UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

Curso de Pós-Graduação em Engenharia Elétrica - CPGEE

PROPOSTA DE ARQUITETURA DE HARDWARE E SOFTWARE PARA SISTEMAS TEMPO-REAL DISTRIBUÍDOS

MARCELO GÖTZ

Dissertação para obtenção do título de Mestre em Engenharia Elétrica

Porto Alegre

2001

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

ESCOLA DE ENGENHARIA

Curso de Pós-Graduação em Engenharia Elétrica - CPGEE

PROPOSTA DE ARQUITETURA DE HARDWARE E SOFTWARE PARA SISTEMAS TEMPO-REAL DISTRIBUÍDOS

MARCELO GÖTZ

Engenheiro Eletricista

Dissertação apresentada ao Curso de Pós-Graduação em Engenharia Elétrica - CPGEE, como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Automação Industrial.
Desenvolvida no Laboratório Automação Industrial do Departamento de Engenharia Elétrica da Universidade Federal do Rio Grande do Sul.

Porto Alegre

2001

PROPOSTA DE ARQUITETURA DE HARDWARE E SOFTWARE PARA SISTEMAS TEMPO REAL DISTRIBUÍDOS

MARCELO GÖTZ

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador:________________________________

Prof. Carlos Eduardo Pereira - UFRGS

Dr. pela Universidade de Stuttgart - Alemanha

Banca Examinadora:

Prof. Altamiro Amadeu Suzim, UFRGS

Dr. pelo Instituto Nacional Politécnico de Grenoble – INPG –França 1981

Prof. João Cesar Netto, UFRGS

Dr. pela Universidade Católica de Louvain - Bélgica – 1995

Prof. Walter Fetter Lages, UFRGS

Dr. pelo Instituto Tecnológico de Aeronáutica – ITA – 1993

Coordenador do CPGEE:____________________________

Prof. Dr. Carlos Eduardo Pereira

Porto Alegre, fevereiro de 2002.

Dedico este trabalho a minha mãe Antônia Elvira, a minha namorada Adriana e a minha irmã Fabiana pela compreensão e apoio inestimáveis, que foram de suma importância durante o meu caminho de aprendizado e trabalho.

AGRADECIMENTOS

Quero agradecer a todas as pessoas que colaboraram de alguma forma para a realização deste trabalho e em especial ao meu orientador, Carlos Eduardo Pereira, que desde a graduação tem me proporcionado um crescimento profissional bastante sintonizado com a atualidade e também por possibilitar que eu continue os estudos em Paderborn. Agradeço também aos meus colegas de trabalho do laboratório de automação, Carlos Mitidieri, Cristiano Brudna, Leandro Becker, Ronaldo Hüsemann, Rafael Wild, João Pacheco, Rafael Zeilmann. A todos os professores relacionados ao curso de pós-graduação do CPGEE e demais pessoas não mencionadas mas que me incentivaram na realização deste trabalho. Por fim, um agradecimento a CAPES pela provisão da bolsa de mestrado.

SUMÁRIO

1 INTRODUÇÃO ... 1

1.1 MOTIVAÇÃO ... 2

1.2 OBJETIVO ... 4

1.3 ORGANIZAÇÃO DO TEXTO ... 4

2 REVISÃO TEÓRICA ... 5

2.1 SISTEMA OPERACIONAL ... 5

2.1.1 Processos ... 7

2.1.2 Escalonamento ... 9

2.1.3 Comunicação e Sincronização ... 12

2.1.4 Gerenciamento de memória ... 14

2.1.5 Sistema de arquivos ... 15

2.1.6 Sistema de entradas e saídas ... 15

2.1.7 Chamadas de sistema ... 17

2.2 SISTEMAS TEMPO REAL ... 17

2.2.1 Caracterização de um sistema operacional tempo real ... 17

2.2.2 Caracterização do desempenho de sistemas tempo-real ... 18

2.2.3 Previsibilidade do comportamento do sistema ... 19

2.2.4 Processos tempo-real ... 19

2.2.5 Processos periódicos e aperiódicos ... 20

2.2.6 Processos dinâmicos e estáticos ... 20

2.2.7 Importância relativa dos processos ... 21

2.2.8 "DeadLock" ... 21

3 TRABALHOS RELACIONADOS ... 22

3.1 ARQUITETURA MULTICONTROLADA – PONTREMOLI, M. ... 22

3.2 ARQUITETURA SPRING PARA SISTEMAS TEMPO-REAL ... 24

3.3 ARQUITETURA DE SUPORTE PARA SISTEMAS TEMPO-REAL DO TIPO "HARD" ... 26

3.4 ANÁLISE COMPARATIVA ... 28

4 TRABALHOS COMPLEMENTARES 30
4.1 EXTENSÃO TEMPO-REAL PARA LINUX 30
4.1.1 RT-Linux 31
4.1.2 RED-Linux 33
4.2 SISTEMA OPERACIONAL DE CÓDIGO FONTE ABERTO PARA SISTEMAS EMBARCADOS – BRUDNA, C. 36
5 ARQUITETURA PROPOSTA 38
5.1 VISÃO GERAL 38
5.1.1 Bloco Principal 40
5.1.2 Bloco Secundário 40
5.1.3 Particionamento do sistema 40
6 PROJETO DE HARDWARE E SOTWARE 45
6.1 PROJETO DE HARDWARE 45
6.1.1 Escolha dos microprocessadores 46
6.1.2 Bloco principal 46
6.1.3 Bloco secundário 48
6.1.4 Comunicação entre os uP 50
6.1.5 Comunicação com meio externo 52
6.2 PROJETO DE SOFTWARE 52
6.2.1 Escolha do sistema operacional 52
6.2.2 Modificações no sistema operacional 53
7 IMPLEMENTAÇÃO 60
7.1 PROPOSTA ORIGINAL 60
7.1.1 Bloco principal 61
7.1.2 Bloco secundário 65
7.2 ARQUITETURA PARA TESTES 69
7.2.1 Alternativa 1: Aplicação emulada 70
7.2.2 Alternativa 2: Núcleo emulado 71
7.2.3 Alternativa 3: Emulação Mista 71
8 RESULTADOS OBTIDOS 73
8.1 AMBIENTE DE REALIZAÇÃO DOS TESTES 73
8.2 ARQUITETURA TRADICIONAL 76
8.3 TAREFAS TEMPO-REAL 77

8.4 AVALIAÇÃO DOS RESULTADOS OBTIDOS 79
9 CONCLUSÕES E TRABALHOS FUTUROS 81
9.1 TRABALHOS FUTUROS 82

LISTA DE FIGURAS

Figura 2.1 Camadas de um sistema computacional 5
Figura 2.2 Sistema Tempo-Real Distribuído........ 6
Figura 2.3 Diagrama de estados de processos........ 9
Figura 2.4 Camadas do sistema de entrada e saídas........ 16
Figura 3.1 Arquitetura Spring 25
Figura 3.2 Arquitetura proposta em [HAL96]........ 27
Figura 4.1 Diagrama em blocos RT-Linux 31
Figura 5.1 Unidade de processamento 39
Figura 5.2 Camadas de um sistema computacional Linux........ 41
Figura 5.3 Arquitetura conceitual do Linux 42
Figura 6.1 Diagrama em blocos do Hardware proposto........ 45
Figura 6.2 Conexão da memória com a CPU principal........ 47
Figura 6.3 Conexão da memória com a CPU secundária........ 49
Figura 6.4 Conexão da memória Dupla-Porta........ 51
Figura 6.5 Diagrama com os vários subsistemas Linux........ 55
Figura 6.6 Atraso no atendimento de uma tarefa em Linux........ 56
Figura 6.7 Redução do tempo de atendimento em RED-Linux 57
Figura 7.1 Abrangência dos níveis de sistema e de usuário........ 61
Figura 7.2 Formato da mensagem de uma chamada de sistema 62
Figura 7.3 Mensagem do bloco Principal indicando fim de execução de tarefa........ 64
Figura 7.4 Mensagem do escalonador ao bloco Principal para reinicio........ 66
Figura 7.5 Código para STI na interrupção emulada 68
Figura 7.6 Hardware disponível........ 69
Figura 7.7 Diagrama em blocos da Alternativa I 70
Figura 7.8 Diagrama em blocos da Alternativa II........ 71
Figura 7.9 Diagrama em blocos da Alternativa III........ 72
Figura 8.1 Chamada de sistema para acionamento de I/O 75
Figura 8.2 Chamada de sistema “pause()”, utilizada para........ 75
Figura 8.3 Tarefa para “carregar” o sistema........ 77
Figura 8.4 Tarefa não tempo-real para acionamento cíclico........ 77
Figura 8.5 Tarefa tempo-real para acionamento cíclico........ 78
Figura 8.6 Tempos associados a ativação de tarefa........ 79

LISTA DE TABELAS

Tabela 8.1 Medidas realizadas em ambiente não Tempo-Real 76
Tabela 8.2 Medidas realizadas em ambiente Tempo-Real 78

RESUMO

Um sistema tempo-real caracteriza-se por possuir requisitos temporais para execução de suas atividades, e de acordo com a sua tolerância ao atendimento destes requisitos é classificado em *hard-real-time* ou *soft-real-time*. O presente trabalho se propõe a apresentar uma arquitetura de hardware e software para suporte a sistemas tempo-real embarcados de baixo custo com objetivo de aplicação em pesquisas no meio acadêmico e que possa ser usado até em ambientes *hard-real-time*. A motivação para este trabalho está na necessidade de incorporação de garantias temporais (determinismo) em sistemas operacionais, características estas tão necessárias para sistemas tempo-real, e que são problemáticas de serem mantidas em sistemas dinâmicos que usam arquiteturas de hardware e software convencionais. Apoiado em estudos já realizados neste sentido, esta proposta pretende suprir o suporte em hardware, usando para tal microcontroladores de 32bits com alta capacidade de processamento e um ambiente de software confiável, já conhecido, com porte para sistemas embarcados e com código fonte aberto: o uClinux, porém com modificações para a sua adaptação no hardware proposto e para enfatizar as suas características tempo-real.

ABSTRACT

Real-time systems are characterized by the fact that not only logical but also timing correctness properties have to be satisfied. Typically, a real-time system is divided into two categories: hard-real-time, if missing a deadline may lead to catastrophic consequences, and soft-real-time, if a late completion gracefully degrades the performance without causing damage. This work presents a low cost embedded hardware and software architecture to support real-time systems. While mainly intended for research purposes, the proposed architecture should provide support to the development of hard-real-time systems. The proposed architecture addresses a common problem in conventional architectures: the maintenance of a deterministic temporal behavior, essential in real-time systems, and damaged by an overload caused by operating systems activities. The proposed architecture make use of a 32bits high performance microcontroller, a reliable, popular and open source code operating system to embedded applications uClinux, and enhance these with extensions to better cope with real-time systems development.

1 INTRODUÇÃO

Um sistema computacional é definido como Tempo-real quando o seu funcionamento não depende apenas de um correto processamento das entradas para a geração das saídas, mas também que tais atividades obedeçam restrições temporais específicas. Um sistema Tempo-real diferencia-se então, dos demais, por possuir requisitos temporais específicos para o processamento de suas atividades, o que significa, não raro, uma complexidade e custo relativamente maiores comparado aos outros sistemas.

No passado recente, a grande maioria dos sistemas tempo-real eram implementados de forma estática, e o seu código era escrito em linguagens de baixo nível (como *assembly*, por exemplo), ocasionado sobretudo pela raridade de recursos de hardware e software disponíveis com as funcionalidades necessárias. Hoje, com o crescente avanço da eletrônica, informática e principalmente da microeletrônica, a utilização de sistemas tempo-real tem sido aumentada e já é possível encontrar um grande número de sistemas embarcados utilizando esta filosofia, e estes geralmente sendo utilizados em sistemas distribuídos.

Junto a esta crescente utilização e evolução dos sistemas, a complexidade destes também aumentaram, exigindo que metodologias e ferramentas no auxilio do desenvolvimento e também da manutenção de tais sistemas fossem desenvolvidas. Diante desta realidade surgiram vários segmentos de estudo, abordados pela comunidade científica, relacionados a sistemas tempo-real. Propostas de modelagem de sistemas, linguagem de programação, ferramentas de análise e sistemas operacionais, entre outros, são hoje bastante estudadas.

A maioria destas linhas de pesquisa estão tendo significativas contribuições, no que diz respeito a novos estudos, novas propostas e novos conceitos pela comunidade científica. Entretanto, focalizando-se apenas uma unidade de processamento, mais especificamente no que diz respeito a sua organização interna que provê a base necessária para o atendimento das restrições temporais em um sistema tempo-real, vê-se que esta ainda não foi totalmente tratada de forma consistente.

Hoje, percebe-se que a maioria das unidades de processamento adotam como solução em hardware as arquiteturas convencionais, mesmo para sistemas tempo-real. As adaptações realizadas para que tais arquiteturas suportem a aplicação em ambientes tempo-real, são o aumento da velocidade dos componentes, inclusão de linhas de I/O rápidas e o uso de processadores de maior capacidade. Entretanto, apesar de aumentar a velocidade de processamento dos programas sendo executados, tais arquiteturas de hardware não apresentam características essenciais aos sistemas tempo-real: previsibilidade na execução dos processos e determinismo temporal. Outros requisitos de sistemas tempo-real são atendidos via software, pela elaboração de sistemas operacionais e pelo cuidado na elaboração da programação da aplicação. Porém, com o aumento da complexidade dos sistemas tempo-real, torna-se cada vez mais difícil que os problemas identificados em tais ambientes sejam solucionados desta maneira.

1.1 MOTIVAÇÃO

A questão básica que é levada em conta em um sistema tempo-real e que o caracteriza, é o determinismo temporal. Neste caso não é apenas necessário que haja um correto processamento das informações, mas também que os resultados estejam disponíveis dentro de um determinado intervalo de tempo conhecido, após a entrada dos dados.

Pode-se então concluir que a caracterização “tempo-real” está ligada à aplicação a que este se destina. Uma determinada solução computacional que garanta a execução de um laço de controle de processos térmicos com constante de tempo da ordem de segundos, não necessariamente serve para o controle de posicionamento de um robô em linha industrial com requisitos temporais da ordem de milisegundos. Entretanto, ambos os sistemas apresentam requisitos temporais e são considerados como tempo-real, uma vez que o conceito de tempo-real diferencia-se do conceito de velocidade de processamento (ou seja, um sistema computacional rápido NÃO constitui um sistema tempo-real)

Sendo o sistema tempo-real, pode-se enquadrá-lo como: crítico (“*hard real-time*”) ou então não-crítico (“*soft real-time*”). Em um sistema não-crítico, algum possível atraso no processamento étolerável, mesmo que não seja desejável. Já em um sistema tempo-real crítico se o tempo do processamento ultrapassar o máximo previsto, os resultados podem ser catastróficos [LUN00].

