers. Esta estrutura de dados representa a primitiva de serviço entre as duas tarefas. Através de uma primitiva a tarefa informa ao Executivo que este buffer contém uma mensagem para a outra tarefa. O Executivo adiciona este buffer (usando o apontador no seu cabeçalho) na cadeia de

recepção da outra tarefa. Depois de ter processado a mensagem esta outra tarefa pode instalar este buffer usado na sua reserva de buffers para uso posterior.

3.1.6 A Temporização

O uso de temporização é importante no ambiente de rede. Os temporizadores são usados para determinar a destruição de variáveis de estado e para verificar o recebimento de reconhecimento de mensagens.

O Executivo deve oferecer um mecanismo de temporização que gerencie um número elevado de temporizadores de maneira eficiente. Como já foi mencionado, o uso de um relógio de tempo-real de período 20ms fornece uma resolução temporal adequada para uma interface de rede.

3.2 A Modularização do Executivo

Objetivando uma melhor estruturação, o Executivo foi dividido em quatro gerentes conforme ilustrado na figura 3.1. Cada gerente é responsável por um grupo distinto de serviços.

O núcleo do Executivo contém o código de inicialização, a rotina do relógio de tempo-real, e as estruturas de dados necessários para os mecanismos de mudança de contexto, alocação dinâmica, etc. O Gerente de Execução é responsável pela gerência das tarefas que estão sendo executadas sob o Executivo, oferecendo uma série de primitivas para este fim. Estas primitivas incluem as de mudança de prioridade, de suspensão de tarefa e de mudança cooperativa de tarefa. O Gerente de Temporização oferece serviços de criação, leitura e remoção de temporizadores. O Gerente de Comunicação oferece serviços de troca de mensagem entre as tarefas do PFC e também entre as tarefas e o Hospedeiro. Por último, o Gerente de Memória oferece serviços de alocação e liberação dinâmica de

memória.

A interface do Executivo é um mecanismo através do qual as tarefas podem invocar as primitivas das quatro gerências. Este mecanismo é implementado através de uma interrupção em software.

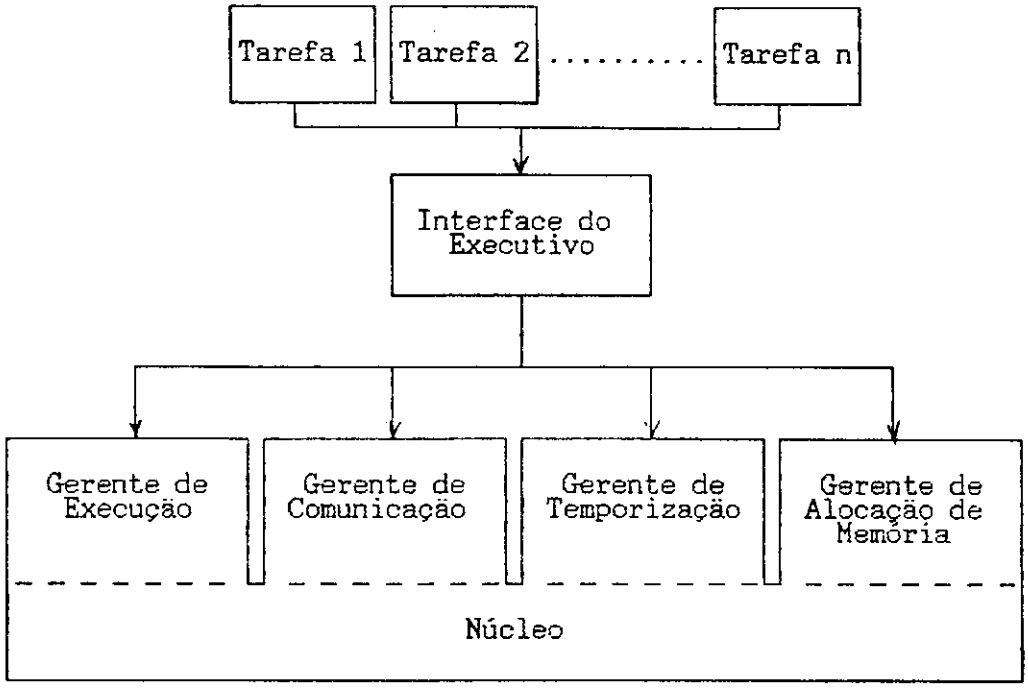


Figura 3.1 A Modularização do Executivo

3.3 O Núcleo

O núcleo do Executivo é responsável pelo relógio de tempo-real que inicia o mecanismo de mudança preemptiva de contexto. O Executivo usa um escalonamento de tarefas baseado em prioridades. Antes de descrever o processo da mudança de contexto em si, é definida em seguida o que o contexto de uma tarefa representa.

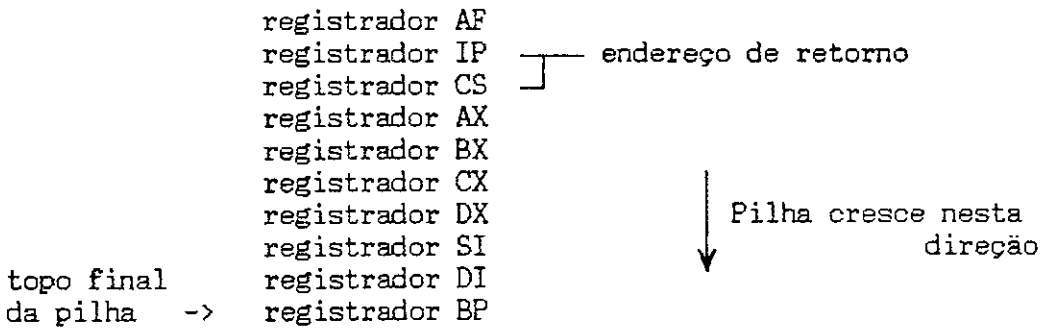
O contexto de uma tarefa é salvo em dois casos. Primeiro, quando ela invoca uma primitiva do Executivo e, segundo, quando ocorre uma

interrupção do relógio de tempo-real. O contexto da tarefa é implementado pelo empilhamento de todos os registradores do microprocessador para formar um quadro de contexto na pilha. O endereço deste quadro é armazenado junto com outros dados da tarefa numa estrutura de dados chamada de *task_data* que é a estrutura mais importante do Executivo. O array *task* é de estruturas *task_data*, um elemento para cada tarefa, conforme indicado abaixo:

```
struct task_data {
        unsigned priority;
        unsigned countdown;
        struct message *msg;
        struct timer *timer;
        struct timer *timeout;
        void *stack;
} task [MAX_TASKS];
```

É o apontador *stack* que armazena a posição do quadro de contexto.

O processo de mudança de contexto começa com uma interrupção. No caso da invocação de uma primitiva a interrupção é feita por software através da instrução INT do 80186. No caso do relógio de tempo-real a interrupção é feita pelo hardware. Os dois tipos de interrupção têm suas rotinas de serviço específicas, mas implementam o mesmo procedimento. Como resultado da interrupção o registrador AF do 80186 e o endereço de retorno são empilhados. A rotina de interrupção empilha todos os demais registradores, formando um quadro de contexto na pilha com o seguinte formato:



A rotina de interrupção passa o apontador de pilha para uma função do Executivo. No caso do relógio de tempo-real esta função é chamada *tick*, e no caso da invocação de uma primitiva há uma função específica para cada primitiva. A função armazena este apontador na variável *stack* da estrutura *task* da tarefa. Neste ponto, o contexto da tarefa está salvo. Para restaurar o contexto, este apontador de pilha é passado para a função *restore_context* escrita em assembler que desempilha todos os registradores e faz um retorno de interrupção, a execução passando assim para a nova tarefa exatamente no ponto em que a tarefa parou de executar no período de execução anterior. Há três situações sob as quais uma mudança de contexto acontece:

- mudança preemptiva iniciada pelo relógio de tempo-real a cada 20ms;
- mudança cooperativa iniciada pela própria tarefa com uso de uma primitiva para este fim e
- mudança induzida com o uso de uma primitiva do tipo *x_sleep* que suspende a tarefa atual.

O mecanismo de mudança preemptiva é o seguinte. Um dos temporizadores do 80186 é usado como relógio de tempo-real. Ele é programado para gerar uma interrupção a cada 20ms (50Hz). A rotina de interrupção deste temporizador cria o quadro de contexto (salvando o contexto) e chama a função *tick* que é escrita na linguagem C. Depois de atualizar os temporizadores e examinar a interface PFC/PP para entrada e saída de mensagens, a função *tick* chama a função *get_ready_task* que devolve o número da próxima tarefa a ser executada. A função *tick* tira da estrutura *task* o apontador de pilha para esta tarefa e o passa para a função *restore_context*, que inicia a execução desta tarefa.

A função *get_ready_task* usa a seguinte técnica para selecionar uma tarefa. Cada tarefa na sua estrutura *task* tem a variável *priority* e a variável *countdown*. A variável *priority* é a prioridade "default" da

tarefa, e a variável *countdown* é usada para selecionar a próxima tarefa a ser executada. A variável *countdown* de cada tarefa é inicializada com o valor da variável *priority*. A função *get_ready_task* entra no laço seguinte:

```
int get_ready_task(void)
{
   for (;;) {
      if (++xtask > lasttask)
          xtask = 1;
      if (task[xtask].sleep == Ø) {
          if (--task[xtask].countdown == Ø) {
              task[xtask].countdown = task[xtask].priority;
              return xtask;
          }
      }
}
```

A variável *xtask* é global e contém o número da tarefa atualmente em execução. A função continua em laço decrementando a variável *countdown* de cada tarefa que não está suspensa (o *sleep* maior de que zero) até uma delas chegar ao valor zero. Então esta variável *countdown* é reinicializada com o valor de prioridade da tarefa (contido na variável *priority*) e a identidade da tarefa é passada de volta à função *tick*.

Neste mecanismo o tempo de UCP recebido por tarefa é inversamente proporcional ao valor numérico da prioridade (o valor 1 sendo o de mais alta prioridade). Se houver tarefas T1, T2, ...Tn com prioridades iguais a P1, P2,...Pn, a proporção do tempo de UCP que a tarefa Ti receberá é dada por:

$$\frac{1/Pi}{1/P1 + 1/P2 + \ldots + 1/Pi + \ldots\ldots + 1/Pn} \times 100\%$$

3.4 A Interface ao Executivo

Os serviços do Executivo são ativados através de primitivas. Para cada primitiva há uma pequena rotina de interface escrita em "assembler". O conjunto destas rotinas é contido no arquivo *taskasm.obj* que é link-editado com o código da tarefa.

Cada função de interface carrega o registrador BX do 80186 com um valor que representa a primitiva sendo invocada e depois usa a instrução INT do 80186 para invocar uma interrupção de software (especificamente a instrução INT 32). Os parâmetros da primitiva permanecem na pilha da tarefa.

A rotina INT do Executivo salva o contexto da tarefa e usa o valor no registrador BX para chamar a função de serviço apropriada. Ela passa dois parâmetros. O primeiro é o apontador de pilha da tarefa que a função chamada necessita para restaurar o contexto depois de ter executado a primitiva. Caso a função venha a necessitar, ela pode guardar este apontador e restaurar o contexto de uma outra tarefa. O segundo parâmetro é um apontador para os parâmetros da primitiva na pilha da tarefa - a função usa este apontador para obter estes parâmetros.

3.5 O Gerente de Execução

O Gerente de Execução é responsável pela gerência das tarefas do Executivo. Através de primitivas as tarefas podem alterar dinamicamente suas prioridades, podem se suspender por um período de tempo específico, podem se suspender até o acontecimento de um evento (estouro de temporizador, chegada de mensagem), e podem executar uma mudança cooperativa de contexto. A seguir há uma descrição dos mecanismos internos do Gerente do Execução seguida por uma definição das primitivas

que este oferece.

3.5.1 Os Mecanismos do Gerente de Execução

Um dos serviços oferecidos pelo Gerente de Execução é o de mudança de prioridade. A primitiva que executa este serviço pöe o novo valor de prioridade na variável *priority* da estrutura *task* da tarefa.

Um outro serviço é o de suspensão de tarefa por um período específico. A primitiva x_sleep que implementa o serviço copia o período de suspensão da tarefa na variável *sleep* da estrutura *task* da tarefa. A cada pulso de relógio que segue este valor em *sleep* é decrementado até chegar de novo em zero. A partir deste momento a tarefa é mais uma vez elegível para receber a UCP.

Um outro tipo de suspensão é aquela invocada com a primitiva *x_wait_event*. Nesta suspensão a tarefa só recomeça quando houver um evento, que nesta primeira versão do Executivo é definido como sendo a chegada de uma mensagem ou um estouro de temporização.

3.5.2 As Primitivas do Gerente do Execução

O Gerente de Execução oferece oito primitivas para o gerenciamento da execução de tarefas:

- *x_set_priority*
- *x_get_priority*
- *x_sleep*
- *x_kill*
- *x_lock*
- *x_unlock*
- *x_swap*
- *x_wait_event*

A seguir há uma definição das funções de interface em linguagem C (veja seção 3.4) que as tarefas usam para invocar estas primitivas.

A Primitiva *X_SET_PRIORITY*

Esta primitiva altera dinamicamente a prioridade da tarefa invocadora. O valor inicial da prioridade é dado pelo configurador do PFC. Sua função de interface é

x_set_priority(unsigned new_priority)

O parâmetro *new_priority* é o novo valor da prioridade da tarefa. A prioridade mais alta é de Ø e a mais baixa é de 255. O algorítmo de seleção de tarefa é mais eficiente quando a maioria das tarefas tem prioridades na faixa 1 - 16.

Primitiva *X_GET_PRIORITY*

Esta primitiva devolve à tarefa invocadora a sua prioridade atual. Esta primitiva é usada normalmente na fase de inicialização da tarefa para que a sua prioridade inicial possa ser conhecida. Com o uso da primitiva *x_set_priority* uma tarefa pode aumentar ou diminuir temporariamente sua prioridade de acordo com as suas necessidades dinâmicas. Sua função de interface é a seguinte:

*x_get_priority(unsigned *priority)*

O Executivo usa o apontador *priority* para devolver a prioridade à tarefa.

A Primitiva *X_SLEEP*

Esta primitiva permite a suspensão de uma tarefa por um período específico. O período da suspensão é garantido, não havendo nenhuma maneira para a tarefa ser desbloqueada neste período. Sua função de interface é:

x_sleep(long count)

O tempo de suspensão é especificado em unidades de 20ms no

parâmetro *count*.

A Primitiva *X_SWAP*

Esta primitiva força uma mudança de tarefa, em que a tarefa invocadora efetivamente passa o que resta da sua fatia de tempo para a próxima tarefa. Em efeito, esta primitiva efetua mudança cooperativa de contexto. Sua função de interface é:

x_swap(void)

Não há parâmetros para esta primitiva.

A Primitiva *X_LOCK*

Esta primitiva desativa o mecanismo de mudança preemptiva de tarefa. A tarefa que usa esta primitiva terá acesso exclusivo à UCP. Enquanto a mudança preemptiva estiver desativada, o relógio tempo-real ainda funciona cuidando dos temporizadores e da comunicação PFC/PP. Enquanto a mudança preemptiva estiver desativada, as tarefas ainda podem fazer uma mudança cooperativa usando a primitiva *x_swap*. Sua função de interface é:

x_lock(void)

Não há parâmetros para esta primitiva.

A Primitiva *X_UNLOCK*

Esta primitiva reativa o mecanismo de mudança preemptiva de contexto. Ela é normalmente usada em conjunto com a primitiva x_lock para garantir acesso exclusivo às regiões críticas de memória ou para reativar a mudança preemptiva de contexto. Sua função de interface é:

x_unlock(void)

Não há parâmetros para esta primitiva.

A Primitiva *X_KILL*

Esta primitiva desativa permanentemente a tarefa invocadora. A memória ocupada pela tarefa e de qualquer mensagem pendente é passada para o gerente de memória e torna-se disponível para a alocação dinâmica. Sua função de interface é:

*x_kill(long size, void *task)*

O parâmetro *size* é o tamanho da tarefa em bytes, que inclui as áreas de código, dados e pilha. Para calcular este valor a tarefa utiliza os rótulos *start_code* e *end_stack* declarados como globais no módulo *taskasm.asm* (veja seção 3.9). Estes rótulos definem o começo da área de código e o fim da área da pilha respetivamente. O parâmetro *task* é um apontador para o começo da área de memória ocupada pela tarefa. A tarefa deve usar o rótulo *start_code* para este parâmetro.

A Primitiva *X_WAIT_EVENT*

Esta primitiva suspende a tarefa invocadora até acontecer um evento. Um evento pode ser o estouro de um temporizador ou a chegada de uma mensagem. Na reativação da tarefa o Executivo devolve para a tarefa um código que representa o tipo de evento que aconteceu. Sua função de interface é:

*x_wait_event(int *status)*

O apontador *status* indica para onde o Executivo deve devolver o valor que representa o evento que aconteceu. Dois valores são possíveis:

X_TIMER_EVENT	se há um temporizador estourado e
X_MESSAGE_EVENT	se há uma mensagem.

Estas duas macros são definidas no arquivo *x.h*.

3.6 O Gerente de Comunicação

O gerente de comunicação é responsável pela comunicação inter-tarefa e tarefa/hospedeiro.

3.6.1 O Mecanismo de Comunicação

As comunicações inter-tarefa e tarefa/hospedeiro são feitas com unidades chamadas de mensagens. Uma mensagem é um bloco de memória com a seguinte estrutura:

```
struct message {
        struct msg_hdr;
        struct task_data;
}
```

A estrutura *msg_hdr* é o cabeçalho da mensagem e a sua inclusão é obrigatória, sendo definido no arquivo *x.h*. As suas variáveis são para uso interno do Executivo. O parâmetro *task_data* é uma estrutura que representa os dados que a tarefa quer enviar. No PFC esta estrutura normalmente representaria uma primitiva da interface entre as camadas da arquitetura MAP. A estrutura *msg_hdr* é definida assim:

```
struct msg_hdr {
        unsigned size;
        int task;
        struct msg_hdr *next;
}
```

Estas três variáveis são manipuladas pelo Executivo - a tarefa não precisa inicializá-las. A variável *size* representa o tamanho da mensagem em bytes. A variável *task* é usada para guardar a identidade da tarefa emissora. A variável *next* é usada no encadeamento de mensagens na cadeia de recepção da tarefa receptora.

Para enviar uma mensagem a tarefa usa a primitiva *x_send_message*. Ela passa como parâmetros a identidade da tarefa receptadora, o tamanho da mensagem, e um apontador para uma estrutura do tipo *message* definida anteriormente nesta seção. No caso das tarefas do PFC suas identidades são na faixa de 1 até n, onde n é a última tarefa a ser carregada. Os processos do hospedeiro são identificados com números negativos.

Quando o destino da mensagem é uma outra tarefa, o Executivo instala esta mensagem no fim da cadeia de mensagens da tarefa destino (usando o apontador *next* do cabeçalho). A figura 3.2 mostra o encadeamento de mensagens.

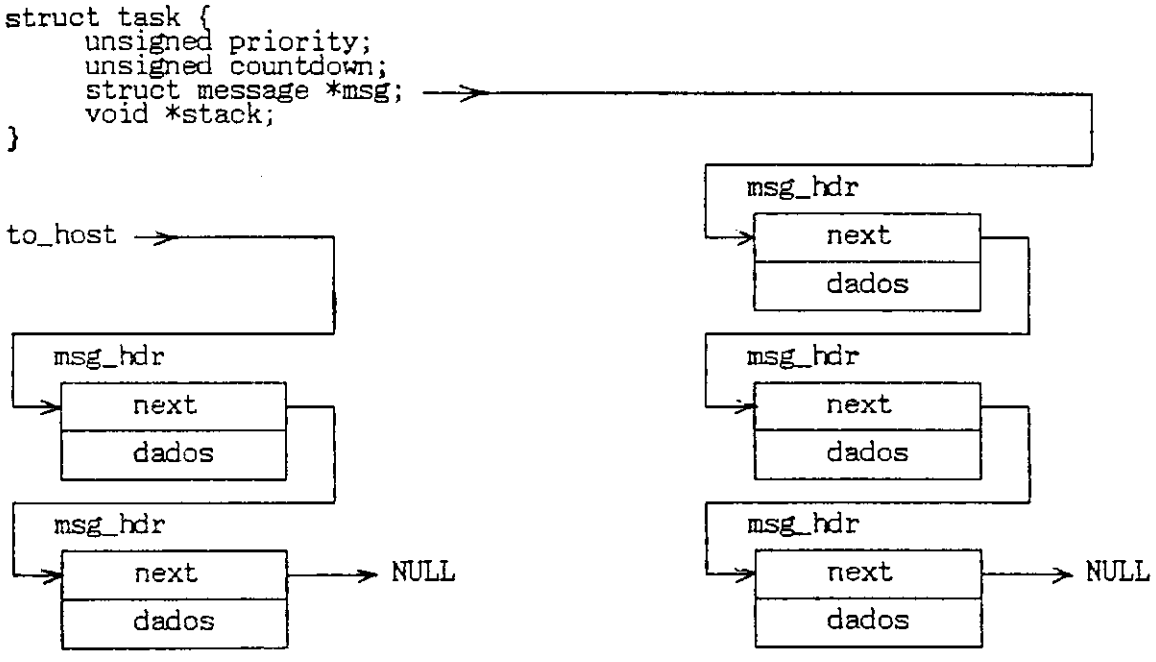


Figura 3.2 - Encadeamento de Mensagens

No caso onde o destino da mensagem é o hospedeiro a mensagem é colocada no fim de uma cadeia cuja início é dado pelo apontador *to_host*. As mensagens nesta cadeia estão esperando sua vez de entrarem no buffer da memória comum para serem recebidas pelo hospedeiro. A rotina do

relógio de tempo real é responsável pela verificação deste buffer a cada 20ms para enviar as mensagens que estão esperando na cadeia e para receber as mensagens vindas do hospedeiro.

