Joaquim Norberto Cardoso Pires da Silva

Sistema de Integração de Instrumentos Virtuais

Faculdade de Ciências e Tecnologia da Universidade de Coimbra
Departamento de Física
Julho de 1993

à Dina

Agradecimentos

O autor deseja agradecer a todas as pessoas que directa ou indirectamente contribuíram para a realização deste trabalho de Tese de Mestrado, especialmente ao orientador Prof. Doutor Francisco Cardoso pela orientação, ao Prof. Doutor Morão Dias, responsável pelo Grupo de Controlo e Gestão do Departamento de Engenharia Mecânica, pela amizade e dedicação e aos Eng.os Manuel Carvalho, Jorge Landeck e Augusto Marques pelo apoio, disponibilidade e dedicação desinteressada prestados na revisão do texto.

Índice

1 Introdução

Temos assistido nos últimos tempos a um grande desenvolvimento e especialização da instrumentação electrónica, colocada ao serviço das mais variadas áreas da actividade humana (científica, industrial, militar, etc.), como consequência da crescente complexidade e profundidade dos assuntos estudados. O resultado foi a proliferação de instrumentos dedicados e específicos, aos quais os fabricantes pretendem dar alguma flexibilidade através da introdução de microprocessadores. Traduz-se essencialmente por capacidades de reconfiguração, permitindo a utilização do instrumento em várias situações particulares com exigências próprias (ex.: múltimetro digital com *display* gráfico). É o conceito de flexibilidade, ao nível da utilização e da configuração, que está na base deste trabalho.

Se baseados em microprocessadores, as funções e características dos instrumentos são, em última análise, definidos pelo *software* de aplicação. Parece, portanto, lógico que, ao pretender um instrumento flexível, se deva actuar ao nível do *software*. A única limitação é o *hardware*, isto é, não se podem definir operações que não sejam suportadas fisicamente pelo *hardware*. Este novo conceito de instrumento, no qual as características fundamentais são definidas por *software*, foi introduzido em 1983 por *Truchard* e *Kodosky* da *National Instruments*, com a designação de **Instrumento Virtual**. Para que possa ser posto em prática, é necessário definir uma base *hardware* expansível, permitindo assim a generalidade, bem como *software* de definição e gestão dos Instrumentos Virtuais (que também são programas). Esta generalidade baseia-se num pressuposto: embora os Instrumentos Virtuais (**IV**) sejam essencialmente definidos por *software,* necessitam contudo de *hardware* adequado

que os suporte. Se esse *hardware* for modular e aberto podemos adaptar o sistema a um grande número de situações particulares.

Neste trabalho aborda-se a metodologia a seguir na realização de Instrumentos Virtuais de elevada *performance*, dando particular ênfase ao *software* do Sistema de Instrumentação Virtual (**SIV**). Deixamos o *software* de definição dos Instrumentos Virtuais, tipicamente ao nível de uma estação gráfica, para um trabalho futuro.

Inicia-se o trabalho com um capítulo de revisão, no qual se explora e estabelece formalmente o conceito de **Instrumentação Virtual**, bem como a estratégia que adoptamos na definição do **SIV**. Faz-se uma revisão dos sistemas existentes no mercado, assinalando as características importantes de cada um e que, em certa medida, constituem pontos de referência do **SIV**.

No capítulo III justifica-se a arquitectura do SIV delineada no capítulo anterior. Começa-se por fazer uma breve revisão sobre o *interface* **VME** e a rede *Ethernet*, com o objectivo de, por um lado, introduzir formalmente esses dois meios de comunicação apontando as suas características fundamentais e, por outro lado, reunir argumentos que permitam justificar a arquitectura adoptada para o **SIV**.

No capítulo IV apresenta-se e discute-se o *software* de gestão dos instrumentos virtuais (**SGIV**) suportados pelo **SIV**. Foca-se com especial atenção o algoritmo do *scheduler* e o algoritmo de constituição do *scheduler*, no sentido de estabelecer e discutir as condições em que um determinado processo pode ser aceite. Termina-se o capítulo com a apresentação do protocolo de comunicação que adoptamos para comunicar sobre o **VME**.

O capítulo V é um resumo geral do sistema e das possibilidades que ele pode abrir à exploração laboratorial. Aborda-se o *software* de definição dos **IV** ao nível da estação gráfica, e apontam-se metodologias para a avaliação da *performance* global do sistema. No fim do capítulo faz-se um revisão do trabalho executado, retiram-se conclusões e apontam-se algumas das vias a seguir no futuro.

2 Instrumentação Virtual: uma perspectiva.

2.1 - Instrumento virtual

Todos nós estamos habituados a ver a bancada de trabalho de qualquer engenheiro ou cientista repleta de aparelhos de medida e análise, que estes utilizam para "ver" alguns aspectos daquilo que estão a estudar. Vejamos dois exemplos:

a) Engenheiro-analista de sistemas

Uma das funções de um analista de sistemas é verificar o funcionamento de placas electrónicas que, em alguns casos, ele próprio projectou e simulou. Os aparelhos de medida e análise que usa frequentemente nessa tarefa são, entre outros: o analisador lógico, o multímetro, o osciloscópio, a ponta de prova lógica, o gerador de sinais, etc.

b) Físico nuclear (experimentalista)

Fazer o espectro de uma fonte radioactiva, é uma das tarefas frequentes de um experimentalista nuclear, para a qual se exige um analisador multicanal, com toda a panóplia de módulos electrónicos para tratamento e processamento do sinal do detector, um osciloscópio para visualizar os sinais e, possivelmente, um multímetro para, por exemplo, verificar a tensão de polarização do detector.

Este tipo de situação ocorre frequentemente e nem a tentativa dos fabricantes de instrumentação de basear os seus instrumentos em microprocessadores, o que permite a fácil integração de vários instrumentos num só aparelho, a conseguiu melhorar claramente, dada a variedade de instrumentos, suas especificidades funcionais e operacionais. Os aparelhos deste tipo existentes no mercado são caros e

pouco funcionais, sendo, de uma maneira geral, preteridos em relação aos seus equivalentes individuais.

Neste trabalho, o objectivo primordial são instrumentos laboratoriais e não instrumentos industriais, os quais têm outro tipo de exigências como a robustez mecânica e a facilidade de transporte frequente, perfeitamente dispensáveis num ambiente laboratorial.

Num laboratório de investigação e desenvolvimento (I&D), os dados obtidos de um qualquer objecto em estudo são, mais tarde ou mais cedo, sujeitos a análise no computador. Seria importante, interessante e bastante mais fácil, que esse computador pudesse, de alguma forma, controlar os instrumentos, definir os seus parâmetros, e digamos, o tipo de saídas, utilizando um ambiente gráfico baseado em janelas como o *Windows* da *Microsoft*.

Instrumentos deste tipo, que se baseiam num *hardware* genérico, e cujas características são essencialmente definidas por *software*, recorrendo-se a um ambiente gráfico para definição dos instrumentos, visualização e análise dos resultados obtidos, chamam-se **Instrumentos Virtuais**.

A ideia é a de definir um suporte de *hardware*, a sua arquitectura, capaz de suportar os instrumentos em que estamos interessados, deixando a definição de cada um deles, qual a entrada que ocupa, que tipo de operações a efectuar sobre os dados, que tipo de pré-processamento, que tipo de saída(s) de dados, ..., para o *software* de *interface* com o utilizador.

Um Instrumento Virtual (**IV**) é então um instrumento que não existe fisicamente, daí o nome de virtual, mas que pode ser definido, ou redefinido, facilmente em qualquer altura [SANTORI 90]. Imagine-se um ambiente gráfico no qual seja possível definir facilmente, usando o *rato* por exemplo, um conjunto de operações a realizar sobre valores obtidos, de forma a definir, de uma ou mais entradas do *hardware*, o tipo de saída(s) para os resultados: ficheiro, representação gráfica no monitor, ambos, ..., guardar este *setup* num processo, à boa maneira do **UNIX**, e representá-lo nesse interface por um *icon*. No final teremos vários *icons*, um

para cada processo (que aqui representa um instrumento) que definimos anteriormente, e que poderemos executar a qualquer altura. Neste processo estaria definido, inclusivamente, um *interface* gráfico próprio do instrumento, bem como valores pré-definidos para cada parâmetro que poderiam ser alterados dentro desse ambiente, recorrendo-se a um sistema de menus do tipo *pull-down*.

Ao longo deste trabalho será definido o conceito de Instrumento Virtual, bem como sustentada a arquitectura escolhida para o implementar já apresentada em [MARQUES 92]. O objectivo específico deste trabalho é o de definir e desenvolver em termos de *hardware*, *software* e comunicações o módulo integrador citado no referido trabalho.

2.2 - Soluções actuais

O conceito de Instrumento Virtual, cuja ideia geral se apresenta acima, foi estabelecido ao longo do tempo, de implementação em implementação, de vários fabricantes que seguem várias filosofias e estratégias de desenvolvimento. Será mais facilmente entendido se conhecermos algo pormenorizadamente a estratégia seguida por alguns desses fabricantes na construção das suas soluções. A selecção feita, visa abranger a maioria das opções existentes no mercado, as quais se podem resumir em três tipos:

2.2.1 - Placas de aquisição, análise e controlo para PC

Existem no mercado da especialidade inúmeras placas de vários fabricantes, desenhadas para os diferentes tipos de Computadores Pessoais: PC/XT(Bus de 8 bits, 4 MHz)/AT(Bus ISA-*Industry Standard Architecture*- de 16 bits, 8MHz)/EISA(Extended ISA, 32 bits, 8 MHz), PS/2(MCA-*Micro Channel Architecture*) e os vários Macinstosh (SE, SE30, Série II,...). Estas placas podem ser dedicadas ou não, existindo portanto placas com funções únicas e placas com várias funções, sejam elas de DSP (*Digital Signal Processing*), conversão A/D (Analógico-

Digital) e D/A (Digital-Analógico), entradas e saídas (I/O) digitais, contadores e *timers* (temporizadores digitais programáveis).

Com estas placas os fabricantes costumam apresentar *software* que permite construir aplicações adaptados a cada utilizador, possibilitando-lhe uma eficiente e completa exploração do seu *hardware*. Entre os vários fabricantes deste tipo de tecnologia com ferramentas de *software* de desenvolvimento, como a *Analog Devices*, *Burr Brown*, *Keithley MetraByte/Asyst/Dac*, *Data Translation*, *Omega*, *Scientific Soluctions*, *National Instruments*, ..., selecciona-se a *Keithley* devido a duas *packages* de *software* que estão dentro da filosofia deste trabalho, e a *National Instruments* devido à sua *package* de *software* denominada *LabView*, sendo que da apresentação e discussão desse *software* se podem tirar alguns ensinamentos.

a) **ViewDac**

Esta package de *software* da *Keithley Asyst*, foi uma das primeiras a aparecer para sistemas baseados em processadores 386 e 486 da *Intel* ou compatíveis, e baseia-se num sistema de *menus* e janelas para criar e executar rotinas usando quase exclusivamente o *rato*. Uma aplicação construída com o *ViewDac* permite explorar, sem restrições, a velocidade e capacidade de memória deste tipo de computadores pessoais de 32 bits.

A aquisição de dados pode ser feita à velocidade máxima do *hardware*, o qual pode ser de vários fabricantes (a lista inclui os mais importantes fabricantes de placas de aquisição, teste e controlo) e não somente da *Keithley*, e de várias placas simultaneamente, usando centenas de canais com parâmetros de velocidade e resolução diferentes. Outra das características importantes deste *software* é a tentativa de multitarefa, que permite opções de definição e execução simultânea de aplicações ou execução simultânea de duas ou mais aplicações, etc. É possível ainda criar paineis frontais próprios para cada aplicação apresentando, em biblioteca, funções de aquisição e controlo como, por exemplo, funções de controlo tipo **PID** (*Proportional Integral and Diferential*) e **PWM** (*Pulse Width Modulation*), funções de linearização

dos vários tipos de termopares, etc. As aplicações construidas neste ambiente podem ser guardadas em disco para posterior utilização.

Esta aplicação corre num computador pessoal baseado num processador 80386, com coprocessador 80387, ou num 80486, devidamente equipado de **RAM**, disco e controlador de vídeo.

O *ViewDac* possui algumas características que é importante ter em conta:

1. A definição dos instrumentos é feita sem recorrer a uma linguagem de programação, não exigindo treino nem experiência;
2. Possibilidade de multitarefa, o que permite operações simultâneas de aquisição, controlo e análise;
3. Possibilidade de definir paineis frontais próprios para cada instrumento, à medida e gosto do utilizador.

b) **Easyest**

Esta *package* de *software* tem essencialmente as mesmas potencialidades do *ViewDac*, introduzindo, no entanto, a representação de operações em *icons*, substituindo o ambiente de menus e multi-janelas por um ambiente de uma só janela, onde se distribuem vários *icons* definidos pelo utilizador ou da biblioteca. Embora a introdução de *icons* facilite a utilização do ambiente, deve ser procurado um compromisso entre a utilização de *icons*, menus e janelas, como adiante se verá.

c) **LabView**

O *LabView*, da *National Instruments*, foi desenvolvido por *Jeffrey Kodosky* e *James Truchard* (actuais vice-presidente e presidente da empresa) entre 1983 e 1986 em *Austin-Texas*, junto à universidade da cidade, originalmente para uma base de *hardware,* baseada no *bus* **VXI**, desenhada para funções de teste [SANTORI 90].

Hoje o *LabView* suporta várias bases de *hardware* que vão desde sistemas modulares baseados no *bus VXI*, com interface **GPIB** (*General Purpose Interface Bus*), às placas *Plug-In* para Macintosh.

O que torna mais interessante o LabView não é a sua qualidade gráfica, que permite, por exemplo, bons paineis frontais para os instrumentos, ou qualquer outro tipo de característica relacionada com a utilização dos instrumentos, mas sim a extraordinária simplicidade da definição dos instrumentos. Esta potencialidade resulta da introdução da ideia de Instrumentação como uma hierarquia de Instrumentos Virtuais, na qual todos os Instrumentos Virtuais de um nível hierárquico têm o mesmo tipo de construção.

A definição de instrumentos é feita recorrendo a uma linguagem gráfica denominada **G** [SHU 85], com a qual o utilizador especifica o instrumento como provavelmente o faria com papel e lápis, ou seja, usando um diagrama de blocos. Define-se assim um instrumento interligando blocos, cada um representando um Instrumento Virtual dentro de três níveis funcionais distintos:

Nível 1 - Aquisição de Dados

Neste nível encontram-se os instrumentos de controlo do *hardware*, seja ele baseado no *bus* do Macintosh ou no *bus* **VXI**, das comunicações, *RS232/422/485* ou **GPIB**, *Ethernet*, ..., e da aquisição de dados: aquisição analógica, digital e controlo de contadores e *timers*.

Nível 2 - Análise de dados

Neste nível encontram-se os instrumentos com funções de *DSP*, como o gerador de sinais, filtro digital, análise em tempo e em frequência, com funções estatísticas, como a estatística descritiva, *fitting*, álgebra matricial, etc.

Nível 3 - Saída de Dados

Os instrumentos implementados neste nível incluem as funções de apresentação de dados, gráficos [DAVIS 82], cor, etc, de armazenamento de dados e apresentação dos resultados, como a impressão, a organização de relatórios, os ficheiros **ASCII** e binários, etc.

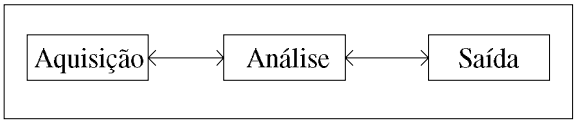


Fig.1 - LabView - Níveis hierárquicos considerados para a definição de Instrumentos Virtuais. Cada um dos níveis é constituído por Instrumentos Virtuais elementares.

Esta hierarquização dos Instrumentos Virtuais, cuja inspiração resultou dos diagramas de blocos que qualquer engenheiro utiliza para visualizar o problema que pretende resolver, e esteve na base da linguagem **G**, não deve ser considerada uma definição de Instrumento Virtual. *Kodosky* e *Truchard*, homens da instrumentação e do *software*, perceberam perfeitamente que um instrumento é, de uma maneira geral, uma associação de pequenos instrumentos individuais (instrumentos elementares) com funções específicas e bem definidas, em que a saída de um é a entrada do seguinte. É o que acontece numa aplicação de *software*, onde se associam rotinas com funções específicas, quiçá existentes em biblioteca ou escritas originalmente para outra aplicação.

Um **Instrumento Virtual** é visto por estes dois autores como um **instrumento cujas funções básicas e capacidades são definidas por software**; eles individualizaram com exactidão as funções básicas a desempenhar, dividiram essas funções por três níveis (fig.1), partindo do princípio de que os instrumentos em que estamos interessados apresentam exigências dentro de cada um desses níveis.

2.2.2 - Sistemas de aquisição, análise e controlo, modulares e fechados

Este tipo de sistemas são apresentados pelos fabricantes como soluções expansíveis, para as quais oferecem vários módulos opcionais, não permitindo, geralmente, qualquer tipo de desenvolvimento por parte do utilizador. São essencialmente sistemas baseados num *bus* próprio do fabricante, não divulgado, com um *interface*[*] próprio ou *standard*. Para estes sistemas, os fabricantes oferecem *software* do tipo do anteriormente descrito que não são mais do que diferentes versões para as referidas bases de *hardware*.

A título exemplificativo, descreve-se sucintamente um sistema deste tipo realizado no Laboratório de Instrumentação e Sistemas de Automação, do Departamento de Física, da Faculdade de Ciências e Tecnologia da Universidade de Coimbra [PIRES 93].

SITAC - Sistema Inteligente de Teste, Automação e Controlo

Este sistema não é propriamente modular, visto que se agrupam numa única placa todos os blocos de memória, **RTC** (Relógio de tempo Real), conversão A/D e D/A, I/O e comunicações, deixando para módulos exteriores apenas as funções de condicionamento de sinais. No entanto, como o sistema está vocacionado para ligação em rede RS485, podem coexistir no mesmo chassis mais do que uma dessas placas.

O sistema baseia-se no *Single Board Computer* **NORMA**, que tem as características apresentadas na Tabela 1, e numa placa **NORBA485** (fig. 2).

Microprocessador	-Hitachi HD64180S08 NPU a 8MHz ou 10MHz
Memória	-EPROM - 32K -RAM - 128K + 128K ou 512K (com pilha de backup e RTC)
Comunicações	-1 canal RS232C - BRmáx=38400 -1 canal RS485 com protocolo HDLC - até 2.2 Mbps
Módulo digital	-16 saídas digitais independentes -16 entradas digitais com acoplamento óptico
Módulo analógico	-16 entradas analógicas independentes ou -8 diferenciais -4 saídas analógicas

[*] Meio de comunicação entre o sistema de aquisição de dados e o computador do utilizador.

LCD	-2 ou 4 linhas
Teclado	-20 teclas
Display de LEDs	-3 dígitos

Tabela 1- Características da **Norma**.

A placa **Norba485** tem 3 objectivos fundamentais:

1. Permitir que o computador hospedeiro possa funcionar como um dos nós de uma rede de topologia *Bus*, com protocolo de sinalização RS485.

2. Libertar o referido computador da tarefa de receber, analisar e estruturar mensagens.

3. Funcionar como *buffer* de mensagens. Optou-se por fazer *polling* à placa em períodos de tempo regulares (*Time-slots*), em vez de usar interrupções. Esta opção representa uma grande simplificação em termos de programação, nomeadamente se pretendermos utilizar um ambiente integrado do tipo do *Microsoft Windows*[*] .

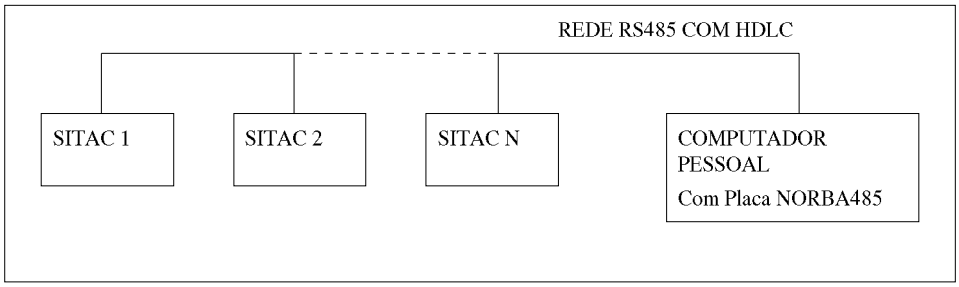


Fig. 2 - Configuração do sistema de aquisição teste e controlo. Cada SITAC é constituído por um ou mais SBC-NORMA, uma fonte e várias placas de terminação de sinais.

Com estas duas placas, **NORBA** e **NORMA**, é possível, por um lado, construir sistemas de sinalização e controlo e, por outro lado, estabelecer uma rede de interligação dos vários sistemas e de um ou mais computadores pessoais que podem funcionar como pontos de definição e vigilância de operações, bem como processamento e arquivo de dados.

A primeira aplicação construída para este sistema foi desenvolvida em colaboração com o Centro de Neuro-Ciências (**CNC**) da Faculdade de Ciências e

[*] Até à versão 3.1.

Tecnologia da Universidade de Coimbra, e constitui uma rede de monitorização e controlo de processos laboratoriais na área das biomedicinas.

2.2.3 - Sistemas de aquisição, análise e controlo , modulares e abertos

Neste grupo estão os sistemas baseados num *bus standard*, perfeitamente definido e disponível, como o **VME** ou o **VXI**, com um *interface* próprio do fabricante ou *standard*. Existem numerosas empresas como a *National Instruments*, *Bruel & Kjaer*, *Omega*, *Keithley Metrabyte*, ..., que apresentam sistemas deste tipo, com *software* de definição de Instrumentos Virtuais do tipo do referido anteriormente. Refiro a abordagem da *National Instrument*s e a da *Bruel & Kjaer* a este tipo de sistema, por duas razões:

1. Solução de *hardware* muito próxima da aqui defendida;

2. *Software* de definição dos Instrumentos Virtuais, de grande qualidade gráfica e enorme simplicidade de utilização.

O conceito de modularidade subjacente, a largura de banda e o sucesso comercial do bus **VME**, fazem com este *standard* seja particularmente atraente como plataforma de desenvolvimento de instrumentação. A popularidade do *interface* **GPIB** (existe no mercado uma enorme variedade de instrumentos **GPIB**) torna-o um modelo em termos de comunicações e/ou protocolos de controlo de instrumentos. As duas empresas referidas reconheceram esta evidência, e em conjunto com outras como a *Hewlett Packard*, *Philips*, *Test & Measurement*, ..., desenvolveram o *standard* **SCPI** (*Standard Commands for Programmable Instrumentation*).

História do GPIB

O *bus* de *interface* **GPIB** foi desenvolvido originalmente pela *Hewlett Packard* com o objectivo de ligar os seus instrumentos programáveis aos seus computadores: era conhecido então por **HP-IB** (*Hewlett Packard Interface Bus*). As suas características, entre as quais se destaca a velocidade de comunicação (até 1Mbyte/s), rapidamente o tornaram popular. Foi em 1975 aceite como um *standard* internacional

denominado **IEEE 488.1**. Sofreu uma revisão em 1987, no sentido de definir com precisão como deveriam comunicar os controladores e os instrumentos, tornando-se no *standard* **IEEE 488.2**. Em 1990, um consórcio de empresas, definiu o **SCPI**, que se pretende seja o futuro *standard* em termos de *interface* com instrumentos. Hoje é conhecido por **GPIB**, visto que é usado por outros fabricantes e não só pela *Hewlett Packard*.

_____

Ao *standard* **GPIB** o consórcio **SCPI** juntou o *bus* **VXI**, cuja especificação combina o *bus* **VME** e o **GPIB**, que pretendem que seja uma plataforma modular que satisfaça as necessidades das aplicações de instrumentação futuras. Tanto a *National Instruments* com a *Bruel & Kjaer*, oferecem uma grande variedade de módulos para sistemas deste tipo com ferramentas de *software* muito avançadas, como é o caso do *LabView* da *National Instruments*.

2.3 - Sistema de Instrumentação Virtual

Impõem-se apresentar a base de *hardware* aqui defendida para suportar os instrumentos virtuais, designado por **Sistema de Instrumentação Virtual (SIV)**.