Avanços em áreas como microeletrônica e informática tem disponibilizado a criação de microprocessadores e microcontroladores com crescente capacidade de processamento, a custos cada vez mais reduzidos. O número de aplicações conhecidas como embarcadas ("*embedded*"), nas quais os sistemas computacionais encontram-se "embutidos" dentro dos mais variados dispositivos (de telefones celulares a eletrodomésticos e automóveis) cresceu exponencialmente nos últimos anos e já supera o número de sistemas microprocessados presentes em computadores. A maioria destas aplicações caracterizam-se por sistemas tempo-real, uma vez que limites de tempo são normalmente impostos aos tempos de processamento envolvidos.

Aliada a esta crescente utilização de sistemas tempo-real embarcados, percebe-se uma tendência de que tais aplicações sejam distribuídas (automação residencial, automação industrial, controle de aeronaves). Tais indicativos apontam uma crescente complexidade nos sistemas utilizados, fazendo-se necessária a utilização de conceitos mais elaborados no desenvolvimento de projetos (como objetos distribuídos), o que exige também um maior suporte em nível de software. Surge daí a necessidade de utilização de unidades de controle que possibilitem o suporte a requisitos tempo-real.

As unidades de controle tradicionais, mesmo as utilizadas em sistemas tempo-real, são geralmente constituídas de um microcontrolador, memórias, dispositivos de comunicação e, associado, um sistema operacional como suporte de software. As soluções disponíveis comercialmente possuem um custo alto, além de serem “sistemas fechados”, o que impossibilita a realização de modificações, seja em hardware ou em software, para alterar certas características de funcionamento. Tais alterações são muitas vezes necessárias em ambientes de pesquisa para proporcionar a realização de estudos de caso para experimentação de idéias, integração com outras ferramentas ou comprovações de análises.

O processamento exigido nas unidades de controle pode ser classificado, de maneira simplificada, em atividades da própria aplicação e do próprio sistema operacional. Quando é utilizada somente uma CPU, estas atividades concorrem entre si para o seu uso, o que significa que a capacidade de processamento total disponível pela CPU para a aplicação (que é o objetivo final do controle) fica diminuída. A solução natural para tal problema é a adoção de um microcontrolador de maior capacidade, porém associado a isto está um custo maior do sistema. Além disto, problemas como a não garantia do determinismo temporal e o comportamento do sistema para certas aplicações começam a não ser mais garantidos, significando que para certos sistemas tempo-real, não seja mais possível a sua utilização.

Nos sistemas embarcados de maior complexidade, exposto em parágrafos anteriores, há a indicação da necessidade de utilização de unidades de processamento, que não as convencionais, para permitir um suporte tanto em nível de desenvolvimento do projeto, como o uso de recentes filosofias (objetos distribuídos), e também um suporte em hardware com soluções para os problemas anteriormente apresentados.

1.2 OBJETIVO

O objetivo do presente trabalho é propor uma arquitetura de hardware e software de baixo custo para a utilização em sistemas tempo-real distribuídos. Em nível de software pretende-se possibilitar o uso de um sistema operacional completo para sistemas embarcados, o uClinux, com características tempo-real, dando assim suporte para desenvolvimento de projeto [BRU 00].

Em nível de hardware, propõe-se uma arquitetura alternativa para sistemas tempo-real, onde principalmente ataca-se o problema da garantia de determinismo temporal (essencial para sistemas tempo-real), baseando-se em conceitos já levantados em outros trabalhos ([PON98], [HAL92] e [STA92]).

1.3 ORGANIZAÇÃO DO TEXTO

O capítulo 2 apresenta uma revisão dos conceitos utilizados, tais como sistemas operacionais e sistemas tempo-real. Tal discussão visa facilitar a compreensão do texto da dissertação. No capitulo 3 são discutidas algumas propostas relacionadas a este trabalho, as quais apresentam objetivos semelhantes. Isto serve de comparação entre o presente trabalho e os atualmente apresentados na comunidade científica. O capítulo 4 apresenta duas versões de extensões para tempo-real do sistema operacional Linux, sendo que suas soluções são discutidas a fim de trazê-las para serem implementadas na versão para sistemas embarcados, o uClinux. O capítulo 5 apresenta efetivamente a arquitetura proposta nesta dissertação, descrevendo os seus blocos principais. O capítulo 6 apresenta o projeto de hardware e software da arquitetura apresentada. No capítulo 7 é mostrada a implementação do sistema operacional. O capítulo 8 apresenta os testes realizados com o hardware disponível e suas análises. E no capítulo 9 são realizadas conclusões a respeito do trabalho e indicados os futuros trabalhos para a continuidade desta pesquisa.

2 REVISÃO TEÓRICA

2.1 SISTEMA OPERACIONAL

Um sistema computacional, seja ele de grande/médio porte ou então sistema embarcado, pode ser dividido em três partes: a parte física (hardware), o sistema operacional e os programas da aplicação (os quais permitem a interação com os usuários). O sistema operacional é um programa que atua como intermediário entre a aplicação e o hardware. É ele que disponibiliza os recursos de hardware, tais como unidades de disco e portas de entrada e saída, aos programas aplicativos executando. Através dele é possível que o usuário desenvolva a sua aplicação de maneira eficiente e conveniente, o que torna o sistema útil, do ponto de vista prático.

Figura 2.1 Camadas de um sistema computacional

O sistema operacional pode ser visto como uma extensão virtual do hardware disponível, e é responsável pelo gerenciamento de seus recursos. Assim, para os desenvolvedores de software, o acesso direto aos dispositivos de entrada e saída, memória e CPU é feita de maneira transparente e com maior facilidade, uma vez que a complexidade está embutida no sistema operacional.

Nos computadores modernos adota-se o conceito de multiprogramação, que significa que a CPU está executando diversos programas simultaneamente. Neste caso, a simultaneidade não se refere a executar no mesmo instante de tempo (o que exigiria várias CPUs), significa que o usuário o percebe desta maneira. Fisicamente a CPU está alternando entre os diferentes programas para executá-los em frações relativamente pequenas de tempo.

Isto permite um maior aproveitamento dos recursos. Por exemplo, durante o tempo em que um programa espera pela leitura de dados de um dispositivo externo, outro programa pode usar a CPU para realização de suas operações.

Entre os tipos de sistemas operacionais encontram-se os de tempo-real. Estes são usados quando há a necessidade de um processamento dos dados pelo programa em intervalos de tempos específicos, pré-determinados. Além destes dados serem processados corretamente, estes devem ser feitos respeitando requisitos temporais definidos.

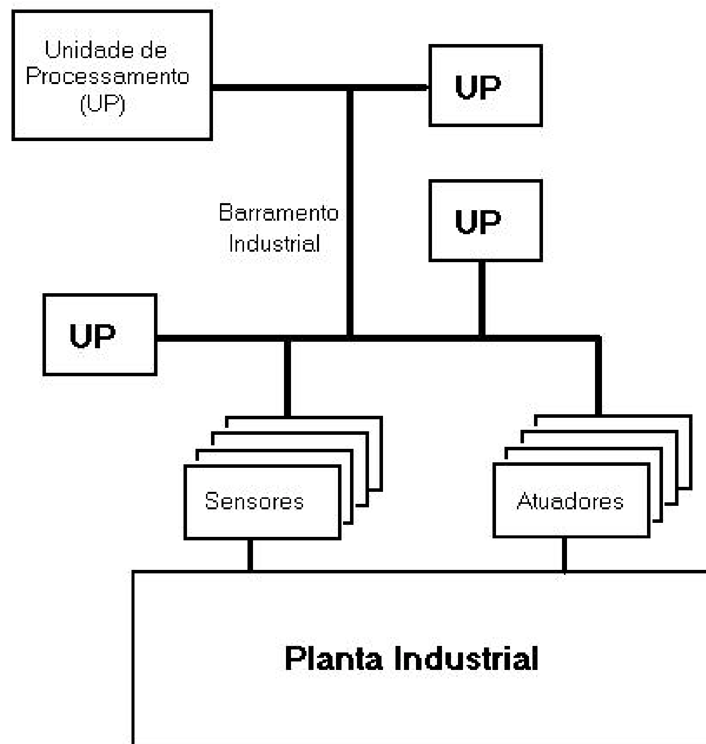


Figura 2.2 Sistema Tempo-Real Distribuído

Atualmente uma outra linha de sistemas tem sido alvo de estudos na comunidade científica: os sistemas distribuídos. Tais sistemas caracterizam-se pela existência de diversas CPUs, interligadas via canais de comunicação, nas quais as aplicações são executadas. Cada unidade possui a sua CPU, memória, *clock*, entradas e saídas, e comunica-se com as outras através de algum canal de comunicação, como, por exemplo, um barramento industrial.

Cada unidade de processamento em um sistema distribuído pode variar de tamanho e funcionalidade. Ela pode incluir pequenos microcontroladores (sistemas embarcados), microcomputadores, estações de trabalho (*workstations*) e outros.

Há uma variedade de razões para se construir um sistema distribuído, e as principais são [KIR 88]:

- Crescimento incremental: um sistema distribuído apresenta facilidades de expansão, ao contrário de outros sistemas que não permitam expansão, apenas por repetição.
- Confiabilidade: sistemas distribuídos podem ser potencialmente mais confiáveis devido à multiplicidade e a um certo grau de autonomia de suas partes. Esta pode ser conseguida através de uma correta distribuição lógica, que é mais importante do que a distribuição física, que está mais ligada a questões de desempenho, tempo de resposta, organização do sistema e principalmente à possibilidade de ter-se controle explícito sobre o processamento e informações locais.
- Estrutura: Sistemas distribuídos podem refletir a estrutura organizacional à qual eles servem;
- Proteção: sistemas distribuídos podem oferecer mais segurança do que sistemas centralizados. A segurança resulta muito mais da distribuição lógica do sistema do que a distribuição física.

Como este trabalho possui o foco voltado para uma unidade de processamento, cabe aqui realizar algumas revisões de algumas partes funcionais que fazem parte do sistema operacional geralmente encontrado nestes.

2.1.1 Processos

Para facilitar a implementação de uma aplicação, em nível de software, esta é dividida em vários conjuntos de pequenos programas chamados de processos ou tarefas. Podemos pensar em um processo como sendo um programa que geralmente é criado com a finalidade de realizar o processamento de dados referentes a uma mesma funcionalidade.

Imaginando-se um sistema que deva controlar o nível de um tanque de armazenagem de água, de maneira simplista pode-se definir alguns processos:

- O algoritmo de controle do nível do tanque, por exemplo, um PID.
- A leitura de um ou mais sensores para a mensuração do nível e vazão.
- Controle de um atuador para a entrada de água.

- Uma interface com o usuário, para que se possa interagir com o sistema, definindo o nível desejado e visualização dos valores atuais.

Neste ponto pode-se perceber que se uma unidade de processamento com apenas uma unidade de processamento for usada para tal finalidade, nela tem-se um sistema operacional onde mais de um processo deve ser executado. E esta situação é a que ocorre com a maior freqüência. Cada tarefa compartilhará, com as demais, os recursos: CPU, memória e os dispositivos de entrada e saída.

Temos então o que chamamos na literatura de um processamento multitarefa. Na realidade, quando há somente uma CPU para a execução de todos os processos, existe um pseudoparalelismo (paralelismo aparente). Isto é enfatizado quando cada processo ocupa a CPU durante frações de tempo de dezenas ou centenas de milisegundos, pois duas ou mais tarefas não podem estar sendo executadas ao mesmo tempo. Sendo assim, o sistema operacional deve ser capaz de reservar espaços de tempo para cada processo e agendar a seqüência de execução destes "pedaços" de processos.

Em alguns sistemas operacionais, chamado de preemptivos, o processo em execução pode ser interrompido pelo sistema operacional, a fim de permitir que outro processo de maior importância ou prioridade possa executar e passe a ocupar o processador. Quando o processo de maior prioridade termina sua execução ou libera a CPU por algum motivo, o processo que se encontrava executando deve retomar sua execução. Em sistemas operacionais ditos não preemptivos, o próprio processo pode liberar a CPU para que outro processo a ocupe, sendo assim também chamados de processos cooperativos. Para que isto seja possível deve-se armazenar todas as informações pertinentes ao processo que estava executando, tais como os valores de todas as suas variáveis e o ponto exato em que o programa estava sendo executado. Estas informações constituem o que se chama de contexto de um processo, e a troca de um processo por outro é denominada de chaveamento de contexto ("*context switching*"). Entre outras informações que fazem parte do contexto de cada tarefa, tem-se:

- Identificador do processo
- Estado da tarefa
- Estado do hardware (registros da CPU, *flags*)
- Informações do escalonador

- Informações para o gerenciamento da memória

Entre estes, o estado da tarefa é definido pelo sistema operacional para controlar a sua execução (da tarefa). Na Figura 2.3 vê-se um diagrama de estados para um processo. O nome de cada estado é arbitrário, mas usado em vários sistemas operacionais. Também pode haver diferença no detalhamento destes processos para cada sistema.