Para receber uma mensagem a tarefa usa a primitiva *x_receive_message*. Se houver uma mensagem presente na sua cadeia de recepção o Executivo devolve à tarefa um apontador para esta mensagem. O Executivo tira então esta mensagem do início da cadeia.

O processo número -1 do Hospedeiro é por convenção considerado como o processo "console" por onde são enviados mensagens de estado e de erro. Este processo "console" pode guardar estas mensagens num arquivo ou mostrá-las no monitor.

3.6.2 As Primitivas do Gerente de Comunicação

O Gerente de Comunicação oferece duas primitivas para a comunicação inter-tarefa. Elas são:

- *x_send_message*
- *x_receive_message*

A seguir há uma definição das funções de interface que as tarefas usam para invocar as primitivas.

A Primitiva *X_SEND_MESSAGE*

Esta primitiva envia uma mensagem para uma outra tarefa do PFC ou para um processo do Hospedeiro. Sua função de interface é:

*x_send_message(int task,unsigned size,struct message *msg)*

O parâmetro *task* identifica a tarefa receptora da mensagem. Se for um número positivo isso identifica uma tarefa local do PFC. Se for um número negativo isso identifica um processo do Hospedeiro. O

parâmetro *size* indica o tamanho total da mensagem em bytes. O parâmetro *msg* é um apontador para uma estrutura do tipo *message*. Esta estrutura tem que ter a estrutura *msg_hdr* como cabeçalho (a *msg_hdr* é definida no arquivo *x.h*). O conteúdo deste cabeçalho não tem importância na invocação desta primitiva.

A Primitiva *X_RECEIVE_MESSAGE*

Esta primitiva é usada para receber uma mensagem. Não há suspensão envolvida e no caso onde não houver mensagem na cadeia de recepção da tarefa um código de erro é devolvido imediatamente. Sua função de interface é:

*x_receive_message(int *task,struct message **msg, int *status)*

O Executivo devolve a identidade da tarefa emissora usando o apontador *task*. O endereço da mensagem é devolvido usando o apontador *msg*. O resultado da primitiva é devolvido usando o apontador *status*. Este resultado terá dois valores possíveis:

FAIL se não havia mensagem disponível e

OK se a mensagem foi devolvida.

Estas macros são definidas no arquivo *general.h*.

3.7 O Gerente de Temporização

O Gerente de Temporização é responsável pela criação, leitura e destruição dos temporizadores. O núcleo do Executivo (especificamente a rotina do relógio de tempo-real) é responsável pela atualização dos temporizadores a cada 20ms.

3.7.1 O Mecanismo de Temporização

Um temporizador é basicamente uma estrutura de dados interligada

com outros temporizadores numa cadeia. Cada tarefa tem sua própria cadeia de temporizadores, tendo um apontador *timer_head* (veja seção 3.3) que localiza o primeiro temporizador na cadeia. A estrutura *timer* de um temporizador é a seguinte:

```
struct timer {
      struct timer *next;
      long count;
      long id;
}
```

O apontador *next* indica o próximo temporizador na cadeia. O último temporizador tem o valor nulo em *next*. Os temporizadores são ordenados de maneira tal que a variável *count* representa o período incremental (em unidades de 20ms) deste temporizador em relação ao temporizador anterior. Por exemplo, se três temporizadores forem criados com tempos de 100, 140, e 150 unidades de 20ms, a cadeia resultante seria aquela indicada na figura 3.3.

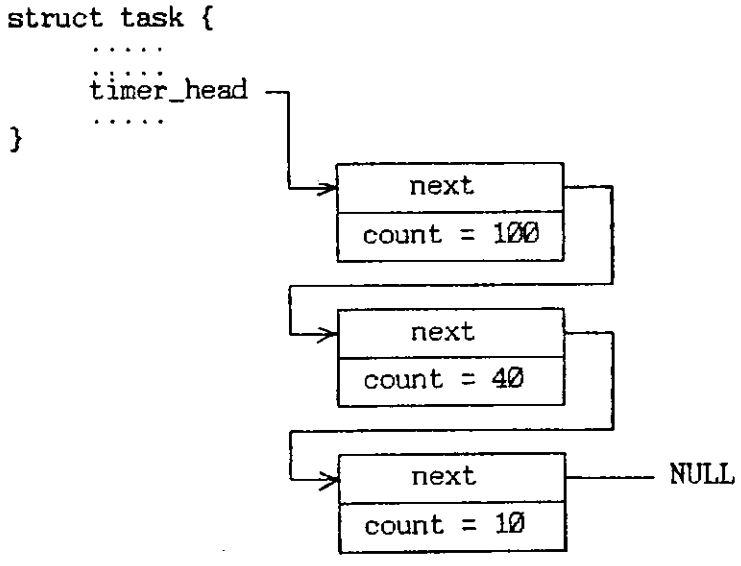


Figura 3.3 - Exemplo de uma Cadeia de Temporizadores

Desta maneira, a atualização dos temporizadores a cada 20ms é limitada a atualização do primeiro temporizador da cadeia, o que torna o processo eficiente.

A variável *id* da estrutura *timer* é um valor de identificação deste temporizador usado pela tarefa. Este valor é passado pela tarefa na criação do temporizador. Sendo um tipo "long" ele pode igualmente representar um valor inteiro ou um apontador (nos modelos "compact" e "large" dos compiladores para a família 8086 os apontadores são de quatro bytes).

A criação de um temporizador é feita pela tarefa com a primitiva *x_set_timer*. Na criação o Executivo percorre a cadeia até chegar ao ponto que corresponde ao tempo de estouro do novo temporizador. Ele então usa o Gerente de Memória para alocar uma pequena quantidade de memória para conter uma nova estrutura do tipo *timer*. Esta estrutura è inserida na cadeia de temporizadores neste ponto.

A eliminação de um temporizador é feita com a primitiva *x_kill_timer*. A tarefa passa o valor de identificação (a variável *id*) para o Executivo que o usa para percorrer a lista de temporizadores até chegar nele. Este temporizador é removido da cadeia e sua memória devolvida ao gerente de memória.

A leitura de um temporizador é feita com a primitiva *x_read_timer*. Usando o valor de *id* passado pela tarefa, o Executivo percorre a cadeia de temporizadores, até encontrar aquele com um valor de identidade igual a *id*. No processo, cada temporizador que é passado tem seu valor *count* adicionado à uma soma. No fim da varredura, esta soma é devolvida à tarefa.

A atualização dos temporizadores de todas as tarefas é feita pelo núcleo do Executivo na sua rotina de relógio de tempo-real. Para cada tarefa a rotina decrementa o valor de *count* do primeiro temporizador na cadeia. Se este valor chega ao zero então este temporizador é removido

da cadeia e inserido na cadeia de temporizadores estourados. O início desta cadeia é dado pelo apontador *timeout_head* na estrutura *task*.

A obtenção de temporizadores estourados é feita com a primitiva *x_get_timer*. O Executivo tira o temporizador mais antigo da cadeia de temporizadores estourados (usando o apontador *timeout_head*) e devolve seu valor de *id* à tarefa. A memória que o temporizador ocupou é devolvida para uma reserva de temporizadores inativos que o Executivo mantém para não ter que alocar e desalocar blocos de memória para cada temporizador. Se esta reserva já estiver cheia então a memória é realmente devolvida ao Gerente de Memória.

3.7.2 As Primitivas do Gerente de Temporização

O Gerente de Temporização oferece cinco primitivas para a criação, leitura e destruição de temporizadores. Estas primitivas são:

- *x_set_timer*
- *x_get_timer*
- *x_read_timer*
- *x_modify_timer*
- *x_kill_timer*

A seguir há uma definição das funções de interface que as tarefas usam para invocar esta primitivas.

A Primitiva *X_SET_TIMER*

Esta primitiva cria um novo temporizador. Sua função de interface é:

*x_set_timer(long count, long id, int *status)*

O parâmetro *count* é o período de temporização em unidades de 20ms.

O parâmetro *id* é um identificador pelo qual o temporizador vai ser

reconhecido pela tarefa criadora. Observe-se que o uso do tipo "long" permite a passagem de um valor inteiro ou de um apontador (usando "type casting" da linguagem C). O Executivo devolve o resultado da primitiva usando o apontador *status*, que pode assumir os seguintes valores:

OK se o temporizador for criado e

FAIL se o temporizador não for criado.

Estas macros são definidas no arquivo *general.h*.

A Primitiva *X_GET_TIMER*

Esta primitiva obtém o mais antigo dos temporizadores estourados da tarefa invocadora. Sua função de interface é:

*x_get_timer(void *id, int *status)*

O Executivo devolve o resultado da primitiva para o inteiro apontado pelo *status*, que pode assumir os seguintes valores:

OK se havia temporizador estourado e

FAIL se não havia temporizador estourado.

Se houver um temporizador estourado o seu valor de identificação é devolvido usando o apontador *id*.

A Primitiva *X_READ_TIMER*

Esta primitiva devolve o tempo que ainda resta para o estouro de um temporizador. Sua função de interface é:

*x_read_timer(long id, long *count)*

O Executivo procura na cadeia de temporizadores pelo temporizador com a identidade igual ao *id*. O período que falta para o estouro da temporização é devolvido usando o apontador *count*.

A Primitiva *X_MODIFY_TIMER*

Esta primitiva modifica o período de um temporizador já

existente. Sua função de interface é:

x_modify_timer(long id, long adjust)

O parâmetro *id* identifica o temporizador a ser alterado. O parâmetro *adjust* é o período em unidades de 20ms que deve ser adicionado ao período do temporizador. O seu valor pode ser positivo ou negativo.

A Primitiva *X_KILL_TIMER*

Esta primitiva elimina um temporizador da tarefa invocadora. Este temporizador pode ser um dos ativos que não estouraram, ou um dos estourados esperando a invocação da primitiva *x_get_timer*. Sua função de interface é:

x_kill_timer(long id)

O parâmetro *id* identifica o temporizador a ser eliminado.

3.8 O Gerente de Memória

O Gerente de Memória é responsável pelo mecanismo de alocação e devolução de blocos de memória. Há uma área reservada da memória especificamente para alocação dinâmica que é a área entre a última tarefa e o fim da memória. Esta área varia em tamanho de acordo com a quantidade de tarefas instaladas e a quantidade de memória presente.

3.8.1 O Mecanismo de Alocação de Memória

Todos os blocos livres de memória são interligados numa cadeia. A ordem na cadeia corresponde a ordem física dos blocos. No começo de cada bloco livre há uma estrutura *mem_block* da seguinte forma:

```
struct mem_block {
        struct mem_block *next;
        unsigned size;
}
```

O apontador *next* localiza o próximo bloco livre na cadeia e a variável *size* indica o tamanho do bloco. Para indicar o primeiro bloco na cadeia o Executivo utiliza o apontador *free_mem_head*. A figura 3.4 mostra uma cadeia de três blocos livres.

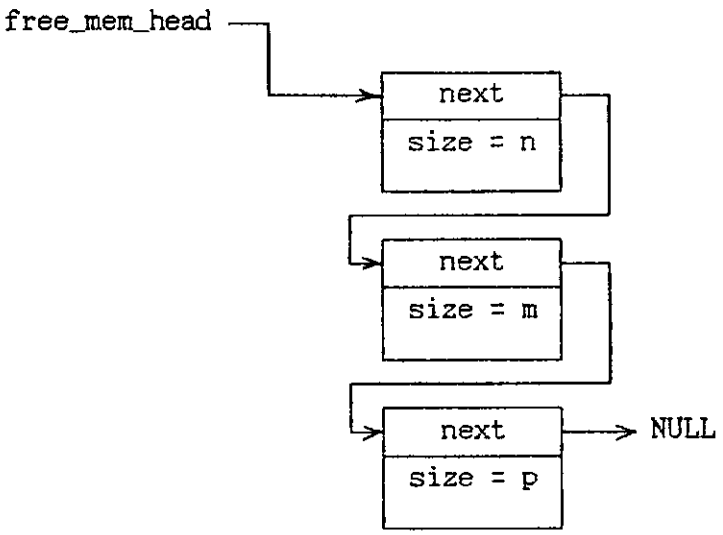


Figura 3.4 - Uma Cadeia de Blocos Livres de Memória

Devido à importância da eficiência na alocação e liberação de memória ela é granularizada, sendo alocada em múltiplos fixos de 16, 32, 64, 128 ou 256 bytes. O valor especifico usado é determinado pelo configurador do PFC durante o processo de inicialização (seção 3.9.3). Estes valores permitem que todos os cálculos feitos durante a alocação de memória sejam baseados em inteiros de 16 e não de 32 bits.

Uma granularização fina (de 16 ou 32 bytes) reduz o desperdício interno de memória e é recomendada quando a quantidade de memória disponível para alocação é limitada.

Uma granularização grossa (de 128 ou 256 bytes) reduz a fragmentação da memória permitindo maior rapidez na sua alocação mas devido ao desperdício extra de memória é recomendada apenas quando o PFC dispõe de ampla memória.

O mecanismo de alocação procura ser eficiente e simples. A tarefa usa a primitiva *x_get_memory* para pedir uma quantidade *x* de bytes. O Executivo transforma este valor *x* em *y = (x/block_size)+1* parágrafos (cada parágrafo é de 16 a 256 bytes). O *block_size* é o valor usado para a granularização da memória. O Executivo depois percorre a cadeia de blocos livres até achar uma área que tem um valor da variável *size* igual ou maior a *y*. Isso é uma aplicação da técnica `primeiro encaixe´ ("first fit") [HOR 82] que na prática oferece um melhor desempenho geral em comparação com as técnicas `próxima encaixe´ ("next fit") e `melhor encaixe´ ("best fit"). O Executivo então aloca os últimos *y* parágrafos deste bloco para a tarefa. Em seguida ele reduz o tamanho deste bloco. Se o bloco for exatamente do tamanho solicitado, então ele é eliminado da cadeia de blocos livres.

Para devolver um bloco a tarefa usa a primitiva *x_return_memory*. O Executivo percorre a cadeia até encontrar o ponto de inserção (os blocos são encadeados em ordem física). Uma estrutura *mem_block* é colocada no início do novo bloco livre e o bloco é inserido na cadeia. O Executivo examina os dois blocos vizinhos para verificar se pode haver uma fusão de blocos.

No processo de devolução uma mensagem de erro é gerada se o bloco devolvido contém uma área já considerada pelo Executivo como sendo livre. Esta mensagem é enviada ao processo padrão de erro do hospedeiro (processo -1).

3.8.2 As Primitivas do Gerente de Memória

Tem-se duas primitivas que as tarefas podem utilizar para alocar de devolver blocos de memória. Estas são:

- *x_get_memory*
- *x_return_memory*

A seguir há uma definição das funções de interface para estas primitivas.

A Primitiva *X_GET_MEMORY*

Esta primitiva é usada para obter um bloco de memória. Sua função de interface é:

*x_get_memory(unsigned count,void **mem,int *status)*

O parâmetro *count* é o número de bytes solicitados. Ele pode ser de valor qualquer entre 1 byte e 64 kbytes. O parâmetro *mem* é um apontador para um apontador no qual o Executivo devolve o endereço do começo do bloco alocado. O Executivo usa o parâmetro *status* para devolver o resultado de execução da primitiva. O seu valor pode ser:

FAIL se não havia memória suficiente.

OK se a memória foi alocada.

Estas macros são definidas no arquivo *general.h*.

A Primitiva *X_RETURN_MEMORY*

Esta primitiva é usada para devolver um bloco de memória ao Gerente de Memória do Executivo. Sua função de invocação é:

*x_return_memory(unsigned count,void *mem)*

O parâmetro *count* é o tamanho em bytes do bloco a ser devolvido e pode assumir o valor de 1 até 64k. O parâmetro *mem* é um apontador para o começo do bloco a ser devolvido. Se o bloco devolvido se sobrepõe a um bloco já considerado livre uma mensagem de erro é gerada.

3.9 A Geração e Implantação das Tarefas

Esta seção descreve o processo da geração e implantação de tarefas no PFC. O código do Executivo e das tarefas é contido em arquivos do tipo EXEC do MS-DOS. Estes arquivos são gerados pelo processo normal de compilação no ambiente MS-DOS. O Executivo e a tarefa LLC/MAC deste trabalho foram escritos na linguagem C que é uma linguagem que permite a criação de código portável. O compilador C5.1 da Microsoft foi adotado porque ele é um compilador `conceituado' entre projetistas de sistemas e serve como padrão de referência para os demais compiladores C do ambiente MS-DOS, além do que este compilador tem uma biblioteca rica e um bom otimizador.

3.9.1 O Formato da Tarefa

Cada módulo do código fonte de uma tarefa que usa as primitivas do Executivo deve incluir o arquivo *x.h*. Este arquivo define as funções de interface das primitivas e algumas macros.

Ao contrário de um programa EXEC do MS-DOS, não é permitido o término de uma tarefa através da saída da sua função *main*. Um laço infinito tem que existir dentro da tarefa para que isso não aconteça.

Estes dois requisitos são exemplificados a seguir. Esta tarefa simples muda o estado de uma lâmpada LED (decodificado no endereço de saída Øx8Ø) a cada segundo. O seu código fonte é:

```
#include "x.h"
void outpw(unsigned,unsigned);
main (void)
{
      for(;;) {
            x_sleep(50);
            outpw(0x80,0);
      }
}
```

A função *outpw* é da biblioteca Microsoft C5.1 e sua inclusão é automática quando o compilador pesquisa a biblioteca padrão.

3.9.2 A Geração das Tarefas

A compilação e `link-edição' do código da tarefa é feita da mesma maneira que um programa normal no MS-DOS. Um exemplo da geração de uma tarefa é o seguinte:

```
masm taskasm;
cl /Gs /Od /c task.c
link taskasm task;
```

O arquivo *taskasm.asm* contem o código em assembler que define a alocação da pilha, o código de inicialização da tarefa e as rotinas de interface ao Executivo. Este código de inicialização substitui o código normalmente inserido pelo compilador que só serve para o ambiente MS-DOS.

A opção /Gs do compilador inibe a invocação da sonda de pilha ("stack probe") que normalmante verifica se a pilha da tarefa está dentro dos seus limites. Se a opção /Gs não for usada o projetista da tarefa terá que escrever sua própria rotina em assembler para testar a pilha.

A opção /Od do compilador inibe a otimização feita pelo compilador. A inclusão da otimização tem que ser feita com cuidado devido principalmente às peculiaridades do otimizador do compilador C5.1 da Microsoft.

3.9.3 A Implantação das Tarefas

A inicialização do PFC é feita através de um programa carregador chamado *loader*. Para este trabalho o *loader* foi implementado para o ambiente MS-DOS, mas nada impede que um equivalente seja escrito para outros ambientes. O fato de ser escrito em C5.1 da Microsoft ajuda na sua portabilidade (por exemplo, este compilador está disponível para XENIX).

O programa *loader* obtém a configuração do PFC num arquivo *config.sys* que deve ser preparado individualmente para cada sistema. Este arquivo é do tipo ASCII e contem os seguintes campos:

```
%slot slot_number
%mem mem_size wait_states block_size
%task name_1 priority_1
%task name_2 priority_2
  :     :         :
%task name_n priority_n
```

A linha *%slot* define o endereço alocado ao PFC em termos do número do espaço reservado para cartões, que pode ser de Ø a 7. O parâmetro dado tem que corresponder às chaves de seleção de endereço do hardware do PFC (chaves S1, S2 e S3).

A linha *%mem* define a quantidade, velocidade e 'granularização' de memória no PFC. O parâmetro *mem_size* é a quantidade de memória em kilobytes e pode assumir os valores de 64, 128, 256 ou 512. O parâmetro

wait_state define o número de estados de espera a ser usado no acesso a esta memória. O número dado é relacionado com a velocidade da memória conforme a tabela 3.1. O parâmetro *block_size* determina o tamanho das parágrafos de memória alocada dinamicamente pelo Gerente de Memória e pode ter valores de 16, 32, 64, 128 ou 256 bytes.