O sistema (fig. 4) deve permitir a inclusão de *hardware* suplementar conducente à definição de novos instrumentos, seja ele comercial ou desenvolvido pelo utilizador. Um sistema deste tipo, é entendido como uma base de desenvolvimento que inclua *hardware* dedicado a aplicações específicas do utilizador, assim como *hardware* comercial genérico para os instrumentos mais usados, para os quais é desnecessário qualquer desenvolvimento. Está-se, portanto, perante um sistema modular e aberto baseado num *bus standard* comercialmente importante.

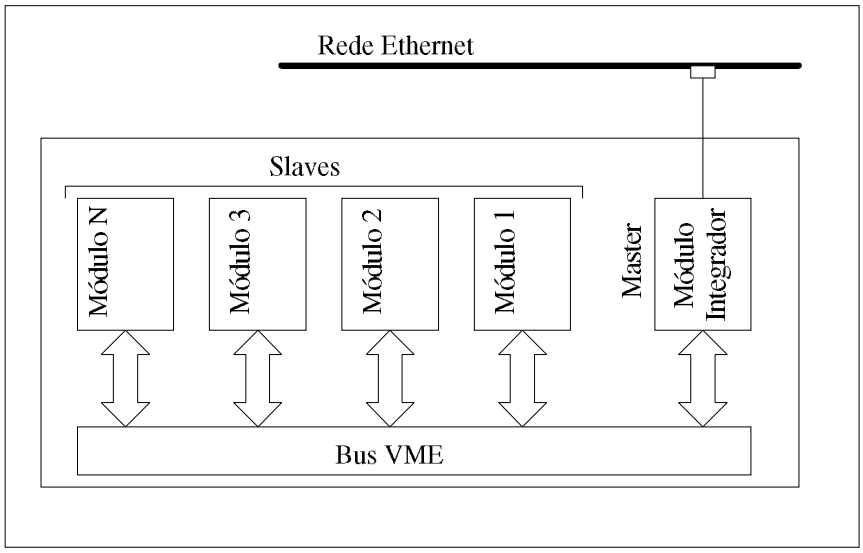


Fig. 4 - SIV, Sistema de Instrumentação Virtual.

A modularidade é aqui fundamental para possibilitar novos desenvolvimentos, aumentar e diversificar o número de instrumentos (resume-se à inclusão de novos módulos e actualização do *software)*, desde que exista um módulo de gestão de todo o sistema responsável pelo mapeamento de processos, pré-processamento e integração dos resultados e gestão das comunicações.

Nesta área da instrumentação o *bus* mais usado é o **VME**, o qual não tem actualmente alternativa, nomeadamente se a existência no mercado de elevado número de produtos para o *bus* é um factor importante a ter em conta.

A definição e visualização dos **IV** é feita recorrendo a uma estação gráfica **UNIX** de elevada performance, ligado ao **SIV** através de rede local *Ethernet*. Esta rede, do tipo **CSMA/CD**, permite uma comunicação rápida e segura entre o **SIV** e a estação de definição e desenvolvimento, de acordo com as exigências das funções de teste e análise dos instrumentos a definir.

A arquitectura anteriormente apresentada tem três pontos críticos os quais podem comprometer a eficiência do **SIV**, pelo que são alvo de uma atenção cuidada. Esses pontos, intimamente ligados ao módulo integrador do sistema, são:

1. Comunicações sobre o *bus* **VME**.

Define-se um protocolo de comunicação sobre o *bus* **VME**, de forma a permitir mensagens compactas, que deverão ser passadas à velocidade do microprocessador do módulo integrador.

2. *Software* do módulo integrador.

É necessário garantir os *deadlines* dos vários processos do *schedule*, apresentando critérios que permitam determinar se um processo pode ser aceite pelo *scheduler* no momento em que é solicitado.

3. Comunicações sobre a rede *Ethernet*.

O acesso à rede *Ethernet* é estatístico, havendo uma probabilidade finita de não ser possível comunicar em condições de elevadas taxas de ocupação da rede. É necessário colocar todas as funções *time-critical* no **SIV**, determinando as condições em que é possível garantir o funcionamento em tempo real do sistema (**SIV** + Terminal gráfico).

Nos capítulos seguintes abordaremos estes três pontos, definindo as condições e os meios que permitem ultrapassar as consequências negativas que teriam na eficiência do **SIV**.

3 Sistema de Instrumentação Virtual

3.1 Introdução

O objectivo deste capítulo é o de justificar a arquitectura do **SIV** anteriormente apresentada. Começa-se o capítulo com uma com uma breve apresentação do *Interface* **VME**, na qual se referem as características mais importantes do *Interface* de interesse para a instrumentação virtual. Em seguida estuda-se a rede *Ethernet* com o objectivo de, por um lado, relembrar os pormenores mais importantes das redes locais (*Local Area Networks* - **LAN**), e por outro lado, rever as teorias que estão na base da comunicação digital de dados. Procura-se ao longo do capítulo, embora de forma breve, uma primeira avaliação dos potenciais pontos de estrangulamento do sistema recorrendo aos resultados dos numerosos estudos realizados sobre o **VME** e a *Ethernet*. Estes dois meios de comunicação entre computadores* têm caracteristicas comuns: ambos partilham a topologia *Bus*, estando cada canal de comunicação, fisicamente representado por um dos computadores da rede, ligado a todos os outros (*Broadcast Channels*). A diferença fundamental entre os dois meios de comunicação reside na vocação de cada um deles, a *Ethernet* para estabelecimento de uma rede local e o **VME** para o estabelecimento de um sistema distribuído. Embora exista bastante sobreposição entre os dois conceitos, eles são distintos estabelecendo-se essa distinção essencialmente ao nível do *software*. O utilizador de uma rede local tem a noção clara de que existem vários máquinas no sistema que usa, pois a utilização dos recursos da rede a partir de uma máquina à qual

* O termo computador . usado neste contexto no sentido de especificar qualquer m quina, seja qual for a sua fun.Æo, que partilhe o mesmo meio de comunica.Æo.

se fez *Login*, exige o endereçamento explícito das outras. Num sistema distribuído, embora existam várias máquinas, tudo é transparente para o utilizador, encarregando-se o *software* de fazer a distribuição de tarefas por cada uma delas.

A arquitectura aqui proposta, explora as potêncialidades destes dois conceitos, através da realização de um sistema distribuído global com dois níveis hierárquicos: o primeiro nível é constituído por uma ou mais estações de definição de **IV**, ligadas por rede local *Ethernet* aos vários sub-sistemas **SIV** que constituem o segundo nível (fig. 5).

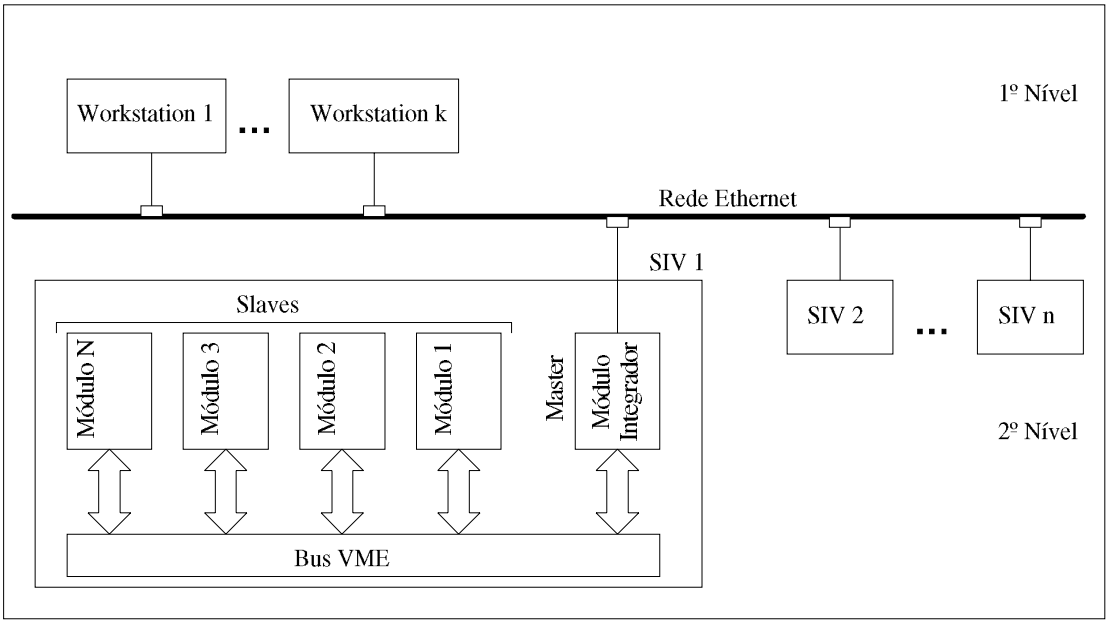


Fig. 5 - Níveis hierárquicos do Sistema Global de Instrumentação Virtual.

3.2 - O *Interface (Bus)* VME

Um *bus* é um meio de comunicação entre módulos independentes de um mesmo sistema (*Closely coupled configuration*). Podemos considerar esses módulos como nós de uma rede, cujo meio de comunicação (*Communication Subnet*) é o próprio *bus*. Num meio de comunicação deste género, tal como numa rede por cabo, só um módulo pode transmitir em determinado momento, recorrendo-se a mecanismos de arbitragem, caso sejam permitidos vários módulos *master*, ou a um modo de funcionamento de um só *master*, o qual controla totalmente o *bus*,

designando a seu tempo qual dos outros módulos (*slaves*) deve transmitir. A especificação do *bus* define um *interface* paralelo entre módulos do mesmo sistema, bem como protocolos que estabelecem, com precisão, a comunicação entre eles.

3.2.1 - Objectivos e estrutura do VME

O *Interface* **VME**, hoje um *standard* largamente aceite e utilizado, foi especificado para atingir os seguintes objectivos principais:

a. Servir de base para a realização de sistemas modulares distribuídos, idealizados com uma configuração *closely coupled*.

b. Definir protocolos de comunicação entre os módulos do sistema.

c. Permitir elevada largura de banda, possibilitando optimização em termos funcionais e/ou de *performance* sem afectar a compatibilidade.

d. Possibilitar o desenvolvimento de sistemas cuja limitação principal seja a tecnologia usada, e não o *interface* de comunicação.

Para atingir estes objectivos, o *interface* **VME** define quatro grupos de linhas (Barramentos ou *buses*) e um conjunto de módulos funcionais opcionais (fig. 6). A cada um dos *buses* do **VME**[*] corresponde uma categoria funcional, as quais em conjunto executam as tarefas específicas permitidas pelo *interface* (Apêndice A).

[*] Chama-se *bus VME* ao conjunto dos quatro *buses* do *Interface* VME, pelo que ambas as designa.äes sÆo usadas quando nos referimos . especifica.Æo VME.
[*] Retirado do livro "The VME Handbook", 2¦ Edi.Æo (VITA).

Fig. 6 - *Buses* e módulos do **VME**.

O interface **VME** possui potencialidades importantes que o tornam bastante atraente para sistemas de elevada eficiência, que resultam das seguintes características:

1. *Bus* de transferência de dados assíncrono de alta velocidade, até 80Mbps, que efectivamente não limita a *performance* de um sistema distribuído baseado no

VME. O ponto crítico do sistema em que estamos interessados pode ser o *software* do *master*, ou módulo integrador, que tentarei obviar através de um algoritmo de mapeamento de processos com prioridades, como adiante se verá, bem como um protocolo de passagem de mensagens adequado.

2. Capacidades de gestão de interrupções. Existem funções prioritárias, *time-critical*, cujos resultados depois de obtidos pelo *slave* devem imediatamente ser recolhidos no *master*, que é também o *interrupt-handler*. O *bus* de interrupções com prioridades do **VME** é adequado para lidar com estas situações.

3. Suporte ao multiprocessamento, permitindo o estabelecimento de um sistema distribuído de elevada eficiência dados os mecanismos que disponibiliza para arbitrar e adquirir o *bus*.

4. Versatilidade. O *interface* **VME** admite vários tipos de módulos, de diferentes *performances* e exigências, permitindo uma gama de opções bastante alargada e de acordo com as necessidades do utilizador.

A opção pelo **VME** parece ser acertada, também porque este *standard* está perfeitamente implantado no mercado. Muitos utilizadores, quiçá a sua grande maioria, não estão preparados nem interessados em desenvolver módulos para o sistema pelo que a existência no mercado de módulos adequados às suas necessidades é um factor importante.

Nota: *Bus* **VXI**

O *bus* **VXI**, **VME** *Extended bus*, adiciona algumas linhas à especificação **VME** através da definição das linhas de utilizador do conector **P2,** e no formato mais popular, adicionando um novo conector **P3**. As transferências de informação são semelhantes às transferências num sistema **GPIB** o qual utiliza um conjunto de comandos para as iniciar (modernamente tal como definidos no SCPI - *Standard Commands for Programmable Instrumentation*). Outras funções incluem dois *clocks* suplementares de 10MHz e de 100MHz, um *Star bus* que permite a ligação em estrela

dos módulos possibilitando comunicação de muito alta velocidade, um *Trigger bus* com sinais de *trigger* **TTL** e **ECL** e protocolos programáveis, um *Local bus* que permite ligação independente entre módulos adjacentes, um *Analog Summing bus* que pode ser usado como rede analógica e um *Module Identification bus* usado para configuração automática do sistema.

A utilização destes novos recursos exige módulos próprios para **VXI**, apesar da compatibilidade com o **VME**, que tornam o *bus*, dada a sua condição de emergente, menos interessante que o **VME** no momento. No entanto, as suas reais potencialidades fazem com que deva ser considerado em sistemas futuros.

3.3 - Ethernet

A rede *Ethernet* foi implementada pela primeira vez na *Xerox Corporation* em 1976 por *Robert Metcalfe* e *David Boggs*, tendo por base a tese de doutoramento de *Metcalfe* apresentada no **MIT** no ano anterior. Tornou-se rapidamente num *standard* industrial e comercial, tendo nessa altura a *Intel* desenhado para ela um *chip* controlador.

O seu modo de funcionamento exige que todas as máquinas ligadas pela rede estejam em permanente monitorização do meio de comunicação (*ether*). Se este estiver livre (*Idle*), qualquer uma das máquinas pode transmitir. Se transmitirem duas ao mesmo tempo, há uma colisão. Nessa situação, ambas suspendem a transmissão, esperam um período de tempo aleatório e tentam novamente. Este processo pode continuar indefinidamente.

Usar a rede *Ethernet*, embora com as limitações inerentes, tal como visto no Apêndice B, não levanta grandes problemas ao **SIV**, e apresenta a vantagem de já existir na maioria dos estabelecimentos de educação, investigação e desenvolvimento. A *performance* do sistema é garantida, se transferirmos para o **SIV**, como previsto, todas as operações *time-critical*. Será, no entanto, de prever algum atraso em instrumentos que exijam apresentação de dados *on-line*, nomeadamente em situações de elevado tráfego. Nessas situações não é possível garantir operações em tempo real.

Estudos realizados pelo Centro de Informática da Universidade de Coimbra sobre a rede *Ethernet* que serve a Universidade permitiram calcular uma taxa de ocupação média da rede na ordem de 1%. Nestas condições, a probabilidade de uma estação, que pretende transmitir, encontrar a rede livre é de cerca de 0.99 pelo que, admitindo uma taxa de transmissão de 10Mbps, é possível garantir operações em tempo real. Os estudos sobre a performance e estabilidade da *Ethernet* [GALLAGER 85] [MEDITCH 83], apontam para atrasos de transmissão preocupantes quando a taxa de ocupação ultrapassa os 60% (fig.7).

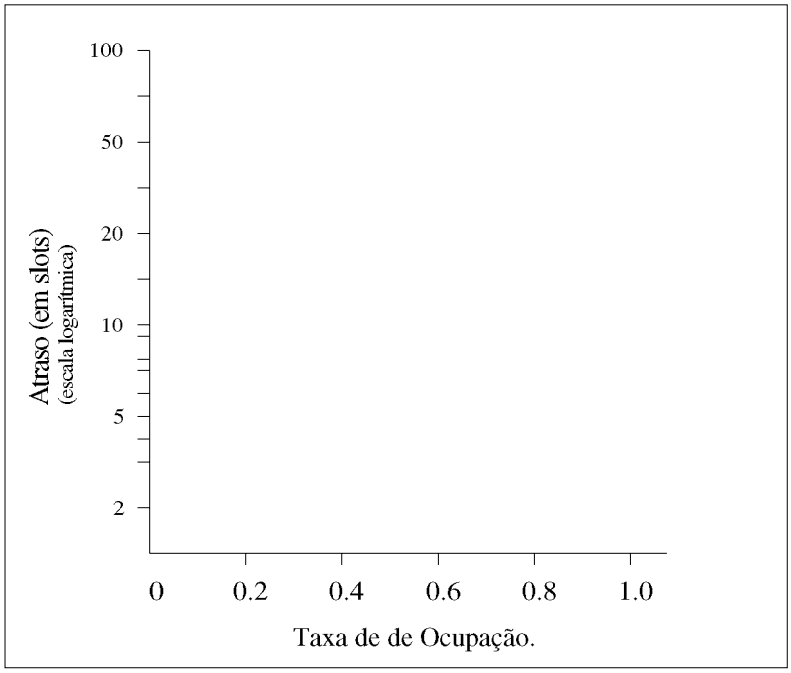


Fig. 7 - Atraso de transmissão, para *frames* de 100 *slots* [MEDITCH 83].

3.4 - Arquitectura Global do Sistema

Ao longo das secções anteriores foi recolhida informação que permite definir e justificar a arquitectura global do sistema. O nosso objectivo aqui, recordo, é construir um sistema de elevada eficiência que suporte o desenvolvimento de **IV**, preferencialmente dentro das áreas do controlo, análise e teste em tempo real. A abordagem de multitarefa (multi-**IV**) que pretendemos fazer, garantida pelo sistema operativo e pelo *software* de aplicação, aponta claramente para um sistema

distribuído. Esse é um ponto pacífico. A única dúvida, neste campo, reside na opção por um sistema modular ou integrado. A resposta é também clara se atendermos ao seguinte:

1. Não é possível definir *hardware* tão genérico que suporte todos os instrumentos actuais e futuros. O sistema deve possibilitar definição de novos instrumentos através da integração de novos desenvolvimentos de *hardware*.

2. A evolução contínua e rápida da electrónica permite constantes melhoramentos do *hardware*, traduzindo-se pela possibilidade de definição de instrumentos mais eficientes. O sistema tem de possibilitar a substituição dos módulos por versões mais actualizadas e eficientes.

3. Um sistema deste tipo pode ser usado por pessoas das mais diferenciadas áreas, em aplicações com especificidades e graus de exigências diferentes. Este facto exige a possibilidade de vários configurações de *hardware* à medida do utilizador e da aplicação.

A modularidade surge, assim, tão somente como uma exigência imposta pelos objectivos que se pretendem atingir.

O meio de comunicação entre os vários módulos, suporte do sistema distribuído, é o **VME**. Nesta altura deve ser clara a opção pelo **VME**, a qual resulta de três factores fundamentais:

1. Performance,

- Bus de transferência de dados de elevada performance, até 80 Mbps.
- Bus de interrupções com prioridades bastante rápido.
- Protocolos de transferências de dados que permitem acessos sequenciais.

2. Versatilidade,

- Várias alternativas de módulos funcionais.
- Largura de banda elevada.

3. Importância comercial,

- Grande número de fabricantes de módulos para **VME**.
- Grande número de produtos para o bus.
- *Standard* largamente usado.

Os **IV** serão suportados, em última análise, pelos vários módulos *slave* do sistema, existindo um módulo *master* responsável pelo controlo do *bus*, das comunicações e, principalmente, da gestão dos vários processos (instrumentos) existentes.

Para definição dos instrumentos, bem como para a apresentação gráfica dos resultados, é usada uma estação gráfica com sistema de operação **UNIX** ligada ao **SIV** por *Ethernet*. A rede *Ethernet* garante, como vimos, comunicações com o nível de *performance* necessário para a realização dessas operações. A utilização de um sistema gráfico baseado no sistema de operação **UNIX** justifica-se por razões de *performance*, qualidade gráfica e capacidade de multitarefa, permitindo o uso simultâneo de vários instrumentos.

É possível, nesta altura, estabelecer algumas das características dos módulos *slaves* e do módulo *master*. O ênfase neste ponto continua a ser posto sobre o *hardware*, sendo o *software* descrito no capítulo seguinte.

3.4.1 - Módulos *slaves*

Genericamente serão módulos de características A32, D32, SEQ e *interrupt-requesters*, usando qualquer uma das linhas de interrupção, de maneira a possibilitar vários esquemas de arbitragem. O interface com **VME** é feito através de *buffers* de leitura e de escrita, nos quais serão colocados os comandos do *master* e os resultados da execução desses comandos, que podem ser *FIFO* ou *Dual-Ported RAM* (fig. 8).

3.4.2 - Módulo *master*

O módulo *master* do **SIV** é, como temos vindo a referir, responsável pela gestão dos **IV**. As operações que lhe estão atribuídas são:

a) Gestão das comunicações com a rede *Ethernet*: os *frames* de definição dos **IV**, provenientes da **EGDIV** (Estação Gráfica de Definição de Instrumentos Virtuais), devem ser interpretados e tratados, originando novos processos cuja execução é também controlada; Os resultados devem ser estruturados em *frames* e disponibilizados para leitura via rede *Ethernet*.

b) Gestão de processos: a função principal é a definição dos processos que suportam um **IV**, o controlo da sua execução, nomeadamente, através da alteração da sua hierarquia e estado (*running*, *ready* ou *blocked*), bem como terminar (*kill*) processos.

c) Controlo do *bus*: o *master* reúne todas as funções de controlador do sistema, incluindo as funções de arbitragem do *bus*, *interrup-handler*, *bus-requester* e *bus timer*.

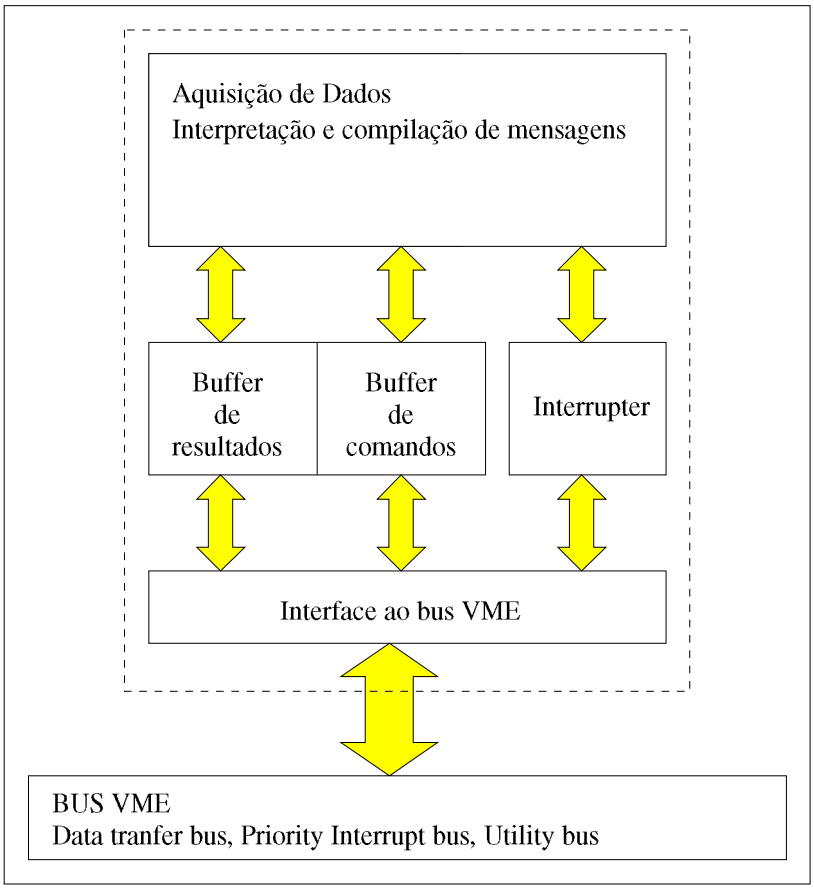


Fig. 8 - Módulo *slave* típico.