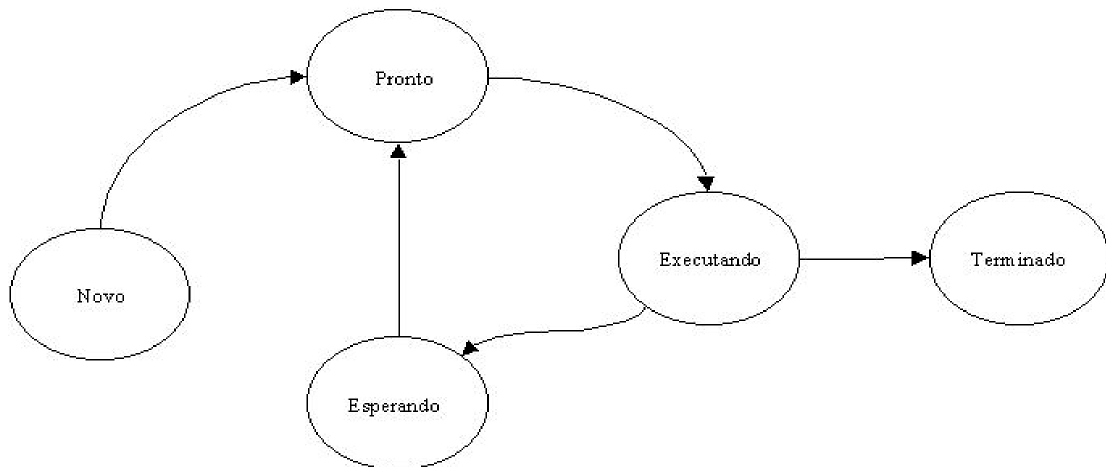


Figura 2.3 Diagrama de estados de processos

- **Novo**: O processo foi criado
- **Executando**: As instruções estão sendo executadas. Ocorre quando o processo está ocupando a CPU
- **Esperando**: O processo está esperando pelo acontecimento de algum evento, o que pode ser o acesso a algum dispositivo externo, por exemplo.
- **Pronto**: O processo está pronto para ser executado e esperando pela CPU para executar as suas instruções.
- **Terminado**: O processo foi terminado, ou terminou a si próprio

2.1.2 Escalonamento

Para que todas as tarefas ocupem a CPU de maneira ordenada e também para objetivar o seu máximo aproveitamento, existe um gerenciamento feito pelo sistema operacional para determinar a ordem segundo a qual os processos que ocuparão o processador no tempo. Para este gerenciamento são executados algoritmos que definem a seqüência de execução das tarefas, os quais são chamados de algoritmos de escalonamento. O módulo do sistema operacional que executa estes algoritmos normalmente é denominado de escalonador.

Para a escolha da tarefa que ocupará a CPU no próximo instante (dentre aquelas que estão prontas para executar, ou seja, encontram-se no estado de "pronta"), o escalonador baseia-se normalmente no conceito de prioridades, as quais relacionam-se com a importância de uma dada tarefa ou com a urgência em que a mesma deve ser executada.

Os algoritmos de escalonamento são classificados de acordo com algumas características de funcionamento. Em relação à preempção das tarefas:

- Algoritmo de escalonamento preemptivo: no momento em que uma tarefa pronta para a execução possui prioridade mais alta da que está sendo executada, o escalonador interrompe a tarefa em execução, salva o seu contexto, e coloca a tarefa de mais alta prioridade em execução.
- Algoritmo de escalonamento não preemptivo: Um tarefa pronta para execução, que tenha uma prioridade maior da que está sendo executada, só poderá entrar em execução quando a tarefa em processamento, por algum motivo, liberar o processador.

Em relação ao tempo de execução do algoritmo tem-se:

- Escalonamento "*off-line*": O algoritmo de escalonamento é executado antes de que o sistema entre em operação. Isto exige que se saiba previamente os tempos de ativação de cada tarefa existente no sistema. É possível se executar algoritmos bastante sofisticados, porém o sistema se mostra bastante inflexível.
- Escalonamento "*on-line*": O algoritmo de escalonamento é executado em tempo de execução, a cada vez que uma tarefa entra ou sai do sistema. Desta maneira é possível implementar sistemas bastante flexíveis, porém acarretando geralmente em uma grande sobrecarga computacional para a execução do algoritmo.

Quanto as prioridades atribuídas às tarefas do sistema, estas também podem possuir características estáticas ou dinâmicas.

- Prioridade fixa: Cada tarefa recebe uma prioridade quando da sua criação, a qual se mantém constante durante todo o ciclo de vida desta.

- Prioridade dinâmica: É utilizada em algoritmos de escalonamento mais elaborados. A prioridade da tarefa atribuída na sua criação pode sofrer alterações durante a evolução do sistema.

Há vários estudos na comunidade científica a respeito de algoritmos de escalonamento, existindo uma variedade muito grande de algoritmos para tal. Entre os mais utilizados pode-se citar:

- FIFO: A primeira tarefa a entrar na fila das tarefas prontas, será executada;
- Round-Robin: Projetado para sistemas de tempo compartilhado. É determinado um intervalo de tempo fixo, em que a cada intervalo, um processo é executado (estando este na fila de tarefas prontas para execução);
- Taxa monotônica (ou "*Rate Monotonic*" - RM): É um caso específico da prioridade fixa, em que a tarefa com o menor tempo de execução recebe a maior prioridade;
- *Deadline* mais próximo primeiro (*Earliest Deadline First* – EDF): Atribui prioridade maior para aquelas tarefas que possuem o *deadline* mais próximo no instante atual. Opera com prioridade dinâmica;
- *Time-sharing*: Possui o objetivo de proporcionar um tempo médio para a execução de cada processo, para que todos tenham o mesmo acesso à CPU e aos recursos.

O algoritmo de escalonamento pode ser considerado um processo, porém do próprio sistema operacional, que será executado com uma freqüência maior do que qualquer outro. Vale salientar que ele também ocupará recursos como memória e tempo de processamento da CPU.

Geralmente neste ponto há um comprometimento entre a qualidade do resultado gerado pelo algoritmo de escalonamento e o tempo de execução do mesmo, o que é conhecido como *overhead*, uma vez que tempo de processamento da CPU é usado pelo sistema operacional para a execução do escalonador, reduzindo assim o tempo em que a CPU fica disponível para tarefas da aplicação. Normalmente tem-se um problema de otimização com características conflitantes: para que se maximize a utilização da CPU pelos processos, necessita-se de um algoritmo eficiente, o que geralmente implica em um algoritmo de

escalonamento mais elaborado e que exige um maior tempo na execução. Isso significa que o tempo total de uso da CPU disponível para os processos será menor.

Alguns critérios, entre outros, são utilizados para avaliação do algoritmo de escalonamento:

- Utilização da CPU: é o percentual do tempo em que CPU está ocupada usando processos da aplicação. Por exemplo, o valor de 80% significa que a CPU esta sendo utilizada 80% do tempo por processos da aplicação e 20% com outros processos (por exemplo, processos do sistema operacional como o escalonador);
- Taxa de saída: Número de processos executados (terminados) por unidade de tempo;
- Tempo total de execução: Tempo de execução do processo. Esta medida leva em conta o tempo que o processo ficou na fila de espera de tarefas prontas para ser executada, e o próprio tempo de execução deste;
- Tempo de espera: O total de tempo em que o processo permaneceu na fila de espera das tarefas prontas;
- Tempo de resposta: total de tempo desde que os dados de entrada do processo estão prontos até o momento da geração dos dados de saída gerados pelo referido processo;
- Tempo de reação: total de tempo desde que algum evento (externo ou interno) seja sinalizado e o instante de ativação do processo correspondente. Para sistemas tempo-real, este tempo é conhecido como tempo de latência.

2.1.3 Comunicação e Sincronização

Apesar de um sistema ser composto por vários processos, cada um com a sua funcionalidade, estes dificilmente funcionam isoladamente. Não raro um processo necessita comunicar-se com outros processos e/ou sincronizar a sua atividade com os demais. Ou seja, mesmo processos concorrentes tendem a ser cooperativos exigindo mecanismos que permitam a comunicação e sincronização.

Deve haver então uma sincronização das atividades entre os processos pois os dados processados por uma tarefa são usados por outra. Para tanto, existem vários métodos

para a comunicação entre os processos. Todos estes mecanismos devem ser disponibilizados pelo sistema operacional.

A comunicação pode ser feita por mensagens enviadas de um processo para outro, através de sinais, memória compartilhada ou outro recurso semelhante. Este serviço de comunicação entre processos é de responsabilidade do sistema operacional. Ou seja, cada sistema operacional provê uma ou mais maneiras de sincronização/comunicação.

Um exemplo de sincronização se dá quando uma tarefa é colocada no estado de espera, enquanto aguarda que uma certa condição permita que esta continue. Esta condição pode ser a ativação de algum sinal de entrada (evento físico) ou mesmo algum sinal de outra tarefa.

Outro problema abordado pela sincronização de processos é o acesso às seções críticas. Quando temos vários processos concorrendo no tempo pelos recursos de hardware, podem existir acesso a informações compartilhadas (tais como variáveis, arquivos, tabelas, etc.) sendo que em especial as operações de escrita necessitam ser sincronizadas para garantir-se a consistência da informação. Uma vez que o escalonador pode executar o chaveamento de contexto entre dois processos, deve-se prover maneiras de sinalização de que um recurso está bloqueado e não disponível naquele instante.

Mecanismos bastante conhecidos em sistemas operacionais que permitem realizar as atividades de sincronização e comunicação são:

- *Flags* de eventos: usados para sinalizar a ocorrência de eventos, sejam externos ou internos (de tarefa para tarefa);
- Filas de mensagens e *Maibloxes*: usadas na comunicação e troca de dados entre as tarefas;
- Semáforos: Gerência o acesso a recursos de hardware ou dados compartilhados entre as tarefas.

Um cuidado especial deve ser tomado com relação à possibilidade de ocorrência de "*deadlocks*", onde uma combinação nas dependências de uso/liberação dos recursos pode levar à paralisação de todos os processos (por exemplo, um processo A utilizando um recurso 1 e bloqueado por estar tentando usar um recurso 2 o qual está bloqueado por um processo B que encontra-se bloqueado aguardando a liberação do recurso 1 por A). Um exemplo clássico

desta situação envolve dois processos, chamados de P1 e P2, e dois recursos, chamados de R1 e R2, da seguinte maneira [TOS 00]:

P1: ...requisita (R1); ...requisita(R2); ...libera(R1 e R2); ...

P2: ...requisita (R2); ...requisita(R1); ...libera(R1 e R2); ...

A situação de "*deadlock*" acontece se os dois processos adquirem o primeiro recurso de que necessitam. Neste caso, cada processo se bloqueia quando requisita o segundo recurso. Os dois processos ficam eternamente bloqueados, cada um esperando que o outro libere o seu recurso. Como nenhum processo pode ser executado (nenhum deles pode ir para a lista de tarefas prontas) esta espera se prolonga para todo o sempre.

Esta situação é especialmente crítica em sistemas tempo-real, pois causam resultados catastróficos.

2.1.4 Gerenciamento de memória

Define-se como unidade de gerenciamento de memória a parte do sistema operacional cuja função é controlar quais partes da memória estão em uso e quais não estão, de forma a alocar memória a processos quando estes precisarem, liberar a memória que estava sendo usada por um processo que terminou e assim gerenciar o uso da memória física do sistema [TAN 92].

Em um sistema computacional dois tipos de memórias podem ser identificadas:

- Memória principal: aquela acessada diretamente pela CPU, a qual tem-se uma maior velocidade no acesso por esta;
- Memória secundária: discos rígidos e outras mídias magnéticas ou óticas utilizadas no armazenamento de dados.

Por sua vez, a memória principal pode ainda ser sub-dividida em:

- Memória do usuário: usada pelos processos da aplicação;
- Memória do sistema operacional: usada pelas atividades relacionadas ao sistema operacional.

A memória do usuário é gerenciada de maneira a alocar o espaço necessário para cada um destes, pois cada processo possui a sua pilha, registros da CPU, variáveis locais, *flags*, e todos os dados relacionados exclusivamente ao processo. Esta classificação dos

diferentes tipos de memórias em relação ao seu uso será usada neste trabalho para otimização destes módulos na arquitetura proposta.

2.1.5 Sistema de arquivos

Para muitos usuários, o sistema de arquivos é o aspecto mais visível de um sistema operacional. Os arquivos podem conter programas binários para serem executados pela CPU, dados diversos e até um conjunto de comandos para o sistema operacional.

A parte do sistema operacional responsável pelo tratamento dos arquivos é denominada **sistema de arquivos**. Este provê uma forma padrão de acesso a dados que podem estar em diversos tipos de mídia, como por exemplo, fitas magnéticas, discos óticos, discos magnéticos e etc.

Assim, o sistema de arquivos deve permitir aos processos, realizar operações com os arquivos. Abaixo alguns exemplos:

- Criação de arquivos;
- Escrita em arquivos;
- Leitura de arquivos;
- Exclusão de arquivos.

A maneira de acessar arquivos é única para cada sistema operacional.

2.1.6 Sistema de entradas e saídas

Um dos principais objetivos desta parte do sistema operacional é a de prover uma maneira simples e uniforme, tanto quanto possível, de acesso aos dispositivos de entrada e saída para o resto do sistema. Uma vez que estes dispositivos variam significativamente na sua funcionalidade e velocidade de acesso (considere um mouse, disco rígido, CD-ROM como exemplo), uma variedade de métodos é necessária para o controle destes. Estes métodos formam o sistema de entradas e saídas de um sistema operacional.

O elemento de hardware básico de entrada e saída, como portas, barramentos e controladores de dispositivos acomodam uma grande variedade de dispositivos de entrada e saída. Com a finalidade de encapsular os detalhes de diferentes dispositivos, o núcleo de um sistema operacional é estruturado com módulos de *driver* para dispositivos.

O *driver* para um dispositivo permite uma maneira uniforme de acesso aos dispositivos de entrada e saída. Muitas chamadas de sistemas oferecem uma maneira padrão de integração destes entre a aplicação e o sistema operacional.

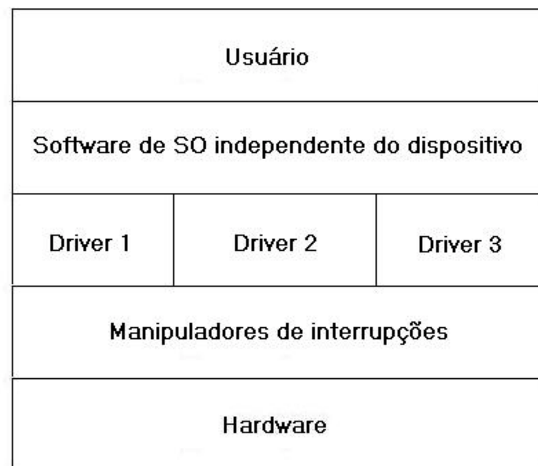


Figura 2.4 Camadas do sistema de entrada e saídas

Para que isto seja possível o software de entrada saída pode ser estruturado em quatro níveis:

- Manipuladores de interrupções: Uma interrupção de hardware indica que um dispositivo de entrada/saída teve o seu estado alterado. Por exemplo, se um processo espera por um dado de um canal serial, este ficará em estado de espera até que o dado chegue (que será sinalizado por uma interrupção). Quando isto ocorrer a rotina de tratamento da interrupção fará com que o processo que espera pelo dado seja desbloqueado;
- *Drivers* de dispositivos: Todo o código que depende do dispositivo está no driver correspondente. Quando é feita a requisição de escrita de um dado em um canal serial, por exemplo, o driver relacionado é a única parte do sistema operacional que conhece os registradores do dispositivo responsável pelo envio do dado;
- Software do sistema operacional independente do dispositivo: o objetivo básico deste software é a realização de funções de entrada e saída que são comuns a todos os dispositivos, e fornece uma interface uniforme para o nível do software do usuário;

- Software do nível de usuário: é o conjunto de rotinas de biblioteca e chamadas de sistema e que são usadas no código da aplicação. Estas realizam a ligação da camada de usuário com software independente do dispositivo.