Valor de "wait_state"	Tipo de memória (ns)
Ø	1ØØns
1	12Øns
2	15Øns
3	2ØØns

Tabela 3.1 - Estados de Espera

As linhas *%task* definem quais tarefas serão carregadas. Os parâmetros *name* são os nomes dos arquivos do tipo EXEC (que contém o código das tarefas). As prioridades iniciais da tarefa são dadas pelos parâmetros *priority*, que podem ter valores de Ø a 256. Para fins de carregamento o Executivo é considerado como uma tarefa normal. Nada impede que o PFC seja usado somente com uma tarefa de teste sem o Executivo (por exemplo para fins de teste do hardware). Se o Executivo for usado ele deve ser a primeira tarefa a ser especificada no arquivo *config.pfc*. Um exemplo de um arquivo *config.pfc* é:

```
%slot Ø
%mem 128 1 32
%task exec 128
%task llc 2
%task mms 4
```

Este exemplo seria típico de uma configuração mini-MAP, tendo as duas tarefas MMS e LLC. O configurador do PFC pode impor vários valores de

prioridade de tarefa numa tentativa de aumentar o desempenho global do sistema.

O programa *loader* utiliza o mecanismo de janelas da memória do PFC (seção 2.11), com qual ele pode ter acesso a toda a memória do PFC. Para cada linha *%task* no arquivo *config.pfc* ele carrega o código do arquivo EXEC correspondente na memória. Usando a tabela de relocação do mesmo arquivo ele depois corrige as entradas relocáveis deste código. A instalação da primeira tarefa começa no endereço 00400H. A figura 3.5 mostra o mapa de memória do PFC depois do carregamento das tarefas EXEC, LLC e MMS do exemplo anterior. Neste exemplo o PFC é configurado com 512 kbytes de memória (00000H - 7FFFFH).

A parte final da memória fisicamente presente (07FFF0H - 07FFFFH) é usada para o código de partida do 80186. Devido à decodificação da memória este código aparece também nos endereço 0FFFF0H - 0FFFFFH onde o 80186 o executa. Note-se que a parte de memória entre a última tarefa e o fim da memória é usada para alocação dinâmica.

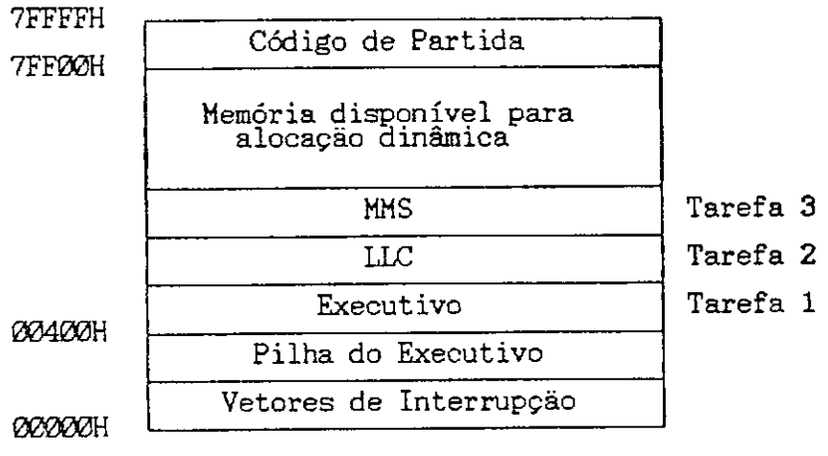


Figura 3.5 O Mapa da Memória do PFC

4. A CAMADA MAC

Este capítulo descreve a implementação da camada MAC do PFC. No início há uma descrição das características principais da MAC, seguida de uma apresentação das primitivas que a MAC oferece à camada LLC e à Gerência de Estação. Em seguida é feita uma análise da forma mais adequada para sua realização levando em conta os recursos do PFC. Para concluir é feito um detalhamento da implementação em si com uma descrição do código e das estruturas de dados utilizadas.

4.1 A Camada MAC 802.4

O controlador de rede MC68824 [MOT 87A] usado neste projeto realiza as funções e mecanismos das especificações IEEE 802.4 e 802.2 [IEEE 88A] que são mais críticas em tempo. Estas incluem:

- o mecanismo de passagem da ficha;
- a temporização das filas de transmissão;
- o mecanismo de entrada e saída do anel e
- o envio de reconhecimentos.

A funcionalidade da MAC executada neste trabalho corresponde ao software que implementa as primitivas de serviço e ao software que gerencia as filas de recepção e transmissão.

A camada MAC fornece dois grupos de primitivas. O primeiro grupo, formado por primitivas responsáveis pela interface entre a camada MAC e a camada LLC, fornece um serviço de transferência de unidades de dados, ponto-a-ponto ou multiponto, sem estabelecimento de conexões. Estas primitivas são:

- MA_UNITDATA.request;
- MA_UNITDATA_STATUS.indication e
- MA_UNITDATA.indication.

O segundo grupo contém as primitivas responsáveis pela interface entre a camada MAC e a Gerência de Estação [ARA 86]. Estas primitivas são:

- MA_INITIALIZE_PROTOCOL.request;
- MA_INITIALIZE_PROTOCOL.confirmation;
- MA_SET_TIMER_LIMIT.request;
- MA_DESIRED_RING_MEMBERSHIP.request;
- MA_DUPLICATE_ADDRESS_DETECTED.indication e
- MA_GROUP_ADDRESS.request.

Na descrição das primitivas que segue, a sua apresentação na forma de funções da linguagem C é arbitrária, usada apenas para mostrar os 'parâmetros' da primitiva numa forma clara. A especificação IEEE 802.4 não impõe nenhuma forma específica de implementação. Assim sendo, a forma das primitivas poderia ser totalmente diferente conforme o critério do implementador.

4.1.1 As Primitivas da Interface MAC/LLC

As três primitivas da interface MAC/LLC compõem o serviço para a transferência de unidades chamadas de MSDUs ("Media access control

Service Data Unit") entre as camadas pares MAC da rede. Estas MSDUs são os quadros ("frames") da rede MAP. O seu formato é o seguinte:

Onde:

Preâmbulo = Seqüência de bits usada para sincronizar o relógio do modem (tamanho configurável)

SD = "Start Delimiter" - Delimitador inicial (1 byte)

FC = "Frame Control" - Controle de quadro (1 byte)

DA = "Destination Address" - Endereço de destino (6 bytes)

SA = "Source address" - Endereço de origem (6 bytes)

Dados = Informação (Ø até 8 kbytes)

FCS = "Frame Check Sequence" - Sequência de verificação de quadro (4 bytes)

ED = "End delimiter" - Delimitador final (1 byte)

A forma de interação das primitivas da camada MAC é mostrada no diagrama seqüencial de tempo da figura 4.1.

A primitiva *MA_UNITDATA.request* é invocada pela camada LLC para enviar dados para uma camada LLC par. A sinopse desta primitiva é a seguinte:

ma_unitdata_request(dest_addr,data,priority,class)

O parâmetro *dest_addr* é o endereço MAC da estação de destino. Este endereço pode ser de uma única estação ou de um grupo de estações. O parâmetro *data* representa os dados a serem transferidos. O parâmetro *priority* é a prioridade MAC a ser usada (de Ø a 7) e o parâmetro *class*

representa a classe de transferência a ser usada e correspondem às opçöes RWR ("Request With Response") e NRWR ("Non Request With Response") da 802.4. A opçäo RWR é usada pela LLC tipo 3 da MAP/EPA, fornecendo um mecanismo de envio de quadro com a recepçäo de reconhecimento. A opçäo NRWR é usada pela LLC tipo 1 do MAP que näo utiliza reconhecimento.

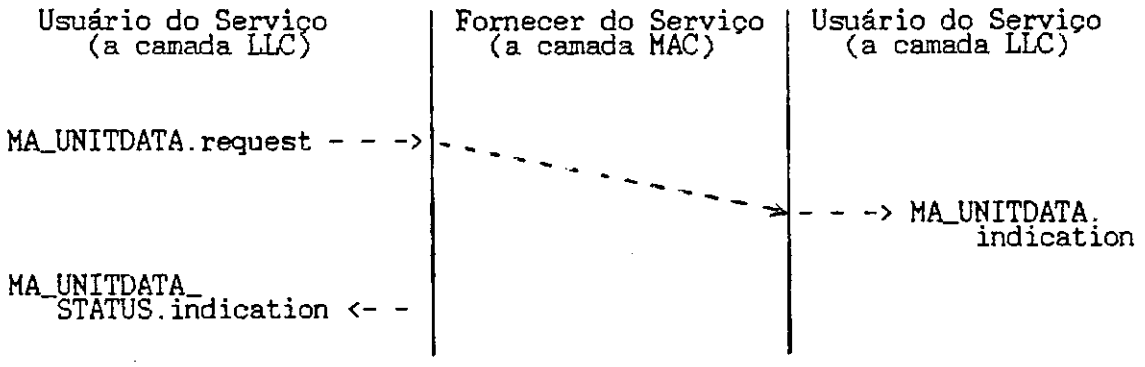


Figura 4.1 As Primitivas da Interface MAC/LLC

A primitiva *MA_UNITDATA_STATUS.indication* é usada para avisar a camada LLC do resultado de uma primitiva *MA_UNITDATA.request* feita anteriormente. A sinopse da primitiva é a seguinte:

ma_unitdata_status_indication(dest_addr,tx_status,priority,class)

O parâmetro *tx_status* ("transmission status") indica o sucesso da transmissäo do quadro passado anteriormente.

A primitiva *MA_UNITDATA.indication* é usada para avisar a camada LLC do recebimento de dados de uma camada par de uma outra estaçäo. A sinopse da primitiva é a seguinte:

ma_unitdata_indication(src_addr,data,rx_status,priority,class)

O parâmetro *src_addr* indica o endereço MAC da estação de origem. O parâmetro *rx_status* ("reception status") indica o estado da recepção.

4.1.2 As Primitivas da Interface MAC/Gerência de Estação

A primitiva *MA_INITIALIZE_PROTOCOL.request* é passada para a camada MAC para configurar o endereço MAC da estação. A primitiva *MA_INITIALIZE_PROTOCOL.confirmation* é passada para o Gerência de Estação para indicar o sucesso de uma primitiva *MA_INITIALIZE_PROTOCOL.request* feita anteriormente. A primitiva *MA_SET_TIMER_LIMIT.request* é passada para a camada MAC para configurar os parâmetros de temporização (por exemplo, o tempo de retenção de ficha). A primitiva *MA_DESIRED_RING_MEMBERSHIP.request* é passada para a camada MAC para pedir a entrada ou saída da estação do anel lógico de estações no qual circula a ficha. A primitiva *MA_DUPLICATE-ADDRESS_DETECTED.indication* é passada para a Gerência de Estação para indicar a detecção de uma situação de duplicação de endereços. A primitiva *MA_GROUP_ADDRESS.request* é passada para a MAC para especificar o conjunto de endereços de grupo a ser reconhecido pela estação.

4.2 A Forma de Implementação da Camada MAC

Esta seção analisa a forma mais apropriada de realizar a camada MAC levando em conta o ambiente do PFC. Devido ao ambiente multi-tarefa do PFC e à interface bem definida na especificação entre a camada MAC e a camada LLC, a primeira possibilidade de implementação que se apresenta é aquela onde a MAC é realizada por uma tarefa independente. Neste caso, a MAC utilizaria a comunicação inter-tarefa do Executivo para efetuar a passagem de primitivas.

Esta forma de implementação é elegante mas por duas razões não foi adotada. A primeira razão é a ineficiência deste método. O uso de mensagens para implementar as primitivas implica numa sobrecarga alta, mesmo com a forma eficiente de comunicação implementada no Executivo. A solução neste caso seria o uso de uma interface baseada em memória compartilhada.

A segunda razão é devida aos controladores de rede. A tecnologia de integração de circuitos tem se desenvolvido a um ponto em que os controladores de rede implementam não somente funções da MAC mas também algumas funções da camada LLC (por exemplo, o TBC se responsabiliza para o envio dos quadros de reconhecimento). Devido a forma complexa em que o TBC controlado se torna imprático ter duas entidades separadas controlando o TBC, e resulta numa fusão das duas camadas numa tarefa só.

Na realidade, na medida que a tecnologia evolui, esta capacidade dos controladores deve se estender até o ponto em que o controlador possa implementar em silício as funções de várias camadas.

A forma de implementação usada é aquela onde as primitivas da MAC são representadas como subrotinas a serem chamadas pelo código da LLC. Assim, a MAC e a LLC ficam integradas numa mesma tarefa. A figura 4.2 mostra a interface mini-MAP no ambiente do PFC.

Como não há uma tarefa MAC a Gerência de Estação envia suas primitivas de gerenciamento da MAC à tarefa LLC. Na figura 4.2 a Gerência de Estação é mostrada como um processo do hospedeiro mas, alternativamente, pode ser parte de uma aplicação ou de uma tarefa do PFC.

4.3 A Implementação da MAC

Esta seção detalha as funções da linguagem C que representam as três primitivas da interface LLC/MAC. Porém, antes de detalhar a

implementação em si, faz-se necessário uma abordagem das estruturas de inicialização do TBC e da maneira na qual os quadros são representados pelo TBC na memória.

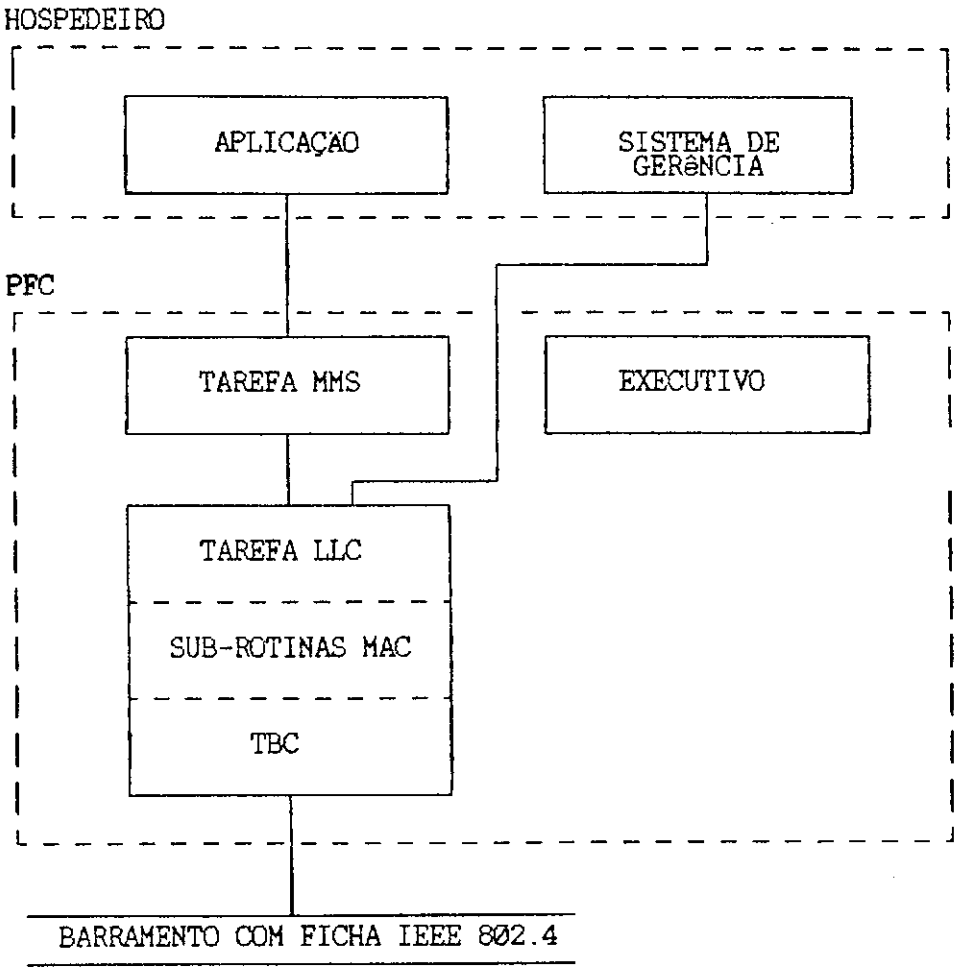


Figura 4.2 - A Interface mini-MAP no ambiente do PFC

4.3.1 A Tabela de Inicialização

O processo de inicialização do TBC envolve duas estruturas de dados: a tabela de inicialização ("initialization table)" e a área privada ("private area"). Usando os valores contidos nas primitivas que

vêm da Gerência de Estação, o microprocessador da interface preenche a tabela de inicialização com todos os parâmetros de configuração do TBC e de operação da MAC. Através da invocação de um comando do TBC (veja a seção a seguir) , o microprocessador passa esta tabela para o TBC, que copia algumas partes da tabela para a área privada á qual somente ele tem acesso.

4.3.2 Os Comandos do TBC

A manipulação do TBC é feita através de 29 comandos. Nesta implementação cada comando do TBC é representado por uma função específica em linguagem C. Os detalhes de cada comando variam, mas o procedimento é normalmente o seguinte:

- zera-se a palavra de confirmação da tabela de inicialização;
- instala-se um parâmetro numa área específica da tabela de inicialização;
- escreve-se o número do comando no registrador de comando do TBC;
- suspende-se a tarefa por um pequeno período e
- examina-se a palavra de confirmação da tabela de inicialização que retorna um código de resultado (sucesso ou insucesso).

Todas as funções que realizam comandos do TBC são nomeadas de forma semelhante visando sua identificação funcional ("tbc_xxx") - por exemplo, *tbc_start()*, *tbc_initialize()* ou *tbc_offline()*.

4.3.3 Os Descritores de Quadro e de Buffer

O TBC usa uma combinação de dois tipos de estrutura de dados para representar os quadros. São eles: os descritores de quadro FD ("Frame

Descriptor") e os descritores de buffer BD ("Buffer Descriptor"). Uma parte do software que gerencia o TBC é aquela que prepara estas estruturas para a transmissão e as desvincula no caso de recepção.

Para cada quadro existe um descritor de quadro FD. O FD contém dados do tipo:

- o endereço MAC de destino;
- o endereço MAC de origem;
- o tamanho dos dados e
- um apontador para o primeiro BD.

Para cada FD há um ou mais BDs encadeados. O FD contém um apontador para o primeiro BD da cadeia. Para cada BD há um buffer de dados. O BD contém dados do tipo:

- um apontador para o buffer de dados;
- o tamanho do buffer;
- o deslocamento ("offset") dos dados a partir do começo do buffer e
- um apontador para o próximo BD na cadeia.

Um exemplo da representação de um quadro por FDs e BDs é mostrado na figura 4.3, onde os BDs e o FD são representados numa forma bastante simplificada.

4.3.4 A Função "*ma_unitdata_req*"

Esta função implementa a primitiva de serviço *MA_UNITDATA.request* da MAC, ela é chamada quando a LLC quer enviar uma PDU para uma outra estação, e é assim definida:

*void ma_unitdata_req(fd *fd,unsigned priority)*

O parâmetro *fd* é um apontador para um descritor de quadro FD que

representa o quadro a ser enviado. Este FD é totalmente preparado pelo software da camada LLC. O parâmetro *priority* é a prioridade MAC a ser usada (Ø - 7).

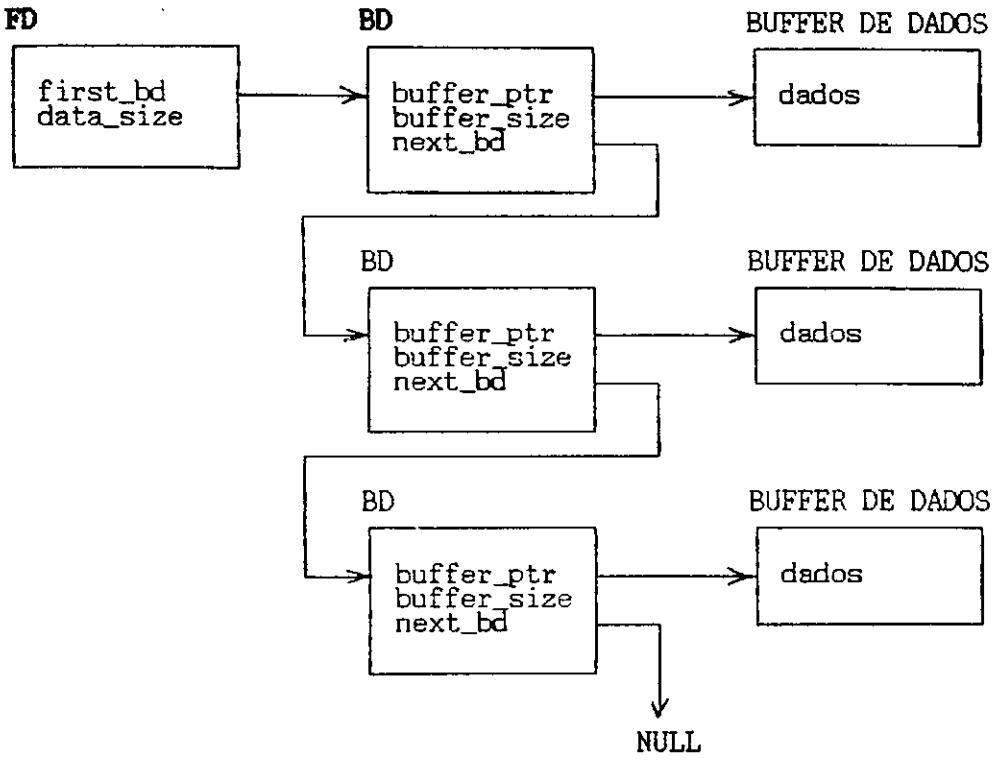


Figura 4.3 - A Representação de um Quadro

Esta função instala o FD em uma das quatro filas de transmissão do TBC. Cada fila representa duas prioridades MAC conforme a tabela 4.1. Para cada fila de transmissão o software mantem dois apontadores *tx_tail* e *tx_head* que apontam, respectivamente, para o fim e o começo de uma cadeia de FDs. Esta função instala novos FDs a serem transmitidos no fim da cadeia usando o apontador *tx_tail*. O TBC processa os FDs no começo da fila mas no lugar de utilizar os apontadores *tx_head* ele utiliza apontadores próprios. Cada vez que FDs são inseridos numa fila vazia esta função invoca o comando "start" do TBC (usando a função *tbc_start*), que força o TBC a recomeçar a transmissão de uma fila.

