Universidade de Aveiro
Departamento de Electrónica e Telecomunicações

SISTEMAS INTEGRADOS EM INSTRUMENTAÇÃO: PROGRAMAÇÃO E ENSAIO

Fernanda de Madureira Coutinho

Aveiro
Abril de 1997

Resumo

O desenvolvimento de sistemas baseados em microprocessadores/microcontroladores de 8 ou de 16 *bits* destinados a aplicações industriais ou de instrumentação (sistemas integrados) é normalmente efectuado por projectistas com maior experiência no desenvolvimento de *hardware*. No entanto, o nível de exigência relativo à funcionalidade, tempo de desenvolvimento e custo de *software* é cada vez maior. Este facto favorece o recurso a metodologias sistematizadas de projecto, escrita e ensaio do *software* para sistemas integrados. Vários factores dispersos devem ser tidos em consideração: técnicas de projecto de sistemas em tempo real, verificação e ensaio do conjunto *hardware/software*, aproveitamento máximo das potencialidades das várias ferramentas utilizadas, desenvolvimento de aplicações baseadas em modelos multitarefa, etc. Neste trabalho pretende-se expor, analisar e integrar as várias fases do desenvolvimento de *software* para sistemas integrados. Aborda-se com especial atenção a utilização de núcleos de tempo real nesse tipo de sistemas. Esboçam-se ainda metodologias que permitem a caracterização funcional e quantitativa destes núcleos, tendo em atenção as suas utilizações mais típicas.

Índice

Capítulo 1

Introdução

Os sistemas baseados em microprocessadores são utilizados numa grande variedade de aplicações práticas. Muitas vezes são apresentados numa plataforma de *hardware* dedicada à aplicação em causa, constituindo parte integrante de sistemas mecânicos, electrónicos ou electro-mecânicos, sendo por essa razão designados por sistemas integrados (*embedded systems*).

O desenvolvimento de sistemas deste tipo implica obviamente endereçar as vertentes de *hardware* e de *software* as quais, no seu caso, estão fortemente interligadas. Quando o sistema se destina a um fim muito específico, a engenharia está, muitas vezes, associada ao processo e à interface. Por esse motivo, a ênfase é habitualmente posta no *hardware* sendo o *software* relegado para um plano limitado ao estritamente necessário. É claro que tal não acontece quando as aplicações, seja pelo número de unidades a construir, seja pelo valor do mercado correspondente, permitem o recurso a equipas de desenvolvimento de dimensão suficiente para juntarem especialistas de todos os domínios de interesse.

A ênfase no *hardware* é ainda notória na maior parte da bibliografia existente sobre sistemas integrados de pequeno porte. Há inúmeras publicações sobre *hardware* e interface e poucas sobre técnicas de *software* específicas para estes sistemas. As opções sobre o processador a utilizar são assim habitualmente tomadas com fundamentos perceptíveis, enquanto existe a tendência para utilizar o *software* que está mais facilmente acessível.

As considerações anteriores, essencialmente válidas na Instrumentação actual, reflectem as motivações que levaram aos estudos necessários para produzir esta

dissertação. O seu objectivo é assim analisar e caracterizar o processo de desenvolvimento de *software* para sistemas integrados. Aproveita-se para apreciar, nesta óptica, a funcionalidade de várias ferramentas e métodos de desenvolvimento. Entre outros resultados, consegue-se assim obter um guia que pretende sintetizar de forma concisa os conhecimentos necessários para o projecto do *software* destinado a sistemas integrados.

Para manter a corência do conjunto, muitos dos assuntos são tratados desde um nível razoavelmente básico. Tal permite obter um texto que se pode tornar útil mesmo para quem não possua conhecimentos muito avançados de programação.

No estudo apresentado analisam-se as características de várias ferramentas, sendo identificados diversos aspectos funcionais que tornam clara a sua utilidade ao longo de todo o processo de desenvolvimento. É analisada também a interacção entre os vários tipos de ferramentas para a obtenção do resultado final. Utilizam-se, sempre que possível, elementos sobre essas ferramentas recolhidos junto de fabricantes e fornecedores.

O desenvolvimento de *software* para sistemas integrados necessita de cuidados que não se encontram no mesmo processo quando destinado a outro tipo de plataformas, sendo alguns deles analisados ao longo do texto. É abordada a programação destes sistemas numa linguagem de alto nível, apresentando-se exemplos de soluções para alguns dos problemas mais comuns.

É feita uma análise de núcleos de tempo real para sistemas integrados, examinando-se as diferenças entre o desenvolvimento de *software* com e sem um núcleo. São analisados vários produtos comerciais cujas especificações foram obtidas junto de fornecedores e estuda-se a forma como algumas das suas características influenciam o desenvolvimento equilibrado do *software*.

O texto encontra-se subdividido nos seguintes capítulos:

1. **Introdução** - Este capítulo.
2. **Caracterização de Sistemas Baseados em μP/μC** - Apresentam-se variantes e formatos de sistemas baseados em μP/μC, definindo-se também o âmbito da tese.
3. **Desenvolvimento e Ensaio** - São expostos vários tipos de ferramentas utilizadas ao longo do desenvolvimento e são apresentadas algumas soluções disponíveis comercialmente.

4. **Tópicos de Programação de Sistemas Integrados -** É analisado o ciclo de desenvolvimento de *software* para sistemas integrados. São referidas as questões relacionadas com a programação destes sistemas numa linguagem de alto nível.
5. **Núcleos para Sistemas Integrados-** São expostas as características principais de núcleos de tempo real e qual a sua funcionalidade. Faz-se uma súmula dos dados referentes a vários produtos comerciais, terminando com uma breve análise comparativa.
6. **Conclusões e Trabalhos Futuros -** É feita uma síntese de conclusões parciais expressas ao longo do texto e a indicação de aspectos desta área de trabalho que parecem merecer uma análise mais aprofundada.

Apêndice A - Teste e Inicialização de Dispositivos de Memória - São apresentadas várias questões relativas ao teste de dispositivos de memória durante a fase de inicialização do sistema.

Apêndice B - Publicações Periódicas e Fornecedores de Ferramentas - Índice de publicações periódicas e fornecedores de ferramentas relacionadas com o desenvolvimento de aplicações para sistemas integrados.

Capítulo2

Caracterização de Sistemas Baseados em μP/μC

2.1 - Introdução

Os sistemas baseados em microprocessadores podem ser caracterizados segundo vários aspectos relativamente ao seu *hardware* e *software*.

Este capítulo identifica diferentes tipos de sistemas baseados em μP/μC e indica qual o tipo específico de sistemas que são analisados ao longo desta dissertação.

2.2 - *Hardware*

A nível de *hardware*, um sistema baseado em μP/μC pode ser caracterizado segundo os seguintes aspectos:

- Sistemas monoprocessador vs. sistemas multiprocessador.
- Sistemas autónomos vs. sistemas em rede.
- Sistemas baseados em μP vs. sistemas baseados em μC.

Um sistema multiprocessador utiliza vários μC/μP's para executar concorrentemente as aplicações. Consegue atingir maiores níveis de desempenho do que um sistema monoprocessador, mas o custo é maior e o desenvolvimento de *software* e de *hardware* mais complexo.

Um sistema em rede consiste em vários sistemas independentes baseados em μC/μP's, situados em localizações físicas diferentes. Estes sistemas trocam informação entre si através de uma rede de comunicação. Um sistema autónomo trabalha sozinho, não interagindo com outros para cumprir a sua função.

Um microcontrolador é uma variação de um microprocessador, especialmente adaptada a projectos relativamente simples, com limitações a nível de espaço e de custo. Num microcontrolador, um único circuito integra um microprocessador, memória e vários outros periféricos que são normalmente utilizados em pequenos sistemas integrados.

2.3 - *Software*

O funcionamento dos sistemas baseados em μC/μP é completamente dependente do *software* que executam. O *software* pode variar entre uma estrutura fortemente organizada, em que o código é subdividido em várias tarefas e módulos distintos, e a ausência de uma estrutura bem definida. Entre estes dois extremos podem existir formas intermédias de organização mais ou menos estruturadas, como por exemplo uma única tarefa principal para todo o processamento. As duas abordagens extremas podem designar-se por estática ou dinâmica:

- **Abordagem estática**: o *software* é construído como um único bloco monolítico, que se encarrega de efectuar todas as funções do sistema. Nesta abordagem, a interacção com o mundo exterior pode ser feita através de: *polling* aos vários periféricos, directamente através de rotinas de serviço a interrupções, ou através de interrupções que assinalam ao código principal a existência de novos dados por meio de estruturas globais partilhadas (p.ex. *flags*) entre a rotina de serviço à interrupção e o código principal.
- **Abordagem dinâmica**: o *software* é constituído por várias tarefas independentes, que podem interagir e trocar informação entre si. As várias tarefas são executadas concorrentemente (concorrência simulada em sistemas monoprocessador). Esta abordagem obriga normalmente à utilização de um núcleo (sistema operativo muito simplificado e minimalista) para gestão do escalonamento das tarefas, comunicação e sincronização entre elas.

O *software* para sistemas integrados é habitualmente desenvolvido de forma cruzada, recorrendo a um computador hospedeiro. O *software* é escrito e compilado no sistema hospedeiro, normalmente um computador pessoal, sendo em seguida transferido para o sistema alvo. Geralmente, utiliza-se uma linguagem de alto nível para desenvolver o *software* (C na maioria das aplicações), sendo, por vezes, necessário recorrer ao *assembly* em determinadas secções do código.

2.4 - Sistemas Integrados

Os sistemas baseados em μP/μC analisados nesta dissertação são designados por sistemas integrados (em inglês *embedded systems*). Consideram-se como sendo sistemas integrados sistemas de utilização específica, baseados em μP/μC de baixo custo, normalmente usados em aplicações de instrumentação ou de controlo industrial.

O *hardware* de sistemas integrados é, na maior parte das vezes, baseado em μP/μC de 8 ou de 16 *bits*. Ao contrário de sistemas informáticos de utilização genérica, são normalmente limitados a nível do *hardware* disponível, tendo por vezes que funcionar com restrições de memória e não possuindo sequer periféricos para interacção com o utilizador, excepto pequenos *displays* (p. ex. de cristal líquido ou de 7 segmentos) e teclados básicos (numéricos com eventualmente algumas teclas de função).

Vão considerar-se aqui apenas os sistemas autónomos, porque os sistemas em rede utilizam por norma sistemas informáticos de uso genérico, que caem fora do âmbito desta dissertação, ou, em alternativa, podem ser constituídos por nodos que podem ser analisados independentemente.

Abordar-se-ão desde soluções estáticas até soluções dinâmicas para o *software* do sistema integrado.

Alguns sistemas têm que garantir um tempo de resposta máximo a determinados impulsos externos. Estes são designados por sistemas em tempo real. Grande parte dos sistemas integrados têm especificações desta natureza. Esse facto vai reflectir-se na forma como o sistema é projectado, desenvolvido e testado.

Capítulo 3

Desenvolvimento e Ensaio

3.1 - Introdução

O desenvolvimento de sistemas integrados é uma actividade multidisciplinar, que obriga à utilização de uma grande diversidade de ferramentas e ao domínio do projecto de sistemas de *hardware* e de técnicas de desenvolvimento de *software*.

Pretende-se, neste capítulo, dar a conhecer as ferramentas principais, a sua funcionalidade, a forma como se utilizam ao longo do ciclo de desenvolvimento, quais as etapas em que pode decompôr-se esse ciclo no caso de um sistema integrado. Abordam-se ainda questões relativas à especificação, ao projecto, à integração, à verificação e ao teste desses sistemas.

Começa por introduzir-se, na secção 3.2, os conceitos básicos de um sistema de desenvolvimento cruzado. São apresentadas em seguida, na secção 3.3, várias ferramentas utilizadas nestes sistemas de desenvolvimento. É efectuada uma descrição sumária da sua funcionalidade, indicando-se, em alguns casos, características segundo as quais ferramentas idênticas podem diferir. Na secção 3.4, expõem-se vários ambientes de desenvolvimento comerciais, apresentando-se uma breve resenha das plataformas alvo/hospedeira em que funcionam e qual o conjunto de ferramentas que oferecem. A secção 3.5 ilustra o ciclo típico de desenvolvimento de sistemas integrados, com ênfase no *software*. São analisados alguns processos de especificação, projecto e codificação. Por fim, são exibidas características de algumas ferramentas comerciais utilizadas durante o desenvolvimento do sistema. A secção 3.6 aborda questões relacionadas com a verificação e ensaio de *software* para sistemas integrados.

3.2 - Sistemas de Desenvolvimento Cruzado

Os sistemas integrados são, por norma, sistemas fechados. Isto significa que, ao contrário do que se verifica num computador pessoal, não possuem as condições necessárias para serem utilizados como plataforma de desenvolvimento para o seu próprio *software*. Na verdade, estes sistemas não possuem algumas facilidades essenciais como, por exemplo, memória de massa, *hardware* para interacção com utilizadores ou mesmo um sistema operativo completo.

Por esta razão, é normal a utilização de um computador de uso pessoal, designado por sistema hospedeiro, para desenvolver o *software* para um sistema integrado. O ambiente de desenvolvimento funciona num sistema, mas produz código para outro, sendo por este motivo designado por sistema de desenvolvimento cruzado.

Apesar de bastar um compilador e um *linker/locator* cruzados no sistema hospedeiro para criar o código executável, é preciso, para proceder ao teste e ensaio do *software,* que este último seja executado; a não ser que se recorra a simulação, torna-se necessária a existência de um sistema alvo onde o programa é executado. O programa é escrito e compilado no sistema hospedeiro e só então é transferido para o sistema alvo. O próprio teste do *software* pode ser conduzido através de aplicações residentes no sistema hospedeiro.

3.3 - Ferramentas de Desenvolvimento e Ensaio

3.3.1 - Introdução

A variedade e diversidade de ferramentas existente actualmente, colocam o projectista numa posição algo confortável e, paralelamente, delicada. Confortável, por um lado, porque existe um leque muito vasto de possibilidades de escolha; delicada, por outro, porque o projectista tem a responsabilidade adicional de decidir quais as mais apropriadas para a aplicação em causa, respeitando compromissos entre as facilidades oferecidas, questões de compatibilidade, custos e recursos envolvidos [Microtec96].

Esta secção procura apresentar uma descrição sucinta dos principais tipos de ferramentas que se encontram na maioria dos projectos de sistemas integrados, nomeadamente editores, compiladores, *linkers*, *debuggers*, ferramentas de análise de desempenho, simuladores, monitores residentes, emuladores, ferramentas de CAD/CAE e instrumentação de apoio ao *debbuging*. Estas ferramentas podem ser

adquiridas individualmente a partir de fornecedores distintos. No entanto, é também normal que sejam vendidas várias ferramentas diferentes num único conjunto.

A integração de várias ferramentas distintas num conjunto único não só diminui o risco de incompatibilidades, como também permite um certo grau de optimização do processo de desenvolvimento. Por exemplo, um compilador é tipicamente vendido juntamente com um *linker*. Se a este conjunto se adicionar um editor de texto integrado com o compilador, o projectista pode editar o código fonte, compilar e ligar os ficheiros objecto sempre dentro do mesmo ambiente de desenvolvimento.

3.3.2 - Ferramentas para Projecto Assistido por Computador

Existem ferramentas para auxílio do projecto de *software* ou *de hardware*. As ferramentas para apoio ao desenvolvimento de *software* (CASE) são utilizadas principalmente em projectos de envergadura bastante elevada, pelo que não são frequentemente empregues em sistemas integrados. Já as ferramentas de CAD/CAE para auxílio ao projecto de *hardware*, que incluem desde o desenho de esquemas até ao traçado do *layout* de PCB's, são profusamente utilizadas nestes sistemas. O resultado de um programa de desenho de esquemas pode ser utilizado como entrada para um programa de *layouts*, automatizando assim, em grande parte, o projecto do *hardware*. Exemplos de aplicações desta natureza são o OrCAD e o Tango.

3.3.3 - Editor

Um editor é utilizado para gerar e alterar os vários documentos que vão sendo produzidos ao longo do ciclo de projecto e desenvolvimento do sistema. Um editor deve tratar texto, gráficos, imagens e quaisquer outros objectos descritivos que sejam requeridos no projecto e desenvolvimento do sistema.

Existem editores específicos que podem ser úteis em diferentes fases. Um editor de texto pode ser utilizado para escrever o código fonte do programa desenvolvido, enquanto um programa de processamento de texto mais elaborado pode servir para escrever o manual do utilizador. Deste modo, é possível criar editores especializados nestas funções, como, por exemplo, um editor de texto com identação automática e sensível à sintaxe da linguagem de programação utilizada ou um programa de desenho especializado na representação de diagramas de fluxo de dados.

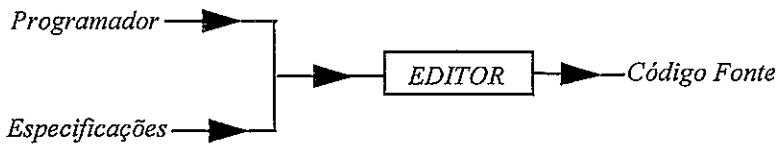

Figura 3. 1 - Funcionalidade de um editor na programação.

3.3.4 - Compilador e *Linker/Locator*

Um compilador é usado para transformar o código fonte, gerado pelo programador, em código objecto. O código objecto é a tradução do código fonte em código máquina, mas no qual as referências à memória ainda não estão resolvidas. Estes endereços não são ainda determinados neste instante para permitir ao utilizador a interligação de vários programas compilados em alturas diferentes ou até em linguagens diferentes, já que na altura da compilação é impossível ao compilador conhecer qual o tamanho e localização de funções externas utilizadas pelo programa.

A qualidade do código gerado pelo compilador deverá aproximar-se, tanto quanto possível, da qualidade atingível quando a codificação do programa é efectuada directamente em *assembly.* Se é certo que poucos ou nenhuns compiladores conseguem atingir este nível de qualidade, também é verdade que a maior parte dos compiladores actuais gera código bastante melhor do que aquele que um programador "mediano" consegue criar em *assembly.*

Os compiladores específicos para desenvolvimento de sistemas integrados devem gerar código para o processador que irá ser utilizado no sistema final, independentemente das características do sistema hospedeiro em que funcionam. Designam-se habitualmente por compiladores cruzados.

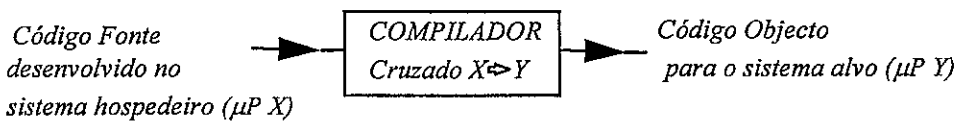

Figura 3. 2 - Funcionalidade de um compilador cruzado.

Um *linker* é uma ferramenta que agrega vários ficheiros de código objecto num único ficheiro de código executável. É o *linker* que junta ao código do utilizador o

código das funções de biblioteca utilizadas pelo programa. Um *linker* pode ter a capacidade de ligar ficheiros objecto provenientes de vários compiladores e linguagens diferentes ou então ser específico a um só compilador.

Um *linker* produz código relocatável, isto é, que não utiliza referências de memória fixas, podendo assim ser carregado numa região de memória arbitrária. Antes de executar o programa, é necessário ajustar todas as referências de memória segundo o espaço de endereçamento do sistema alvo para que a aplicação funcione correctamente. Num computador pessoal é o sistema operativo que está encarregue desta tarefa. Num sistema integrado não existe sistema operativo e as aplicações são geralmente guardadas em ROM, em endereços fixos. Por esta razão, quando se produz código para um sistema integrado, é necessário avançar um passo para além da *linkagem* no sentido de atribuir valores fixos e pré-determinados a todas as referências de memória do programa, antes da transferência deste para o sistema alvo. Esta função é feita pela ferramenta designada por *locator* a qual é, normalmente, anexada ao *linker*.

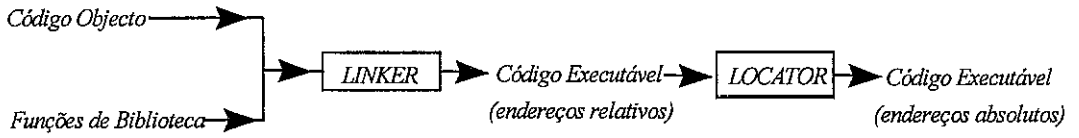

Figura 3. 3 - Funcionalidade de um linker-locator.

O conjunto compilador/*linker* constitui uma das ferramentas mais importantes no desenvolvimento de um sistema integrado. Destas ferramentas depende a qualidade do código gerado, a qual se pode medir basicamente por dois factores: tamanho e velocidade do código. Estes dois factores variam inversamente. É possível optimizar um programa para que este ocupe menos espaço ou para que funcione mais rapidamente, mas, geralmente, é impossível conseguir o nível máximo de optimização nas duas vertentes. Entre as optimizações efectuadas por um compilador podem ser encontradas as seguintes:

- **Alocação global de registos e variáveis** - Os registos do μC são alocados de forma a minimizar os acessos à memória central. O espaço ocupado por variáveis é re-utilizado, se possível.
- **Remoção de código invariante de ciclos** - Código que produza o mesmo resultado em cada iteracção é removido para fora desse ciclo.

- **Propagação de cópias de constantes** - Valores constantes são propagados pelo código, em vez de serem armazenados em memória central e mais tarde recuperados.
- **Combinação de instruções** - Instruções semelhantes são combinadas numa única (p. ex. duas adições -> uma adição).
- **Eliminação de acessos a memória desnecessários** - O compilador recorda-se dos valores de cada registo e da sua origem e aproveita-se deste facto para eliminar operações de acesso à memória desnecessárias.
- **Eliminação de código morto** - O compilador não armazena valores em variáveis que não são mais utilizadas.
- **Optimização de saltos** - Combina várias instruções de salto numa única.
- ***Inlining* de funções** - Em alguns casos, substitui chamadas a funções pelo próprio código da função.
- **Simplificação da passagem de parâmetros** - Se for chamada uma função que não utiliza parâmetros (ou utilize poucos parâmetros), é gerado código de chamada mais simples e rápido, mas que não respeita as convenções normais de passagem de parâmetros.

A qualidade do código gerado pelo compilador é importante, mas não é de forma alguma o único factor a ter em consideração na sua avaliação. Como um compilador é apenas uma de entre muitas ferramentas necessárias, é essencial garantir que são todas compatíveis entre si. A compatibilidade deve estar (relativamente) assegurada quando são utilizadas ferramentas provenientes do mesmo fornecedor, mas não existe nenhuma garantia de compatibilidade (pelo contrário...) entre ferramentas de fornecedores diferentes. Esta é uma possível razão para se preferir a utilização de um conjunto de ferramentas provenientes de um mesmo fornecedor.

Uma outra questão importante reside na rapidez de compilação. A diferença dos tempos de compilação do mesmo programa entre dois compiladores diferentes, com o mesmo nível de optimização, pode atingir vários minutos. Considerando que pode ser necessário recompilar o código várias vezes durante a fase de *debugging* do programa, constata-se que o tempo de compilação pode por vezes tornar-se um factor a ter em conta no tempo total de desenvolvimento do sistema. Este factor pode depender do computador hospedeiro podendo, em muitos casos actuais, não ser já significativo.

Outras características importantes que um compilador cruzado deve apresentar resultam das particularidades dos sistemas integrados. O conjunto compilador/*linker-locator* utilizado no desenvolvimento de um sistema integrado deve permitir a divisão de memória entre zona de dados e zona de código, e por vezes entre memória interna e externa ao microcontrolador. Num computador pessoal, tanto o código como os dados estão colocados em RAM externa, sendo os programas carregados a partir de um disco rígido ou disquete. O mesmo não acontece num sistema integrado. Neste, o código das aplicações é armazenado em ROM, poderá existir memória interna e externa ao microcontrolador e pode ser desejável considerar ainda um espaço de endereçamento para I/O. O compilador tem de permitir a diferenciação entre estes espaços de endereçamento quando são declaradas variáveis e constantes. Além disso, o projectista precisa de trabalhar com algumas estruturas que deverão estar localizadas em endereços pré-definidos (como, por exemplo, a tabela de interrupções). Sendo assim, é necessário que o compilador/*linker-locator* forneça mecanismos que permitam ao projectista especificar a localização em memória das várias funções e estruturas de dados que compõem o programa.

Não se pode deixar de considerar algumas questões relativas à própria implementação do compilador. Existirá um tamanho máximo para os ficheiros tratados pelo compilador, quer a nível de código fonte, quer a nível de código objecto? Existem compiladores mais antigos que apenas admitem ficheiros de código fonte com um tamanho máximo de 64k. Esta limitação pode tornar-se inaceitável mesmo quando se projecta um sistema relativamente simples.

Uma outra questão diz respeito à compatibilidade do compilador. Se existirem algumas normas de definição da linguagem, até que ponto é que estas são seguidas pelo compilador? Esta questão pode ser relevante se no futuro se desejar utilizar um novo compilador. Se não existir um mínimo de compatibilidade assegurada com outras ferramentas, a migração para um novo compilador pode implicar alterações substanciais no código já escrito.

Uma questão não menos importante tem a ver com as funções de biblioteca oferecidas. Será que existem todas as funções necessárias ao sistema em desenvolvimento? Os núcleos para sistemas integrados são frequentemente distribuídos como funções de biblioteca de um determinado compilador, sendo também importante, por isso, a escolha do núcleo para determinar qual o compilador mais apropriado para a aplicação em desenvolvimento.

Estes são apenas alguns dos pontos a ter em atenção para a escolha de um compilador, não existindo, no entanto, uma fórmula fixa para fazer a selecção do mesmo.

3.3.5 - *Debugger*

Um *debugger* é utilizado para monitorizar e verificar o funcionamento correcto do programa quando este já está compilado e ligado. Um *debugger* cruzado é um *debugger* que permite testar, a partir de um sistema, uma aplicação que é executada num sistema diferente.

Os *debuggers* cruzados podem classificar-se segundo os seguintes aspectos:

- Meio de execução.
- Nível de *debugging*.

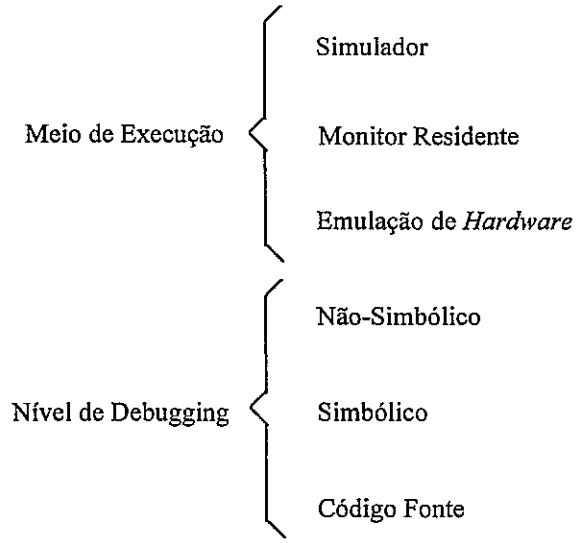

Figura 3. 4 - Diversos tipos de debugging.

É necessário que a aplicação em análise seja executada de forma controlada, para permitir o acesso ao seu estado interno e para controlar ou observar o seu fluxo de execução. Isto é conseguido através de uma de três alternativas:

- Simulação em *software* do processador e *hardware* necessários.
- Utilização de uma placa de desenvolvimento para onde é carregada e executada a aplicação. Estabelecendo-se, neste caso, a comunicação entre este sistema alvo e o sistema hospedeiro através de um programa monitor

que envia para o *debugger* cruzado o estado da aplicação em execução no sistema alvo.

- Utilização de emulação do *hardware* alvo para execução controlada da aplicação.

Um *debugger* pode ainda variar ao nível em que a depuração é efectuada. Podem identificar-se três níveis distintos: não-simbólico, simbólico e código fonte.

O *debugging* não-simbólico corresponde ao nível mais baixo de todos. Neste tipo de *debugging* o código da aplicação é apresentado na sua forma mais simples, ou seja, código "de-assemblado". Não existe qualquer relação visível entre o código apresentado e o programa em código fonte correspondente se este último tiver sido escrito em linguagem de alto nível.

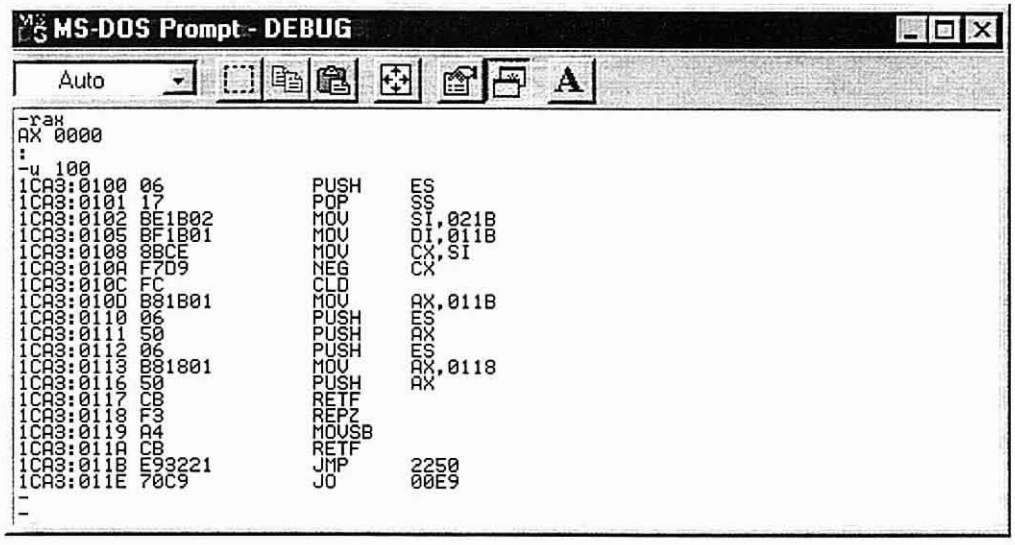

Figura 3. 5 - Debugger não simbólico: Debug do MS-DOS da Microsoft Corp.

O nível seguinte corresponde ao *debugging* simbólico. A aplicação continua a ser exibida sob a forma de código "de-assemblado", mas são mantidas referências ao programa em código-fonte da aplicação, tais como nomes de variáveis ou funções.

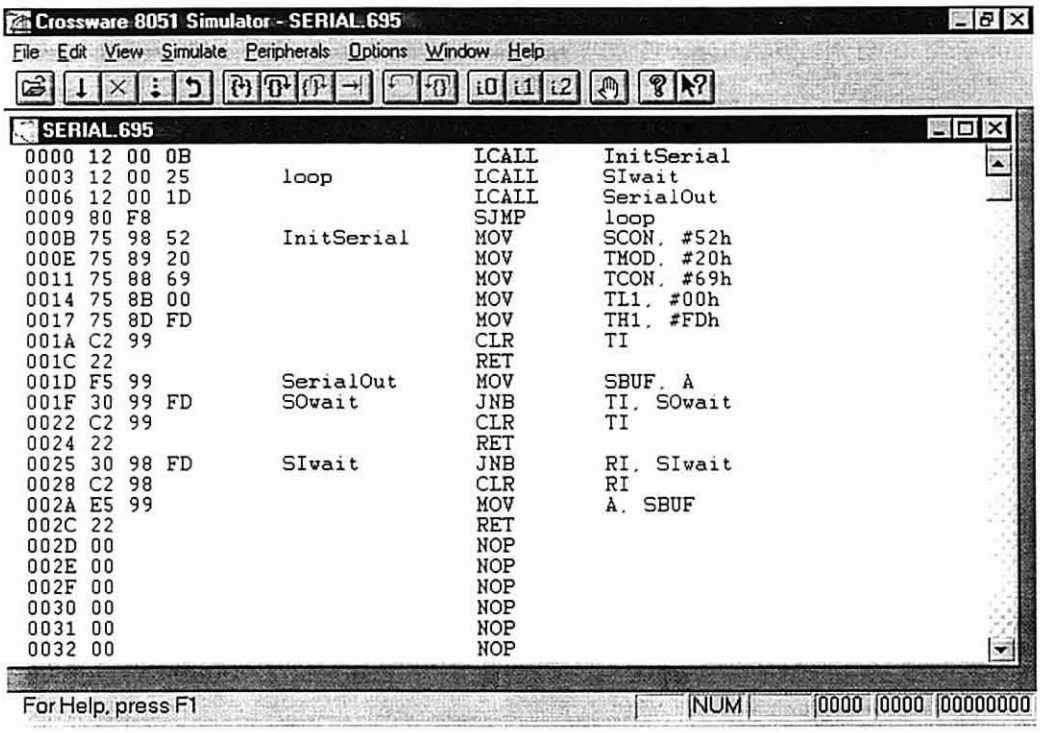

Figura 3. 6 - Debugger simbólico: Crossware 51 Simulator da Crossware Products.

Por fim, o *debugging* de mais alto nível corresponde ao *debugging* da aplicação através do próprio código fonte, em alto nível, da aplicação.

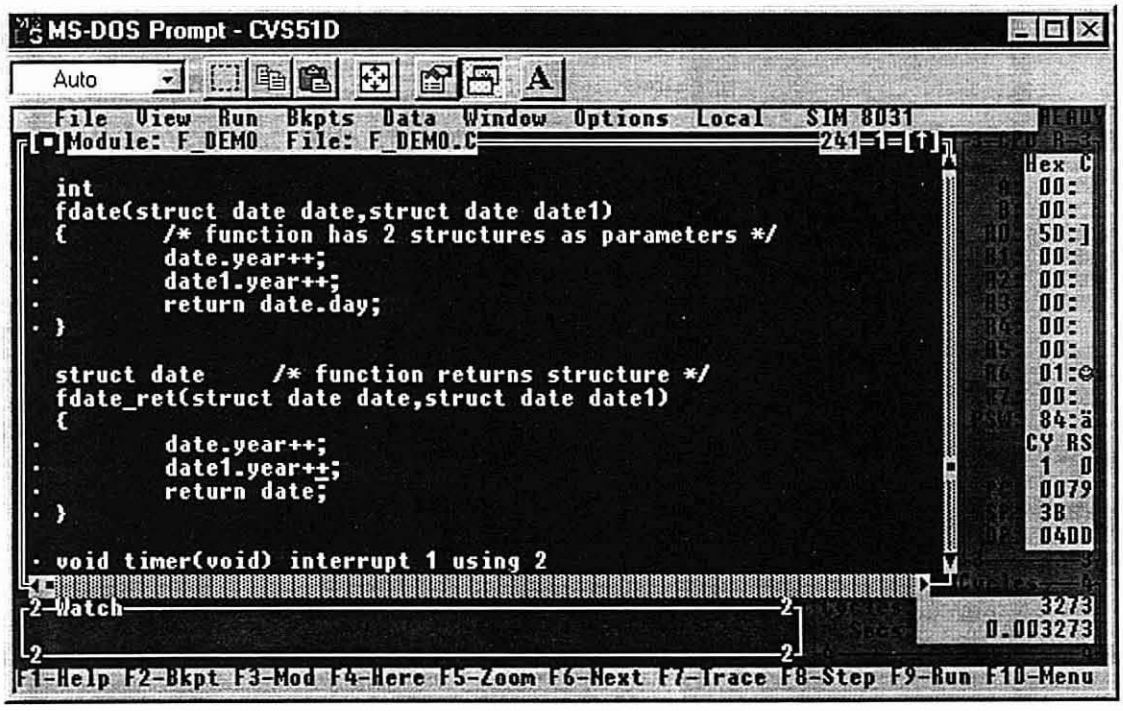

Figura 3. 7 - Debugger de código fonte: ChipView 51 da ChipTools Inc..

Um *debugger* deve oferecer, pelo menos, as seguintes funções:

- Execução do programa passo a passo (instrução a instrução).
- Execução do programa até um determinado ponto (*breakpoint*).
- Inspecção e alteração dos dados utilizados pelo programa em cada instante (registos e conteúdo da memória).

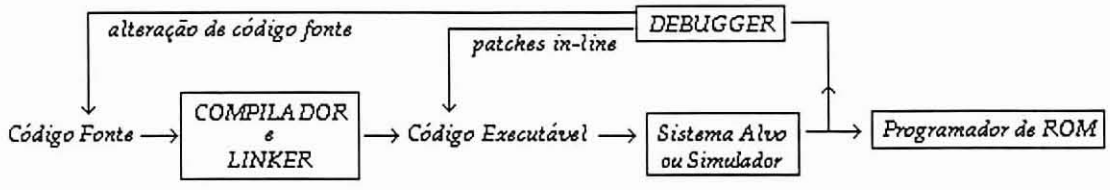

Figura 3. 8 - Funcionalidade de um debugger cruzado.

Quando é utilizado um núcleo, pode ser útil ao projectista aceder ao seu estado interno. As suas estruturas de dados e outra informação que pode ser relevante para o teste do sistema estão "escondidas" dentro do núcleo, pelo que, se o projectista não souber exactamente o que procurar, nunca irá conseguir obter toda a informação que deseja.