2.1.7 Chamadas de sistema

Chamadas de sistema permitem estabelecer uma comunicação da camada de aplicação (os processos) com o sistema operacional. Os programas de usuário solicitam serviços do sistema operacional através de chamadas de sistema. Estes serviços podem ser, na maioria das vezes, classificados em cinco categorias:

- Controle de processos;
- Manipulação de arquivos;
- Manipulação de dispositivos de entrada/saída;
- Informações de manutenção (dados do sistema, etc..);
- Comunicação.

O número e a variedade de chamadas ao sistema operacional é diferente para cada sistema.

Quando a chamada de sistema é executada, o controle é passado para o nível do sistema operacional. Quando este terminar a atividade relacionada a respectiva chamada, o controle é retornado ao processo que originou a chamada.

2.2 SISTEMAS TEMPO REAL

Para melhor entendimento de um sistema tempo-real, as seções seguintes trazem as características e propriedades de um sistema deste tipo.

2.2.1 Caracterização de um sistema operacional tempo real

Para que um sistema operacional possa ser aplicado em algum sistema de controle tempo-real, este deve satisfazer algumas necessidades, principalmente no que se refere a temporização e previsibilidade do seu comportamento.

Algumas destas características devem estar presentes em um sistema operacional para que ele possa ser definido com tempo-real. Porém, além destes itens, a própria aplicação a que este se destina é que determinará se será ou não tempo-real [STA88].

- O sistema operacional deve ser multitarefa e preemptivo, para permitir que tarefas com maior prioridade sejam executadas primeiramente pelo sistema, interrompendo a execução de outras com menor prioridade, e desta maneira poder atender aos requisitos temporais de cada tarefa;
- O sistema operacional deve oferecer mecanismos de sincronização e comunicação entre as tarefas, através dos meios já mencionados: eventos, sinais, *mailboxes* e semáforos. Além disto, estes mecanismos devem ter também características temporais e comportamentos conhecidos;
- Um serviço de escalonamento para tarefas periódicas, eventos assíncronos, com atribuição de prioridades a tarefas, deve existir;
- O comportamento de qualquer subparte do sistema operacional (chamadas de sistema, atendimento à interrupções, chaveamento de contexto, escalonamento e comunicação) deve ser conhecido e previsível, sobretudo em seu tempo de execução.

Em muitos sistemas operacionais tempo-real, estas características são projetadas para serem rápidas, embora a rapidez seja relativa e sozinha não caracterize um sistema operacional como tempo-real [STA88].

Muitos sistemas operacionais tempo-real têm estendidas as suas características de temporização e previsibilidade para a sua utilização em sistemas distribuídos. Isto significa que a possibilidade de comunicação entre as várias unidades de processamento distribuídas deve ser possível. A comunicação ponto-a-ponto entre as tarefas distribuídas deverá, neste caso, ser previsível e ter requisitos temporais.

2.2.2 Caracterização do desempenho de sistemas tempo-real

Alguns conceitos são muito utilizados em sistemas operacionais tempo-real. E que geralmente são utilizados na avaliação e caracterização destes sistemas.

- Latência da interrupção (tempo entre a chegada da interrupção e a ativação da tarefa): Este valor deve estar associado às requisições da aplicação e deve ser previsível. Este também depende do número de interrupções simultâneas pendentes;

- Para cada chamada de sistema, o seu máximo tempo de execução deve ser conhecido. Este deve ser previsível, e independente da carga do sistema;
- Deve ser conhecido o tempo máximo que o sistema operacional e os *drivers* mascaram as interrupções.

2.2.3 Previsibilidade do comportamento do sistema

O tempo de processamento gasto pelo sistema deve ser previsível, especialmente para processos periódicos e críticos. Isto significa que deve ser possível determinar o tempo gasto para processar os dados de entrada e a geração da saída no pior caso.

Isto fica claro quando, por exemplo, tem-se o controle simultâneo de vários servo-motores dos braços de um robô que realiza a atividade de retirar as peças de uma esteira. Se o braço se movimentar muito rapidamente, haverá momentos em que não haverá mais peças na esteira. E, se este se movimentar muito lentamente, as peças se acumularão na esteira. Deve ser totalmente previsível, neste caso, os tempos gasto para que cada uma das unidades receba o comando de posicionamento, e o realize no servo-motor.

A previsibilidade de um sistema é tanto mais difícil de ser determinada quanto mais complexo for este. Em sistemas tempo-real estáticos, pode ser previsível o seu comportamento durante vários tempos de execução, porém para sistemas dinâmicos, necessita-se fazer o uso de avaliações estocásticas.

2.2.4 Processos tempo-real

As tarefas tempo-real são caracterizadas por possuírem tempos de execução, de ativação bem específicos. Porém, para se conhecer tais valores, métodos estocásticos são usados, e geralmente os piores casos são levados em conta. Além destas especificações temporais, as tarefas tempo-real podem possuir outras necessidades e requisitos.

- Necessidades de recursos: Uma tarefa pode requisitar certos recursos além da CPU, como dispositivos de entrada e saída, base de dados, etc.;
- Precedência entre as tarefas: Quando uma tarefa for muito complexa (quando por exemplo, necessitar vários recursos do sistema), esta pode ser repartida em subtarefas, cada qual responsável por um conjunto de recursos;

- Concorrência: As tarefas podem usar recursos compartilhados, e duas ou mais tarefas podem concorrer por estes no tempo. A consistência nos resultados deve ser obtida.

Com o modelo abstrato do sistema tempo-real, podemos naturalmente pensar na aplicação decomposta em processos quando na sua implementação. Estes podem ser distribuídos (alocados em duas ou mais unidades de processamento) ou pertencerem a apenas uma unidade de processamento concorrendo para o processamento pela CPU. Podemos caracterizar alguns tipos de processos tempo-real como segue abaixo.

2.2.5 Processos periódicos e aperiódicos

No universo das tarefas tempo-real, pode-se caracterizar um processo pela sua periodicidade, aos quais pode-se dizer [LAW92]:

- **processos periódicos:** são processos executados repetidamente, em períodos regulares de tempo;
- **processos aperiódicos**: são processos executados em pontos não determinados do tempo.

Processos periódicos geralmente são usados quando na amostragem de dados e/ou controles regulares, por exemplo. Já um processo aperiódico, geralmente está associado a ocorrência de algum evento, que pode ser algum sinal de entrada ou então uma condição de execução de outro processo.

Sob ponto de vista do escalonamento de tarefas, os processos aperiódicos aumentam a sua complexidade. Porém, se for possível determinar um tempo mínimo entre a ocorrência de dois processos deste tipo, a computação pode ser simplificada, uma vez que pode-se considerar este tempo como o período deste processo (representando uma situação de "pior caso", ou seja, com a maior freqüência). Os processos aperiódicos para os quais pode-se definir um período mínimo de distância entre ocorrências dos processos são chamados de esporádicos.

2.2.6 Processos dinâmicos e estáticos

Os processos podem existir permanentemente durante a execução em um sistema tempo-real, ou então ele pode ser criado, existir e ser terminado dinamicamente. Tem-se então a distinção de dois tipos:

- **processos estáticos**, existem deste a inicialização do sistema e permanecem ativos durante todo o funcionamento deste;
- **processos dinâmicos**, são criados durante a operação do sistema e ainda podem ser terminados mesmo com o sistema em operação.

A possibilidade de termos processos dinâmicos torna o projeto do sistema muito mais flexível. Por outro lado, se estará aumentando a complexidade do sistema.

É mais difícil determinar a previsibilidade com relação aos requisitos temporais em sistemas tempo-real dinâmicos (os que suportam processos dinâmicos). Usualmente métodos estatísticos são utilizados para a especificação destes requisitos.

2.2.7 Importância relativa dos processos

Como definido anteriormente, pode-se também atribuir categorias aos processos em relação a sua periodicidade, de acordo com a sua importância relativa, como foi sugerido em [RAM89]:

- **processos críticos**: os requisitos temporais devem ser obedecidos, com pena de os resultados serem desastrosos;
- **processos essenciais**: se o “*deadline*” for ultrapassado não significa em algum desastre, apesar de prejudicar o funcionamento do sistema;
- **processos não essenciais**: em que a não observância dos tempos não resulta em nenhum efeito no sistema em um futuro próximo, mas o seu prolongamento pode vir a gerar problemas.

2.2.8 “DeadLock”

Conforme descrito na seção “2.1.3 Comunicação e Sincronização”, situações de “*deadlock*” são extremamente indesejáveis e devem ser evitadas. Tais situações comprometem o funcionamento do sistema e resultados catastróficos podem acontecer, especialmente se relacionados a processos críticos. É necessário então uma analise do sistema tempo-real para verificação de potenciais “*deadlocks*”.

3 TRABALHOS RELACIONADOS

É possível encontrar na literatura algumas propostas com objetivos semelhantes às apresentadas neste trabalho. São arquiteturas de software e hardware que abordam os problemas de garantia de determinismo temporal e do "*overhead*" causado pelo processamento das atividades do sistema operacional. Em todos estes trabalhos encontra-se a discussão de que arquiteturas convencionais de hardware já não são mais soluções adequadas para a complexidade exigida nos sistemas tempo-real que estão surgindo.

Em todos é feita a adoção de mais de um processador para uma unidade de processamento, permitindo a divisão da carga computacional entre, basicamente, atividades da aplicação e atividades do sistema operacional. Cada qual traz também a proposta de um suporte em nível de software para uso desta plataforma.

3.1 ARQUITETURA MULTICONTROLADA – PONTREMOLI, M.

Este trabalho é resultado de um dissertação de mestrado [PON98], a qual apresenta uma arquitetura desenvolvida com componentes de baixo custo, baseada em microcontroladores de 8 bits. A arquitetura proposta tem o objetivo de proporcionar uma unidade de processamento a ser utilizada no desenvolvimento de aplicações com requisitos tempo-real para sistemas de controle distribuídos, com alta garantia de determinismo temporal

Nesta arquitetura são utilizados três microcontroladores responsáveis basicamente por:

- Processador principal: Gerenciamento dos temporizadores, interrupções, escalonamento;
- Processador de comunicação: Responsável pela comunicação das tarefas com o meio externo, utilizando-se um protocolo industrial, e pela comunicação entre as tarefas internas à mesma unidade de processamento;
- Processador de execução: Responsável pela execução das tarefas da aplicação.

O estudo realizado neste trabalho identificou três grandes grupos funcionais distintos, necessários para uma unidade de processamento a serem utilizadas em sistemas distribuídos:

- Funções destinadas à troca de mensagens entre tarefas, sejam elas pertencentes à mesma unidade ou não (bloco de comunicação);
- Funções destinadas ao gerenciamento das tarefas da aplicação, como o escalonamento (bloco gerencial);
- Tarefas da aplicação (bloco principal).

Para cada grupo foi então designado um processador, visando buscar o determinismo temporal em cada grupo. Assim, os três grupos podem executar as suas atividades em paralelo, e principalmente, as tarefas da aplicação ficam separadas da carga proporcionada pelo sistema operacional.

A arquitetura foi prototipada utilizando-se três microcontroladores AT89C52. A comunicação entre os blocos principal e de comunicação foi implementada de maneira bidirecional utilizando-se duas memórias “FIFO”. Entre os blocos gerencial e principal é disponibilizado um canal de comunicação serial síncrono. E para os blocos de comunicação e gerencial, é feita a comunicação através de pinos específicos para este fim.

Uma das vantagens desta arquitetura, e que também foi objetivada no desenvolvimento, é a de permitir que vários algoritmos de escalonamento sejam facilmente implementados e testados, uma vez que este encontra-se isolado (fisicamente) da aplicação.

Apesar de possuir uma boa arquitetura de hardware, e que apresentou resultados bastante satisfatórios para os seus objetivos, não há suporte em nível de software suficiente para que as implementações sejam feitas de maneira mais integrada ao hardware, dificultando a sua utilização prática.

Com as conclusões tiradas deste trabalho, e como solução às restrições levantadas em nível de software, a arquitetura apresentada na presente dissertação propõe em nível de software o uso de um sistema operacional completo, do qual se possui ampla documentação e ambientes de desenvolvimento prontos, possibilitando assim que trabalhos já desenvolvidos para este sistema possam ser aproveitados em grande parte.

Como proposta em nível de hardware, a arquitetura traz a solução de uso de microcontroladores de 32bits, que suportem o sistema operacional pretendido e que ofereçam

uma maior capacidade de processamento. Também foi dada atenção na elaboração da arquitetura interna para evitar o uso de canais de comunicação serial, entre os blocos internos, que foi identificado em [PON98] como um dos pontos de "estrangulamento" da capacidade funcional da unidade.

3.2 ARQUITETURA SPRING PARA SISTEMAS TEMPO-REAL

A arquitetura apresentada em [NIE92], denominada de Spring pelos idealizadores, é uma coleção altamente integrada de hardware e software para construção de complexos sistemas tempo-real. Foi desenvolvida para ser usada em sistemas hard-real-time, com alta performance e previsibilidade.

Já em desenvolvimento há mais de 10 anos, uma das mais importantes características do sistema Spring é o particionamento funcional, através do qual é realçada a previsibilidade do sistema pelo isolamento das tarefas da aplicação das interrupções externas. Isto é conseguido pelo uso de processadores distintos para a aplicação e para o resto do sistema.

Designado para ser utilizado também em sistemas distribuídos, cada nó de uma rede Spring, ou seja, cada unidade de processamento é composta atualmente [STA97] por 5 processadores:

- Um processador de sistema (SP) responsável por funções administrativas;
- Três processadores para aplicação (AP) que suportam as tarefas da aplicação;
- Um processador para gerenciar uma placa de I/Os.

Para cada unidade também está incluída uma placa de gerenciamento global de memória, que não está associada a nenhum processador. Cada SP e AP é constituído de uma CPU 68020 da Motorola, uma unidade de gerenciamento de memória (MMU) 68851, uma unidade de ponto flutuante (FPU) 68881, e 4Mb de memória local. Para a comunicação entre os processadores da mesma unidade, está disponível um barramento VME.