Sendo assim, o *master* deve ser um *master* do tipo A32, D32 e SEQ. Não há necessidade de introduzir transferências de 64 *bits*, que obrigariam a multiplexar o *bus* de dados e o *bus* de endereços. As transferências de 32 bits, que podem ter taxas na ordem dos 40 Mbps, garantem largamente as exigências do **SIV**.

Tendo em conta estas caracteristicas, optou-se pela placa de CPU SYS68K/CPU_40 fabricada pela *Force Computers*, que é uma placa de CPU de alta *performance* baseada no microprocessador *Motorola* 68040 a 33Mhz [SYS68K/CPU40 90].

3.5 - Conclusão

Neste capítulo abordou-se, com algum pormenor, a arquitectura do **SIV**. Apresentaram-se argumentos que permitem justificar a opção por um sistema modular, bem como os suportes de integração nos diferentes níveis hierárquicos:*bus*

VME e rede *Ethernet*. Tendo realizado uma alocação funcional em que os instrumentos são realizados fisicamente pelos vários *slaves*, cabendo ao *master* a gestão dos processos (instrumentos) existentes, avançou-se na definição de algumas características dos *slaves* e do *master*. Foi abordado com algum pormenor o protocolo **CSMA/CD**, em primeiro lugar porque as comunicações podem ser um dos pontos fracos do sistema e, em seguida, porque a comunicação de dados, redes e protocolos de comunicação é um dos nossos interesses.

4 Realização de Instrumentos Virtuais

4.1 - Introdução

O objectivo principal deste capítulo é apresentar e discutir o *software* de gestão dos Instrumentos Virtuais (**SGIV**) suportados pelo **SIV**. Inicia-se o capítulo com uma revisão dos conceitos de processo e multiprocessamento, procurando adapta-los à realidade do **SIV**. Esta revisão é fundamental para se perceber a estrutura do *software* de gestão dos **IV**, nomeadamente o algoritmo de escalonamento, *scheduling algorithm*, e a transição entre **IV**, *context/process switch*.

Da análise da estrutura do **SGIV** resultam condições a impor ao *software* de operação desenvolvimento e suporte, e ao *hardware* do módulo de gestão, o *master* definido no capítulo anterior. Estas serão algumas das conclusões importantes deste capítulo. Termina-se o capítulo, definindo mecanismos adequados de passagem de mensagens através do bus **VME**. Estes mecanismos foram definidos tendo em conta que o **SIV** é um sistema distribuído e que a definição dos **IV** é feita na **EGDIV** [TANENBAUM 87] [SHCOEFFLER 84].

4.2 - Processos e multiprocessamento

Os **IV** são definidos essencialmente por *software*, isto é, são programas que correm sobre uma base de *hardware*, cuja arquitectura já definimos. Esses programas incluem, para além do código propriamente dito, alocação de memória, para o código e para os resultados, e valores de determinados registos, contadores e variáveis. Este conjunto de informação denomina-se **processo**.

Os **IV**, tal como os definimos, são processos que temos de construir e gerir, visto que em cada instante existirão vários **IV** em actividade. Este facto exige critérios de partilha da CPU pelos vários processos e escalonamento desses mesmos processos (*Process Scheduling*), ou seja, multiprocessamento [TANENBAUM 87].

Um processo, criado através de uma função especial do sistema operativo, pode estar num de três estados possíveis* (fig. 9):

1. RNG
Running State
- O processo usa a CPU nesse instante.

2. RDY
Ready State
- O processo está temporariamente parado para permitir que outro processo possa correr: espera autorização do *scheduler* para retomar a CPU.

3. BLK
Blocked State
- O processo está bloqueado até receber a informação que espera. Nessa altura passará a **RDY**.

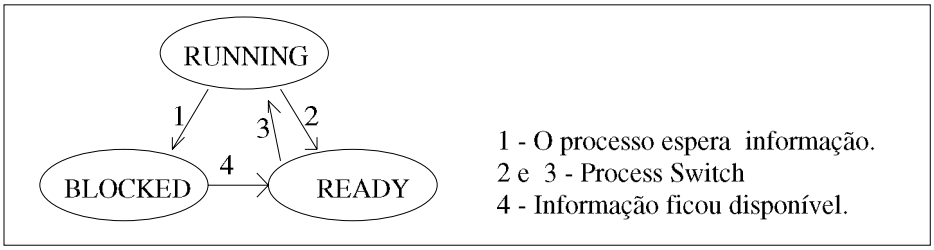


Fig. 9 - Estados de um processo [TANENBAUM 87].

Os vários processos criados partilham a CPU de acordo com o *scheduler*, desde que estejam activos, isto é, no estado **RNG** ou **RDY**. O *scheduler* tem também as funções de *interrupt-handler* e gestor de comunicações entre processos, ou seja, entre aplicações e entre estas e os processos do sistema operativo (fig. 10).

*No inicio, na maioria dos casos, fica no estado RDY.

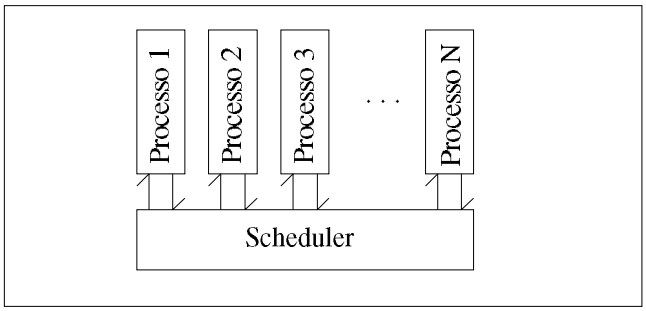


Fig. 10 - Scheduler: constitui a camada mais baixa de um sistema operativo multi-processo (é o núcleo de multiprocessamento).

4.2.1 - Scheduling

Se a determinada altura existem vários processos activos então é necessário decidir qual deles tem acesso à CPU em primeiro lugar e durante quanto tempo. Esta tarefa é desempenhada, como já se viu, pelo *scheduler* do sistema operativo. Neste problema existem dois graus de liberdade,

1. Acesso à CPU por parte de um dos processos activos: são necessários critérios de selecção.
2. Execução completa do processo seleccionado, *Run to Completation*, ou partilha da CPU entre os vários processos activos, *Preemptive Scheduling*.

Estes dois graus de liberdade exigem, dada a sua importância para o **SIV**, uma abordagem mais detalhada.

4.2.1.1 - Algoritmos de Scheduling

Existem vários algoritmos de *scheduling* optimizados para várias aplicações específicas, dos quais saliento:

1. Round Robin

É talvez o mais simples e o mais usado dos algoritmos de *scheduling*. A ideia geral consiste em atribuir a cada processo um intervalo de tempo (*quantum*) no qual ele está autorizado a utilizar a CPU. Ao fim desse tempo a CPU é dada a outro

processo, *Process Switch*. Se o processo terminou ou entrou em estado **BLK**, antes do fim do *quantum*, o *switch* é feito nessa altura. Tudo o que é necessário fazer é manter uma lista actualizada dos processos activos: ao primeiro processo da lista é atribuído um *quantum*, findo o qual o processo é colocado no fim da lista, repetindo-se o procedimento para o processo do topo da lista (fig.11).

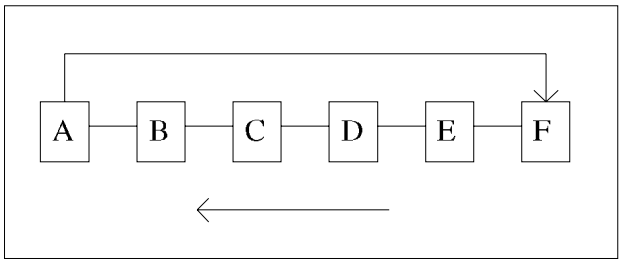


Fig. 11 - Round Robin.

É necessário que o *quantum* não seja demasiado grande, o que conduziria a atrasos significativos entre acessos consecutivos à CPU, nem muito pequeno, evitando assim perda de tempo de processamento devido aos atrasos de *switching*.

Este algoritmo de *scheduling* atribui igual importância a todos os processos. Isso pode ser um inconveniente em determinadas situações nas quais o acesso de determinado processo à CPU é influenciado por eventos externos.

2. Priority Scheduling

A ideia geral é a de identificar processos prioritários atribuindo a CPU ao processo activo de maior prioridade. Este algoritmo, embora intuitivamente mais eficiente, exige alguns cuidados, pois pode conduzir à monopolização da CPU pelo processo mais prioritário. A possibilidade de alterar dinamicamente as prioridades, bem como o agrupamento dos processos em classes de prioridades, no interior das quais funciona um *Round Robin*, são procedimentos usados para evitar essa situação (fig. 12).

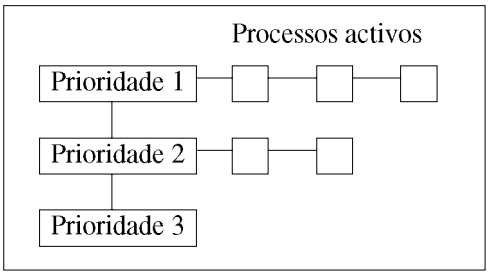


Fig.12 - Algoritmo com 3 prioridades (a prioridade 1 é a de maior prioridade) [TANENBAUM 87].

3. Policy-Driven

Esta fórmula de *scheduling* é bastante interessante, e baseia-se na ideia de atribuir a cada processo um tempo de acesso à CPU proporcional a **Error!** , sendo **n** o número de processos activos.

É necessário registar o tempo de vida de cada processo, ou seja, o intervalo de tempo que decorreu desde que um determinado processo foi criado, o número de processos e calcular o tempo de CPU que deve ser atribuído a cada processo. O que o *scheduler* tem de fazer é calcular o rácio

α = **Error!**

e executar o processo com menor α, enquanto ele se mantiver no fim da lista de valores de α.

Até certo ponto, este algoritmo é um caso especial de um algoritmo com prioridades, os processos criados na mesma altura têm igual probabilidade, e prioridades* dinâmicas, que dependem do tempo de vida de cada processo e do número de processos. Esta visão do problema é bastante interessante para sistemas de tempo real, pois atribui maior prioridade aos processos mais antigos, que por isso devem ser terminados mais rapidamente.

4.2.2 - Comunicação entre processos
IPC - Interprocess Communication

* Os processos criados ao mesmo tempo t.m igual probabilidade.

Os processos necessitam de comunicar com outros processos, possibilitando a realização das mais variadas funções, sendo por isso necessário definir e utilizar mecanismos de comunicação apropriados [TANENBAUM 87] [SHRIVASTAVA 82].

O núcleo do sistema operativo pode ser entendido como um espaço de endereçamento no qual existem os vários processos. Neste espaço, os processos podem comunicar livremente através da consulta de ponteiros para estruturas partilhadas. Essas estruturas podem ser zonas de memória, ficheiros, outros processos, etc. O problema resume-se à gestão do acesso a um recurso comum, nomeadamente quando vários processos o querem utilizar "simultaneamente" (*Race Conditions*). Para evitar esta situação torna-se necessário proibir o acesso a esse recurso quando ele está a ser usado por um processo (*Mutual Exclusion*). O problema pode ser colocado da seguinte forma: Existem determinados recursos comuns. Os processos passam parte do seu tempo a realizar operações que não conduzem a *race conditions*. Quando é necessário utilizar um recurso, dizemos que o processo entrou na região crítica, *critical region*, para esse recurso. O que temos de evitar, é a existência simultânea de mais do que um processo na região crítica para determinado recurso.

Existem várias formas de conseguir a exclusão mútua, usadas nos vários sistemas operativos para tempo real. Vejamos alguns exemplos:

1. Disabling Interrupts

Como o *switching* entre processos é originado por interrupção, uma ideia simples é a de desactivar as interrupções antes de entrar na zona crítica, voltando a activá-las logo à saída. Se for possível, como é na maioria dos casos, desactivar selectivamente as interrupções, esta solução é bastante atractiva dada a sua simplicidade. Para o **SIV** esta solução não apresenta nenhum inconveniente.

2. Lock Variables

A ideia é a de ter uma variável de controlo para cada recurso comum. Um processo antes de utilizar o recurso, verifica a variável de controlo, inicialmente

colocada a zero (*unlock*). Se a variável for zero o processo tem acesso ao recurso, colocando a variável a um (*lock*), não permitindo assim o acesso ao recurso por parte de outros processos. Esta solução não apresenta nenhum inconveniente para o **SIV**, apresentando em relação à anterior a vantagem de não desactivar as interrupções.

3. Strict Alternation

Nesta solução existe uma variável, um *bit* por cada processo, associada a cada recurso comum, que especifica qual dos processos pode ter acesso ao recurso. Um processo para usar o recurso, verifica se é a sua vez, variável a um (*turn*), e utiliza-o nessa eventualidade. Esta solução, assim como a anterior, são do tipo *busy waiting*, pois exigem o teste continuo de uma variável até ela apresentar determinado valor (*turn* ou *unlock*). É no entanto pouco interessante pois dificulta acessos consecutivos ao mesmo recurso, e não conduz a acessos rápidos, visto que a autorização de acesso é dada a cada processo seguindo-se um método do tipo *Round Robin* simples, sem se atender às necessidades efectivas de acesso a esse recurso por parte de cada processo.

4. Semaphores

Esta solução, apresentada por *Dijkstra* em 1965, é uma generalização da ideia de *lock variables*. Existem funções, **P()** e **V()**, que permitem aceder ao recurso comum e proibir o acesso de outros processos, colocando-os em lista de espera. Como as funções **P()**-*Down* e **V()**-*Up* pertencem ao sistema operativo, verificar ou alterar o valor de um semáforo e eventualmente colocar o processo em estado **BLK**, caso o recurso esteja ocupado, é feito com uma simples instrução elementar. Esta é uma técnica para regular o acesso a recursos, usada na maioria dos sistemas operativos para tempo real.

Uma operação **P()** verifica se o semáforo é maior que zero, se for decrementa-o e continua, se não for o processo que pediu a operação **P()** entra em estado **BLK** (*sleep*).

Uma operação **V()** incrementa o valor do semáforo. Nessa situação um dos processos que foi colocado em **BLK**, numa operação **P()** anterior, é colocado no estado **RDY**. A escolha, caso existam vários processos no estado **BLK** na altura da operação **V()** é feita aleatoriamente, por ordem de chegada, etc.

5. Message Passing (Mailbox)

Passar mensagens entre processos é outro dos métodos usados [TANENBAUM 87] [SHRIVASTAVA 82], o qual se resume à utilização de procedimentos (*system calls*) da biblioteca com a seguinte definição genérica:

envia(destino, &mensagem) e

recebe(origem, &mensagem).

Os mecanismos de passagem de mensagens possuem algumas exigências especiais que têm a ver essencialmente com:

1. Evitar perda de mensagens, **2.** Endereçamento e **3.** Segurança (ver **4.5**).

Existem vários mecanismos de passagem de mensagens, desenvolvidos por isto ou por aquilo, sendo o mais interessante aquele em que o método de passagem de mensagens é idêntico ao da chamada a um procedimento [SHRIVASTAVA 82]. Neste caso, um determinado processo quando pretende o serviço de um outro, utiliza o procedimento **envia** para o pedir, espera o fim do serviço e recolhe os resultados com o procedimento **recebe**. O problema aqui reside em conceber um mecanismo que permita vários acessos ao mesmo recurso, sem perda de alguma das chamadas. O mecanismo de **RPC**-*Remote Procedure Calls*, representado na figura 13, utiliza uma nova estrutura de dados chamada **mailbox**, no sentido de acumular chamadas a um mesmo recurso.

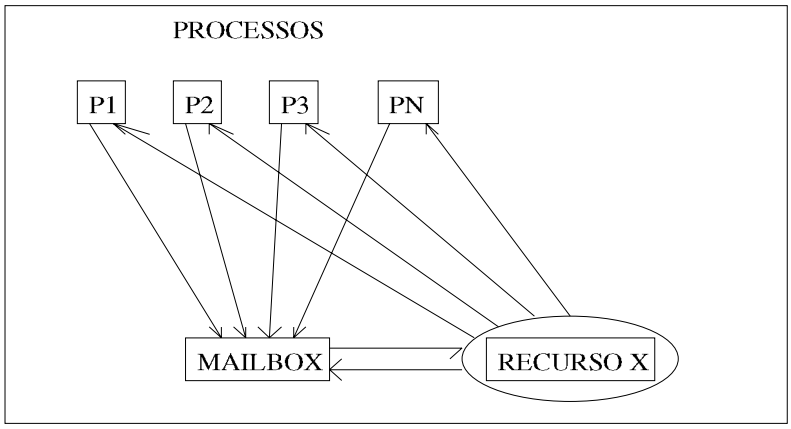


Fig. 13 - Acesso a um recurso: as chamadas são acumuladas na **mailbox**.

As mensagens enviadas pelos processos são *system calls* **envia**. O processo **i** quando pretende usar o recurso **x** procede da seguinte forma:

```
Acesso_a_recurso(x);
{       Repeat
            No_operation;
        Until (testmailbox_recurso(x) == 1);          ⇐ P1
        envia(recurso_x, serviço + parâmetros);
        Wait(recebe(recurso_x, resposta_serviço); } ⇐ P2
```

Esta rotina tem dois pontos passíveis de colocar o processo no estado **BLK** (assinalados com P1 e P2). No ponto P1 testa-se a **mailbox**, que fisicamente é um *buffer*, ficando o processo bloqueado se essa estrutura estiver cheia. Em termos práticos, um processo fica bloqueado, estado **BLK**, quando tenta comunicar com outro cuja **mailbox** está cheia: deve esperar até que uma mensagem seja retirada da **mailbox**. No ponto P2 espera-se pela resposta ao pedido de serviço, ficando o processo bloqueado até essa resposta estar disponível.

Os mecanismos de **RPC** apresentam enormes vantagens de facilidade de utilização, simplicidade e eficiência. Por essa razão, o sistema operativo a utilizar deve permitir essa forma de **IPC**.

4.3 - Sistema operativo

Um sistema distribuído de tempo real, como o **SIV**, apresenta algumas exigências ao nível do sistema operativo (**SO**), nomeadamente na eficiência e flexibilidade da gestão dos processos, tratamento das interrupções, mecanismos de **IPC** e facilidade de reconfiguração. Uma organização monolítica do **SO**, caracterizada por uma certa independência dos vários módulos do sistema operativo, está completamente desajustada para o nosso caso. Os sistemas operativos modernos estão organizados em camadas, tendo por base um conjunto mínimo de funções, a partir do qual se constroem as restantes camadas do **SO** e as aplicações, que se denomina **Núcleo do Sistema Operativo** ou **Kernel**. Nas camadas seguintes do **SO** estão as funções de controlo do I/O e das comunicações, as funções de gestão da memória, etc., que são construidas usando as funções do **Kernel**.

Temos vindo a definir um sistema de instrumentação virtual de elevada eficiência, com as seguintes características fundamentais:

1. Sistema modular e distribuído sobre **VME**, suporte de multiprocessamento, baseado em módulo *master* comercial. A realização dos **IV** é feita ao nível deste sistema.

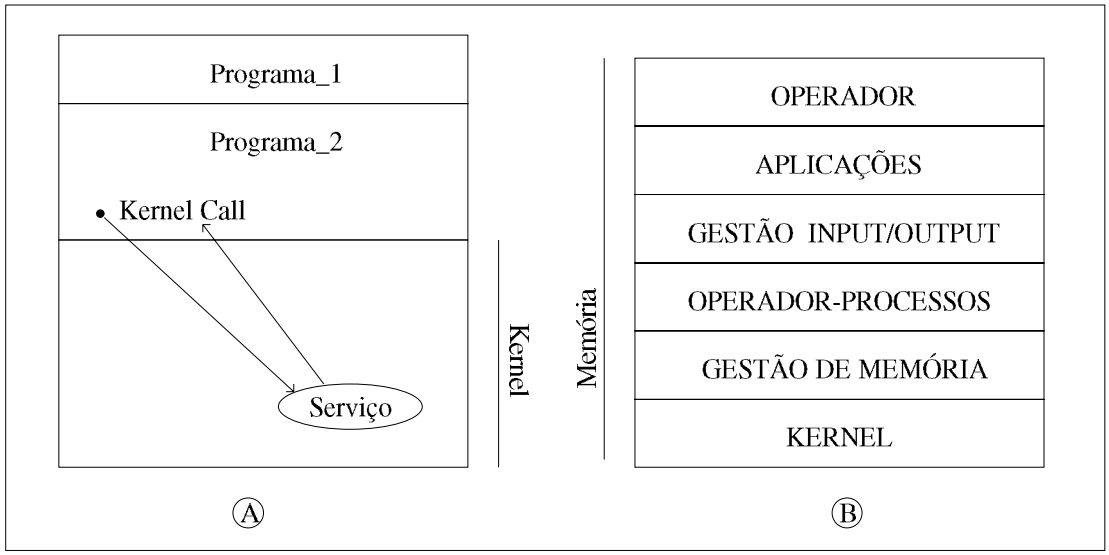


Fig. 14 - Organização de um sistema operativo: A-monolítico, B-camadas [TANENBAUM 87].

2. Definição dos instrumentos em estação gráfica **UNIX**.

3. Utilização da *Ethernet* como rede de interligação entre a estação gráfica e o **SIV**.

Um sistema deste tipo coloca algumas exigências ao **SO**, tanto ao nível do núcleo como das camadas superiores. Abordam-se de seguida alguns desses requisitos.

4.3.1 - Kernel

O **SO** do **SIV**, e mais propriamente o núcleo, não pode ser um factor limitativo do funcionamento do sistema, de acordo com os objectivos e características apresentados no capítulo 3. Sendo assim, o núcleo do **SO** deve responder às seguintes solicitações:

1. Multiprocessamento

Um dos objectivos do **SIV** é o de permitir correr vários instrumentos ao mesmo tempo, várias processos se quisermos, possibilitando ao utilizador informação de várias grandezas e parâmetros do objecto em estudo. Por exemplo, numa aplicação de monitorização e controlo de condições ambientais, em ambiente fechado, existem várias variáveis a monitorizar e algumas a controlar, como a temperatura e a humidade. Uma opção acertada será a definição de dois processos: um para a monitorização e outro para o controlo. O **kernel** deve possuir mecanismos que permitam atribuir a CPU a cada um destes processos, possibilitando que ambos sejam executados.

2. Scheduling

O mecanismo de *scheduling* em que estamos interessados deve ser do tipo *preemptive scheduling*. A ideia de permitir o acesso à CPU de um processo só depois

de o anterior ter completado a sua execução, *Run to Completation*, não nos serve. O **kernel** deve disponibilizar vários algoritmos de *scheduling*, nomeadamente o *Round Robin* e o *Priority Scheduling*. Estes algoritmos permitem, em conjunto, estabelecer grupos de processos de diferentes prioridades, correndo um *Round Robin* entre os processos do mesmo grupo, que têm igual prioridade obviamente.

3. Comunicações entre Processos

As comunicações entre processos são extremamente importantes, visto que uma mesma aplicação utiliza ela própria vários processos, isto é, uma aplicação é realizada pela execução de vários processos, os quais necessitam por isso de trocar informação: pedido de serviços e resultados. Para além disso, o acesso aos processos auxiliares, os recursos comuns, tem de ser controlado para evitar *race conditions*. O **kernel** tem de possuir meios e mecanismos de comunicação entre processos e partilha de recursos comuns.

4. Interrupções

Como existem grandes vantagens em ter funções para processar interrupções, permitindo a introdução de prioridades e redução do tempo de processamento das interrupções, é importante que existam mecanismos de comunicação entre processos e interrupções.

5. Performance

O **kernel** deve estar optimizado para as piores condições de funcionamento, permitindo assim tempos de execução constantes. Uma optimização em condições particulares pode conduzir a tempos de execução bastante mais rápidos em média, mas de gama alargada, isto é, com uma grande dispersão.

4.3.2 - Biblioteca e utilitários

A biblioteca do sistema operativo deve incluir as primitivas de controlo do *hardware*, visto que ele é comercial, e primitivas de comunicação sobre a *Ethernet*.