Existem ferramentas de auxílio ao *debugging* que têm a função de mostrar o estado interno do núcleo. Quando combinadas com um *debugger*, permitem ao utilizador inspeccionar o estado do núcleo do sistema (número e estado de tarefas, mensagens, semáforos, etc) quando, por exemplo, a execução do programa pára num *breakpoint*.

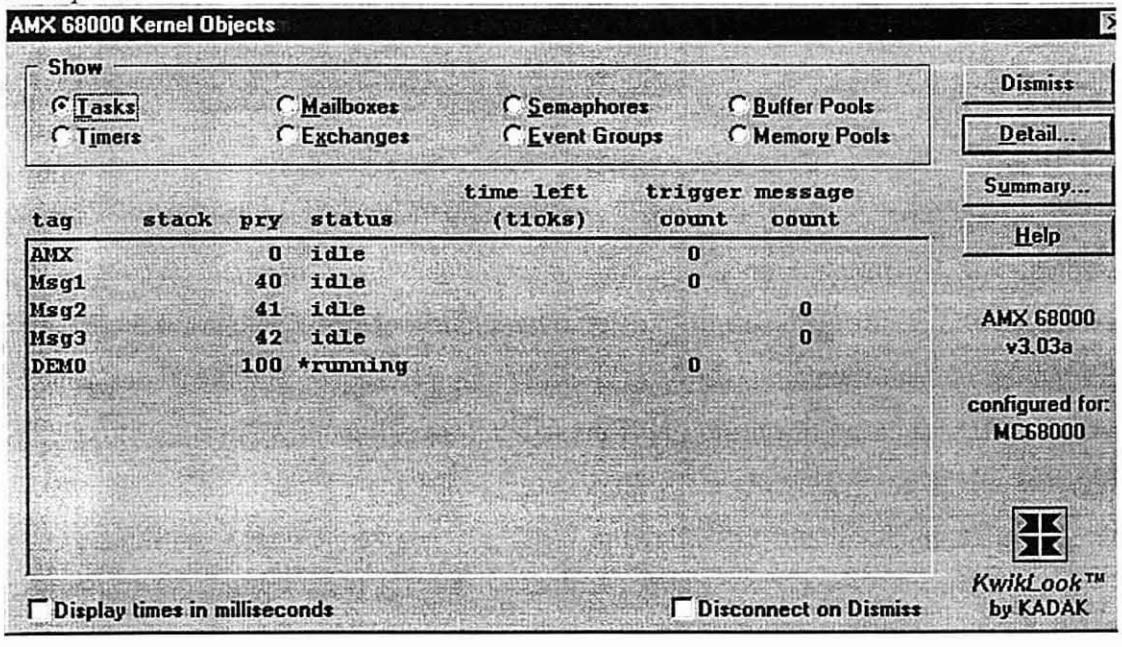

Figura 3. 9 - KwikLook Manager para inspecção do núcleo AMX da Kadak Production Ltd.

3.3.6 - Ferramentas de Análise de Desempenho

As ferramentas de análise de desempenho fazem a análise da velocidade e do tamanho do código, permitindo obter um perfil do comportamento do programa em execução. A análise do programa é feita a vários níveis, nomeadamente a nível do módulo, do procedimento e da linha de código. Esta análise deve indicar, entre outros parâmetros, o tempo que o programa demora a correr, o tempo que os componentes individuais (funções e procedimentos) demoram a ser executados e quantas vezes estes são invocados.

As aplicações básicas deste tipo de ferramentas incluem:

- Medição do tempo de latência de interrupções.
- Identificação de secções de código que ocupem uma grande percentagem da carga temporal do sistema. Estas são as zonas do programa que mais beneficiam com a optimização.
- Identificação dos componentes do sistema que não foram activados durante um determinado teste.

Normalmente, este tipo de ferramenta vem já integrado com outras, nomeadamente emuladores. Podem também constituir unidades independentes que são depois integradas no ambiente de desenvolvimento.

3.3.7 - Simulador

Os simuladores podem ser divididos em duas classes distintas: simuladores de *software* e simuladores de *hardware*. Um simulador de *software* representa o sistema final através de um programa que corre no computador hospedeiro, enquanto um simulador de *hardware* utiliza componentes físicos para representar o sistema final.

O uso de simuladores pode revelar-se muito útil porque permite o desenvolvimento e *debugging* de *software* para *hardware* que não esteja ainda disponível (neste caso é comum a utilização de simulação por *software*) ou no qual não seja possível testar o *software* directamente. Este último caso pode ocorrer em equipamento cuja operação incorrecta (mesmo que apenas para efeitos de teste) poderá ser inadmissível ou mesmo perigosa. Nestes casos é vulgar a utilização de uma simulação em *hardware* dos componentes que serão controlados pelo sistema integrado. Uma outra vantagem dos simuladores reside no facto de facilitarem a

experimentação de novas configurações por ajuste de algumas características do sistema final no simulador.

O uso de um simulador tem, no entanto, uma grande limitação. De facto, quando se trata de um sistema final que usa dispositivos I/O e respectivas interrupções, é difícil, senão impossível, reproduzir com precisão e exactidão o ambiente em que funciona. Neste caso, os resultados obtidos podem não ser representativos da realidade, sendo a sua validação uma das maiores dificuldades com que se depara a simulação.

3.3.8 - Ferramentas para Ligação entre Sistema Alvo e Sistema Hospedeiro

A aplicação é desenvolvida no sistema hospedeiro, mas necessita de ser executada/testada no sistema alvo. Para que tal seja possível, é preciso que existam mecanismos para estabelecer um meio de comunicação entre os dois sistemas. Estes são utilizados para possibilitar o carregamento do código no sistema alvo e permitir o *debugging* cruzado, a partir do sistema hospedeiro.

3.3.8.1 - Monitor Residente

Um monitor é um programa armazenado na ROM do sistema alvo, servindo de interface entre este e o sistema hospedeiro. É utilizado, frequentemente, para permitir o *debugging* cruzado do sistema alvo. O monitor tem a capacidade de receber o código executável e comandos do sistema hospedeiro através de uma ligação série, obedecendo, normalmente, à norma de comunicação RS-232 ou outra semelhante. O código é guardado numa zona de memória (RAM) especificada pelo programador.

Quando o microcontrolador utilizado tem espaços de endereçamento distintos para código e dados, como acontece, por exemplo, com os microcontroladores da família MCS-51, então o *hardware* do sistema alvo tem que sobrepor os dois espaços de endereçamento, para permitir não só o carregamento do programa em RAM como também a sua execução.

Para ser possível a utilização de um monitor, é preciso que uma boa parte do sistema já esteja operacional (μP, RAM) e é, evidentemente, necessário desenvolver (ou adquirir) e instalar o próprio programa monitor. É, porém, de utilização relativamente simples, rápida e flexível. Um monitor pode, inclusivamente, ser

instalado no sistema final para que, se algum problema for detectado, seja possível identificar a causa da falha no próprio local [Rosenthal94].

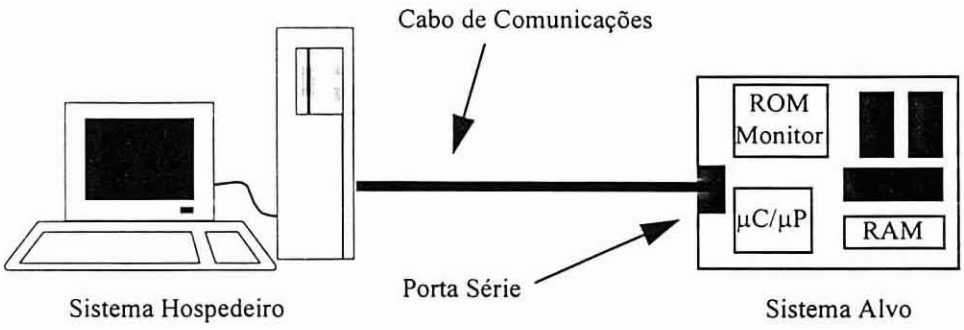

Figura 3. 10 - Monitor residente.

3.3.8.2 - Emuladores

Um emulador é *hardware* que substitui um componente do sistema final no sistema alvo, mas que apresenta facilidades de monitorização do comportamento do sistema que o componente real não possui.

Existem emuladores para vários componentes de um sistema, como, por exemplo, para o µP, para a memória RAM ou para periféricos. Cada um deles apresenta uma maior flexibilidade e diversidade de funções do que os componentes que substituem, o que facilita o processo de desenvolvimento e *debugging* do *software* de um sistema integrado. Por exemplo, se o emulador permitir alguma espécie de configuração ou calibração, pode ser possível testar o comportamento do programa em vários ambientes de *hardware* distintos, utilizando sempre o mesmo equipamento de teste.

3.3.8.2.1 - Emulador de µP

Um emulador de µP, também conhecido por ICE (*In-Circuit-Emulator*), substitui, no sistema alvo, o microprocessador que irá funcionar no sistema final [Ganssle90]. As funções principais de um emulador são:

- Análise em tempo real da execução da aplicação.
- Controlo da execução.
- Carregamento da aplicação.

O emulador carrega o código executável na sua memória ou na do sistema alvo (também se podem ligar periféricos utilizados no sistema final) e executa a aplicação em seguida, oferecendo as facilidades habituais de um analisador lógico e de um *debugger* (traço (*trace*) em tempo real, execução passo a passo, pontos de paragem, etc). Isto permite ultrapassar a principal deficiência de uma simulação em *software* que é, tal como já se referiu, a impossibilidade de testar o programa directamente com os componentes físicos. É uma ferramenta que oferece um grande controlo sobre a execução do programa dado que é o emulador que controla o seu fluxo, tendo-se, em qualquer instante, acesso ao estado do microprocessador e, consequentemente, ao estado do sistema. Além disso, esta monitorização é feita através de *hardware*, pelo que não é necessário o desenvolvimento de rotinas de *debugging* como no caso dos emuladores de memória, analisados seguidamente. Apesar de ser uma das soluções mais poderosas das apresentadas, é também uma das mais dispendiosas.

Nos tempos mais recentes, a tendência dos fabricantes de processadores é para, além de oferecer processadores competitivos, fornecer simultaneamente ferramentas de desenvolvimento de alta qualidade como sejam os emuladores de μP.

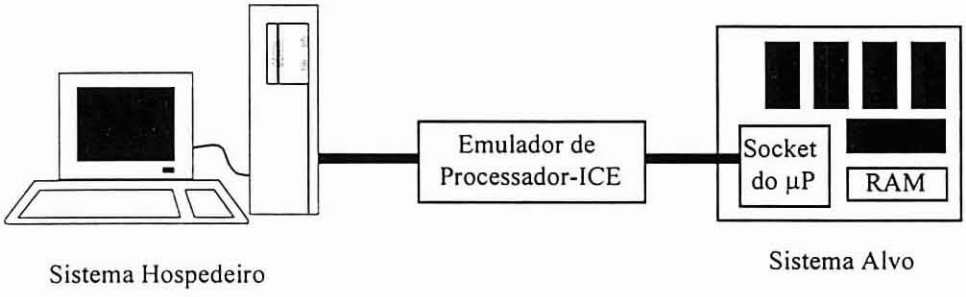

Figura 3. 11 - Emulador de processador.

3.3.8.2.2 - Emulador de Memória de Código

O emulador de memória é utilizado de forma análoga ao emulador de μP, só que neste caso é a memória de código do sistema alvo e não o μP que é substituída. O emulador é ligado ao sistema hospedeiro e a uma placa na qual já se encontra operacional um μP semelhante ao que irá ser utilizado no sistema final, ou mesmo a placa definitiva do sistema alvo. Tal como no caso do emulador de μP é também possível ligar a esta placa periféricos semelhantes aos que serão utilizados no funcionamento do sistema final. Uma vez gerado, no sistema hospedeiro, o código executável, é transferido para

o emulador, iniciando-se, posteriormente a execução do programa através de um RESET do μP.

O debugging é feito com recurso a rotinas de *software* tal como no caso de um programa monitor, pois tem de ser implementado usando mecanismos de execução passo a passo e pontos de paragem fornecidos pelo próprio μP. Quando se opta por esta técnica, à partida, a escolha e aquisição do processador já deve estar realizada e este convenientemente integrado no sistema alvo.

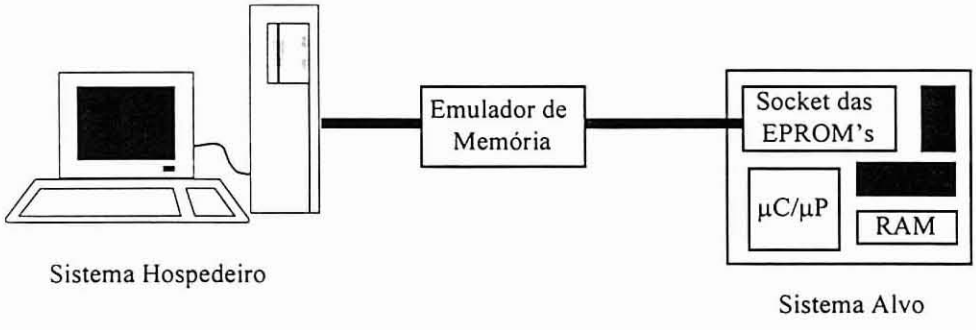

Figura 3. 12 - Emulador de memória.

3.3.9 - Instrumentação de Apoio ao *Debugging*

O funcionamento das ferramentas de *debugging* pode ser complementado através do recurso a instrumentação para análise e teste de *hardware*. Estes instrumentos permitem ao projectista observar os sinais eléctricos que percorrem as várias linhas condutoras no *hardware*. A monitorização de determinados sinais pode fornecer indicações sobre eventuais comportamentos anómalos do *software*. Por exemplo, podem ser provocadas interrupções esporadicamente e sem razão aparente por causa de ruído na linha de pedidos de interrupção [BlakesleeLiband93].

Entre os instrumentos utilizados contam-se:

- **Analisador Lógico** - é um dispositivo para monitorização das linhas do *bus* de um determinado sistema. Todos os pinos do processador podem ser monitorizados ou, pelo menos, uma significativa parte deles. O fluxo de execução do programa e o estado do sistema são examinados observando o fluxo de dados e instruções entre o μP e a RAM/ROM. Uma característica muito importante dos analisadores lógicos no *debugging* de sistemas em tempo real é a análise do comportamento do sistema de uma forma não intrusiva. O analisador lógico pode funcionar conjuntamente com um

debugger, permitindo analisar o funcionamento do sistema relativamente ao programa fonte.

- **Osciloscópio** - pode ser utilizado para tentar identificar a causa de uma falha no funcionamento do sistema, analisando o comportamento eléctrico dos sinais que circulam nas linhas do sistema. O osciloscópio é bastante útil para identificar pequenas interferências transitórias nestas linhas.

3.4 - Ambientes de Desenvolvimento Cruzado Comerciais

3.4.1 - Introdução

Um sistema (ou ambiente) de desenvolvimento é um conjunto diverso de ferramentas, compatíveis entre si, que oferecem várias facilidades necessárias nas fases de projecto e ensaio.

Os ambientes de desenvolvimento cruzado disponíveis comercialmente podem diferir em vários aspectos. Uma característica importante será qual ou quais o(s) sistema(s) alvo para os quais esse ambiente gera aplicações. Como exemplo, podemos considerar os ambientes de desenvolvimento para sistemas baseados em M68x00 ou i80x86. Ao contrário de ferramentas para outros sistemas alvo, existem no mercado compiladores genéricos para computadores baseados em processadores desta família. Os fabricantes aproveitam esse facto para apresentarem ambientes de desenvolvimento que trabalham em conjunto com estes compiladores, e, por norma, não tentam desenvolver um compilador próprio (se bem que existam muitas excepções).

Por outro lado, no caso de sistemas alvo baseados em pequenos microprocessadores como os da família MCS-51 ou Z80, nenhum sistema hospedeiro actual será baseado nesses microcontroladores, pelo que não existem ferramentas "nativas" que o projectista possa aproveitar. Neste tipo de situações, o projectista necessita que lhe sejam disponibilizadas todas as ferramentas necessárias para o desenvolvimento do seu sistema alvo.

Na secção seguinte são apresentados vários ambientes de desenvolvimento de vários fornecedores distintos. É feita uma breve descrição do produto e são expostas tabelas com informação relativa aos componentes individuais de cada ambiente de desenvolvimento.

3.4.2 - Exemplos de Ambientes de Desenvolvimento Comerciais

Apresenta-se, em seguida, a carteira de produtos de algumas companhias que produzem ferramentas cruzadas para desenvolvimento de sistemas integrados. São expostas, em formato de tabela, as características das ferramentas que compõem o ambiente de desenvolvimento. É indicada a existência de compiladores, quais as linguagens suportadas, a existência de um interface a partir do qual se possam utilizar todas as outras ferramentas, etc.

- **Companhia: Crossware Products**

	Série XPC	Série XDS	Série PSDS	WSM
Compilador	ASM	ASM, *linker*	C, ASM, *linker*	
Monitor				
Hardware				
Debugging				Simulador/*debugger*
Ambiente de desenvolvimento integrado	Editor DOS, Ajuda *on-line*.	Editor DOS, Ajuda *on-line*, gestão de biblioteca, *make*.	Editor DOS, Ajuda *on-line*, gestão de biblioteca, *make*.	
Núcleo				

Tabela 3. 1 - Carteira de produtos cruzados da Crossware Products.

- **Sistemas Hospedeiros:**
  DOS e Windows.
- **Sistemas Alvos:**
  PSDS: 8051,68000 e CPU32.
  XPC: 8051,68000,68020,8048,Z80,8085,6301,6305,65C02,740,68HC11 e 8086.
  XDS: 8051,68000,68020,8048,Z80,8085,6301,6305,6809,65C02,740 e 68HC11.
  WSM: 8051.

A *Crossware Products* produz ferramentas de desenvolvimento para uma vasta gama de microprocessadores/microcontroladores. A sua carteira de produtos é composta principalmente por 3 *packages* distintos: o XPC, o XDS e o PSDS.

A *package* mais simples - a XPC - encontra-se disponível para uma grande gama de processadores: inclui um *assembler* absoluto, um editor para DOS com ajuda *on-line* e um emulador de terminal para ligação a um sistema alvo.

A *package* XDS é semelhante à XPC, mas inclui um *assembler* relocatável, o respectivo *linker/locator* e utilitários para gestão do projecto (*make*, gestor de biblioteca). O código gerado é compatível com a norma IEEE695 [GrayMulchandani97] para permitir *debugging* a nível de código fonte com *debuggers* que suportem essa norma. O código compilado com os *assemblers* da série XPC pode ser compilado com os *assemblers* da série XDS sem problema.

A gama mais completa é a PSDS, a qual inclui um *assembler* semelhante ao do XDS e um compilador de C. As funções C de biblioteca incluem as funções *standard*, como funções para aritmética inteira e de vírgula flutuante. O compilador de C é compatível com a norma IEEE695.

A *Crossware Products* disponibiliza também um simulador do 8051 para Windows (WSM). Este complementa as outras ferramentas, permitindo a execução e *debugging* do código gerado pelo *assembler* ou compilador de C. Permite definir pontos de paragem, detectar *overflows* da pilha, comunicação com o porto série, geração de interrupções e utilização do monitor e teclado do PC para I/O.

- **Companhia: Keil Software Incorporation**

	A51 - Macro Assembler Kit	C51 - C Language Compiler Kit	PK51 - Professional 8051 Development kit
Compilador	ASM (A51); *Linker* (L51); Conversor obj->hex (OH51)	C (C51); *Linker* (L51); Conversor obj->hex (OH51)	C (C51); ASM (A51); *Linker* (BL51)
Monitor			MON51
Hardware			
Debugging			DS51. (Simbólico através de simulador.);TS-51 (Interface para monitor e emuladores populares).
Ambiente de desenvolvimento integrado	PV51 (Editor p/DOS, gestão de bibliotecas (Lib51)).	PV-51 (Editor p/ DOS, gestão de bibliotecas, (Lib51)); PC-Lint : analisador semântico de C.	PV-51 (Editor p/ DOS, gestão de bibliotecas, (Lib51)); PC-Lint: analisador semântico de C.
Núcleo			RTX-51.

Tabela 3. 2 - Carteira de produtos cruzados da Keil Software, Inc.

- **Sistemas Hospedeiros:**
  DOS e Windows.
- **Sistemas Alvos:**
  MCS 51/151/251.

A *Keil Software* oferece um conjunto de ferramentas para desenvolvimento de sistemas baseados nos microprocessadores das famílias intel MCS 51/151/251. As partes integrantes das várias *packages* são:

- ProView51: um ambiente de janelas para DOS semelhante ao oferecido pelos compiladores para DOS da Borland.
- A51: um *assembler* para 8051 e MCS251. Produz código objecto no formato Intel OMF-51. Inclui as seguintes ferramentas:
  - ProView51.
  - L51: *Linker/Locator.*
  - Lib51: Criação e manutenção de Bibliotecas.
  - OH51: Conversor de código objecto para formato Hex.
- C51: compilador de C para MCS-51 e derivados. Permite a definição de funções reentrantes e interrupções; bibliotecas para manuseamento de *strings*, vírgula flutuante e I/O e produz código objecto no formato Intel OMF-51. Inclui as seguintes ferramentas:
  - ProView51.
  - PC-Lint: Um analisador semântico de código fonte C.
  - A51: *Macro Assembler Kit.*
  - L51: *Linker/Locator.*
  - Lib51: Criação e manutenção de Bibliotecas.
  - OH51: Conversor de código objecto para formato Hex.
- DS51: simulador e *debugger* para aplicações desenvolvidas em A51, C51 ou PLM/51 da intel. Inclui as seguintes ferramentas:
  - MON-51: Monitor residente.
  - TS-51: Emulador de Terminal para efectuar *debugging* cruzado com monitor ou alguns ICE's comerciais (*Nohau, Hitex, Intel*).
- BL51: *Linker* 2Mb com bancos de código.
- RTX-51: Núcleo de tempo real para 8051 com código fonte incluído.

♦ **Companhia: Nohau Corporation**

	EMUL-51
Compilador	*Assembler* e de-*assembler* para alterações *in-line* do código.
Monitor	
Hardware	Emulador, caixa externa opcional.
Debugging	*Debugging*, através do emulador, de aplicações geradas em alguns compiladores comerciais pré-definidos.
Ambiente de desenvolvimento integrado	Ambientes próprios para DOS ou Windows, ou através do ChipView da Chip Tools.
Núcleo	

Tabela 3. 3 - Carteira de produtos cruzados da Nohau Corporation.

- **Sistemas Hospedeiros:**
  IBM PC/ XT/ AT, PS/2 ou compatível, HP e Sun.
- **Sistemas Alvos:**
  EMUL-51: MCS-51 e variantes: AMD, Dallas, Intel, MHS, OKI, Philips, Siemens,Yamaha;
  EMUL-251: intel MCS-251.
  EMUL-196: intel 80C196
  EMUL-296: intel 80C296
  EMUL-16: Motorola Inc. 68HC16, 68300
  EMUL-68: Motorola Inc. 68HC11

A *Nohau Corporation* produz uma gama diversificada de emuladores para várias famílias de microprocessadores. Os emuladores podem ser instalados como uma placa de expansão do computador, ou então numa caixa externa ligada pelo porto série. O interface com o utilizador é feito através de um programa para DOS ou Windows, ou então pode funcionar juntamente com o simulador do 8051 da *ChipTools Inc.*. O interface permite a criação de macros, tais como estruturas do género REPEAT...UNTIL ou IF... THEN...ELSE para efeitos de teste da aplicação. Os resultados de uma sessão de teste podem ser armazenados num ficheiro em disco, permitindo uma análise posterior. Tem incorporado um *assembler* para alterações *in-line* durante a execução do código.

O utilizador pode definir pontos de paragem: em instruções de alto ou baixo nível, em acesso a dados, em sinais externos, em conjuntos de endereços ou na execução do programa fora de limites. Existe uma placa de expansão opcional que permite fazer a definição de pontos de paragem em função dos sinais do processador e *bus: rd, wr,* endereços, dados, *opcodes*, etc. O emulador é compatível com os formatos de ficheiros *Intel* HEX/OBJ/OMF/SYM, *Avocet*, *Archimedes*, *IAR*, *Franklin/Keil*, *2500AD*, *Intermetrics*, *Relms* e outros. Permite assim *debugging* simbólico para código gerado com compiladores da *Keil*, *Tasking*, *IAR* e outros.

Apresenta-se ainda, de forma mais sucinta, a informação relativa a mais dois ambientes de desenvolvimento cruzado dos quais foram recolhidos dados:

- **Companhia: Avocet Systems Incorporation**

	AVCASE51 Complete Development Package
Compilador	ASM; C; *linker*; *Make.*
Monitor	
Hardware	
Debugging	Simulador c/ *debugging* em alto nível.
Ambiente de desenvolvimento integrado	IDE para Windows; Gestão de biblioteca; Ajuda sensível a contexto.
Núcleo	

Tabela 3. 4 - Carteira de produtos cruzados da Avocet Systems, Inc.

- **Sistemas Hospedeiros:**
  Windows 95, Windows 3.1 e PC DOS.
- **Sistemas Alvos:**
  80C51XA, 80C51, 68K/68xxx, 68HC11, 68HC05, Z80, Z180, 64180, Z8, 6809, 8048, TMS320C10, TMS320C20, 8085, H8/300, H8/500, 8096/C196KB/KC, 65816.

♦ **Companhia: Archimedes *Software* Incorporation**

	IDE - 51
Compilador	ANSI; C; *Macro Assembler*; *Linker* relocatável.
Monitor	
Hardware	
Debugging	*Debugger* de C/ASM; Simulador para o *chip* respectivo e alguns periféricos.
Ambiente de desenvolvimento integrado	Ambiente IDE; Editor; *Make*; *Help on-line*.
Núcleo	

Tabela 3. 5 - Carteira de produtos cruzados da Archimedes Software, Inc.

- **Sistema Hospedeiro:**
  Windows(3.1).
- **Sistemas Alvos:**
  MCS-51, MCS-251

3.5 - Ciclo de Desenvolvimento

3.5.1 - Introdução

Esta secção expõe as várias fases do ciclo de desenvolvimento de um sistema integrado. Este pode decompor-se em dois ramos distintos: o desenvolvimento do *software* e o desenvolvimento do *hardware*. Vai analisar-se com mais pormenor o processo de desenvolvimento de *software*.

Cada fase do ciclo de desenvolvimento de *software* é analisada individualmente. É feita uma breve descrição de cada fase e são expostas algumas ferramentas e a sua forma de utilização.

O processo típico de desenvolvimento de sistemas integrados é esquematizado na Figura 3. 13.

Qualquer etapa do processo de desenvolvimento admite retrocesso para uma das etapas anteriores ou início da etapa corrente, caso seja detectado um problema que exija uma reformulação ou reconstrução do sistema. A actividade de teste destina-se a verificar quando é necessária a ocorrência deste retrocesso. Esta actividade é analisada em maior detalhe na secção 3.6.

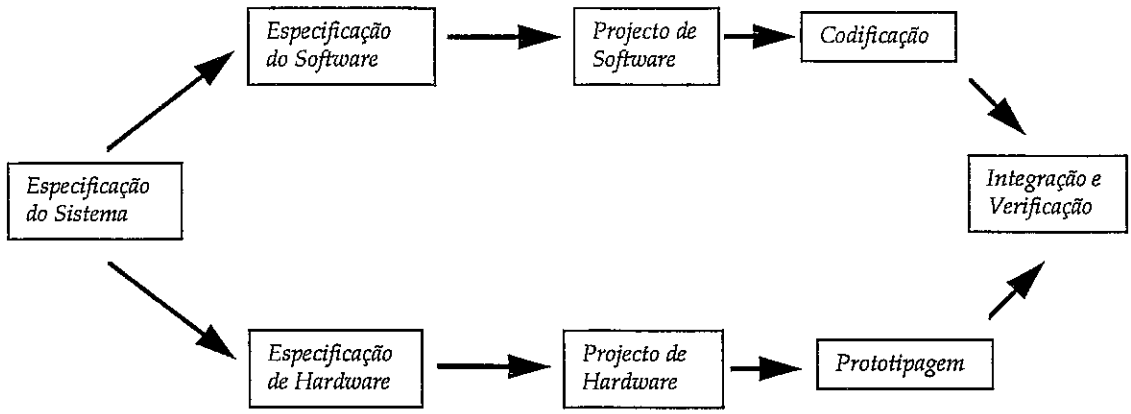

Figura 3. 13 - Processo de desenvolvimento de um sistema baseado em microcontrolador.

Para mais informação sobre o ciclo de desenvolvimento de sistemas integrados e suas etapas consultar [Švéda92], [Cooling91], [Walls96a] ou [Rosenthal96].

3.5.2 - Especificação de Sistemas Integrados

Quando se pretende desenvolver qualquer sistema em geral, o primeiro passo a dar nesse sentido consiste na especificação do mesmo, isto é, na descrição, o mais explícita e o menos ambígua possível, dos requerimentos e da funcionalidade do sistema pretendido. Erros que eventualmente se cometam nesta fase inicial podem tornar-se em erros extremamente dispendiosos e difíceis de corrigir se tardiamente detectados. Consequentemente, várias ferramentas e métodos foram desenvolvidos para ajudar e melhorar o processo de especificação.

Quando se estabelecem as especificações de um sistema existem duas vertentes bem distintas a contemplar:

- Definição dos objectivos do sistema.
- Definição dos objectivos do projecto.

Os objectivos do sistema são elaborados pelo cliente. Aqui, o cliente indica que sistema pretende e quais os objectivos que o mesmo deve atingir. Estes constituem frequentemente a base do acordo contratual entre o cliente e o projectista que desenvolve o sistema.

Os objectivos de projecto são desenvolvidos pelos projectistas tendo por base o conhecimento dos objectivos do sistema. Num sistema integrado, estas especificações podem subdividir-se em especificações de *hardware* e especificações de *software*. A

especificação dos objectivos do projecto revela-se por vezes bastante difícil devido a especificações incompletas, ambíguas ou mesmo erradas vindas do cliente. Daí a necessidade de existir uma interacção constante entre o cliente e o projectista ao longo do desenvolvimento do sistema, um pouco mais acentuada na fase inicial do projecto. Os objectivos de projecto devem incluir os aspectos de *hardware* e de *software* necessários para satisfazer as especificações do sistema exigidas pelo cliente.

As especificações de *software* são constituídas por detalhes sobre o funcionamento do sistema, como, por exemplo: actividades que o sistema deverá executar, temporizações necessárias para certas actividades, etc. Esta especificação indica qual deverá ser a funcionalidade do sistema final.

As especificações de *hardware* quantificam as características materiais do sistema. Estas incluem dimensões físicas (peso, dimensão), processador utilizado, memória disponível, mapeamento de memória e periféricos, entre outras.

A obtenção da especificação do sistema constitui apenas a fase inicial do ciclo de desenvolvimento.

3.5.3 - Projecto de *Software*

O projecto do *software* pode ser visto como sendo a utilização de formas de representação intermédias para indicar qual deve ser o resultado final, com o objectivo de servir de guia e de ilustração para a fase de codificação e desenvolvimento do código [Laplante93]. Entre estas formas de representação intermédia encontram-se os diagramas de fluxo, pseudo-código, autómatos de estado finito (FSM), redes de *Petri*, notação de *Warnier-Orr*, cartas de estado (*statecharts*) e outras.

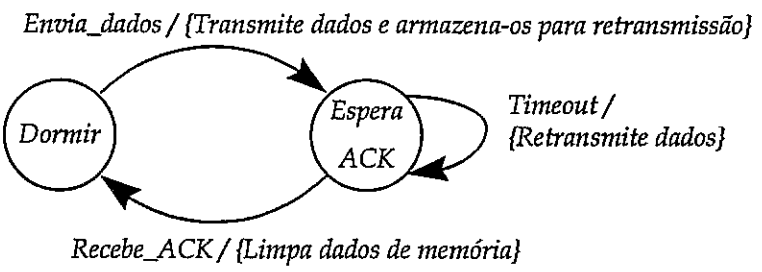

Figura 3. 14 - Autómato de estado finito - Transmissão de dados com ACK positivo.

Para sistemas muito simples, esta fase por vezes é ignorada. Os projectistas passam directamente da especificação do sistema para a fase de codificação do *software*. No entanto, existem técnicas, relativamente simples, que favorecem uma abordagem mais estruturada do problema e não obrigam a um grande dispêndio de esforço por parte do projectista. A título de exemplo, apresentam-se duas destas técnicas de projecto.

A Figura 3. 14 apresenta a representação em FSM de código para transmissão de dados com ACK positivo. Trata-se de uma técnica de especificação de sistemas que assenta no facto de que muitos sistemas podem ser representados por um número finito de estados. A mudança de estado depende do tempo ou da ocorrência de um acontecimento específico. A codificação do sistema por estados traz a vantagem de ser um método simples e de se prestar a uma codificação quase imediata. Tem a desvantagem de não conseguir representar sistemas concorrentes (a não ser que se utilizem múltiplos autómatos) e de ser por vezes demasiado simples para representar alguns sistemas reais.

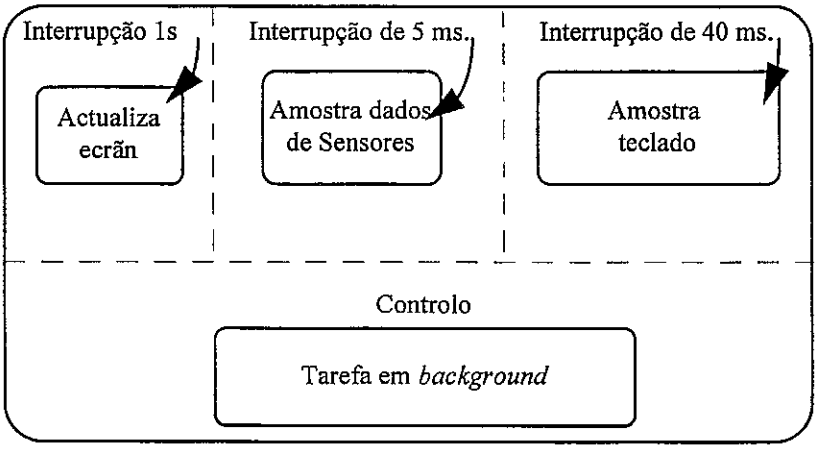

Figura 3. 15 - Representação através de uma carta de estado de um sistema de controlo com um teclado, ecrã e sensores.

Uma carta de estados é uma representação mais evoluída integrando aspectos das FSM e diagramas de fluxo de dados. Permite representar concorrência e favorece o desenvolvimento modular, já que os estados são hierarquizados, podendo-se representá-los em vários níveis de detalhe. Existem ferramentas (p.ex *STATEMATE* da *i-Logix*) que, a partir de uma carta de estados, efectuam a simulação do sistema e geram o código fonte da aplicação automaticamente.

Para exemplos de outros métodos e/ou ferramentas, consultar [Sacha93], [Antoniazzi++93] ou [Drusinsky95].

3.5.4 - Desenvolvimento e Codificação do *Software*

As representações ou esquemas obtidos na fase de desenvolvimento anterior são utilizadas para guiar esta fase, a qual envolve a escrita do código fonte, compilação, *debugging* e ensaio do código. O programador recorre a um sistema de desenvolvimento instalado no sistema hospedeiro, mas pode ser necessária a existência de um sistema alvo para execução/*debugging* da aplicação desenvolvida.

3.5.4.1 - Instalação de um Sistema de Desenvolvimento

É descrita nesta secção a instalação de um sistema de desenvolvimento cruzado no sistema hospedeiro. São abordadas questões relacionadas com a configuração do sistema de desenvolvimento e com os métodos de comunicação entre o sistema hospedeiro e o sistema alvo.

- **Configuração do sistema**

  Aquando do ciclo de desenvolvimento da aplicação, os sistemas hospedeiro e alvo comunicam entre si, normalmente, através de uma ligação série. Um meio de comunicação possível, e o adoptado na maioria dos ambientes de desenvolvimento cruzado, consiste no uso de um programa monitor, o qual, uma vez instalado no sistema alvo, servirá de interface entre este e o sistema de desenvolvimento hospedeiro. Uma outra possibilidade é a utilização de um emulador de processador (*ICE*).

  Qualquer que seja o meio de comunicação escolhido, um dos primeiros passos a tomar diz respeito à configuração dos parâmetros de comunicação entre o sistema alvo e o hospedeiro. Esta configuração consiste num conjunto de parâmetros que definem o porto de comunicação utilizado, o *baudrate* (geralmente entre 9600 e 115200), a interrupção utilizada, entre outras características.

  Por vezes, é necessário definir, no sistema hospedeiro, as características do *hardware* do sistema alvo, nomeadamente o seu mapa de memória, tipo de processador utilizado, etc.