Cada nó possui 2 canais de comunicação com o meio externo; Um canal *Ethernet* para suporte à comunicação não tempo-real e para permitir o *downloading* da plataforma de desenvolvimento, e um canal ótico para realizar uma conexão em anel com outras unidades de processamento e proporcionar uma memória reflexiva de 2Mb para cada nó. Reflexiva pois qualquer nó que modificar algum dado na sua área de memória (reflexiva), reflete na

modificação do mesmo dado na mesma região de memória dos outros nós. Isto significa um modelo de memória compartilhada de 2Mb (fisicamente distribuída mas logicamente centralizada).

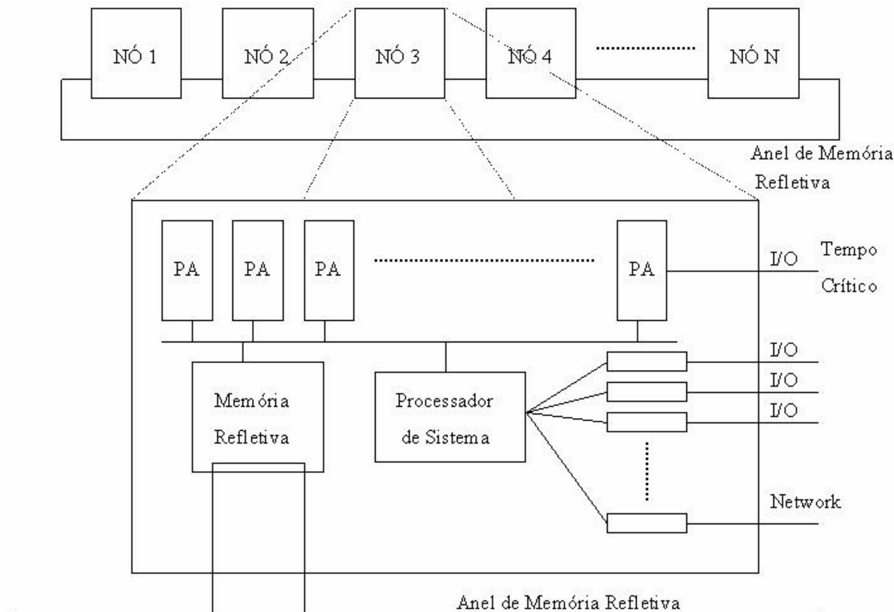


Figura 3.1 Arquitetura Spring

Como suporte ao uso deste hardware, foi desenvolvido um conjunto específico de aplicativos para geração do código, especificação dos requisitos temporais, análise off-line e *debug*. A linguagem de especificação, chamada de SDL, possui suporte para especificação dos requisitos tempo-real do sistema e detalhes da plataforma de hardware e software. A linguagem de programação, chamada de Spring-C, funciona em conjunto com a linguagem de especificação, e um aspecto chave do deste compilador é que este identifica todos os pontos críticos automaticamente e disponibiliza estas informações *off-line*, para prévia análise do funcionamento, e de forma *on-line* para o escalonamento dinâmico.

O escalonamento é executado em um co-processador *(Spring Scheduling Co Processor* – SSCOP) implementado em um *chip* VLSI. Apesar de estar implementado o algoritmo de escalonamento Spring, este componente tem capacidade de incorporar outros tipos de algoritmos de escalonamento.

Atualmente estão sendo feitas investigações de uso de outros processadores para as unidades, especialmente analisando-se o efeito dos processadores RISC no sistema Spring.

O desenvolvimento de filtros adaptativos para controle das entradas tempo-real estão sendo também alvo de estudos.

A solução em hardware apresentada em Spring, como o uso de canais óticos de comunicação e chip *VLSI* para a execução do escalonamento, entre outros, implica diretamente em um alto custo associado. Comparativamente, a solução proposta na arquitetura alvo desta dissertação utiliza-se de microcontroladores e componentes conhecidos e de baixo custo, o que significa um custo baixo para o desenvolvimento.

Realizando a comparação em nível de software, a arquitetura Spring faz uso de uma plataforma totalmente proprietária, além de todas as ferramentas de programação serem específicas, reduzindo a portabilidade do sistema e o acesso a tais ferramentas. Ao contrário da arquitetura alvo desta dissertação que faz uso de um ambiente conhecido e com código fonte aberto.

3.3 ARQUITETURA DE SUPORTE PARA SISTEMAS TEMPO-REAL DO TIPO "HARD"

Esta proposta apresentada em [HAL92], especialmente desenvolvida para sistemas embarcados, consiste em uma arquitetura multiprocessada formada por um ou mais processadores de uso geral, convencionais, e um coprocessador para o sistema operacional. Esta estrutura fornece um suporte para a implementação de sistemas tempo-real do tipo *hard*, com forte suporte a previsibilidade de comportamento, juntamente com o suporte em nível de software.

O objetivo proposto neste trabalho é o de concentrar todas as atividades administrativas do sistema em um processador (chamado de coprocessador), permitindo que o processador responsável pela execução das atividades da aplicação seja interrompido somente se necessário. Os processos externos e os periféricos são controlados por tarefas no processador da aplicação, juntamente com as tarefas da aplicação, via linhas de dados. Porém todo os sinais de controle destes periféricos (geradores de interrupções, geração de sinais) são tratados pelo coprocessador, que também é responsável por todos os serviços do sistema operacional.

O coprocessador é constituído de 3 níveis hierárquicos e distintos, com funções distintas, e foi desenvolvido focando-se o uso de um algoritmo de escalonamento do tipo EDF, adotado para esta arquitetura.

- Nível de hardware: Neste nível são implementadas funções básicas de hardware. É gerado um relógio de alta resolução, com característica tempo-real, responsável pela base de tempo do sistema. Contadores de tempo para a geração de eventos programados;
- Nível de reação primária: Reconhecimento de eventos, ou seja, interrupções, sinais, eventos de tempo, *status* de transferência de dados e sincronização. Gerenciamento dos tempos de escalonamento e instantes críticos. Registro de eventos para *debug* de erros;
- Nível secundário de reação: Implementação do escalonamento do tipo EDF. Gerenciamento do armazenamento hierárquico orientado a tarefa. Execução (secundária) da reação aos eventos previamente tratados no nível anterior. Gerenciamento da sincronização e uso das variáveis compartilhadas. Comunicação com o processador das tarefas.

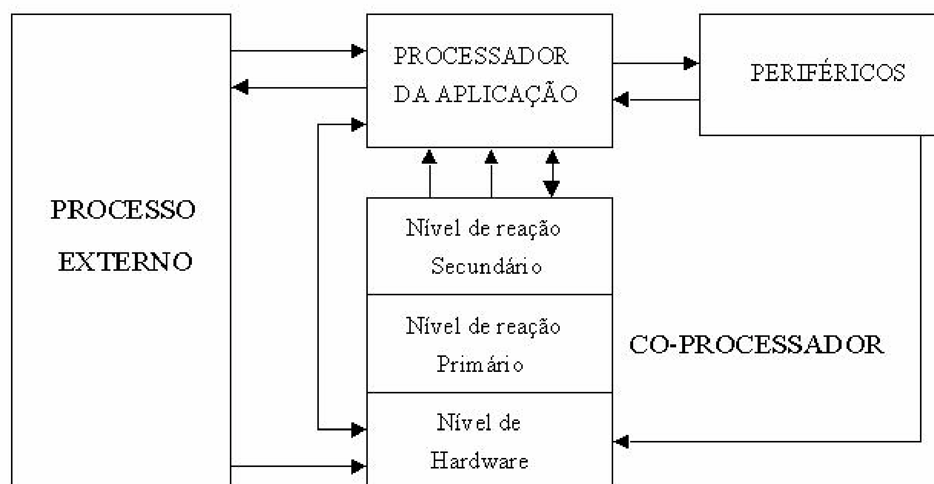


Figura 3.2 Arquitetura proposta em [HAL96]

Em nível de software é desenvolvida uma linguagem de programação, baseada em PEARL, chamada de miniPEARL [HAL96], com várias modificações, incluindo certas modificações. A linguagem de programação, o compilador e o analisador em tempo de execução são desenvolvidos visando-se a plataforma de hardware e o sistema operacional.

O compilador foi desenvolvido de maneira a gerar um código eficiente e previsível para o sistema alvo. Isto é possível pela utilização de uma linguagem de programação simples e também pela chamada “macros de interpretação” (“*translation*

macros”), específicos para cada sistema (Motorola, etc.). Como estas macros trazem a informação específica do sistema alvo, é possível saber quais os recurso do processador serão utilizados na compilação do código, permitindo uma total previsibilidade de tempo de execução e comportamento do sistema. Outra facilidade das macros é a de permitir que o programa gerado seja independente do sistema.

Atualmente, o processador para a aplicação utilizado é o MC68307, da Motorola. Para a função do coprocessador está sendo usado um “*transputer*” juntamente com o componente MC68306 que realiza as funções de mais baixo nível em hardware. Porém está em desenvolvimento a implementação do coprocessador baseado *em “Logic Cell Array*”.

Como em Spring, o ambiente de desenvolvimento em software apresentado nesta arquitetura também é específico, dificultando a portabilidade e o uso de trabalhos já desenvolvidos. Além de sua plataforma de hardware, que também é fechada.

3.4 ANÁLISE COMPARATIVA

Comparando as três propostas anteriormente apresentadas com a arquitetura apresentada nesta dissertação, pode-se concluir que todas buscam atender objetivos semelhantes: o compromisso de uma unidade de controle que apresente características de forte determinismo temporal e previsibilidade de comportamento. Porém as soluções adotadas em nível de software e hardware se diferenciam.

Nas propostas apresentadas em [NIE92] e [HAL96], faz-se o uso de um hardware dedicado e desenvolvido exclusivamente para a aplicação implementada, o que leva a um maior custo. Diferentemente, na presente dissertação é seguida basicamente a mesma linha de [PON98], onde são utilizados microcontroladores de baixo custo e de uso geral, com a diferença de que serem de 32bits.

Do ponto de vista de software, em todas as versões comparadas o sistema operacional e as funções de mais baixo nível foram desenvolvidas especialmente focadas na arquitetura de hardware proposta, prejudicando a portabilidade e extensão dos serviços implementados. Em [PON98] e [HAL96] os sistemas operacionais são bastantes restritos em suas funcionalidades. O presente trabalho é baseado em um sistema operacional de código aberto e de uso bastante difundido: uClinux, além de possuir ferramentas de programação também com as mesmas características. Isto facilita em muito a portabilidade dos códigos gerados e permite o aproveitamento de trabalhos já desenvolvidos com maior facilidade.

A proposta da presente dissertação representa então uma extensão ao trabalho desenvolvido por Pontremoli [PON98], visando oferecer um melhor suporte para programação de sistemas distribuídos (usando, por exemplo, ambientes como o proposto em [BRU00]).

Para permitir uma maior compreensão do trabalho realizado em nível de software, relacionado ao uso sistema operacional uClinux como ambiente tempo-real, serão apresentados, no capítulo seguinte, alguns trabalhos que tratam de extensões tempo-real para o sistema operacional Linux.

4 TRABALHOS COMPLEMENTARES

Os trabalhos apresentados neste capítulo são realizados com base em arquiteturas convencionais. Porém o intuito de serem abordados é o de enfatizar e justificar o uso do sistema operacional Linux como suporte de sistema operacional, e de permitir uma maior compreensão do trabalho realizado em software.

A primeira seção (4.1 Extensão Tempo-Real para Linux) traz duas propostas de extensões tempo-real para o Linux, mostrando a possibilidade de uso deste sistema em ambientes tempo-real. A segunda seção (4.2 Sistema Operacional de Código Fonte Aberto para Sistemas Embarcados – Brudna, C.) traz o uso da versão do Linux para sistemas embarcados, denominada uClinux, e tem o seu enfoque no uso deste sistema em ambientes embarcados, não preocupados com a questão tempo-real.

Uma das grandes motivações de se utilizar um sistema operacional com o código fonte aberto estão: o seu baixo custo e a facilidade de adaptabilidade e os desenvolvimentos futuros. Em contra partida, os sistemas de tempo-real comerciais tem o seu custo muito elevado e geralmente não oferecem o seu código fonte. Em um ambiente de pesquisa isto se torna um fator pouco aceitável.

Algumas destes trabalhos a seguir apresentados (seção 4.1 Extensão Tempo-Real para Linux) são para plataformas tipo desktop, porém os seus conceitos podem ser estendidos para aplicações em sistemas embarcados no qual se enquadra esta dissertação.

4.1 EXTENSÃO TEMPO-REAL PARA LINUX

Na sua forma original, o sistema operacional Linux não apresenta características de sistema tempo-real. Para superar esta limitação, surgiram algumas propostas para tornar o Linux um sistema operacional com requisitos tempo-real. Dentre estas propostas foram escolhidas duas para comparação no presente texto por serem bem conhecidas na comunidade científica e por apresentarem idéias de implementação interessantes e que pudessem ser aproveitadas no hardware apresentado nesta proposta.

4.1.1 RT-Linux

Como descrito em [YOD97]: “RT-Linux é uma variante *Hard Real Time* do conhecido Linux, que o torna utilizável em controle de robôs, sistemas de aquisição, plantas de manufatura e outras máquinas ou instrumentos com requisitos temporais”.

Um dos objetivos da proposta RT-Linux, é a de uso de uma ferramenta de uso prático, um sistema operacional tempo-real, para pesquisa que não seja proprietário [YOD97]. Uma vez que o código fonte do Linux é livre, e possui uma política de gerenciamento do seu uso bastante clara (GNU *General Public License*), este objetivo é alcançado.

Também está entre os objetivos do RT-Linux estudar quais algoritmos de escalonamento estão vindo a ser mais úteis para aplicações em tempo-real o que é possível pela facilidade de troca dos algoritmos de escalonamento pelo próprio usuário (através de um módulo do núcleo).

Para incorporar as características tempo-real ao sistema operacional LINUX, a solução adotada foi a de criar um novo núcleo de sistema. Este núcleo agora é completamente preemptivo e que está ligado diretamente ao hardware do sistema. O sistema operacional Linux padrão coexiste com este núcleo tempo-real, porém este é considerado uma tarefa de baixa prioridade para o este.