Estas primitivas, apesar de complexas e difíceis de construir, estão disponíveis no mercado. A existência de primitivas de controlo e exploração do *hardware*, nomeadamente, gestão de memória, controlo do I/O e controlo das comunicações sobre o *bus*, com protocolo de transferência de mensagens incluído por nós, é de fundamental importância para a execução do trabalho em tempo útil. As aplicações devem ser construídas sobre uma base de desenvolvimento conhecida, embora se admita a hipótese de trabalhar ao nível do **SO** incluindo primitivas específicas ou alterando, para as nossas necessidades, as já existentes. O desenvolvimento de aplicações pressupõe a existência de utilitários de desenvolvimento, inspecção e *debuger*. Estes devem ser incluídos no sistema operativo ao nível das camadas superiores do **kernel**.

Nota: VxWorks^{TM}

O *VxWorks^{TM}* é um sistema operativo de tempo real, desenvolvido pela *Wind River Systems Inc.*, organizado em camadas tendo por base um **kernel** reduzido: apresenta somente as capacidades de multiprocessamento, **IPC** e sincronização; tudo o resto é deixado para as camadas superiores. Este **SO** satisfaz completamente os requisitos apresentados anteriormente, tanto ao nível do **kernel** como da biblioteca. Esta inclui primitivas para gestão de memória, TCP/IP, NFS-*Network File System*, RPC-*Remote Procedure Calls*, vários tipos de *timers*, utilitários de monitorização e mais de uma centena de rotinas úteis. Ao nível dos utilitários inclui um *linker-loader* compatível **UNIX**, um compilador C com *debbuger*, etc.

O *VxWorks^{TM}* foi desenvolvido para a generalidade das aplicações em tempo real, e não exclusivamente para o **SIV** (como é óbvio). Atendendo às exigências específicas de cada aplicação nenhum **kernel** pode ser perfeito para cada uma delas. No entanto, deve permitir uma configuração especifica para a aplicação, no sentido de obter a melhor performance possível. O **kernel** do *VxWorks^{TM}* tem capacidades importantes de configurabilidade, através da disponibilização de algoritmos de *queuing* seleccionáveis pelo utilizador.

Apresenta também excelentes registos de *performance*:

- *Process/Context Switch* ⇒ 17 μs
- *Alteração de semáforo* ⇒ 8 μs
- *Interrupt Latency*[*] ⇒ <10 μs

4.4 - Software de gestão dos Instrumentos Virtuais Estrutura e funcionalidade

A realização de Instrumentos Virtuais é essencialmente uma questão de *software*, embora se coloquem, como vimos, algumas exigências em termos de *hardware*, as quais resultam da necessidade de não limitar a construção dos vários instrumentos de análise, controlo e teste em que estamos interessados. Os **IV** são programas que correm sobre esse *hardware* de acordo com regras de *scheduling*, partilha de recursos comuns e gestão temporal bem definidos.

Nesta questão do *software* de realização de Instrumentos Virtuais distinguem-se dois níveis distintos, para além do **SO** (fig.15):

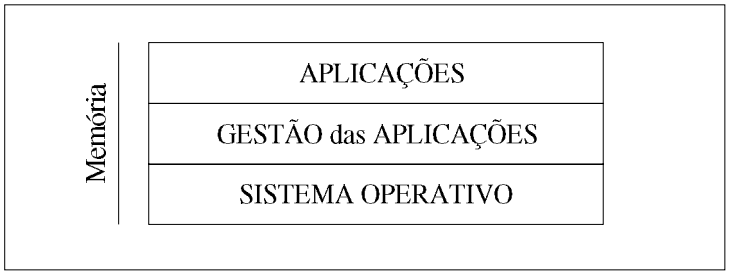


Fig. 15 - Níveis do *software* do Sistema de Instrumentação Virtual.

1. Software de aplicação propriamente dito.

É constituído pelo código dos **IV**, cada um formando um processo. Esse código é essencialmente gerado na **EGDIV**, reservando-se para o **SIV** a tarefa de parametrizar esses comandos com endereços, zonas de memória, definição de mensagens, etc, de acordo com critérios bem definidos. Esta parametrização não deve ser feita na **EGDIV** porque é em grande parte dependente do *hardware*, recorrendo-se

[*] Periodo de insensibilidade a interrup.äes na altura em que se detecta e se dirige uma interrup.Æo.

à consulta de tabelas (actualizadas sempre que se altera o sistema -por exemplo, acrescentando, retirando ou substituindo módulos), que constituem uma pequena base de dados. Referiremos alguns dos aspectos fundamentais desta base de dados, sem lhe dedicar uma atenção individualizada, porque a sua construção e operação não apresenta nada de fundamental.

2. Gestão das aplicações.

Nesta camada actua-se ao nível do sistema operativo, nomeadamente com o *Scheduler* e com a gestão de interrupções. Estas podem ter duas origens principais:

1. Módulos *Slaves* e

2. Comunicação sobre a rede *Ethernet*.

No **SIV**, cada processo (**IV**) tem de terminar dentro de um espaço de tempo pré-definido (*deadline*). Se qualquer um dos processos não termina nesse intervalo de tempo, então dizemos que o **SIV** falhou [TSAI 90] [WILLIAMS 84] [MA 82] É nosso objectivo definir tempos de resposta mínimos que possam ser garantidos pelo **SIV** [LEINBAUGH 80] [ZHAO 87.1] [ZHAO 87.2] [MA 84]. Com este propósito seguimos essencialmente o trabalho de *Leinbaugh* [LEINBAUGH 80] e *Zhao* [ZHAO 87.2] [ZHAO 87.1]. Esses tempos de resposta dependem obviamente do número de processos existentes. É necessário portanto decidir se os pedidos de constituição de novos processos, que continuam a chegar aleatoriamente, podem ser atendidos satisfazendo as suas próprias restrições temporais, bem como as dos processos já constituídos. Este problema foi tratado por *Zhao* [ZHAO 87.1] [ZHAO 87.2], utilizando várias regras heurísticas.

Estes são alguns dos aspectos a tratar nesta secção procurando dar uma ideia geral da estrutura e modo de funcionamento do *software*, particularizando e discutindo os aspectos fundamentais. Basicamente usaremos um algoritmo de *Scheduling* com prioridades dinâmicas [ZHAO 87.1] [ZHAO 87.2] [MA 82] [HYMAN 91], alterando a prioridade dos vários processos (**IV**) de acordo com os *deadlines* respectivos, atribuindo maior prioridade às rotinas de pré-processamento

(*handler*) de interrupções e comunicação sobre a *Ethernet* (fig.16). Assumo que o funcionamento do *timer* associado ao *Scheduler* é assegurado pelo sistema operativo, sobrepondo-se ao próprio *Scheduler*.

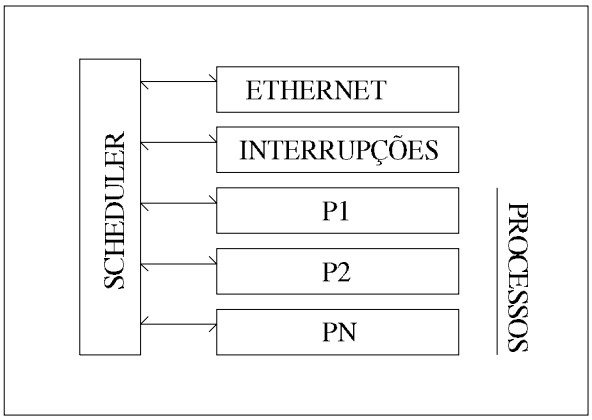


Fig. 16 - Organização de prioridades. Os processos têm prioridades dinâmicas.

4.4.1 - Tempos de resposta (mínimos) garantidos.

Os processos em que estamos interessados, que realizam os **IV**, têm as seguintes características fundamentais:

1. Cada processo é constituído por uma série de comandos (segmentos) que são executados por ordem, um de cada vez. É evidente, são programas.
2. Os processos usam recursos comuns, como por exemplo os módulos do **SIV**, zonas de memória e estruturas de dados, etc.
3. O acesso a esses recursos comuns pode ser feito em modo exclusivo utilizando o conceito de região crítica. De uma maneira geral, um recurso comum pode ser usado no já referido modo exclusivo ou em modo partilhado, no qual vários processos podem usar simultaneamente o recurso. No **SIV** existem recursos usados em modo exclusivo e recursos usados em modo partilhado. Por exemplo, os módulos do **SIV**, de acordo com a arquitectura definida no capítulo 2, são recursos partilhados. Qualquer tipo de estruturas de dados, como p. ex. tabelas, que admitem alteração por parte dos processos, só podem funcionar em modo exclusivo, utilizando-se semáforos para regular o acesso a esses recursos.

4. Cada processo tem um determinado limite de tempo para ser executado, *deadline*. Esse tempo deve ser garantido pelo *Scheduler*. Chama-se *Tempo Garantido do processo i*, $\mathbf{TG_i}$, ao tempo que decorre desde o inicio do processo até à sua execução completa.

Tendo em conta estas características é possível calcular o **TG** de determinado processo.

4.4.1.1 - Cálculo de TG

O cálculo de **TG**, para cada processo, depende dos seguintes factores:

1. O tempo máximo de CPU requerido por cada segmento.
2. O tempo máximo de acesso a um recurso: Aqui estamos interessados nos recursos externos ao módulo *master*, aos quais podemos chamar dispositivos periféricos, como os módulos *slave*, eventual unidade de memória de massa, etc.
3. Existência de regiões críticas, ou seja, recursos usados em modo exclusivo. O recurso é reservado antes de o segmento que o usa ser executado, sendo imediatamente libertado no fim desse segmento. Não se admitem regiões críticas que se estendam a mais que um segmento.

Fig. 17 - Exemplo de dois processos [LEINBAUGH 80]. As cores representam segmentos e a dimensão dos rectângulos é proporcional ao tempo de execução.

O tratamento dos processos, realizado pelo *Scheduler*, deve obedecer às seguintes regras:

1. Segmentos que usam recursos comuns são prioritários.

2. Se a determinada altura existem somente segmentos que não usam recursos, segmentos simples, então o *Scheduling* é feito tendo em conta a cadência de cada processo para os segmentos simples (**CSS_i**). O **CSS_i** é um rácio do tipo do α definido para o algoritmo de *Scheduling Policy_Driven*. Veremos adiante como calcular o **CSS_i**.

3. O acesso a dispositivos periféricos é feito segundo um critério do tipo **FCFS** (*First Come First Served*). Notar que o acesso a um módulo *slave* é feito através de memórias **FIFO**, que realizam esse critério.

4. Um segmento pode usar vários recursos. A reserva dos recursos é feita simultaneamente. O objectivo é o de evitar que o processo fique bloqueado por impossibilidade de acesso a um dos recursos (*deadlock*). Se existe mais do que um pedido de reserva para o mesmo recurso, operações **P()** distintas, é atendido o primeiro a chegar.

O Tempo de Processamento **TP_i** de cada processo depende de eventuais entradas no estado **BLK** e do atraso provocado por outros processos (com **CSS_i** maiores). O tempo de processamento de cada processo deve ser calculado separadamente, para cada um, nas piores condições de funcionamento:

$$\mathbf{TP_i = TTR_i + TTD_i + TTS_i/CSS_i + A_i} \qquad (7)$$

em que **TTR_i** é o tempo total necessário para segmentos que usam recursos, **TTD_i** é o tempo total necessário para segmentos que usam dispositivos periféricos, **TTS_i/CSS_i** é o tempo total necessário para segmentos simples à cadência **CSS_i** e **A_i** é o tempo total perdido por atrasos provocados por outros processos e, por entradas no estado **BLK**. Todos estes tempos são calculados nas piores condições de funcionamento.

Um processo pode ser bloqueado ou atrasado nos seguintes casos:

1. O processo está num segmento simples enquanto outro processo usa a CPU, porque está num segmento que usa recursos: Estes têm geralmente prioridade.
2. O processo precisa de aceder um dispositivo que está a ser usado por outro processo. Por exemplo, é o que acontece quando pedimos um serviço a um módulo *slave* que está a executar um pedido anterior e/ou tem alguns no *buffer* de recepção.
3. O processo pretende iniciar um segmento que usa recursos que estão a ser utilizados. A reserva do recurso, operação **P()**, não é concedida, entrando o processo no estado **BLK**.
4. O processo está num segmento que usa recursos, existindo outros processos activos em segmentos para outros recursos.

De uma maneira geral, os processos, porque eventualmente usam dispositivos e recursos idênticos, podem bloquear ou atrasar os outros processos, isto é, o tempo de execução do processo i é afectado pela existência de outros processos. Essa é a razão pela qual existe a componente $\mathbf{A_i}$ na fórmula (7). Sendo assim, o acesso a um recurso ou dispositivo (esta distinção é feita em termos funcionais, embora os recursos e os dispositivos sejam tratadas da mesma forma) é constituído por duas componentes temporais distintas (fig. 18):

1. Tempo de CPU necessário.
2. Atrasos do tipo 2) e 3) anteriores, que se referem à interferência de outros processos.

Dispositivo ou Recurso	AD_j ou AR_j

Fig. 18 - Acesso a um dispositivo ou recurso.

Potencialmente, um processo j pode ser executado **Error!** vezes no tempo de execução do processo i. O atraso total é então,

$$\mathbf{A_i} = \mathbf{Error!} \qquad (8)$$

em que $\mathbf{ADR_i} = \mathbf{AD_i} + \mathbf{AR_i}$.

Tendo em conta estas considerações temos que,

$$\mathbf{TG_i} = \mathbf{TP_i} = \mathbf{TTR_i} + \mathbf{TTD_i} + \mathbf{TTS_i/CSS_i} + \mathbf{A_i} \qquad (9)$$

com **Error!**. Os vários $\mathbf{TG_i}$ mínimos podem ser calculados estabelecendo as relações de proporcionalidade entre eles, recorrendo-se a uma variável Ψ para definir os vários conjuntos de $\mathbf{TG_i}$. Estas relações de proporcionalidade especificam as velocidades relativas de execução dos vários processos

$$(\mathbf{TG_1},\mathbf{TG_2},\mathbf{TG_3}, ...,\mathbf{TG_n}) = \Psi \cdot (\mathbf{TG_1}\text{ relativo},\mathbf{TG_2}\text{ relativo},\mathbf{TG_3}\text{ relativo}, ...,\mathbf{TG_n}\text{ relativo}) \qquad (10)$$

Para que um determinado conjunto seja solução, é necessário que **Error!**. Este somatório é decrescente com Ψ em primeiro grau pelo que podemos usar Ψ como parâmetro para determinar um conjunto de $\mathbf{TG_i}$, para o qual **Error!** seja o mais próximo de um possível (sem exceder).

Nota: Exemplo e conceito de Overhead do sistema.

No sentido de ilustrar o cálculo de $\mathbf{TG_i}$, apresenta-se de seguida um exemplo com 6 processos que representa o clássico problema "reader-writer" [LEINBAUGH 80] [SHA 90] [SHA 91] (fig. 19).

Processo 1 (P1)

1ms	Dispositivo - D1 5ms	10ms	Dispositivo - D2 20ms

Processo 2 (P2)

1ms	Dispositivo - D1 5ms	10ms	Dispositivo - D3 20ms

Processo 3 (P3)

Dispositivo - D4 20ms	10ms	1ms	Dispositivo - D1 5ms

Processo 4 (P4)

1ms	Dispositivo - D5 5ms	10ms	Dispositivo - D6 20ms

Processo 5 (P5)

1ms	Dispositivo - D5 5ms	10ms	Dispositivo - D7 20ms

Processo 6 (P6)

Dispositivo - D8 20ms	10ms	1ms	Dispositivo - D5 5ms

Fig. 19 - Dois problemas "reader-writer" combinados [LEINBAUGH 80].

Neste exemplo temos, $\mathbf{TTR_i}$=1ms, $\mathbf{TTD_i}$=25ms e $\mathbf{TTN_i}$=10ms (i=1..6). Vamos supor, em primeiro lugar, que os vários $\mathbf{TG_i}$ são iguais, isto é, as relações de proporcionalidade entre eles são todas iguais a um: as respectivas velocidades de execução são idênticas.

Os processos 1, 2, 4 e 5 são semelhantes apresentando $\mathbf{A_i}$ = 2 x 5 + 10ms. Na realidade, o atraso provocado por outros processos é de somente 10ms pois, se considerarmos o processo 1 por exemplo, o acesso a D1 pode ser atrasado uma vez pelo processo 2 e outra pelo processo 3.

Os processos 3 e 6 tem $\mathbf{A_i}$ = 2 x 5 + 10ms. Por exemplo, o processo 3 pode ser atrasado uma vez por P1 e outra por P2, e não duas vezes por cada visto que a reserva de recursos é feita simultaneamente.

Sendo assim temos,

$$\mathbf{CSS_i} = \mathbf{Error!} \qquad (11)$$

Se fizermos Ψ=106 temos $\mathbf{CSS_i}$ = **Error!**, ficando **Error!**=1. O tempo garantido para a execução de cada processo é 106ms. Este tempo deve ser considerado um limite superior.

Vamos repetir o exemplo permitindo que os processos 1, 2 e 3 sejam executados duas vezes mais rapidamente que os processos 4, 5 e 6. Neste caso temos $\mathbf{A_i}$ = 2 x 5 + 10ms para i = 1, 2 e 3, e $\mathbf{A_i}$ = 2 x 2 + 3 x 3 + 10ms para i=4, 5 e 6. Podemos então calcular os $\mathbf{TG_i}$ de acordo com,

$$(\mathbf{TG_1},\mathbf{TG_2},\mathbf{TG_3},\mathbf{TG_4},\mathbf{TG_5},\mathbf{TG_6}) = \Psi \,.\, (\mathbf{1,1,1,2,2,2}) \qquad (12)$$

O somatório dos $\mathbf{CSS_i}$ aproxima-se de 1 para Ψ = 85.8ms, do qual resulta,

$\mathbf{CSS_i}$ = 0.251 e $\mathbf{TG_i}$ = 85.8ms para i = 1, 2 e 3,

$\mathbf{CSS_i}$ = 0.0816 e $\mathbf{TG_i}$ = 171.6 ms para i = 4, 5 e 6.

Até agora temos considerado um sistema operativo ideal, isto é, cujas operações são executadas em intervalos de tempo que podemos desprezar. As operações de inicio de processo, inicio de novo segmento, pedido de dispositivo, pré-processamento de interrupções e sincronização por operações **P()** e **V()**, necessitam de intervalos de tempos que devem ser considerados e adicionados aos tempos totais definidos anteriormente. Para além disso, também o *timer* de alta precisão do *Scheduler* necessita de algum tempo para, p. ex., fazer o *context switch*. Estes custos temporais denominam-se *overhead's* temporais do sistema. Se na equação (9) introduzirmos os *overhead's* e usarmos (10), temos

$\mathbf{CSS_i}$ = **Error!** (13)

em que, $\mathbf{O_i}$ é o *overhead* adicionado aos tempos totais definidos anteriormente, $\mathbf{M_i}$ é o *overhead* máximo para dispositivos periféricos (essencialmente igual ao tempo de insensibilidade a interrupções vezes o número de acessos a dispositivos no processo i), $\mathbf{B\&O_i}$ é atraso total (incluindo *overhead's*) por influência de outros processos e $\mathbf{T\&O_i}$ é o intervalo de tempo (incluindo *overhead's*) que pode ser usado na gestão de interrupções do *timer* (*Scheduling*) durante a execução do processo i. Neste intervalo de tempo devemos incluir o atraso provocado pelo pré-processamento da interrupção da *Ethernet*, que pode ser considerada uma operação interna do *scheduler*.

4.4.2 - Constituição de um schedule

Como já foi referido anteriormente, o número de processos, o tipo de processos e as respectivas restrições temporais não são estáticas no tempo, isto é, de forma aleatória e continua chegam pedidos de constituição de novos processos (**IV**). O problema aqui é o de decidir se determinado processo pode ser constituído,

garantindo o seu próprio *deadline*, bem como os dos outros processos existentes. À organização temporal desse conjunto de processos, sobre a qual se vai debruçar o *scheduler*, chama-se *schedule* [ZHAO 87.1] [ZHAO 87.2] [MA 82]. Devemos determinar se com o novo *schedule*, resultado da introdução de um novo processo, é possível garantir as exigências temporais de todos os processos existentes. Nesse caso, o *schedule* diz-se **fazível**.

Os processos apresentam os seguintes parâmetros característicos:

1. Tempo de processamento, $\mathbf{T_j}>0$.

2. *Deadline*, $\mathbf{DL_j}$.

3. Necessidade de recursos, $\mathbf{R_j}=(\mathbf{R_j}(1), \mathbf{R_j}(2), ..., \mathbf{R_j}(r))$ em que, r é o número de recursos, $\mathbf{R_j}(i)=0$ se $\mathbf{P_j}$ não necessita do recurso $\mathbf{R_j}$, $\mathbf{R_j}(i)=1$ se $\mathbf{P_j}$ necessita $\mathbf{R_j}$ em modo partilhado e $\mathbf{R_j}(i)=2$ se $\mathbf{P_j}$ necessita de $\mathbf{R_j}$ em modo exclusivo.

Um *schedule* **S** consiste num conjunto de fatias temporais $\mathbf{F_k}$, k=1, ..., **NF**, em que **NF** é o número de fatias de **S**. A uma determinada fatia $\mathbf{F_k}$ está associado um instante inicial $\mathbf{IIF_k}$, um comprimento temporal $\mathbf{DF_k}$ e um subconjunto de processos que podem correr durante essa fatia de tempo, isto é, desde $\mathbf{IIF_k}$ a $\mathbf{IIF_k}+\mathbf{DF_k}$. Concretamente, fig. 20, uma fatia temporal é um vector de dimensão r (nº de recursos), cujo elemento de ordem i informa qual o processo que usa o recurso $\mathbf{R_i}$ e em que modo.

Dado um determinado conjunto de n processos, dizemos que um *schedule* está completo se, para todos os valores de j=1, ..., n,

$$Tj \leq \mathbf{Error!} - \delta \cdot K_j \qquad (14)$$

em $\mathbf{K_j}$ é o número de vezes que $\mathbf{P_j}$ é bloqueado para dar lugar a outro processo (*Preemptive Scheduling*) e δ o *overhead* relativo ao tempo de troca (*Switching*) de contexto.

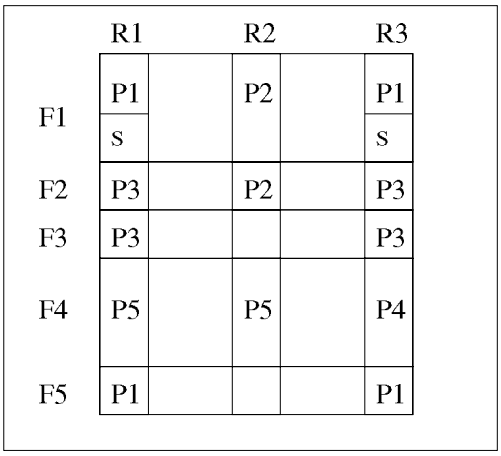


Fig. 20 - Divisão em fatias de um *schedule* [ZHAO 87.1]. Os espaços S são transições entre processos.

Dizemos que um *schedule* é **fazível** se,

$$\text{máx}(\mathbf{IIF_k}+\mathbf{DF_k} : \mathbf{P_j}\in\mathbf{S_k}) \leq \mathbf{DL_j} \qquad (15)$$

isto é, se a ultima fatia temporal contendo $\mathbf{P_j}$ termina antes do *deadline* de $\mathbf{P_j}$.

Pretende-se determinar um *schedule* fazível e completo para os processos existentes, isto é, identificar uma sequência de fatias na qual seja possível garantir o *deadline* de todos os processos. Este é um problema de pesquisa [ZHAO 87.1] [ZHAO 87.2] [MA 82]. O algoritmo de pesquisa que usamos, parte de um *schedule* vazio e percorre o espaço de pesquisa até encontrar um *schedule* fazível. Esse *schedule*, em princípio, não é único, pelo que o algoritmo deve conduzir ao *schedule* óptimo. Se o algoritmo não conduzir a um *schedule* fazível, então não é possível garantir os requisitos temporais do conjunto de processos existentes.