Se for usado um monitor, é preciso indicar qual é o código de inicialização que lhe será agregado para arranque do sistema alvo; e isto porque o código de inicialização depende do *hardware* alvo utilizado. Normalmente, os ambientes de desenvolvimento são fornecidos com várias versões do código de inicialização, versões estas já préconfiguradas para determinados sistemas alvo. No entanto, o programador tem geralmente a possibilidade de adaptar o código de inicialização às características específicas do seu sistema particular. No caso de um ICE, será a própria aplicação do utilizador que deverá incluir o código de inicialização do sistema porque neste caso essa aplicação será o primeiro código que o processador irá utilizar.

- **Estabelecimento da comunicação entre o sistema hospedeiro e o sistema alvo**

  A utilização de um programa monitor implica que este deva ser programado em ROM para ser introduzido no sistema alvo. Para o conseguir, deve compilar-se e relocatar-se o código do monitor segundo o mapa de memória do sistema alvo (esta informação deve estar já disponível no sistema hospedeiro). Resta então proceder à conversão do ficheiro binário para um formato adequado a um programador de PROM's (Intel *Hex*, por exemplo). Depois de programada, a PROM é inserida no sistema alvo.

  A comunicação com o sistema alvo é feita por um programa de comunicações residente no sistema hospedeiro, o qual envia o código de aplicação compilado, ligado e relocatado para o sistema alvo. Este é recebido e armazenado pelo programa monitor, o qual desencadeia a sua execução. Uma outra função deste programa de comunicações respeita à recepção de informação sobre o estado do sistema alvo, a qual é transmitida pelo monitor e pode ser utilizada, por exemplo, para efectuar a verificação e acompanhamento da execução do código com um *debugger* cruzado.

  A função do monitor pode ser desempenhada por um *ICE*. Este recebe o código da aplicação do sistema hospedeiro e armazena-o no sistema alvo. Um *ICE* pode aceitar vários tipos de ficheiros diferentes, desde ficheiros objecto até ficheiros Intel *HEX*. Tal como o monitor, o *ICE* pode funcionar juntamente com um *debugger* no sistema hospedeiro para fazer o *debugging* cruzado.

3.5.4.2 - Desenvolvimento de uma Aplicação no Sistema Hospedeiro

Uma vez instalado e operacional o ambiente de desenvolvimento, pode iniciar-se o desenvolvimento da aplicação propriamente dito.

- **Escrita e compilação do código fonte**

  Depois de escrito o seu código fonte, a aplicação a desenvolver deve ser compilada por um compilador compatível com as ferramentas utilizadas. Por exemplo, é normal sistemas hospedeiros, baseados num PC ou compatível, que desenvolvem código para um processador da família 80x86, permitirem a utilização de um dos três compiladores mais conhecidos no mercado, o Borland C/C++, MS Visual C, ou Watcom C.

- **Locação do código executável**

  Depois de obtido o ficheiro executável (*.EXE num PC), o projectista deve utilizar o *locator*[1] para transformar o código relocatável em código binário com endereços absolutos. O *locator* necessita de conhecer o mapa de memória do sistema alvo e as regiões de memória onde se pretende que os vários componentes do programa (código, dados, pilha, e outros) residam. Quando o *locator* é executado, adapta o código para este funcionar nas zonas de memória indicadas.

- **Carregamento e execução no sistema alvo**

  Por fim, depois de compilada e relocatada, a aplicação é enviada para o sistema alvo para que aí seja executada. Esta acção pode ser desencadeada por um comando semelhante a "Run APLICAÇÃO". O código é enviado pelo porto série para o sistema alvo, onde é recebido pelo monitor residente ou emulador, que o armazena em RAM e o executa em seguida, ou então é carregado e executado num simulador.

3.5.5 - Integração *Hardware/Software*

Uma vez desenvolvido o protótipo do *hardware* e o *software,* é necessário integrar estas duas componentes para obtenção de uma primeira versão funcional do sistema. Existem vários métodos para efectuar esta ligação, dependendo da forma como foi

[1] Os ficheiros *.EXE no MS-DOS e sistemas compatíveis são constituidos por código relocatável. As referências só são resolvidas quando a aplicação é lançada pelo sistema operativo, conforme os recursos que estejam disponíveis nesse momento.

desenvolvido o protótipo de *hardware*. Os protótipos de *hardware* normalmente enquadram-se numa das classes seguintes:

- Simulação em *software* do *hardware* alvo.
- Placa-protótipo.

Se o protótipo desenvolvido é uma simulação em *software* do sistema físico, os resultados obtidos podem não ser suficientemente representativos do sistema real. Nesta fase do desenvolvimento, é desejável o teste do *software* com o *hardware* que irá ser efectivamente adoptado no sistema final.

Uma placa-protótipo pode receber o *software* desenvolvido através de um de diversos métodos, tais como a programação de uma PROM ou através de um emulador de *hardware*. Deve, em seguida, testar-se o sistema para detecção de possíveis erros. Nesta fase pretende-se já observar o comportamento do sistema como um todo, pelo que quanto mais próxima estiver do sistema final a configuração protótipo+*software* mais representativos serão os resultados.

3.5.6 - Utilização de Ferramentas Comerciais durante o Desenvolvimento do Sistema

Esta secção descreve como alguns ambientes de desenvolvimento comerciais funcionam no sentido de se obter a aplicação final. São também apresentadas algumas ferramentas utilizadas como apoio a esse desenvolvimento, tais como ferramentas para projecto de *software* ou pesquisa de informação de referência. Esta secção apresenta-se subdividida nas várias fases do desenvolvimento, e em cada uma dessas sub-secções são apresentadas as ferramentas/ambientes relevantes para essa etapa.

3.5.6.1 - Projecto de Software

As ferramentas para auxílio ao projecto de *software* têm como entrada uma especificação do sistema. Esta especificação é validada pela ferramenta, sendo em seguida produzido código fonte automaticamente. Este código é utilizado pelo projectista com esqueleto para a aplicação a desenvolver.

Na tabela seguinte são apresentados exemplos de ferramentas para projecto de *software*.

	Representação intermédia das especificações do sistema	Geração automática de código
Statemate	Cartas de estado.	Código fonte em C ou ADA.
FuzzyLogic (MCS-96 & MCS-51)	Conjuntos de regras *"fuzzy"*.	Código fonte em C.

Tabela 3. 6 - Exemplo de ferramentas para projecto de software.

3.5.6.2 - Implementação

A implementação do *software* da aplicação passa pelo desenvolvimento do código fonte e posterior transformação desse código numa versão executável da aplicação.

- **Desenvolvimento do código fonte** - O editor de texto utilizado para gerar os ficheiros de código fonte pode estar especialmente concebido para esse efeito. Existem editores de texto que fazem a identação automática do código ou identificam palavras-chave da linguagem em cores distintas. Por vezes, o editor de texto encontra-se integrado conjuntamente com outras ferramentas (nomeadamente compilador, *linker* e *debugger*) no mesmo ambiente. No entanto, a edição e manutenção do código fonte implica não só a utilização de um editor de texto, mas também de ferramentas que permitam uma gestão mais ou menos automatizada das várias versões dos ficheiros que vão sendo produzidos. Os ficheiros gerados não são considerados individualmente, mas sim como fazendo parte de um projecto maior (que pode incluir outros ficheiros). Isto facilita, numa etapa posterior, a identificação dos ficheiros que se mantêm actualizados, quais os ficheiros que necessitam de ser recompilados, etc. Estes mecanismos são assegurados por ferramentas designadas normalmente por *make*.
- **Obtenção de uma versão executável da aplicação a partir do código fonte** - Este é o culminar da fase de projecto de *software*. A geração de código implica a utilização de um compilador para obtenção de código objecto da aplicação a partir do seu código fonte, a utilização de um *linker/locator* para ligação do código da aplicação com as rotinas de biblioteca utilizadas e, por fim, a alocação do código aos endereços indicados pelo programador. Alguns aspectos que influenciam o desempenho nesta fase são: a rapidez de

compilação, a *linkagem* e o formato dos ficheiros intermédios. A rapidez é importante para *debugging*, onde pode ser repetido várias vezes o ciclo: Gera Código Executável - Testa Código - Altera Código Fonte - Gera Código Executável - Testa Código.... O formato dos ficheiros intermédios é importante para garantir que existe compatibilidade entre os ficheiros gerados pelo compilador e *linker* com a ferramenta de *debugging* utilizada. Existem alguns formatos de código objecto que incluem informação simbólica sobre a aplicação, a qual pode ser utilizada para se conseguir o *debugging* simbólico desta com vários *debuggers* disponíveis comercialmente - exemplo: formatos OMF-51 ou IEEE695.

- **Pesquisa de informação de referência** - O programador de uma aplicação necessita invariavelmente de aceder a vários tipos de informação de referência durante a fase de codificação. Esta informação de referência pode ser relativa à sintaxe ou à semântica da linguagem usada, características do sistema alvo para o qual a aplicação se destina ou relativa às capacidades das ferramentas de desenvolvimento utilizadas. Este tipo de informação é facilmente integrável no ambiente de desenvolvimento utilizado.

	Edição e manutenção do código fonte	**Obtenção de versão executável da aplicação a partir de código fonte**	**Pesquisa de infor. de referência sobre ling. de desenv., inicialização de periféricos, etc**
Crossware PSDS	IDE DOS; Gestão de projectos	*Assembly* ou C; biblioteca C standard e numérica;Formatos OMF-51 ou IEEE695.	Ajuda *on-line* para ferramentas.
Archimedes IDE-8051	IDE DOS / Windows; Gestão de projectos	C; biblioteca C *standard* e numérica; Formato OMF-51	Ajuda *on-line* para ferramentas.
AvCase IDE	IDE Windows; Gestão de Projectos.	C ou *assembly*; Formato IEEE695; Bibliotecas com respectivo código fonte.	Ajuda *on-line* para ferramentas.
Keil PK51	IDE DOS / Windows; Gestão de Projectos	*Assembly* ou C; Biblioteca C standard e número; Formato OMF-51	Ajuda *on-line* para ferramentas.

Tabela 3. 7 Ferramentas comerciais para auxílio ao desenvolvimento de sistemas integrados.

Verifica-se que a maioria dos sistemas de desenvolvimento não cobre todas as necessidades de informação de referência do programador, pelo que surgem ferramentas dedicadas exclusivamente a esta tarefa. Entre estas encontra-se, por exemplo, a aplicação ApBuilder da Intel Corporation (fig. 3.16). Trata-se de uma aplicação de domínio público (grátis) que armazena informação de referência relativa a alguns dos microcontroladores desta companhia. A aplicação inclui os manuais dos referidos microcontroladores e gera automaticamente o código de inicialização para os vários periféricos dos controladores, conforme a funcionalidade pretendida pelo programador. O código é gerado em *assembly* ou em "C" e pode ser exportado para qualquer compilador/*assembler* para esse microcontrolador, servindo como plataforma base para a escrita do código fonte.

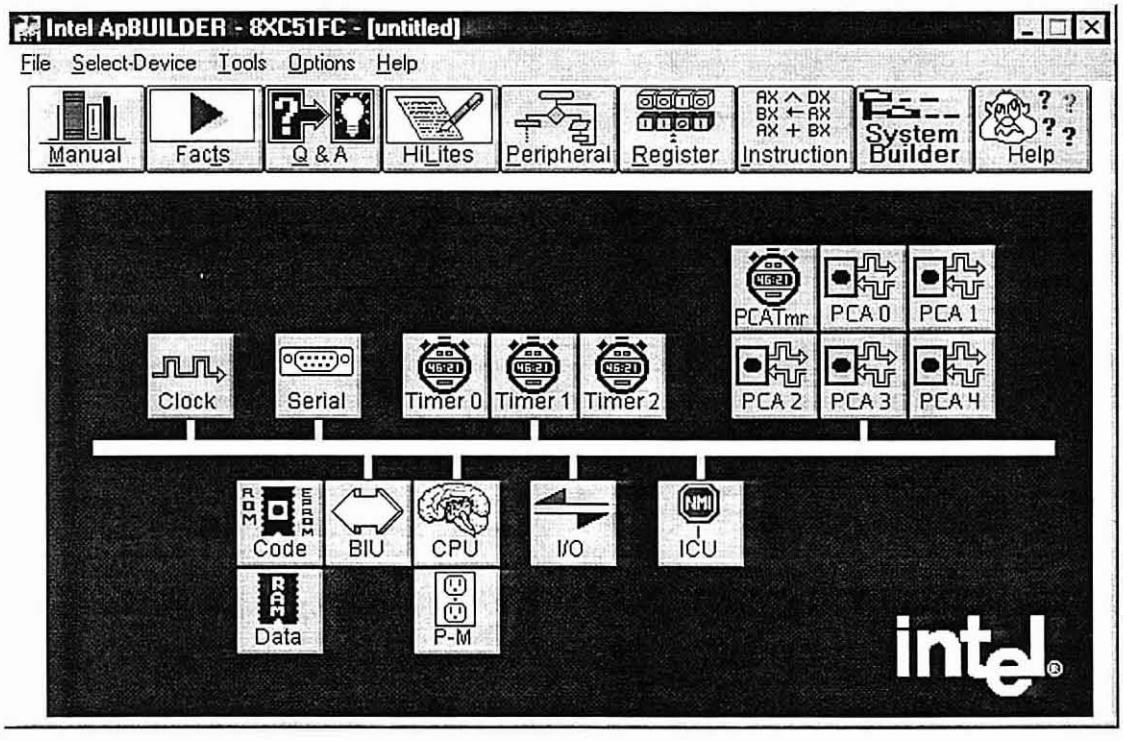

Figura 3. 16 - Aplicação para consulta de informação de referência: ApBUILDER da Intel.

3.6 - Verificação e Ensaio do Sistema

3.6.1 - Introdução

Nenhum programa se pode considerar totalmente isento de defeitos. Por mais testado e analisado que tenha sido, existem sempre situações excepcionais não previstas ou erros não detectados.

Um erro no funcionamento do sistema pode ter origem em inúmeras fontes, nomeadamente em erros de codificação, especificações incorrectas, ruído eléctrico, temporizações incorrectas, etc. Não será possível, normalmente, recriar todas as situações perante as quais o sistema terá de funcionar, pelo que o teste completo e exaustivo do mesmo se torna impossível. Por essa razão, o objectivo do *debugging* e teste de um sistema integrado não é a eliminação completa de todos os erros e falhas do sistema, mas sim a obtenção de um sistema com uma fiabilidade elevada e operacional [Rosenthal95a].

Um sistema operacional pode apresentar falhas no seu comportamento, desde que estas não diminuam a sua eficiência de tal forma que o mesmo se torne inutilizável para a função que lhe é destinada. Por exemplo, um computador pessoal pode facilmente bloquear-se uma vez por semana sem que o utilizador considere sequer a hipótese de o devolver ao fabricante. Por outro lado, se o sistema informático de uma instituição bancária falhasse uma vez por semana, essa instituição poderia sofrer prejuízos incalculáveis.

Um sistema integrado necessita de uma fiabilidade relativamente elevada. Estes sistemas geralmente funcionam autonomamente, de forma não supervisionada, pelo que pode ser complicado detectar a falha e recuperá-la de forma satisfatória. Além disso, estes sistemas, geralmente, controlam equipamento industrial que, pela sua natureza, não pode admitir falhas frequentes na sua operação. Uma outra razão para que os sistemas integrados necessitem de uma fiabilidade mais elevada deve-se ao facto de o grau de exigência dos clientes para com estes produtos ser muito elevado, pelo que a detecção de uma falha implica geralmente a devolução do sistema ao fabricante.

3.6.2 - Verificação e Ensaio no Ciclo de Desenvolvimento

Já se referiu que o ciclo de desenvolvimento de um sistema integrado admite retrocessos durante as suas várias etapas. A actividade de verificação (ou ensaio) determina se e quando esses retrocessos ocorrem.

Existem pelo menos três momentos em que o produto em desenvolvimento deve ser testado.

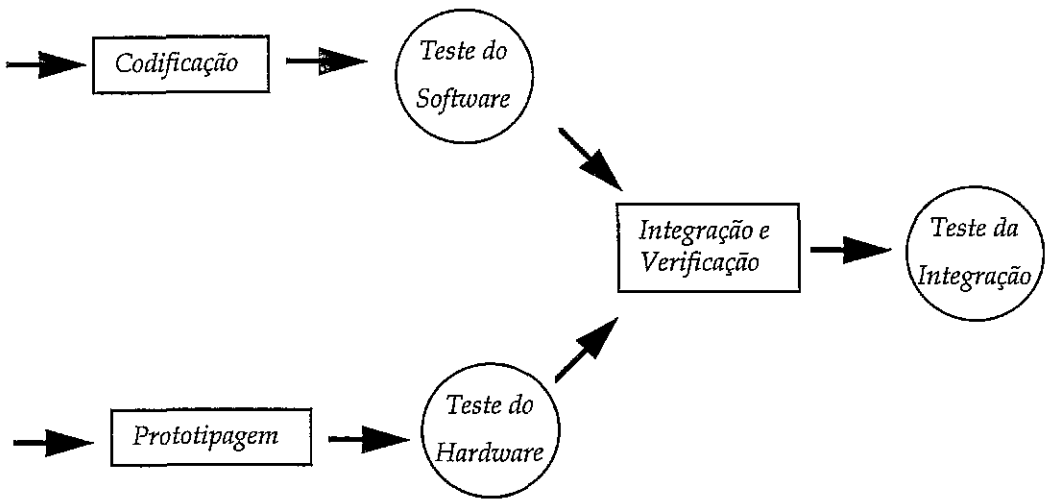

Figura 3. 17 - Teste e verificação no processo de desenvolvimento de um sistema integrado.

Por que é que é aconselhável testar o *hardware* e o *software* independentemente, dado que a validação do conjunto protótipo+*software* valida ambas as componentes?

Esta separação é útil para facilitar a identificação da origem de falhas. A identificação de erros, quando o teste é feito apenas no final do ciclo de desenvolvimento, pode ser difícil de resolver porque a causa do erro tanto pode ter tido origem numa falha de *hardware* como de *software*. Se se optar pela separação dos dois campos:

- O *hardware* é projectado e o seu protótipo testado na "ausência" do *software*. É óbvio que para testar o *hardware* será necessário *software* para verificar o comportamento correcto do sistema, mas este *software* não está necessariamente relacionado com o programa que deverá correr no sistema final [Ganssle95f].

- O *software* é desenvolvido na "ausência" do *hardware*. Usam-se ferramentas de *software* no sistema hospedeiro para se levar a cabo a compilação e o *debugging* inicial, sendo comum a utilização de um SBC (*Single Board Computer*) ou simulador para efeitos de teste e de verificação.

3.6.3 - Verificação e Ensaio do *Software*

A verificação da correcção do *software* inicia-se com a própria codificação. Durante a compilação do código, o compilador é capaz de detectar vários erros possíveis, tais como erros de sintaxe, variáveis não declaradas e outros. Estes erros são rapidamente detectados e facilmente corrigidos por alteração do código fonte e recompilação.

No entanto, nem todos os erros são detectados pelo compilador. Existe uma gama de erros que não é detectada durante a compilação e surge apenas durante a execução do programa *(run-time errors)*. Estes erros são de detecção e correcção mais difícil do que os erros de compilação, mesmo porque o erro poderá surgir apenas esporadicamente, sendo difícil saber em que circunstâncias ocorreu. Para correcção destes erros é necessário o recurso a um *debugger*, que permite acompanhar a execução do programa passo a passo, sendo assim possível identificar o momento no qual a falha ocorre.

A detecção dos erros de execução torna necessária a existência de uma plataforma onde o código seja executado. Se esta plataforma for o protótipo de *hardware* do sistema final, deve ter-se em atenção que as eventuais falhas de *hardware* e *software* podem ser confundidas.

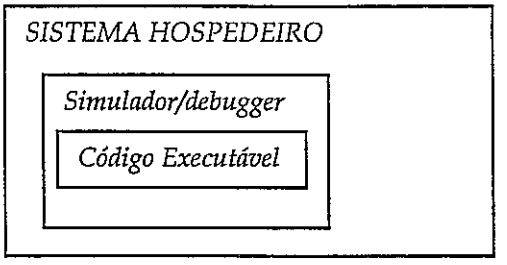

Figura 3. 18 - Debugging com simulador no sistema hospedeiro.

Uma possibilidade é a utilização de um simulador do *hardware* alvo (fig 3.18). O simulador corre no sistema hospedeiro e imita em *software* os componentes de *hardware* necessários para que o programa funcione. Neste caso, o programa não é testado com equipamento real pelo que os resultados obtidos podem não ser fiáveis [KattilakoskiHonka91].

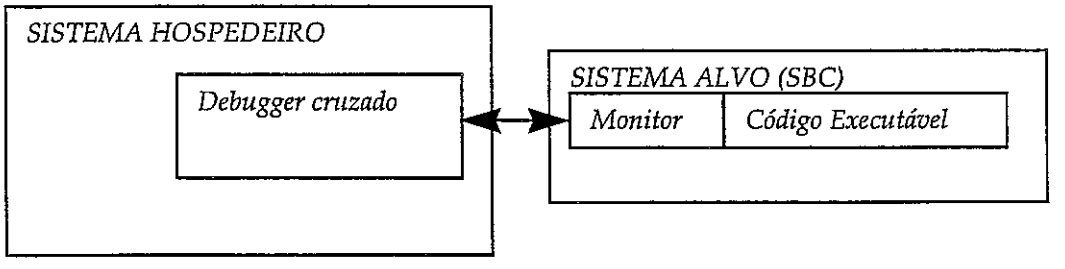

Figura 3. 19 - Debugging através de um SBC.

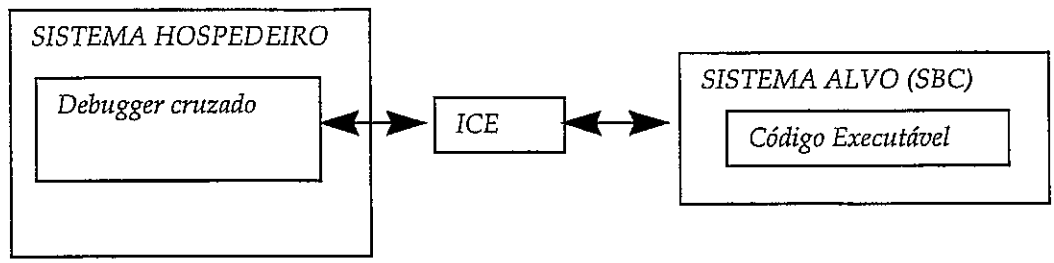

Figura 3. 20 - Debugging no sistema alvo através de ICE.

É também possível a utilização de uma placa de desenvolvimento (*SBC - Single Board Computer*). Esta é uma placa de *hardware* ligada ao sistema hospedeiro, normalmente através de uma ligação série tipo RS-232 [Seyer91]. O programa pode ser carregado e executado na RAM da placa, onde por norma se encontra um programa monitor (fig. 3.19). Este programa é utilizado para conduzir o *debugging* do *software*. Pode também ser usado um ICE para efectuar a ligação Sistema Hospedeiro -> *SBC*, eliminando assim a necessidade de existência de um monitor (fig. 3.20). O *ICE* também pode ser utilizado independentemente, sem o *SBC* (fig. 3.21). Neste caso, o programa é armazenado e executado internamente no emulador.

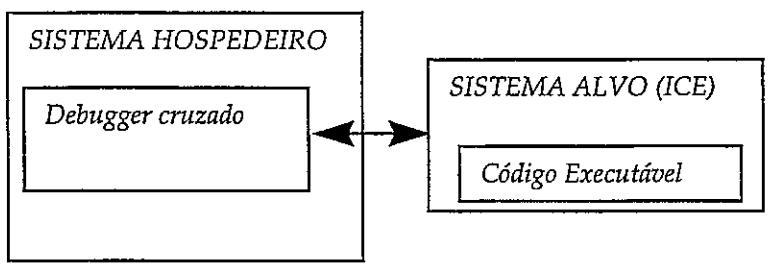

Figura 3. 21 - Debugging através de um ICE, sem hardware adicional.

A optimização do código efectuada pelo compilador pode dificultar o *debugging*. O processo de optimização torna o código muito menos legível. Além disso, a relação entre uma instrução do código fonte e a sua tradução em código máquina torna-se um pouco difusa, já que a optimização pode alterar a ordem de instruções e inclusivé retirar ou acrescentar instruções novas. Por esta razão, o *debugging* de um programa é feito geralmente sobre uma versão não optimizada. Apenas quando o código já está completamente desenvolvido é que deverá ser compilado no grau máximo de optimização.

3.6.3.1 - Teste e Verificação das Aplicações

Esta secção descreve as funcionalidades que uma ferramenta para teste e ensaio de uma aplicação deve fornecer. São expostas algumas possibilidades para a implementação destas funcionalidades, apresentando-se, por fim, várias ferramentas comerciais.

Um aspecto do teste e verificação de aplicações consiste na identificação e remoção de falhas, feita normalmente através da análise dinâmica (em execução) do programa. Uma outra vertente reside na obtenção de valores quantitativos do desempenho do programa e verificação do seu comportamento mediante certas entradas predefinidas.

- A identificação dinâmica de falhas na aplicação é conseguida através da monitorização do seu estado interno e da detecção de condições de erro. Conforme o nível de *debugging* oferecido, estas condições podem ser descritas a nível não-simbólico, simbólico ou de código fonte.
- A correcção *in-line* de falhas torna o processo de *debugging* mais rápido. A correcção das falhas encontradas é imediata e interactiva, sem necessidade de recompilação do código fonte. É evidente que, para que estas alterações tenham este efeito imediato, deverão ser efectuadas a nível das instruções de *assembly* e não do código fonte da aplicação, pelo que esta facilidade é geralmente oferecida apenas nos *debuggers* não simbólicos e simbólicos.
- O fornecimento de dados de entrada à aplicação pode ser efectuado de várias formas distintas:
  * Directamente pelo próprio programador, através do teclado do sistema hospedeiro, durante a execução da aplicação.
  * Pela leitura de um ficheiro onde foram previamente armazenados os valores necessários.
  * Através de uma linguagem de programação de macros para geração automática de sequências mais complexas de dados de entrada.
- A visualização do estado interno da aplicação pode ser efectuada a vários níveis: o primeiro nível inclui o estado interno do processador, o conteúdo da pilha, etc. O segundo nível de visualização é fornecido pelo *debugging* simbólico, sendo capaz de indicar o estado da aplicação através do valor de variáveis e outras estruturas definidas no programa. Tal como no nível anterior, a execução da aplicação é apresentada através de instruções *assembly*.

Por fim, o nível de apresentação mais abstracto consiste na utilização não só das estruturas de dados definidas pelo programador, mas também do código fonte da aplicação.

	Identificação dinâmica de falhas na aplicação	Correcção *in-line* de falhas	Estimulação das entradas do sistema através de vectores definidos pelo utilizador	Visualização do estado interno da aplicação
CrossWare51	Simbólica	Simbólica	Ficheiro ou teclado	Simbólica
Chip-View	Código Fonte		Ficheiro ou teclado	Código Fonte
SimCase	Código Fonte		Ficheiro ou geração de entradas através de macros	Código Fonte
DS51	Código Fonte		Ficheiro ou geração de sinais através de macros	Código Fonte
AVSim	Simbólica	Simbólica	Ficheiro ou Teclado	Simbólica

Tabela 3. 8 - Características de algumas ferramentas de verificação e ensaio.

3.6.3.2 - Debugging de Interrupções

A natureza assíncrona das interrupções dificulta o seu *debugging*. Um erro pode surgir esporadicamente, sem que se consigam determinar as condições que o despoletaram. As principais razões pelas quais um sistema que utilize interrupções pode falhar são as seguintes ([Ganssle96a], [Ganssle96b]):

- Rotina de Serviço à Interrupção (RSI) demasiado lenta.
- Pedido de interrupção não atendido.
- Problemas de reentrância.
- Esgotamento da pilha.

As RSI's precisam ser executadas dentro de um tempo limite máximo. Uma RSI demasiado lenta pode ser responsável por interrupções não atendidas ou esgotar a pilha disponível. Por esta razão, o seu tempo de execução deve ser monitorizado pelo projectista.

Um analisador de desempenho é capaz de indicar os tempos de execução máximos, médios e mínimos da rotina. Se essa ferramenta não estiver disponível,

então uma outra forma de o fazer será através da activação de um *bit* de teste pela RSI. O projectista pode então utilizar um osciloscópio para examinar o tempo de execução da RSI. Se este *bit* for activado à entrada e desactivado à saída por todas as RSI's, o projectista pode observar qual a carga de processamento correspondente.

As interrupções não atendidas podem ser detectadas comparando, através de contadores de impulsos, o número de pedidos de interrupção e o número de *Interrupt_Acknowledges*. O número de pedidos deve ser sempre igual ao número de *IACK's*. Se o microcontrolador não disponibilizar um pino para *IACK*, então deve contar-se o número de interrupções atendidas em *software*, através de um contador mantido pela RSI. As interrupções perdidas são muitas vezes causadas por RSI's demasiado demoradas ou código que inibe as interrupções durante muito tempo. Para prevenir estes problemas é conveniente minimizar a utilização de código que iniba as interrupções, e durante o processamento de RSI's permitir interrupções o mais cedo possível.

Quando é utilizada uma linguagem de alto nível para programação de uma RSI deve ter-se em atenção o facto de que a utilização de variáveis globais (ou de algumas funções de biblioteca) tornam a RSI não reentrante. Isto significa que a RSI não preserva de forma completa o contexto de *software* do programa que interrompe (por exemplo, altera uma variável global) e sendo assim não deve, em caso algum, ser interrompida por uma outra instância de si própria.

Os problemas de reentrância podem manifestar-se de forma errática. Deve analisar-se o código produzido pelo compilador para procurar chamadas a funções de biblioteca inesperadas (cálculos em vírgula flutuante e outras operações normais em linguagens de alto nível podem produzir acessos a funções de biblioteca).

Os problemas de reentrância podem também surgir se for utilizada uma interrupção NMI (*Non-Maskable Interrupt*) para desempenhar funções normais. Como esta interrupção não é mascarável, vai interromper qualquer secção de código que esteja a ser executada, mesmo que as interrupções estejam inibidas. A NMI deve ser utilizada exclusivamente para tratar eventos catastróficos, como por exemplo falha de energia, falha de *hardware*, etc.

Os problemas com interrupções originam muitas vezes o esgotamento da pilha. Para lidar com estes problemas, pode monitorizar-se a pilha frequentemente, para verificar se um determinado limite pré-estabelecido não é ultrapassado. Este limite deve ser muito maior (duas ou três vezes, por exemplo) do que o tamanho máximo

previsto, para que, quando este limite for ultrapassado, se possa afirmar com alguma segurança que ocorreu um erro.

3.6.3.3 - Verificação e Ensaio de Software Baseado num Núcleo

O ensaio de *software* baseado num núcleo levanta algumas dificuldades. O núcleo mantém armazenadas internamente várias estruturas de dados que reflectem o estado do sistema. O acesso a estas estruturas pode ser difícil porque o projectista pode não saber onde as mesmas estão armazenadas ou qual o seu formato interno.

Para contornar estas dificuldades, alguns núcleos são fornecidos conjuntamente com ferramentas para auxílio ao *debugging* que permitem a consulta do estado interno do núcleo. Estas ferramentas podem ou não ser utilizadas em conjunto com um *debugger* cruzado no sistema alvo.

Um exemplo deste género de ferramentas é o *Kwiklook Manager* para o núcleo AMX. Esta ferramenta pode funcionar de duas formas distintas. Pode ser utilizada conjuntamente com um *debugger* cruzado, ou então independentemente de um *debugger*. Neste último caso, é instalado no sistema alvo código que monitoriza o estado do núcleo e que devolve informação a pedido de uma tarefa. Estes dados são recebidos e apresentados no computador hospedeiro pelo *Kwiklook Manager* (ver Figura 3. 9).

3.6.4 - Verificação e Ensaio de *Software* e *Hardware* Integrados

Uma vez verificados o *software* e o *hardware* individualmente, chega o momento de testar o sistema como um todo. Convém que o objecto do ensaio seja o mais próximo possível do sistema final, para que a verificação seja o mais válida possível. Podem ser utilizados ICE's ou analisadores lógicos neste teste.

A análise do sistema pode ser feita através de um traço (*trace*) da execução do programa. O traço da execução do programa é um registo da actividade do processador num determinado intervalo de tempo. Através da análise deste traço é possível detectar o ponto onde a execução falha.

Por exemplo, o emulador do μC8051 da Nohau Corp. permite armazenar um traço com 16K registos de 48 *bits*. Em cada registo são armazenados: o conteúdo dos *buses* de dados, endereços e controlo, instrução executada, nível das interrupções e conteúdo de dois portos. O programador pode definir condições para as quais o traço

começa ou pára de ser armazenado e pontos de paragem em função dos conteúdos do *bus*. O traço obtido pode depois ser analisado simbolicamente (através do respectivo programa em código fonte) ou através das instruções "de-assembladas".

O traço pode ser utilizado para analisar o estado do sistema quando ocorre alguma falha no seu comportamento. Por exemplo, uma falha de um sistema é frequentemente provocada por algum erro de programação que faz com que o fluxo de execução do programa salte para uma zona fora do espaço de endereçamento onde se encontra o código do programa. Este tipo de problemas pode ser analisado programando o emulador/analisador lógico para fornecer um traço da execução do programa quando é acedida uma instrução fora do espaço de endereçamento previsto.

Alguns emuladores/analisadores guardam, juntamente com o estado do processador/*bus*, o instante em que a leitura foi obtida. Desta forma, é possível medir o tempo de execução de uma instrução ou conjunto de instruções.

Endereço	**Conteúdo Memória**	**Instrução**
00F0	*FA*	*cli*
00F1	*B80F00*	*mov ax,0Fh*
00F3	*BB0E00*	*mov bx,0Eh*
.	.	.
.	.	.
.	.	.
012F	*C3*	*reti*

O projectista indica ao analisador lógico que deve iniciar a contagem do tempo quando o endereço 00F0h estiver presente na linha de endereços, juntamente com o valor B8FAh na linha de dados, dando indicações para a cronometragem parar quando estiver presente o par endereço/valor 012Fh - C3h...

É importante que as interrupções estejam inibidas quando se procede a esta cronometragem, já que se ocorresse uma outra interrupção enquanto o tempo de execução é medido, este apresentaria valores incorrectos. O tempo de execução da RSI pode ser então determinado através da comparação dos instantes em que a leitura foi obtida.

Figura 3. 22 - Exemplo de determinação do tempo de execução de uma RSI.

Capítulo 4

Tópicos de Programação de Sistemas Integrados

4.1 - Introdução

O desenvolvimento de *software* para um sistema integrado envolve alguns cuidados e algumas práticas de programação geralmente não utilizadas no projecto de *software* para computadores com sistemas operativos como o UNIX ou mesmo o MS-DOS. Nestes, a atenção do programador é dedicada quase exclusivamente ao tratamento e manipulação dos dados processados pela aplicação e a garantir que o seu fluxo de execução seja correcto. A interacção com os periféricos é feita quase sempre por intermédio do sistema operativo.

A programação de sistemas integrados obriga ao domínio do *hardware* onde o programa irá funcionar, sendo preciso programar e controlar muitos pormenores das partes muitas vezes designadas como sendo de baixo nível. Isto deve-se à ausência de um sistema operativo ou à grande simplicidade dos núcleos utilizados nos sistemas integrados.

Uma dificuldade adicional resulta da natureza de tempo real de grande parte destes sistemas. Para estes, é necessário que o tempo de resposta se encontre dentro de limites predefinidos, o que raramente é uma preocupação, por exemplo, em aplicações para computadores pessoais [Walls96b].

Devido aos poucos recursos disponíveis nos sistemas integrados e ao controlo de todas as características de baixo nível que é necessário efectuar, é por vezes preciso sacrificar um pouco a facilidade de desenvolvimento, clareza e portabilidade do

programa (obtidos com a utilização de uma linguagem de alto nível), em benefício de uma maior eficiência e flexibilidade do código (obtidos com a utilização de *assembly*). A utilização de *assembly* é praticamente indispensável nas rotinas de arranque do sistema. Por isso expõe-se ao longo deste texto a forma como se podem interligar programas escritos em linguagens diferentes.

Este capítulo tenta esclarecer tópicos de programação específicos dos sistemas integrados, nomeadamente quais os aspectos que devem ser dominados e que não são, na esmagadora maioria dos casos, visíveis na programação e desenvolvimento de aplicações para outro tipo de plataformas. Na secção 4.2 é dado algum ênfase à programação em alto nível, abordando-se questões relativas à interligação de programas desenvolvidos em C com código *assembly*. Na secção 4.3 é abordada a programação de rotinas de inicialização, seguindo-se alguns aspectos do tratamento de Entrada/Saída na secção 4.4. São analisadas, na secção 4.5, algumas optimizações introduzidas pelos compiladores que podem ser inadequadas em aplicações para sistemas integrados, terminando-se na secção 4.6 com a análise de questões relativas à programação de interrupções.