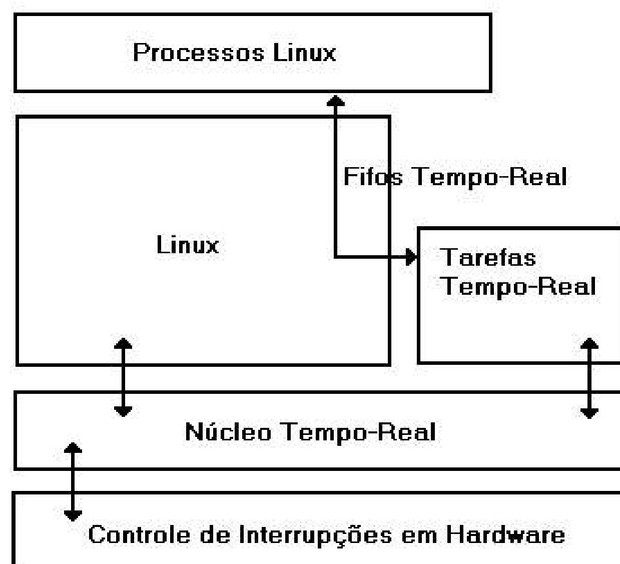


Figura 4.1 Diagrama em blocos RT-Linux

Este núcleo tempo-real foi elaborado de maneira a ser totalmente preemptivo, não possui alocação dinâmica de memória, não mantém nenhuma sincronização de estrutura de dados e as interrupções nunca são desligadas (o que previne o atraso em certas ocasiões ao

atendimento de uma interrupção para tarefas RT). Se estas características tivessem de ser incorporadas ao atual Linux, significaria uma completa rescrita do código fonte do núcleo.

Sempre que houver alguma tarefa RT para ser executada, esta será executada preferencialmente às tarefas não tempo-real, seja qual for o estado de operação do Linux. Isto significa que se a aplicação tempo-real implementada for bastante "pesada", ou seja, causar bastante carga na CPU, o Linux padrão terá mínima capacidade de processamento e poderá até nunca mais vir a ser executado.

As tarefas tempo-real são consideradas como módulos do *kernel*, sendo que no desenvolvimento destes, usa-se uma API definida para a caracterização destas tarefas, quanto aos parâmetros de qualidade de serviço ("QoS"). Isto traz alguns benefícios, alguns dos quais ajudam para que:

- As tarefas RT possuem privilégio de ser parte do núcleo e assim ter acesso direto ao hardware;
- As tarefas RT podem ser incluídas dinamicamente;
- O tempo da troca de contexto entre as tarefas é diminuído, pois todas compartilham o espaço de memória do núcleo;
- A memória das tarefas não é colocada fora de página, isto evita os acessos a memória com atrasos imprevisíveis.

Pelo fato de haver a separação das tarefas RT e não RT, sendo que as primeiras são executadas pelo novo núcleo e as outras no Linux padrão, as tarefas tempo-real não podem mais fazer o uso das chamadas de sistema diretamente. Porém, para contornar esta situação, as tarefas RT podem se comunicar com as tarefas do Linux através de duas maneiras:

- Escrita/Leitura em memória compartilhada, entretanto nenhum mecanismo de sincronização no uso desta é oferecido;
- Filas do tipo FIFO tempo-real. A sincronização é oferecida e de maneira invisível ao usuário

Na atualidade, um grupo de pesquisas e empresas ([UCL01], [LIN01b]) está estudando as primeiras versões da proposta RT-Linux para sistemas embarcados, o uCLinux. Porém uma das maiores dificuldades encontradas é na característica estática que é conferida a esta versão, uma vez que para sistemas embarcados, o sistema operacional Linux não suporta

a inclusão "*on-line*" de módulos, isto impede que tarefas RT sejam criadas e terminadas de forma dinâmica.

Outra proposta de extensão utilizando a mesma fundamentação do Linux RT, é o RTAI ([RTL 01a]). Nascido a partir do Linux RT, quando o sistem operacional ainda não havia suporte à ponto flutuante, o RTAI diferencia-se por possuir um conjunto mais amplo de funções disponíveis ao usuário (uma API[1] maior). De um ponto de vista mais amplo, a linha de pesquisa do RT-Linux defende que a manutenção da característica hard-real-time ao sistema se deve a conservação de um núcleo "leve", pensamento não compartilhado pelo grupo de pesquisa do RTAI.

4.1.2 RED-Linux

Outra proposição oferecida para se usar o sistema operacional Linux como ambiente Tempo-Real é a extensão RED-Linux. Porém, ao contrário do RTLinux, este não agrega um outro núcleo executivo, mas sim realiza modificações no núcleo do Linux para torná-lo capaz de trabalhar com tarefas tempo-real e não tempo-real.

Um primeiro trabalho desenvolvido neste sentido foi descrito em [WAN01b] sendo chamado de RTE-Linux, para depois haver a proposta da implementação nesta arquitetura de um escalonamento tempo-real genérico e facilmente modificável pelo usuário [WAN01a], tendo então o nome de RED-Linux

Este trabalho identifica os vários pontos em que o atual Linux compromete o seu uso como sistema operacional tempo-real e aponta soluções para tais. Tem-se então neste caso apenas um núcleo de sistema operacional, capaz de atender tanto a tarefas tempo-real como a tarefas não tempo-real.

Uma primeira observação é a da necessidade de uma resolução temporal melhorada e bem precisa, com relação aos atuais 10ms de "*tick*[2]" para o Linux. Isto se mostra necessário pois várias aplicações na vida real na área de automação industrial necessitam uma resolução mais refinada, na ordem de microsegundos. Para tanto utiliza-se a mesma técnica adotada em RTLinux: *Microtimer* [BAR97a].

Outra observação feita está no tempo de resposta de ativação das tarefas. Este tempo compreende deste o instante da ocorrência do evento (associada a tarefa) e o início da execução desta, geralmente definido como latência de ativação da tarefa [BAR97a].

---

[1] API (Application Program Interface) é o conjunto de funções que permite a interação com o sistema operacional e utilizada na programação da aplicação.
[2] Tempo entre cada interrupção (pulso) do relógio é denominado de *tick*.

O Linux usa o desligamento temporário de interrupções para resolver o problema de acesso à regiões críticas do sistema operacional. Neste sentido, é usada uma técnica também usada em RT-Linux e KURT [SRI95], chamada de emulação de interrupção. Consiste basicamente em não permitir que o Linux desligue as interrupções (o que pode causar tempos de latência imprevisíveis no sistema). Quando é feito o pedido de desligamento das interrupções pelo Linux, e havendo a ocorrência de alguma, esta é analisada, e se for referente a uma tarefa RT, esta acionada prontamente. Caso contrário, é armazenada e informada ao Linux quando houver um pedido de reabilitação das interrupções.

Um aspecto do Linux, que também vai contra as características de um sistema operacional tempo-real, é a de ser monolítico e não preemptivo, no que diz respeito a atividades do próprio sistema. Havendo alguma interrupção o algoritmo de escalonamento terá grandes chances de ser chamado imediatamente, fazendo com que o processo corrente seja trocado por outro. Porém se esta interrupção chegar durante a execução de alguma rotina de serviço do núcleo (chamada de sistema, por exemplo), esta será terminada para que, somente após, o escalonamento seja chamado.

Esta característica de ser não preemptivo em serviços do sistema, também contribui para a sincronização no acesso a regiões críticas do sistema operacional, juntamente com o controle de habilitação das interrupções. Quando na chamada de sistema somente esta própria pode chamar o escalonamento, quando pedir para colocar a tarefa no estado e espera por algum evento.

As chamadas de sistema podem levar alguns microsegundos quando ajustando a prioridade em algum processo ou até alguns segundos quando realizando leitura de grandes blocos de dados de disco. Isto significa que as tarefas que usam chamadas de sistema podem ter tempos de execução imprevisíveis [SAL88].

Alguns serviços do sistema operacional são repartidos em vários pequenos blocos de execução para que a granularidade de tempo seja menor. Ou seja, ao invés de executar todo o serviço de uma só vez, existem chamadas para colocar a tarefa em estado de espera em vários pontos do código de execução do serviço.

Porém para aplicações tempo-real, estes blocos possuem tempos de execução relativamente longos. A idéia implementada em RTE-Linux é a de diminuir ainda mais estes blocos para que o sistema operacional possa reagir mais rapidamente a eventos externos. A

resolução do *kernel* é determinada então pelos blocos que possuem o maior tempo de execução.

Para a implementação desta idéia, foi necessário fazer uma análise cuidadosa em todo o núcleo do sistema operacional. Porém o trabalho é menor do que rescrever um outro núcleo totalmente preemptivo, o que exigiria também o uso de muito mais semáforos ou outros mecanismos de sincronismo no acesso a regiões críticas.

A proposta RED-Linux é a de usar todos os recursos de RTE-Linux para implementar características tempo-real e incorporar um escalonamento termpo-real genérico em que o algoritmo para tal possa ser modificado a qualquer momento pelo usuário. A idéia é a de dividir o escalonamento em duas partes: Alocador (*Allocator*) e o Despachador (*Dispatcher*).

O *Dispatcher* é implementado através de um módulo que é "linkado" ao núcleo RED-Linux dinamicamente. Neste núcleo está implementado o algoritmo de escalonamento para as rotinas tempo-real. Somente quando não houver nenhuma rotina tempo-real para ser executada, o *Dispatcher* passa o controle para o escalonador tradicional do Linux, para escolher alguma outra tarefa não tempo-real para usar a CPU.

A função do *Allocator* é a de determinar os atributos da nova tarefa tempo-real e informá-los ao *Dispatcher*. O *Allocator* geralmente é implementado como um processo no nível do usuário. E assim é feito para facilitar a sua troca pelo usuário.

As funções tempo-real são criadas primeiramente de maneira normal, no Linux tradicional, e usando uma serviços de uma API, esta é transformada em tempo-real. Isto é feito pelo *Allocator*, que examina as condições QoS para a tarefa e aceita ou não a sua inclusão no *Dispatcher*.

Existe também um mecanismo de comunicação entre o *Dispatcher* e o *Allocator*, para permitir que informações sobre as tarefas tempo-real sejam repassadas para o *Allocator* e este avalie o desempenho do algoritmo de escalonamento. Assim tem-se a possibilidade de se usar algoritmos de escalonamento bastante elaborados que levam em conta o tempo de execução em tempo de execução das tarefas, por exemplo.

Como mostrado em [WAN01a] e [WAN01b], a filosofia apresentada em RED-Linux pode facilmente ser transportada para outros sistemas operacionais que desejam se tornar tempo-real, inclusive em versões para sistemas embarcados como o uClinux.

Levando-se em conta que o uClinux não possui o módulo de gerenciamento de memória (mmu), muitos problemas de acesso à memória virtual, falhas de página e o chamado troca (*swap*) em disco estão descartados. Estes fatos acarretam em inclusões de tempo de execução imprevisíveis às tarefas, acabando com as chances de se ter um determinismo temporal.

4.2 SISTEMA OPERACIONAL DE CÓDIGO FONTE ABERTO PARA SISTEMAS EMBARCADOS – BRUDNA, C.

Dentre os trabalhos relacionados, este é o único que não tem a intenção da utilização de um sistema operacional tempo-real. Ele está relacionado neste capitulo para ser enfatizada a idéia quanto ao uso de um sistema com código fonte aberto e livre, apresentando a versão do sistema Linux na sua versão para sistemas embarcados.

Apresentado como um trabalho de dissertação pelo CPGEE [BRU00], propõe um ambiente de hardware e software como suporte à criação de sistemas de automação baseado em objetos distribuídos, utilizando-se o barramento CAN como canal de comunicação com o meio externo. Neste trabalho podemos perceber a utilização de outras ferramentas de auxílio de desenvolvimento desenvolvidas pelo mesmo grupo de pesquisa, como o SIMOO-RT[1] [BEC99a] [BEC99b] e AO/C++[1]

O sistema operacional utilizado baseia-se na versão para sistemas embarcados do Linux, o uClinux, e faz uso de um processador de baixo custo e 32bits, o 68332. Porém já existe em desenvolvimento e estudos vários portes para outros processadores: i960, ARM, 8086 e MIPS entre outros [UCL01]. Outras versões para sistemas embarcados usando o Linux como fonte podem ser encontradas em BlueCat [WOR01], ET-Linux [SRL01] e Hard-Hat [VIS01], porém estas necessitam de no mínimo um processador 386 e uma unidade de disco.

Este sistema operacional traz consigo praticamente todos os recursos de um sistema para desktop, como acesso a um canal de comunicação TCP/IP, temporizadores, interrupções e outros. A grande diferença encontra-se na ausência de uma unidade de gerenciamento de memória (MMU).

As ferramentas para o desenvolvimento são também disponíveis com licença pública de uso e são: *gcc* (compilador), *binutils* (ferramentas auxiliares para compilação),

[1] SIMOO-RT - ambiente integrado de modelagem orientada a objeto

linux 2.0.38 (núcleo), *pilot* (programas do ambiente de usuário), *genromfs* (gerador de sistemas de arquivo em ROM), *coff2ftt* (conversor de código em COFF para FLAT), *uC-libc* (biblioteca C) e *uC-libm* (biblioteca matemática)

[1] AO/C++ é – extensão do C++ para suporte a criação de objetos concorrentes e distribuídos

5 ARQUITETURA PROPOSTA

Avaliando-se as vantagens e desvantagens dos trabalhos previamente apresentados, chega-se a uma outra proposta de arquitetura de software e hardware para ambientes tempo-real distribuídos que apresente característica de um forte determinismo temporal e versatilidade de uso no desenvolvimento e estudo de algoritmos de escalonamento.

5.1 VISÃO GERAL

Um sistema operacional tempo-real para ambientes embarcados, aliada a uma arquitetura de hardware de baixo custo que suporte este sistema e com chances de que o usuário possa avaliar diferentes tipos de algoritmos de escalonamento sem causar sobrecarga ao sistema, é o principal objetivo deste trabalho.

Para alcançá-lo e avaliando os resultados dos trabalhos apresentados anteriormente, uma solução possível seria a adoção do uClinux incluindo as modificações apresentadas em RED-Linux. Porém, como mostrado em [PON98], tal proposta não eliminaria o problema de que a aplicação (tarefas no nível de usuário) sofram a interferência da sobrecarga na CPU quando é adotado um algoritmo de escalonamento diferente e mais elaborado.

Este fato é ainda mais significativo pois um dos objetivos da presente dissertação é a de que o sistema proposto possa suportar o estudo de vários algoritmos de escalonamento tempo-real e de manter o determinismo temporal deste sistema.

Na proposta RED-Linux há um suporte para o desenvolvimento de algoritmos tempo-real bastante elaborados, os quais levam em conta a mensuração dinâmica de certas características das tarefas para ser usado pelo escalonamento. Nestes casos porém, há uma sobrecarga do sistema, inserida principalmente pela necessidade de comunicação do *Dispatch* para o *Allocator*.