O espaço de pesquisa assemelha-se a uma árvore cujos *vertex*, ligações entre dois ramos, constituem um *schedule* parcial, o qual tem mais uma fatia em relação ao *schedule* que lhe deu origem. O objectivo é o de definir meios que permitam avançar de *vertex* em *vertex*, em direcção a um *schedule* fazível. O critério é o de determinar se dado *vertex* é passível de conduzir a um *schedule* fazível ou não. Dizemos que o *vertex* é **fortemente fazível** [ZHAO 87.1] [ZHAO 87.2], se a resposta for afirmativa. Um *vertex* que não seja fortemente fazível, nunca conduzirá a um *schedule* fazível e completo.

4.4.3 - Algoritmo de pesquisa

Em cada nível de pesquisa deve ser identificado o conjunto de processos que constituirão a respectiva fatia (fig. 21). O sucesso do algoritmo depende da correcta identificação desse conjunto de processos. Os processos são seleccionados de acordo com os seus requisitos temporais e necessidade de recursos. Basicamente, constrói-se uma fatia com um processo principal, a que chamamos primário, e um ou vários processos secundários. O processo primário é escolhido tendo por base exclusivamente considerações temporais. Os processos secundários são escolhidos usando uma função heurística **H**(). Esta função **H**() deve ter em conta requisitos temporais e necessidade de recursos: essa seria, obviamente, uma função **H**() ideal. A função **H**() é aplicada a todos os processos disponíveis. O processo com menor valor de **H**() é seleccionado para a fatia. Repete-se o processo até não ser possível incluir mais processos na fatia. Falaremos mais à frente de algumas possibilidades para **H**(), bem como do desempenho a que conduzem.

O algoritmo usa a variável **INF**, para indicar o inicio da nova fatia. Quando a fatia $\mathbf{F_k}$ é identificada fazemos,

$\mathbf{IIF_k=INF}$ e

$$\mathbf{DF_k=\min(\min(T'_j:P_j\in F_k),\ \min(\eta_h\text{-}INF:P_h\in F_k\ e\ T'_h>0))} \qquad (16)$$

em que $\mathbf{T'_j}$ é tempo de processamento que resta a $\mathbf{P_j}$ e $\mathbf{\eta_h=DL_h-T'_h}$. O primeiro termo de (16) especifica que a duração da fatia não deve exceder o menor valor de $\mathbf{T'_h}$, e o segundo termo especifica que não deve ser maior que $\mathbf{\eta_h}$. Por exemplo, se $\mathbf{DF_k=0}$, então para determinado $\mathbf{P_h\in F_k}$, $\mathbf{\eta_h\text{-}INF=0 \Rightarrow \eta_h=INF}$. O processo $\mathbf{P_h}$ deve ser incluído em $\mathbf{F_k}$ pois, caso contrário, falhará o seu *deadline*. Depois de estabelecida a fatia, o valor de **INF** para a fatia seguinte é colocado a $\mathbf{INF=IIF_k+DF_k}$.

Procedure P_Schedule(task_set:task_set_type; var schedule: schedule_type; var schedulable: boolean);

```
VAR INF: Integer;
      MRDR: vector_type;
      SUBSET:task_set_type;
BEGIN
        schedule:=empty;
        schedulable:=true;
        INF:=current_time;
       REPEAT
        calculate_MRDR(INF,MRDR,task_set);
        IF strongly_feasible(INF,MRDR,task_set,schedule) THEN
           BEGIN
                Let Pf be the task with minimum η e T'f>0;
                SUBSET:={Pf};
                WHILE more tasks can run in paralel with those in SUBSET  DO
                BEGIN
                     apply H() to each candidate task;
                     Let Ps be the task with the minimum value of H() funtion
                     among the candidate tasks;
                     SUBSET:= SUBSET ∪ {Ps};
                END;
                calculate the start time and the length of the new slice;
                apende the new slice to the schedule;
                INF := INF + DFs;
           END ELSE shedulable:= false;
      UNTIL full(schedule, task_set) or Not shedulable;
END;
```

Fig. 21 - Algoritmo de pesquisa e *scheduling* (adaptado de [ZHAO 87.1]).

Em cada nível de pesquisa também se calcula o vector **MRDR** [ZHAO_87.1] [ZHAO 87.2], *Minimum Resource Demand Ratio*, que dá uma indicação das necessidades em termos de recursos de cada um dos processos ainda existentes,

$$\mathbf{MRDR=(MRDR_1, MRDR_2,..., MRDR_r).} \qquad (17)$$

A ideia é a de manter baixos os valores $\mathbf{MRDR_i}$, escalonando imediatamente os processos que usam o recurso i quando o valor $\mathbf{MRDR_i}$ for grande. Se a qualquer altura, o valor de $\mathbf{MRDR_i}$ para o recurso $\mathbf{R_i}$ for maior que um, então não é possível escalonar os processos que restam cumprindo os respectivos *deadlines*.

É possível nesta altura, definir formalmente o conceito de *schedule* fortemente fazível, com a ajuda das estruturas **INF** e **MRDR**. Um *schedule* diz-se **fortemente fazível** se,

1. O vector **MRDR** associado ao *schedule* apresenta $\mathbf{MRDR_i} \leq 1$, para i=1,...,r.

2. $\mathbf{DF_{k-1} > 0}$, isto é, a ultima fatia tem comprimento temporal maior que zero.

Inclui-se de seguida, a título de exemplo, uma lista de regras e funções heurísticas **H**():

1. *Deadline* mínimo $\Rightarrow \mathbf{H(P_j) = DL_j}$.

2. $\boldsymbol{\eta_j}$ mínimo $\Rightarrow \mathbf{H(P_j) = \eta_j}$.

3. Resto de tempo de processamento mínimo $\Rightarrow \mathbf{H(P_j) = T'_j}$.

4. Utilização mínima de recursos.

5. Utilização máxima de recursos.

Zhao, Ramamritham e Stankovic [ZHAO 87.1] [ZHAO 87.2] estudaram e simularam estas e outras funções heurísticas. Os resultados mostram que 1) e 2) conduzem, de uma maneira geral, a melhores resultados que as outras regras, o que nos leva a concluir que os requisitos temporais são os mais importantes.

4.5 - Passagem de mensagens sobre o VME

O sistema que temos vindo a explorar é um sistema distribuído, baseado no *bus* **VME**. É portanto também um sistema modular, cuja estruturação funcional foi definida no capítulo 3. O módulo *master* do sistema é responsável, para além da gestão global do sistema e dos **IV**, pelas comunicações sobre o *bus*. O **VME** disponibiliza, como vimos, meios de endereçar e especificar, ao nível físico, a comunicação entre elementos do sistema. No **SIV**, como em qualquer outro sistema distribuído, é necessário especificar uma sintaxe para as mensagens que passam pelo *bus*. Notar que a comunicação com os módulos é feita através de *buffers* de comunicação. Os módulos, por exemplo, têm uma parte inteligente (um microprocessador) que sequencialmente recolhe uma mensagem do *buffer*, a interpreta e executa.

Basicamente, um *bus* é uma rede de comunicação. O *master* do sistema é o gestor da rede, sendo em princípio único. É possível, do ponto de vista formal, que existam vários módulos *master* no sistema, solução que não utilizaremos. A nossa

estratégia de sistema é a de distribuir funções e operações pelos vários constituintes do sistema. Não faz sentido, à luz desta estratégia, tentar resolver eventuais problemas de saturação ou tentativas de generalidade, através da integração de um outro módulo *master*. A melhor solução, é a de montar um novo sistema que realize essas novas funções (fig.22).

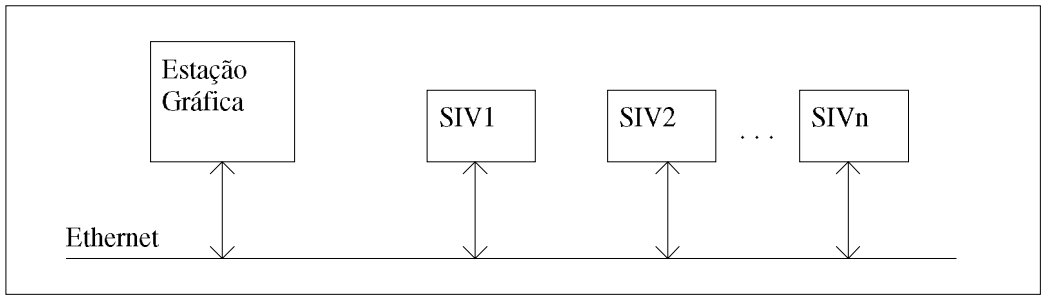


Fig. 22 - Sistema alargado de instrumentação virtual.

A perspectiva de rede é bastante interessante e útil, pois permite utilizar conceitos e conhecimentos sobre redes, que são normalmente meios de comunicação por cabo (comunicação série), na comunicação paralela pelo *bus*.

As mensagens serão organizadas em *frames*, de acordo com uma estrutura que vamos definir, e passadas entre o emissor e o receptor, tendo em conta que:

1. O *master* é o gestor do *bus*, aliás de acordo com a especificação **VME**.

2. As mensagens do módulo *master* são comandos e, eventualmente, validações de comandos. As mensagens dos *slaves* são respostas a comandos (fig.23).

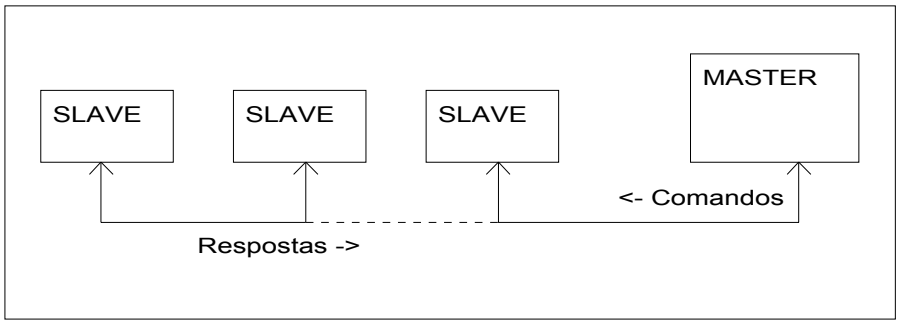


Fig. 23 - Organização funcional das comunicações sobre o **VME**.

3. Os *slaves* não transmitem para o *master*, isto é, não estão autorizados a pedir o *bus*. Quando têm algo a comunicar, colocam a mensagem no *buffer* de respostas, sinalizam o *master* através de uma linha de interrupção, o qual procederá posteriormente à recolha dessa informação. Neste momento, podemos estabelecer um paralelismo com os protocolos de comunicação por cabo, nomeadamente com o **HDLC** - *High Level Data Link Control*. O modo de funcionamento que propomos é um dos modos de funcionamento previsto no protocolo **HDLC**, a que se dá o nome de **NRM** - *Normal Response Mode*. Neste modo, os *slaves* só podem transmitir depois de serem devidamente autorizados pelo *master*.

Estamos perante um protocolo de comunicação de dados com três níveis distintos:

1. Meio de comunicação

Este primeiro nível é constituído pela especificação **VME**, da qual já falamos. É um nível físico que envolve definição eléctrica, temporal e funcional dos vários *buses* do **VME** que é no fundo o nosso meio de comunicação, isto é, o suporte físico da nossa rede.

2. Estrutura das mensagens

Este é um nível de *software*, constituído pela sintaxe que vamos definir para as mensagens. Transmitir uma mensagem implica estruturar a informação de acordo com determinadas regras e, receber significa interpretar essa informação, coloca-la à disposição numa estrutura de dados que constituirá o *buffer* de recepção e, fazer uma primeira verificação de erros de comunicação (erros físicos resultantes de mau funcionamento, perda de sinal, etc.).

Basicamente, uma mensagem deverá ter a seguinte estrutura,

DLMT	NB	IND	CMD	DADOS	DLMT

em que,

DLMT - *Delimitador* (1 *byte*)

- É o delimitador de inicio e fim de mensagem. O interpretador de mensagens deve entender que entre dois campos **DLMT** consecutivos, existe uma mensagem.

NB - *Número de bytes* (2 *bytes*)

- Este campo é usado para fazer algum controlo de erros de comunicação, especificando o número de *bytes* da mensagem. Poderíamos utilizar um método mais rigoroso, como por exemplo, o **CRC**, *Cyclic Redundancy check*, baseado em polinómios de redundância cíclica, o qual seria algo desajustado para a nossa aplicação. A verificação do número de *bytes* é claramente suficiente dadas as particularidades do meio de comunicação, bem como das distâncias envolvidas. Os erros de comunicação resultarão essencialmente de mau funcionamento, ao nível do subsistema físico de comunicações de ambas as partes que comunicam, ou ao nível do próprio meio de comunicação. Estamos a falar de falhas de *hardware*, do próprio *bus*, etc. Para esses erros, o método que propomos não tem nenhum inconveniente em relação a outros (como o **CRC**) que terão eventualmente um processamento mais demorado.

IND - *Índice* (1 *byte*)

- Este campo tem a função de identificar a mensagem. Um *frame* de comando tem um índice que especifica esse *frame*. A resposta a esse comando deve possuir o mesmo índice, para que possa ser identificada como tal. O *master* gere os índices de acordo com uma tabela que pode ter 256 índices (0 a 255). Um índice depois de atribuído a um comando só é disponibilizado quando chega a resposta a esse comando, isto é, quando chega uma resposta com esse índice. Comandos que não exigem resposta têm um índice fixo comum, que está sempre disponível.

CMD - *Comando* (1 *byte*)

- Neste campo é especificado o comando que o *master* pretende que seja executado. Os *slaves* podem executar determinado tipo de operações. Essas operações constituem a tabela de operações desse *slave*. A cada operação é atribuído um código que especifica a operação. É evidente que essa atribuição é feita de comum acordo, isto é, existe um tabela no *master* e uma cópia no *slave*, permitindo assim uma correcta composição e interpretação dos comandos. A tabela do *master* é obtida, a par de outras informações, através do comando de pedido de *setup*, que é executado sempre que se introduz um novo módulo no **SIV**.

DADOS (n *bytes*)

- Neste campo é colocada toda a informação suplementar, isto é, num *frame* de comando são colocados os parâmetros necessários ao comando especificado, e num *frame* de resposta são colocados os resultados da execução do comando.

3. Verificação de sintaxe

Este é também um nível de *software* no qual se verifica se a mensagem faz sentido, isto é, se está sintaticamente correcta e se os vários campos se conjugam entre si. Numa mensagem de comando, deve ser verificado o campo **CMD**, bem como os seus parâmetros, no sentido de detectar comandos inexistentes e parâmetros cujo número ou valor não fazem sentido para o comando especificado. Numa mensagem de resposta é verificado o campo **DADOS**, no sentido de descobrir se os resultados são consistentes com o comando enviado. Outro campo a verificar é o campo **IND**, no sentido de detectar índices disponíveis, isto é, respostas a comandos que não existiram.

4.6 - Conclusão

Neste capítulo foi feita uma revisão dos conceitos e metodologias usados quando se pretende multiprocessamento em máquinas de uma só CPU. Definiram-se os mecanismos apropriados para construir *software* multi-**IV** no **SIV**, nomeadamente mecanismos de *scheduling*, que garantem o *deadline* de cada processo e, comunicações sobre o *bus*. Nos aspectos relacionados com o *scheduling* seguimos, em grande parte, os trabalhos documentados em [LEINBAUGH 80], [ZHAO 87.1] e [ZHAO 87.2]. O estabelecimento e optimização dos mecanismos de *scheduling* propostos nesses trabalhos resultaram de trabalho teórico e de simulação computacional, na qual se recorreu a rotinas próprias para recrear o ambiente típico de um sistema distribuído de tempo real. As nossa conclusões são validadas pelos resultados que obtiveram, embora uma simulação do nosso sistema particular fosse interessante para obter as *performances* possíveis do sistema. É um trabalho a considerar no futuro. A comunicação sobre o *bus* VME é um dos aspectos de fundamental importância para o **SIV**, na qual colocámos alguma da nossa experiência sobre protocolos de comunicação de dados.

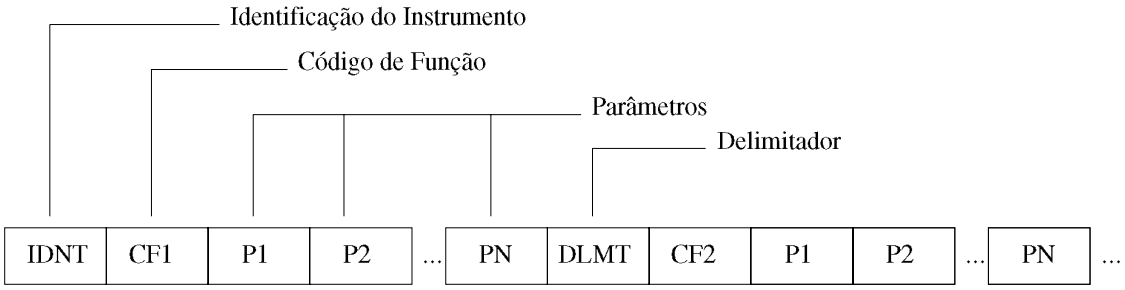
5 Conclusão

5.1 - Introdução

O objectivo principal deste trabalho de tese de mestrado foi o de projectar um sistema de instrumentação virtual. Neste capítulo final fazemos uma reflexão crítica do trabalho realizado, retirando as conclusões que poderão orientar o trabalho futuro na área. Inicia-se o capítulo apontando as possibilidades abertas por este sistema à exloração laboratorial de situações de análise, teste e controlo. Termina-se o capítulo apresentando uma metodologia a seguir na determinação da performance global do sistema.

5.2 - Software de definição dos IV. Uma perspectiva.

O software de definição dos **IV** permite a definição gráfica dos instrumentos através da utilização de menus do tipo *pull-down*, isto é, a funcionalidade do instrumento, painel frontal de comando e de apresentação de dados, tipos de saídas, ..., são definidos sem escrever uma linha de código. Recorre-se a uma linguagem gráfica do tipo da linguagem **G** da *National Instruments*, de forma a que a programação de instrumentos se resuma à associação das funções elementares que o constituem na forma de um diagrama de blocos. A cada bloco, exceptuando os blocos do nível hierarquico de saída, corresponde uma função do **SIV**, isto é, as mensagens enviadas via ethernet para o **SIV** são essencialmente uma sequência de códigos de função e respectiva parametrização.



As mensagens devem incluir também os requesitos temporais do instrumento, isto é, o tempo máximo para que o instrumento complete todas as suas funções. O **SIV** determina se pode ou não garantir esse requesito temporal, aceitando o instrumento em caso afirmativo.

O software de definição dos **IV** definido desta forma sobre uma base de multiprocessamento, tem como vantagem principal a facilidade de definição dos instrumentos, os quais são definidos gráficamente sob a forma de diagramas de blocos em que o utilizador, para além de especificar os blocos, deve definir a interligação entre eles, isto é, parametriza as funções de transferência dos vários blocos e define os fluxos de sinal entre eles - a sua interligação.

5.3 - Avaliação Global do Sistema.

O Sistema Global de Instrumentação Virtual foi definido como um sistema distribuido e modular com dois níveis hierárquicos distintos:

- O primeiro nível tem a função de definição dos **IV**, função que é realizada utilizando uma *workstation* gráfica **UNIX**.
- O segundo nível é constituído pelos sub-sistemas de realização de **IV**, baseados no *bus standard* **VME**.

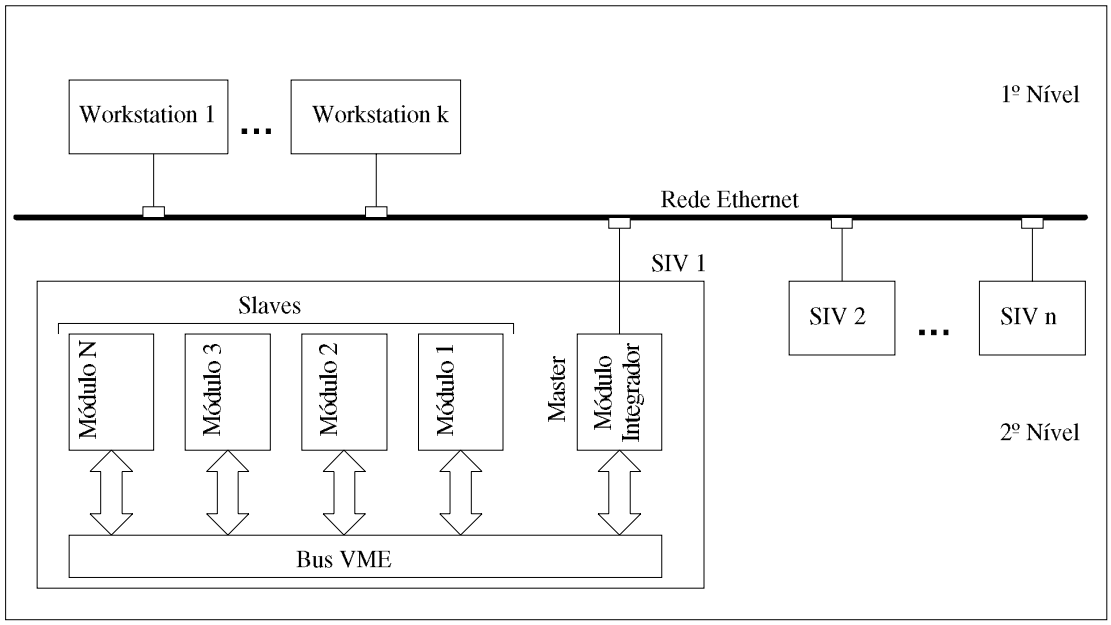
Os **IV** são suportados, em última análise, pelos vários módulos *slave* do sistema, cabendo ao módulo *master* as funções de controlo do *bus*, das comunicações, e principalmente, da gestão dos vários processos (instrumentos) existentes. O estudo

de definição do sistema permitiu identificar os pontos críticos do sistema, os quais discutimos com especial atenção. Esses pontos são:

1. Comunicações sobre a *ethernet*.

2. Algoritmo de *scheduling*. Constituição de um *schedule*.

3. Comunicações sobre o *bus*.



Os resultados dessa discussão permitem estabelecer um critério de avaliação da performance global do sistema, o que se impõe visto que o estudo realizado permite avaliar somente somente cada uma daquelas componentes de forma apenas individual.

Fazer a avaliação global do sistema significa testar o sistema numa gama alargada de situações de funcionamento, especialmente aquelas que contribuem para colocar dificuldades ao nível dos pontos críticos anteriormente referidos. Cada um desses pontos está hoje suficientemente estudado para merecer um tratamento individualizado. No entanto, a forma como contribuem para a performance global do sistema deve ser determinada experimentalmente, nas três situações seguintes:

1. Situação onde são esperados variações na taxa de ocupação da rede *Ethernet*.

2. Situação em que existe um elevado número de utilizadores do sistema.

3. Situação em que os intrumentos definidos usam várias funções realizadas por vários módulos diferentes do **SIV**.

Os resultados destes estudos serão uma medida da performance global do sistema, cujo conhecimento é fundamental para determinar os limites do sistema e estabelecer uma comparação com sistemas actuais.

5.4 - Trabalho futuro.

O trabalho futuro reside essencialmente ao nível da *Workstation* de definição dos **IV** e visualização dos resultados, isto é, será essencialmente um trabalho de desenvolvimento de **software**. Esse trabalho de desenvolvimento do software de definição dos **IV** implica:

1. Desenvolvimento de uma linguagem gráfica de programação de acordo com as exigências já citadas de facilidade, versatilidade e rapidez de programação dos instrumentos virtuais e visualização dos seus resultados.

2. Desenvolvimento de funções elementares genéricas que permitam explorar os módulos existentes. Essas funções constituirão uma biblioteca que será usada pelos futuros utilizadores na definição dos seus instrumentos.

3. Definição de um ambiente gráfico de trabalho que permita o desenvolvimento e *debbug* das aplicações.

Todo este trabalho de desenvolvimento deverá ser antecedido por uma avaliação cuidadosa das possibilidades do sistema, isto é, pela avaliação da sua performance global.

Referências

[CHEN 89] - Jason S. J. Chen e Victor O. K. Li, "Reservation CSMA/CD: A Multiple Access Protocol for LAN's", IEEE Journal on Selected Areas in Communications Fevereiro\89.

[CHLAMTAC 79] - Imrich Chlamtac, William Franta e K. Dan Levin, "BRAM: The Broadcast Recognizing Access Method", IEEE Trans. on Communications Agosto\79.