4.2 - Programação de um Sistema Integrado em Alto Nível

4.2.1 - Introdução

Em 1972, Dennis Ritchie cria a linguagem C como suporte para o desenvolvimento do sistema operativo UNIX. O objectivo era a criação de uma linguagem que possuísse as vantagens das de alto nível, mas que fosse praticamente tão flexível e eficiente como o *assembly*. O resultado foi uma das linguagens mais utilizadas no presente, inclusivamente no desenvolvimento de sistemas integrados. A linguagem C deve o seu sucesso a uma grande flexibilidade, simplicidade, eficiência do código gerado e portabilidade para um grande número de plataformas distintas; mantém um certo nível de estruturação do código fonte sem, no entanto, ser tão rígida como o Pascal. É provavelmente a linguagem existente com mais suporte. Ver [MillerQuilici93] para um texto introdutório à linguagem. Existem compiladores de C para virtualmente todos os μPs e μCs existentes no mercado, o que oferece um elevado grau de portabilidade entre diferentes sistemas, tornando-a uma linguagem de primeira escolha para a programação de sistemas integrados.

Outro género de linguagens de programação utilizadas em sistemas integrados são as especialmente dedicadas ao desenvolvimento de sistemas em tempo real, tal como o Ada ou o PEARL. Estas linguagens apresentam mecanismos explícitos para definição de tarefas, sincronização entre estas, controlo da temporização e outras características semelhantes. Para uma comparação entre várias linguagens de programação em alto nível para sistemas em tempo real consultar [Halang90], [Laplante93] ou [Cooling96].

4.2.2 - Interligação de Programas Escritos em Linguagens Diferentes

As linguagens de programação, além de existirem em grande número, apresentam uma grande variedade de características e diferentes vantagens e benefícios. Como as características são assim tão variáveis, pode por vezes ser desejável construir uma determinada parte de um programa numa linguagem diferente da escolhida como linguagem principal. Por exemplo, por razões de facilidade e compactação do código, são muitas vezes utilizadas, num programa escrito numa linguagem de alto nível, rotinas em *assembly* para acesso a *hardware*. Outra situação em que se interligam programas escritos em linguagens diferentes é quando se possui código já desenvolvido numa outra linguagem, o qual se pretende integrar num novo a criar [Mazur92].

Os programas escritos em diferentes linguagens são compilados/assemblados separadamente, obtendo-se um ficheiro de código objecto para cada um, ou então são escritos em ficheiros de código fonte únicos, por exemplo, através da utilização de código *assembly in-line.*

A ligação final entre os vários ficheiros é feita com um *linker,* que os deve juntar num só programa executável. Para tornar esta ligação possível, é necessário ter alguns cuidados. Por exemplo, é preciso utilizar os mesmos nomes para as funções e variáveis comuns e, além disso, declarar as variáveis e funções definidas ou utilizadas noutro programa como externas ou públicas, respectivamente. Por exemplo, a rotina de inicialização de um sistema integrado é normalmente escrita em *assembly* e o programa principal numa linguagem de alto nível, normalmente C.

Todos os exemplos apresentados serão sobre a interligação entre as linguagens C e *assembly* devido à sua quase sistemática utilização na construção de sistemas integrados.

4.2.2.1 - Chamadas de Funções e Passagem de Parâmetros

A ligação entre programas desenvolvidos em linguagens diferentes é feita, por norma, através da utilização de uma função, escrita numa certa linguagem, num outro programa escrito numa linguagem diferente.

Uma das questões essenciais para a interoperação desses módulos distintos é a forma como é feita a chamada, passagem de parâmetros e devolução de valores por uma função. Se um programador conhecer estes detalhes, pode escrever manualmente os parâmetros nas localizações em que a outra função espera que eles sejam fornecidos.

O processo básico de passagem de parâmetros consiste no esquema seguinte: existe um zona de trabalho acessível tanto à função chamada como ao código principal que a invoca. Os parâmetros são escritos pelo código principal nessa zona de trabalho, e em seguida é chamada a função. Quando esta função retorna, o resultado também é devolvido numa zona de trabalho comum (pode ou não ser a mesma que é utilizada para a passagem dos parâmetros).

Conforme o microcontrolador e compilador utilizados, o espaço de trabalho pode consistir em:

- **Uma zona de memória fixa** - Neste caso a função não é reentrante, porque não existe nenhum mecanismo para suportar instâncias múltiplas dos parâmetros da função.
- **Uma pilha** - Enquanto houver memória suficiente, permite a existência de várias instâncias da função. É a única forma de garantir a reentrância da função. É o método mais lento e mais dispendioso em termos de memória.
- **Um conjunto de registos** - É o método mais rápido, mas é limitado relativamente à quantidade de informação que pode transmitir e também não permite a reentrância da função.

Os sistemas integrados são, por norma, limitados em termos de memória de dados disponível. Por esta razão, muitos compiladores para microcontroladores não utilizam uma implementação baseada em pilha. Por exemplo, no caso do microcontrolador 8051 [MacKenzie95], a utilização de uma pilha tem duas desvantagens importantes: se a pilha for mantida em memória interna (128 *bytes* no 8051), poderá facilmente ocorrer uma insuficiência de espaço. Se a pilha for mantida em memória externa, então o acesso à pilha torna-se demasiadamente demorado, com

consequências visíveis a nível do desempenho do sistema. Para exemplos de programação do microcontrolador 8051 consultar [YeralanAhluwalia95].

Mesmo nos sistemas baseados em pilhas, por questões de eficiência, é normal a utilização de registos para devolução de valores e transmissão de alguns dos parâmetros. Neste caso, apenas quando o número de registos disponível já não é suficiente para transmitir todos os dados, é que o sistema recorre à memória de dados (através de uma pilha ou zona de memória reservada) para a passagem dos parâmetros.

Para que o programador consiga chamar uma função definida numa outra linguagem, é necessário que conheça o método de passagem de parâmetros utilizado.

A frequente utilização, em sistemas integrados, de processadores Intel com memória segmentada obriga a cuidados específicos no desenvolvimento do *software*. Nestes processadores, um endereço é composto por um segmento e um deslocamento. Para encontrar um determinado endereço de memória, o valor do segmento utilizado é adicionado a um valor de deslocamento. O resultado dá o endereço físico da célula de memória referida (ver [LiuGibson86] ou [Antonakos96] para informação sobre processadores desta família).

Uma possível fonte de problemas nos μP i80x86, derivada da organização segmentada de memória, é o facto de existirem dois modos diferentes de chamar e retornar de uma subrotina, o modo *near* e o modo *far*. O modo *near* é utilizado para aceder a funções dentro do mesmo segmento, pelo que apenas é guardado e restaurado o deslocamento do endereço de retorno. Já no modo *far*, a subrotina chamada pode encontrar-se num outro segmento, pelo que é necessário guardar o endereço de retorno completo, ou seja, um segmento e um deslocamento. Se uma rotina *far* for chamada como sendo *near* ou vice-versa, nunca retorna ao sítio esperado, continuando a execução em parte incerta da memória (além disso, os próprios parâmetros da função podem ser mal interpretados, se forem transmitidos através da pilha).

4.2.2.1.1 - Chamada de uma Função C por Código Assembly

Se uma função em C não tiver parâmetros de entrada e não devolver nenhum valor, então pode ser chamada em *assembly* apenas invocando o seu nome da função. Em caso contrário, é necessário algum cuidado adicional na passagem de parâmetros através da pilha e na recuperação do valor de retorno.

O exemplo seguinte ilustra como é possível invocar uma rotina escrita em C a partir de um rotina de *assembly*. O exemplo apresentado é relativo a um sistema baseado num processador da família i80x86. No exemplo, os parâmetros são passados pela pilha já que esta é a solução utilizada pela maioria dos compiladores de C para a família de processadores i80x86.

Os parâmetros são colocados na pilha por ordem inversa pela qual estão declarados no cabeçalho da função. Se o tamanho do resultado da função o permitir, é normal este ser devolvido num registo do processador (AX). Nos outros casos, o resultado da função fica armazenado na pilha quando a função termina [Brown94].

A seguinte função escrita em C:

```
public  int soma (int a,b)
        {
        return a+b;
        };
```

Figura 4. 1 - Demonstração de passagem de parâmetros: função "soma".

pode ser traduzida em *assembly* de um μP intel 80x86 (ver [Johnson93] para informação de referência) com o código seguinte:

```
mov bp,sp          ;bp=topo da pilha
mov ax,[bp+2]      ;[bp+2]=primeiro argumento. Note-se que em [bp] está o
                   ;(deslocamento) endereço de retorno.
mov bx,[bp+4]      ;[bp+4]=segundo argumento
add ax,bx          ;Soma primeiro argumento com segundo e devolve resultado em AX
ret 4              ;Retorna da função e retira quatro bytes da pilha (argumentos)
```

Figura 4. 2 - Demonstração de passagem de parâmetros: tradução em assembly da função "soma".

Se a rotina anterior for chamada de um programa em *assembly*, então deve ser declarada antes como externa nesse programa:

```
extrn  _soma
```

b	[bp+4]
a	[bp+2]
endereço de retorno (deslocamento)	[bp]
resultado	ax

Tabela 4. 1 - Localização de parâmetros, endereço de retorno e valor devolvido pela função.

Assim, para calcular, por exemplo, X=soma(X,5), pode usar-se o seguinte fragmento de código:

```
mov ax,5        ;Note-se que os parâmetros são guardados na pilha pela ordem inversa da
push ax         ;que são declarados
mov ax,[X]
push ax
call _soma      ;"_soma" devolve resultado em AX
mov [X],ax
```

Figura 4. 3 - Chamada da função "_soma" a partir de código em assembly.

4.2.2.1.2 - Chamada de uma Função Assembly por Código C

Para chamar uma rotina de *assembly* num programa escrito em C é necessário declará-la como sendo uma função externa no programa C e pública no programa em *assembly*. Quando for chamada esta rotina, o programa em C procede como normalmente, ou seja, põe os parâmetros na pilha e chama em seguida a função. Como a maior parte das rotinas em *assembly* utiliza um esquema de passagem de parâmetros através de registos, pode ser necessária a utilização de um pequeno interface em código *assembly* para carregar os registos com os valores transmitidos através da pilha [Brown94].

A sintaxe para incluir código *assembly* num programa em C varia entre compiladores diferentes. A sintaxe aqui representada é semelhante à adoptada nos compiladores de C da Microtec Research [BigazziWeinberg91]:

asm([t_saida,] "instrução" [,"instrução",...]);

Os argumentos entre [ ] são opcionais. Esta pseudo-função devolve um valor de tipo *t_saida* (opcional). Seguem-se uma ou mais instruções *assembly* (entre " "). Podem referir-se variáveis e funções do programa em C respeitando a seguinte sintaxe:

	Identificador em C	Identificador em *Assembly*
Variáveis globais	buffer	_buffer
Variáveis locais	x	'x'
Funções	main	_main

Tabela 4. 2 - Referências a identificadores de C dentro da função "asm".

O exemplo seguinte ilustra como é possível utilizar rotinas de *assembly* num programa em C. O exemplo apresentado é relativo a um sistema baseado num processador da família i80x86.

```
public soma
soma proc far  ; O procedimento soma é declarado como far, já que o segmento de código do
               ;programa em C pode estar localizado num outro segmento.
add bx,cx      ;Soma dois números (contidos em BX e CX) e devolve o resultado em AX
mov ax,bx
retf           ;Retorna de um procedimento far
endp
```

Figura 4. 4 - Procedimento em assembly "soma".

Esta função pode ser utilizada num programa em C tal como se demonstra em seguida:

```
extern void far _soma( )      /* Declaração da função escrita em assembly como função
                              externa*/

/*A função soma_c serve de interface entre o programa em C e o código em assembly. */
int soma_c(int a,b);
{
asm    (
```

```
        "mov bp,sp",        /*Copiam-se os parâmetros da pilha para os registos */
        "mov bx, [bp+4]",   /*utilizados na função em assembly */
        "mov cx, [bp+6]"
        );
_soma( )                    /*Chamada da função */
asm     (
        mov bp,sp",         /*Copia-se o resultado da função para o espaço antes */
        mov [bp+4],ax"      /* ocupado pelo parâmetro a. */
        );
return a;                   /* Devolve resultado da função*/
}
```

Figura 4. 5 - Chamada do procedimento "_soma" a partir de um programa em C.

a	[bp+6]
b	[bp+4]
endereço de retorno (segmento)	[bp+2]
endereço de retorno (deslocamento)	[bp]

Tabela 4. 3 - Localização de parâmetros e endereço de retorno na pilha.

Uma outra forma de resolver o problema será a escrita do código de interface na rotina de *assembly*, em vez de o fazer com recurso a código *in-line* no código fonte em C. Tal pode ser vantajoso no que respeita à portabilidade do código de alto nível. Neste caso, o código de alto nível é idêntico para as várias plataformas (contendo apenas a declaração da função externa e a sua chamada). A parte dependente da plataforma (neste caso o método de passagem de parâmetros) é tratada pelo código em *assembly*. Apenas este terá pois de ser específico para cada plataforma.

4.3 - *Software* de Inicialização

4.3.1 - Introdução

O programador de um sistema integrado tem a responsabilidade de incluir o código de inicialização do sistema no seu programa. O programador é incumbido desta tarefa

devido às características específicas do *hardware* e mesmo *software* que compõem um sistema integrado, não sendo normalmente possível utilizar uma solução "*standardizada*" sem alterações. O código de inicialização é muito variável entre dois sistemas diferentes e implica geralmente a utilização de *assembly* já que é necessário aceder e inicializar periféricos e estruturas de controlo de baixo nível.

O código de inicialização deve ser considerado como sendo um módulo separado, independente do programa principal, que lhe é ligado apenas aquando da *linkagem*. Existirá então um módulo de inicialização diferente para cada tipo de sistema em que o programa irá funcionar. Quase todos os compiladores são fornecidos com o código fonte da rotina de inicialização que deve ser agregada ao código gerado. Este ficheiro pode ser alterado e recompilado pelo programador se este necessitar de uma rotina de inicialização diferente.

O conjunto mínimo de funções que uma rotina de inicialização de um sistema integrado deve desempenhar são:

- Inicialização da pilha (e outros registos de segmento num μP 80x86).
- Inicialização de constantes e variáveis globais.
- Chamar rotina principal (*main*).

Estas operações são normalmente predefinidas no ficheiro de inicialização agregado pelo compilador. Este conjunto mínimo de funções pode ser complementado, por exemplo, com as operações seguintes:

- Teste da RAM.
- Teste da ROM.
- Teste de periféricos.

Estes testes não são normalmente incluídos no código de inicialização fornecido pelo compilador. Ao contrário das primeiras três operações, que são normalmente feitas em *assembly*, estas já podem fazer parte do programa de alto nível ([Brown94], [Walls96c], [Warntjes96]). É também da responsabilidade do programador localizar a rotina de inicialização de tal forma que esta seja executada no arranque do sistema.

As rotinas de inicialização de um compilador podem também ter código específico para o sistema alvo a que se destinam. Por exemplo, o compilador IC86 da Intel Corporation é fornecido conjuntamente com um núcleo de tempo real (RMX), e

o seu código de inicialização contém rotinas para tratamento das estruturas de dados deste núcleo. Por outro lado, o compilador Borland C++ 3.1 da Borland International Incorporation é destinado principalmente a gerar aplicações para DOS e Windows, pelo que a rotina de inicialização que inclui por defeito tem rotinas para interacção com estes sistemas operativos.

4.3.2 - Características da Rotina de Inicialização

Um aspecto da rotina de inicialização que o programador necessita de verificar é o que se relaciona com a sequência de arranque do microprocessador utilizado. De facto, a utilização dessa rotina deve ser tal que seja ela efectivamente o primeiro código a ser executado. Todas as famílias de μP's possuem alguma forma de determinar qual a primeira instrução a ser executada quando a alimentação é ligada. Dependendo da família utilizada, o μP tanto pode iniciar a execução num endereço predefinido como ir buscar o endereço da primeira instrução a uma zona de memória predefinida. Os μPs i80x86 utilizam a primeira abordagem, enquanto os μPs Motorola 68xxx utilizam a segunda.

Por exemplo, nos μP's intel 80186, a primeira instrução executada é a que se encontra na posição absoluta de memória FFFF0h. Já que apenas restam 16 *bytes* para o topo do espaço de endereçamento, não é possível encaixar aí a rotina de inicialização, pelo que esta instrução é quase sempre um salto para outra posição de memória onde se localiza o efectivo início dessa rotina. Nos μPs da Motorola [Clement92], o endereço da primeira instrução (e o valor inicial do registo SSP) encontra-se armazenado na posição de memória 00000h.

Uma das operações que pode ser efectuada durante a inicialização é o teste da RAM. Este teste implica normalmente a alteração de todo o conteúdo da RAM, para verificação posterior da integridade dos dados armazenados ([Laplante93], [Ganssle95d]). Isto significa que a rotina de inicialização não pode utilizar variáveis (todos os valores necessários devem ser mantidos em registos) nem chamar subrotinas (já que a pilha também reside em RAM).

Os métodos normais de teste de RAM não são aconselháveis em RAMs não voláteis. Isto deve-se à possibilidade de ocorrência de corrupção de uma célula de memória em teste se o sistema for desligado antes de o seu conteúdo original ser restaurado. Por essa razão, são aconselháveis para este caso métodos semelhantes aos utilizados para teste de ROM. O teste de ROM é feito com recurso a técnicas normais

de detecção de erros, como, por exemplo, paridades, CRCs e *checksums*. São apresentados, no Apêndice A, vários pormenores relativos ao teste de RAM e ROM e mapeamento de memória durante a inicialização.

4.3.3 - Inicialização da Zona de Dados

Uma das funções da rotina de inicialização é o armazanamento na zona de dados dos valores preestabelecidos na codificação do programa. Considere-se o seguinte exemplo de código:

```
int a;
int b=2;
const int c=3;
void  main ( )
        {
        a=1;
        b=0;
        };
```

Figura 4. 6 - Exemplo para ilustração de inicialização de zona de dados.

As variáveis da Figura 4. 6 são localizadas da seguinte forma:

- **Variável 'a' e variável 'b':** têm que ser localizadas em RAM, porque de outra forma o seu valor nunca poderia ser alterado.
- **Constante 'c'**: fica armazenada em ROM.

Esta situação levanta uma ligeira dificuldade: a variável 'b' tem que ter um valor inicial apesar de ser mantida em RAM. A solução para este problema é a seguinte: durante o processo de inicialização do sistema, os valores de inicialização de variáveis têm que ser copiados para RAM de forma a estarem acessíveis ao programa desde o seu início.

Esta inicialização não é necessária em aplicações informáticas normais, onde não se recorre a memória ROM e todos os dados/código são carregados de disco para RAM por um sistema operativo. Esta inicialização é portanto específica de sistemas integrados. O programador deve ter o cuidado de verificar se o código gerado pelo

compilador trata desta inicialização, especialmente se esta ferramenta não for especificamente dedicada a sistemas integrados.

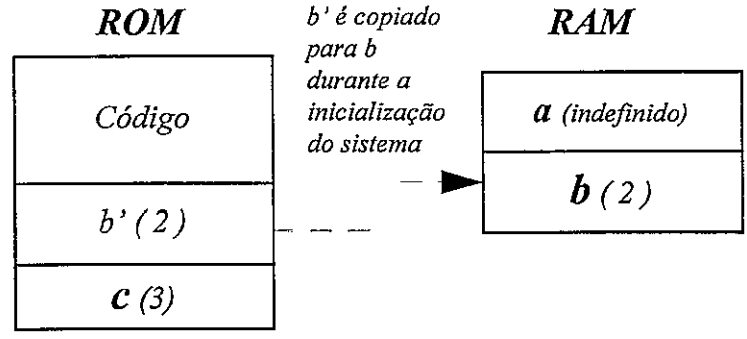

Figura 4. 7 - Inicialização de valores iniciais de variáveis não constantes.

Se o código de inicialização do compilador não efectuar esta cópia, então o programador pode resolver o problema da forma que se segue. Para cada variável inicializada que pretende usar no programa, define uma variável não inicializada e uma variável constante. Só resta copiar o conteúdo da variável constante para a variável não inicializada no início do programa.

```
int matriz [10];
const int matriz_ini [] ={0,1,2,3,4,5,6,7,8,9};
void init_variaveis (int de[], int para[], int numero)
{
for (i=0; i<numero; i++)
        {
        para[i]=de[i];
        };
};
void main( )
{
init_variaveis (matriz_ini, matriz, 10);
}
```

Figura 4. 8 - Exemplo de inicialização de variáveis.

4.3.4 - Teste de Periféricos

O teste de periféricos é geralmente baseado na capacidade de *loop-back* oferecida por alguns dispositivos. Isto permite um esquema de teste análogo ao da memória RAM, em que é emitido pelo dispositivo um valor que é lido por ele próprio. Por exemplo, um controlador de comunicações pode enviar um valor para o seu próprio porto de entrada série. O valor enviado e recebido deve ser o mesmo.

O teste de um periférico pode também incluir a verificação das prestações temporais do mesmo. São utilizadas várias sequências de comandos e dados que pretendem verificar a capacidade de resposta do periférico. Se a resposta não se encontrar dentro das especificações, então pode assumir-se que o periférico não está em bom estado de funcionamento. Pode ser utilizado um *watchdog* para verificação dessas temporizações.

4.3.5 - Memórias Não Voláteis

Uma das operações que o código de inicialização de alguns compiladores (p.ex. Borland C++ 3.1) efectua é colocar a "0" o valor das variáveis. Muitos compiladores para sistemas integrados apresentam um comportamento semelhante. O compilador necessita de inicializar as variáveis em RAM com os seus valores iniciais (armazenados em ROM), "complementando" por vezes este processo com a inicialização a "0" de variáveis sem valor inicial explícito.

Esta operação não é aceitável se o sistema utilizar RAM não volátil porque, nesse caso, os dados aí armazenados seriam perdidos em cada arranque do sistema. Para este problema existem pelo menos duas soluções:

- Edição e alteração do código de arranque para que essa inicialização não seja feita.
- A zona de memória não volátil pode ser acedida indirectamente através de um ponteiro, que é carregado em cada arranque do sistema com o endereço correcto. Como a zona de memória não volátil não faz directamente parte da zona de dados utilizada pelo programa, não é afectada pelo processo de inicialização.

```
char * NVRAM;
NVRAM=(char *) 0x8000;     /* os acessos à memória não volátil devem ser feitos
                           indirectamente através de *NVRAM */
```

Figura 4. 9 - Acesso a memória não volátil através de ponteiro.

4.3.6 - Exemplo de Rotina de Inicialização para i8086

A rotina de inicialização apresentada é constituída por duas partes:

- Uma rotina escrita em *assembly* que se encarrega de inicializar os registos de segmento e chamar o programa principal. Esta rotina pode substituir o código de inicialização incluído por defeito por um compilador. A rotina pode ser obtida eliminando as secções irrelevantes do código de inicialização fornecido pelo compilador.
- Um teste da RAM, o qual é escrito em C não utilizando funções ou variáveis globais, para que o espaço de dados por ele ocupado seja reduzido ao mínimo necessário.

0x00000 0x07FFF	RAM de 32k
0x08000 0xEFFFE	Não utilizado
0xEFFFF 0xFFFFF	ROM de 64k

Tabela 4. 4 - Espaço de memória.

```
_TXT  segment public 'CODE'
      assume cs:_TXT

public _inicializacao
extrn  _main              /*Declaração de funções e dados externos para linkagem*/
```

```
_inicializacao proc far
        cli
        xor ax,ax
        mov ss,ax
        mov ax,8000h
        ;Quando ocorre um push, o SP é decrementado 2 bytes e em seguida o valor do
        ;registo é guardado na posição resultante. Se o SP estivesse a apontar para a última
        ;posição disponível na memória (7FFFh), como seria natural esperar quando
        ;ocorresse o primeiro push, iria sobrar um byte desaproveitado no topo da RAM.
        ;Sendo assim, inicializa-se o SP um byte acima para que não exista nenhum byte
        ;desaproveitado.
        mov sp,ax
        xor ax,ax
        push ax             ;Os registos de segmento são inicializados para apontarem
        pop ds              ;para a RAM localizada no início do espaço de
        push ax             ;endereçamento.
        pop es
        sti
        call _main          ;Chamada da função principal escrita em C. O nome da
                            ;função em C é precedido por um _
        cli
PARA:   hlt                 ;Suspende funcionamento do processador.
        jmp PARA
_inicializacao endp
_TXT ends

;O segmento de código seguinte deve ser localizado pelo programador no endereço absoluto
;FFFF0h. Neste está contida a primeira instrução executada pelo processador, a qual
;consiste num salto para a rotina de inicialização atrás definida.
_TXT2 segment public 'CODE'
        jmp _inicializacao
_TXT2 ends
        end
```

Figura 4. 10 - Exemplo de rotina de inicialização para o processador 80x86.

```
char far * ponteiro _at_ (0x7FFC);
void main ( )
{
for(ponteiro=(char far *) 0 ;ponteiro < (char far *) 0x7FFC; ponteiro++)
        {
        * ponteiro = 0x55;
        };
for(ponteiro=(char far *) 0 ;ponteiro < (char far *) 0x7FFC; ponteiro++)
        {
        if (* ponteiro != 0x55)
                {
                while(1);
                };
        };
for(ponteiro=(char far *) 0 ;ponteiro < (char far *) 0x7FFC; ponteiro++)
        {
        * ponteiro = 0xAA;    /*Preenche memória com valor de teste*/
        };
for(ponteiro=(char far *) 0 ;ponteiro < (char far *) 0x7FFC; ponteiro++)
        {
        if (* ponteiro != 0xAA)
                {
                while(1);       /* Se existe algum problema, então suspende a execução
                                entrando neste ciclo infinito. Seria possível definir outro
                                comportamento, como por exemplo enviar uma mensagem de
                                erro ao utilizador*/
                };
        };
/* Resto do programa */
};
```

Figura 4. 11 - Código de teste à RAM em C.

4.4 - Tratamento da Entrada/Saída

O acesso a periféricos pode ser feito de duas formas distintas: através de um espaço de endereçamento próprio e independente da àrea da memória de dados, ou então, através do mapeamento desses periféricos na zona da memória de dados.

No primeiro caso, existem instruções máquina específicas para acesso a esse espaço de endereçamento. Tal é o caso das instruções *in* e *out*, para leitura e escrita num periférico, respectivamente. Existem pinos do μP/μC dedicados a distinguir se o acesso corrente é ao espaço de memória ou ao espaço de entrada/saída.

No segundo caso, o acesso a periféricos é feito através das mesmas instruções utilizadas para aceder a memória. Não existe distinção, sob o ponto de vista do μP/μC, entre um acesso a memória ou um acesso a um periférico. A determinação do objecto endereçado é efectuada por *hardware* adicional, que distingue os acessos conforme o endereço a que se destinam (por exemplo: 0000h-7FFFh (memória), 8000h-FFFFh (periféricos)).

O acesso a periféricos mapeados no espaço de endereçamento da memória é feito endereçando, para leitura ou escrita, as posições de memória onde o periférico é mapeado. Existem nas linguagens de alto nível três mecanismos distintos para conseguir efectuar a localização de variáveis nos endereços correspondentes aos periféricos:

1. Utilização normal de variáveis, localizadas em seguida, pelo *linker*, nas zonas mapeadas.
2. Utilização de ponteiros para acesso directo às zonas mapeadas.
3. Utilização de primitivas do tipo *AT* para localização imediata das variáveis nas zonas mapeadas.

A primeira hipótese é normalmente menos flexível que as outras, porque os *linkers*, por norma, localizam apenas zonas de endereçamento inteiras (ou segmentos, conforme a terminologia), não permitindo a especificação individual do endereço de cada variável [Beach96].

```
/* Módulo 1: declaração dos dados.*/
xdata unsigned char array[30];      /*Array de chars residente na memória externa de
                                    dados. Os acessos a este array vão ser mapeados num
                                    ou mais periféricos.
```

```
Fim Módulo 1*/

/*Módulo 2: contém o código executável */
extern xdata unsigned char array[30] ;
main()
{
...
/*Leitura de valor do periférico para variável i*/
  i = array[0] ;
 ...
}
/*Fim Módulo 2*/
```

Se os dois módulos anteriores forem linkados com:

L51 modulo1.obj, modulo2.obj XDATA (8000H)

então o linker vai considerar que o espaço de dados externo começa no endereço 8000h. O hardware do sistema deve encarregar-se de reconduzir todos os acessos a memória nesta região para o(s) respectivo(s) periférico(s).

Figura 4. 12 - Exemplo de localização de periféricos em memória através do linker, para o compilador C51 para o microcontrolador 8051.

O programador pode desejar que as variáveis sejam armazenadas na zona de memória indicada pela mesma ordem em que são definidas no programa de código fonte. Existem compiladores que fornecem directivas de compilação que garantem este efeito.

```
/*No compilador C51, a directiva ORDER obriga a que a localização relativa dos dados em
memória seja a mesma que a existente no código fonte:*/
#pragma ORDER
unsigned char xdata RTC_secs ;
unsigned char xdata RTC_mins ;
unsigned char xdata RTC_hours ;
main() {  RTC_mins = 1 ; }
```

Este programa é linkado e locatado em seguida com:

L51 main.obj XDATA(0FA00h)

As variáveis serão localizadas da seguinte forma:

FA00 - RTC_secs

FA01 - RTC_mins

FA02 - RTC_hours

Figura 4. 13 - Exemplo de utilização da directiva "ORDER" (C51/8051).

A segunda alternativa para acesso a periféricos mapeados em memória é a utilização de ponteiros. O endereço (ou periférico, ou registo de periférico) para o qual um ponteiro aponta pode variar dinamicamente ao longo do programa.

```
/*Primeiro caso: atribuição do endereço ao ponteiro durante a inicialização*/
xdata char xdata *a_ptr = 0x8000 ;
/*Segundo caso: atribuição do endereço ao ponteriro durante a execução do programa*/
char xdata *xdata_ptr;                  /* Ponteiro para uma posição não definida de
                                           memória externa */
    main()
    {
    xdata_ptr=(char xdata*) 0x8000 ;    /*Aponta para 0x8000 em xdata */
    *xdata_ptr=0xFF;                    /*Escrita em periférico */
    }
```

Figura 4. 14 - Exemplo de determinação de endereços com ponteiros para C51/8051.

Alguns compiladores definem estruturas que permitem aceder directamente à memória como se se tratasse de um *array (*C51 da Hitex Ltd., por exemplo*)*. Por exemplo, a designação XBYTE[0] representaria a posição de memória de dados externa 0, XBYTE[0xFFFF] representaria a posição FFFFh, etc.

O último método de mapeamento consiste na utilização de primitivas de localização na própria definição das variáveis.

```
/* Localização de cada variável individualmente*/
unsigned char xdata RTC_secs _at_ 0xFA00;
unsigned char xdata RTC_mins _at_ 0xFA01;
unsigned char xdata RTC_hours _at_ 0xFA02;
main( )
        {
        RTC_mins = 1 ;
        }
```

Figura 4. 15 - Mapeamento directo das variáveis em memória (C51/8051).

A optimização da utilização de variáveis pelo compilador pode trazer alguns efeitos indesejáveis. Por exemplo, se a variável *in_perif* representar a entrada de um determinado periférico, então a rotina seguinte pode ser "optimizada" de forma incorrecta pelo compilador:

```
char in_perif _at_ 0x8000;
char espera( )
{
char i;
while (i=in_perif);
return i;
};
```

Figura 4. 16 - Exemplo para ilustração de "volatile".

Como o valor de *in_perif* não é alterado explicitamente dentro do programa, então o compilador pode, por questões de eficiência, armazenar o seu valor num registo do μP e utilizar o valor deste registo em referências futuras. Ou seja, neste caso o periférico seria acedido apenas uma vez no início do ciclo, o valor lido seria armazenado num registo do μP e utilizado nas iteracções subsequentes do ciclo. A rotina não funciona como previsto pelo programador. O problema reside no facto de o valor de *in_perif* poder ser alterado por uma entidade externa sem conhecimento do programa.

O problema com a rotina anterior seria solucionado declarando a variável *in_perif* como sendo volátil. Isto é feito através da declaração:

char volatile in_perif _at_ 0x8000;;

Sempre que houver um acesso à variável *in_perif*, o valor desta é lido novamente da sua fonte já que o compilador assume que ela foi alterada desde a última vez que foi usada.

4.5 - Problemas Provocados pelo Compilador

4.5.1 - Introdução

As optimizações introduzidas por compiladores podem não ter em conta características particulares de aplicações para sistemas integrados. Desta forma, são por vezes introduzidas alterações que impossibilitam o funcionamento correcto do programa. A maioria destes problemas tem origem na utilização de periféricos mapeados em memória.

O código de inicialização introduzido pelo compilador também pode ser inapropriado: muitas vezes inclui código desnecessário, dependente de um sistema operativo, que pode mesmo ser inadequado para a aplicação.

4.5.2 - Eliminação de Código Invariante de Ciclos

Esta optimização pode interferir com acessos repetidos a um periférico mapeado em memória. Considere-se o seguinte exemplo:

```
char * perif_1, *perif_2;
perif_1=(char *) 0x1000;      /*perif_1 aponta para periférico mapeado em 1000h*/
perif_2=(char *) 0x2000;      /*perif_2 aponta para periférico mapeado em 2000h*/
for (i=0;i<10;i++)
        {
        * perif_1=0xFF;       /*Envia 10 vezes o valor FFh para o periférico 1 */
        * perif_2=i;
        }
```

Figura 4. 17 - Código para exemplificação de optimização incorrecta de código invariante.

Este trecho de código pode ser "optimizado" da seguinte forma pelo compilador:

```
char * perif_1, *perif_2;

perif_1=(char *) 0x1000;      /*perif_1 aponta para periférico mapeado em 1000h*/
perif_2=(char *) 0x2000;      /*perif_2 aponta para periférico mapeado em 2000h*/
*perif_1=0xFF;                /*Envia apenas 1 vez o valor FFh para o periférico 1 */
for (i=0;i<10;i++)
        {
        *perif_2=i;
        };
```

Figura 4. 18- Optimização incorrecta de código invariante.

4.5.3 - Eliminação de Recarregamento de Registos

Esta optimização pode resultar em código incorrecto se não se tiver o cuidado de declarar como sendo de tipo volátil as variáveis que acedem a periféricos mapeados em memória. Ver secção "4.4 - Tratamento da Entrada/Saída".

4.5.4 - Eliminação de Código Morto

A eliminação de código morto pode provocar a eliminação de código de acesso a periféricos mapeados em memória. Considere-se o seguinte exemplo:

```
...
perif=0;                      /* perif é um periférico mapeado em memória, utilizado */
for (i=0;i<10;i++)            /* apenas para escrita (p.exemplo: um display ou qualquer */
{                             /* actuador) */
perif=i;
};
...
```

Figura 4. 19 - Código para exemplificação de optimização incorrecta de código morto.

Como o compilador detecta apenas escritas na variável "*perif*", pode decidir que esta se trata de uma variável morta, i.e., que não é utilizada. Desta forma, o compilador pode simplesmente ignorar as atribuições feitas a essa variável.

4.5.5 - Alinhamento de Variáveis

Os compiladores para processadores de 16+ *bits* costumam alinhar as variáveis para que os acessos à memória sejam todos em fronteiras de 16+ *bits*. Desta forma evita-se a sobrecarga de processamento adicional que implica um acesso desalinhado à memória. Esta optimização tem duas consequências importantes para sistemas integrados:

- O espaço de dados necessário é maior o que, no caso de um sistema integrado, pode ser incomportável.
- A localização de variáveis em memória pode não corresponder às expectativas do projectista, o que pode originar problemas no acesso a periféricos mapeados em memória.

```
/*Matriz que mapeia 16 periféricos*/
char perifs [16] _at_ (0x8000);
char i;
for (i=0;i<16;i++)
        {
        perifs[i]=i;
        };
/* Este código não está necessariamente errado, mas é preciso ter em atenção que o
compilador pode gerar código que alinhe os elementos da matriz "perifs" nos endereços
8000h, 8002h, 8004h,..., etc, enquanto que o programador pode pensar que está a aceder às
posições 8000h, 8001h, 8002h,...*/
```

Figura 4. 20 - Exemplo de acessos alinhados/desalinhados a memória.

4.5.6 - Inadequação da Rotina de Inicialização do Compilador

O código de inicialização incluído pelos compiladores pode ser inapropriado para sistemas integrados. Esta secção indica quais as operações que devem ser removidas da rotina de inicialização do compilador.