A proposta apresentada no presente trabalho busca combinar as vantagens dos sistemas apresentados: aspectos de aumento do determinismo de uma arquitetura de hardware com mais de um processador, uso de um sistema operacional de código aberto e com suporte

para o desenvolvimento de aplicações distribuídas, como o Linux e utilização de processadores de baixo custo, como é o caso dos microcontroladores de 32 bits.

Ter-se-á então uma arquitetura que englobará as vantagens de um ambiente de desenvolvimento de baixo custo e um sistema operacional já desenvolvido, consagrado e em constante evolução, aliado a uma arquitetura de hardware de baixo custo que suportará este sistema operacional.

Em [PON98] foram usados três microprocessadores de 8 bits (80C52) para realização de funções distintas, sendo que um deles é dedicado quase que exclusivamente à tarefas da aplicação. Para este trabalho, a proposição é a de se usar dois processadores, agora de 32 bits mantendo um deles dedicado às tarefas da aplicação.

Neste sentido, as atividades de comunicação, gerenciamento temporal, escalonamento e gerenciamento dos sinais de entrada e saída (incluindo os assíncronos) serão feitos por outro microcontrolador. Isto não seria possível para apenas um microcontrolador de 8 bits, mas com 32bits e uma maior capacidade de processamento, o objetivo se torna viável.

Em termos de funcionalidade, a proposição é a de manter-se a divisão em 3 blocos funcionais, porém nesta o bloco gerente e de comunicações são suportados por um microcontrolador, constituindo o bloco secundário, enquanto que as tarefas são executadas em outro microcontrolador no bloco principal. O sistema operacional será então dividido nestas duas partes, como mostrado na Figura 5-1.

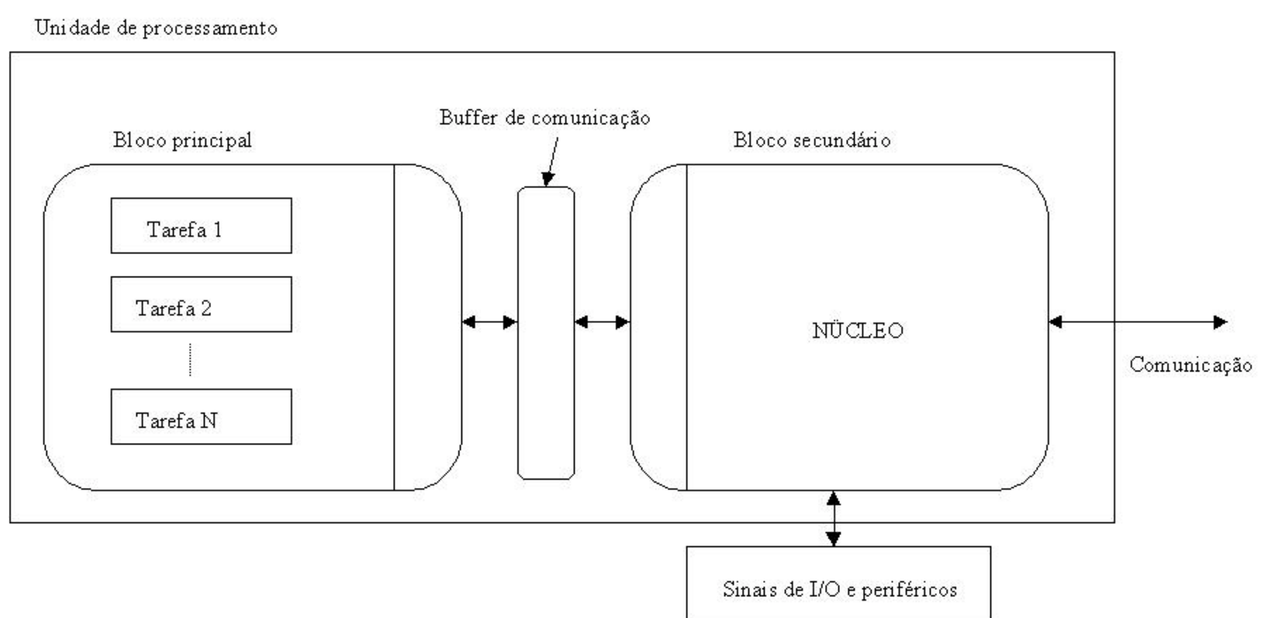


Figura 5.1 Unidade de processamento

5.1.1 Bloco Principal

Será equivalente ao bloco executor apresentado em [PON98], e suas principais atividades são [KIR88]:

- Execução das tarefas da aplicação;
- Gerenciamento da área de memória associada a cada tarefa;
- Troca do contexto quando há a necessidade de que outra tarefa ocupe a CPU em lugar da atual;
- Comunicação com o bloco secundário, onde está o restante do sistema.

5.1.2 Bloco Secundário

Este bloco englobará todas as atividades que não pertençam à aplicação, quais sejam:

- Gerenciamento do sistema de arquivos: organização da memória associada aos arquivos;
- Gerenciamento do sistema de entrada e saída: sinais assíncronos, acesso a dispositivos de entrada e saída;
- Gerenciamento da temporização do sistema: para ser possível atender aos requisitos temporais das tarefas tempo-real;
- Escalonamento das tarefas: onde será implementado os algoritmos de esalonamento;
- Gerenciamento da comunicação com o meio externo: Uso de TCP/IP, CAN (esta atividade está relacionada ao gerenciamento dos sinais de entrada e saída).

5.1.3 Particionamento do sistema

Para que seja possível utilizar-se do sistema operacional uClinux nesta plataforma de hardware é necessário realizar-se um estudo para particionamento do sistema, conforme as atividades de cada bloco funcional descritas anteriormente. O objetivo desta análise é verificar se o "particionamento teórico" anteriormente apresentado, o qual é baseado apenas nas diversas funcionalidades realizadas pelo sistema operacional, é realmente adequado e não

compromete o desempenho do sistema. Em especial aspectos como localização de variáveis do sistema operacional, concorrência de suas operações, são analisados.

Como orientação de desenvolvimento, primeiramente foram focadas as atividades no estudo do particionamento do sistema operacional partindo-se de seu projeto original (não se usando as extensões para tempo-real). Em um segundo momento realizou-se o estudo da implementação das características tempo-real neste sistema particionado.

Todavia, com esta proposta de hardware contendo dois processadores, pode-se antecipar vantagens no uso das idéias apresentadas em RED-Linux como solução de adaptação do uClinux para um sistema tempo-real, pois permite mais facilmente a sua implementação e disponibiliza o acesso às chamadas de sistema em tempo-real, atividade não possível na extensão RT-Linux. Basicamente, as extensões propostas em RED-Linux dizem respeito a uma melhor gerência das tarefas e do escalonamento, afetando quase que exclusivamente as operações realizadas pelo bloco secundário. Isto significa uma maior portabilidade dos trabalhos já desenvolvidos neste sistema.

Para se analisar o particionamento do sistema operacional, deve-se avaliar a sua estrutura e verificar as atividades correspondentes ao sistema e a aplicação. A Figura 5.2 mostra os principais níveis de um sistema computacional utilizando-se do Linux.

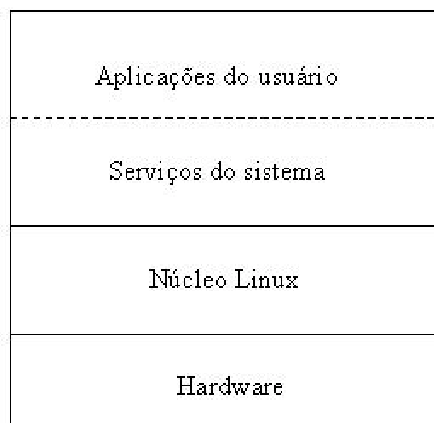


Figura 5.2 Camadas de um sistema computacional Linux

- Aplicações do usuário: É a aplicação propriamente dita. São conjuntos de processos, concorrentes, que implementam a atividade desejada pelo usuário.
- Serviços do sistema: São serviços tipicamente considerados como parte do sistema operacional (como o interpretador de comandos *shell*), porém fazem parte da aplicação. Também faz parte deste sub-nível uma biblioteca de

funções que podem ser utilizadas pelo usuário na produção da aplicação. Nesta biblioteca encontram-se versões mais avançadas do que as básicas chamadas de sistema oferecidas pelo sistema operacional e também funções que não correspondam diretamente a chamadas de sistema, como funções matemáticas e rotinas de manipulação de *strings*.

- Núcleo Linux: o núcleo do sistema operacional propriamente dito. Aqui encontram-se implementado o gerenciamento de arquivos, dos dispositivos de entrada e saída, o gerenciamento de memória e etc.
- Hardware: Todas as partes físicas que compõem o sistema que tem o seu uso gerenciado pelo sistema operacional.

Em nível de software as camadas "Aplicativos do usuário" e "Serviços do Sistema" compõe as tarefas da aplicação. A camada "Núcleo Linux" é o coração do sistema, e onde são realizadas todas as atividades que o qualificam como um sistema operacional, como o gerenciamento da memória, de processos, dos dispositivos de entrada e saída e etc.

A camada "Núcleo Linux" pode ser divida, como mostrado na Figura 5.3 em cinco subsistemas principais:

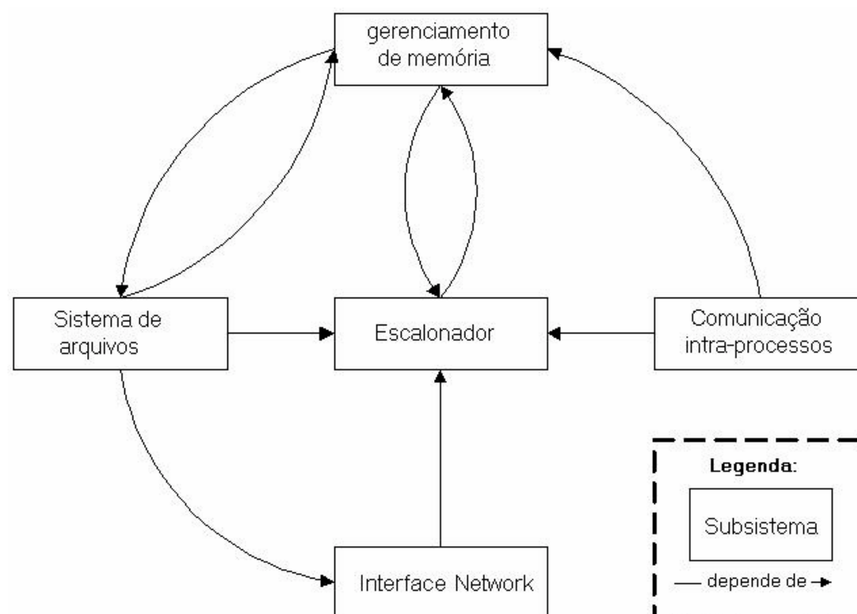


Figura 5.3 Arquitetura conceitual do Linux

- Escalonamento dos processos: Responsável pelo controle de acesso dos processos à CPU. Aqui é executado o algoritmo de escalonamento. Neste módulo é feito o uso e gerenciamento da temporização do sistema.
- Sistema de arquivos ("*File Systems*"): Disponibiliza a leitura/escrita a dispositivos externos bem como o acesso a vários tipos de arquivos (executáveis, armazenagem de dados, etc). Este módulo é responsável também pela gerência dos eventos assíncronos.
- Comunicação intra-processos: Disponibiliza serviços de comunicação entre os processos em um sistema Linux.
- Gerenciamento de memória: gerencia o uso da memória principal e secundária do sistema, alocando o espaço necessário para cada processo e para as atividades do sistema operacional.
- Interface Network: Provê o acesso à comunicação com o meio externo.

Este diagrama enfatiza o principal subsistema: o escalonador. Todos os outros subsistemas dependem deste, uma vez que o usam quando necessitam suspender ou reiniciar algum processo. Isto acontece quando algum processo necessita ser suspenso para esperar pela conclusão de alguma atividade de hardware, ou então reativar este processo quando a atividade estiver terminada.

Para o caso da versão para sistemas embarcados do sistema operacional Linux, o uClinux, não existe o gerenciador de memória. Isto traz algumas limitações em relação ao sistema padrão, como por exemplo não há o uso do disco rígido como memória secundária (não possibilitando o uso de *swap*). Também no módulo de comunicação inter-processo, dentre os meios de comunicação, não é possível o uso de memória compartilhada para tal.

Com esta visão do sistema Linux, apresentada torna-se possível distinguir quais são os componentes de software que ficarão alocados em cada bloco (Principal e Secundário) na arquitetura proposta. Como se tem o objetivo de separar a aplicação das atividades do sistema, as camadas Aplicativos do Usuário e Serviços do Sistema são alocados no bloco Principal, pois estas correspondem exclusivamente a códigos da aplicação. O Núcleo Linux ficará alocado no bloco Secundário, pois suas atividades (explanadas através dos seus subsistemas) correspondem às atividades previamente definidas para este bloco.

Em uma arquitetura convencional, as atividades do sistema operacional são realizadas usando-se o mesmo espaço de memória de dados, memória de código e *stack*, sendo este ambiente denominado de nível privilegiado, ou nível de sistema. Isto acontece pois o núcleo é criado como um único bloco monolítico, assim todo o código do núcleo ocupa o mesmo espaço de endereço. O núcleo possui total acesso aos recursos de hardware.

Para a execução dos processos da aplicação existe uma região de memória (contexto) associada a cada um destes quando o escalonador decide pela ativação de um determinado processo, é realizado um "chaveamento de contexto" para a sua região de memória. Os processos (aplicação) são executados em um nível não privilegiado, ou nível de usuário. Com a arquitetura proposta apresentada, o bloco Principal executará as atividades em nível de usuário enquanto que o bloco Secundário executará as atividades em nível de sistema.

Pela análise realizada, percebe-se que o sistema Linux já promove uma divisão, entre as atividades do sistema operacional e da aplicação, o que facilita o particionamento físico destas unidades entre os blocos pré definidos para a arquitetura proposta. Um dos gerenciamentos de memória, necessário para a realização do chaveamento de contexto dos processos, não será executado no bloco Secundário, mas sim terá de ser executado no Principal, pois neste estará a memória de código e dados relacionado aos processos.

6 PROJETO DE HARDWARE E SOTWARE

Neste capítulo serão apresentadas as justificativas de utilização dos componentes envolvidos e do suporte a nível de software utilizado.

6.1 PROJETO DE HARDWARE

Para a implementação das idéias até aqui apresentadas, e tornar completo o objetivo desta dissertação, é proposto um hardware projetado para suportar as implementações de *software* sugeridas. A Figura 6-1 traz o diagrama em blocos da arquitetura de hardware proposta.