Prioridade MAC	Fila
Ø,1	Ø
2,3	1
3,4	2
5,6	3

Tabela 4.1 - A Relação entre a Prioridade MAC e a Fila do TBC

O algorítmo da função *ma_unitdata_request* é:

```
void ma_unitdata_req(FD *fd, int priority)
{
      int q;

      /* calcula fila a ser usada */
      q = priority / 2;

      /* se fila estiver vazia ... */
      if (tx_tail[q] == NULL) {

            /* coloca FD como único elemento na fila
               e manda TBC recomeçar fila */
            tx_head[q] = fd;
            tbc_start(q,fd);
      }

      /* ... a fila não está vazia */
      else {
            /* coloca quadro no fim de fila */
            tx_tail[q]->next_fd = fd;

            /* se quadro anterior já confirmado ... */
            if (tx_tail[q]->conf_ind & CFD)
                  tbc_start(q,fd);
      }
}
```

A função examina se a fila de transmissão correspondente à prioridade

MAC está ou não vazia. Se estiver vazia ela coloca o FD como elemento único da fila e chama a função *tbc_start* que invoca o comando "start" do TBC.

Se a fila não estiver vazia então esta função coloca o FD no fim da fila. Existe a possibilidade do TBC terminar de transmitir os FDs anteriores nesta cadeia antes do termino da instalação do novo FD. Por isso, a função examina o FD anterior da cadeia para verificar se o TBC já o confirmou. Caso tenha confirmado então do ponto de vista do TBC a fila está vazia. Neste caso, a função invoca a função *tbc_start*, garantindo assim a transmissão do FD que acabou de ser instalado.

4.3.5 A Função *"ma_unitdata_status"*

Esta função implementa a primitiva de serviço *MA_UNITDATA_STATUS.indication* da MAC. Ela é chamada pela LLC para testar a confirmação, pelo TBC, dos quadros nas filas de transmissão. Estes quadros foram instalados anteriormente com a primitiva *MA_UNITDATA.request*. O primeiro quadro encontrado, com sua transmissão confirmada pelo TBC, é devolvido à função invocadora. A função é definida assim:

```
int ma_unitdata_status(int q, fd **fd)
```

O parâmetro *q* indica a fila de transmissão a ser testada. O parâmetro *fd* é um apontador para um apontador de FD (indireção dupla). A função acima usa este apontador para devolver o endereço de um FD confirmado, retornando "OK" se o quadro for confirmado e "FAIL" caso contrário (estas macros são definidas no arquivo *x.h*). O algoritmo desta função é o seguinte:

```
int ma_unitdata_status(int q, FD **result)
{
      FD *fd;

      /* retorna se fila vazia */
      if (tx_tail[q] == NULL)
            return FAIL;

      /* retorna se TBC ainda não confirmou o FD mais antigo */
      fd = tx_head[q];
      if (fd->conf_ind & CFD == Ø)
            return FAIL;

      /* se há quadro depois deste ...*/
      if (fd->cntl_next_fd & NPV == NPV) {

            /* se próximo quadro não confirmado ... */
            if (fd->next_fd->conf_ind & CFD == CFD)

                  /* se este quadro indica que fila está vazia ... */
                  if (fd->conf_ind & EMP == EMP)
                        tbc_start(q,fd->fd_next;

            /* atualiza cabeça da fila */
            tx_head[q] = fd->next_fd;
      }

      /* ... não há quadros depois deste */
      else {
            tx_tail[q] = NULL;
            tx_head[q] = NULL;
      }

      /* retorna quadro confirmado */
      *result = fd;
      return OK;
}
```

A função testa a fila para ver se ela está vazia e, caso esteja, é devolvido um "FAIL". Em seguida, a função testa se o TBC já confirmou o quadro mais antigo da fila (usando o apontador em *tx_head*). Se não houve confirmação, a função devolve "FAIL".

Se houve confirmação então há um quadro confirmado nesta fila. Existe uma situação onde o TBC termina a transmissão de uma fila enquanto a função *ma_unitdata_req* está adicionando um novo quadro. Neste caso, é possível que o TBC considere a fila como sendo vazia sem tomar

conhecimento do novo quadro. Para detectar esta condição a função examina o próximo quadro da fila (se existir). Se este próximo quadro não foi confirmado e o quadro inicial indica que a fila esta vazia (através da máscara EMP) então a função invoca o comando *tbc_start* do TBC para recomeçar a transmissão da fila.

No caso onde não existe um próximo quadro, a função marca a fila como sendo vazia através da anulação dos apontadores *tx_head* e *tx_tail*. Finalmente a função devolve o endereço do FD do quadro confirmado e informa à função invocadora do sucesso da chamada devolvendo o estado "OK".

4.3.6 A Função *"ma_unitdata_ind"*

Esta função implementa a primitiva de serviço *MA_UNITDATA.indication* da MAC. Esta função é chamada pela LLC para testar a chegada de um quadro numa fila de recepção específica. O primeiro FD encontrado na fila é devolvido à LLC. A função é definida assim:

*int ma_unitdata_request(int q, FD **fd)*

O parâmetro *q* indica a fila a ser examinada. O parâmetro *fd* é um apontador para um apontador de FD (dupla indireção). Se um quadro for encontrado, a função usa este apontador para devolver seu endereço. São retornados os valores "OK" se chegou um quadro ou "FAIL" caso contrário.

Ao contrário das filas de transmissão, as filas de recepção do TBC nunca podem ficar vazias pois o TBC necessita do último FD em cada fila para continuar o encadeamento dos quadros recebidos. Para atender este requisito de operação, cada fila de recepção dispõe dos apontadores *rx_head* e *rx_dummy*. O *rx_head* marca o primeiro FD na fila de quadros

recebidos. O *rx_dummy*, quando válido, marca um único FD que é uma copia do último FD na fila.

Normalmente esta função retira o quadro indicado pelo *rx_head* exceto no caso onde ele é o último quadro na fila. Neste caso o software tem que fazer uma copia deste quadro (marcado com *rx_dummy*) e deixa o último quadro intacto, para que o TBC possa continuar o encadeamento na chegada dos demais quadros. Para marcar a fila como sendo vazia, a funçäo coloca *rx_head* igual a *rx_dummy*.

O algorítmo da funçäo é:

```
int ma_unitdata_ind(int q, FD **fd)
{
      /* se fila vazia ... */
      if (rx_head[q] == rx_dummy[q]) {

            /* se TBC colocou mais quadros na fila ... */
            if (rx_head[q]->conf_ind & NPV)

                  /* atualiza cabeça da fila */
                  rx_head[q] = rx_head[q]->next_fd;
            else
            /* fila vazia */
                  return FAIL;
      }

      /* se quadro não é o último ... */
      if (rx_head[q]->conf_ind & NPV) == NPV) {

            /* devolve endereço do FD e o retira da fila */
            *fd = rx_head[q];
            rx_head[q] = rx_head[q]->next_fd;
      }
/* ... quadro é o último */
      else {
            /* faz copia do FD */
            memcpy(rx_dummy[q], rx_head[q],sizeof(FD));

            /* devolve copia para função invocadora */
            *fd = rx_dummy[q];

            /* marca fila como vazia */
            rx_dummy[q] = rx_head[q];
      }
retorna OK
}
```

Esta função começa a examinar a fila para verificar se ela já foi marcada como vazia. Se positivo, a função testa o bit NPV ("NEXT POINTER VALID") do último FD para verificar se o TBC colocou mais quadros recebidos desde que esta função marcou a fila como vazia. Se existir mais um quadro, então, o apontador de cabeça da fila é atualizado. Se não existirem mais quadros então a fila está realmente vazia e a função retorna o valor "FAIL".

Se o próximo quadro não é o último na cadeia, então esta função devolve o endereço do seu FD, retira este FD da cadeia, e retorna "OK". Porém, se o quadro é o último, então ela faz uma copia do FD (usando *rx_dummy)* e devolve esta copia para a função invocadora, deixando assim o último FD na fila intacto para que o TBC possa continuar com seu encadeamento.

5. A CAMADA LLC

Este capítulo descreve a implementação da camada LLC tipo 3 na interface mini-MAP. No início há uma descrição das características principais da camada onde são citadas suas primitivas e variáveis de estado. Em seguida há uma análise dos aspectos da LLC que mais influenciam a forma da sua implementação. Para concluir, há um detalhamento da implementação em si.

5.1 A Camada LLC tipo 3

A LLC tipo 3 é orientada para aplicações que necessitam de um reconhecimento numa transferência ponto-a-ponto mas que querem evitar a complexidade dos serviços orientados a conexão da LLC tipo 2. A operação da LLC tipo 3 é descrita como sendo do tipo "envia um quadro, recebe um reconhecimento", em que cada unidade de dados enviada resulta no retorno de um reconhecimento.

A LLC tipo 3 foi adotada para a mini-MAP por duas razões. A primeira delas é que seu protocolo é simples e garante um fluxo

confiável de dados ponto-a-ponto. A segunda razão é que a LLC tipo 3 permite a uma estação interrogar ("poll") outras estações para dados (usando a opção "Request With Response" RWR da MAC). Isso facilita a incorporação na rede de estações do tipo "sensor", atuador e controlador dedicado. Estas estações, do ponto de vista da camada MAC, não precisam fazer parte do anel em que circula a ficha.

A interface entre a LLC e a MMS dá-se através de pontos de acesso chamados de SAPs ("Service Access Points") conforme a figura 5.1. A transferência entre camadas pares MMS dá-se sempre entre um ponto de acesso de origem SSAP ("Source Service Access Point") e um ponto de acesso de destino DSAP ("Destination Service Access Point").

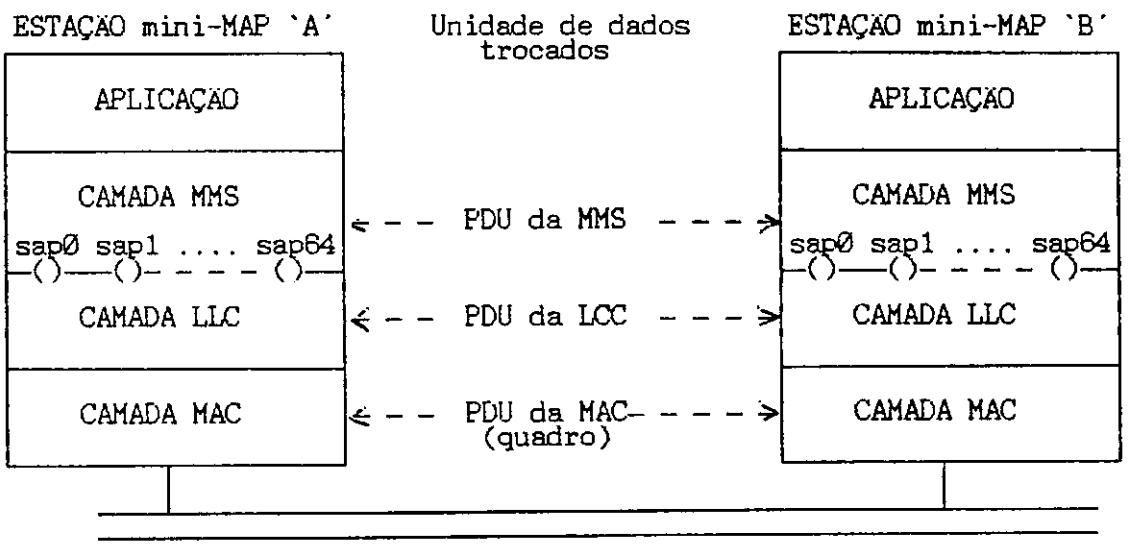
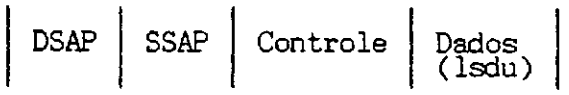


Figura 5.1 - Os Pontos de Acesso

A unidade de dados passada na interface entre as camadas LLC e MMS é chamada de LSDU ("Link Service Data Unit"). A LSDU é incorporada a uma PDU ("Protocol Data Unit") da LLC e enviada a uma camada LLC par usando os serviços da camada MAC. A PDU da LLC consiste, basicamente, da LSDU

mais um cabeçalho com três campos - controle, SSAP e DSAP. Seu formato é o seguinte:



Onde:

DSAP = "Destination Service Access Point" - ponto de acesso do destino (1 byte).

SSAP = "Source Service Access Point" - ponto de acesso na origem (1 byte).

Controle = controle (1 byte).

Data = informaçäo (Ø ou mais bytes).

Do ponto de vista da camada MAC, a unidade de dados recebida da LLC (a PDU da LLC) é chamada de MSDU ("MAC Service Data Unit") e é incorporada numa PDU da MAC (o quadro da rede MAP/mini-MAP) e enviada a uma camada MAC par.

5.1.1 Os Serviços da Camada LLC

A camada LLC tipo 3 oferece os seguintes serviços à camada MMS:

- O Serviço "Data Unit Transmission"
- O Serviço "Data Unit Exchange"
- O Serviço "Reply Data Unit Preparation"

A seguir há uma descrição de cada um destes serviços. A apresentação das primitivas de serviço na forma de funções da linguagem C é arbitrária e está sendo usada apenas para mostrar os parâmetros de maneira clara.

5.1.2 O Serviço "Data Unit Transmission"

O serviço "Data Unit Transmission" (Transmissão de Unidade de Dados) é usado pela camada MMS para o envio de dados a camadas pares. As PDUs de reconhecimento recebidas neste serviço não contém dados. O serviço dispõe das seguintes primitivas:

- *DL_DATA_ACK.request*;
- *DL_DATA_ACK.indication* e
- *DL_DATA_ACK_STATUS.indication*.

A primitiva *DL_DATA_ACK.request* é invocada pela camada MMS para transferir dados para uma camada par. A sinopse desta primitiva é a seguinte:

dl_data_ack_request(ssap,addr,dsap,data,priority)

O parâmetro *ssap* representa o ponto de acesso de origem (SSAP). O parâmetro *addr* ("destination address") representa o endereço MAC da estação de destino. O parâmetro *dsap* representa o ponto de acesso de destino a ser usado. O parâmetro *data* representa uma LSDU. O parâmetro *priority* representa a prioridade a ser usada na transferência.

A primitiva *DL_DATA_ACK_STATUS.indication* é passada para a camada MMS para indicar o sucesso de uma primitiva *DL_DATA_ACK.request* invocada anteriormente. Para a rede mini-MAP o sucesso significa que a LLC recebeu um reconhecimento válido. A sinopse desta primitiva é a seguinte:

dl_data_ack_status_indication(ssap,da,dsap,priority,tx_status)

O parâmetro *tx_status* indica o sucesso da transmissão.

A primitiva *DL_DATA_ACK.indication* é passada para a camada MMS

para indicar a recepção de uma PDU. A sinopse desta primitiva é a seguinte:

dl_data_ack_indication(addr,ssap,dsap,data,priority,rx_status)

O parâmetro *addr* representa o endereço MAC da estação de origem. O parâmetro *ssap* representa o ponto de acesso de origem usado. O parâmetro *rx_status* indica o sucesso da recepção.

5.1.3 O Serviço "Data Unit Exchange"

Este serviço é usado pela camada MMS para a troca de dados com camadas pares, sendo que as PDUs de reconhecimento podem incluir LSDUs. A estação envia uma LSDU para uma outra estação e recebe uma segunda LSDU de volta. Este serviço dispõe das seguintes primitivas:

- *DL_REPLY.request*;
- *DL_REPLY_STATUS.indication* e
- *DL_REPLY_indication*.

A primitiva *DL_REPLY.request* é invocada pela camada MMS para transferir dados para uma entidade par. A sinopse desta primitiva é a seguinte:

dl_reply_request(ssap,addr,dsap,data,priority)

A primitiva *DL_REPLY_STATUS.indication* é passada à camada MMS para indicar o sucesso de uma primitiva *DL_REPLY.request* acionada anteriormente. Com esta primitiva a MMS recebe os dados que vieram como parte da PDU de reconhecimento. A sinopse desta primitiva é a seguinte:

dl_reply_status_indication(ssap,addr,dsap,data,priority,tx_status)

O parâmetro *data* representa uma LSDU.

A primitiva *DL_REPLY.indication* é passada à camada MMS para indicar a recepção de uma PDU. A sinopse desta primitiva é a seguinte:

dl_reply_indication(sa,ssap,dsap,data,priority,rx_status)

O parâmetro *rx_status* representa o estado da recepção.

5.1.4 O Serviço "Reply Data Unit Preparation"

Este serviço é usado pela camada MMS para preparar uma LSDU que será incluída como dados numa futura PDU de reconhecimento. Esta LSDU é guardada num ponto de acesso, esperando a recepção de um quadro do serviço "Data Unit Exchange" (o TBC se encarrega de monitorar as PDUs recebidas para determinar a hora da inclusão da LSDU na PDU de reconhecimento). Este serviço dispõe de duas primitivas:

- *DL_DATA_UPDATE.request*
- *DL_DATA_UPDATE_STATUS.indication*

A primitiva *DL_DATA_UPDATE.request* é invocada pela camada MMS para fornecer a LSDU de um ponto de acesso. A sinopse desta primitiva é:

dl_reply_update_request(sap,data)

O parâmetro *data* representa a LSDU que será guardada no ponto de acesso dado pelo parâmetro *sap*.

A primitiva *DL_DATA_UPDATE_STATUS.indication* é passada de volta à camada MMS para confirmar o sucesso da primitiva *DL_DATA_UNIT.request* feita anteriormente. A sinopse desta primitiva é:

dl_reply_update_status_indication(sap,status)

O parâmetro *status* indica o sucesso na alocação da LSDU.

5.1.5 As Variáveis de Estado

A camada LLC tipo 3 utiliza os seguintes tipos de variável de estado:

- V(RI) - variável de seqüência de recepção;
- V(RB) - variável de estado de recepção e
- V(SI) - variável de seqüência de transmissão.

As variáveis de estado da LLC tipo 3 são usadas para detectar quadros duplicados. O oitavo bit do campo de controle da PDU é usado para o seqüenciamento dos quadros. Na transmissão de um quadro este bit assume o valor da variável V(SI), esta variável sendo invertida depois pronta para o próximo quadro. Na recepção, o oitavo bit do quadro é comparado com a variável V(RI) e, se forem iguais, significa que o quadro é uma duplicata e pode ser ignorado.

A variável V(RB) é usada para guardar o estado de recepção de um quadro recebido com erro. Normalmente, existe uma V(RI) e uma V(RB) para cada combinação de SA, SSAP e prioridade, e existe uma V(SI) para cada combinação de DA, SSAP, e prioridade.

A especificação da LLC tipo 3 define um caso especial onde existe uma simplificação no uso das variáveis V(RI) e V(RB). Quando a camada MAC garante a entrega de PDUs na seqüência correta e também exclusividade de acesso ao meio enquanto qualquer retransmissão está sendo feita, então só existe a necessidade de uma variável V(RB) e uma variável V(RI) para cada ponto de acesso (ou seja, somente 64 de cada uma). O barramento com ficha da IEEE 802.4 satisfaz estas condições.

Esta simplificação permite que as variáveis V(RI) e V(RB) de cada ponto de acesso sejam mantidas pelo TBC numa tabela de 64 entradas, onde

sua manutenção é feita automaticamente com muita eficiência. Por outro lado, as variáveis V(SI) continuam sendo gerenciadas pelo software da LLC.

Para evitar o acumulo de uma grande quantidade de variáveis V(SI) na memória, é alocado um temporizador para cada uma que, depois de um certo período sem uso, causa sua destruição. Este temporizador é reinicializado toda vez que a combinação DA, SSAP e prioridade que corresponde a esta V(SI) é usada para uma transferência. Se o temporizador estourar então a variável V(SI) correspondente é destruida.

5.2 A Forma da Implementação

Esta seção identifica os aspectos principais que influenciam a implementação da camada LLC no PFC.

5.2.1 Elegância contra Eficiência

Em termos da realização desta tarefa existem duas estratégias possíveis. A primeira é aquela em que a implementação tenta seguir ao máximo a `aparência' da LLC tipo 3 como definida na especificação IEEE 802.2. Isso facilita a implementação da camada e a manutenção futura do código.