Tal como já foi referido, cada compilador inclui o seu código de inicialização em cada programa. Em compiladores que não sejam especificamente construídos para desenvolvimento de aplicações para sistemas integrados, apenas uma parte diminuta deste código interessa incluir na aplicação, porque grande parte das operações de inicialização são, normalmente, relativas a um sistema operativo. Entre estas podem encontrar-se:

- Inicialização dos parâmetros da função *main* (*argv* e *argc*).
- Salvaguarda de vectores de interrupção (divisão por 0, erro fatal, etc).
- Gestão do término da aplicação (num sistema integrado, a aplicação normalmente nunca termina).
- Outras operações específicas a um sistema operativo concreto ao qual o compilador se destina, como, por exemplo, verificação da versão do s.o. e gestão da memória do sistema.

Estas operações devem ser removidas pelo programador para que o arranque se processe normalmente.

Um exemplo típico de um compilador deste género é o Borland C++ 3.1 da Borland International Inc. Neste compilador, o código de inicialização por ele agregado varia conforme se esteja a compilar uma aplicação para DOS (lib\startup\c0.asm) ou para Windows (lib\startup\c0w.asm). O código de inicialização para DOS será o mais simples e o mais facilmente adaptável a um sistema integrado.

Mesmo que um compilador esteja preparado para sistemas integrados, é normal que a rotina de inicialização contenha código que não interessa. Por exemplo, a rotina de inicialização do compilador IC86 tem código que é necessário apenas para o núcleo RMX, que é fornecido juntamente com esse compilador.

4.6 - Interrupções

4.6.1 - Introdução

As interrupções são profusamente utilizadas em sistemas integrados. O mecanismo de interrupções é necessário para estabelecer uma comunicação eficiente com os periféricos. Quando um periférico tem dados prontos para enviar ao microcontrolador, ou quando um temporizador detecta a passagem de um determinado intervalo de tempo, efectua um pedido de interrupção para assinalar esse facto ao microcontrolador. Durante o tempo em que não está a processar nenhum pedido de interrupção, o microcontrolador pode estar a executar código com prioridade baixa (diz-se por vezes que está a executar em *background*). Se não for necessário executar nenhum trabalho adicional para além do processamento de interrupções, o microcontrolador pode executar em *background* um simples ciclo infinito, enquanto espera por um pedido de interrupção.

A alternativa a interrupções para comunicação com os periféricos é a utilização de um mecanismo de *polling*, em que, para determinar se aqueles têm alguma informação para transmitir ao processador, são consultados sequencialmente de forma cíclica. O *polling* impossibilita a existência de processamento em *background,* e quanto maior o número de periféricos, menor será a frequência com que cada um pode ser amostrado ([Rosenthal95b], [Laplante93]).

4.6.2 - Programação de Interrupções

Uma rotina de serviço à interrupção (RSI) é o código executado em resposta a um pedido de interrupção. Não é possível utilizar uma função escrita em C como RSI, sem ter alguns cuidados [Ganssle94a]:

- Uma função destinada a ser uma RSI não deve devolver valores nem aceitar parâmetros.
- A função deve ser reentrante, para que seja possível interromper a própria RSI [Ganssle93].
- Deve garantir-se que é efectuada a salvaguarda e restauração do contexto no início e no fim da rotina.

O ANSI C, por exemplo, não fornece mecanismos que garantam automaticamente alguns dos pontos anteriores (p.ex: comutação de contexto), pelo

que, por vezes, é necessária a utilização de linguagem *assembly in-line.* No entanto, grande parte dos compiladores utilizados para o desenvolvimento de aplicações para sistemas integrados são fornecidos com extensões à linguagem que permitem definir uma função como sendo uma RSI, garantindo assim a correcção dos parâmetros, a salvaguarda e restauração automática do contexto.

```
timer_int() interrupt
{
/* Código da RSI */
};
```

Figura 4. 21 - Definição de rotina de serviço à interrupção em C através da operação "interrupt".

Contudo, nenhuma extensão da linguagem é capaz de garantir uma característica extremamente importante: a reentrância. A reentrância é a capacidade de uma função ser capaz de funcionar correctamente, mesmo que seja interrompida por uma outra instância dela própria em qualquer ponto [Ganssle92b]. Não é necessário que uma RSI seja reentrante se se garantir que as interrupções estão inibidas durante todo o seu processamento. No entanto, se uma RSI for demasiado demorada e não puder ser interrompida, são maiores as probabilidades de surgir durante esse período um outro pedido de interrupção que será ignorado.

Para diminuir ao mínimo o tempo em que as interrupções estão inibidas, é prática normal utilizar uma RSI não reentrante e não interrompível apenas para fazer as actividades mais urgentes e continuar o processamento no código em *background.* Uma outra solução será desenvolver uma RSI que seja completamente reentrante. Esta reentrância permite atender qualquer outra interrupção antes de terminar a RSI da interrupção corrente. Deve-se ter em atenção que um encadeamento excessivo de interrupções (provocado por frequências de interrupção elevadas e/ou tempos de execução de RSI's elevados) pode originar o *overflow* da pilha.

Para que uma função possa ser reentrante, nunca pode, em ocasião alguma, utilizar variáveis globais. Por vezes, pode-se ter a ideia de que é possível "esquecer" esta regra quando a variável global é acedida apenas uma vez, por exemplo para

incrementar um contador. Tal prática pode originar problemas quando o compilador decompõe uma instrução atómica de C num conjunto de várias instruções *assembly* não atómicas.

```
long int c;
        :
void x ( ) interrupt;     /*RSI que utiliza variável global */
{
        asm("sti");       /*Permissão de interrupções*/
        c++;
};
```

Aparentemente, pode parecer que nenhum problema relacionado com reentrância pode surgir apesar de ser utilizada uma variável global, já que esta é utilizada uma única vez. Desta forma, nunca existiria nenhum problema porque uma outra interrupção poderia apenas surgir antes ou depois de actualizado o contador C, e nunca durante. No entanto, este incremento pode ser traduzido pelo compilador no seguinte conjunto de instruções:

```
sti
mov ax, [c]
inc ax
mov [c], ax          ; Qualquer interrupção que surja durante a execução destas
mov ax, [c+2]        ; instruções pode resultar em corrupção do valor de "c"
adc ax,0
mov [c+2], ax
```

Figura 4. 22 - Problemas de reentrância numa RSI.

Para garantir que a reentrância não é perdida quando uma RSI necessita de utilizar variáveis globais, é necessário inibir a aceitação de outras interrupções enquanto a variável é manipulada.

Quando é utilizada uma primitiva como *interrupt* para construir uma RSI, o compilador normalmente desactiva as interrupções no início da rotina, para prevenir os problemas de reentrância, activando-as outra vez apenas quando esta termina.

Desta forma, pode ser necessária a utilização de uma instrução específica, eventualmente em *assembly in-line*, para permitir as interrupções desde o início da rotina.

A utilização de variáveis globais não é o único motivo para promover a não reentrância. Um outro motivo possível é a utilização de outras funções não reentrantes, em especial as funções de biblioteca. Grande parte das funções de biblioteca oferecidas pelos compiladores são não reentrantes, o que limita a sua utilização em RSI's.

É aconselhável, sempre que se escreve uma RSI em alto nível, analisar o código resultante produzido pelo compilador. Deve-se procurar chamadas a funções de biblioteca, verificar se as interrupções estão permitidas ou inibidas e, por fim, obter uma estimativa do tempo de execução da RSI. Este tempo é um factor importante porque limita a frequência máxima de pedidos de interrupção.

Mesmo que todos os cuidados anteriores tenham sido tidos em consideração, tal não é suficiente para garantir que um pedido de interrupção é de facto atendido. Isso não depende apenas do funcionamento correcto do código da rotina de serviço a interrupção, mas também da detecção do pedido e subsequente identificação e execução da RSI.

O método para identificação da rotina que deve ser executada em resposta a um determinado pedido de interrupção, depende do microcontrolador utilizado. É normal a existência de uma tabela que indexa os endereços das RSI's relativamente ao número da interrupção. Assim, associado a cada interrupção, existe o respectivo vector, o qual representa um deslocamento dentro dessa tabela. Este deslocamento aponta para a posição da tabela que contém o endereço da RSI.

Quer na família intel 80x8x, quer na família Motorola 68xxx, a tabela de interrupções reside sempre no endereço absoluto zero, pelo que, no caso de sistemas integrados, o programador deverá fazer-se auxiliar de ferramentas de *link-and-locate* para garantir que esta tabela esteja na posição correcta de memória. No caso específico dos microprocessadores i80x8x e M68xxx, trata-se de uma tabela que ocupa 1K de memória, permitindo assim o acesso a 256 posições distintas (já que 256 endereços de 32 *bits* ocupam 1K). Normalmente, as aplicações não necessitam de todas as 256 possibilidades, pelo que as posições livres da tabela são preenchidas com o endereço de uma rotina que não faz nada senão retornar da interrupção.

Uma outra forma de determinação da RSI é a utilizada em alguns microprocessadores e microcontroladores mais simples, tal como o 8051 ou Z80. Nestes, o endereço inicial de cada uma das RSI's é fixo e não pode ser determinado pelo utilizador. Para conseguir que seja executada uma determinada função, é necessário existir neste endereço fixo um salto para ela.

```
/* Exemplo para Intel 8086: Definição da Tabela de Vectores de Interrupção.
Protótipos das funções de serviço à interrupção: uma função de interrupção não devolve
nenhum valor nem aceita parâmetros de entrada. Quaisquer dados devem ser passados*/
void rsi_1 (void);      /*através dos registos*/
void rsi_2 (void);

/* Função nula: apenas retorna da interrupção; usada para preencher posições não
utilizadas da tabela */
void rsi_null (void);

typedef struct endereco_funcao
        {
        void (far * funcao_de_servico) ( );    /*Ponteiro far para uma função de serviço à */
        } ENDERECO_RSI;                        /*interrupção */

/* Declaração da tabela de interrupções. Esta tabela deverá ser localizada pelo
programador no endereço absoluto 0x0000 */
ENDERECO_RSI tabela_de_vectores [ ] =
        {
        rsi_null,
        rsi_1,
        rsi_2,
        rsi_null,
        rsi_null,
        } AT(0x0000);
```

Figura 4. 23 - Exemplo de definição da Tabela de Vectores de Interrupção (i8086).

4.6.3 - Salvaguarda/Comutação de Contexto

A forma pela qual o processo de comutação é implementado depende do microcontrolador ou microprocessador. Uma possibilidade é a utilização pela RSI de um conjunto de registos diferente do utilizado pelo programa interrompido, ou então utilizar uma estrutura, como a pilha, para armazenar todos os registos que sejam alterados durante o funcionamento da RSI.

```
/*Exemplo para Intel: código de entrada de uma rotina de interrupção. Esta rotina pode ser
utilizada como esqueleto básico para qualquer rotina de interrupção, se o compilador não
suportar a utilização de primitivas "interrupt".*/
void rsi ( )
        {
        /* a sintaxe para incluir código assembly in-line pode variar de compilador para
        compilador*/
        asm     (
                "sti",                  /*permite interrupções*/
                "push ax",              /*guarda todos os registos utilizados*/
                "push bx",
                "push cx",
                "push dx",
                "push ds",
                "push es",
                "push bp",
                "push si",
                "push di"
                );
        /* se esta função tivesse algum código útil, este seria colocado aqui */
        asm     (
                pop di                  /*restauram-se todos os registos */
                pop si
                pop bp
                pop es
                pop ds
                pop dx
```

```
        pop cx
        pop bx
        pop ax
        iret                    /*retorna da interrupção*/
        );
};
```

Figura 4. 24 - Exemplo para Intel: código de "interface" de uma rotina de interrupção.

A salvaguarda de contexto é mais eficiente se o microcontrolador suportar a utilização de bancos de registos. Neste caso, para comutar de contexto, é necessário apenas mudar à entrada e saída da RSI o banco de registos utilizado. Outra hipótese é recorrer a um banco de registos para armazenar os valores daquele que está correntemente em uso. O recurso a registos traz consideráveis vantagens a nível de rapidez quando comparado com uma implementação com pilha.

Praticamente todos os compiladores de C para 8051 possuem a primitiva "*using*", que indica ao compilador um banco de registos que deve ser utilizado para armazenar o contexto durante a execução da RSI.

```
timer0_int() interrupt 1 USING 1    /*Definição de uma RSI para o Timer 0 do µC 8051*/
{
/*Código da RSI */
}
```

Figura 4. 25 - RSI para o Timer 0 do 8051, usando o banco de registos 1.

4.7 - Conclusões

As programação de sistemas integrados não é muito diferente da programação de aplicações para outros sistemas. Envolve, no entanto, uma série de cuidados que se não forem tomados podem impossibilitar de todo o bom termo do projecto. Entre estes encontram-se:

- Programação ou ajuste da rotina de inicialização às características específicas do *hardware* e *software* utilizado.
- Localização correcta de dados e código no espaço de memória, para permitir o acesso a periféricos e gravação do código numa ROM.

- Controlo das optimizações introduzidas pelo compilador, para que o código que interage com os periféricos não seja optimizado de forma incorrecta.

Além destes cuidados, é necessário dominar algumas vertentes de programação pouco utilizadas em outro tipo de aplicações, tal como a interligação de programas escritos em linguagens diferentes e a programação de RSI´s.

Capítulo 5

Núcleos para Sistemas Integrados

5.1 - Introdução

Um núcleo fornece os mecanismos básicos para a operação de um sistema multitarefa, tais como as primitivas de comunicação, sincronização e escalonamento das tarefas ([Moore95a], [Moore95b], [Porat92]).

Este capítulo apresenta os conceitos básicos sobre núcleos específicos para utilização em sistemas integrados. São abordados aspectos relacionados com a programação e projecto de sistemas que recorrem a núcleos, concluindo-se com uma apresentação e análise de características de alguns núcleos comerciais (*RTKernel* da Ontime Informatik Gmbh, *smx* da MicroDigital Inc. e *AMX* da Kadak Products Ltd.), baseada em dados recolhidos junto das respectivas companhias.

Na secção seguinte são apresentados os conceitos elementares relativos aos núcleos e à sua utilização. É apresentada uma descrição sucinta da sua estrutura interna, modo de funcionamento e formato de distribução. São indicadas algumas das funcionalidades típicas dos núcleos, terminando-se com uma descrição breve de questões relacionadas com sistemas de tempo real.

Na secção 5.3 são discutidos aspectos práticos da utilização de núcleos e a sua programação, incluindo o projecto de *software* para tempo real, utilização correcta das funções que disponibilizam, entre outros assuntos de carácter aplicacional.

Por fim, na secção 5.4, são analisados os três núcleos comerciais acima referidos, sendo feita uma tentativa de comparação quantitativa que se pensa poder permitir basear uma eventual escolha em critérios fundamentados.

5.2 - Conceitos sobre Núcleos de Tempo Real

5.2.1 - Introdução

Num sistema monoprocessador a concorrência é sempre simulada. Se bem que podem existir várias tarefas em processamento simultâneo, em cada instante apenas uma está em execução. O paralelismo é simulado através da partilha do tempo de processamento do CPU. A atribuição de tempo de CPU pelas tarefas, ou seja, o seu escalonamento, ou é controlada por *software* específica e cuidadosamente elaborado para o efeito ou então é decidida por um núcleo. Em qualquer dos casos o escalonamento baseia-se em determinados critérios de prioridade, habitualmente definidos de forma fixa pelo programador.

Os núcleos são vocacionados para sistemas integrados. Quando comparados com sistemas operativos "normais", pode-se afirmar que são de dimensão e funcionalidade muito reduzidas (ver [Kauler95] ou [SainGonsalves97] para exemplos de núcleos para sistemas integrados). Um núcleo está preparado para fornecer os serviços de gestão de tarefas consumindo um mínimo de recursos. Há funções típicas de um sistema operativo convencional que não são desempenhadas por ele, como por exemplo a gestão de ficheiros e o interface com o utilizador ([BakerScallon86], [Ready86], [Bradley94]).

5.2.2 - Requisitos de um Núcleo de Tempo Real

Para que um núcleo possa ser utilizado num sistema integrado, deve ser capaz de funcionar com recursos escassos de *hardware*. Como normalmente não é utilizada memória de massa em sistemas integrados, um núcleo tem que estar preparado para funcionar juntamente com a aplicação a partir de uma ROM (deve ser *ROMable*).

Embora alguns núcleos possam gerar aplicações que correm sobre um sistema operativo hospedeiro (p.ex: MS-DOS), todos os núcleos têm a capacidade de gerar aplicações que não dependem de nenhum sistema operativo, pois só assim é que estas podem ser *ROMable* e "auto-suficientes" quando introduzidas no sistema alvo.

Grande parte dos sistemas desenvolvidos com o suporte de um núcleo são sistemas de tempo real [Walls96b]. Um sistema em tempo real tem que reagir a estímulos externos dentro de limites temporais prédefinidos. Se uma tarefa está associada ao tratamento desse acontecimento, então o núcleo deve fornecer mecanismos que possibilitem a interrupção de qualquer actividade de importância

menor que esteja em curso para dar início à execução daquela tarefa. Para que tal seja possível, a esmagadora maioria dos núcleos para sistemas integrados oferece um esquema de escalonamento baseado em prioridades. Este esquema é habitualmente preemptivo, ou seja, a execução de uma tarefa é interrompida no momento em que uma tarefa de prioridade superior está pronta para entrar em execução.

5.2.3 - Vantagens na Utilização de um Núcleo

A utilização de núcleos no projecto de sistemas integrados oferece vantagens tais como:

- Parte do trabalho da aplicação é feito pelo núcleo. Além de reduzir o tamanho de código fonte da aplicação, o código do núcleo já foi testado e utilizado anteriormente noutras aplicações. É pois mais fiável do que código construído de raiz.
- A compartimentação do funcionamento do sistema em tarefas mais ou menos independentes simplifica o projecto e teste do *software*. Facilita também a manutenção do código fonte da aplicação e torna-o mais legível.
- O núcleo fornece um interface entre o *hardware* e o programa de aplicação, pelo que o programador não necessita de lidar com algumas das particularidades do sistema. Mesmo assim, a abstração do *hardware* não é tão grande como no caso de sistemas operativos convencionais.
- Existem técnicas de projecto de sistemas em tempo real baseadas em sistemas multitarefa. A utilização de um núcleo de tempo real possibilita a utilização destas técnicas, simplificando assim o desenvolvimento desses sistemas.

5.2.4 - Distribuição Comercial de Núcleos

Os núcleos são normalmente fornecidos num de dois formatos distintos, ou em ambos simultaneamente:

- Código fonte.
- Biblioteca de um compilador.

No primeiro formato o programador é responsável pela compilação do código do núcleo, depois de lhe ter introduzido eventuais alterações específicas da aplicação a

que se destina. Este formato de distribuição permite normalmente uma maior portabilidade do núcleo, porque pode ser compilado para várias plataformas distintas.

No segundo, a aplicação é desenvolvida no compilador para o qual é fornecida a biblioteca, sendo o núcleo acedido através das funções aí disponíveis. Em qualquer um dos casos, a aplicação final é obtida através da *linkagem* do código objecto da aplicação com o código objecto do núcleo.

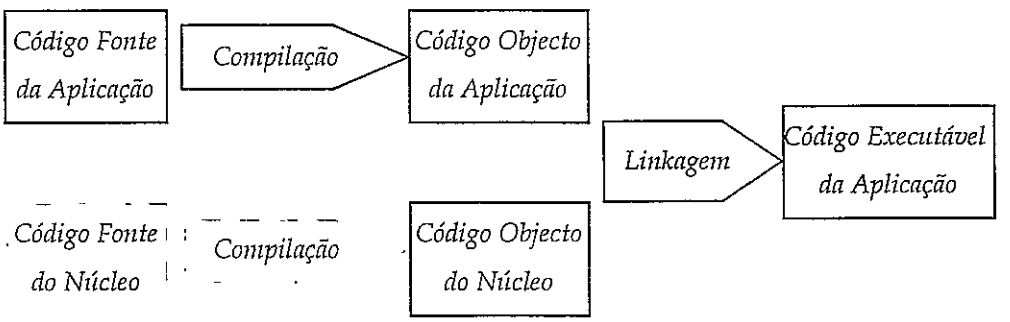

Figura 5. 1 - Desenvolvimento de software com um núcleo.

5.2.5 - Estrutura do Núcleo

Os componentes essenciais que existem em qualquer núcleo são:

- **Escalonador de tarefas** - decide qual a próxima tarefa que deve entrar em execução, conforme a política de escalonamento seguida (ver secção 5.2.8 para uma breve descrição de algumas das políticas de escalonamento mais importantes). O escalonamento pode ser auxiliado por *hardware* específico (ver [ChatterjeeStrosnider96], [Colnaric94] ou [Cooling94] para exemplos).
- ***Dispatcher*** - efectua a comutação de tarefas segundo a indicação de ordem dada pelo escalonador. É normalmente activado periodicamente através de um temporizador, seja por indicação das tarefas, ou através de um pedido feito numa RSI. Esta última característica permite que uma tarefa seja activada a partir de uma RSI para responder a um determinado acontecimento.

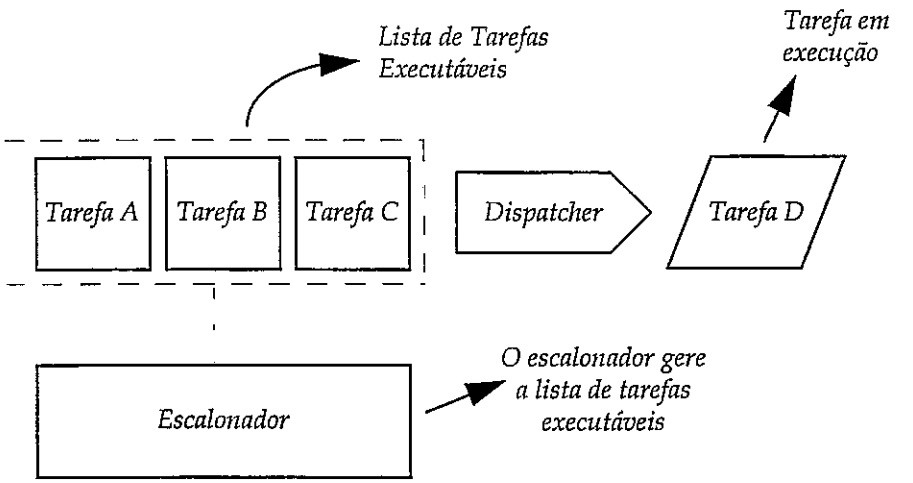

Figura 5. 2 - Estrutura interna básica de um núcleo.

As tarefas e outros objectos do núcleo, tais como semáforos e caixas de mensagens, são representados internamente através de blocos de controlo (*control blocks*). Um bloco de controlo não é mais do que uma estrutura de dados que contém toda a informação relativa a um objecto. O núcleo mantém listas dos blocos de controlo dos vários objectos definidos. Por exemplo, o bloco de controlo de uma tarefa, normalmente designado por TCB *(Task Control Block)*, contém a informação necessária para que o *dispatcher* consiga pô-la em execução quando está à espera de tempo de processamento.

Uma tarefa consiste num conjunto que inclui o seu código e informação sobre o seu contexto de *software* (este último normalmente armazenado no TCB). Em geral, o código da tarefa é escrito como uma função de uma linguagem de alto nível. As variáveis locais dessa função e os seus parâmetros, se existirem, são armazenados na pilha individual de cada tarefa.

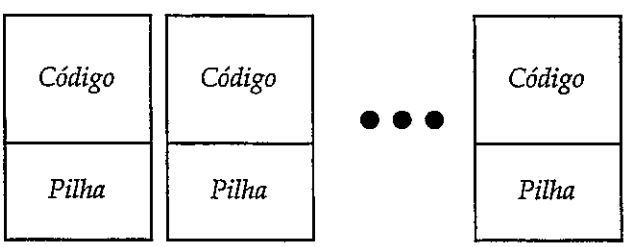

Figura 5. 3 - Várias tarefas com o respectivo código e pilha.

Várias tarefas diferentes podem partilhar o mesmo código. No entanto, sendo distintas, cada uma dispõe do seu TCB e da sua pilha individual.

Como cada pilha obriga a um acréscimo no espaço de RAM ocupado e como esse recurso é por vezes limitado em sistemas integrados, existem alguns núcleos que oferecem a capacidade de alocação dinâmica de pilhas. Neste caso existe um conjunto de pilhas livres, uma das quais será atribuída sempre que é criada uma nova tarefa. Quando uma tarefa termina, a respectiva pilha retorna ao conjunto de pilhas livres, ficando de novo disponível.

5.2.6 - Mecanismos e Funções suportados pelo Núcleo

Esta secção descreve os mecanismos e funções oferecidas pelos núcleos para suporte de um ambiente multitarefa. Entre estas contam-se as seguintes:

- **Sincronização** - Num ambiente multitarefa, tornam-se necessários mecanismos de sicronização, que permitam a operação em concorrência das várias tarefas. Por exemplo, pode ser necessário impedir que duas tarefas distintas tentem comunicar com o mesmo periférico simultaneamente.
- **Comunicação** - O núcleo fornece vários meios distintos para transmissão de informação entre tarefas ou entre periféricos e tarefas.
- **Mecanismos de Temporização** - Os mecanismos de temporização são disponibilizados para permitir a sincronização das tarefas.
- **Gestão de Interrupções** - Num sistema baseado num núcleo, as interrupções são normalmente associadas a um periférico e à tarefa que o gere. A interrupção utiliza mecanismos de comunicação para enviar os dados recebidos do periférico para a tarefa, para posterior processamento. O núcleo fornece os mecanismos necessários para adaptação do funcionamento das interrupções ao ambiente multitarefa.

5.2.6.1 - Mecanismos de Sincronização

Existem dois mecanismos de sincronização que são fornecidos por grande parte dos núcleos. Estes são:

- Semáforos.
- Tabelas de Acontecimentos *(event tables)*.

Um semáforo é um mecanismo usado para impedir o acesso simultâneo à mesma zona de dados ou a um mesmo recurso por duas tarefas distintas. Permite pois a sincronização da execução dessas tarefas. Um semáforo consiste em:

semáforo=contador + lista de tarefas em espera

O contador pode ser visto como o indicador do número de recursos de um determinado tipo que se encontram ainda disponíveis, enquanto a lista indica quais as tarefas que requisitaram esse recurso e que aguardam que o acesso a ele lhes seja autorizado.

Sobre um semáforo podem ser feitas duas operações básicas, "espera" ou "assinala".

A operação "espera" requisita um recurso. Se o recurso não está disponível nesse momento, então a tarefa requisitante vai para a lista de processos à espera, isto é, fica bloqueada. Quando há (pelo menos um) recursos disponíveis, a atribuição é imediata e o contador actualizado.

A operação "assinala" liberta um recurso. O contador correspondente é pois incrementado. O recurso é de imediato atribuído se existir pelo menos uma tarefa na lista de espera.

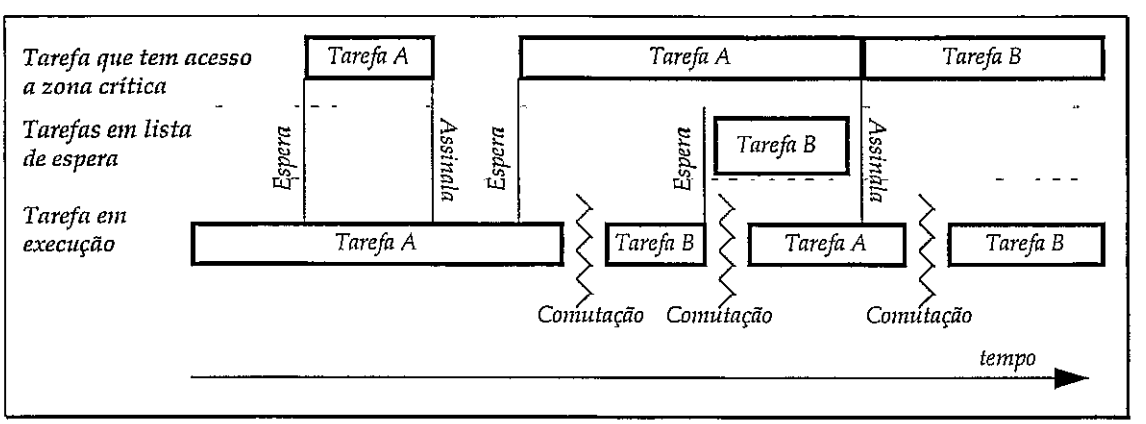

Figura 5. 4 - Exemplo do funcionamento das operações "assinala" e "espera" para duas tarefas.

As tabelas de acontecimentos são utilizadas quando uma tarefa necessita de esperar pela ocorrência não de um acontecimento mas de vários ([Moore95c], [Laplante93]). Uma tabela de acontecimentos é composta por um conjunto de *bits* (*event flags*) que podem ser ligados ou desligados individualmente. Sempre que isto

acontece, o núcleo verifica se existe alguma tarefa para a qual todos os acontecimentos de que está à espera já estão assinalados. Se sim, então a tarefa correspondente é colocada na lista de executáveis.

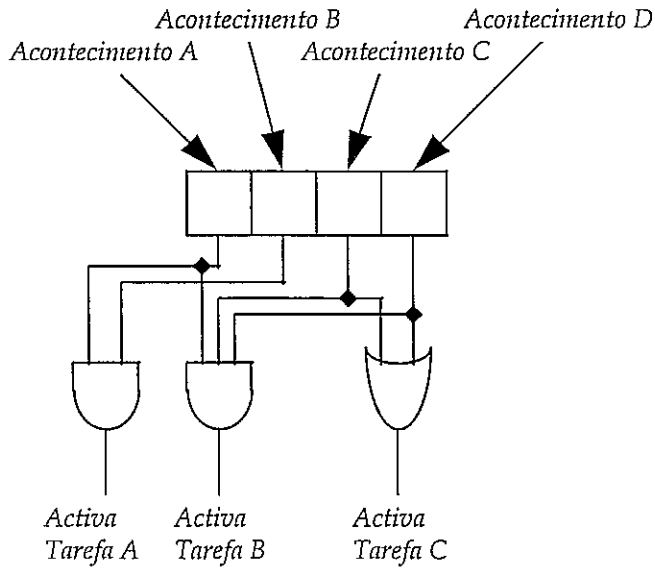

Figura 5. 5 - Operação de uma tabela de acontecimentos.

Como se pode ver na Figura 5. 5[1], as tarefas podem ficar bloqueadas à espera que surja uma determinada conjugação de acontecimentos. O núcleo encarrega-se de tornar a tarefa executável, quando os acontecimentos para ela estipulados ocorrerem.

Este mecanismo de sincronização tem a limitação de possuir apenas um *bit* por acontecimento, pelo que qualquer acontecimento que surja quando o *bit* correspondente não está limpo (i.e. o acontecimento anterior do mesmo tipo não está processado) é ignorado. As tabelas de acontecimentos são apropriadas para lidar com acontecimentos que estão intrinsecamente sincronizados com o código, de tal forma que nunca possa surgir um semelhante antes que o anterior seja processado.

Para lidar com vários acontecimentos simultâneos e assíncronos, existem esquemas semelhantes em que se substitui as *flags* por semáforos.

[1] O teste aos acontecimentos é apresentado na figura através de portas lógicas apenas por uma questão de clareza. O processo é conduzido inteiramente por *software*.

5.2.6.2 - Funções de Comunicação

As facilidades de comunicação oferecidas pelos núcleos são utilizadas para transmitir informação entre as várias tarefas ou então entre uma tarefa e um periférico ([Moore95c] e [Laplante93]).

***Pipes* - Cadeias de Caracteres**

As cadeias de caracteres são utilizadas em transmissões de baixa velocidade, onde os caracteres são provavelmente recebidos e processados um de cada vez.

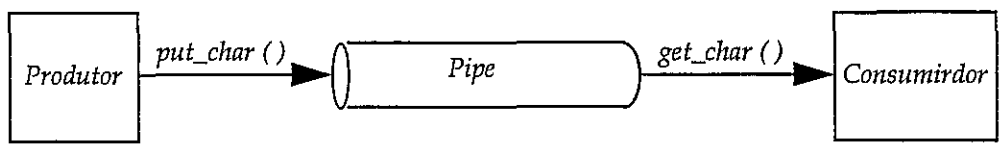

Figura 5. 6 - Pipe.

A *pipe* é um canal que armazena caracteres entre um produtor e um consumidor de dados. Os caracteres são armazenados e recuperados da *pipe* individualmente. Por esta razão, uma *pipe* é apropriada para a transmissão a baixa velocidade de cadeias de caracteres de tamanho indeterminadado (p.ex. dados de um teclado).

Mensagens - Comunicação por blocos de tamanho fixo

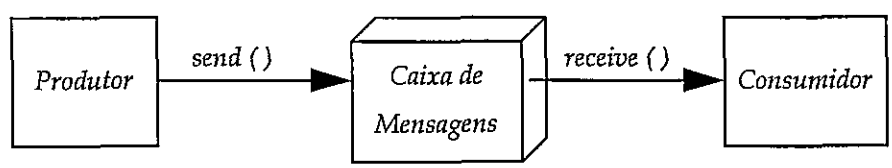

Figura 5. 7- Caixa de mensagens.

As mensagens são utilizadas para comunicação de blocos de caracteres de média velocidade. Neste caso, a caixa de mensagens trabalha não com caracteres individuais, mas sim com blocos de caracteres, normalmente de tamanho fixo. Existem duas formas distintas de caixas de mensagens - a versão "normal" e os *exchanges*. Na primeira versão, a caixa de correio é atribuída de forma fixa a uma tarefa receptora ou par de tarefas. A segunda variante, os *exchanges*, permitem a troca

de mensagens sem que os receptores ou produtores de informação conheçam quem vai ler ou quem produziu a mensagem. Qualquer tarefa pode escrever e receber dados de um *exchange.*

Baldes - Adaptação Bloco/Cadeia de Caracteres

Para adaptar ritmos diferentes de produção/consumo de dados, existem mecanismos para efectuar a transição entre um regime de comunicação por blocos e um regime de comunicação por cadeia de caracteres (e vice-versa). Estes são designados por baldes (*buckets*), e têm o seguinte funcionamento:

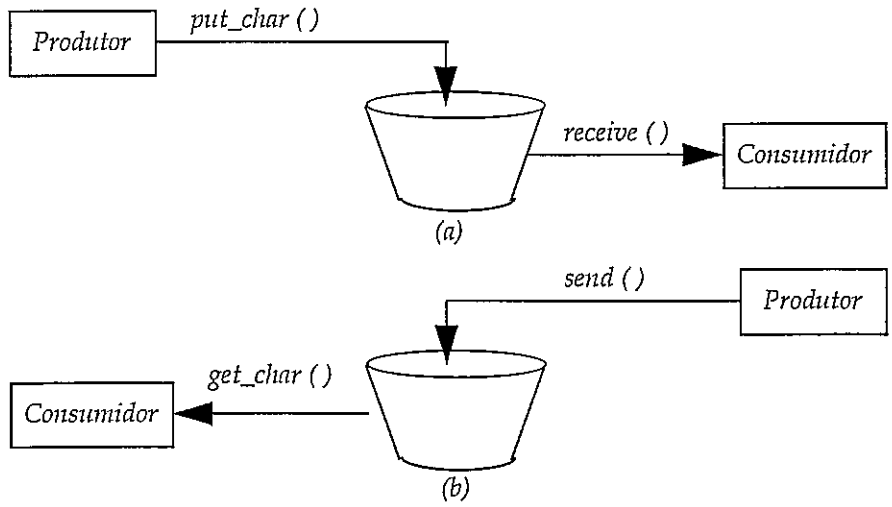

Figura 5. 8 Adaptação cadeia de caracteres-> bloco de caracteres (a) e vice-versa(b).

O produtor de informação pode escrever caracteres num balde, sendo estes lidos pelo consumidor em bloco, através de uma mensagem e *vice-versa.* A possibilidade de adaptação entre ritmos de transmissão desiguais é resultado da diferença entre as funções utilizadas para "encher" o balde e as funções para o "esvaziar".

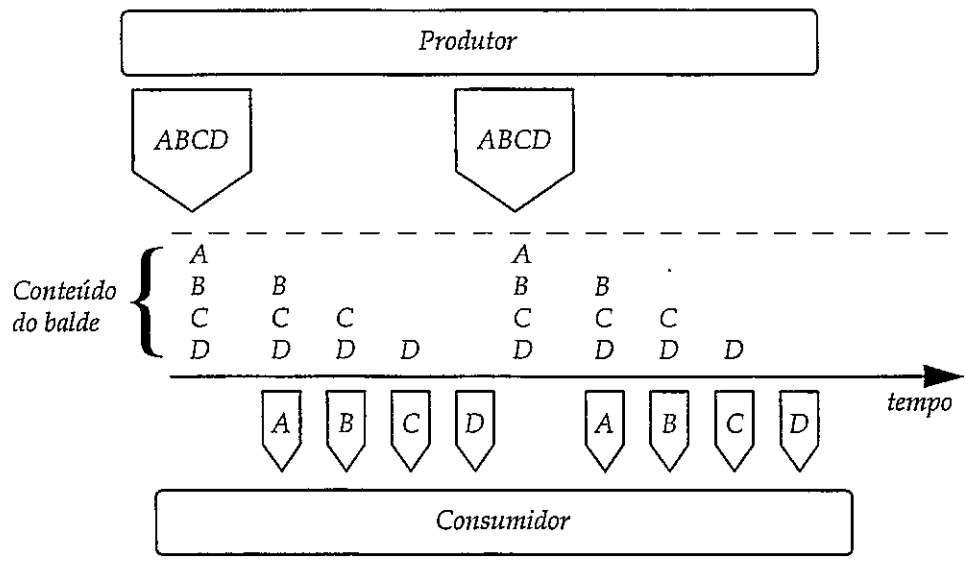

Figura 5. 9 - Adaptação entre ritmos diferentes de produção/consumo de informação através de um balde.