[DAVIS 82] - Alan Davis e Robert Keller, "Data flow programs graphs", IEEE Computer, Fevereiro\82.

[GALLAGER 85] - Robert G. Gallager, "A Perspective on Multiaccess Channels", IEEE Trans. on Information Theory Março\85.

[GREENE 81] - Edward Greene e Anthony Ephremides, "Distributed Reservation Control Protocols for Random Access Broadcasting Channels", IEEE Trans. on Communications Maio\81.

[HALSALL 92] - Fred Halsall, "Data Communications, Computer Networks and Open Systems", 3º edition, Addison-Wesley, 1992.

[HYMAN 91] - Jay Hyman, Aurel Lazar e Giovanni Pacifici, "Real Time Scheduling with Quality of Service Constraints" IEEE Journal on Selected Areas in Communications Setembro\91.

[KIESEL 83] - Wikhard M. Kiesel e Paul J. Keuhn, "A New CSMA/CD Protocol for Local Area Networks with Dynamic Priorities and Low Collision Probability", IEEE Journal on Selected Areas in Communications Novembro\83.

[LEINBAUGH 80] - Dennis W. Leinbaugh, "Guaranteed Response Times in a Hard-Real-Time Environment", IEEE Trans. on Software Engineering Janeiro\80.

[MA 82] - Richard Perng-Yi Ma, Edward Lee e Masahiro Tsuchiya, "A Task Allocation Model for Distributed Computing Systems", IEEE Trans. on Computers Janeiro\82.

[MA 84] - Richard Perng-Yi Ma, "A Model to Solve Timing-Critical Application Problems in Distributed Computers Systems" Janeiro\84.

[MARQUES 92] - Augusto Marques, "Um sistema de alta eficiência para suporte a Instrumentos Virtuais", Tese de Mestrado, Dept. de Física da U. Coimbra, Outubro\92.

[MEDITCH 83] - James S. Meditch e Cin-Tau A. Lea, "Stability and Optimazation of the CSMA and CSMA/CD Channels", IEEE Trans. on Communications Junho\83.

[PIRES 93] - J. Norberto Pires, "Sitac-Descrição Geral do Sistema", Encontro Nacional de Automação - CENERTEC, Maio\93.

[SANTORI 90] - Michael Santori, "An instrument that isn´t really", IEEE Spectrum, Agosto\90.

[SHA 90] - Lui Sha, Ragunathan Rajkumar e John P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization" Setembro\90.

[SHA 91] - Lui Sha, Ragunathan Rajkumar, Sang Hyuk Son e Chun-Hyon Chang, "A Real-Time Locking Protocol" IEEE Trans. on Computers Julho\91.

[SHCOEFFLER 84] - James Shcoeffler, "Distributed Computer Systems for Industrial Process Control", IEEE Computer Fevereiro\84.

[SHRIVASTAVA 82] - S. K. Shrivastava e F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism", IEEE Trans. on Computers Julho\82.

[SHU 85] - Nan Shu, "Visual programing language: A perspective and a dimensional analysis", IEEE Computer, Agosto\85.

[SYS68K/CPU40 90] - Force Computers, 8.90/20.0/A1/Rev. 1, 1990.

[TANENBAUM 87] - Andrew S. Tanenbaum, "Operating Systems- Design and Implementation", Printice-Hall International Editions 1987.

[TANENBAUM 89] - Andrew S. Tanenbaum, "Computer Networks", 2º edition, Printice-Hall International Inc., 1989.

[TSAI 90] - Jeffrey Tsai, Kwang-ya Fang e Horng-Yuan Chen, "A Noninvasive Architecture to Monitor Real-Time Distributed Systems" Março\90.

[WILLIAMS 84] - Theadore Williams, "The Development of Reliability in Industrial Control Systems", IEEE Micro Dezembro\84.

[ZHAO 87.1] - Wei Zhao, Krithi Ramamritham e Jonh Stankovic, "Preemptive Scheduling Under Time and Resource Constraints", IEEE Trans. on Computers Agosto\87.

[ZHAO 87.2] - Wei Zhao, Krithi Ramamritham e Jonh Stankovic, "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems", IEEE Trans. on Software Engineering Maio\87.

Apêndice A

O Bus VME: Notas fundamentais.

A.1. Introdução

Os objectivos do *Bus* **VME** apresentados na secção 3.2.1 conseguem-se através da definição de quatro grupos de linhas (Barramentos ou *Buses*) e um conjunto de módulos funcionais opcionais. O objectivo deste apêndice é o de fazer uma breve revisão sobre os vários tipos de *buses* e módulos do **VME**.

A.2. Transferência de dados

Os módulos do sistema comunicam através do *bus* de transferência de dados (*Data Transfer Bus* - **DTB**), que contem o *bus* de Dados, o *bus* de Endereços e o *bus* de Controlo. O número de linhas usadas em cada um desses *buses* depende da opção utilizada:

Slave D8 - O *slave* só pode ler ou escrever 8 *bits* de cada vez utilizando as linhas D0-D7.

Master D8 - O *master* pode escrever ou ler das linhas D0-D7 ou D8-D15, mas só 8 *bits* em cada transferência.

Master* ou *Slave D16 - São permitidas transferências de 8 ou 16 *bits*.

Master* ou *Slave D32 - São permitidas transferências de 32 *bits*, sendo necessário um *bus* em modo expandido.

Master ou Slave D64 - São permitidas transferências de 64 *bits*, usando D0_D32 e A0-A31 que são multiplexadas.

Master A16 - Coloca somente 16 linhas de endereço (*Short Adress*) e códigos AM apropriados (*Short Adress Modifier Codes*).

Slave A16 - Descodifica 16 linhas de endereço e responde somente a códigos AM próprios.

Master A24 - Coloca 16 ou 24 linhas de endereço (*Short ou standard Adress*) de acordo com os códigos AM apresentados (*Short or Standard Adress Modifier Codes*).

Slave A24 - Descodifica 16 ou 24 linhas de endereço de acordo com os códigos AM apresentados.

Master e Slave A32 - Permite a utilização de 32 endereços, o qual é especificado pelo código AM apresentado. Esta opção exige o modo expandido.

Master e Slave A64 - Permite a utilização de 64 endereços, o qual é especificado pelo código AM apresentado. Esta opção exige a utilização das linhas A0-A31 e D0-D31 que são multiplexadas.

Master BTO(n) - Especifica um *master* que é capaz de gerar o seu próprio *Time-Out*, depois de **n** nanosegundos sem obter resposta do *slave*.

Master SEQ - Permite o pedido de acesso sequencial.

Slave SEQ - Permite resposta a um acesso sequencial.

Nota - Os códigos AM permitem que e *master* envie informação adicional para o *slave*, existindo códigos com funções definidas, códigos destinados a funções a definir pelo utilizador e códigos reservados aos futuros desenvolvimentos do *interface*.

No sentido de rever e ilustrar o funcionamento do **DTB**, apresenta-se na fig. A1 um ciclo típico de leitura de um *byte*. A figura foi tirada do livro "The VMEbus Handbook", 2ª edição (VITA).

A.3. Árbitro

A existência de vários módulos *master* exige meios de pedir e aceitar o *bus*, bem como mecanismos de arbitragem. O **VME** define linhas, o *bus* de arbitragem, e módulos que controlam a transferência de dados, os controladores do sistema.
Os árbitros classificam-se pelo algoritmo de arbitragem que usam, sendo os mais populares:

a. **Árbitro Simples** - É o tipo mais simples de árbitro, visto que monitoriza somente uma das linhas de pedido de *bus*, por convenção a linha **BR3#**. É bastante popular dada a sua simplicidade, e bastante rápido em sistemas pequenos, apresentando a desvantagem de "preferir" o *master* fisicamente mais próximo de si, pois é este que mais vezes consegue o *bus*.
b. **Árbitro com prioridade** - Este tipo de árbitro atribui prioridades a cada uma das linhas de interrupção, convencionando-se que **BR3#** tem a maior prioridade e **BR0#** a menor.
c. **Árbitro Round-Robin** - Este tipo de árbitro dá igual probabilidade a cada linha de pedido de *bus*, utilizando nesse sentido um método de rotação de prioridades, muito parecido a um interruptor rotativo de quatro posições.

Nota 1- Se o *master* que pediu o *bus* não colocar a linha **BBSY#** depois de o ter obtido, o árbitro pode retirar-lhe o *bus* ao fim de **x** ?s (*Arbitration time-out*).
Nota 2 - Em relação ao pedido de *bus* é possível considerar vários métodos, sendo o mais usado e simples o método de *Release when done*, no qual o *master* só liberta o *bus* quando tiver terminado a sua transferência.

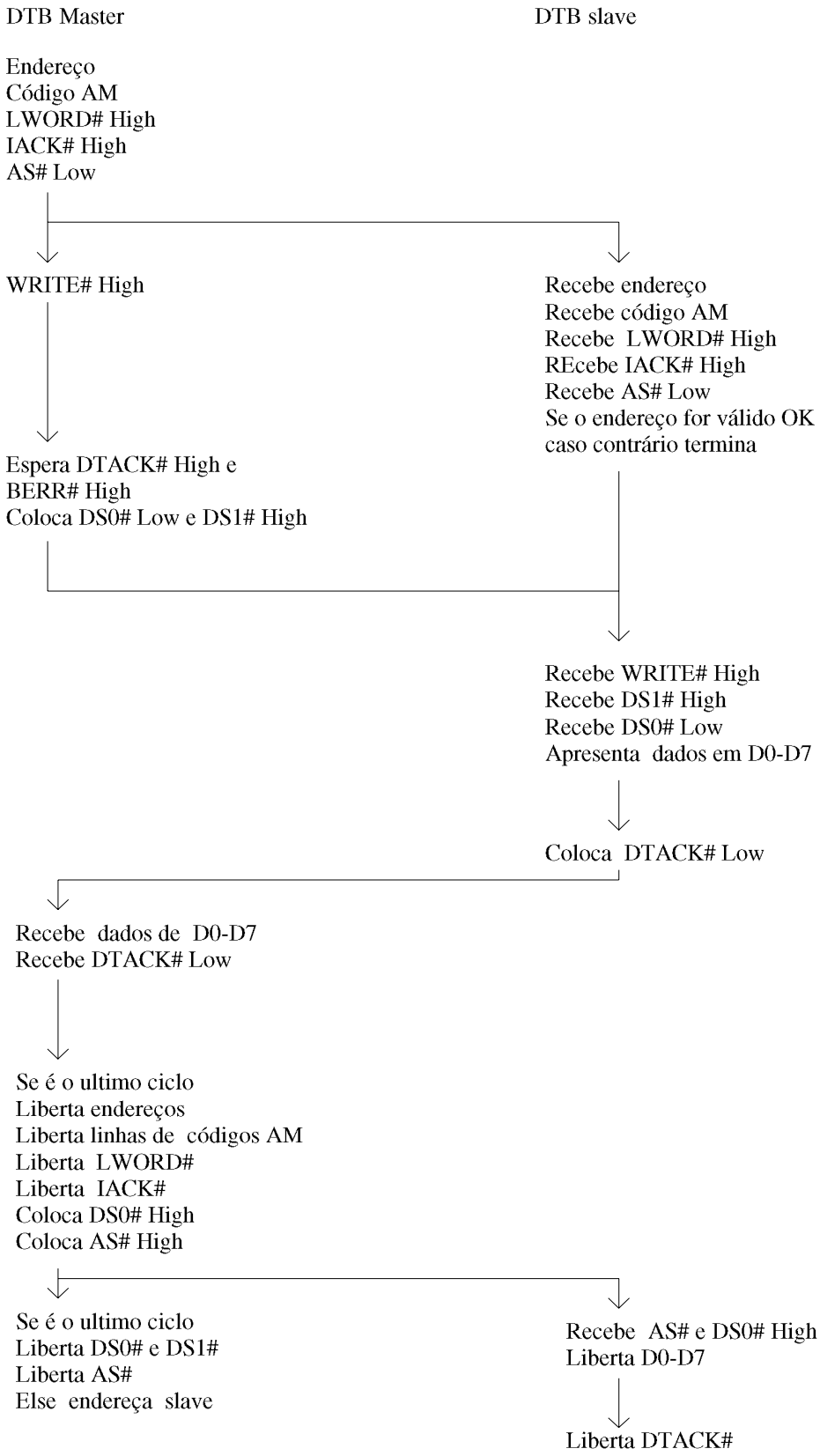


Fig. A1 - Ciclo de leitura de um *byte*.

A.4. Interrupções

O *bus* de interrupções do **VME** possui 8 linhas de interrupção, **IRQ7#**-**IRQ0#**, sendo a linha **IRQ7#** a de maior prioridade. Um ciclo de interrupção começa com o respectivo pedido através de uma das referidas linhas. O *interrupt-handler*, depois de ter obtido o *bus*, o que o torna um *bus-requester*, activa as linhas **A1**-**A3**, **IACK**# e **AS**#. A linha de **IACK#** informa que ciclo é um ciclo de resposta a interrupção, e as linhas **A1**-**A3** informam do seu nível de prioridade. Quando as linhas **AS#** e **DS#** estiverem activas, a cadeia **IACK#** propaga-se de módulo para módulo, partindo do módulo 01, determinando qual deles pediu interrupção. Esse módulo, coloca no *bus* o vector **STATUS/ID**, que pode ser usado pelo *handler* para determinar a origem da interrupção, terminando o ciclo com a linha de **DTACK#**. O *bus* de interrupções do **VME** é um *bus* com prioridades, o que permite ciclos de interrupção rápidos, pois o *handler* não precisa de verificar todas as fontes de interrupção para determinar qual delas pediu a interrupção.

Existem duas classes de geradores de interrupções:

ROAK-*Release On Acknowledge*

Este tipo de gerador de interrupções retira o seu pedido de interrupção, negando a linha de interrupção que usou, quando responde a um ciclo de *acknowledge*.

RORA-*Release On Register Access*

A linha de interrupção só é negada quando se tem a certeza que o *handler* está a executar a rotina de serviço à interrupção. Assim, o pedido de interrupção é retirado quando o *handler* tem acesso a um registo interno do módulo **RORA** alterando o seu estado.

A.5. Utilitários

O *bus* de utilitários do **VME** fornece linhas que permitem fazer a inicialização e monitorização do sistema, detectar falhas de alimentação e gerar sinais periódicos. Possui um *clock* de 16MHz que pode ser usado para qualquer fim, e linhas para inicializar correctamente o sistema ou terminar a sua operação. Essas linhas são:

SYSFAIL#- Informa de uma falha do sistema, de *hardware* ou *software*, permitindo assim que os sistemas **VME** tenham capacidades de monitorização.

ACFAIL#- Informa que a alimentação irá terminar no mínimo dentro de 4 ms, permitindo que os módulos terminem as suas operações e executem as suas rotinas de s*hut-down*, evitando assim perda de dados, ou problemas relacionados com o corte abrupto da alimentação.

SYSRESET#- É usado para fazer *reset* a um sistema **VME**, mantendo_se activo, pelo menos, 200 ms depois dos +5V estarem estáveis após o *power-up*. Para além do *power-up reset*, esta linha pode ser usada para transmitir um *reset* manual do sistema.

Apêndice B
Comunicação de dados
Notas sobre a Ethernet.

B.1. Introdução

O objectivo deste apêndice é o de fazer uma breve revisão sobre a especificação *Ethernet* apontando as suas características mais importantes: aborda-se o algoritmo **CSMA/CD**, analisando a sua *performance* e sugerindo alterações em situações particulares de elevado tráfego. Inicia-se o apêndice com uma revisão dos princípios teóricos da comunicação de dados e das limitações inerentes à transmissão por cabos [TANENBAUM 89] [HALSALL 92].

B.2. Notas sobre transmissão de dados

É bem conhecido actualmente que uma função periódica g(t), bem comportada, de período T pode ser representada,

g(t) = **Error!** c + **Error!** + **Error!** (1)

com f = **Error!** e em que a_n e b_n são as amplitudes dos termos de ordem n (harmónicos) e c é uma constante. Uma função escrita na forma anterior diz-se que foi decomposta em *série de Fourier*. Multiplicando ambos os lados da equação (1) por sen(2πkft) e integrando para valores de t entre 0 e T obtém-se a expressão para a_n; Da mesma forma, multiplicando ambos os termos de (1) por cos(2πkft) e integrando entre 0 e T obtêm-se as expressões para b_n e c:

an = **Error! Error!** bn = **Error!Error! Error!** c = **Error!Error! Error!**

A transmissão de um sinal é limitada pela largura de banda do meio de transmissão. A cada harmónico, e respectiva frequência, corresponde uma determinada energia transmitida que é proporcional a $\sqrt{,a_n^2 + b_n^2}$. Da transmissão de um sinal resulta sempre uma perda de energia, que não é igual para todas as frequências, originando distorção do sinal transmitido. Normalmente, os harmónicos são transmitidos com pouca atenuação até uma frequência f_c, frequência de *cutoff*, sendo depois rapidamente atenuados. Esta frequência f_c é na maior parte dos casos uma limitação do meio de transmissão, embora possa resultar de uma limitação à largura de banda posta à disposição dos utilizadores. A existência de f_c não proíbe a transmissão, mas limita a taxa de transmissão.

Nyquist demonstrou em 1924 que um canal de largura de banda H, para um sinal a transmitir com V níveis distintos, admite uma taxa de transmissão máxima igual a $\mathbf{2H \log_2 V}$ b/s. Se o canal tiver ruído, este pode ser medido pela relação sinal-ruído η = **Error!** , em que S é a potência de sinal e N a potência do ruído. Neste caso, como demonstraram *Shannon* e *Hartley* para um sinal genérico com um número indeterminado de níveis, a taxa máxima de transmissão, para um canal de largura de banda H, é $\mathbf{H \log_2 (1+}$**Error!**$\mathbf{)}$ b/s. Esta é uma fórmula teórica baseada inteiramente em argumentos da teoria da informação. Na prática, interessa saber qual deve ser o nível mínimo do sinal, em relação ao nível do ruído, para se obter uma probabilidade de erro aceitável. Tendo em conta que a energia média transmitida por *bit* é E = ST = **Error!**, em que S é a potência do sinal, T o período de 1 *bit* e R a taxa de transmissão, podemos quantificar o efeito do ruído. Como N_o = kT é potência do ruído para uma largura de banda de 1Hz, sendo k a constante de *Boltzmann* e T a temperatura em *Kelvin*, temos,

Error! = **Error!** .

(2)

Na prática, devemos escolher uma relação **Error!** apropriada para se obter uma probabilidade de erro pequena. O valor de **Error!** depende do tipo de modulação usado, mas se fixarmos um valor chegamos à conclusão, a partir de (2),

que a potência do sinal aumenta com o aumento de R (taxa de transmissão) e/ou com o aumento de T (temperatura).

Por último, um sinal demora um período de tempo finito a propagar-se entre o emissor e o transmissor, através do meio de transmissão. A velocidade de propagação por cabo coaxial ou par entrançado, que são os meios em que estamos interessados, é da ordem de 2 x 10^8 m/s. Podemos definir então dois importantes parâmetros:

$\mathbf{T_p}$ - atraso de propagação = **Error!** e,

$\mathbf{T_x}$ - atraso de transmissão = **Error!**

B.3. Avaliação da Ethernet

A *Ethernet* é uma rede com protocolo **CSMA/CD** (*Carrier Sense Multiple Access with Colision Detection*) persistente com probabilidade 1 (*1_persistent*). Segundo este protocolo (fig. B1), antes de transmitir uma estação verifica o estado do cabo (o *ether*) no sentido de detectar se mais alguém está a transmitir. Se detectar o estado *idle*, transmite o seu *frame*. Se ocorrer uma colisão, as estações que colidiram abortam imediatamente a transmissão (*Colision Detection*), esperam um período aleatório de tempo e repetem o processo. O protocolo é denominado *1-persistent*, porque qualquer estação, com algo para transmitir, transmite com probabilidade 1 logo que encontra o canal no estado *idle*. Essa probabilidade poderia ser p<1, protocolo *p_persistent*, resultando que num estado *idle* a estação transmitia com probabilidade p ou deferia, para a *time-slot* seguinte, com probabilidade q=1-p.

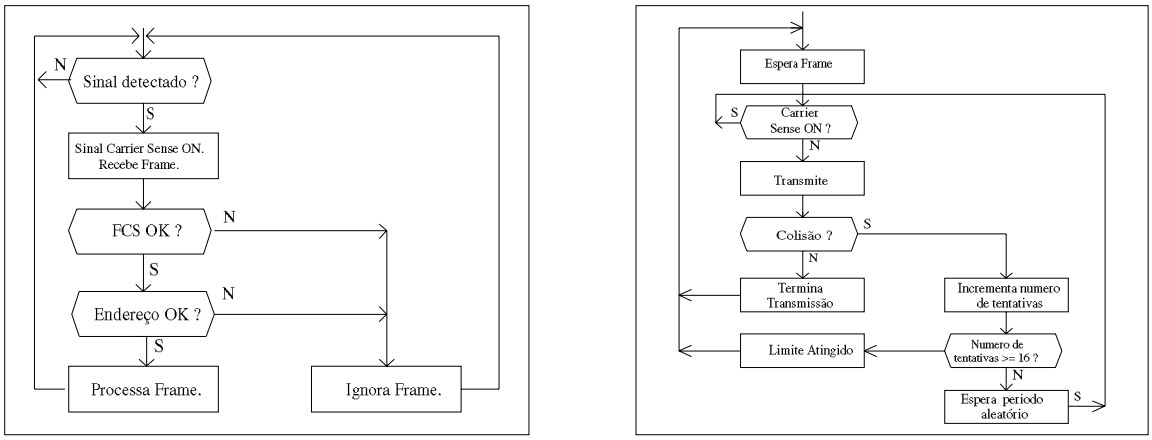


Fig. B1 - Protocolo **CSMA/CD**: Recepção e transmissão.

A *Ethernet* utiliza na codificação dos dados a transmitir o código de *Manchester* (fig. B2). A existência de uma transição no meio de cada *bit*, permite que o receptor fique sincronizado com o transmissor.

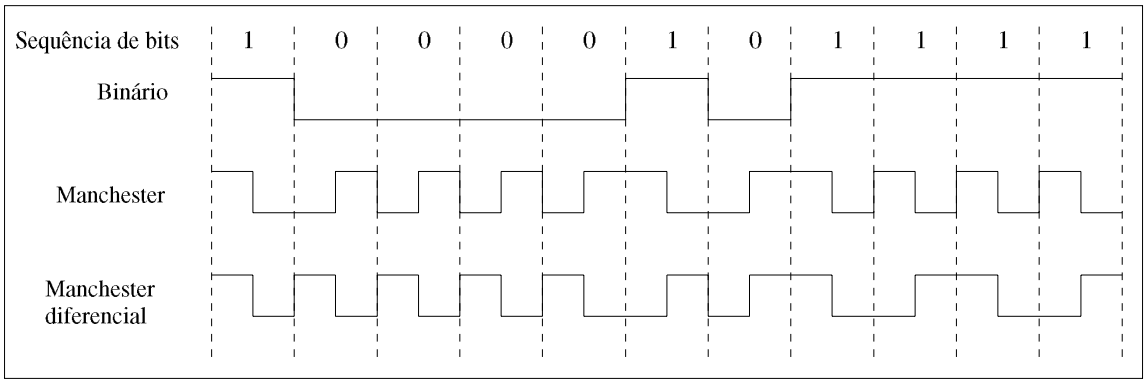


Fig. B2 - Três tipos de codificação de dados: a) Binário, b) *Manchester*, c) *Manchester* diferencial.

Nota: Tipos de cabos usados na Ethernet

É possível usar dois tipos de cabos: *Thick Ethernet* e *Thin Ethernet*. O *Thin Ethernet* é o mais usado, em redes pouco extensas, e corresponde a um cabo coaxial (*coax*) de 50 ohm que usa fichas *standard* **BNC** para fazer ligações e junções. É possível usar par entrançado em certas situações específicas.

B.3.1. Configuração típica da Ethernet

A configuração habitual da *Ethernet* está representada na fig. B3. Os *Transceivers* (dispositivos cravados no *coax*) possuem a electrónica responsável pela

detecção do estado da linha e pela detecção e sinalização de colisões. A ligação ao computador é feita recorrendo a um cabo, constituído por 5 pares entrançados devidamente isolados que pode ter no máximo 50 metros, e a uma placa de *interface*. Esta placa é responsável pela transmissão e recepção de *frames*, organização dos *frames* a transmitir, verificação de **FCS**, etc.