A segunda estratégia é aquela em que a implementação tenta atingir a mais alta eficiência possível, com estruturas de dados e procedimentos que utilizam ao máximo os recursos do controlador de rede e do ambiente em geral. Note-se que esta estratégia não contradiz a especificação da LLC, que não impõe nenhuma restrição na forma de implementação.

No início do trabalho, nossa tendência era de seguir a primeira estratégia. Porém, na medida que o trabalho progrediu, tornou-se claro que esta estratégia resultaria numa grande queda de desempenho. Como a

eficiência é de importância fundamental para uma interface de rede, a segunda estratégia se tornou predominante.

5.2.2 As Atividades da LLC

Quando se leva em conta as características do TBC e os requisitos da especificação LLC, chega-se à conclusão que existem cinco atividades concorrentes a serem gerenciadas pela tarefa LLC. Estas atividades envolvem:

- a interface entre as camadas LLC e MMS;
- a interface entre a camada LLC e Gerência de Estação;
- a confirmação de transmissão;
- a recepção e
- os temporizadores ligados às variáveis V(SI).

Uma forma de implementação destas atividades seria como cinco "threads", mas a versão 1.Ø do Executivo não dispõe de tal mecanismo, além de qual sua eficiência seria baixa.

Como o TBC oferece um mecanismo de geração de interrupções, uma forma bastante eficiente de implementação é como cinco rotinas de serviço de interrupção. Por exemplo, o TBC poderia gerar uma interrupção quando recebe um quadro ou quando termina de transmitir um quadro. O propósito disso é de tornar a LLC "event driven", em que a tarefa só consome tempo na UCP quando há o que fazer.

Por tornar isso possível seria necessário a incorporação no Executivo de um mecanismo de semáforos, para que a tarefa MMS pudesse 'acordar' a LLC quando há uma mudança no estado da interface MMS/LLC (veja 3.2.3).

Mas como o Executivo não suporta semáforos por enquanto, para chegar numa implementação inicial da LLC decidimos implementar estas

atividades na forma de funções normais (sem ser de interrupção) e fazer um "polling" continua delas. Reconhecemos a ineficiência desta forma.

A seguir estas atividades serão consideradas em detalhe.

5.2.3 A Interface entre as Camadas LLC e MMS

A interface entre as camadas LLC e MMS determina a forma em que as primitivas da LLC são implementadas. A forma da interface adotada no início da implementação é aquela em que o serviço de comunicação do Executivo é usado para enviar mensagens entre as duas tarefas. Cada mensagem representa uma primitiva.

Porém, um estudo da especificação da camada MMS revela a seguinte característica necessária à LLC. Apenas no sentido da LLC para a MMS, cada ponto de acesso da interface necessita do seu próprio buffer para que a MMS possa processar todas as primitivas de um ponto de acesso individualmente. O uso do serviço de comunicação do Executivo não permite isso. Na direção oposta (MMS para a LLC), um único buffer serve para todos os pontos de acesso.

Uma solução para este problema é a extensão do Gerente de Comunicação do Executivo para permitir a existência de vários 'canais' entre tarefas (cada canal sendo usado para um ponto de acesso). A vantagem desta solução é a sua elegância, mas a desvantagem é a sua baixa eficiência. Por exemplo, para examinar um ponto de acesso uma tarefa teria que chamar a primitiva *x_receive_message* do Executivo. Devido a alta freqüência em que esta operação necessita ser feita, a sobrecarga imposta por estas chamadas causa uma ineficiência elevada.

A solução adotada é de usar uma estrutura de dados compartilhada entre as duas tarefas. Esta estrutura é criada pela LLC na sua fase de inicialização e depois passada como mensagem para a tarefa MMS. Esta estrutura contém um apontador para cada ponto de acesso. Este apontador

indica o início de uma cadeia de primitivas. Assim sendo, a tarefa que quer examinar um ponto de acesso precisa somente comparar o apontador correspondente com o valor nulo para saber se há ou não uma primitiva aguardando na cadeia.

O acesso a esta estrutura num dado momento tem que ser exclusivo, o que implica o uso das primitivas *x_lock* e *x_unlock* do Executivo.

5.2.4 A Interface entre a LLC e a Gerência de Estação

Esta interface é usada pela Gerência de Estação para passar as primitivas do tipo *MA_GROUP_ADDRESS.request* que configuram as camadas MAC e LLC. Depois da fase de inicialização estas primitivas são raramente passadas, o que possibilita o uso do serviço de comunicação do Executivo para implementá-las.

Neste caso, a atividade ligada a esta interface é o testar de chegada das mensagens que representam esta primitivas. Isso é feito com o uso da primitiva *x_receive_message* do Executivo.

5.2.5 A Confirmação de Transmissão

Esta atividade consiste do uso da primitiva *MA_UNITDATA-STATUS.indication* da MAC para processar os quadros nas filas de transmissão que tenham sido confirmados pelo TBC (ou seja, aqueles quadros que já receberem um reconhecimento).

Os quadros confirmados são examinados e as primitivas *DL_DATA_ACK_STATUS.indication* e *DL_REPLY_STATUS.indication* enviadas à camada MMS. Esta última primitiva terá como um dos seus 'parâmetros' os dados recebidos na PDU de reconhecimento.

5.2.6 A Recepção

A atividade envolvendo a recepção consiste do uso da primitiva *MA_UNITDATA.indication* da MAC para processar os quadros recebidos. A PDU do quadro recebido é convertida para uma LSDU e enviada a MMS através das primitivas *DL_DATA_ACK.indication* e *DL_REPLY.indication*.

5.2.7 A Gerência dos Temporizadores Estourados

Esta atividade consiste da obtenção dos temporizadores estourados, usando a primitiva *x_get_timer* do Executivo, e a destruição das variáveis de estado V(SI) correspondentes.

O uso dos temporizadores pode ser eliminado dependendo de uma opção do configurador da interface. Neste caso, as variáveis V(SI) criadas permanecem na memória. Sem temporizadores a tarefa LLC se torna mais eficiente e a carga no Executivo imposta pela manutenção deles é reduzida.

5.3 A Implementação da Camada LLC

Esta seção descreve em detalhes a realização da camada LLC. De início há uma descrição da implementação da LSDU e da interface LLC/MMS, seguida de uma descrição do código principal.

5.3.1 A LSDU

A LSDU é a unidade de dados passada na interface entre as camadas LLC e MMS. Nesta implementação, uma LSDU é representada pela seguinte estrutura:

```
typedef struct lsdu {
        int service;
        struct lsdu *next;
        struct lsdu *response;
        unsigned size;
        unsigned status;
        unsigned state;
        unsigned priority;
        char address[6];
        unsigned char dsap;          ┐
        unsigned char ssap;          │
        unsigned char control;       ├── PDU
        unsigned char data[];        ┘
} LSDU;
```

A última entrada *data* da estrutura representa os dados da LSDU, enquanto as outras entradas formam um cabeçalho com formato particular para esta implementação. A vantagem deste formato é que a instalação de uma estrutura destas na interfaces LLC/MMS em si representa uma primitiva sem a necessidade de outros mecanismos ou ações. Isso é possível porque todos os dados relevantes a uma primitiva estão contidos na LSDU.

As entradas *dsap*, *ssap* e *control*, juntamente com a entrada *data*, são agrupadas no formato de uma PDU da LLC. Desta maneira, a manipulação de dados para conversão entre PDUs e LSDUs é reduzida.

A entrada *service* identifica o serviço sendo usado para esta LSDU. A entrada *size* indica o tamanho em bytes do arranjo *data*. A entrada *status* representa o resultado da primitiva (seu significado depende da primitiva). A entrada *priority* indica a prioridade usada. A entrada *address* indica o endereço MAC da estação para a qual a transferência está sendo feita.

A entrada *state* é usado para controlar o processamento da LSDU pela tarefa receptora. Ela pode assumir três valores:

- "NEW" se a LSDU ainda não está sendo processada;
- "ACCEPTED" se a LSDU está sendo processada e
- "CONFIRMED" se o processamento da LSDU já terminou.

A entrada *next* é um apontador usado no encadeamento desta LSDU com

outras na interface LLC/MMS (veja seção 5.3.2). A entrada *response* é um apontador usado quando esta LSDU representa a primitiva *DL_REPLY_STATUS.indication*. Neste caso, a *response* aponta para a LSDU que representa os dados recebidos no reconhecimento.

5.3.2 A Interface LLC/MMS

A interface entre as camadas LLC e MMS é baseada numa estrutura de dados compartilhada, com o seguinte formato:

```
struct llc_mms {
        struct msg_hdr;
        int type;
        LSDU *to_mms[64];
        LSDU *to_llc;
}
```

Esta estrutura é criada e inicializada pela LLC durante a fase de inicialização, depois da qual ela é passada à tarefa MMS na forma de uma mensagem. O acesso à estrutura pelas tarefas é feito através de apontadores.

As entradas *msg_hdr* e *type* são usadas somente na passagem da estrutura como mensagem para a tarefa MMS. A entrada *msg_hdr* é o cabeçalho obrigatório de mensagem exigido pelo Executivo. A entrada *type* identifica o tipo desta mensagem para a tarefa MMS.

A entrada *to_mms* é um arranjo de apontadores, um para cada ponto de acesso da interface LLC/MMS. Este arranjo é usado para a transferência de LSDUs no sentido da LLC para a MMS. Cada apontador indica o começo de uma cadeia de LSDUs. Estas cadeias formam os buffers exigidos pela especificação MMS.

A entrada *to_llc* é um único apontador usado para a transferência de LSDUs no sentido da MMS para a LLC. Neste sentido a especificação da

MMS não exige buffers individuais para cada ponto de acesso. Uma única cadeia contém todas as LSDUs indo no sentido da MMS para a LLC.

A figura 5.2 ilustra um possível estado da interface.

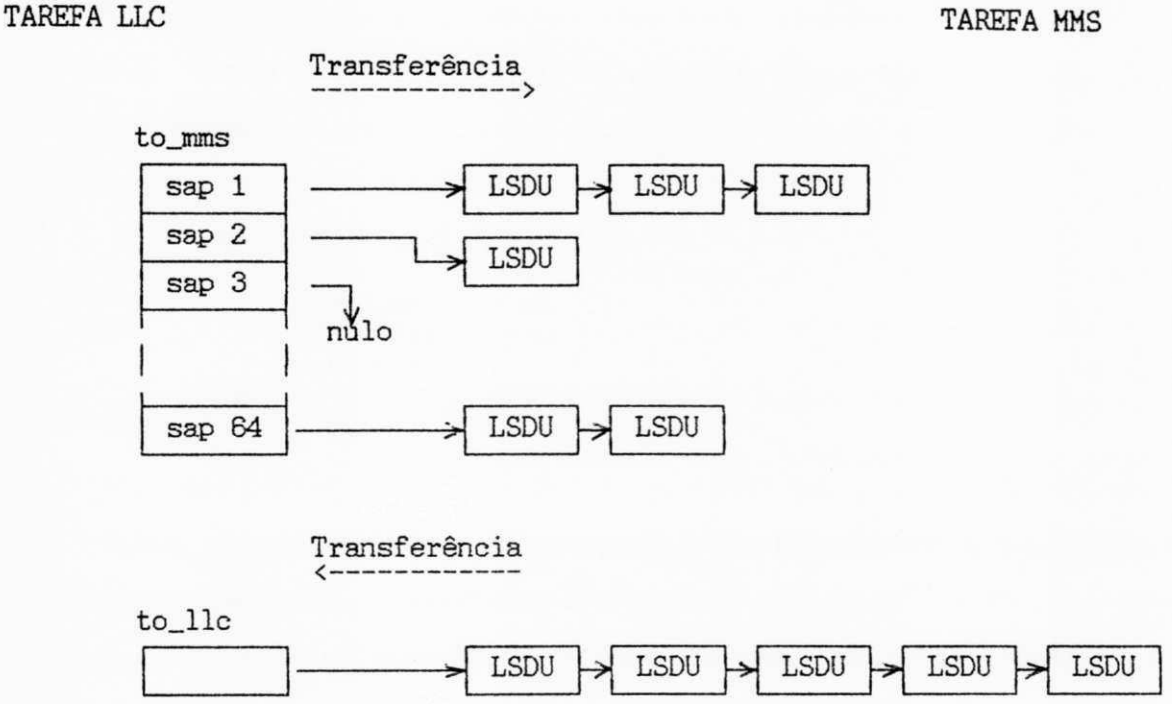


Figura 5.2 - A Interface LLC/MMS

5.3.3 A Implementação das Primitivas

Para exemplificar a forma em que as primitivas são executados, é usado aqui o serviço "data_unit_exchange" como exemplo. A primitiva *DL_REPLY.request* é acionada quando a camada MMS instala uma LSDU na cadeia cujo início é dado pelo apontador *to_llc* da estrutura *llc_mms*. A camada MMS inicializa as seguintes variáveis da LSDU:

- *service* = DL_REPLY;
- *state* = NEW;

venha a ser especificada.

5.3.4 A Implementação das Variáveis de Estado

O controle das variáveis V(SI) é de responsabilidade do software da LLC. Uma V(SI) deve ser criada para cada combinação DS, SSAP e prioridade usada na transferência de uma PDU. Nesta implementação uma V(SI) é representada pela seguinte estrutura de dados:

```
typedef struct {
        unsigned char vsi;
        unsigned char mac[6];
        VSI *next;
        void *last;
} VSI;
```

A entrada *vsi* contém o valor da V(SI), que pode ser Ø ou 1. A entrada *mac* contém o endereço da outra estação. As entradas *next* e *last* são apontadores usados no encadeamento das variáveis V(SI)s.

A LLC mantêm um arranjo de apontadores *vsi_head*. Cada apontador localiza a primeira estrutura *vsi* em uma cadeia. Este arranjo tem duas dimensões, a primeira é indexada pela SSAP e a segunda pela prioridade. A Figura 5.3 mostra a cadeia em que três variáveis V(SI) foram criadas com o mesmo valor de SSAP e prioridade.

A função *get_state* localiza uma variável de estado. Sua definição é:

*VSI *get_state(ssap,priority,da)*

Usando os parâmetros *ssap* e *priority* para indexar o arranjo *vsi_head*, a função *get_state* percorre a cadeia indicada até chegar na V(SI) que tem um valor *mac* igual ao parâmetro *da*. Se não existir, é criada, então,

- *priority* = prioridade a ser usada;
- *address* = endereço MAC (6 bytes) da estação de destino;
- *dsap* = DSAP e
- *ssap* = SSAP.

Posteriormente, a tarefa LLC começa a processar a primitiva, colocando o valor "ACCEPTED" na variável *state* da interface e criando os descritores de FD e BD que descrevem o quadro para o TBC. Usando a primitiva *MA_UNITDATA.request*, a LLC instala este quadro em uma das filas de transmissão do TBC.

Quando a tarefa LLC obtem a confirmação do quadro (através da primitiva *MA_UNITDATA_STATUS.indication*), ela atualiza as seguintes variáveis da LSDU:

- *state* = CONFIRMED;
- *status* = estado da recepção;
- *response* = endereço da LSDU recebido na PDU de reconhecimento.

Esta atualização representa a primitiva *DL_REPLY_STATUS.indication*. Por sua parte, a tarefa MMS examina periodicamente a LSDU para reconhecer a acontecimento da primitiva *DL_REPLY_STATUS.indication*.

Quando a LLC recebe uma PDU de uma outra estação ela converte os dados do quadro para uma LSDU. Para facilitar este procedimento, os buffers de recepção são criados com um espaço inicial não preenchido que corresponde à parte frontal da estrutura LSDU. A criação da LSDU corresponde simplesmente à inicialização desta área inicial.

O ponto de acesso especificado na PDU determina a cadeia em que a LSDU será inserida (o ponto de acesso é usado para indexar o arranjo *to_mms* da estrutura *llc_mms*). A instalação da LSDU na cadeia corresponde à ativação da primitiva *DL_REPLY.indication*.

Uma modificação futura da estrutura *llc_mms* poderia ser a introdução de bandeiras ("flags") para tornar a varredura da estrutura mais eficiente. Isso pode ser feito logo que a interface da tarefa MMS

uma nova V(SI) com valor zero. Como esta nova V(SI) tem uma alta chance de ser referenciada num futuro imediato, ela é colocada no começo de sua cadeia.

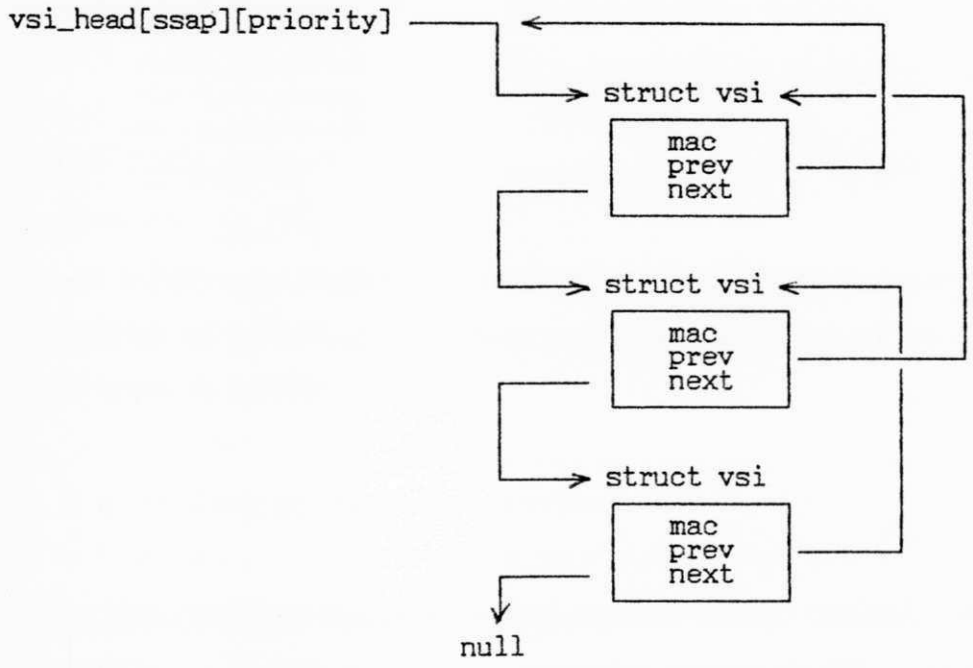


Figura 5.3 Cadeia de Variáveis de Estado

Para cada variável V(SI) normalmente existe um temporizador. A ligação entre a V(SI) e o temporizador é feita no instante que é criada a V(SI), quando também é criado um temporizador usando como identidade o próprio endereço da V(SI).

5.3.5 A Função Principal

A função principal da LLC foi implementada na forma de um laço infinito em que cinco funções são invocadas seqüencialmente. Cada uma destas funções gerencia uma das atividades mencionadas na seção 5.2. A

função principal é a seguinte:

```
void main(void)
{
    initialize();
    for (;;) {
        check_interface();
        check_messages();
        check_tx_queues();
        check_rx_queues();
        check_timeouts();
    }
}
```

O laço infinito satisfaz o requisito do Executivo que a tarefa nunca faz um retorno da função *main*. A seguir há uma descrição de cada uma destas cinco funções auxiliares.

5.3.6 A Função "check_interface"

Esta função examina a estrutura de dados *llc_mms* da interface LLC/MMS a procura de primitivas oriundas da tarefa MMS . Seu algorítmo é o seguinte:

```
void check_interface(void)
{
    LSDU *lsdu;

    /* garante acesso exclusivo à interface */
    x_lock();

    /* examina todas as LSDUs da cadeia de entrada */
    lsdu = interface->to_llc;
    while (lsdu != NULL) {

        /* se LSDU não foi processada .. */
        if (lsdu->state == NEW) {
            lsdu->state = ACCEPTED;
            allocate_bd(....);
```

```
            allocate_fd(....);
            ma_unitdata_request(....);
            lsdu = lsdu->next;
        }
    }
    x_unlock();
}
```

O apontador *interface* é usado pela tarefa LLC para ter acesso à estrutura *llc_mms*. As primitivas x_lock() e x_unlock() do Executivo são usadas para garantir acesso exclusivo à estrutura *llc_mms* que é compartilhada com a tarefa MMS. A função varre a cadeia (cujo início é dado pelo apontador *to_llc*). Quando uma nova LSDU é encontrada, esta função cria os descritores FD e BD e passa o quadro resultante à MAC usando a primitiva *MA_UNITDATA.request*.

5.3.7 A Função "check_messages"

Esta função usa a primitiva *x_get_message* do Executivo para obter qualquer mensagem vinda da Gerência de Estação. Seu algorítmo é o seguinte:

```
void check_messages(void)
{
    struct message *msg;
    int status;

    /* pega próxima mensagem */
    x_receive_message(&msg,&status);
    if (status == OK) {

        /* processa mensagem de acordo com seu tipo */
        switch(msg->type) {
            case MA_DESIRED_RING_MEMBERSHIP_REQUEST:
                tbc_inring(msg->param1);
                break;
            case MA_INITIALIZE_PROTOCOL_REQ:
                .......
            case MA_SET_TIMER_LIMIT:
                ....
            case ..........
```

```
        }
    }
}
```

Há uma entrada "case" para cada tipo de mensagem que a tarefa recebe. A primeira entrada, dada como exemplo, é da primitiva *MA_DESIRED-RING_MEMBERSHIP.request*, cuja ação é invocar a função *tbc_inring* que insere ou retira a estação do anel.