5.2.6.3 - Temporização/Temporizadores

Todos os núcleos apresentam algum processo de contabilização do tempo sob a forma de temporizadores desenvolvidos em *software* a partir de marcas de relógio geradas no *hardware* do processador. Estes temporizadores permitem assim a interligação entre os mecanismos de medição de tempo e os de sincronização entre tarefas. Entre outras aplicações, são também utilizados para gerar *timeouts* para algumas funções do sistema e tempos de espera para uma determinada tarefa.

A utilização de *timeouts* é indispensável para limitar o máximo tempo que uma tarefa fica bloqueada à espera de um recurso. Evita-se assim que uma tarefa fique indefinidamente bloqueada na sequência, por exemplo, de anomalias devidas a erros de sincronização ou deficiente funcionamento de um periférico.

Uma outra aplicação de temporizadores é a suspensão de tarefas durante determinados intervalos de tempo. O núcleo suspende a execução de uma tarefa e torna-a executável quando esse intervalo de tempo tiver decorrido. A suspensão de tarefas durante um determinado intervalo de tempo pode ser útil, por exemplo, para efectuar *polling* a um periférico.

5.2.6.4 - Gestão de Interrupções

Por norma, as RSI's não podem utilizar grande parte das funções do núcleo. Para que tal fosse possível, as chamadas ao sistema teriam que ser completamente reentrantes. Isto deve-se ao facto de existir a possibilidade de ocorrer uma outra interrupção durante a execução de uma função do núcleo chamada por uma RSI. A RSI da nova interrupção poderia por sua vez necessitar de utilizar a mesma chamada ao núcleo, causando problemas de reentrância. Para impedir a ocorrência de problemas deste género, as funções do núcleo teriam que funcionar com as interrupções inibidas durante a sua execução, com a consequente diminuição da reactividade do sistema.

Nos sistemas integrados, a função das RSI's é principalmente comunicar ao código em *background* a ocorrência de um determinado acontecimento. O processamento do acontecimento pode assim continuar em *background* após o término da RSI. Alguns núcleos disponibilizam um tipo especial de tarefa destinado apenas a continuar o processamento de interrupções em *background*, permitindo assim que outras interrupções possam ser servidas. Conforme o núcleo, estas são designadas por *link service routine*, tarefas de serviço à interrupção (TSI) ou outra designação equivalente. Desta forma, existem as seguintes estruturas para tratamento de interrupções:

- Rotina de serviço à interrupção.
- Tarefa de serviço à interrupção.

O tratamento de entrada/saída em sistemas integrados é normalmente baseado nestes mecanismos. São utilizados os meios de transmissão de informação já apresentados para a transmissão de dados entre a RSI e TSI, e entre a TSI e as tarefas normais.

5.2.7 - Mecanismos de Escalonamento oferecidos por Núcleos Comerciais

Os núcleos oferecem vários mecanismos para o escalonamento de tarefas:

- **Preempção** - É a capacidade de um núcleo interromper o processamento de uma tarefa em qualquer instante, mesmo que esta ainda não tenha terminado, para dar início à (ou continuar a) execução de uma tarefa de

prioridade superior, ou seja, mais importante. Se não existir capacidade de preempção, então só existe comutação quando a tarefa em execução termina.

- ***Time-Slicing*** - Alguns núcleos oferecem a possibilidade de dividir o tempo de processamento do μP/μC em intervalos que são alocados ciclicamente pelas tarefas executáveis.

O escalonamento das tarefas em sistemas integrados é normalmente baseado num esquema preemptivo. Este esquema garante que, quando uma tarefa mais importante fica pronta a ser executada, é suspensa a execução de qualquer tarefa menos prioritária para que a primeira possa entrar em execução de imediato. Este tipo de funcionamento está adequado a sistemas que operam com restrições a nível do tempo de resposta, ou seja, sistemas de tempo real.

A alternativa principal a um esquema preemptivo é o *time-slicing*. O escalonamento em *time-slicing* distribui o tempo de processamento entre as várias tarefas prontas a executar. Cada tarefa tem alocado um determinado tempo de processamento, ao fim do qual a tarefa é retirada de execução para dar entrada à seguinte da lista de espera. Se a tarefa retirada de execução ainda não terminou, então é colocada em fila de espera por uma nova fatia de tempo do processador. Este esquema de processamento garante um tempo de processamento equalitário para todas as tarefas, e é relativamente simples e transparente ao utilizador. Tem a desvantagem de considerar todas as tarefas de igual importância, não existindo nenhum mecanismo que assegure um tempo de resposta rápido a um determinado acontecimento. Consultar [Moore91] para uma comparação entre *time-slicing* e prioridades/preempção.

Existem esquemas de escalonamento híbridos que misturam preempção com *time-slicing*. Estes esquemas são baseados em prioridades e preempção, mas para prioridades semelhantes as tarefas executam em *time-slicing*.

O tempo de resposta do sistema depende do seu escalonamento. Um conjunto de tarefas diz-se escalonável se for possível garantir que nenhum *deadline* é ultrapassado quando se utiliza uma determinada política de escalonamento.

A comutação de tarefas pode acontecer numa das seguintes situações:

- Quando uma tarefa em execução desbloqueia uma tarefa de prioridade superior. Tal pode ser resultante de operações tais como: assinalar um

semáforo, enviar uma mensagem, auto-suspensão, ou término da tarefa em execução.

- Quando surge uma interrupção de *hardware*, que torna activa a tarefa a ela correspondente.
- Quando o temporizador do sistema detecta que expirou o tempo de processador atribuído à tarefa em execução.

5.2.8 - Políticas de Escalonamento

A escolha de um dos mecanismos apresentados na secção anterior não é por si só suficiente para determinar o tempo que demora a resposta e processamento de um acontecimento. De facto, torna-se necessário definir ou utilizar a política de escalonamento que melhor se adeque ao problema. Um escalonador que a siga, deverá garantir que o tempo de resposta e o processamento do sistema é menor do que um determinado limite (*deadline*). Esse limite tem de ser observado se se pretende que o sistema apresente características de tempo-real.

Uma política de escalonamento deve atribuir as prioridades de forma lógica a cada tarefa. Esta decisão depende tipicamente de factores como o tempo total de execução de cada tarefa, o seu *deadline*, o tempo que falta para se atingir o *deadline*, o tempo ao qual a tarefa se encontra à espera de execução, entre outros. Algumas das políticas de escalonamento mais conhecidas são:

- ***Shortest-First*** - Esta política dá maior prioridade às tarefas com tempo de execução menor.

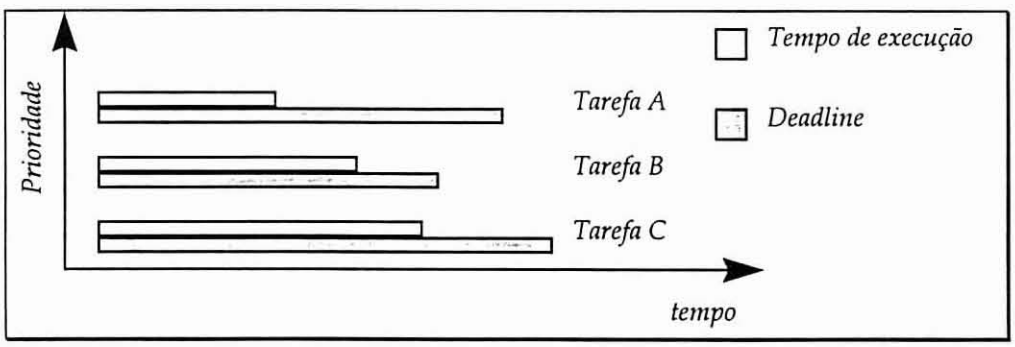

Figura 5. 10 - Exemplo de escalonamento de três tarefas com a política Shortest-First.

- **RMS** (*Rate Monotonic Scheduling*) - Este sistema considera que as tarefas têm um período de activação fixo. As prioridades maiores são atribuídas de forma fixa às tarefas com menor período de activação.

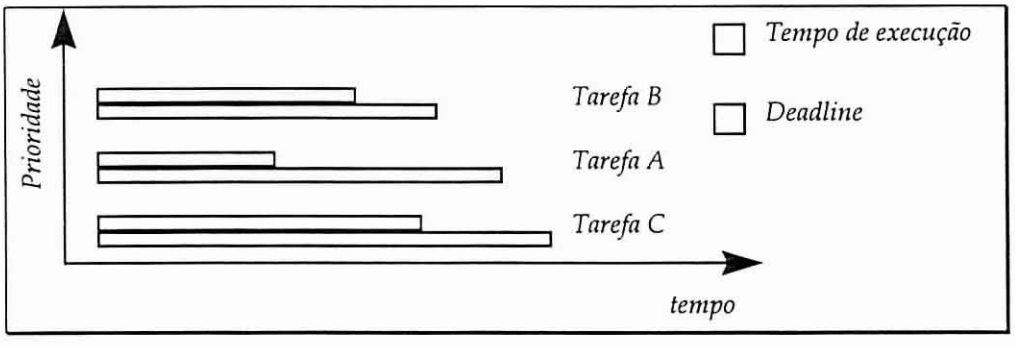

Figura 5. 11 - Exemplo de escalonamento de três tarefas com a política RMS.

- **EDS** (*Earliest Deadline as Soon as possible*) - Este é um sistema onde as prioridades são atribuídas de forma dinâmica, ou seja, variam durante a execução da aplicação. A tarefa com maior prioridade, neste caso, é a tarefa com o *deadline* mais próximo. É distinta do RMS porque o *deadline* de uma tarefa (e consequentemente a sua prioridade) pode variar durante a sua execução.

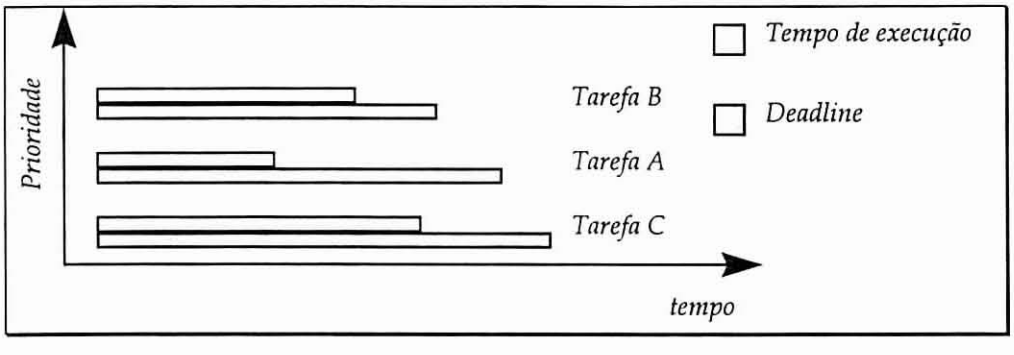

Figura 5. 12- Exemplo de escalonamento de três tarefas com a política EDS.

- **EDL** (*Earliest Deadline as Late as possible*) - Esta política é semelhante à EDS, mas neste caso a prioridade não depende directamente dos *deadlines* mas sim da folga entre o tempo de execução e o seu *deadline*. As tarefas com menor folga são as de maior prioridade.

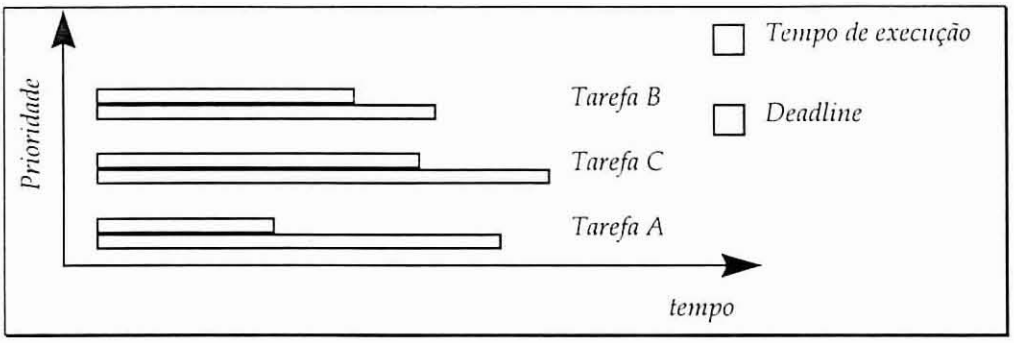

Figura 5. 13- Exemplo de escalonamento de três tarefas com a política EDL.

Para uma descrição mais pormenorizada de várias políticas de escalonamento, consultar, por exemplo, [Korouśic-Seljak94].

5.2.9 - RMS

Este texto vai analisar, como exemplo, a aplicação de uma das políticas de escalonamento mais utilizada, o RMS (ver secção 5.2.8). Esta política apresenta as seguintes vantagens:

- É de implementação relativamente simples.
- Existe um critério que permite indicar se um conjunto de tarefas é escalonável ou não, dados os seus *deadlines* e o seu tempo de execução [LiuLayland91].

As políticas EDS e EDL têm prioridades dinâmicas, o que torna a sua implementação mais complexa relativamente ao RMS. Por outro lado, não existe um critério que garanta a escalonabilidade de algumas políticas mais simples, como o *shortest-first*.

A aplicação da política de escalonamento RMS pressupõe a inexistência de tarefas que sejam activadas aperiodicamente, porque o seu *deadline* é sempre considerado igual ao seu período de ocorrência. O RMS também assume que as tarefas são independentes, no sentido em que não pode existir sincronização entre elas.

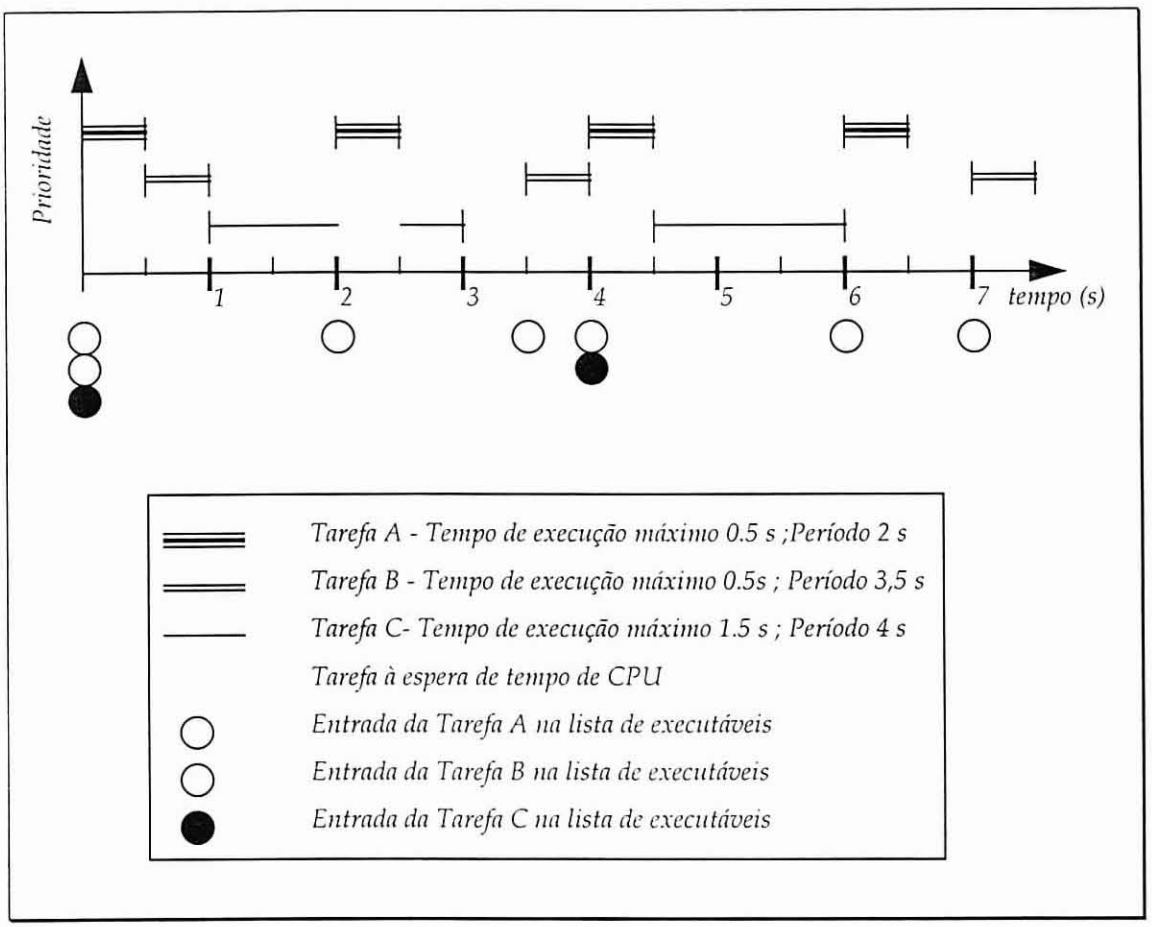

Figura 5. 14 - Exemplo de escalonamento de tarefas segundo a política RMS (com preempção).

As tarefas que são activadas aperiodicamente necessitam de tratamento especial. Uma forma de resolver este problema em RMS é a utilização de uma tarefa periódica com uma frequência harmónica igual ou superior à componente de frequência mais elevada dessa tarefa. Este esquema mantém a garantia de escalonabilidade do conjunto de tarefas.

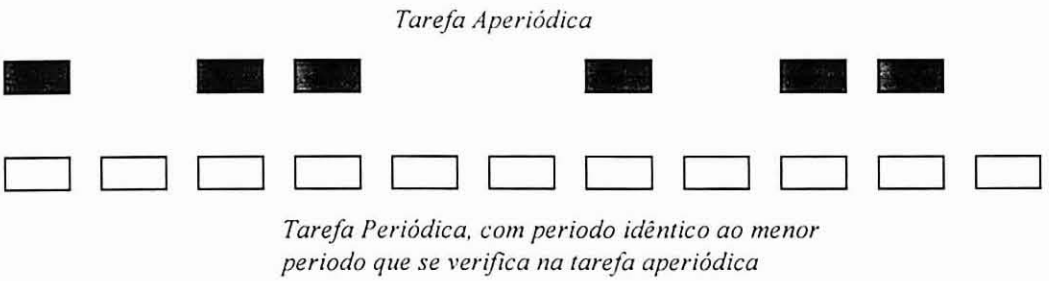

Figura 5. 15 - Adaptação de tarefa aperiódica para tratamento em RMS.

Como a tarefa que processa o acontecimento aperiódico ocorre com uma frequência muito superior ao que será efectivamente necessário para tratar o acontecimento, a maior parte das vezes esta tarefa não executará nenhum trabalho. O tempo correspondente à tarefa pode ser aproveitado para distribuir o processamento de acontecimentos cujo *deadline* não seja crítico. Neste caso, tem que existir uma fila de trabalho para que o processamento seja transportado ao longo de vários períodos de activação da tarefa. Este mecanismo é designado por *delayed server*.

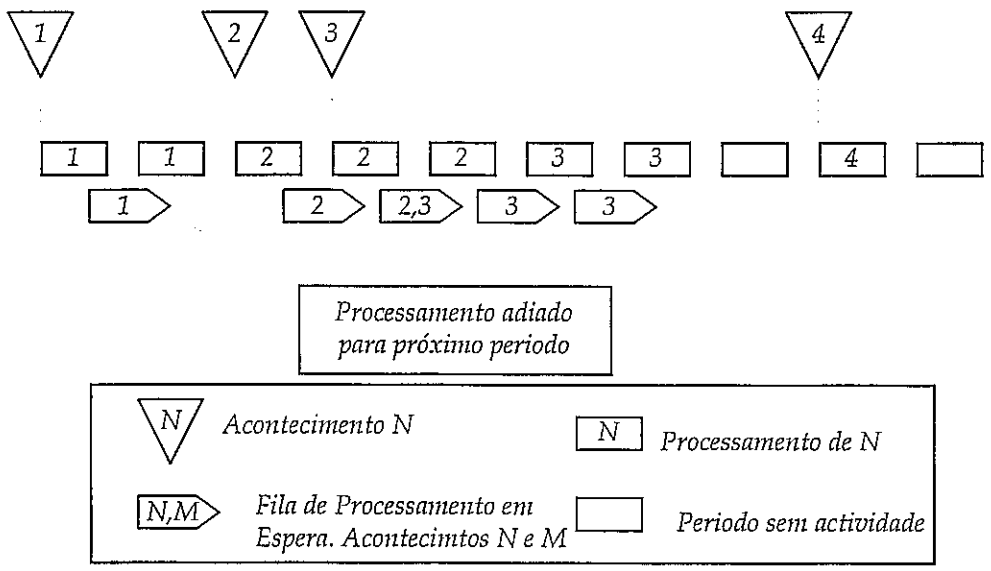

Figura 5. 16 - Processamento de acontecimentos assíncronos em RMS - (Delayed Server).

A limitação do escalonamento RMS relativamente aos acontecimentos aperiódicos significa que geralmente os sistemas baseados neste esquema não utilizam interrupções, ou pelo menos utilizam-nas de forma bastante limitada e controlada. Por causa desta limitação, a interacção com periféricos em RMS é feita tipicamente com o recurso a *pooling*.

5.3 - Perspectivas de Utilização de um Núcleo

5.3.1 - Introdução

A programação de uma aplicação com um núcleo apresenta várias peculiaridades. Por um lado, o projecto do *software* irá ser adaptado ao modelo multitarefa suportado pelo núcleo, e por outro, é necessário dominar e saber utilizar as várias chamadas de sistema. Esta secção apresenta tópicos de programação de uma

aplicação com utilização de um núcleo. Inicialmente é apresentado o projecto de *software* para sistemas em tempo real baseado na política de escalonamento RMS. Em seguida, são discutidos vários aspectos relativos à utilização prática das funções do núcleo, nomeadamente os aspectos de inicialização, programação das tarefas, tratamento de entrada/saída e gestão das interrupções.

5.3.2 - Projecto de um Sistema de Tempo Real usando RMS

O projecto de um sistema de tempo real utilizando um núcleo pode ser feito através da construção de um sistema multitarefa preemptivo com base numa política de escalonamento do tipo das descritas na secção anterior. A utilização da política de escalonamento permite ao projectista a aplicação de determinados critérios de escalonabilidade que garantem que o sistema se comporta dentro das suas especificações temporais.

Tal como já se referiu, a política de escalonamento RMS é uma das mais utilizadas para projecto de sistemas de tempo real. Esta secção apresenta tópicos de projecto de um sistema em tempo real com o auxílio dessa política de escalonamento.

Para saber se um determinado conjunto de tarefas é escalonável utilizando uma política RMS, é necessário conhecer o tempo máximo de execução de cada tarefa, para determinação posterior da carga de processamento máxima [Laplante93]. Se esse tempo for designado por $T_C(i)$ para a tarefa i, então é possível garantir que um conjunto de N tarefas, cada uma com período $T_P(i)$, é escalonável se (ver [LiuLayland91]):

$$\sum_{i=1}^{N} \frac{T_C(i)}{T_P(i)} \leq N\left(2^{\frac{1}{N}} - 1\right)$$

A quantidade $U_N = N\left(2^{\frac{1}{N}} - 1\right)$ indica qual a carga de trabalho máxima para N tarefas e tem como valor assimptótico 0.693 quando N$\rightarrow +\infty$. Isto significa que um sistema com escalonamento RMS nunca falha um *deadline* se a carga do processador não for superior a cerca de 69%, independentemente do número de tarefas no sistema.

O critério de escalonabilidade do RMS utiliza o tempo de execução de cada tarefa. A determinação deste tempo é feita pelo programador do sistema através de ferramentas de *profiling*[2] ou através da análise do código gerado pelo compilador.

A existência do critério de escalonabilidade sugere a decomposição do desenvolvimento de um sistema em tempo real utilizando o escalonamento RMS nas seguintes fases:

- Identificação preliminar das tarefas necessárias à resolução do problema.
- Codificação das tarefas.
- Análise e medição do tempo de execução individual de cada tarefa.
- Verificação da escalonabilidade do conjunto de tarefas resultantes.

Caso o conjunto não seja escalonável, deve-se optimizar as tarefas para reduzir o seu tempo de execução ou, para uma alteração mais radical, reestruturar o seu número (se tal for possível).

Um indicador de quais as tarefas em que a optimização será mais eficaz é o valor de $\frac{T_C}{T_P}$, pois quando elevado, indica as tarefas que ocupam a maior carga de processamento.

Para a política de escalonamento RMS, os serviços do núcleo mais utilizados são o mecanismo de preempção de tarefas e o mecanismo de temporização e *timeout*. As prestações destes dois mecanismos são fundamentais para o funcionamento preciso de um sistema RMS.

Num sistema RMS, o mecanismo de temporização é normalmente usado na suspensão de uma tarefa até ao seu próximo período de activação. Sob esta perspectiva, existem algumas características (prestações) do temporizador que interessam considerar. Os três factores seguintes são importantes para essa análise:

- O tempo de execução da chamada para inicialização do temporizador e suspensão da tarefa.
- A resolução do mecanismo de temporização.
- A precisão do mecanismo de temporização.

[2] As ferramentas de *profiling* são utilizadas para medição de tempos de execução, verificação do número de vezes que é executada uma subrotina, etc.

Se uma tarefa tiver de ser suspensa durante um tempo de espera determinado, há que chamar a correspondente função do núcleo. O tempo de execução dessa função deve ser descontado no tempo de suspensão pretendido. O tempo de suspensão deve ser T_p-T_l, onde T_p é o tempo de espera pretendido e T_l é o tempo de latência da instrução de suspensão. Para que tal seja possível, é evidente que a resolução do mecanismo de temporização deve ser suficiente para representar um valor da ordem de grandeza de T_l. Se tal não for possível, provavelmente o tempo de espera indicado é de tal forma superior a T_l que este pode ser ignorado. A precisão do mecanismo de temporização pode ser dada pelo tempo que decorre entre o tempo de suspensão indicado pela tarefa e o instante ao fim do qual é detectada a passagem deste tempo. A resolução e a precisão do temporizador são normalmente da mesma ordem de grandeza.

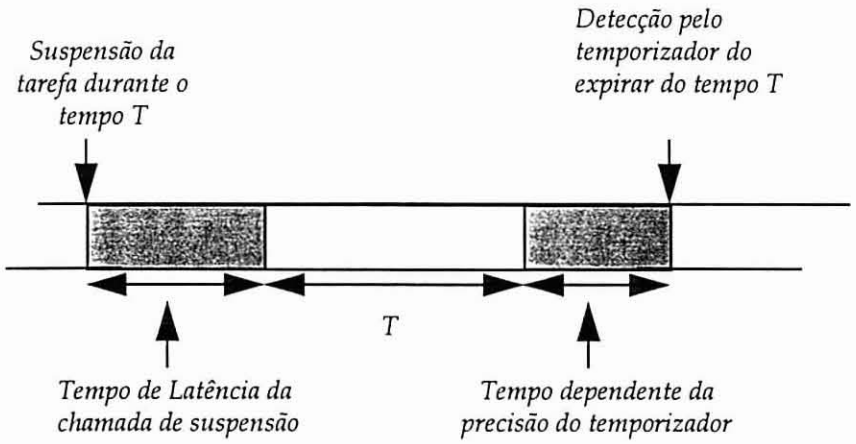

Figura 5. 17 - Temporizações do núcleo aplicadas a um sistema RMS.

O tempo que resulta da (im)precisão do temporizador não pode ser compensado de forma semelhante ao tempo de latência da chamada de suspensão porque, ao contrário deste último, não é um tempo fixo mas varia normalmente entre 0 e P (a precisão), sendo P o valor da frequência de amostragem do temporizador.

O tempo de comutação da tarefa pode ser compensado descontando esse tempo no tempo de suspensão da tarefa. A chamada que suspende a tarefa RMS será então:

$$suspend(ct,\ T_p - T_l - T_{com})$$

Onde T_p é o tempo de espera pretendido, T_l é o tempo de latência da chamada de suspensão e T_{com} é o tempo de comutação. No entanto, apesar de se obter o período desejado através da compensação deste tempo, o tempo de comutação tem influência a

nível da carga total de processamento do sistema. Um sistema com T_{com} aproximadamente igual a zero consegue dedicar quase todo o seu tempo de processamento para as tarefas. Um sistema com T_{com} elevado e comutações significativas vai ver substancialmente reduzido o tempo disponível para a execução do código das tarefas.

Como a condição de escalonabilidade de RMS envolve a carga do processador, a carga imposta pela comutação vai ser um factor que se deve ter em conta quando se pretende verificar o seu cumprimento.

A carga do processador imposta pelo tempo de comutação de tarefas é de T_{com}/T_p, onde T_p é o período da tarefa. Desta forma, para N tarefas, temos a seguinte ocupação do processador relativa à comutação:

$$C_N = T_{COM} \sum_{i=1}^{N} \frac{1}{Tp_i}$$

Juntando esta fórmula ao critério de escalonabilidade para RMS, obtém-se:

$$\sum_{i=1}^{N} \frac{Tc_i + T_{COM}}{Tp_i} < U_N$$

Verifica-se que o tempo de comutação de tarefa deve ser adicionado ao tempo de execução de cada tarefa. Desta forma, torna-se num factor limitativo da escalonabilidade do conjunto de tarefas. Quanto maior for o número de tarefas e quanto menor for o período T_p de cada tarefa, maior a influência do tempo de comutação na carga do processador.

As interrupções devem ser evitadas a todo o custo em sistemas RMS, porque são activadas assincronamente. Sempre que tal for possível, deve-se substituir a RSI por uma tarefa com a frequência adequada (de forma semelhante ao apresentado na Figura 5. 15). Quando tal não é possível, deve-se ter em mente que é necessário contabilizar a carga de processamento que as interrupções irão implicar. Se não for determinável o ritmo de pedidos de interrupção (i.e. não for possível estabelecer uma frequência máxima), então é difícil ou mesmo impossível garantir a escalonabilidade do conjunto de tarefas.

Uma fonte de interrupção que nunca pode ser completamente desactivada é a gerada pelo temporizador. Esta interrupção é responsável pela activação de uma rotina

que, entre outras tarefas, irá verificar o decorrer dos *timeouts*. Para uma precisão de P segundos, esta rotina é activada com uma frequência $f = \frac{1}{P}$Hz. Quanto maior for esta frequência, maior será a carga imposta por esta interrupção ao processamento. Para uma precisão de P segundos, a carga de processamento imposta pelo temporizador será:

$$\frac{T_I + T_T}{P}$$

T_I é o tempo de latência de interrupções e T_T é o tempo de execução da rotina do temporizador. Esta carga de processamento imposta pelo temporizador deve ser adicionada à carga de processamento imposta pelas tarefas e comutações no critério de escalonabilidade RMS, obtendo-se assim:

$$\frac{T_I + T_T}{P} + \sum_{i=1}^{N} \frac{T_C^i + T_{COM}}{T_P^i} < U_N$$

É apresentado nos gráficos seguintes um exemplo da carga de processamento imposta pela comutação de tarefas e temporizador do núcleo. Para desenhar os gráficos, foram utilizadas as prestações temporais do núcleo AMX (386-20Mhz):

- *AMX clock handler* *20.5 μs*
- *Latência de interrupção* *31.9 μs*
- *Espera por intervalo de tempo* *35.7 μs*
- *Comutação de tarefa* *17.6 μs*

Foi considerado um conjunto de 5 tarefas para desenhar o gráfico 5.1. A frequência das várias tarefas é aumentada linearmente, verificando-se um correspondente aumento linear da carga de processamento. Os períodos iniciais das tarefas são de 10, 5, 1, 0.5, 0.01 e 0.01 segundos.

Verifica-se que, nas condições apresentadas, o impacto na carga de processamento da comutação de tarefas é muito superior ao causado pelo *clock handler*, para frequências de operação do temporizador "razoáveis".

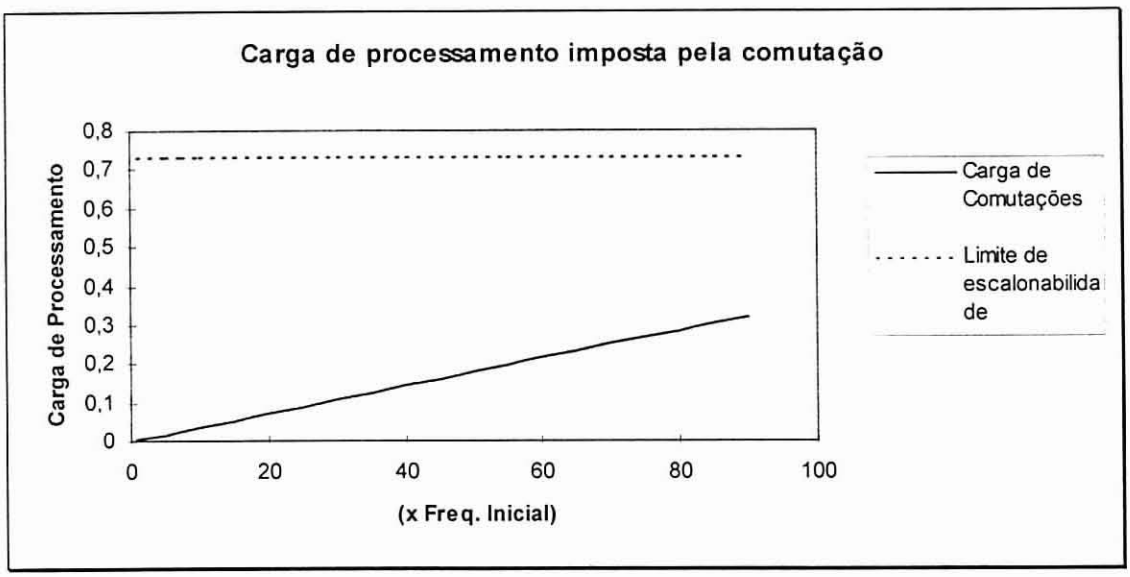

Gráfico 5. 1 - Carga de processamento imposta pela comutação de tarefas.

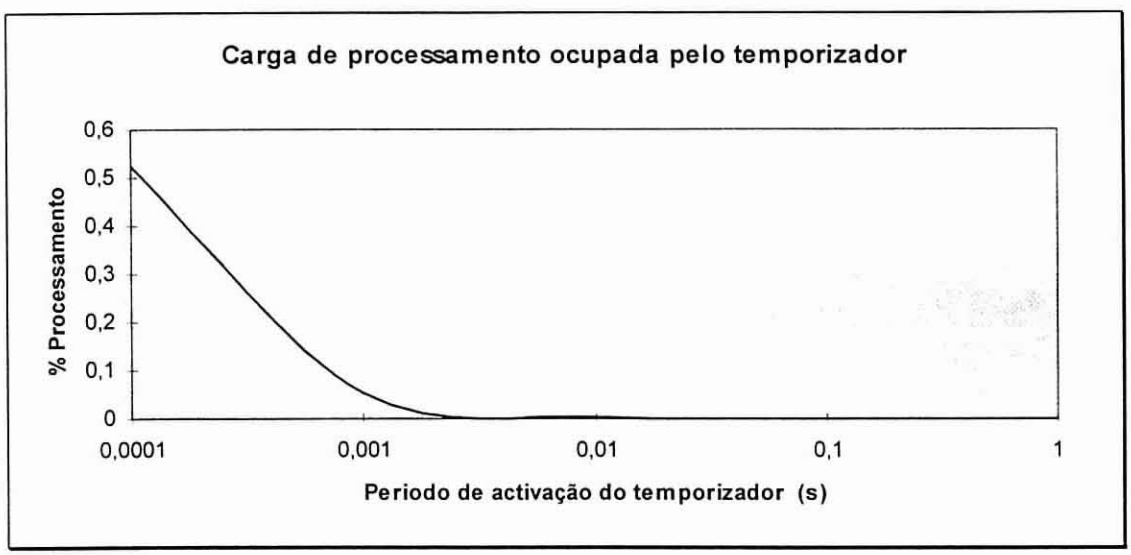

Gráfico 5. 2 - Carga de processamento imposta pelo temporizador do núcleo, para diferentes valores do período de activação deste.