O *coax* pode ter no máximo 500 m, sendo necessário repetidores para distâncias maiores. Basicamente a função de um repetidor é amplificar e retransmitir o sinal que recebe, permitindo que ele se continue a propagar em boas condições.

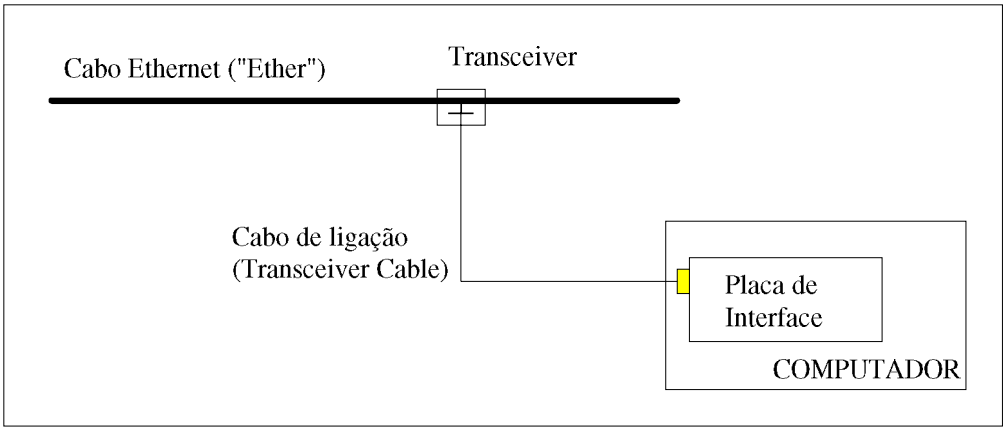


Fig. B3- Configuração típica da *Ethernet*.

B.3.2. Estrutura dos Frames

O *frames* da *Ethernet* têm o seguinte formato,

Preamble	Start	Destination Address	Source Address	Length	Data	Pad	FCS

Fig. B4 - Formato dos frames

Preamble (7 *bytes*)

- Inicio de *frame*. Todos os *bytes* são iguais a AA(h). A função deste campo é permitir que receptor se sincronize com o transmissor, visto que a codificação de *Manchester* deste campo produz uma onda quadrada de 10Mhz durante 5.6µs.

Start (1 *byte*)

- É o inicio do *frame* propriamente dito, sendo constituído pelo código AB(h).

Destination Address (6 *bytes*)

- Especifica a(s) máquina(s) que deve(m) receber o *frame*. O *bit* mais significativo deste campo pode ser 0, endereço ordinário, ou 1, endereço de grupo. A possibilidade de endereços de grupo permite *multicast* visto que o *frame* é recebido por todas as máquinas do grupo. Um endereço só de 1 significa que a mensagem deve ser lida por todas as máquinas (*All-Station Broadcast Address*).

Source Address (6 *bytes*)

- Especifica a máquina que enviou o *frame*.

Length (2 bytes)

- Este campo especifica o tamanho do campo **Data** variando entre 0 e 1500. É possível ter o campo **Data** de 0 *bytes* embora isso cause problemas na distinção entre *frames* e bocados de *frames* truncados na altura de uma colisão. Por esta razão, foi definido que um *frame* deve ter no mínimo 64 *bytes*. Existe ainda outra razão para esta exigência de comprimento mínimo, que é a necessidade de que a mensagem chegue ao fim do cabo, onde pode colidir com outra, e a sinalização de uma eventual colisão chegue ao transmissor, antes deste terminar a sua transmissão, evitando assim perda de dados devido a uma colisão não detectada.

Pad (0-46 *bytes*)

- Este campo é usado, no caso de o *frame* ter menos de 64 *bytes*, para garantir o tamanho mínimo.

FCS - *Field check sequence* (4 *bytes*)

- Este campo é o resultado da aplicação de um código polinomial, **CRC** polinomial *code*, ao *frame*. O receptor executa sobre o *frame* o mesmo cálculo, usando o mesmo código do transmissor, verificando se o resultado é igual ao valor deste campo. Em caso de erro de transmissão, o resultado da comparação será quase de certeza diferente, detectando-se assim o erro. Um **FCS** de 16 *bits* detecta todos os erros simples (1 *bit*) e duplos (2 *bits*) todos os erros com um número ímpar de *bits*,

todos os *bursts* de até 16 *bits*, 99.997% dos *bursts* de 17 *bits* e 99,998% dos *bursts* de 18 *bits* ou mais.

B.3.3. Performance.

Os protocolos **CSMA/CD**, como a *Ethernet*, funcionam como vimos segundo o modelo representado na fig. B5. No instante t_o uma estação terminou de transmitir o seu *frame*. Nesta altura, qualquer uma das estações pode transmitir. Se duas ou mais tentam transmitir ao mesmo tempo, há uma colisão. Cada estação detecta e sinaliza a colisão, espera um período de tempo aleatório e repete o processo. Sendo assim, existem 3 estados distintos no **CMSA/CD**: *Idle*, contenção e transmissão. Depois de uma primeira colisão, a estação espera 0 ou 1 *time-slots* antes de voltar a tentar. Se registar uma nova colisão na segunda tentativa, a estação espera 0, 1, 2 ou 3 *time-slots*. De uma maneira geral, depois de i colisões consecutivas, a estação espera aleatoriamente entre 0 e 2^i-1 *time-slots*. No entanto, ao fim de 10 colisões consecutivas o número máximo de *time-slots* é fixado em 1023. O controlador de *Ethernet* desiste de tentar e informa que existe um erro, quando regista 16 colisões consecutivas.

Este algoritmo permite que uma colisão seja rapidamente resolvida se o número de estações que colidem é pequeno, embora esse período de espera possa ser relativamente grande se o número de estações que colidem for também grande. Numerosos estudos, publicados durante a década de oitenta, apontam alterações ao protocolo *Ethernet,* ou mesmo novos protocolos **CSMA/CD**, os quais apresentam melhores resultados que a *Ethernet* em condições de elevado tráfego [CHLAMTAC 79] [KIESEL 83] [GREENE 81] [GALLAGER 85].

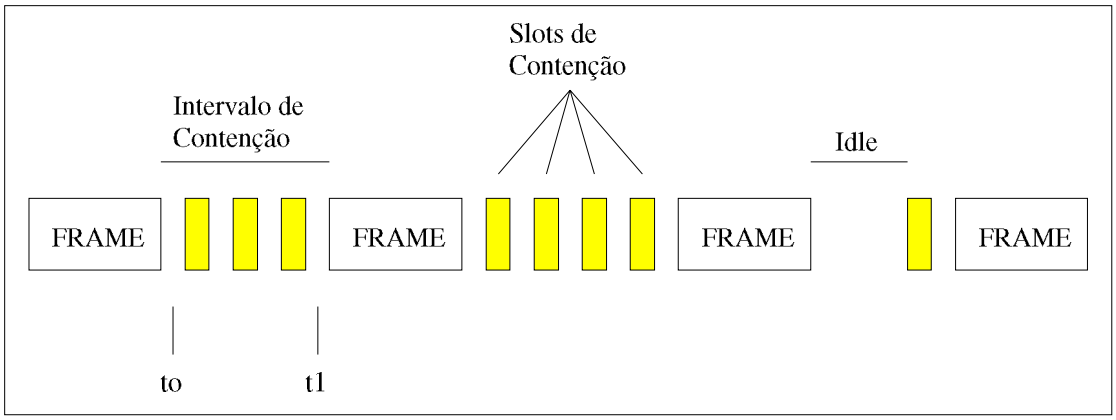


Fig. B5 - Estados de um protocolo CSMA/CD [TANENBAUM 89].

Antes de introduzir um desses estudos vou apresentar as equações que permitam avaliar a performance da *Ethernet* nas condições anteriormente referidas, utilizando, por simplicidade, o método usado por *Metcalfe* e *Boggs* [TANENBAUM 89].

Vamos supor que temos sempre k estações prontas a transmitir. Se cada estação transmite com probabilidade p, durante um *slot* de contenção, então a probabilidade A de uma delas obter o cabo é dada por uma lei de distribuição de probabilidades binomial,

$$A = kp(1-p)^{k-1} \qquad (3)$$

O valor de A é máximo para p = **Error!** , com A → **Error!** à medida que k → ∞.

A probabilidade de o intervalo de contenção ter exactamente j *slots* é dada por $A(A-1)^{j-1}$, pelo que o número médio de *slots* por intervalo de contenção é,

Error!= **Error!** (4)

Como cada *slot* tem uma duração 2τ*, o intervalo médio de contenção, w, é **Error!**. O melhor valor de w é 2τe, obtido para o valor óptimo de p referido acima.

Se um *frame* médio demora P segundos a ser transmitido, quando existem muitas estações com *frames* para enviar, então,

---

* Se τ for o tempo de propaga.Æo entre as duas esta.äes mais distantes, durante uma *time-slot* . necess rio garantir que um *frame* viaje da primeira . ultima esta.Æo e que a sinaliza.Æo de uma eventual colisÆo chegue . esta.Æo emissora, isto ., a *time-slot* deve ter uma dura.Æo de 2τ que . o tempo para dar uma volta completa . rede, *round-tripe time*.

Eficiência do canal = **Error!** (5)

Sabendo que o intervalo de contenção depende do comprimento do cabo, podemos concluir de (5) que a eficiência também depende desse factor, dando origem a topologias diversas e bastante diferenciadas. A especificação da *Ethernet* não permite mais do que 2.5 Km de cabo com quatro repetidores entre *transceivers*. Para permitir esta distância, possibilitando que uma colisão seja ouvida pela estação mais longínqua, a *time-slo*t foi fixada em 51.2μs. Com uma taxa de transmissão de 10Mbps isto corresponde a 512 *bits*, ou seja, 64 *bytes* que é o comprimento mínimo de qualquer *frame*.

A equação (5) pode ser escrita em função da dimensão do *frame*, F, da largura de banda, B, do comprimento do cabo, L, e da velocidade de propagação, c, para o valor óptimo w = 2τe. Sendo assim, fazendo em (5) P= **Error!** e **Error!** = **Error!** temos,

Eficiência do canal = **Error!** (6)

Na fig. B6 representa-se a eficiência do canal, para 2τ = 51.2μs e para uma taxa de transmissão de 10Mbps , em função do número de estações prontas a transmitir. Verifica-se que os *frames* de 64 *bytes* são os conduzem a uma menor eficiência, e que para *frames* de 1024 *bytes* a eficiência é de 0.85 sendo o período de contenção de e∗64 *bytes* = 174 *bytes*.

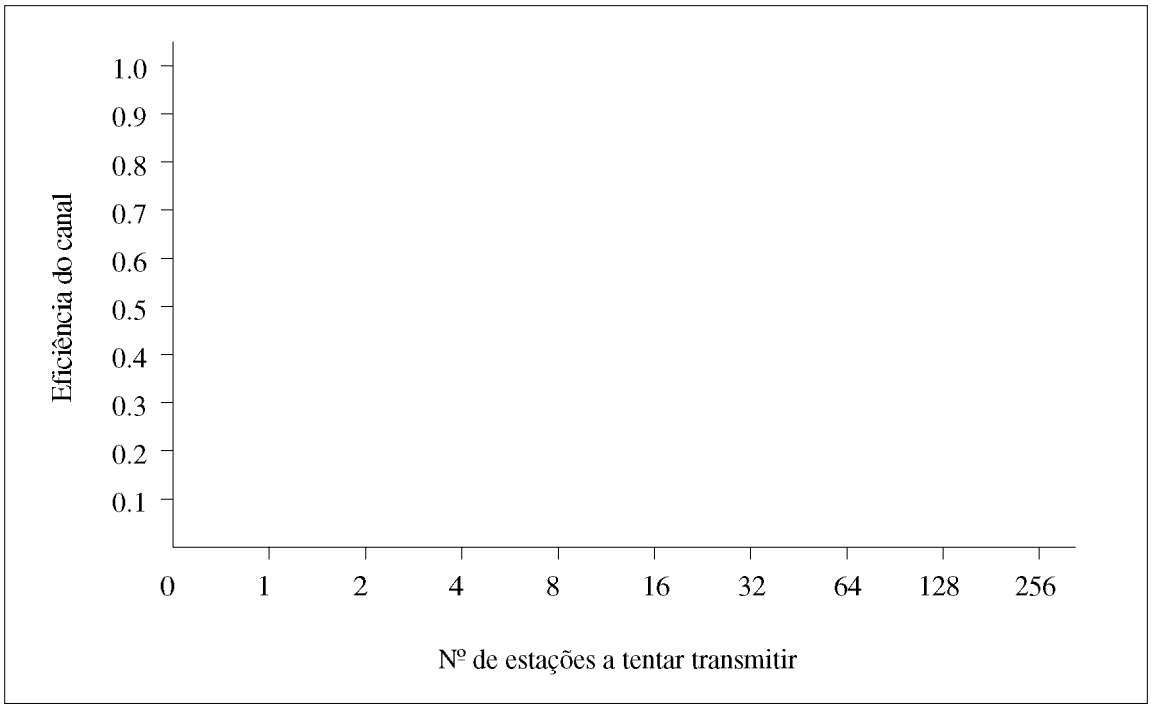


Fig. B6 - Eficiência da Ethernet para *time-slots* de 512 bits e taxa de transmissão de 10Mbps [HALSALL 92].

B.4. Alterações ao Ethernet: RCSMA/CD - *Reservation* CSMA/CD

Os protocolos **CSMA/CD** funcionam bem em situações de tráfego pouco intenso, degradando-se rapidamente a sua *performance* quando o tráfego aumenta, devido às colisões. Notar que o sistema que temos vindo a estudar destina-se a operar em instituições de investigação e desenvolvimento, normalmente situadas em instalações universitárias, nas quais o número de estações prontas a transmitir pode, em determinados períodos, ser elevado. Protocolos que não admitem colisões, *collision-free protocols*, apresentam grandes atrasos sob condições de reduzido tráfego, mas a sua eficiência relativa aumenta com a intensificação do tráfego.

A solução que a seguir apresento é um protocolo híbrido [CHEN 89], que se comporta como **CSMA/CD** para tráfego pouco intenso, adaptando-se automaticamente para um protocolo livre de períodos de contenção, *contention-free protocol*, para tráfego mais intenso. Este protocolo, denominado **RCSMA/CD** - *Reservation CSMA with Collision Detection*, é um híbrido entre o **CSMA/CD** e o **BRAM** [CHLAMTAC 79], *Broadcast Recognizing Access Method*. No inicio tudo ocorre como num protocolo **CSMA/CD**. Na altura de uma colisão, as estações que

colidem suspendem a transmissão e o canal entra no estado de contenção. Se nesse estado ocorre um transmissão com sucesso, isto é, se uma estação transmite sem interferência o seu *frame*, o canal entra num novo estado denominado estado de escalonamento, *scheduling state*. Supondo que foi a estação i que terminou com sucesso a sua transmissão, qualquer estação j ≠ i[*] pronta a transmitir determina, utilizando um algoritmo de escalonamento (*scheduling*) que depende de i e j, a *time-slot* de escalonamento que deve esperar para iniciar a sua transmissão. Ao fim de N-1 *time-slots* de escalonamento, supondo que N é o numero de estações, o canal volta ao estado de contenção e o processo repete-se. Este protocolo pode ser modificado, alterando o algoritmo de escalonamento no sentido de permitir acessos prioritários, solução com algum interesse para o **SIV**. Neste caso, depois de uma colisão, o canal entra directamente no estado de escalonamento, não passando pelo estado de contenção.

Os protocolos apresentados anteriormente foram sugeridos e estudados por *Jason Chen* e *Victor Li* em 1989 [CHEN 89]. De uma maneira geral, estes protocolos funcionam melhor que o *Ethernet* em condições de elevado tráfego, sendo sensivelmente iguais quando o tráfego é reduzido, apresentando atrasos de propagação menores e uma eficiência de canal superior.

[*] Identifica i e j com o n£mero de identifica.Æo (ID) da esta.Æo.

Sistema de Integração de Instrumentos Virtuais

J. Norberto Pires - Departamento de Engenharia Mecânica, Universidade de Coimbra, Portugal
Francisco J.A. Cardoso - Departamento de Física, Universidade de Coimbra, Portugal

Resumo

Neste artigo aborda-se a metodologia a seguir na realização de Instrumentos Virtuais de elevada *performance*, dando particular ênfase ao *software* do Sistema de Instrumentação Virtual (SIV). Define-se a arquitectura adoptada para o sistema, que se pode caracterizar por uma estrutura modular e distribuída para tempo real, constituída por uma estação UNIX de elevada performance, para definição dos instrumentos virtuais e visualização de resultados, ligada por rede local *Ethernet* a um sistema objecto distribuído, modular e aberto que realiza os instrumentos. Esse sistema é realizado sobre o bus VME, com uma placa CPU *Master* - com funções de gestão, distribuição funcional e integração - baseada no microprocessador MC68040, a correr *WxWorks,* e vários módulos *slave* específicos apropriados a cada aplicação.

1. INTRODUÇÃO

Tem-se assistido recentemente a um grande desenvolvimento e especialização da instrumentação electrónica colocada ao serviço das mais variadas áreas da actividade humana (científica, industrial, militar, etc.), como consequência da crescente complexidade e profundidade dos assuntos estudados. O resultado foi a proliferação de instrumentos dedicados e específicos, aos quais os fabricantes pretendem dar alguma flexibilidade através da introdução de microprocessadores. Traduz-se essencialmente por capacidades de reconfiguração, permitindo a utilização do instrumento em várias situações particulares, com exigências próprias. É o conceito de flexibilidade, ao nível da utilização e da configuração, que está na base deste trabalho.

Instrumentos que se baseiam num *hardware* genérico, e cujas características são essencialmente definidas por *software*, recorrendo-se a um ambiente gráfico para a sua definição, visualização e análise dos resultados obtidos, denominam-se Instrumentos Virtuais. Um Instrumento Virtual (IV) é um instrumento que não existe fisicamente, daí o nome de virtual, mas que pode ser definido, ou redefinido, facilmente em qualquer altura (Santori 1990). A ideia é a de definir um suporte de *hardware*, a sua arquitectura, capaz de suportar os instrumentos em que estamos interessados.

Ao longo do artigo será definido o conceito de Instrumento Virtual, bem como sustentada a arquitectura escolhida para o implementar já apresentada em (Marques 1992). O objectivo específico é o de definir, em termos de *hardware*, *software* e comunicações, o módulo integrador citado no referido trabalho.

2. SISTEMA DE INSTRUMENTAÇÃO VIRTUAL

2.1 - Conceito de Instrumento Virtual. Um instrumento é, de uma maneira geral, uma associação de pequenos instrumentos individuais (denominados instrumentos elementares) com funções específicas e bem definidas, em que a saída de um é a entrada do seguinte. Um Instrumento Virtual é um instrumento cujas funções básicas e capacidades são definidas por *software*. É necessário, portanto, identificar com exactidão as funções básicas a desempenhar (instrumentos individuais), agrupando-os em 3 níveis distintos: Aquisição, Análise e Saída. Esta potencialidade resulta da introdução da ideia de Instrumentação como uma hierarquia de Instrumentos Virtuais, na qual todos os Instrumentos Virtuais de um determinado nível hierárquico têm o mesmo tipo de construção. A definição de instrumentos deve ser feita recorrendo a uma linguagem gráfica (Shu 1985), com a qual o utilizador define um instrumento como provavelmente o faria com papel e lápis, ou seja, usando um diagrama de blocos. Define-se, desta forma, um instrumento interligando blocos, cada um representando um Instrumento Virtual dentro de três níveis funcionais distintos:

Nível 1 - Aquisição de Dados. Neste nível encontram-se os instrumentos de controlo do *hardware*, seja ele baseado no *bus* do *Macintosh*, do PC ou no *bus* VME/VXI, das comunicações, *RS232/422/485* ou GPIB, *Ethernet*, ..., e da aquisição de dados: aquisição analógica, digital e controlo de contadores e *timers*.

Nível 2 - Análise de dados. Neste nível encontram-se os instrumentos com funções de *DSP*, como o gerador de sinais, filtro digital, análise em tempo e em frequência, com funções estatísticas, como a estatística descritiva, *fitting*, álgebra matricial, etc.

Nível 3 - Saída de Dados. Os instrumentos implementados neste nível incluem as funções de apresentação de dados, cor, etc, de armazenamento de dados e apresentação dos resultados, como a impressão, a organização de relatórios, os ficheiros ASCII e binários, etc.

2.2 - Sistemas Actuais. Os sistemas de instrumentação actualmente existentes no mercado podem ser classificados em três grupos principais:

1. Placas de aquisição, análise e controlo para PC. Existem no mercado da especialidade inúmeras placas de vários fabricantes, concebidas para os diferentes tipos de Computadores Pessoais. Estas placas podem ser dedicadas ou não, existindo, portanto, placas com funções únicas e placas com várias funções, sejam elas de DSP (*Digital Signal Processing*), conversão A/D (Analógico-Digital) e D/A (Digital-Analógico), entradas e saídas (I/O) digitais, contadores e *timers* (temporizadores digitais programáveis).
2. Sistemas de aquisição, análise e controlo, modulares e fechados. Este tipo de sistemas são apresentados pelos fabricantes como soluções expansíveis, para as quais oferecem vários módulos opcionais, não permitindo, geralmente, qualquer tipo de desenvolvimento por parte do utilizador. São essencialmente sistemas baseados num *bus* próprio do fabricante, não divulgado, com um *interface* próprio ou *standard* (Pires 1993).
3. Sistemas de aquisição, análise e controlo, modulares e abertos. Neste grupo estão os sistemas baseados num *bus standard*, perfeitamente definido e disponível, como o VME ou o VXI, com um *interface* próprio do fabricante ou *standard*.

Com estes sistemas, os fabricantes costumam apresentar *software* que permite construir aplicações adaptadas a cada utilizador, possibilitando-lhe uma eficiente e completa exploração do seu *hardware*; existem no mercado inúmeras ferramentas deste tipo, algumas sob a forma de objectos que podem ser incluídos em ambientes de programação muito diversos (Pascal, C/C++, Basic, ...).

2.3 - Arquitectura do Sistema de Instrumentação Virtual. Na fig.1 é apresentada a base de *hardware* concebida para suportar os instrumentos virtuais, designado por Sistema de Instrumentação Virtual (SIV). O objectivo é o de construir um sistema de elevada eficiência que suporte o desenvolvimento de IV, preferencialmente dentro das áreas do controlo, análise e teste em tempo real. A abordagem de multitarefa (multi-IV) que pretendemos fazer, garantida pelo sistema operativo (*WxWorks*) e pelo *software* de aplicação, aponta claramente para um sistema distribuído. A opção por um sistema modular é também clara, se atendermos ao seguinte:

1. Não é possível definir *hardware* tão genérico que suporte todos os instrumentos actuais e futuros. O sistema deve possibilitar a definição de novos instrumentos através da integração de novos desenvolvimentos de *hardware*.
2. A evolução contínua e rápida da electrónica permite constantes melhoramentos do *hardware*, traduzindo-se pela possibilidade de definição de instrumentos mais eficientes. O sistema tem que possibilitar a substituição dos módulos por versões mais actualizadas e eficientes.

3. Um sistema deste tipo pode ser usado por pessoas das mais variadas áreas, em aplicações com especificidades e graus de exigência diferentes. Este facto exige a possibilidade de várias configurações de *hardware,* à medida do utilizador e da aplicação.
A modularidade surge, assim, como uma exigência imposta naturalmente pelos objectivos que se pretendem atingir.