5.3.8 A Função "check_timeouts"

Esta função testa os temporizadores para verificar se houve estouro. Nesta implementação da LLC, o uso dos temporizadores apenas determinam a destruição de variáveis de estado V(SI). O algorítmo desta função é:

```
/* reserva de estruturas V(SI) */
extern struct state_var *sv_pool[SP_POOL_SIZE];

void check_timers(void)
{
    VSI *v;
    int status;

    /* se houver temporizador estourado ... */
    x_get_timer(&v,&status);
    if (status == X_OK) {

        /* remove variável V(SI) da sua cadeia */
        *v->last = v->next;

        /* devolve a memória da variável */
        if (sv_pool_count < SP_POOL_SIZE)
            sv_pool[sv_pool++] = v;
        else
            x_return_memory(sizeof(VSI),v);
    }
}
```

Um temporizador estourado é obtido usando a primitiva *x_get_timer* do Executivo, que devolve para a variável *v* a identidade do temporizador.

Nesta implementação, a identidade de um temporizador é o endereço da variável V(SI) correspondente.

Para evitar a alocação e liberação frequente de memória, a LLC mantém uma reserva ("pool") de estruturas *vsi*. Ao completar a destruição da variável V(SI), a memória usada por ela é devolvida para esta reserva. Se a reserva estiver cheia, então a memória é devolvida ao Gerente de Memória do Executivo.

5.3.9 A Função "check_tx"

Esta função testa o resultado da primitiva *MA_UNIT-DATA_STATUS.indication* da MAC, para processar os quadros que foram confirmados nas filas de transmissão. Seu algorítmo é o seguinte:

```
void check_tx_status(void)
{
    FD *fd,*resp_fd;
    BD *bd,*resp_bd;
    int q;

    /* repete para cada fila de transmissão */
    for (q = 0; q < 4; q++) {

        /* repete para cada quadro confirmado da fila */
        while (ma_unitdata_status(q,&fd) == OK) {

            /* converte o quadro numa LSDU */
            prepare(fd,&bd,&lsdu);

            /* se não houve reconhecimento ... */
            if (!fd->conf_ind & NP) {

                /* confirma lsdu e devolve estado ruim */
                lsdu->status = NO_RESPONSE;
                lsdu->state = CONFIRMED;
            }
            else {

                /* prepara o quadro de reconhecimento */
                prepare(resp_fd,&resp_bd,&resp_lsdu);

                /* se quadro não é duplicado */
```

```
                if (fd->vsi != resp_lsdu->control & PF) {

                    /* confirma lsdu */
                    lsdu->response = resp_lsdu;
                    lsdu->status = OK;
                    lsdu->state = CONFIRMED;
                }

                /* recicla descritores da PDU de reconhecimento */
                recycle(resp_fd,resp_bd);
            }

            /* recicla descritores do quadro transmitido */
            recycle(fd_ptr,bd_ptr);
        }
    }
}
```

Para cada fila de transmissão esta função obtém todos os quadros confirmados pelo TBC. A confirmação é dada pela palavra *con_ind* do descritor FD. Se o resultado for negativo então não foi recebido nenhum reconhecimento e a LSDU será marcada como "CONFIRMED" e seu estado como "NO_RESPONSE".

Quando o FD desta LSDU é criado, o estado de V(SI) é guardado na variável *vsi* do FD. Para que o quadro de reconhecimento seja válido, o bit de seqüenciamento (o oitavo bit da palavra de controle da PDU) deve ser o oposto do bit guardado em *vsi*, caso contrário, o quadro recebido é uma duplicação e é ignorado.

Se o quadro for válido, ele é convertido numa LSDU de resposta. A LSDU original é atualizada com o estado OK e o endereço da LSDU de reconhecimento.

5.3.10 A Função "check_rx"

Esta função usa a primitiva *MA_UNITDATA.indication* da MAC para processar os quadros recebidos. Seu algorítmo é:

```
void check_rx_status(void)
{
    FD *fd;
    BD *bd;
    LSDU *lsdu;
    int q;

    /* repete para cada fila */
    for (q = 0; q < 4; q++) {

        /* obtem quadros recebidos */
        while (ma_unitdata_ind(q,&fd) == OK) {

            /* converte quadro para LSDU */
            prepare(fd,&bd,&lsdu);

            /* instala LSDU na interface LLC/MMS */
            indication(lsdu);

            /* recicla FD e BDs */
            recycle(fd);
            recylce(bd);
        }
    }
}
```

Esta função testa as quatro filas de recepção para processar todos os quadros recebidos. Cada quadro é convertido numa LSDU e instalado na interface LLC/MMS. Seus descritores (FD e BD) de quadro são reciclados (postos em reserva, prontos para serem utilizados posteriormente).

6. CONCLUSÃO

O objetivo deste trabalho foi o desenvolvimento de três partes fundamentais de uma interface inteligente mini-MAP, visando a obtençäo de uma arquitetura do hardware adequada e a otimizaçäo dos recursos de software criados.

Este trabalho está situado em um contexto de pesquisa mais abrangente que visa o desenvolvimento de uma plataforma de hardware mini-MAP para o desenvolvimento de protocolos e aplicativos para a automação industrial. Neste contexto, a avaliaçäo completa da eficácia da interface desenvolvida depende da implantaçäo de pelo menos uma camada do protocolo adicional (camada MMS) que se encontra atualmente em fase de desenvolvimento. No entanto, podemos oferecer de imediato algumas considerações e conclusões sobre o MAP e a arquitetura do PFC aqui desenvolvida.

6.1 A Arquitetura MAP

A arquitetura MAP foi projetada para grandes centros de produçäo

industrial, com seus computadores de grande e médio porte. Pela experiência que a General Motors tem com o MAP e o entusiasmo com que ela continua a promové-la sugere-se que o MAP seja apropriado para estes centros.

O trabalho feito no projeto do PFC é mais orientado para a arquitetura mini-MAP que, com sua arquitetura simplificada de três camadas, talvez seja o ponto de partida para aquelas instituições que queiram avaliar o MAP mas que não disponham dos recursos necessários para uma implementação baseada nas sete camadas da arquitetura MAP.

Os méritos técnicos do projeto MAP são geralmente aceitos pela comunidade envolvida com a automação industrial. Porém, até agora sua difusão tem sido lenta. A seguir tecemos algumas considerações visando esclarecer esta afirmativa.

O custo de uma interface MAP, MAP/EPA ou mini-MAP é alto. Isso deve-se ao fato de que estas arquiteturas são projetadas para o ambiente industrial. Este ambiente dispõe de uma multidão de equipamentos diferentes, cada um precisando de uma interface específica. Devido ao tamanho reduzido do mercado industrial em relação ao comercial, o investimento necessário para desenvolver estas interfaces não esta disponível..

Nossa experiência neste projeto tem demonstrado que o esforço envolvido no desenvolvimento de uma interface de rede é grande. Uma das razões para isso é que as especificações necessárias não são claras, dificultando a assimilação do conteúdo e deixando muito escopo para mal-interpretação. Em nossa opinião, a implementação de uma interface MAP de sete camadas é somente viável para aquelas instituições que se dediquem exclusivamente a produção de interfaces de rede e que tenham as ferramentas e experiência adequadas.

Portanto, os fornecedores de sistemas industriais estão atualmente implantando sistemas baseados em outras alternativas para a comunicação

inter-estação. Estas alternativas incluem redes locais difundidas no ambiente comercial (por exemplo, Netware com adaptadores Ethernet) e interfaces seriais do tipo RS422. A escassez de interfaces MAP tem prejudicado sua adoção e, quando estão disponíveis, seus altos preços as colocam em desvantagem em relação as alternativas citadas.

O propósito da arquitetura MAP/EPA é viabilizar o MAP para operação em tempo-real. Porém, com esta demora na adoção da MAP/EPA pode ser que surjam alternativas mais atraentes e que resultem no abandono do MAP/EPA. Um candidato para isso é o barramento de campo que está sendo avaliado atualmente.

6.2 A Arquitetura do PFC

Do ponto de vista técnico, a arquitetura do PFC é eficiente, retirando do hospedeiro a maior parte da carga de processamento da interface MAP. A incorporação do Executivo cria um ambiente otimizado para o software de rede, facilitando sua implementação. Porém, existem outros critérios, além do técnico, que determinam a adequação da arquitetura do PFC para casos específicos.

Uma alternativa ao uso da arquitetura do PFC é o uso de uma interface mais simples e o aumento da potência computacional do hospedeiro. Isso, talvez, seja a alternativa mais atraente quando se trata de um hardware padrão como, por exemplo, da família IBM PC. Neste caso, é mais eficaz mudar a placa mãe do que instalar uma interface inteligente.

Por outro lado, existem situações onde uma interface inteligente é mais apropriada. O primeiro caso é quando não há um "upgrade" disponível ou quando o "upgrade" é mais caro do que uma interface inteligente (situações comuns com equipamentos que não seguem um padrão difundido de hardware).

Um segundo caso é típico de minicomputadores considerados obsoletos. Como o custo de equipamento atualizado deste porte é, em muitos casos, exorbitante, a vida útil destes minicomputadores pode ser aumentada com o uso de interfaces inteligentes.

Um terceiro caso é quando um "upgrade" da UCP implica na adoção de uma UCP nova. Neste caso, as versões iniciais destas novas UCPs normalmente têm falhas de projeto (por exemplo, depois de 2 anos de uso, novas falhas ainda estão sendo descobertas no 80386). Estas falhas prejudicam a confiabilidade do equipamento e, conseqüentemente, sua adequação para o ambiente industrial, onde o equipamento tem que operar continuamente sem falha.

Em todas as classes de computadores, dos supercomputadores aos microcomputadores, os projetistas estão visando multiprocessamento como a alternativa para continuar o crescimento de desempenho que tem caracterizado a área de informática desde sua origem. Portanto, a exploração de arquiteturas como a da PFC é uma parte fundamental neste processo.

7. REFERêNCIAS BIBLIOGRáFICAS

[ARA 86] Araújo, J.F.M. et al, `Redes Locais de Computadores - Tecnologias e Aplicaçöes´, McGraw-Hill, 1986.

[COM 84] Comer, D., `Operating System Design - The XINU Approach´, Prentice-Hall, 1984.

[CRO 85] Crowder, R., `The MAP Specification´, Control Engineering, outubro 1985, pp. 22-25.

[DEI 84] Deital, H.M., `An Introduction to Operating Systems´, Addison-Wesley, 1984.

[DON 81] Donovan, N., `Operating Systems´, McGraw-Hill, 1981.

[DIR 88] Dirvin, R.A., Larkin, A.M., `Single VLSI chip helps you implement EPA factory networks´, Electronic Design News, 4 Agosto 1988, pp. 145-152.

[GLA 88] Glass, G., `Weighing the Options - Comparing the Many Flavors of Multitasking´, Byte, Julho 1988.

[GM 87] GM, `MAP Specification - Version 3.Ø´, General Motors Corporation, 1987.

[GRO 86] Groff, J.R., Weinberg, P.N., `Sistema Operacional Unix: Um Guia Conceitual´, EBRAS, 1986.

[GUI 80] Guimaräes, C.C, `Princípios de Sistemas Operacionais', Editora Campos, 1986.

[HEN 88], Hensler, C., Sarno K., `Marrying Unix and the 80386', Byte, Abril 1988.

[HOR 82] Horowitz, E., `Fundamentos de Estrututas de Dados', Editora Campos, 1982.

[HYD 88] Hyde R.L., `Overview of Memory Management', Byte, Abril 1988.

[IEEE 88A] IEEE, `IEEE Std 802.4, Token Passing Bus Access Method and Physical Layer Specification', Institute of Electrical and Electronic Engineers, 1988.

[IEEE 88B] IEEE, `IEEE Std 802.2, CSMA-CD Access Method and Physical Layer Specification', Institute of Electrical and Electronic Engineers, 1988.

[INT 81] INTEL, `iAPX 86,88 User's Manual', Intel Corporation, Agosto 1981.

[INT 83A] INTEL, `iAPX 86,88,186 and 188 User's Manual', Intel Corporation, Maio 1983.

[INT 83B] INTEL, `Microsystem Components Handbook - Volume 1', Intel Corporation, 1983. pp. 3/256-3/333, 3/358-3/411.

[ISA 85] ANSI/ISA - 72.01, `PROWAY - LAN Industrial Data Highway', Instrument Society of America, 1985.

[ISO 85] ISO, `Information Processing Systems - Open Systems Interconnection - Basic Reference Model', International Standards Organization, 1985.

[ISO 88] ISO/IEC, `Standards for Local Area Networks: Logical Link Control - Type 3 Operation, Acknowledged Connectionless Service', ISO/IEC JTC 1/SC 6 N 4960, 1988.

[JAM 87] Jamsa, K., `Windows Programming Secrets', McGraw-Hill, 1987.

[JOB 90] Joberto, S.B.M., Turnell, D.J.,'Considerations about the Implementation of mini-MAP Architectures on a Front-End Processor', 1990.

[LAC 88] Lacobucci, E,. 'OS/2 Programmers Guide', Osborne McGraw-Hill, 1988.

[LET 88] Letwin,G., 'Inside OS/2', Microsoft Press, 1988.

[MEN 88] Mendes,M.J., Magalhães, M., 'Redes Locais Industriais e o Projeto de Padronizaçäo MAP/TOP', SBA: Controle & Automaçäo, Vol. 2, Nº 1, pp.55-70.

[MEN 89] Mendes,M.J., 'Comunicaçäo Fabril e o Projeto MAP/TOP', Escola Brasileiro Argentina de Informática, IV EBAI, Argentina, Jan. 1989.

[MOT 87A] MOTOROLA, 'MC68824 Token Bus Controller User's Manual', Motorola Incorporated, 1987.

[MOT 87B] MOTOROLA, 'MC68824 Token Bus Controller to MC68010 Hardware Interface', MC68824 Application Note, Motorola Incorporated, 20 Fevereiro 1987.

[MOT 87C] MOTOROLA, 'MC68824 Token Bus Controller to iAPX 80186 Interface', MC68824 Application Note, Motorola Incorporated, 25 setembro 1987.

[MOT 87D] MOTOROLA, 'Enhanced Token Bus Controller MC68824 - Addendum to MC68824 Users Manual',Motorola Incorporation, 30 setembro 1987.

[PAG 88] Paglione, A. et al., 'Protocol RS-511 - Um Modelo de Implementaçäo', Out. 1988.

[PAR 88] Parker, M.B., 'First Come, First Served', Byte, Julho 1988.

[ROB 88] Robie, J., 'Fair Share', Byte, Julho 1988.

[SMI 88] Smith, B.E., 'It's a Natural', Byte, Julho 1988.

[SPA 86] Spanier, S., 'FEPs Ease Migration to New LAN Protocols', Mini-Micro Systems, Setembro 1986, pp. 93-101.

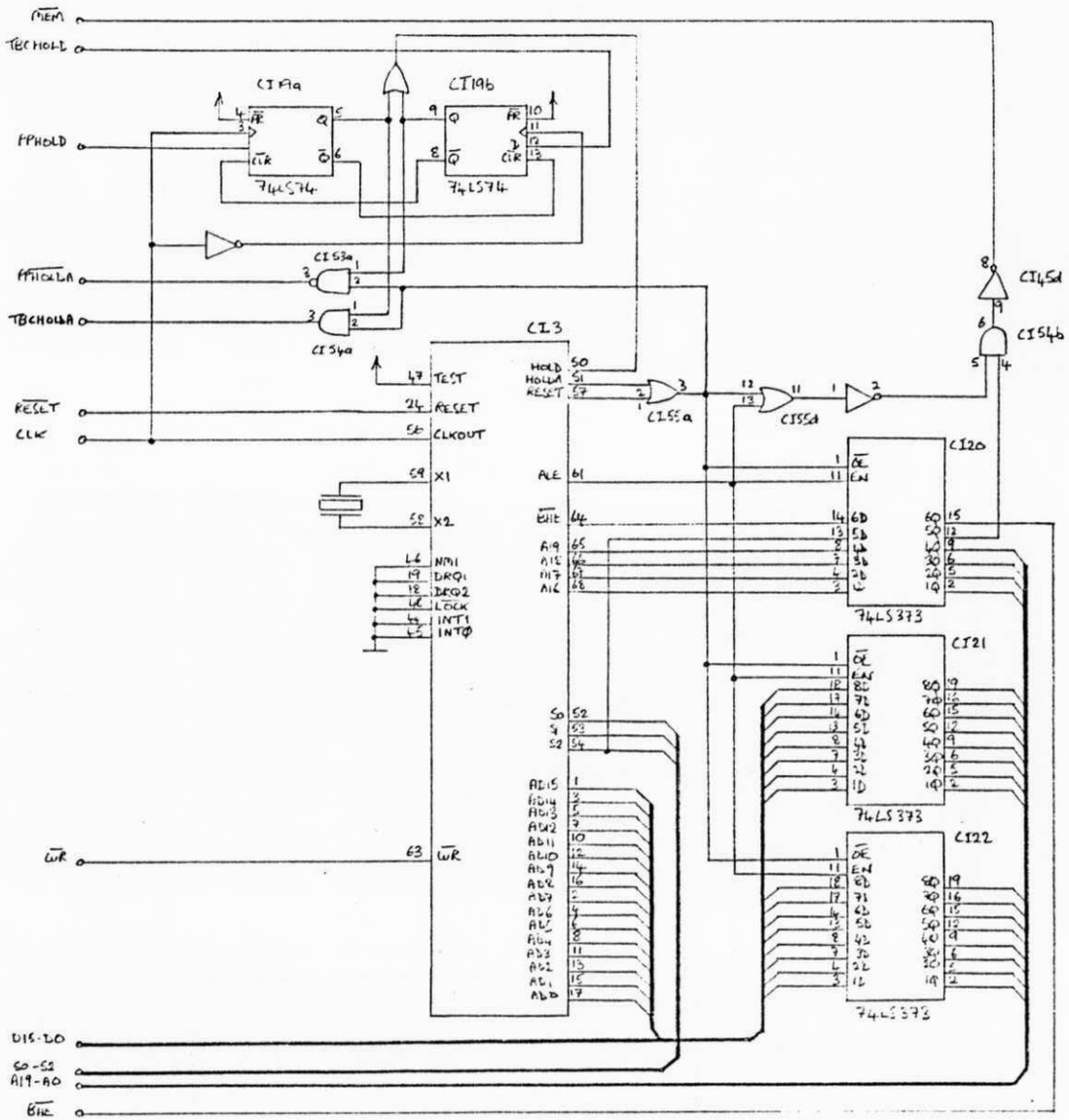
[TAN 87] Tanenbaum, A.S., `Operating Sistems: Design and Implementation', Prentice Hall, 1987.

[TEL 87] TELEBRáS, `Documentação Técnica do Projeto PP (Processador Preferêncial), 1987.

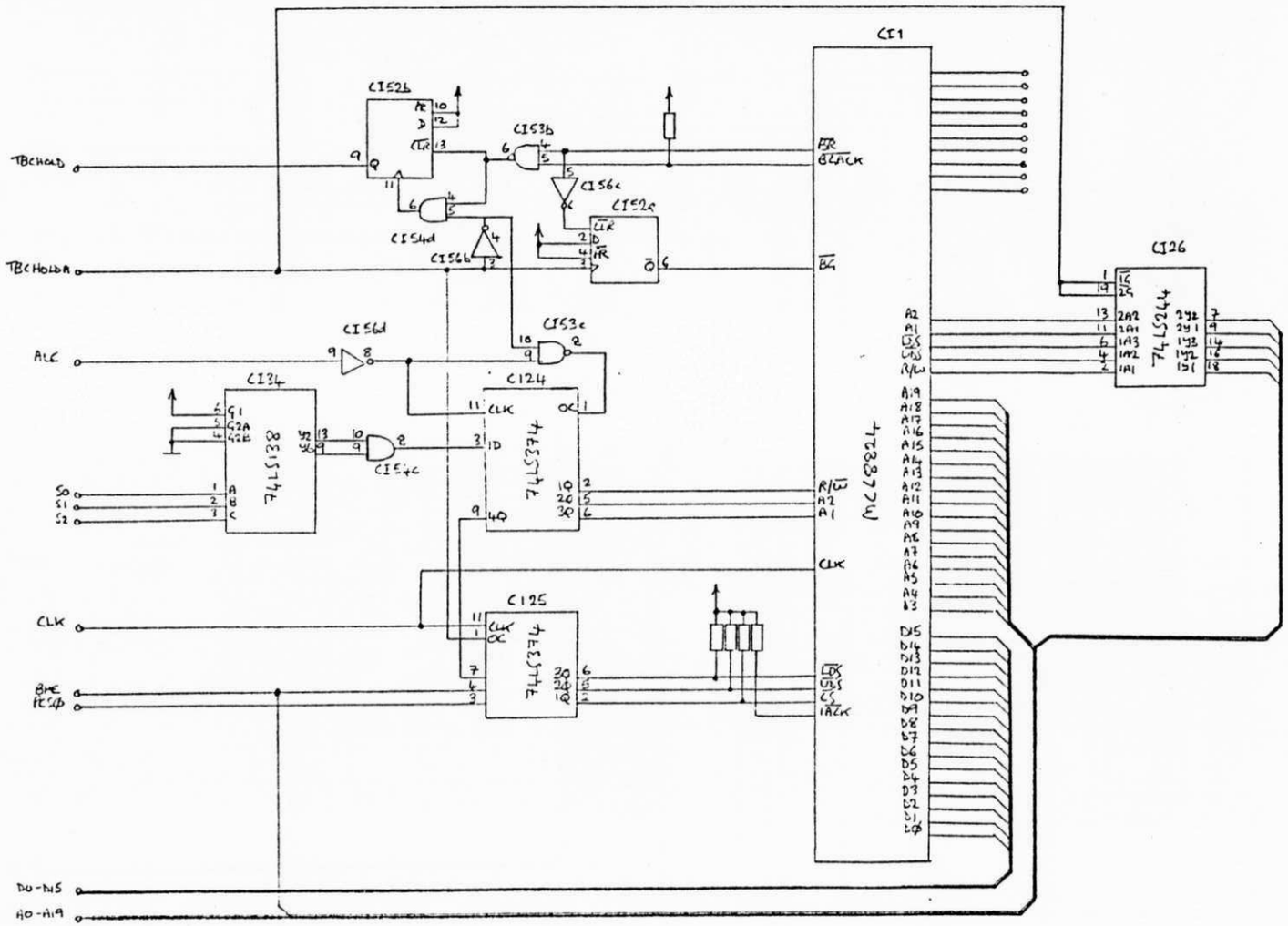
[TUR 89] Turnell, D.J., Joberto S.B.M., `Executivo Multi-Tarefa para Processador Frontal de Rede', 7º Simpósio Brasileiro de Redes de Computadores, Porto Alegre, Março 1989.