5.3.3 - Tópicos de Programação com um Núcleo

Esta secção apresenta alguns aspectos práticos da programação de núcleos. Encontram-se aqui aspectos relativos à criação de tarefas ou outros objectos do núcleo, pormenores relativos à escrita de tarefas em linguagens de alto nível, programação de interrupções (apesar de não serem aconselháveis em sistemas baseados na política RMS, são profusamente utilizados noutro tipo de sistemas, pelo que são aqui discutidas), tratamento de E/S e temporizadores.

5.3.3.1 - Criação/Inicialização dos Objectos do Núcleo

Qualquer que seja o núcleo utilizado, é necessário definir e inicializar os vários componentes do *software* do sistema. O próprio núcleo necessita de ser inicializado. Para esse efeito, todos os núcleos fornecem uma função que deve ser chamada antes de serem utilizadas quaisquer outras chamadas ao núcleo (por exemplo *go_smx* ou *RTKernelInit* para os núcleos *smx* da MicroDigital Inc. e *RTKernel* da OnTime Computing Gmbh., respectivamente). São também disponibilizadas funções para definição e inicialização das tarefas e outros objectos do núcleo. O programador necessita de definir as tarefas e os outros objectos do núcleo para que este possa criar e inicializar os respectivos blocos de controlo.

Por exemplo, uma função de criação de tarefa tem normalmente como parâmetros, a indicação de qual a função que contém o código da tarefa, qual a sua prioridade e qual o tamanho da pilha que vai utilizar. Estes parâmetros podem variar de núcleo para núcleo.

As funções de criação de objectos do núcleo devolvem normalmente um ponteiro para o respectivo bloco de controlo, denominado vulgarmente por *handle*. Este é utilizado como parâmetro para as funções do núcleo que operam sobre esses objectos.

5.3.3.2 - Escrita de Tarefas em Linguagem de Alto Nível

O código das tarefas é escrito como uma função normal. Geralmente esta é uma função sem parâmetros e sem argumentos, se bem que alguns núcleos permitem a utilização de parâmetros como método simples de passagem de dados. O código de uma tarefa pode chamar outras funções como qualquer função normal.

```
unsigned int tarefa_main (unsigned int parametro)
{
/* Processa parâmetro */
return (resultado);
};

void tarefa_init (void)
        {
```

```
unsigned int resultado;
tarefa=create_task(tarefa_main,HI_PRIORITY,STACKSIZE);
/* A TCB tem um campo (rv) dedicado à passagem de parâmetros de e para a tarefa */
tarefa->rv= 5            /* Dado para processar pela tarefa */
start (tarefa);
resultado=tarefa->rv.  /*Valor de retorno devolvido no mesmo campo da TCB*/
};
```

Figura 5. 18 - Passagem de parâmetros para tarefa através do TCB (núcleo smx*).*

É possível criar mais do que uma tarefa a partir do mesmo código, ou seja, da mesma função. Neste caso, vão ser criados dois ou mais TCB's diferentes, um por cada tarefa, permitindo assim que as várias tarefas tenham contextos diferentes e possam evoluir (ser executadas) de forma distinta apesar de partilharem o mesmo código. É necessário que o código da tarefa seja reentrante, porque o mesmo código vai ser utilizado concorrentemente. A partilha de código por várias tarefas permite não só a poupança de espaço, mas é também vantajosa para a modularidade do sistema. Por exemplo, é mais fácil manter e actualizar um sistema em que se encontram vinte tarefas partilhando o mesmo código a tratar de vinte portos de comunicação idênticos do que um sistema em que uma única tarefa necessariamente com código mais longo e complexo trata de todos os portos. Se se pretender aumentar ou diminuir o número de canais, basta ajustar o número de tarefas em funcionamento. Além disso, a separação do tratamento dos vários portos de comunicação permite um melhor isolamento no caso da ocorrência de erros. Se ocorrer um erro no tratamento de um porto, como o serviço está distribuído por várias tarefas distintas, o funcionamento dos outros portos provavelmente não irá ser afectado, o que não seria o caso se todo o serviço fosse assegurado por uma única tarefa. Neste último caso, um problema num porto que causasse a falha da tarefa iria impedir o funcionamento de todos os canais.

5.3.3.3 - Gestão de Interrupções

Para que as interrupções possam utilizar as funções do núcleo para comunicar com as TSI's e as tarefas em *background*, é necessário garantir que o escalonador é activado sempre que uma RSI termina. Tal permite processar uma eventual comutação

de tarefa necessária devido a chamadas efectuadas ao núcleo dentro da RSI. Para este efeito, alguns núcleos fornecem as primitivas:

- ENTER_ISR()
- EXIT_ISR()

A primitiva ENTER_ISR deve ser utilizada no início da RSI, e incrementa um contador de encadeamento de interrupções. A primitiva EXIT_ISR decrementa o contador de encadeamento e retorna da RSI, activando o escalonador se não existirem mais interrupções encadeadas. Este esquema é destinado a garantir que o escalonador do núcleo é executado imediatamente antes de se retomar o processamento do código em *background* interrompido. Não é necessária a utilização destas primitivas se a RSI não interage com o código em *background.*

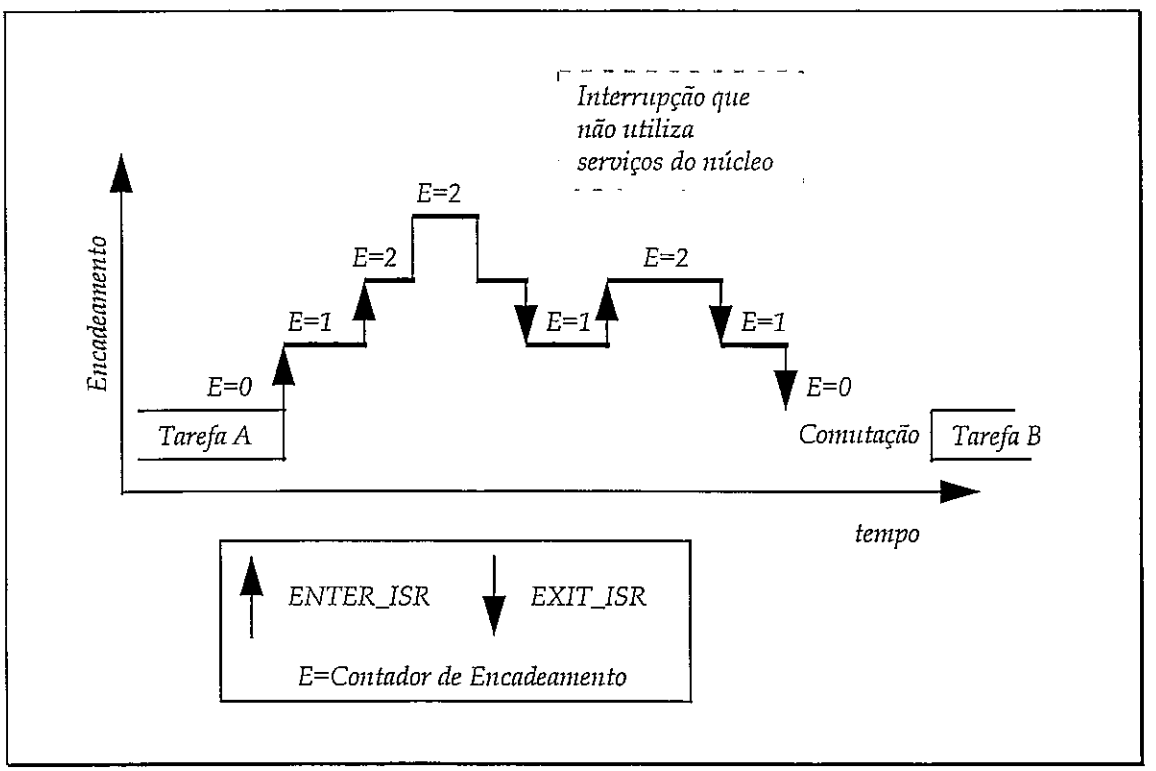

Figura 5. 19 - Exemplificação de encadeamento de interrupções e ENTER_ISR e EXIT_ISR

As tarefas de serviço à interrupção são geralmente executadas a um nível de prioridade superior ao de outras tarefas normais. Isto significa que todas as TSI's terminam antes do processamento retornar às tarefas em *background.* As TSI's podem ser interrompidas por RSI's, que podem por sua vez activar as suas TSI's para que entrem em execução quando terminem. Se existirem várias TSI's prontas a entrar em

execução, a ordem pela qual são executadas depende do núcleo. É normal a utilização de um esquema FIFO (núcleo *smx*) ou então a atribuição a cada TSI de uma prioridade semelhante à da interrupção que a origina.

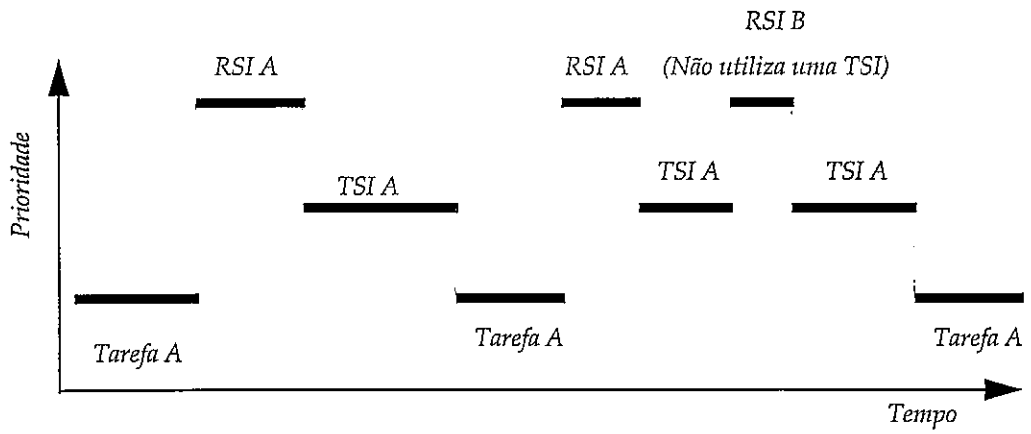

Figura 5. 20 - Encadeamento de Tarefas, TSIs e RSIs.

Na Figura 5. 21 é apresentado um exemplo de definição de TSI com o núcleo *smx*. A macro *INVOKE (A)* é utilizada para lançar a TSI A quando a RSI terminar. A interrupção é terminada por EXIT_ISR(), que activa o escalonador para comutar a execução da tarefa corrente para a TSI A. Se a interrupção terminasse normalmente (sem EXIT_ISR()), a TSI A não seria executada aquando do fim da interrupção, porque apesar estar pronta, o escalonador não seria activado e portanto não existiria nenhuma comutação de tarefa.

Todo o processamento urgente deve ser feito na RSI e não na TSI, porque não há garantia que a TSI seja executada imediatamente a seguir à interrupção. Esta pode ser interrompida por outra RSI ou podem existir outras TSI's pendentes que são executadas antes. Isto não impede que as RSI's devam ser tão rápidas e simples quanto possível.

```
/* Declaração de RSI */
void interrupt far  portin_rsi(void)
{
ENTER_ISR();
/* Lê caracter de entrada e armazena em buffer*/
if /* buffer está cheio */
```

```
        {
        /* Invoca TSI */
        INVOKE (portin_tsi);
        }
EXIT_ISR();
}

/* Declaração de TSI*/
void  portin_tsi ( )
{
/*Envia buffer para processamento por uma tarefa*/
/*Aloca novo buffer para recepção de novos caracteres*/
};
```

Figura 5. 21 - Definição de RSI e TSI (Isr) em smx.

5.3.3.4 - Tratamento de Entrada/Saída

O tratamento de E/S é feito à custa de RSI's, TSI's e canais de transmissão de informação. É apresentado na Figura 5. 22 um exemplo de tratamento de dados de um periférico, adaptado de [Moore97]. A RSI é activada quando o periférico transmite um caracter. A RSI utiliza um *bucket* para armazenar os dados que vai recebendo. Quando este *bucket* está cheio, a RSI activa uma TSI para processar os dados aí armazenados.

```
/* A RSI enche bucket_in caracter a caracter. Quando o bucket está cheio, a RSI invoca a
TSI para que esta envie o bloco de caracteres para uma caixa de correio (para que seja
processado posteriormente por uma tarefa) e aloca ao bucket um buffer vazio para receber
os caracteres seguintes. */

BXCB_PTR    bucket_in;
XCB_PTR     vazios, msgs_in;
```

```
/*Quando a TSI está a ser executada, a variável "esvaziar" garante que a RSI não tenta
armazenar nenhum caracter no bucket, pelo que qualquer dado que chegue durante esse
tempo é ignorado. Uma solução para este problema seria utilizar dois buckets. Desta forma
seria possível encher um bucket enquanto o outro está a ser processado pela TSI. */

char    esvaziar;

/* Declaração da RSI */
void interrupt far portin_rsi (void)
{
        char achar;
        ENTER_ISR();
        /* Lê caracter do porto */
        achar = (char) inp (PORT);
        /* Se o bucket não está a ser esvaziado (pela TSI) então... */
        if (!(esvaziar))
                {
                /* Coloca caracter recebido  no bucket*/
                PUT_CHAR (achar, bucket_in);
                /* Verifica se bucket já está cheio - o campo cc do bloco de controlo de um
                balde indica o número de caracteres livres no seu buffer*/
                if (!(bucket_in->cc))
                        {
                        /*Invoca TSI */
                        INVOKE (portin_tsi);
                        esvaziar = TRUE;
                        }
                }
        EXIT_ISR();
}

/*Declaração da TSI*/
void portin_tsi (void)
{
```

```
MCB_PTR empty_msg, full_msg;

/* Remove mensagem de bucket e envia-a para uma caixa de correio */
if (full_msg = get_msg (bucket_in))
        send (full_msg, msgs_in);
/* Aloca mensagem vazia e põe-na no bucket. O espaço "vazios" contém o conjunto de
mensagens livres disponíveis no sistema (o bucket utiliza uma mensagem como buffer).*/
if (empty_msg = receive (vazios, NO_WAIT))
        {
        put_msg (empty_msg, 100, bucket_in);
        esvaziar = FALSE;
        }
}
```

Figura 5. 22 - Recepção e processamento de dados através de uma RSI e de uma TSI (smx).

5.3.3.5 - Temporizadores

Os *timeouts* são utilizados para garantir que uma tarefa não fica indefinidamente bloqueada numa operação, ou então para gerar esperas de determinada duração. No primeiro caso, os *timeouts* especificam o tempo de espera máximo admissível para essa operação, retornando com aviso de erro quando esse tempo tiver decorrido. É necessário ter em atenção que o *timeout* indicado pelo programador não será necessariamente igual ao tempo decorrido até à operação retornar com o aviso de erro. O tempo de suspensão depende do mecanismo utilizado para gerar os *timeouts.*

A verificação dos *timeouts* é feita por uma rotina do sistema (*clock handler*) activada por uma RSI do temporizador de *hardware*. Esta RSI activa, com uma determinada frequência que é normalmente programável, uma tarefa que efectua uma verificação a uma lista que contém todas as tarefas com um *timeout* a decorrer. Se é encontrada uma tarefa cujo *timeout* já decorreu, então esta é tornada executável, continuando assim a sua execução.

O tempo exacto que uma tarefa irá ficar bloqueada dependerá da activação do *clock handler*. Além disso, quando o *timeout* termina, a tarefa em questão não é posta em execução imediatamente, mas é colocada na lista de executáveis. Isto significa que

a sua entrada em execução irá também depender da sua prioridade, das outras tarefas na lista de execução e da tarefa em execução nesse instante.

5.4 - Caracterização e Comparação de alguns Núcleos Comerciais

5.4.1 - Introdução

Esta secção apresenta vários núcleos disponíveis comercialmente, dando uma breve descrição e análise das suas características principais. Os núcleos analisados são o *RTKernel* da Ontime Informatik Gmbh, o *smx* da MicroDigital Inc. e o *AMX* da Kadak Products Ltd. A informação aqui utilizada foi solicitada e é proveniente dos respectivos fornecedores. Nem sempre foi possível obter toda a informação necessária, já que esta é normalmente disponibilizada de forma fixa em folhetos e brochuras, os quais nem sempre possuíam todos os dados relevantes para a análise pretendida. Todos os dados apresentados são retirados desses elementos fornecidos pelos fabricantes durante o segundo semestre de 1996, devendo ter-se em atenção que qualquer especificação do produto (nomeadamente o preço) pode ter sofrido alguma alteração desde então.

A secção inicia-se com a exposição de alguns factores de avaliação, prosseguindo com a descrição de vários núcleos comerciais e terminando com a comparação das suas características.

5.4.2 - Factores de Avaliação de Núcleos

Tendo por base algumas características comuns, os núcleos comerciais variam segundo vários aspectos, tais como:

- Sistema(s) alvo suportado(s).
- Velocidade de execução e recursos ocupados.
- Facilidades de desenvolvimento oferecidas.

Existem tentativas de caracterização de núcleos através de *benchmarking* [McRae96] ou medição de tempos de execução [Singh96]. No entanto, neste texto pretende obter-se uma medida de desempenho através da análise da utilização do núcleo num sistema integrado de tempo real típico (que utilize a política RMS).

As prestações temporais do núcleo são especialmente importantes para sistemas de tempo real baseados em RMS. Entre estas encontram-se os seguintes factores, que são essenciais para caracterizar a carga de processamento imposta por um núcleo num sistema desse género:

- Latência de interrupções.
- Tempos de comutação de tarefas.
- Precisão das temporizações.

Para analisar a prestação temporal dos núcleos apresentados em aplicações de tempo real, deve, tal como já se referiu, verificar-se qual a carga de processamento adicional que estes impõem à aplicação, e não analisar a velocidade de execução de uma ou outra chamada isoladamente. Quanto maior a carga de processamento imposta pelos serviços do núcleo, menos tempo de execução estará disponível para as tarefas do programador. Este aspecto adquire bastante relevância porque, para manter a escalonabilidade de um sistema em tempo real, é normalmente necessário manter a carga de processamento abaixo de um determinado limite. Merece especial análise a carga de processamento imposta pelo temporizador do núcleo. Quando maior for a precisão desse temporizador, mais vezes será activado o *clock handler* e consequentemente maior será a carga de processamento imposta por esta tarefa. Segundo os cálculos apresentados na secção 5.3.2, a carga de processamento imposta pelo núcleo, num ambiente RMS, pode ser dividida em:

- Carga provocada pelo *clock handler*: $C_h = \frac{T_l + T_T}{P}$
- Carga provocada pelas comutações: $C_c = N \cdot T_{COM} \sum_{i=1}^{N} \frac{1}{T_P^i}$

Os factores P, T_p e N não dependem do núcleo utilizado mas sim da aplicação em causa. Para poder comparar os vários núcleos, pode considerar-se que a aplicação com que estes vão operar será idêntica. Partindo deste pressuposto, podem utilizar-se as seguintes medidas de desempenho:

$$M_h = T_l + T_T$$
$$M_c = T_{COM}$$

A carga de processamento imposta pelo núcleo será proporcional a estes dois factores. O peso do factor M_c na carga do processador torna-se maior com o aumento do número de tarefas. Por outro lado, o factor M_h influencia a carga de processamento segundo a taxa de activação do temporizador. Esta taxa depende da precisão necessária assim como do menor intervalo de tempo que o temporizador necessita de medir. Este tempo será normalmente o período de activação da tarefa com maior frequência do sistema.

Um dos recursos mais críticos, além do tempo do processador, é o espaço de memória ocupado pelo núcleo, quer a nível de dados quer a nível de código. O espaço de dados necessário ao núcleo pode ser dividido da seguinte forma:

- Espaço para blocos de controlo.
- Espaço para pilhas e mensagens.
- Espaço para alocação dinâmica pela aplicação.
- Espaço para estruturas globais.

O espaço de código ocupado pelo núcleo depende do número de funções utilizadas pela aplicação. As funções que não são utilizadas pela aplicação não lhe são normalmente ligadas na *linkagem.* Desta forma, o espaço ocupado em ROM por um núcleo pode variar desde um valor mínimo (para a configuração mínima do núcleo) até um valor máximo (quando todas as funções são utilizadas).

A adaptabilidade do núcleo a diversas plataformas alvo é também um factor importante na sua avaliação. Os métodos mais utilizados para permitir alguma flexibilidade em relação aos sistemas alvo baseiam-se em ficheiros de configuração ou na recompilação do código fonte do núcleo.

Por fim, não se pode deixar de considerar o custo do núcleo como sendo um factor de peso na sua avaliação.

5.4.3 - RTKernel

Gera uma aplicação que funciona em MS-DOS, mas que não executa chamadas a esse sistema operativo. Desta forma as aplicações são *ROMable* e podem ser executadas em qualquer sistema baseado em intel 80x86, mesmo que este não possua o MS-DOS.

Este núcleo oferece um escalonamento preemptivo baseado em prioridades ou *round-robin.* É fornecido tanto sob a forma de uma biblioteca do compilador como em

código fonte. São fornecidas bibliotecas para Borland/Turbo Pascal, Borland/Turbo C++ 1.0, 2.0, 3.0, 3.1, 4.0, BC++ PowerPack, Microsoft C 6.0, 7.0 e 8.0 (Visual C++ 1.0, 1.5), para vários modelos de memória.

Como a aplicação produzida se destina a funcionar sob o sistema operativo MS-DOS, o *debugging* pode ser executado no sistema hospedeiro através de *debuggers* convencionais (TurboDebugger da Borland ou CodeView da Microsoft). São fornecidas duas versões do núcleo: a versão normal e a versão de *debug*. A versão de *debug* à aplicação adiciona código de protecção contra erros e possibilita a utilização de ferramentas de inspecção do núcleo (também fornecidas). A versão normal gera código mais compacto e eficiente, eliminando a verificação e protecção contra erros. Esta versão deve ser utilizada quando o código já foi devidamente verificado e testado e quando a aplicação está pronta a ser gravada em EPROM.

O núcleo é fornecido juntamente com *drivers* para *hardware* típico, tal como o teclado, temporizador, comunicação série e comunicação em rede IPX.

O núcleo suporta a utilização de semáforos, caixas de correio e mensagens síncronas entre duas tarefas.

- **Sistemas alvo suportados**

  Sistemas baseados em intel 80x86, com/sem o sistema operativo MS-DOS.

- **Prestações temporais** (i80286 a 12 Mhz):

  Latência de Interrupções - Indisponível

  Tempo de Comutação de Tarefas

  Round-Robin 58 µs

  Semáforo 89 µs

  Activação (depois de suspensão) 115 µs

  Temporizador

  Baseado no circuito temporizador 8253.

  Período de activação do handler 0.1 ms até 55 ms.

  Tempo de execução do handler Indisponível

- **Recursos consumidos**

  Código 16 k

  Dados 6 k + 500 bytes/tarefa.

- **Adaptabilidade**

O código fonte é fornecido como opção e não é incluído na oferta "base", estando assim o utilizador limitado às funções de biblioteca disponibilizadas. Pode ser configurado conforme a aplicação e *hardware*-alvo pretendidos através da alteração de parâmetros definidos nos ficheiros-cabeçalho.

- **Preço**

RTKernel - C 4.5	DM 800.00
Código Fonte	DM 700.00

Figura 5. 23 - Características do núcleo RTKernel.

5.4.4 - SMX

Este núcleo foi desenvolvido para sistemas integrados baseados na família Intel x86 e compatíveis, e suporta todos os modos de endereçamento destes processadores (modo real e modo protegido em 16 ou 32 *bits*), fornecendo bibliotecas de C para cada um destes modos. São suportados os *assemblers* e compiladores de C de 16 e 32 *bits*: BC++, TASM, MS-Visual C++ e MASM. Os sistemas baseados em μP's 8086 ou 80186 utilizam o modo de endereçamento real, enquanto que os μP's 286 e superiores podem operar nos modos protegidos.

É fornecida uma versão do núcleo para teste e desenvolvimento e uma outra optimizada, sem protecções, para utilização no sistema final.

Este núcleo permite alocar e ligar as pilhas às tarefas de forma dinâmica. Desta forma, uma tarefa que termine o seu processamento pode libertar o seu espaço de pilha para que este possa ser reutilizado por uma outra [Moore96a]. A comunicação de dados é assegurada através de semáforos, *exchanges* e tabelas de acontecimentos (*event tables*).

- **Sistemas alvo suportados**

  Sistemas integrados baseados em intel 80x86 ou 80C188.
- **Prestações temporais** (i80286 a 12 Mhz):

Latência de Interrupções	13μs
Tempo de Comutação de Tarefas	
Semáforo	65 μs
Activação (depois de suspensão)	69 μs

Temporizador	
Baseado no circuito temporizador 8253	
Período de activação do handler	0.1 ms até 55 ms.
Tempo de execução do handler	Indisponível

- **Recursos consumidos**

Código	entre 8694 e 29511 *bytes*.
Dados	500 bytes + 8 a 16 bytes/ bloco de controlo

- **Adaptabilidade**

O código fonte é fornecido opcionalmente. Qualquer configuração do núcleo é feita através de parâmetros definidos num ficheiro (*conf.h*). Depois de efectuadas as alterações, este ficheiro deve ser recompilado.

- **Preço**

smx Development Kit - modo real	3500 U.S. $
smx Development Kit - modo protegido	5500 U.S. $
Código Fonte - modo real	2000 U.S. $
Código Fonte - modo protegido	2500 U.S. $

Figura 5. 24 - Características do núcleo smx.

5.4.5 - AMX

Núcleo para aplicações em tempo real disponível para vários processadores distintos (Motorola 68x0, 683xx, 80386, 80486, Pentium, 80x86/88, PowerPC, R30xx, LR33xxx, i960,29K). Oferece semáforos, tabelas de acontecimentos, caixas de mensagens e *exchanges*.

As tarefas são activadas quando recebem uma mensagem, não sendo necessária a utilização de uma primitiva para as ler. A activação (transição de *idle* para executável) de uma tarefa implica necessariamente que esta recebeu uma mensagem. Esta mensagem é colocada pelo escalonador na pilha da tarefa antes desta ser activada. Uma vez finalizado o processamento dessa mensagem, a tarefa pode simplesmente terminar, encarregando-se o núcleo de a activar novamente quando chegar uma nova mensagem.

O núcleo oferece um esquema de escalonamento preemptivo, com *time-slicing* opcional com tempos ajustáveis por tarefa.

- **Sistemas alvo suportados**

  Sistemas integrados baseados em intel 80x86, 68000, PPC32, i960, MIPS R3000A.

- **Prestações temporais** (i386 20MHz, AMX86):

  Latência de Interrupções 31.9 μs

  Tempo de Comutação de Tarefas 17.6 μs

  Temporizador

  Período de activação do handler - Depende do circuito temporizador utilizado (0.1 ms até 55 ms para um 8253)

  Tempo de execução do handler 20.5 μs

- **Recursos consumidos**

AMX i960	código 11-31K, dados 4k
AMX 68000	código 9 -30k, dados 4k
AMX86	código entre 6588 e 16841 *bytes*, dados 1280 bytes + 386 *bytes*/tarefa.

- **Adaptabilidade**

  O código fonte é fornecido juntamente com o núcleo sem custos adicionais. É fornecido um programa de auxílio à configuração do sistema. Este apresenta os parâmetros de configuração que devem ser definidos e calcula o espaço de memória necessário no sistema alvo para a configuração indicada.

- **Preço**

AMX86 v3 (modo real)	3600 U.S. $
AMX386 (modo protegido)	3600 U.S. $
AMX68000	3600 U.S. $
AMX960	7900 U.S. $

Figura 5. 25 - Características do núcleo AMX.

5.4.6 - Comparação das Características dos vários Núcleos

As prestações temporais de um núcleo podem ser analisadas segundo os dois factores M_h e M_c descritos na secção 5.4.2. Não foi possível obter junto dos fabricantes, apesar de várias tentativas feitas nesse sentido, todos os valores necessários para calcular os dois indicadores de performance. É feita uma análise com os valores disponíveis (note-se que a plataforma de execução do núcleo AMX86 é distinta das demais):

	M_h (µs)	M_c (µs)
RTKernel (i80286 12 MHz)	indisp.	115
smx86 (i80286 12 MHz)	13+ T_T	69
AMX86 (i386 20MHz)	52,4	17,6

Tabela 5. 1 - Prestações dos núcleos analisados.

Na secção 5.4.2 concluiu-se que:

- O impacto na carga de processamento de M_h será proporcional à maior frequência de uma tarefa da aplicação.
- O impacto na carga de processamento de M_c será proporcional ao número de tarefas no sistema.

Desta forma pode observar-se, por exemplo, que, para plataformas idênticas, o núcleo smx86 deverá oferecer melhores prestações do que o núcleo RTKernel no caso de aplicações com múltiplas tarefas. Para o mesmo número de tarefas, a carga de processamento imposta pelo núcleo smx86 é cerca de 60% do mesmo valor para o núcleo RTKernel.

Não é possível a avaliação da carga de processamento imposta pelo temporizador para o núcleo RTKernel por falta de informação. Neste aspecto, as prestações do núcleo smx86 parecem ser relativamente semelhantes às prestações do núcleo AMX numa plataforma mais rápida, admitindo um tempo de execução do *handler* do smx86 de aproximadamente 40 µs.

Os núcleos também podem ser avaliados pelos recursos de memória que consomem.

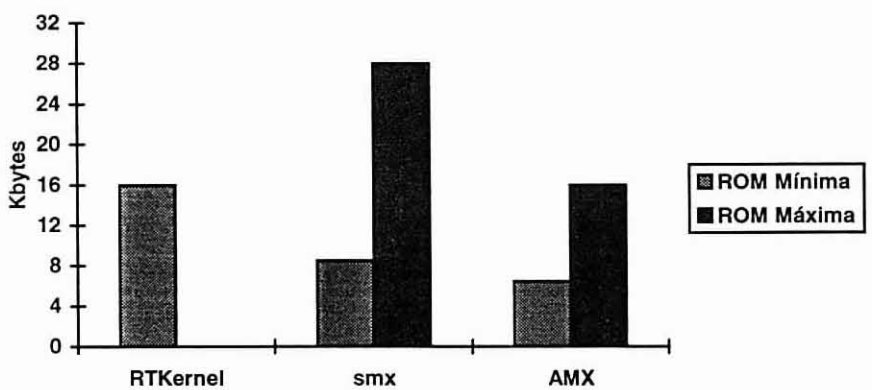

Figura 5. 26 - Ocupação de ROM dos núcleos RTKernel, smx e AMX.

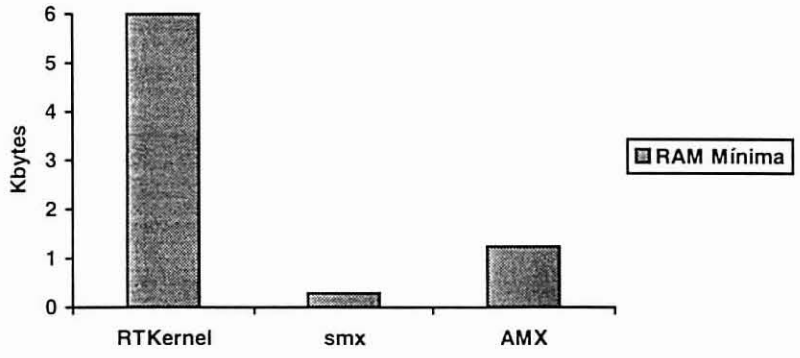

Figura 5. 27- RAM necessária para os núcleos. Não inclui RAM requerida pelas aplicações, ou seja pilhas, blocos de controlo, mensagens, buffers, etc.

O espaço ocupado pelo núcleo deve ser considerado pelo espaço livre que deixa no *chip* de ROM utilizado, porque será este o tamanho máximo que estará disponível para o código da aplicação. Para um *chip* de 32k de ROM, tem-se:

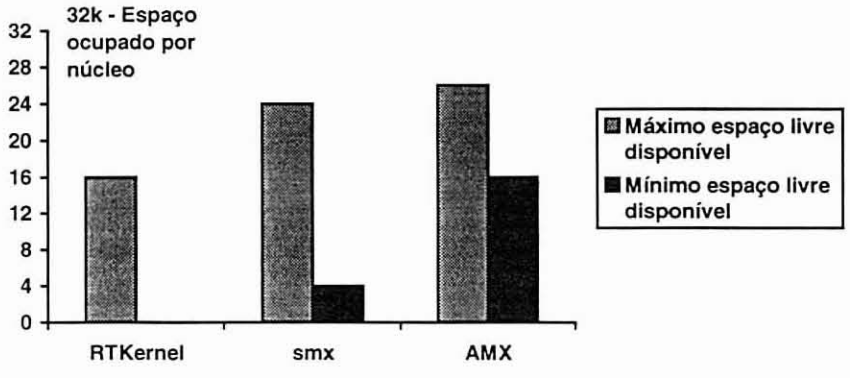

Figura 5. 28 - Espaço disponível para a aplicação se for utilizado um chip de ROM de 32k. Apresentam-se os valores para a utilização *máxima e mínima de ROM por parte dos núcleos.*

5.5 - Conclusões

A carga de processamento adicional introduzida por um núcleo é maioritariamente provocada pelo tempo de comutação de tarefas. Nos núcleos analisados, esse factor é de uma ou mais ordens de grandeza superior à carga imposta pelo temporizador do sistema (dependente da latência de interrupção e do tempo de processamento do *clock handler*).

Verifica-se que a utilização de memória do núcleo smx e do núcleo AMX é razoável, permitindo a coexistência de uma aplicação de tamanho médio (24-26k) no mesmo *chip* de ROM de 32k. O núcleo RTKernel consome completamente essa capacidade de memória, obrigando à utilização de um *chip* de capacidade maior ou um *chip* adicional.

Capítulo 6

Conclusões e Trabalhos Futuros

O desenvolvimento de *software* para sistemas integrados envolve normalmente a utilização de uma grande variedade de ferramentas e o conhecimento das características da plataforma de *hardware* no qual vão ser executados os programas. Por vezes, no funcionamento destes sistemas, o tempo de execução de determinadas tarefas pode ser crítico, sendo essencial, nesse caso, o domínio de técnicas de desenvolvimento de sistemas de tempo real.

Este texto descreve, de forma abrangente, o ciclo de desenvolvimento de *software* para pequenos sistemas integrados, terminando com uma análise das implicações da utilização de núcleos de tempo real nesse processo. Por pequenos sistemas integrados entende-se qualquer sistema baseado em microprocessador ou microcontrolador até 16 *bits* (eventualmente 32) com limitações a nível de memória RAM e ROM. Esta é a dimensão típica de muitos sistemas utilizados em Instrumentação.

O recurso a núcleos de tempo real favorece uma abordagem sistematizada do desenvolvimento de *software* para sistemas integrados, quer a nível de estruturação do código (favorecendo a sua divisão em tarefas), quer da utilização de metodologias de desenvolvimento de *software* para sistemas de tempo real. É também favorável sob o ponto de vista da fiabilidade do sistema desenvolvido, uma vez que o código do

núcleo já foi, em princípio, utilizado em várias aplicações distintas, sendo assim provável que esteja relativamente isento de defeitos.

As motivações que levaram a esta dissertação prendem-se com o facto de se desconhecerem estudos concisos e pragmáticos que permitam aplicar metodologias estruturadas ao desenvolvimento de sistemas integrados de pequeno porte. O mesmo se passa relativamente à sua especificação com vista à implementação numa perspectiva tempo real recorrendo ou não a um núcleo.

Pensa-se pois que o desenvolvimento de Sistemas Integrados na área da Instrumentação poderá beneficiar dos resultados de estudos que conduzam a metodologias, métricas, análises comparativas de ferramentas de *software*, núcleos e todos os elementos relevantes para o processo. Espera-se que esta dissertação possa contribuir um pouco para colmatar a lacuna. O seu cariz inerentemente limitado espacial e temporalmente, leva a que se perspectivem estudos em sequência, tais como:

- Análise e caracterização de ferramentas de desenvolvimento:
    1. Desenvolvimento de métricas e análise comparativa de ferramentas (compiladores, núcleos, etc), sob a perspectiva da sua utilidade para o desenvolvimento de pequenos sistemas integrados.
- Metodologias para desenvolvimento de *software* para sistemas integrados:
    2. Modelação e caracterização de sistemas integrados (segundo a sua funcionalidade e tipologias de *hardware* e *software*), definindo paralelamente um método de especificação.
    3. Desenvolvimento de directivas de implementação que transformem uma especificação da funcionalidade do sistema numa especificação da tipologia do sistema, com uma garantia mínima de viabilidade da solução apresentada.

As mesmas limitações reflectiram-se na vertente do trabalho sobre núcleos. Parecem óbvias as vantagens da utilização de núcleos comerciais pelo que se esboçam metodologias de análise quantitativa. Esta contribuição carece agora de aprofundamento quer ao nível de alargar o número de núcleos estudados, quer de afinar critérios, quer ainda de comparar os resultados teóricos com resultados

experimentais. Esta é uma outra das direcções segundo as quais se espera desenvolver o trabalho subsequente.