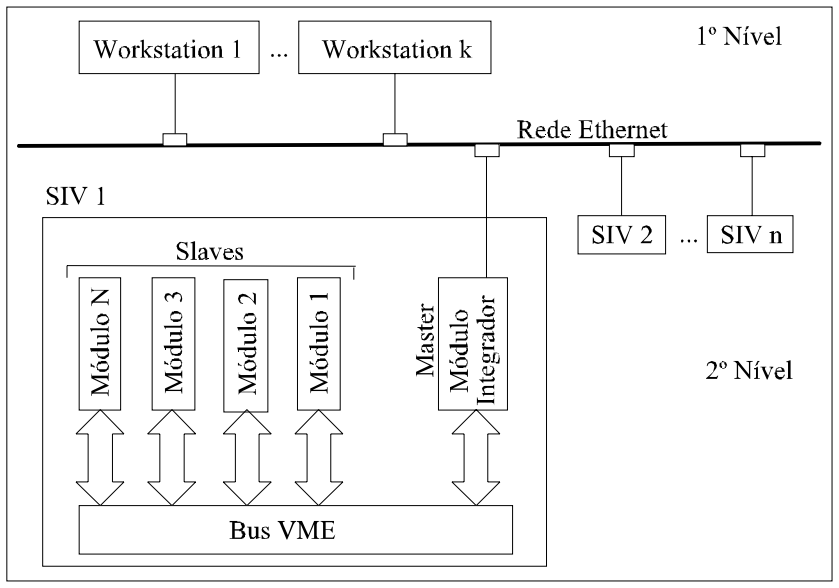


Figura 1 - Arquitectura geral alargada do SIV.

O meio de comunicação entre os vários módulos, suporte do sistema distribuído, é o *bus* VME.

Para a definição dos instrumentos, bem como para a apresentação gráfica dos resultados, é usada uma estação gráfica com sistema de operação UNIX ligada ao SIV por *Ethernet*. A rede *Ethernet* garante comunicações com o nível de *performance* necessário para a realização dessas operações (Pires 1994). A utilização de um sistema gráfico baseado no sistema de operação UNIX justifica-se por razões de *performance*, qualidade gráfica e capacidade de multitarefa, permitindo o uso simultâneo de vários instrumentos. [Nota. Este trabalho foi iniciado em 1991. Na altura, ainda não estava disponível a tecnologia *Pentium*, *PowerPC* ou *Alfa* e ainda não existiam as ferramentas de desenvolvimento e compilação cruzada para o sistema operativo *WxWorks* sob plataforma *Windows* ou *DOS*. Hoje a estação de desenvolvimento seria provavelmente um *PC/Pentium*, um *PC/PowerPC* ou um *PC/Alfa* com o *WxWorks* para *Windows*, que é uma solução claramente mais barata e de comparável *performance* (Pires e J.M.G. Sá da Costa 1995)].

As caracteristicas dos módulos *slaves* e do módulo *master* são espostas de seguida:
Módulos *slaves.* Genericamente serão módulos de características A32, D32, SEQ e *interrupt-requesters*, usando qualquer uma das linhas de interrupção, de maneira a possibilitar vários esquemas de arbitragem. O interface com *bus* VME é feito através de *buffers* de leitura e de escrita, nos quais serão colocados os comandos do *master* e os resultados da execução desses comandos, que podem ser *FIFO* ou *Dual-Ported RAM*.

Módulo *master*. O módulo *master* do SIV é responsável pela gestão dos IV. As operações que lhe estão atribuídas são:
a) Gestão das comunicações com a rede *Ethernet*: os *frames* de definição dos IV, provenientes da EGDIV (Estação Gráfica de Definição de Instrumentos Virtuais), devem ser interpretados e tratados, originando novos processos cuja execução é também controlada; Os resultados devem ser estruturados em *frames* e disponibilizados para leitura via rede *Ethernet*.
b) Gestão de processos: a função principal é a definição dos processos que suportam um IV, nomeadamente o controlo da sua execução, através da alteração da sua hierarquia e estado (*running*, *ready* ou *blocked*), bem como terminar (*kill*) processos.
c) Controlo do *bus*: o *master* reúne todas as funções de controlador do sistema, incluindo as funções de arbitragem do *bus*, *interrup-handler*, *bus-requester* e *bus timer*.
Assim, o módulo *master* é do tipo A32, D32 e SEQ. Não há necessidade de introduzir transferências de 64 *bits*, que obrigariam a multiplexar o *bus* de dados e o *bus* de endereços. As transferências de 32 bits, que podem ter taxas na ordem dos 40 Mbps, garantem largamente as exigências do SIV. Tendo em conta estas caracteristicas, optou-se pela placa de CPU SYS68K/CPU_40 fabricada pela *Force Computers*, que é uma placa de CPU de alta *performance* baseada no microprocessador *Motorola* 68040 a 33Mhz (SYS68K/CPU40 1990).

3. SOFTWARE DE GESTÃO DOS IV

Os IV são programas que correm sobre esse *hardware*, de acordo com regras de *scheduling*, partilha de recursos comuns e gestão temporal bem definidos. No *software* de realização de Instrumentos Virtuais distinguem-se dois níveis distintos, para além do sistema operativo que reside na parte mais baixa da memória:
1. Software de aplicação propriamente dito. É constituído pelo código dos IV, cada um formando um processo. Esse código é essencialmente gerado na EGDIV, reservando-se para o SIV a tarefa de parametrizar esses comandos com endereços, zonas de memória, definição de mensagens, etc, de acordo com critérios bem definidos.
2. Gestão das aplicações. Neste nível, colocado em memória imediatamente acima do sistema operativo, actua-se ao nível do sistema operativo, nomeadamente com o *Scheduler* e com a gestão de interrupções. Estas podem ter duas origens principais: Módulos *Slaves* e Comunicação sobre a rede *Ethernet*.

No SIV, cada processo (IV) tem de terminar dentro de um espaço de tempo pré-definido (*deadline*). Se qualquer um dos processos não termina nesse intervalo de tempo, então dizemos que o SIV falhou. O objectivo é definir tempos de resposta mínimos que possam ser garantidos pelo SIV (Leinbaugh 1980)(Zhao et al 1987a,b). Esses tempos de resposta dependem obviamente do número de processos existentes. É necessário, portanto, decidir se os pedidos de constituição de novos processos, que continuam a chegar aleatoriamente, podem ser atendidos satisfazendo as suas próprias restrições temporais, bem como as dos processos já constituídos. Este problema foi tratado em *Zhao* et al (1987a,b). Utiliza-se um algoritmo de *Scheduling* com prioridades dinâmicas (Zhao et al 1987a,b)(Hyman 1991), alterando a prioridade dos vários processos (IV) de acordo com os *deadlines* respectivos, atribuindo maior prioridade às rotinas de pré-processamento (*handler*) de interrupções e comunicação sobre a *Ethernet*.

3.1. Tempos de resposta (mínimos) garantidos. Os processos em que estamos interessados, que realizam os IV, têm as seguintes características fundamentais:
1. Cada processo é constituído por uma série de comandos (segmentos) que são executados por ordem.
2. Os processos usam recursos comuns, como por exemplo os módulos do SIV, zonas de memória e estruturas de dados, etc.
3. O acesso a esses recursos comuns pode ser feito em modo exclusivo utilizando o conceito de região crítica. De uma maneira geral, um recurso comum pode ser usado no já referido modo exclusivo ou em modo partilhado, no qual vários processos podem usar simultaneamente o recurso.
4. Cada processo tem um determinado limite de tempo para ser executado (*deadline*). Esse tempo deve ser garantido pelo *Scheduler*. Chama-se *Tempo Garantido do processo i* (TG_i) ao tempo que decorre desde o inicio do processo até à sua execução completa.
Tendo em conta estas características, é possível calcular o TG de determinado processo.

3.1.1 - Cálculo de TG. O cálculo de TG, para cada processo, depende dos seguintes factores:
1) O tempo máximo de CPU requerido por cada segmento; 2) O tempo máximo de acesso a um recurso: aqui interessam os recursos externos ao módulo *master*, aos quais se pode chamar dispositivos periféricos, como os módulos *slave*, eventual unidade de memória de massa, etc; 3) Existência de regiões críticas, ou seja, recursos usados em modo exclusivo. O recurso é reservado antes de o segmento que o usa ser executado, sendo imediatamente libertado no fim desse segmento. Não se admitem regiões críticas que se estendam a mais que um segmento.

O tratamento dos processos, realizado pelo *Scheduler*, deve obedecer às seguintes regras:
1) Segmentos que usam recursos comuns são prioritários; 2) Se a determinada altura existem somente segmentos que não usam recursos, segmentos simples, então o *Scheduling* é feito tendo em conta a cadência de cada processo para os segmentos simples (CSS_i); 3) O acesso a dispositivos periféricos é feito segundo um critério do tipo FCFS (*First Come First Served*). Notar que o acesso a um módulo *slave* é feito através de memórias FIFO, que realizam esse critério; 4) Um segmento pode usar vários recursos. A reserva dos recursos é feita simultaneamente. O objectivo é evitar que o processo fique bloqueado por impossibilidade de acesso a um dos recursos (*deadlock*). Se existe mais do que um pedido de reserva para o mesmo recurso, operações

P()-*Down* de *Dijkstra* (Tanenbaum 1987) distintas, é atendido o primeiro a chegar.

O Tempo de Processamento TP_i de cada processo depende de eventuais entradas no estado BLK (processo bloqueado) e do atraso provocado por outros processos (com CSS_i maiores); deve ser calculado separadamente, nas piores condições de funcionamento:

$$TP_i = TTR_i + TTD_i + TTS_i/CSS_i + A_i \qquad (1)$$

em que TTR_i é o tempo total necessário para segmentos que usam recursos, TTD_i é o tempo total necessário para segmentos que usam dispositivos periféricos, TTS_i/CSS_i é o tempo total necessário para segmentos simples à cadência CSS_i e A_i é o tempo total perdido por atrasos provocados por outros processos e por entradas no estado BLK. Todos estes tempos são calculados nas piores condições de funcionamento. Um processo pode ser bloqueado ou atrasado nos seguintes casos:
1) O processo está num segmento simples enquanto outro processo usa a CPU, porque está num segmento que usa recursos - estes geralmente têm prioridade; 2) O processo precisa de aceder a um dispositivo que está a ser usado por outro processo; 3) O processo pretende iniciar um segmento que usa recursos que estão a ser utilizados. A reserva do recurso, operação P(), não é concedida, entrando o processo no estado BLK; 4) O processo está num segmento que usa recursos, existindo outros processos activos em segmentos para outros recursos.

De uma maneira geral, os processos, porque eventualmente usam dispositivos e recursos idênticos, podem bloquear ou atrasar os outros processos, isto é, o tempo de execução do processo i é afectado pela existência de outros processos. Essa é a razão pela qual existe a componente A_i em (1). Sendo assim, o acesso a um recurso ou dispositivo (esta distinção é feita em termos funcionais, embora os recursos e os dispositivos sejam tratadas da mesma forma) é constituído por duas componentes temporais distintas: tempo de CPU necessário e atrasos do tipo 2) e 3) anteriores, que se referem à interferência de outros processos.

Potencialmente, um processo j pode ser executado $\left(\frac{TG_i}{TG_j} + 1\right)$ vezes no tempo de execução do processo i. O atraso total é então,

$$A_i = \sum_{j \neq i} TTR_i \left[\frac{TG_i}{TG_j} + 1\right] + ADR_i \qquad (2)$$

em que $ADR_i = AD_i + AR_i$. Tendo em conta estas considerações tem-se,

$$TG_i = TP_i = TTR_i + TTD_i + TTS_i/CSS_i + A_i \qquad (3)$$

com $\sum_i CSS_i \leq 1$. Os vários TG_i mínimos podem ser calculados estabelecendo as relações de proporcionalidade entre eles, recorrendo-se a uma variável Ψ para definir os vários conjuntos de TG_i. Estas relações de proporcionalidade especificam as velocidades relativas de execução dos vários processos

$$\lceil TG_i \rceil = \Psi . \lceil TG_{i\,Relativo} \rceil \quad (1 = 1...n) \qquad (4)$$

Para que um determinado conjunto seja solução, é necessário que $\sum_i CSS_i \leq 1$. Este somatório é decrescente com Ψ em primeiro grau, pelo que podemos usar Ψ como parâmetro para determinar um conjunto de TG_i, para o qual $\sum_i CSS_i$ seja o mais próximo de **1** possível (sem exceder). Até agora tem-se considerado um sistema operativo ideal, isto é, cujas operações são executadas em intervalos de tempo que podemos desprezar. As operações de inicio de processo, inicio de novo segmento, pedido de dispositivo, pré-processamento de interrupções e sincronização por operações P()-*Down* e V()-*Up* de *Dijsktra*, necessitam de intervalos de tempos que devem ser considerados e adicionados aos tempos totais definidos anteriormente. Para além disso, também o *timer* de alta precisão do *Scheduler* necessita de algum tempo para, em particular, fazer o *context switch*. Estes custos temporais denominam-se *overheads* temporais do sistema. Introduzindo os *overheads* em (9) e usando (4), tem-se

$$CSS_i = \frac{TTN_i}{\Psi . TG_{irelativo} - TTR\&O_i - TTD\&M_i - B\&O_i - T\&O_i} \qquad (5)$$

em que, O_i é o *overhead* adicionado aos tempos totais definidos anteriormente, M_i é o *overhead* máximo para dispositivos periféricos (essencialmente igual ao tempo de insensibilidade a interrupções vezes o número de acessos a dispositivos no processo i), $B\&O_i$ é atraso total (incluindo *overheads*) por influência de outros processos e $T\&O_i$ é o intervalo de tempo (incluindo *overheads*) que pode ser usado na gestão de interrupções do *timer* (*Scheduling*) durante a execução do processo i. Neste intervalo de tempo devemos incluir o atraso provocado pelo pré-processamento da interrupção da *Ethernet*, que pode ser considerada uma operação interna do *scheduler*.

3.2 - Constituição de um *schedule*. O número de processos, o seu tipo e as respectivas restrições temporais não são estáticas no tempo, isto é, de forma aleatória e continua chegam pedidos de constituição de novos processos (IV). O problema aqui é o de decidir se determinado processo pode ser constituído, garantindo o seu próprio *deadline*, bem como os dos outros processos existentes. À organização temporal desse conjunto de processos, sobre a qual se vai debruçar o *scheduler*, chama-se *schedule* (Zhao et al 1987a,b). Devemos determinar se com o novo *schedule*, resultado da introdução de um novo processo, é possível garantir as exigências temporais de todos os processos existentes. Nesse caso, o *schedule* diz-se exequível. Os processos apresentam os seguintes parâmetros característicos:
1) Tempo de processamento, $T_j>0$; 2) *Deadline*, Dl_j ; 3) Necessidade de recursos, $R_j=(R_j(1), R_j(2), ..., R_j(r))$ em que, r é o número de recursos, $R_j(i)=0$ se P_j não necessita do recurso R_j, $R_j(i)=1$ se P_j necessita R_j em modo partilhado e $R_j(i)=2$ se P_j necessita de R_j em modo exclusivo.

Um *schedule* S consiste num conjunto de fatias temporais F_k (k=1, ..., NF), em que NF é o número de fatias de S. A uma determinada fatia F_k está associado um instante inicial IIF_k, um comprimento temporal DF_k e um subconjunto de processos que podem correr durante essa fatia de tempo, isto é, desde IIF_k a IIF_k+DF_k. Concretamente, uma fatia temporal é um vector de dimensão r (nº de recursos), cujo elemento de ordem i informa qual o processo que usa o recurso R_i e em que modo. Dado um determinado conjunto de n processos, dizemos que um *schedule* está completo se, para todos os valores de j=1, ..., n,

$$T_j \leq \left(\sum_{P_j \in F_k} DF_k\right) - \delta . K_j \qquad (6)$$

em que K_j é o número de vezes que P_j é bloqueado para dar lugar a outro processo (*Preemptive Scheduling*) e δ o *overhead* relativo ao tempo de troca (*Switching*) de contexto.

Dizemos que um *schedule* é exequível se, máx($IIF_k+DF_k : P_j \in S_k$) ≤ DL_j , isto é, se a ultima fatia temporal contendo P_j termina antes do *deadline* de P_j. Pretende-se determinar um *schedule* exequível e completo para os processos existentes, isto é, identificar uma sequência de fatias na qual seja possível garantir o *deadline* de todos os processos. Este é um problema de pesquisa (Zhao et al 1987a,b). O algoritmo de pesquisa que se usa, parte de um *schedule* vazio e percorre o espaço de pesquisa até encontrar um *schedule* exequível. Esse *schedule*, em princípio, não é único, pelo que o algoritmo deve conduzir ao *schedule* óptimo. Se o algoritmo não conduzir a um *schedule* exequível, então não é possível garantir os requisitos temporais do conjunto de processos existentes. O espaço de pesquisa assemelha-se a uma árvore cujos *vertex*, ligações entre dois ramos, constituem um *schedule* parcial, o qual tem mais uma fatia em relação ao *schedule* que lhe deu origem. O objectivo é definir meios que permitam avançar de *vertex* em *vertex*, em direcção a um *schedule* exequível. O critério é o de determinar se dado *vertex* é passível de conduzir a um *schedule* exequível ou não. Dizemos que o *vertex* é fortemente exequível (Zhao 1987a,b), se a resposta for afirmativa. Um *vertex* que não seja fortemente exequível, nunca conduzirá a um *schedule* exequível e completo.

3.2.1. Algoritmo de pesquisa. Em cada nível de pesquisa deve ser identificado o conjunto de processos que constituirão a respectiva fatia (fig. 4). O sucesso do algoritmo depende da correcta identificação desse conjunto de processos. Os processos são seleccionados de acordo

com os seus requisitos temporais e necessidade de recursos. Basicamente, constrói-se uma fatia com um processo principal, denominado primário, e um ou vários processos secundários. O processo primário é escolhido tendo por base exclusivamente considerações temporais. Os processos secundários são escolhidos usando uma função heurística H(). Esta função H() deve ter em conta requisitos temporais e necessidade de recursos: essa seria, obviamente, uma função H() ideal. A função H() é aplicada a todos os processos disponíveis. O processo com menor valor de H() é seleccionado para a fatia. Repete-se o processo até não ser possível incluir mais processos na fatia. O algoritmo usa a variável INF, para indicar o inicio da nova fatia. Quando a fatia F_k é identificada faz-se,

IIF_k=INF e

$$DF_k=\min(\min(T'_j:P_j\in F_k), \min(\eta_h\text{-INF}:P_h\in F_k \text{ e } T'_h>0)) \quad (7)$$

em que T'_j é tempo de processamento que resta a P_j e $\eta_h=DL_h-T'_h$. O primeiro termo de (7) especifica que a duração da fatia não deve exceder o menor valor de T'_h, e o segundo termo especifica que não deve ser maior que η_h. Por exemplo, se $DF_k=0$, então para determinado $P_h\in F_k$, η_h-INF=0 $\Rightarrow$ η_h=INF. O processo P_h deve ser incluído em F_k pois, caso contrário, falhará o seu *deadline*. Depois de estabelecida a fatia, o valor de INF para a fatia seguinte é colocado a $INF=IIF_k+DF_k$.

Em cada nível de pesquisa também se deve calcular o vector MRDR (Zhao et al 1987a,b), *Minimum Resource Demand Ratio*, que dá uma indicação das necessidades em termos de recursos de cada um dos processos ainda existentes,

$$MRDR=(MRDR_1, MRDR_2,..., MRDR_r). \quad (8)$$

A ideia é a de manter baixos os valores $MRDR_i$, escalonando imediatamente os processos que usam o recurso i quando o valor $MRDR_i$ for grande. Se a qualquer altura o valor de $MRDR_i$ para o recurso R_i for maior que um, então não é possível escalonar os processos que restam cumprindo os respectivos *deadlines*.

```
Procedure P_Schedule(task_set:task_set_type; var schedule: schedule_type;
var schedulable: boolean);
VAR INF: Integer;
   MRDR: vector_type;
   SUBSET:task_set_type;
BEGIN
  schedule:=empty;
  schedulable:=true;
   INF:=current_time;
  REPEAT
   calculate_MRDR(INF,MRDR,task_set);
   IF strongly_feasible(INF,MRDR,task_set,schedule) THEN
    BEGIN
     Let Pf be the task with minimum η e T'f>0;
     SUBSET:={Pf};
     WHILE more tasks can run in paralel with those in SUBSET DO
       BEGIN
         apply H() to each candidate task;
         Let Ps be the task with the minimum value of H() funtion
         among the candidate tasks;
         SUBSET:= SUBSET ∪ {Ps};
       END;
         calculate the start time and the length of the new slice;
         apende the new slice to the schedule;
          INF := INF + DFs;
    END ELSE shedulable:= false;
  UNTIL full(schedule, task_set) or Not shedulable;
END;
```

Figura 4 - Algoritmo de pesquisa e *scheduling* (adaptado de (Zhao et al 1987b)).

É possível nesta altura, definir formalmente o conceito de *schedule* fortemente exequível, com a ajuda das estruturas INF e MRDR. Um *schedule* diz-se fortemente exequível se,

1. O vector MRDR associado ao *schedule* apresenta $MRDR_i\leq 1$, para i=1,...,r.
2. $DF_{k-1}>0$, isto é, a ultima fatia tem comprimento temporal maior que zero.

Inclui-se de seguida, a título de exemplo, uma lista de regras e funções heurísticas H():

1. *Deadline* mínimo $\Rightarrow H(P_j) = DL_j$.
2. η_j mínimo $\Rightarrow H(P_j) = \eta_j$.
3. Resto de tempo de processamento mínimo $\Rightarrow H(P_j) = T'_j$.
4. Utilização mínima de recursos.
5. Utilização máxima de recursos.

Zhao et al (1987a,b) estudaram e simularam estas e outras funções heurísticas. Os resultados mostram que 1) e 2) conduzem, de uma maneira geral, a melhores resultados que as outras regras, o que permite concluir que os requisitos temporais são os mais importantes.

4. CONCLUSÃO

Neste artigo apresentou-se o conceito de Instrumento Virtual e definiu-se a metodologia geral para os implementar. Foi apresentada a arquitectura geral do sistema de tempo real idealizado para os realizar. Discutiu-se em pormenor o módulo de integração de IV, tanto ao nível do *hardware* como principalmente, do *software*, nomeadamente na garantia de tempos de execução definidos minimos.

Este trabalho foi patrocinado pelo Departamento de Engenharia Mecânica da Universidade de Coimbra - Grupo de Controlo e Gestão, pelo Departamento de Física da Universidade de Coimbra - Grupo de Automação e Instrumentação Industrial e pela Junta Nacional de Investigação Científica e Tecnológica - Programa Ciência (Bolsa de Mestrado nº BM 1824/91).

5. REFERÊNCIAS

• Force Computers, Manual da Placa SYS68KCPU-40 8.90/20.0/A1/Rev. 1, 1990.
• Hyman J., Lazar A. e Pacifici G., "Real Time Scheduling with Quality of Service Constraints" IEEE Journal on Selected Areas in Communications, Setembro 1991.
• Leinbaugh D.W., "Guaranteed Response Times in a Hard-Real-Time Environment", IEEE Trans. on Software Engineering 1980.
• Pires J. N., "Sitac-Descrição Geral do Sistema", Encontro Nacional de Automação - CENERTEC, 1993.
• Pires J.N., "Sistema de Integração de Instrumentos Virtuais", Tese de Mestrado, Dept. de Física da U. de Coimbra, 1994.
• Pires J.N., Sá da Costa J.M.G., "Controlo de Posição e Força de Robôs Manipuladores", II Conferência Ibero-Americana de Engenharia Mecânica, 1995.
• Santori M., "An instrument that isn´t really", IEEE Spectrum, Agosto de 1990.
• Shu N., "Visual programing language: A perspective and a dimensional analysis", IEEE Computer, Agosto 1985.
• Tanenbaum A.S., "Operating Systems- Design and Implementation", Printice-Hall International Editions 1987.
• Zhao W., Ramamritham K. e Stankovic J., "Preemptive Scheduling Under Time and Resource Constraints", IEEE Trans. on Computers, Agosto 1987a.
• Zhao W., Ramamritham K. e Stankovic J., "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems", IEEE Trans. on Software Engineering, Maio 1987b.

Abstract

A strategy to implement high performance Virtual Instruments is presented. The architecture of the real-time system is also presented. Basically is a modular distributed real time system, with a graphic Unix station, for Virtual Instruments definition, connected by ethernet to an open, modular and distributed target system, developed over the VME bus to implement the instruments. The target system uses a CPU board based on the microprocessor MC68040 running WxWorks, with the task of integrating Virtual Instruments and overall system management, and several special purpose slave boards.