[TUR 90] Turnell, D.J., Joberto S.B.M., `Uma Implementação das Camadas MAC/LLC para a Interface Mini-MAP', 8º Simpósio Brasileiro de Redes de Computadores, Abril 1990.

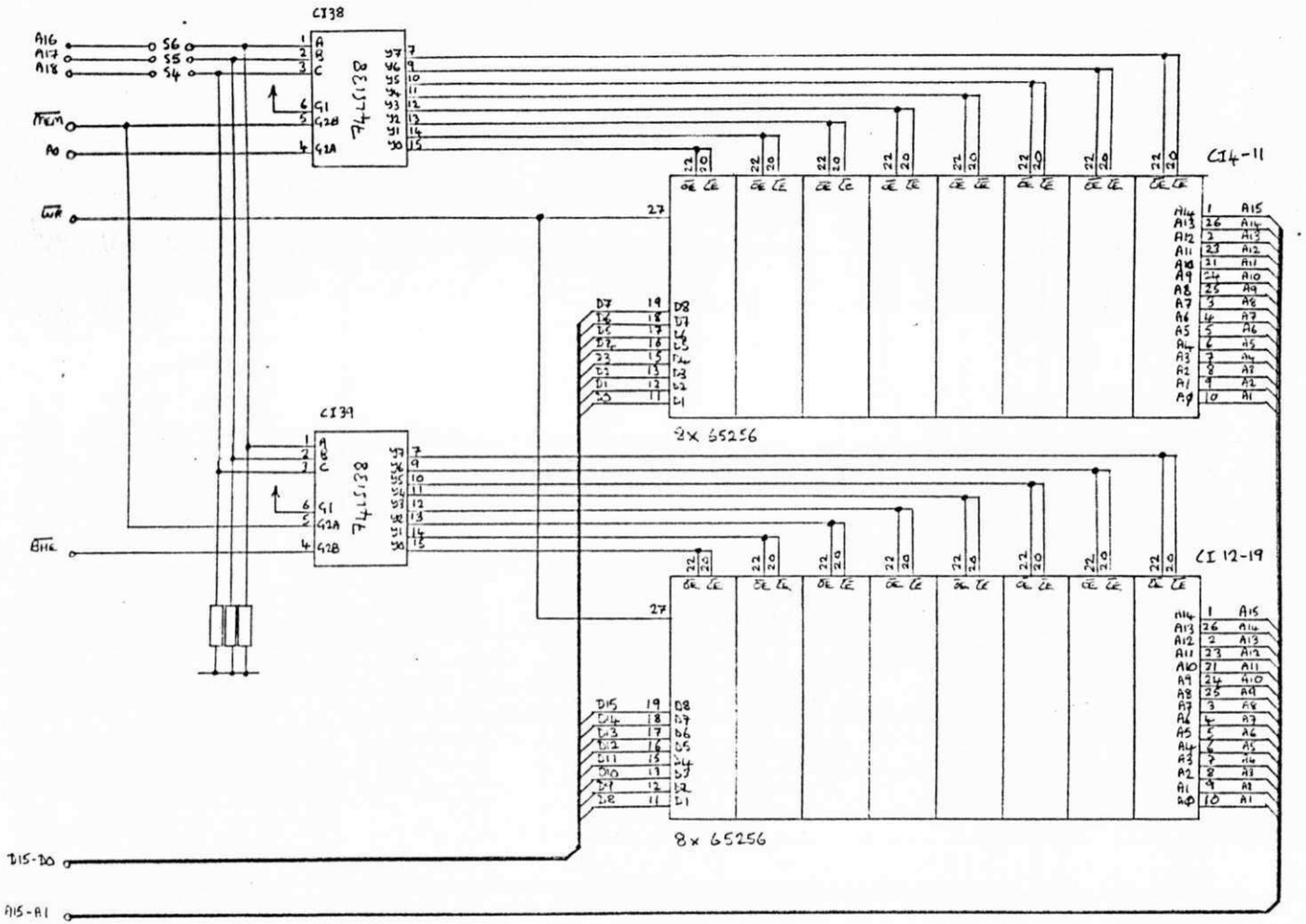
APêNDICE A - OS DIAGRAMAS DE CIRCUITO



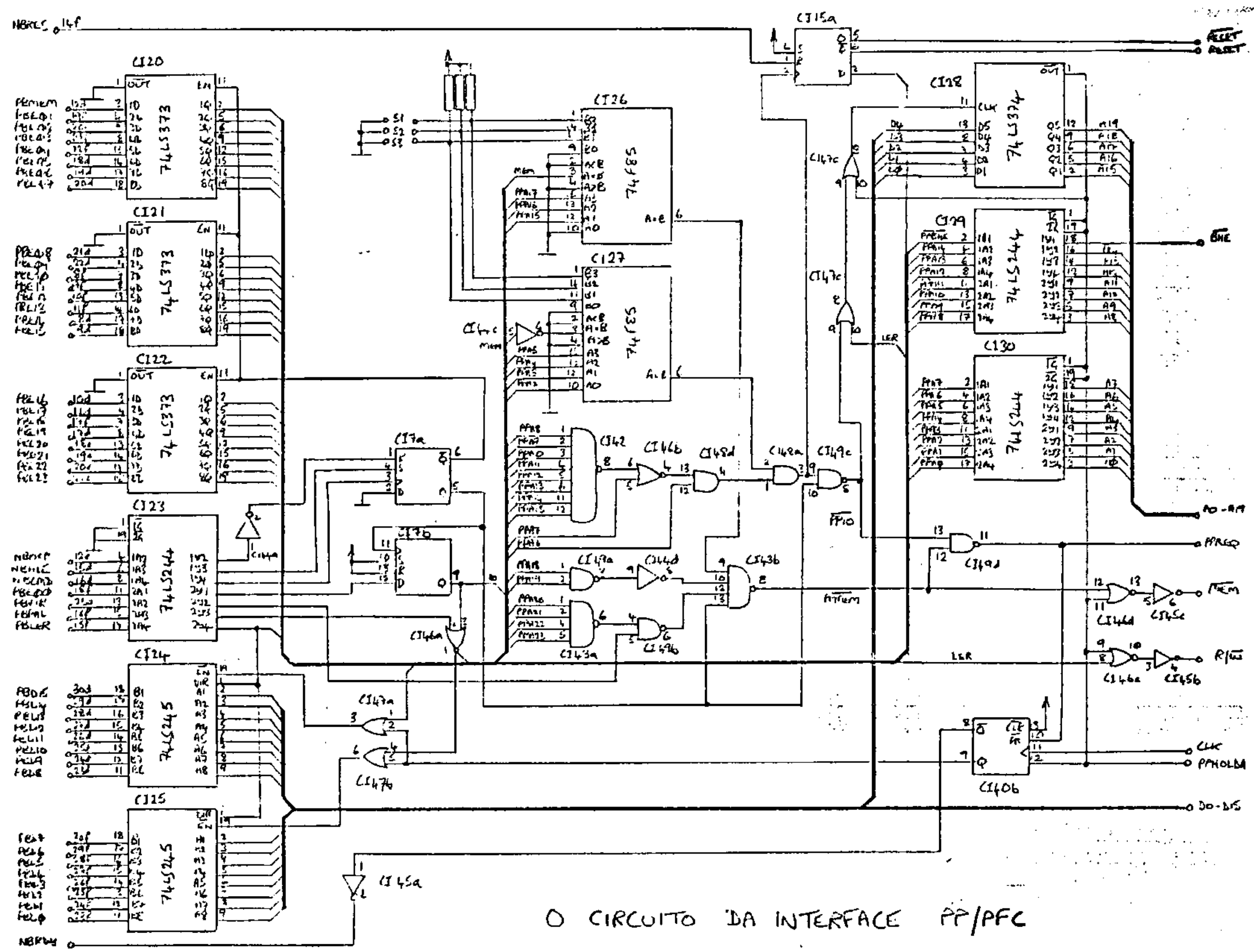
O CIRCUITO DO MICROPROCESSADOR



O CIRCUITO DO CONTROLADOR DE REDE



O CIRCUITO DE MEMÓRIA



O CIRCUITO DA INTERFACE PP/PFC

APêNDICE B - O CóDIGO PRINCIPAL DO EXECUTIVO

```
/*****************************************************************************
*
* Name:         EXEC.C
* Project:      PFC
* Application:  Executive
* Module:       Main Executive module
* Update:       26/06/90
*
*****************************************************************************/

#pragma check_stack(off)

#include "..\pfc\general.h"
#include "..\pfc\x.h"
#include "exec.h"
#include "func.h"
/***/
/*****************************************************************************
*
* GLOBAL VARIABLES
*
*****************************************************************************/

struct init_data *task_data             /* pointer to initialization data supplied by the host */
    = (struct init_data *) 0xffe00000L;
struct task_info near task[TASKS];      /* task information */
unsigned near xtask = 0;                /* current task number */
int near last_task;                     /* last task number */
int near lock = 0;                      /* flag to inhibit task switching */
struct mem_block *free_mem_head = LNULL;/* pointer to first free memory block */
struct msg_hdr *out_msg_head = LNULL;   /* pointer to first message to be sent to host */
struct msg_hdr *out_msg_tail = LNULL;   /* pointer to last message to be sent to host */
struct com_buffer *com;                 /* pointer to structure used in host comunications */
int near timer_pool_index = 0;          /* number of timer structures in pool */
timer *timer_pool[TIMER_POOL_SIZE];     /* pool for timer structures */
unsigned para_size;                     /* memory allocation block size */
unsigned para_scale;                    /* memory allocation prescaled value */

#if defined(DEBUG)
long near end_stack[TASKS];             /* end of stack addresses for all tasks */
#endif
/***/
/*****************************************************************************
*
* MAIN()
*
* Definition:
*   This is the main C function for the Executive. It is called by the
*   'startup' routine in the EXECASM.ASM module when the Executive is
*   booted.
*
* Parameters:
*   No input.
*
* Returns:
*   Never exits.
*
*****************************************************************************/

void main(void)
{
    /* initialize executive and prepare for multi-tasking */
    initialize();

    /* call background function */
    daemon();
}
/***/
/*****************************************************************************
*
* INITIALIZE()
*
* Definition:
*   This function initializes the Executive. It obtains its initialization
*   data from the init_data structure which is placed by the LOADER utility
*   at address 0xfffe0.
```

```
77    *
78    * Parameters:
79    *   No input.
80    *
81    * Returns:
82    *   No output.
83    *
84    ********************************************************************************/
85
86    void initialize(void)
87    {
88        struct stack_frame *stackptr;
89        int i;
90
91        /* set up interrupt vectors */
92        for (i = 0; i < 64; *(void **)(i++ * 4L) = unknown_int);
93        *((void **)(32L * 4L)) = exec_int;
94        *((void **)(19L * 4L)) = rtc;
95        *((void **)(6L * 4L)) = bad_op;
96
97        /* set real timer clock timer */
98        outpw(0xff62,0xc350);       /* count for 20ms */
99        outpw(0xff66,0xe001);       /* control */
100
101       /* set interrupt controller */
102       outpw(0xff32,0x0000);       /* timer control register - priority 0, unmasked */
103
104       /* initialise each task */
105       last_task = task_data->last_task;
106       for (i = 0; i <= last_task; i++) {
107
108           /* initialize stack */
109           if (i != 0) {
110               stackptr = task_data->task[i].stackptr;
111               stackptr = stackptr - 1;
112               stackptr->return_address = task_data->task[i].codeptr;
113               stackptr->af = 0x0200;
114           }
115
116           /* initialise task information for all tasks */
117           task[i].stackptr = stackptr;
118           task[i].state = ACTIVE;
119           task[i].priority = task[i].countdown = task_data->task[i].priority;
120           task[i].msg_head = task[i].msg_tail = LNULL;
121           task[i].timer_head = LNULL;
122           task[i].timeout_head = task[i].timeout_tail = LNULL;
123           task[i].sleep = 0L;
124       }
125
126       /* initialize starting stack addresses */
127       #if defined(DEBUG)
128       for (i = 0; i <= last_task; i++)
129           end_stack[i] = FP_SEG(task_data->task[i].stackptr) * 16L
130                        + FP_OFF(task_data->task[i].stackptr);
131       #endif
132
133       /* initialise free memory */
134       para_size = task_data->para_size;
135       para_scale = para_size / 16;
136       free_mem_head = task_data->start_mem;
137       free_mem_head->size = task_data->memory_size;
138       free_mem_head->next = LNULL;
139       #if defined(DEBUG)
140       free_mem_head->testcode = TESTCODE;
141       #endif
142
143       /* initialise PFC/HOST exchange buffer */
144       com = task_data->com;
145       com->to_host_busy = com->to_pfc_busy = 0;
146
147       /* inform user that initialization is completed */
148       #if defined(DEBUG)
149       status_msg("Executive Initialized");
150       #endif
151
152       /* enable interruptions to allow real time clock to start task switching */
```

```
153         _enable();
154     }
155     /***/
156     /*****************************************************************************
157     *
158     * DAEMON()
159     *
160     * Definition:
161     *   This function runs in the background - it should never be suspended.
162     *
163     * Parameters:
164     *   No Input.
165     *
166     * Returns:
167     *   Never returns.
168     *
169     *****************************************************************************/
170     void daemon(void)
171     {
172         for(;;outpw(0x200,0));
173     }
```

```
/*****************************************************************************
*
* RTC.C
*
* Project:          PFC
* Application:      Executive
* Module:           Real time clock functions
* Update:           26/06/90
*
*****************************************************************************/

#pragma check_stack(off)

#include "..\pfc\general.h"
#include "..\pfc\x.h"
#include "exec.h"
#include "func.h"
/***/
/*****************************************************************************
*
* TICK()
*
* Definition:
*   This function is called from the rct_int assembler routine every time the
*   real-time clock generates an interrupt. It attends to the timers, the
*   PFC/PP communications buffers, and selects the next task to execute. When
*   finished it used the restore_context function to return execution to the
*   next task.
*
* Parameters:
*   No input.
*
* Returns:
*   No output.
*
*****************************************************************************/

void tick(void *stackptr)
{
    register int i;
    int dest_task;
    timer *timer;
    struct msg_hdr *mem, *msg;

    /* check stack pointer and timer and memory chains */
    #if defined(DEBUG)
    stack_test(xtask,stackptr);
    chain_test();
    #endif

    /* repeat for each task */
    for (i = 1; i <= last_task; i++) {

        /* if task is sleeping then decrement sleep count */
        if (task[i].sleep != 0L)
            task[i].sleep--;

        /* if task has timers then attend first in chain */
        if (task[i].timer_head != LNULL) {

            /* see if timer has expired */
            if (--task[i].timer_head->timeout <= 0L) {

                /* take timer out of timer chain */
                timer = task[i].timer_head;
                task[i].timer_head = timer->next;

                /* put timer at end of timeout chain */
                if (task[i].timeout_head == LNULL)
                    task[i].timeout_head = task[i].timeout_tail = timer;
                else {
                    task[i].timeout_tail->next = timer;
                    task[i].timeout_tail = timer;
                }
                timer->next = LNULL;

```

```
77                  /* if task was waiting then wake him up */
78                  if (task[i].state == WAITING) {
79                      task[i].state = ACTIVE;
80                      *task[i].event_status = X_TIMEOUT_EVENT;
81                  }
82              }
83          }
84      }
85
86      /* check input buffer for data */
87      if (com->to_pfc_busy) {
88
89          /* set up local variables */
90          msg = (struct msg_hdr *)com->to_pfc_data;
91          dest_task = msg->to_task;
92
93          /* if memory not available to hold message then discard it */
94          if (allocate_mem(msg->size,&mem) != OK) {
95
96              /* copy message into memory */
97              memcpy(mem,msg,msg->size);
98
99              /* put message into task message chain */
100             if (task[dest_task].msg_head == LNULL)
101                 task[dest_task].msg_head = task[dest_task].msg_tail = mem;
102             else {
103                 task[dest_task].msg_tail->next = mem;
104                 task[dest_task].msg_tail = mem;
105             }
106             mem->next = LNULL;
107         }
108         com->to_pfc_busy = 0;
109     }
110
111     /* check for output data */
112     if (!com->to_host_busy && out_msg_head != LNULL) {
113
114         /* copy message into buffer */
115         memcpy(com->to_host_data,out_msg_head,out_msg_head->size);
116
117         /* remove message from chain */
118         msg = out_msg_head;
119         out_msg_head = out_msg_head->next;
120
121         /* return message memory to free area */
122         return_mem(msg->size,msg);
123
124         /* mark buffer as busy */
125         com->to_host_busy = 1;
126
127     }
128
129     /* return to task context */
130     if (lock)
131         restore_context(stackptr);
132     else {
133         task[xtask].stackptr = stackptr;
134         restore_context(task[get_ready_task()].stackptr);
135     }
136 }
137 /***/
138 /*****************************************************************************
139 *
140 * GET_READY_TASK()
141 *
142 * Definitions:
143 *   This function loops through the tasks looking for one which is ready
144 *   to run. When found the task number is returned. The daemon task (the
145 *   Executive) is never suspended so this function always manages to find
146 *   a ready task. To avoid spending a lot of time looping and the risk of
147 *   missing interrupts, if an active task is not found after 32 loops then
148 *   the daemon task is returned anyway
149 *
150 * Parameters:
151 *   No input.
152 *
```

```
153     * Returns:
154     *   Number of task to be run.
155     *
156     ****************************************************************************/
157
158     int get_ready_task(void)
159     {
160
161         int i;
162
163         /* loop for a limited period looking for an active task */
164         for (i = 0; i < 32; i++) {
165
166             /* increment task number */
167             if (++xtask > last_task)
168                 xtask = 0;
169
170             /* check task data */
171             #if defined(DEBUG)
172             if (xtask > TASKS
173             ||(task[xtask].state != ACTIVE &&  task[xtask].state != WAITING &&  task[xtask].state != DEAD )
174             || task[xtask].countdown > task[xtask].priority)
175                 fatal_msg("task data corrupted for task %", xtask);
176             #endif
177
178             /* check condition of task */
179             if ((task[xtask].state == ACTIVE)
180             && (task[xtask].sleep == 0L)
181             && (task[xtask].countdown-- == 0)) {
182                 task[xtask].countdown = task[xtask].priority;
183                 return xtask;
184
185             }
186         }
187
188         /* ready task not found - return daemon task */
189         return (xtask = 0);
190     }
191
```