Referências Bibliográficas

[Antonakos96] ***An Introduction to the Intel Family of Microprocessors.*** James L. Antonakos, Prentice Hall (1996).

[Antoniazzi++93] ***X-Nets: a Visual Formalism for System Specification and Analysis.*** S. Antoniazzi, A. Balboni, W. Fornaciari, Proceedings EUROMICRO 93, Open System Design: Hardware, Software and Applications, Barcelona (1993).

[BakerScallon86] ***An Architecture for Real-Time Software Systems.*** Theodore Baker, Gregory Scallon, IEEE Software, 2(5):50-58 (Maio 1986).

[Beach96] ***C51 Primer.*** Mike Beach, Hitex Ltd (Março 1996).

[BigazziWeinberg91] ***Embedding Assembly Language in C.*** Antonio Bigazzi, William Weinberg, Microtec Research Inc (1991).

[BlakesleeLiband93] ***Real-Time Debugging Techniques: Hardware-Assisted Real Time Debugging Tecniques for Embedded Systems.*** Thomas Blakeslee, Jan Liband, Embedded Systems Programming, vol.8, nº4 (1993).

[BookerMcKeeman93] ***Design considerations for RISC microprocessor in realtime embedded systems.*** Alan Booker, John McKeeman, Computer Design (Agosto 1993).

[Bradley94] ***A formally based hard real-time kernel.*** Steven Bradley, Microprocessors and Microsystems, vol 18, nº9 (Novembro 1994).

[Brown94] ***Embedded Systems Programming in C and Assembly.*** Jonh Forrest Brown, VNR Computer Library (1994).

[ChatterjeeStrosnider96] ***Quantitative Analysis of Hardware Support for Real Time Operating Systems***. Saurav Chatterjee, Jay Strosnider, Real-Time Systems, 10 pp.123- 142 (1996).

[Clement92] ***Microprocessor Systems Design. 68000 Hardware, Software, and Interfacing***. Alan Clement, PWS Publising Company (1992).

[Colnaric94] ***A Hardware Supported Operating System Kernel for Embedded Hard Real Time Applications***. Matjaz Colnaric, Microprocessors and Microsystems, vol 18, n°10 (Dezembro 1994).

[Cooling91] ***Software Design for Real-Time Systems***. J.E. Cooling, Chapman and Hall (1991).

[Cooling94] ***Task Scheduling in Hard Real-Time Embedded Systems Using Hardware Co-Processors***. Jim Cooling, Microprocessors and Microsystems, vol 18, n°10 (Dezembro 1994).

[Cooling96] ***Languages for the programming of real-time embedded systems. A survey and comparison***. J. E. Cooling, Microprocessors and Microsystems, vol 20, n° 2 (Abril 1996).

[Dawson95] ***Performance enginnering - how well does it really work?***. Kevin Dawson, Microprocessors and Microsystems, vol 19, n°2 (Março 1995).

[Drusinsky95] ***Visually Designing Embedded-Systems Applications***. Doron Drusinsky, Dr. Dobb´s (Junho 95).

[Ganssle90] ***The ICE blues***. Jack G. Ganssle, Embedded Systems Programming (Novembro 1990).

[Ganssle92a] ***The Art of Programming Embedded Systems***. Jack G Ganssle, AP - Academic Press, 1ª edição (1992).

[Ganssle92b] ***Writing Relocatable Code***. Jack G. Ganssle, Embedded Systems Programming (Fevereiro 1992).

[Ganssle93] ***Reentrancy***. Jack G. Ganssle, Embedded Systems Programming (Fevereiro 1993).

[Ganssle94a] ***Coding ISRs***. Jack G. Ganssle, Embedded Systems Programming (Julho1994).

[Ganssle94b] ***Troubleshooting. Avoiding the bog.*** Jack G. Ganssle, EDN, 29 (Setembro 1994).

[Ganssle95a] ***Design Debuggable Hardware.*** Jack G. Ganssle, EDN, 2 (Março 1995).

[Ganssle95b] ***In μP's, smaller is sometimes better.*** Jack G. Ganssle, EDN, 25 (Maio 1995).

[Ganssle95c] ***Living to Learn.*** Jack G. Ganssle, EDN, 6 (Julho 1995).

[Ganssle95d] ***Thanks for the Memories.*** Jack G. Ganssle, Embedded Systems Programming (Agosto 1995).

[Ganssle95e] ***Banking Basics.*** Jack G. Ganssle, EDN, 28 (Setembro 1995).

[Ganssle95f] ***Assume nothing. Test everything.*** Jack G. Ganssle, EDN, 23 (Novembro 1995).

[Ganssle96a] ***Debugging ISRs - Part 1.*** Jack G. Ganssle, Embedded Systems Programming (Maio 1996).

[Ganssle96b] ***Debugging ISRs - Part 2.*** Jack G. Ganssle, Embedded Systems Programming (Junho 1996).

[GrayMulchandani97] ***Object File Formats.*** Rand Gray, Deepak Mulchandani, Dr. Dobb's Journal (Maio 1997).

[Greenbrg90] ***Interrupt Functions and ANSI Keywords.*** R. Greenbrg, Microtec Research Inc. (1990).

[Halang90] ***Comparative Evaluation of High-Level Real-Time Progamming Languages.*** Wolfgang Halang, Real-Time Systems, 2 pp.365-382 (1990).

[Intel95] ***Intel MCS® Tools Handbook***, MW Media, 3ª edição (1995).

[Johnson93] ***Assembly Language for Real Programmers Only.*** Marcus Johnson, Sams Publishing (1993).

[KattilakoskiHonka91] ***A simulation-based system for testing real-time embedded software in the host environment.*** Matti Kattilakoski, Hannu Honka, Microprocessing and Microprogramming 32 (1991).

[Kauler95] ***TERSE: A tiny real-time operating system.*** Dr. Dobb's (Dezembro 1995).

[Kauler96] *A Tiny Microcontroller Dataflow Kernel.* B. Kauler, Microprocessors and Microsystems, vol 20, nº 2 (Abril 1996).

[Korouši c-Seljak94] ***Task Scheduling Policies for Real-Time Systems.*** Barbara Korousic-Seljak, Microprocessing and Microsystems, vol 8, nº9 (Novembro 1994).

[Laplante93] ***Real-Time Systems Design and Analysis, an Engineer's Handbook.*** A. Philip Laplante, IEEE Computer Society Press (1993).

[Leach93] ***Using C in Software Design.*** Ronald Leach, Academic Press Professional (1993).

[LiuGibson86] ***Microcomputer Systems: the 8086/8088 Family Architecture, Programming, and Design.*** Yu-Cheng Liu, Glenn A. Gibson. Prentice Hall, 2ª edição (1986).

[LiuLayland91] ***Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment.*** C. L. Liu, J. W. Layland, ACM vol 20 nº1 (1991).

[MacKenzie95] ***The 8051 Microcontroller.*** I. Scott MacKenzie, Prentice Hall (1995).

[Mazur92] ***Moving from Assembly to C.*** Beth Mazur, Dr. Dobb's Journal (Agosto 1992).

[McRae96] ***Benchmarking Real-Time Operating Systems.*** Eric McRae, Dr. Dobb's (Maio 1996).

[Microtec96] ***Choosing the Right Embedded Software Development Tools.*** Microtec Inc. (1996).

[MillerQuilici93] ***The joy of C.*** Lawrence Miller, Alexander Quilici (1993).

[Moore91] ***Preemptive Virtues.*** Ralph Moore, Printed Circuit Design (Fevereiro 1991).

[Moore95a] ***How to Use Real-Time Multitasking Kernels In Embedded Systems.*** Ralph Moore, Micro Digital Inc. (1995).

[Moore95b] ***Why Use a Preemptive Multitasking Kernel?.*** Ralph Moore, Micro Digital Inc. (1995).

[Moore95c] ***Smx, Simple Multitasking Executive, Version 3.2, User's Guide.*** Ralph Moore, Micro Digital Inc. (Novembro 1995).

[Moore96a] ***Unbound Stacks and Stoppable Tasks***. Ralph Moore, Micro Digital Inc. (1996).

[Moore96b] ***Link Service Routines, an aid to better embedded systems***. Ralph Moore, Micro Digital Inc. (1996).

[Porat92] ***Simple guidelines help you implement a multitasking system***. Zvi Porat, EDN, 10 (Dezembro 1992).

[Ready86] ***VRTX: A Real-Time Operating System for Embedded Microprocessor Applications***. James Ready, IEEE Micro, 6(4):8-17 (Agosto 1986).

[Rosenthal94] ***For digging out hardware bugs, monitor programs fill the bill***. Scott B. Rosenthal, Personal Engineering & Instrumentation News (Julho 1994).

[Rosenthal95a] ***The search for the perfect software might be never ending, but it's not pointless***. Scott B. Rosenthal, Personal Engineering & Instrumentation News (Março 1995).

[Rosenthal95b] ***Interrupts might seem basic, but many programamers still avoid them***. Scott B. Rosenthal, Personal Engineering & Instrumentation News (Maio 1995).

[Rosenthal95c] ***Modern development tools remove complications - but at what price?***.Scott B. Rosenthal, Personal Engineering & Instrumentation News (Novembro 1995).

[Rosenthal96] ***Software Development and Standards Manual***. Scott B. Rosenthal, MicroSol Corp. (Janeiro 1996).

[Ryan88] ***Intel's 80960: An Architecture Optimized for Embedded Control***. David Ryan, IEEE Micro (Junho 1988).

[Sacha93] ***Real-Time Specification Using Petri Nets***. K. Sacha, Proceedings EUROMICRO 93, Open System Design: Hardware, Software and Applications, Barcelona (1993).

[SainGonsalves97] ***KeRTESy: A Real-Time Event-Driven Microkernel***. Biswajit Sain, Timothy A. Gonsalves, Dr. Dobb's Journal (Março 1997).

[Seyer91] ***RS-232 Made Easy Connecting Computers, Printers, Terminals, and Modems***. Martin D. Seyer, PTR Prentice Hall, 2ª edição (1991).

[Singh96] ***Metrics needed for Real-Time Operations.***. Inder Singh, Lynx Real Time Systems Inc. (1996).

[Snyder95] ***Fast Context Switches: Compiler and Architectural Support for Preemptive Scheduling***. Jeffrey S. Snyder, Microprocessors and Microsystems, vol 19, nº1 (Fevereiro 1995).

[Švéda92] ***Microcontroller Software Engineering***. Miroslav Švéda, Microprocessing and Microprogramming 34 (1992).

[Talbot93] ***RISC vs. CISC for realtime: a software perspective***. James Talbot, Computer Design (Agosto 1993).

[Wallace95] ***Functional Programming and Embedded Systems***. Malcom Wallace, PhD Thesis, Department of Computer Science, University of York (Janeiro 1995).

[Walls96a] ***How Software Influences Hardware Design***. Colin Walls, Microtec Research Inc (1996).

[Walls96b] ***Real Time Systems***. Colin Walls, Microtec Research Inc (1996).

[Walls96c] ***Self-Testing in Embedded Systems***. Colin Walls, Microtec Research Inc (1996).

[Warntjes96] ***Using C for inicial hardware verification***. Steve Warntjes, Hewlett Packard (1996).

[YeralanAhluwalia95] ***Programming and Interfacing the 8051 Microcontroller***. Sencer Yeralan, Ashutosh Ahluwalia, Addison Wesley Publishing Company (1995).

Apêndice A

Teste e Inicialização de Dispositivos de Memória

A.1 - Sobreposição de RAM e ROM durante a Inicialização

Devido às particularidades de alguns microcontroladores, por vezes os projectistas de sistemas integrados têm necessidade de alterar o espaço de endereçamento do sistema durante o processo de inicialização. Este problema surge, por exemplo, em sistemas baseados nos microprocessadores 68000.

A que se deve esta necessidade? Recorrendo de novo àquele exemplo, os processadores 68000 vão buscar o endereço da primeira instrução a ser executada e o endereço da pilha à posição de memória 000000h. Isto significa que esta posição de memória deve estar mapeada em ROM no início do arranque do sistema. No entanto, a tabela de excepções do 68000 é também mapeada nas posições de memória mais baixas.

Em alguns sistemas pode ser possível definir esta tabela em ROM de forma fixa, mas existem outros em que a sua definição tem de ser feita de forma dinâmica, durante a execução do programa. Neste caso, põe-se o problema de mapear ROM e RAM nas mesmas zonas de memória. Este problema surge sempre quando é necessário utilizar RAM num espaço que necessite de ter dados armazenados de forma permanente (logo deveriam estar armazenados em ROM). Verifica-se principalmente devido a conflitos entre a localização do código de inicialização e a localização de estruturas dinâmicas, como são, por exemplo, os vectores de excepção.

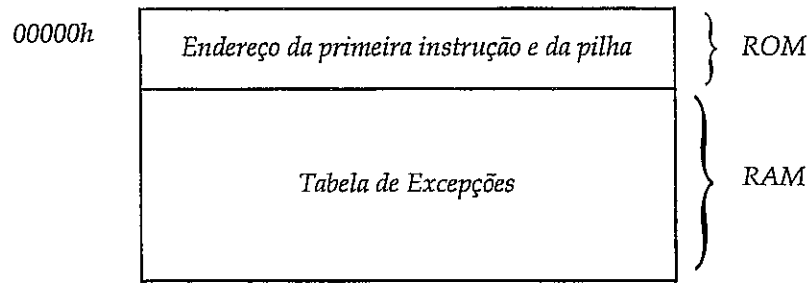

Figura A. 1- Mapa de Memória de um sistema baseado em M68000.

Esta sobreposição de ROM com RAM pode ser resolvida de várias formas distintas. Uma abordagem possível consiste na utilização de uma ROM "fantasma". Esta ROM está activa apenas durante a fase de arranque do sistema, sendo substituída, imediatamente antes da inicialização terminar, por RAM localizada no mesmo espaço de endereçamento. O processo de arranque passa a consistir em:

1. Teste da RAM.
2. Teste da ROM.
3. Copiar código de inicialização para RAM.
4. Continuar execução de rotina de inicialização em RAM.
5. Desligar ROM/ Ligar RAM através da activação de um bit de I/O.
6. Iniciar execução da aplicação.

Esta solução obriga à utilização de *hardware* que efectua a comutação ROM/RAM. Basta no entanto, aceder a um *bit* de I/O, activado pela rotina de inicialização.

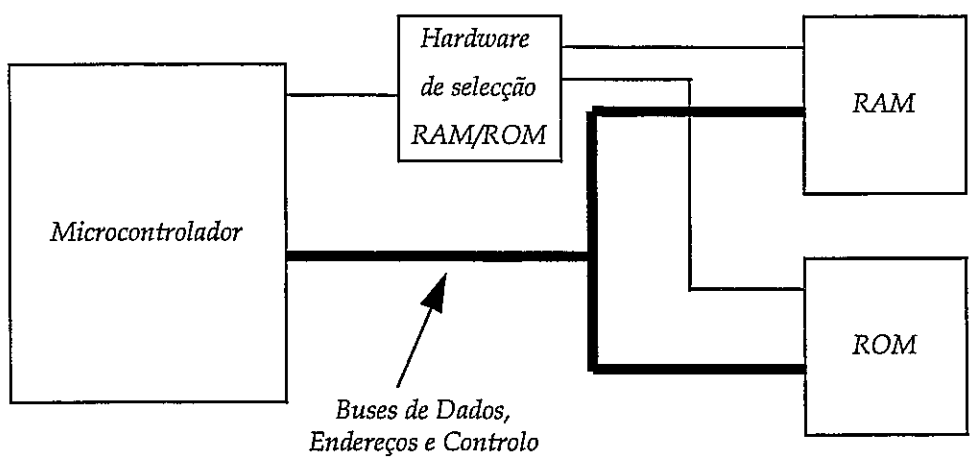

Figura A. 2- Hardware para trocar memória ROM "fantasma" por RAM.

O código da rotina de inicialização deve ser completamente relocatável, para possibilitar a sua execução no novo endereço para onde é copiado. Ou, alternativamente, o código que é executado a partir da RAM já deve ser compilado para ser executado no endereço final, apesar de ser armazenado em ROM na parte baixa de memória.

Uma desvantagem deste método reside na necessidade de existência de um espaço de endereçamento de RAM fixo, para servir de zona de execução intermediária quando a ROM é desligada e a RAM ligada. Além disso, a mudança de endereço de execução do código torna o desenvolvimento e *debugging* do *software* mais complexo, principalmente se for utilizada uma linguagem de alto nível.

Estas dificuldades seriam resolvidas se a transição de ROM para RAM não alterasse os endereços em que se encontram o código e dados. Para o conseguir, é necessário que o programa possa ser executado nos endereços inferiores de memória a partir de ROM, mas que todas as escritas nos mesmos endereços sejam armazenadas em RAM (para permitir a cópia do conteúdo da ROM para RAM). É, portanto, necessário *hardware* que forneça a seguinte descodificação:

	Durante a inicialização	Depois da inicialização
Operações de leitura	ROM	RAM
Operações de escrita	RAM	RAM

Tabela A. 1 - Descodificação da memória antes e depois do arranque do sistema.

Neste caso, não é possível efectuar o teste da RAM antes de fazer a transição ROM->RAM porque, antes desta, só é possível a escrita e não a leitura do conteúdo de RAM. A sequência de inicialização será:

1. Teste de ROM.
2. Copiar conteúdo de ROM para RAM.
3. Desligar ROM/ Ligar RAM através da activação de um bit de I/O.
4. Teste de RAM. Nesta altura pode ser tarde demais, porque se existir um defeito de RAM, o próprio código de inicialização pode ter sido mal copiado. Mas se a RAM for constituida por múltiplos circuitos integrados, deve-se sempre verificar pelo menos os restantes *chips*.
5. Iniciar execução da aplicação.

Note-se que, ao contrário do que acontecia no caso anterior, não é preciso existir um salto para continuação da execução da rotina de inicialização, porque a passagem ROM->RAM foi feita de tal forma que a imagem da memória permanece inalterada, sob o ponto de vista do microprocessador. Também não é necessária a existência de RAM disponível antes da transição, porque não é utilizada nenhuma zona para armazenamento intermediário de código. O *hardware* adicional é ligeiramente mais complicado do que no método anterior, mas o desenvolvimento do *software* é mais simplificado.

A.2 - Teste da RAM

O teste à RAM consiste basicamente em escrever e ler em seguida um determinado valor numa posição de memória. Se o valor lido for diferente do valor escrito, então existe concerteza algum problema relacionado com a RAM.

O acesso a RAM pode falhar por dois motivos fundamentais:

- Falhas internas do circuito de RAM.
- Falhas externas (falhas na descodificação, refrescamento, controlo, etc).

A memória RAM falha internamente, geralmente, apenas durante o *power-up*. Por essa razão é conveniente, e geralmente suficiente, o seu teste durante a inicialização do sistema.

Para garantir que todos os *bits* de todas as posições de memória sejam exercitados durante um teste, são bastante utilizados dois pares de valores:

- O par 00h - FFh: utilizado para detectar *sticky-bits* (*bits* que estão fixos num determinado valor).
- O par 55h - AAh: utilizado para detectar *crosstalk* entre *bits* vizinhos.

Os principais inconvenientes no recurso a todos estes valores são: o tempo elevado que demora a executar o programa de teste (cada valor de teste utilizado obriga a aceder duas vezes a cada posição de RAM) e, como já se referiu, a inexistência da necessidade de testar todos os *bits* individualmente. Além disso, este teste não detecta falhas nas linhas de endereçamento. Por exemplo, se algumas destas estiverem curto-circuitadas, então este teste limitar-se-ia a escrever e a ler o mesmo valor sempre nas mesmas células de memória. Por exemplo, mesmo que o processador pense que está a escrever e a ler nas posições de memória 0000h, 0001h,

... , um erro nas linhas de endereço pode transformar todas as operações de escrita e de leitura em acessos à posição 0000h (ver Figura A. 3). Se existir um erro de descodificação e for seleccionado sempre o mesmo *chip* de RAM, este teste também não o irá detectar.

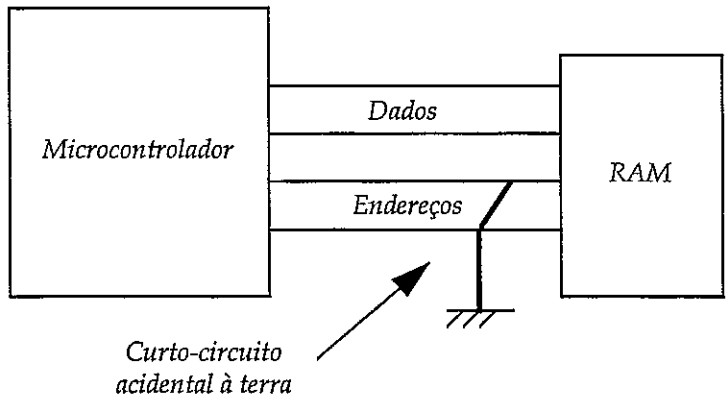

Figura A. 3 - Erro de acesso à RAM por curto-circuito das linhas de endereço.

Este problema surge porque os valores de teste referidos não possuem qualquer informação sobre os endereços em que vão ser armazenados, pelo que o processador não pode ter a certeza da célula de memória que está realmente a ler o valor.

Para resolver a situação anterior, é prática comum utilizar, para efeitos de teste, valores que estejam directamente relacionados com o endereço onde são armazenados. A forma mais simples de o conseguir é usar os *bits* de endereço menos e mais significativos da célula de memória acedida. Ou seja, os valores a utilizar serão:

Endereço	*Valor*
0000h	*00h*
0001h	*01h*
...	
00FFh	*FFh*
0100h	*00h*

e, numa segunda fase de teste

Endereço	*Valor*
0000h	*00h*
0000h	*00h*
...	
00FFh	*00h*
0100h	*01h*

Figura A. 4 - Sequências de teste para detecção de erros de endereçamento.

Desta forma, o teste consegue detectar problemas no endereçamento e descodificação da memória.

Este teste não evita o inconveniente de necessitar de quatro acessos por posição de memória. Se a velocidade de execução do teste for um factor essencial, então é desejável reduzir o teste de forma a ler e escrever um único valor em cada célula de memória.

Para este efeito, é possível desenvolver uma forma alternativa de teste. Em vez de escrever o mesmo valor em todas as posições de memória, é definida uma sequência de *bytes*. Esta é escrita na RAM até que todo o espaço de endereçamento esteja coberto. A RAM é lida em seguida e os seus valores comparados com os da sequência original. Este método limita os acessos à RAM a dois por célula de memória.

Existem duas questões que se podem levantar: qual o comprimento adequado da sequência e quais os valores que esta deve conter?

A sequência pode ser semi - "aleatória", incluindo de preferência os valores de teste "normais" - AAh, 55h, 00h e FFh. O seu valor não é verdadeiramente importante, desde que inclua valores relativamente distintos.

O comprimento da sequência de valores é, no entanto, de relativa importância. Este deve ser tal que o início de cada bloco não esteja alinhado nos *bits* menos significativos do endereço. Isto é importante para garantir que sejam detectados quaisquer erros nas linhas de endereços ou descodificação. A Figura A. 5 apresenta o exemplo de uma situação anómala não detectada por sequências de valores de determinados comprimentos.

Um valor óptimo para o comprimento do bloco de teste é 257. Como é um número primo (não está alinhado nos *bits* menos significativos), vai começar um bloco de dados novo em todas as combinações possíveis de endereços menos significativos (8 *bits*) até 64k. Além desta vantagem, o comprimento 257 tem outra característica agradável: é possível incluir todas as combinações binárias (00h-FFh) na sequência de teste!

Uma vez escritos os dados de teste em RAM, eles vão ser lidos de volta para confirmar que todo o sistema de RAM está operacional. O método normal é a comparação directa do valor lido com o valor previamente escrito. No entanto, é possível fazer esta verificação através de um *checksum* ou CRC. Este método pode ser um pouco mais rápido do que a comparação individual de cada valor, mas tem o inconveniente de não permitir detectar onde a falha ocorreu.

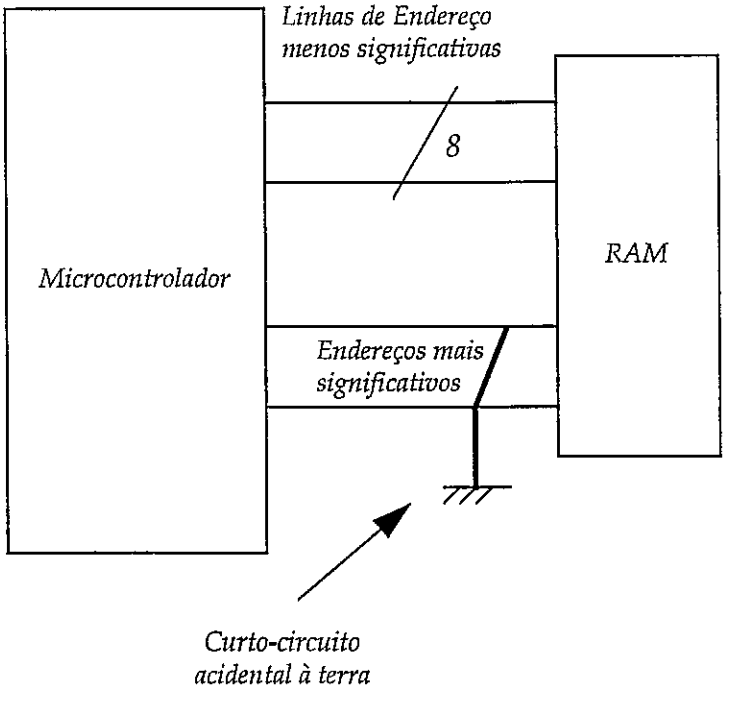

Figura A. 5 - Endereçamento anómalo não detectado por sequências de teste de comprimento 1, 2, 4, 8, 16, 32, 64, 128 e 256 bytes.

A.3 - Teste da ROM

O código de inicialização do sistema está, como aliás todo o programa da maioria dos sistemas integrados, armazenado em ROM. Se a ROM falha de tal forma que o próprio código de inicialização se torne inacessível, então o sistema nem sequer consegue arrancar. A execução do teste de ROM pressupõe que pelo menos o próprio código de teste está acessível.

O teste de uma ROM destina-se principalmente a detectar dispositivos parcialmente programados (devido a um apagamento incompleto, ou remoção inadvertida do dispositivo antes de terminar a programação), além de potenciais problemas no endereçamento e descodificação.

Como não é possível alterar o conteúdo de uma ROM, não é possível a utilização de métodos de teste semelhantes aos utilizados para RAM. Por isso, a detecção de erros numa ROM é limitida a *checksums*, CRC's, paridades, etc.

O *checksum*, apesar de ser facilmente implementável e relativamente fiável, pode não detectar alguns erros de montagem no circuito como, por exemplo, a troca de posição de dois *chips* de ROM no circuito, já que neste caso o *checksum* não é alterado. O CRC é um método mais evoluído, relativamente simples de implementar,

cuja precisão é muito maior do que o simples *checksum*, pelo que também é frequentemente usado.

Um truque comum, utilizado para facilitar a descoberta de problemas de endereçamento, e para oferecer alguma capacidade de diagnóstico mesmo que o programa de teste não chegue a ser executado, é o preenchimento dos espaços não utilizados de ROM com um *op-code* com comportamento determinado e facilmente observável. Este *op-code* deve consistir num único *byte*. Se for utilizada uma instrução com um *op-code* de 2 *bytes*, por exemplo, existe o risco de o problema de endereçamento resultar num salto para uma posição de memória no "meio" da instrução.

Por exemplo, no microprocessador Z80, o *op-code* FFh é uma chamada à posição de memória 38. O programador pode aproveitar-se deste facto e preencher todos os espaços de ROM não utilizados com este valor. Se existir um problema de endereçamento, então existe a probabilidade de o microcontrolador executar múltiplas chamadas à posição 38. As sucessivas chamadas vão provocar sistematicamente dois acessos consecutivos à pilha, os quais podem ser facilmente detectados através da inspecção da actividade do *bus*.

Apêndice B

Publicações Periódicas e Fornecedores de Ferramentas

B.1 - Publicações Periódicas

Apresentam-se os nomes e endereços de algumas publicações periódicas que normalmente abordam assuntos de interesse para a área de sistemas integrados.

C/C++ User's Journal
R & D Publications Inc.
2601 Iowa Lawrence, KS 66046
Telef: (913) 841-1631

Computer Design
Pennwell Publishing Company
One Technology Park Drive, P.O. Box 990, Westford, MA 01886
Telef: (508) 692-0700

Computer Language
Miller Freeman Publications.
500 Howard Street, San Francisco, CA 94105
Telef: (415) 397-1881

Design Automation for Embedded Systems
Kluwer Academic Publishers.

Dr Dobb's Journal
Miller Freeman Publications.
411 Borel Avenue, San Mateo, CA 94402.
Telef: (415) 358-9500

EDN Magazine
Cahners Publishing Company.
275 Washington Street, Newton, MA 02158

Electronics
Penton Publishing.
611 Route #46 West, Hasbrouck Heights, NJ 07604
Telef: (201) 393-6060

Electronic Design
A Penton Publication
P.O. Box 985007, Cleveland OH 44198-5007 USA

Electronic Enginnering Times
CMP Publications.
600 Community Drive, Manhasset, NY 11030
Telef: (516) 562-5000

Embedded Systems Programming
Miller Freeman Publications.
500 Howard Street, San Francisco, CA 94105
Telef: (415) 397-1881

Personal Engineering & Instrumentation News

Personal Enginnering Community.

Box 430, Rye, NH 03870

Telef: (617) 232-3625

Microprocessors and Microprogramming

EUROMICRO

P.O. Box 2346, NL-7301 EA Apeldoorn, The Netherlands

Telef: (31) (55) 557372

Fax: (31) (55) 557393

Microprocessors and Microsystems

Butterworth-Heinemann Ltd.

Linacre House, Jordan Hill, Oxford OX2, 8DP, UK.

Telef: +44 (0) 865 310366

Fax: +44 (0) 865 310898

Real-Time Systems

Kluwer-Academic Publishers

P.O. Box 358, Accord Station, Hingham, MA 02018-0358.

B.2 - Fornecedores de Ferramentas

Apresentam-se o nome de algumas companhias fabricantes e/ou fornecedoras de ferramentas para o desenvolvimento de sistemas integrados. Os dados que constam nesta lista são relativos aos primeiros meses de 1997.

Archimedes Software, Inc.

303 Parkplace Center, Kirkland, WA 98033, USA

Telef.: (206) 822-6300. Fax: (206) 822-8632

Web: http://www.archimedesinc.com

Assemblers, Compiladores de C,Debuggers, Simuladores e Monitores.

Avocet Systems, Inc.

120 Union Street, P.O. Box 490, Rockport , ME 04856, USA

Telef.: (800) 448-8500. Fax: (207) 236-6713

Email: avocet@midcoast.com; Web : http://www.midcoast.com/~avocet

Assemblers, Compiladores de C, Debuggers, Simuladores e Emuladores.

CEIBO, Inc.

7 Edgestone Court, Florissant , MO 63033, USA

Telef.: (314) 830-4084. Fax: (314) 830-4083.

Email: ceibo@trendline.co.il; Web: http://www.ceibo.com

Assemblers, Compiladores de C, Debuggers, Simuladores e Emuladores.

ChipTools Inc.

Telef: (905)274-6244. Fax: (905)891-2715

Email: chiptool@hookup.net; Web: http://www.chiptools.com/

Debuggers, Simuladores, Monitores, Emuladores e Núcleos de tempo real.

CMX Company

5 Grant Street, Suite C, Framingham , MA 01701, USA

Telef:(508) 872-7675. Fax: (508) 620-6828

Email: cmx@cmx.com; Web: http://www.cmx.com

Compiladores de C e Núcleos de tempo real.

Crossware Products

St John's Innovation Centre, Cowley Road, Cambridge, CB4 4WS, UK

Telef: +44 1223 421263. Fax: +44 1223 421006

Email: sales@crossware.com; Web: http://www.crossware.com

Assemblers, Compiladores de C, Debuggers e Simuladores.

Embedded System Prod.

11501 Chimney Rock, Houston, TX 77035-2900

Telef: (800)525-4302. Fax: (713)728-1049

Núcleos de tempo real.

Emulation Technology Inc.

2344 Walsh Avenue, Building F,Sta Clara , CA 95051-1301, USA

Telef.: (408) 982-0660. Fax: (408) 982-0664.

Email: ET@pmail.emulation.com; Web: http://www.emulation.com

Emuladores.

Franklin Software, Inc.

888 Saratoga Avenue, Suite #2, San Jose, CA 95129.

Telef: (408) 296-8051. Fax: (408) 296-8061

Assemblers, Compiladores de C, Debuggers, Simuladores e Núcleos de tempo real.

Hewlett-Packard Company

1900 Garden of the Gods Rd, Colorado Springs, CO 80907, USA.

Fax: (719) 590-2121.

Web: http://www.hp.com

Emuladores.

Hitex Development Tools

2055 Gateway Place, Suite 400, San Jose , CA 95110, USA

Telef.: (408) 298-9077. Fax: (408) 441-9486.

Email: info@hitex.com; Web: http://www.hitex.com

Debuggers, Simuladores, Monitores e Emuladores.

Huntsville Microsystems

3322 South Memorial Parkway, Building 500, P.O. Box,12415, Huntsville, AL 35801, USA

Telef.: (205) 881-6005. Fax: (205) 882-6701

Email: support@hmi.com or sales@hmi.com; Web: http://www.hmi.com

Compiladores de C, Debuggers, Simuladores e Emuladores.

IAR Systems Software

One Maritime Plaza, Suite 1770, San Fransisco, CA 94111 USA

Telef: (415)765-5500. Fax: (415)765-5503

Assemblers, Compiladores de C, Simuladores e Núcleos de tempo real.

Intel Corp.

5000 W. Chandler Boulevard, Chandler , AZ 85226, USA

Telef.: (800) 628-2283 ou (916) 356-3104

Web: http://www.intel.com

Assemblers e Núcleos de tempo real.

Kadak Production Ltd.

#206-1847 West Broadway, Ave. Vancouver, BC, V6J1Y5 Canada.

Telef.: (604) 734-2796. Fax: (604) 734-8114.

Email: amxsales@kadak.com; Web: http://www.kadak.com

Núcleos de tempo real.

Keil Software, Inc.

16990 Dallas Parkway, Suite 120, Dallas , TX 75248-1903, USA

Telef.: (800) 348-8051. Fax : (214) 735-8051

Email : sales@keil.com; Web : http://www.keil.com

Assemblers, Compiladores de C, Debuggers, Simuladores, Monitores e Núcleos de tempo real.

Micro Digital Inc.

12842 Valley View St #208, Garden Grove, CA 92845, USA

Telef: (714) 373-6862. Fax: (714) 891-2363

Email : sales@smxinfo.com; Web: http://www.smx.com

Núcleos de tempo real.

Nohau Corporation

51 E. Campbell Ave, Campbell , CA 95008, USA

Telef.: (408) 866-1820. Fax : (408) 378-7869.

Email: nohau@shell.portal.com ; Web: http://www.nohau.com

Emuladores.

On Time

Hofweg 49, D-22085 Hamburg

Telef: (516) 689-6654. Fax: (516) 689-1172

Email: info@on-time.com
Núcleos de tempo real.

Orion Instruments
1376 Borregas Ave, Sunnyvale , CA 94089, USA
Telef.: (408) 747-0440. Fax: (408) 747-0688.
Email: info@oritools.com; Web: http://www.oritools.com
Emuladores.

Production Lang. Corp.
200 Cochran Road, P.O. Box 109, Weatherford , TX 76048, USA
Telef.: (800) 525-6289. Fax: (817) 9-5098.
Email: PLCorp@aol.com; Web: http://www.plcorp.com
Assemblers, Compiladores de C, Debuggers e Simuladores.

QNX Software Systems Ltd.
175 Terence Matthews Crescent, Kanata, Ontario, Canada K2M 1W8
Telef: (613) 591-0931. Fax: (613) 591-3579
Email: info@qnx.com
Núcleos de tempo real.

Signum Systems
11992 Challenger Court, Moorpark , CA 93021, USA
Telef.: (805) 523-9774. Fax: (805) 523-9776.
Email: sales@signum.com; Web: http://www.signum.com
Emuladores.

Tasking
Norfolk Place 333 Elm Street, Dedham , MA 02026-4530, USA
Telef.: (617) 320-9400. Fax: (617) 320-9212.
Email: support_us@tasking.nl; Web: http://www.tasking.com
Assemblers, Debuggers, Simuladores, Monitores e Emuladores.

Universal Cross-Assemblers

9 Westminster Drive, Quispamsis , N.B. E2E 2V4, Canada

Telef.: (506) 849-8952. Fax: (506) 847-068

Email: 70730.3576@compuserve.com

Assemblers.