APêNDICE C - O CóDIGO PRINCIPAL DA MAC

```
1   /*****************************************************************************
2   *
3   * MAC.C
4   *
5   * Project:          PFC
6   * Application:      LLC
7   * Module:           LLC Media Access Control (MAC) functions
8   * Update:           10/05/90
9   *
10  *****************************************************************************/
11
12  #pragma check_stack(off)
13
14  #include "..\pfc\general.h"
15  #include "..\pfc\x.h"
16  #include "llc.h"
17  #include "func.h"
18  /***/
19  /*****************************************************************************
20  *
21  * MA_UNITDATA_REQUEST()
22  *
23  * Description:
24  *   This function represents the MA_UNITDATA_REQUEST primitive of the MAC.
25  *   The supplied fd is installed in the specified transmit queue which
26  *   corresponds to the MAC priority specified.
27  *
28  * Parameters:
29  *   fd - fd to put in queue.
30  *   priority - MAC priority to be used (0-7)
31  *
32  * Return:
33  *   None.
34  *
35  *****************************************************************************/
36
37  void ma_unitdata_request(FD *fd, int priority)
38  {
39
40      int queue;
41
42      /* if queue is empty ... */
43      queue = priority / 2;
44      if (tx_tail[queue] == LNULL)
45      {
46
47          /* put fd at start of queue and send start command to TBC */
48          tx_head[queue] = fd;
49          tbc_start(queue,intel_to_mot(fd));
50      }
51
52      /* queue is not empty */
53      else {
54
55          /* link frame to queue */
56          tx_tail[queue]->next_fd = intel_to_mot(fd);
57          tx_tail[queue]->cntl_next_fd |= NFV;
58
59          /* if previous tail is confirmed treat this queue as empty */
60          if (tx_tail[queue]->conf_ind & CFD
61          && tx_tail[queue]->conf_ind & EMP)
62              tbc_start(queue,intel_to_mot(fd));
63      }
64
65      /* update queue tail pointer */
66      tx_tail[queue] = fd;
67  }
68  /***/
69  /*****************************************************************************
70  *
71  * MA_UNITDATA_INDICATION()
72  *
73  * Description:
74  *   This function looks for a received FD in the specified queue. In the process
75  *   any response FDs are removed from the receive queue (they are still connected
76  *   to their transmit FDs).
```

```
77    *
78    * Parameters:
79    *   queue - transmit queue to examine
80    *   fd - pointer which will get the recieved RWR frame.
81    *
82    * Return:
83    *   TRUE or FALSE.
84    *
85    ********************************************************************************/
86
87    int ma_unitdata_indication(int queue,FD **fd_ptr)
88    {
89
90        /* if queue was empty ... */
91        if (rx_head[queue] == rx_dummy[queue]) {
92
93            /* ... if more rx frames added by TBC update queue head */
94            if (rx_head[queue]->conf_ind & NPV)
95                rx_head[queue] = mot_to_intel(rx_head[queue]->next_fd);
96            else
97                return FALSE;
98        }
99
100       /* remove any response FDs */
101       while (rx_head[queue]->fd_control && MMM == RESPONSE) {
102
103           /* if next frame is valid move on to it ... */
104           if (rx_head[queue]->conf_ind & NPV == NPV)
105               rx_head[queue] = mot_to_intel(rx_head[queue]->next_fd);
106
107           /* ... else mark queue as empty and abort */
108           else {
109               rx_head[queue] = rx_dummy[queue];
110               return FALSE;
111           }
112       }
113
114       /* if not last frame in queue ... */
115       if ((rx_head[queue]->conf_ind & NPV) == NPV) {
116
117           /* take frame out of queue */
118           *fd_ptr = rx_head[queue];
119           rx_head[queue] = mot_to_intel(rx_head[queue]->next_fd);
120       }
121
122       /* ... else frame is last in queue */
123       else {
124
125           /* if the last frame then copy it to the reserve fd and link the data buffer */
126           memcpy(rx_dummy[queue],rx_head[queue],sizeof(FD));
127           rx_head[queue] = rx_dummy[queue];
128           *fd_ptr = rx_dummy[queue];
129           rx_dummy[queue] = rx_head[queue];
130       }
131
132       return TRUE;
133   }
134   /***/
135   /********************************************************************************
136   *
137   * MA_UNITDATA_STATUS()
138   *
139   * Desription:
140   *   This function looks in a specified transmit queue for a confirmed FD.
141   *
142   * Parameters:
143   *   queue - transmit queue to examine
144   *   fd - where to return the confirmed fd.
145   *
146   * Return:
147   *   TRUE or FALSE.
148   *
149   ********************************************************************************/
150
151   int ma_unitdata_status(int queue,FD **fd_conf)
152   {
```

```
153         FD *fd, *next_fd;
154
155         /* if queue is empty continue to next queue */
156         if (tx_tail[queue] == LNULL)
157             return FALSE;
158
159         /* if fd is not confirmed ... */
160         fd = tx_head[queue];
161         if ((fd->conf_ind & CFD) == 0)
162             return FALSE;
163
164         /* if there is a next fd ... */
165         if ((fd->cntl_next_fd & NPV) == NPV) {
166
167             /* if next fd not confirmed and present fd is mark as empty then restart queue */
168             next_fd = mot_to_intel(fd->next_fd);
169
170             if ((next_fd->conf_ind & CFD) == 0)
171                 if ((fd->conf_ind & 0X2000) == EMP)
172                     tbc_start(queue,fd->next_fd);
173
174             /* move head to next */
175             tx_head[queue] = next_fd;
176         }
177
178         /* ... this was the last frame */
179         else {
180             tx_tail[queue] = LNULL;
181             tx_head[queue] = LNULL;
182         }
183
184         /* return fd */
185         *fd_conf = fd;
186         return TRUE;
187     }
```

APêNDICE D - O CóDIGO PRINCIPAL DA LLC

```
/*****************************************************************************
*
* LLC.C
*
* Project:          PFC
* Application:      LLC
* Module:           Main LLC task module
* Update:           14/06/90
*
*****************************************************************************/

$pragma check_stack(off)

#include "..\pfc\general.h"
#include "..\pfc\x.h"
#include "llc.h"
#include "func.h"
/***/
/*****************************************************************************
*
* GLOBAL VARIABLES
*
*****************************************************************************/

struct tbc_init_table it;                   /* TBC initialization table */
struct tbc_private_area tbc_pa;             /* TBC private area */
FD *tx_tail[4];                             /* tx queue tail pointers */
FD *tx_head[4];                             /* tx queue head pointers */
FD *rx_head[4];                             /* rx queue head pointers */
FD *rx_dummy[4];                            /* rx queue dummy pointers */
FD *tbc_fd_pool_tail;                       /* tbc free fd pool tail */
BD *tbc_bd_pool_tail;                       /* tbc free bd pool tail */
FD *fd_pool[FD_POOL_SIZE];                  /* llc frame descriptor pool */
BD *bd_pool[BD_POOL_SIZE];                  /* llc buffer descriptor pool */
void *buff_pool[BUFF_POOL_SIZE];            /* receive buffer pool */
unsigned fd_pool_count = 0;                 /* fd pool count */
unsigned bd_pool_count = 0;                 /* bd pool count */
unsigned buff_pool_count = 0;               /* receive buffer pool count */
struct llc_interface_msg *interface;        /* table used as interface between LLC and MMS */
unsigned tI = 0x100;                        /* slot time, defaults to 100 */
unsigned long t3 = INFINITE;                /* transmit state variable timeout */
int tbc_state = RESET;                      /* current state of network controller */
unsigned tsa[3];                            /* this station address */
unsigned hi_pri_tht = 0x500;                /* high priority token hold time */
unsigned tar_rot_4 = 0x500;                 /* priority 4 target rotation time */
unsigned tar_rot_2 = 0x500;                 /* priority 2 target rotation time */
unsigned tar_rot_0 = 0x500;                 /* priority 0 target rotation time */
VSI *vsi_head[8][64];                       /* transmit state variables head pointers */
unsigned vsi_pool_count = 0;                /* state variable pool count */
VSI *vsi_pool[VSI_POOL_SIZE];               /* transmit state variable pool */
unsigned char mac_pri[8] = {                /* conversion table for frame priorities */
    0x00,0x80,0x40,0xc0,
    0x20,0xa0,0x60,0xe0
};
/***/
/*****************************************************************************
*
* MAIN()
*
* Description:
*   This function is the main LLC task function. After initializing the
*   TBC it loops forever checking for messages coming through the Executive,
*   checking for LSDUs coming from the MMS, checking for received
*   frames, checking for confirmed transmit frames, and checking for expired
*   timers.
*
* Parameters:
*   No input.
*
* Return:
*   Never terminates.
*
*****************************************************************************/

void main(void)
{
```

```
77
78          /* initialize the TBC and the LLC data structures */
79          initialize();
80
81          /* loop forever */
82          for (;;) {
83
84              /* check for messages from Service Managemente */
85              check_messages();
86
87              /* check interface with the MMS task for incoming LSDUs */
88              check_llc_interface();
89
90              /* check for received frames */
91              check_rx_queues();
92
93              /* check for confirmed tx frames */
94              check_tx_queues();
95
96              /* check for terminated timers */
97              check_timeouts();
98          }
99      }
100     /***/
101     /*****************************************************************************
102     *
103     * CHECK_MESSAGES()
104     *
105     * Description:
106     *   This function checks for messages sent through the Executive message
107     *   mecanism. These messages normally come from Station Management.
108     *
109     * Parameters:
110     *   None.
111     *
112     * Return:
113     *   None.
114     *
115     *****************************************************************************/
116
117     void check_messages(void)
118     {
119
120         /* skeleton message format */
121         struct message {
122             struct msg_hdr msg_hdr;
123             int type;
124         };
125
126         struct message *msg;
127         int task,status;
128
129         /* test for message */
130         x_receive_message(&task,&msg,&status);
131         if (status == OK) {
132
133             /* call message service routine */
134             switch (msg->type) {
135
136                 /* MA_INITIALIZE_PROTOCOL.request */
137                 case M_INIT_PROT_REQ:
138                     init_prot_req(msg);
139                     break;
140
141                 /* MA_DESIRED_RING_MEMBERSHIP.request message */
142                 case M_INRING_REQ:
143                     ring_member(msg);
144                     break;
145
146                 /* MA_SET_TIMER_LIMITS.request message */
147                 case M_SET_TIMER_REQ:
148                     set_timers(msg);
149                     break;
150
151                 /* MA_GROUP_ADDRESS.request message */
152                 case M_GROUP_ADDR_REQ:
```

```
153                 group_addr(msg);
154                 break;
155             }
156
157             /* return message memory to Executive */
158             x_return_memory(msg->msg_hdr.size, msg);
159         }
160     }
161     /***/
162     /*****************************************************************************
163     *
164     * CHECK_LLC_INTERFACE()
165     *
166     * Definition:
167     *   This function checks the llc interface table for incoming LSDUs which
168     *   represent primitives.
169     *
170     * Parameters:
171     *   None.
172     *
173     * Return:
174     *   None.
175     *
176     *****************************************************************************/
177
178     void check_llc_interface(void)
179     {
180
181         FD *fd;
182         BD *bd;
183         LSDU *lsdu;
184         int i;
185
186         /* garantee exclusive access to the interface table */
187         x_lock();
188
189         /* go through all new LSDUs in input chain  */
190         lsdu = interface->to_llc;
191         while (lsdu) {
192
193             /* mark lsdu as accepted by the LLC */
194             lsdu->hdr.state = ACCEPTED;
195
196             /* allocate BD to the LSDU */
197             if (allocate_bd(&bd) != OK) {
198                 lsdu->hdr.state = CONFIRMED;
199                 lsdu->hdr.status = CCCC_UN;
200             }
201             else {
202
203                 /* configure BD to PDU part of the LSDU */
204                 bd->data_buf = intel_to_mot(&lsdu->pdu);
205                 bd->cntl_offset = LABD | sizeof(struct lsdu_hdr);
206                 bd->buf_len = sizeof(LSDU) + lsdu->hdr.data_size;
207
208                 /* allocate FD to LSDU */
209                 if (allocate_fd(&fd) != OK) {
210                     lsdu->hdr.state = CONFIRMED;
211                     lsdu->hdr.status = CCCC_UN;
212                     recycle_bd(bd);
213                 }
214                 else {
215
216                     /* set FD parameters */
217                     fd->conf_ind = 0;
218                     fd->rx_status = 0;
219                     fd->cntl_next_fd = 0;
220                     fd->first_bd = intel_to_mot(bd);
221                     fd->data_length = lsdu->hdr.data_size + 3;
222                     for (i = 0; i < 3; i++) {
223                         fd->da[i] = lsdu->hdr.mac[i];
224                         fd->sa[i] = tsa[i];
225                     }
226                     fd->lsdu = lsdu;
227                     fd->bd = bd;
228
```

```
229                 /* complete processing depending on service */
230                 switch(lsdu->hdr.service) {
231
232                     case DATA:
233
234                         /* set PDU control byte */
235                         lsdu->pdu.control = 0x03;
236
237                         /* set MAC control byte */
238                         fd->fd_control = (unsigned char) RWNR | mac_pri[lsdu->hdr.priority];
239
240                         /* send to MAC */
241                         ma_unitdata_request(fd,lsdu->hdr.priority);
242
243                         break;
244
245                     case DATA_ACK:
246
247                         /* set PDU control byte */
248                         fd->vsi = get_state(lsdu->hdr.mac,lsdu->pdu.ssap,lsdu->hdr.priority);
249                         lsdu->pdu.control = (unsigned char) 0x03 | fd->vsi->value;
250
251                         /* set MAC control byte */
252                         fd->fd_control = (unsigned char) RWR | mac_pri[lsdu->hdr.priority];
253
254                         /* send to MAC */
255                         ma_unitdata_request(fd,lsdu->hdr.priority);
256
257                         break;
258
259                     case REPLY:
260
261                         /* set PDU control byte */
262                         fd->vsi = get_state(lsdu->hdr.mac,lsdu->pdu.ssap,lsdu->hdr.priority);
263                         lsdu->pdu.control = (unsigned char) 0x13 | fd->vsi->value;
264
265                         /* set MAC control byte */
266                         fd->fd_control = (unsigned char) RWR | mac_pri[lsdu->hdr.priority];
267
268                         /* send to MAC */
269                         ma_unitdata_request(fd,lsdu->hdr.priority);
270
271                         break;
272
273                     case REPLY_UPDATE:
274
275                         /* set P/F bit in PDU control byte */
276                         lsdu->pdu.control = 0x13;
277
278                         /* set C/R bit in PDU SSAP byte */
279                         lsdu->pdu.ssap |= 0x01;
280
281                         /* set MAC control byte */
282                         fd->fd_control = (unsigned char) RESPONSE | mac_pri[lsdu->hdr.priority];
283
284                         /* notify TBC of response PDU */
285                         tbc_set_response(lsdu->pdu.ssap,intel_to_mot(fd));
286
287                         /* notify LLC user that LSDU is confirmed */
288                         lsdu->hdr.state = CONFIRMED;
289
290                         break;
291                 }
292             }
293         }
294
295         /* set pointer to next LSDU in input queue */
296         lsdu = lsdu->hdr.next;
297     }
298
299     /* clear input pointer */
300     interface->to_llc = LNULL;
301
302     /* unlock access to interface table */
303     x_unlock();
304 }
```

```
305  /***/
306  /*****************************************************************************
307  *
308  * CHECK_TX_QUEUES()
309  *
310  * Description:
311  *   This function processes all confirmed FDs in each TBC transmit queue.
312  *
313  * Parameters:
314  *   None.
315  *
316  * Return:
317  *   None.
318  *
319  *****************************************************************************/
320
321  void check_tx_queues(void)
322  {
323      FD *fd,*resp_fd;
324      BD *bd,*resp_bd;
325      LSDU *lsdu;
326      RESP_LSDU *resp_lsdu;
327      int queue;
328
329      /* repeat for each TBC transmit queue */
330      for (queue = 0; queue < 4; queue++) {
331
332          /* process all confirmed FDs in queue */
333          while (ma_unitdata_status(queue,&fd)) {
334
335              /* check that receive FD and BD pools are ok */
336              check_rx_pool();
337
338              /* extract BD and LSDU pointers from FD */
339              lsdu = fd->lsdu;
340              bd = fd->bd;
341
342              /* if TBC confirmation was negative ... */
343              if (fd->conf_ind & NP) {
344
345                  /* return no confirmation status to LLC user */
346                  lsdu->hdr.status = CCCC_IT;
347                  lsdu->hdr.state = CONFIRMED;
348              }
349
350              /* ... else TBC confirmation was positive */
351              else {
352
353                  /* if DL_DATA service ... */
354                  if (lsdu->hdr.service == DATA) {
355
356                      /* no response expected - give success indication */
357                      lsdu->hdr.status = RRRR_NR;
358                      lsdu->hdr.state = CONFIRMED;
359                  }
360                  else {
361
362                      /* extract parameters from response FD */
363                      resp_fd = mot_to_intel(fd->fd_ir);
364                      resp_bd = mot_to_intel(resp_fd->first_bd);
365                      resp_lsdu = mot_to_intel(resp_bd->data_buf);
366
367                      /* if response sequence bit differente then give success indication */
368                      if (fd->vsi->value != (resp_lsdu->pdu.control & 0x80)) {
369
370                          /* invert state variable */
371                          fd->vsi->value ^= 0x80;
372
373                          /* if response LSDU present ... */
374                          if (resp_fd->data_length > 4) {
375
376                              /* fill out response LSDU header */
377                              resp_lsdu->hdr.data_size = resp_fd->data_length;
378                              resp_lsdu->hdr.buffer_size = RX_BUFFER_SIZE;
379
380                              /* tie response LSDU to original transmit LSDU */
```

```
                            lsdu->hdr.resp_lsdu = resp_lsdu;

                            /* check for protocol violation */
                            if (lsdu->hdr.service == DATA_ACK)
                                status_msg("data ack pdu contains data");
                        }
                        else {

                            /* no response LSDU */
                            lsdu->hdr.resp_lsdu = LNULL;

                            /* re-use receive buffer containing response PDU */
                            recycle_buff(resp_lsdu);
                        }

                        /* notify indication primitive to LLC user */
                        lsdu->hdr.status = resp_fd->llc_status;
                        lsdu->hdr.state = CONFIRMED;
                    }

                    /* return response FD and BD to pools */
                    recycle_fd(resp_fd);
                    recycle_bd(resp_bd);
                }
            }

            /* return original FD and BD to pools */
            recycle_fd(fd);
            recycle_bd(bd);
        }
    }
}
/***/
/*****************************************************************************
*
* CHECK_RX_QUEUES()
*
* Description:
*   This function processes all of the received FDs in the four TBC queues.
*   The TBC is programed not to store bad FDs or FDs in which the LSDU has
*   null data. Note that response frames present in the TBC receive queues
*   are removed by the MAC.
*
* Parameters:
*   None.
*
* Return:
*   None.
*
*****************************************************************************/

void check_rx_queues(void)
{

    FD *fd;
    BD *bd;
    LSDU *lsdu;
    int queue;

    /* test each receive queue */
    for (queue = 0; queue < 4; queue++) {

        /* process all received FDs */
        while (ma_unitdata_indication(queue,&fd)) {

            /* make sure that receive FD and BD pools are ok */
            check_rx_pool();

            /* extract parameters from received FD */
            bd = mot_to_intel(fd->first_bd);
            lsdu = mot_to_intel(bd->data_buf);

            /* fill-in LSDU parameters */
            lsdu->hdr.data_size = bd->buf_len;
            lsdu->hdr.buffer_size = RX_BUFFER_SIZE;

```

```
457             /* determine service used to send frame */
458             if (fd->fd_control & RWNR == RWNR)
459                 lsdu->hdr.service = DATA;
460             else if (lsdu->pdu.control & PF == PF)
461                 lsdu->hdr.service = REPLY;
462             else
463                 lsdu->hdr.service = DATA_ACK;
464
465             /* if frame was a L_REPLY.indication ... */
466             if (lsdu->hdr.service == REPLY) {
467
468                 /* if bad response status then convert frame to L_DATA_ACK.indication */
469                 if (fd->llc_status & RRRR != RRRR_OK)
470                     lsdu->hdr.service = DATA_ACK;
471             }
472
473             /* place this LSDU at head of chain */
474             lsdu->hdr.next = interface->to_mms[lsdu->pdu.dsap];
475             interface->to_mms[lsdu->pdu.dsap] = lsdu;
476
477             /* if buffer has reached limit then change lsap status */
478             if (++interface->to_mms_count[lsdu->pdu.dsap] > MAX_LSDU_IN_SAP)
479                 tbc_update_lsap_status(lsdu->pdu.dsap,LSAP_FULL);
480
481             /* recycle fd and bd */
482             recycle_fd(fd);
483             recycle_bd(bd);
484         }
485     }
486 }
487 /***/
488 /*****************************************************************************
489 *
490 * CHECK_TIMEOUTS()
491 *
492 * Description:
493 *   This function checks for expired timeouts that indicate that transmit
494 *   state variables can be deleted. Only one timer is processed. The timer
495 *   id used to identify each timer is a pointer to the state variable.
496 *
497 * Parameters:
498 *   None.
499 *
500 * Return:
501 *   None.
502 *
503 *****************************************************************************/
504
505 void check_timeouts(void)
506 {
507     VSI *vsi;
508     int status;
509
510     /* test whether transmit state variables are required */
511     if (t3 != INFINITE) {
512
513         /* test for timeouts */
514         x_get_timer(&vsi,&status);
515         if (status == OK) {
516
517             /* take state variable out of chain */
518             *vsi->last = vsi->next;
519
520             /* return variable memory to either the pool or to the Executive */
521             if (vsi_pool_count < VSI_POOL_SIZE)
522                 vsi_pool[vsi_pool_count++] = vsi;
523             else
524                 x_return_memory(sizeof(VSI),vsi);
525         }
526     }
527 }
```