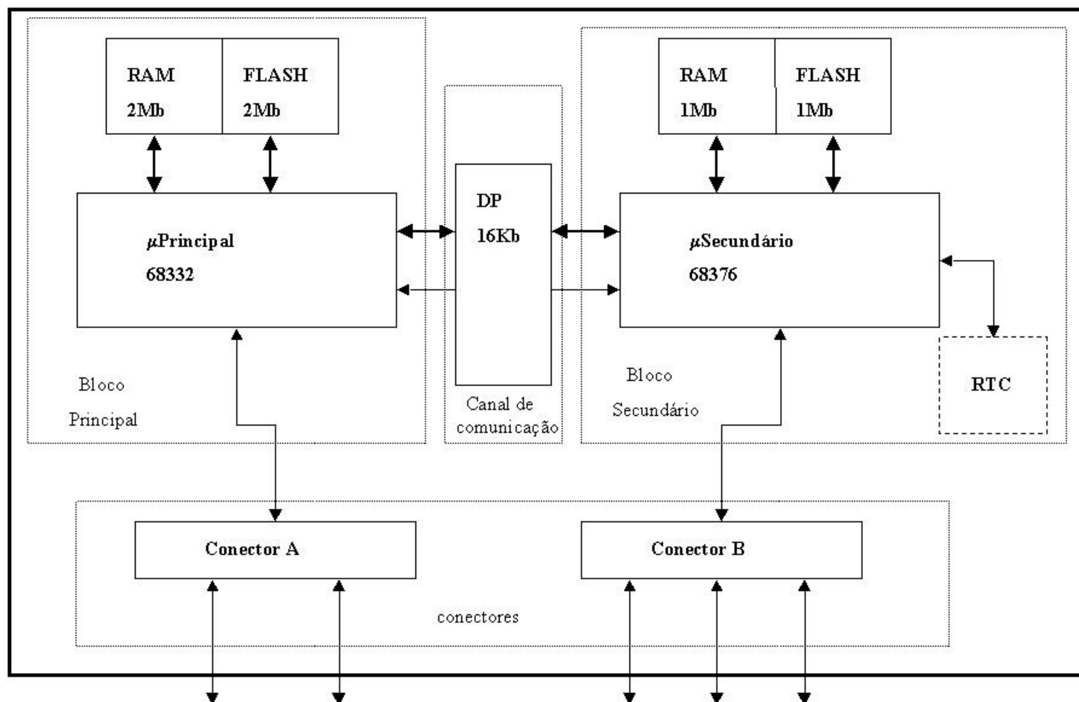


Figura 6.1 Diagrama em blocos do Hardware proposto

6.1.1 Escolha dos microprocessadores

Para a escolha dos microcontroadores a serem utilizados, tem-se como critério a necessidade que tal componente suporte o sistema operacional uClinux, possibilite ferramentas de programação em linguagem de alto nível, em especial que permita o desenvolvimento de programas concorrentes usando linguagens orientadas a objetos, tais como em [BRU00] . Dentre as opções hoje disponibilizadas optou-se pela família 68XXX, fabricados pela Motorola. Tal componente além das características necessárias possui embutido um módulo de comunicação CAN, que é adotado como canal de comunicação com o meio externo e que permite o desenvolvimento de drivers de maior performance além de uma economia no *layout* da placa de circuito impresso. Dentre as suas características, pode-se ressaltar:

- Possuem compiladores disponíveis na *internet*, de licença pública livre GPL;
- Há disponível também as versões para sistemas embarcados do sistema operacional Linux, na versão uClinux para alguns microcontroladore da família 68XXX. O código fonte destas versões é aberto;
- A capacidade de processamento e gerenciamento de memória é maior do que os oferecidos pela linha 80C52 (utilizados na proposta [PON98]), torna possível um agrupamento maior das atividades para um microcontrolador;
- É um componente de baixo custo;
- Possui versões que disponibilizam um módulo CAN embutido.

6.1.2 Bloco principal

Este bloco será responsável pela execução das tarefas da aplicação, não sendo necessário muitos módulos especiais agregados à CPU, apenas uma alta capacidade de processamento. Assim a escolha do processador 68332 se torna uma boa opção, pois permite o aproveitamento do porte do uClinux para este microcontrolador já realizado em [BRU00].

Este processador oferece ainda recursos como uma unidade de temporizadores (TPU), que permite a geração de saídas especiais, geração de eventos programados

(associados a pinos específicos do CHIP) e que podem ser aproveitados para certas aplicações assim como no auxílio nos momentos de teste do protótipo.

A memória associada a este microcontrolador é composta por dois tipos: FLASH e RAM, sendo a primeira utilizada para a execução do código (guarda o código fonte compilado, ou seja, binário). A segunda retém as variáveis, pilha, estruturas de dados, os contextos das tarefas. Para a definição do tamanho a se utilizar, foram avaliados os resultados obtidos em [BRU00] chegando-se aos seguintes valores:

- Memória RAM: 2Mbytes;
- Memória FLASH:2Mbytes.

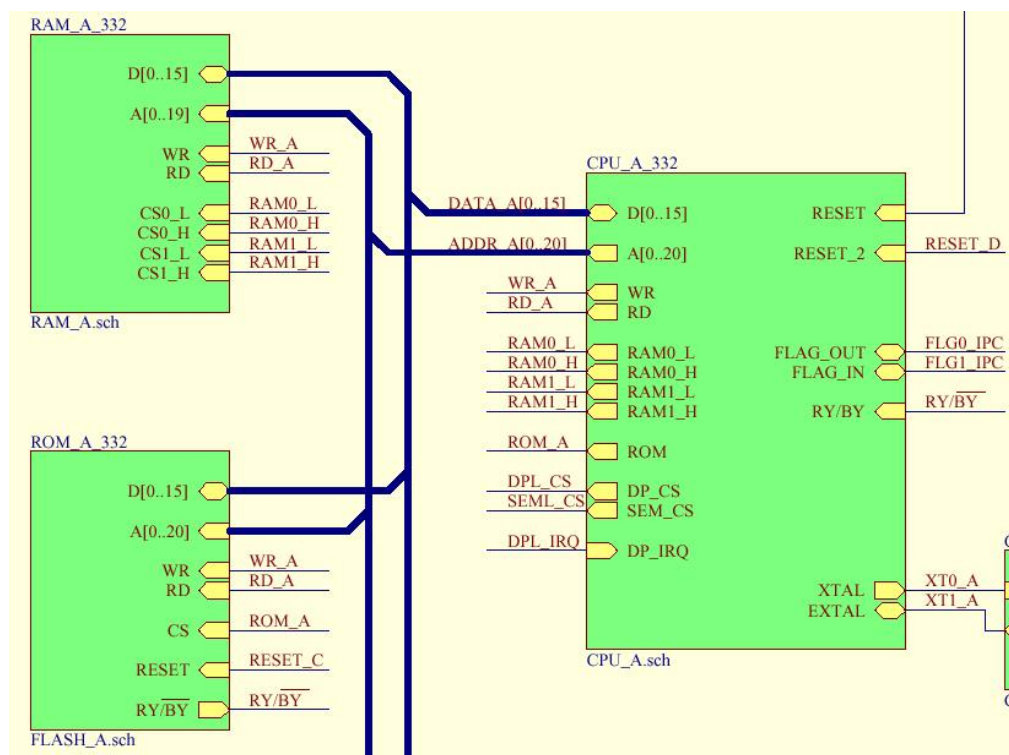


Figura 6.2 Conexão da memória com a CPU principal

A memória de código, sendo do tipo FLASH permite agilidade na etapa de desenvolvimento do sistema, uma vez que esta é escrita e apagada eletronicamente (ao contrário das do tipo EPROM).

Para a escolha dos modelos de memória a serem utilizados, e seus fabricantes, foi tomado como base o mesmo hardware utilizado em [BRU00], pois se mostrou compatível com os microcontroldores a serem utilizados e não interferiram na performance do sistema.

Assim, foram procuradas memórias com a mesma velocidade de acesso juntamente com os processadores também na mesma velocidade.

No bloco de memória FLASH foram utilizados CHIPS de tamanho 1MB, pois é a memória de maior tamanho disponível, considerando-se fatores como: tensão de alimentação de 5V e tempos de acesso compatíveis. E para as memórias RAM, foram utilizadas CHIPS de memória de 512KB, também por serem as de maior tamanho disponíveis levando-se em conta as características anteriormente mencionadas.

Pelos motivos expostos, foram escolhidas as memórias:

- FLASH: Am29F080B-75 : tempo de acesso máximo de 75ns;
- RAM: TC554001FL-70: tempo de acesso máximo de 70ns.

Em relação ao aspecto de implementação física do hardware (*lay-out*), se faz necessário a previsão de adotar "soquetes" para as memórias FLASH, permitindo que estas sejam movimentadas com facilidade entre o circuito em questão e a giga de gravação destas memórias.

6.1.3 Bloco secundário

Conforme discutido anteriormente, o bloco secundário será responsável pela execução do algoritmo de escalonamento, pela gerência de sinais assíncronos e dispositivos de entrada e saída, dentre outros. Sendo assim, a utilização de um microcontrolador que tenha embutidos alguns módulos em hardware, como unidades de temporização por exemplo, e que possam ser utilizados no suporte de tais atividades, é bastante importante.

É também desejável dispor de um canal de comunicação com o meio externo através de um barramento que possibilite a implementação de comunicação tempo-real e valendo-se do trabalho já desenvolvido em [BRU00], a escolha de um microcontrolador que tenha embutida tal funcionalidade seria ideal. A Motorola oferece nesta linha, o processador MC68376, que possui um módulo de comunicação CAN, inclusive com dois canais, o que torna a opção por este componente bastante conveniente.

Este processador também oferece duas unidades de temporizadores, diferentes no seu conceito. Um módulo TPU, oferece (como no MC68332) vários sub-módulos associados a pinos específicos do CHIP que permite a geração de eventos programados, entre outras funções. Disponibiliza também uma unidade de temporizadores configuráveis, que permite a

geração de vários tipos de interrupções. Este módulo é o que oferece grandes chances de geração do *clock* interno para o sistema.

Uma vez que neste bloco não haverá um crescimento do código gerado relacionado à aplicação (apenas servirá para memorizar o código do núcleo e suas variáveis), para a escolha do tamanho da memória optou-se pela escolha dos seguintes valores:

- Memória RAM: 1Mbytes;
- Memória FLASH:1Mbytes.

Em uma primeira versão desta proposta, com os resultados obtidos, estes valores podem ser otimizados, pois uma primeira escolha dos tamanhos ótimos a serem utilizados, em um primeiro protótipo, torna-se difícil. Com relação aos tipos de memória a serem utilizados, os motivos são os mesmos apresentados anteriormente no bloco principal.

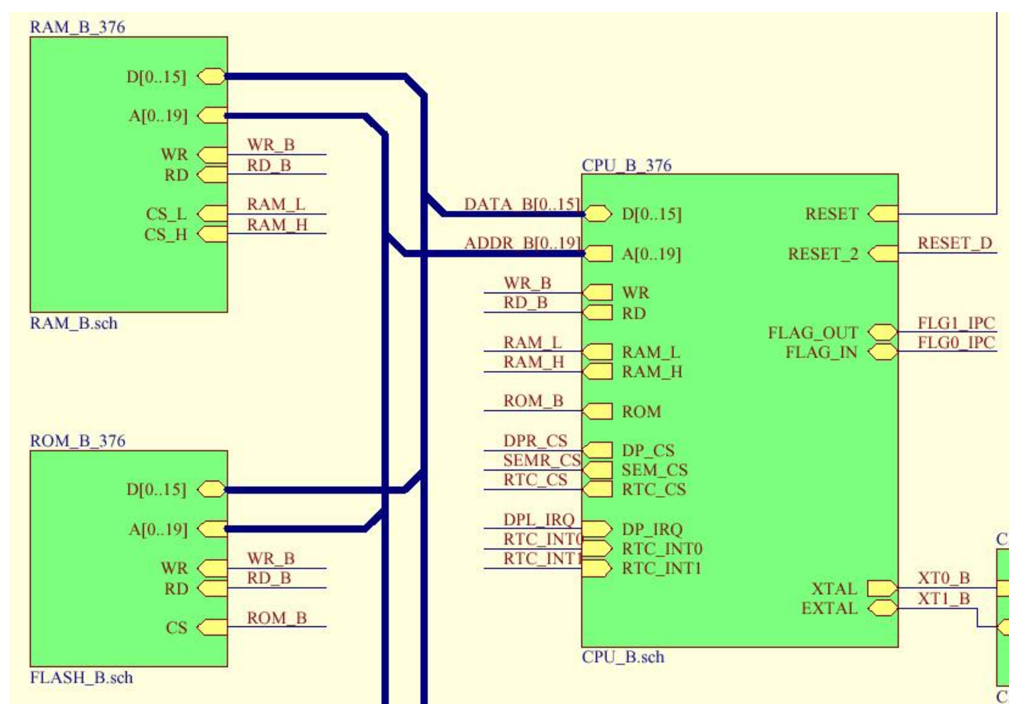


Figura 6.3 Conexão da memória com a CPU secundária

No bloco Secundário foi prevista também a inclusão de um módulo externo à CPU responsável pelo relógio tempo-real. Para a definição deste componente, foi escolhido aquele que oferecesse uma maior granularidade temporal possível, compatível com os tempos de acesso pelo processador e possibilitasse o programação de eventos (utilizando-se data e horário) e gerasse uma interrupção quando este fosse alcançado.

Com base nestas condições chegou-se ao mesmo componente utilizado em [PON98], o MM58167 fabricado pela National, que possui as seguintes característica, dentre outras:

- Resolução temporal de milisegundos;
- Possibilidade de geração de oito tipos diferentes de sinais de interrupção;
- Encapsulamento para versão em montagem de superfície, a qual deve ser adotada na montagem do protótipo.

Para a sua utilização porém, deve-se levar em conta que trata-se de um componente extremamente lento se comparado aos tempos de acesso disponibilizados pelo microcontrolador. Isto significa que, na configuração deste para o seu acesso, vários *wait states* devam ser incluídos no seu ciclo de leitura e escrita, implicando em tempos da ordem de centenas de microsegundos.

6.1.4 Comunicação entre os uP

Para permitir a comunicação entre os dois blocos descritos anteriormente, se faz necessário em hardware algum tipo de canal de comunicação entre estes. Este não pode oferecer grandes limitações quanto a velocidade de comunicação. Assim já fica descartada a possibilidade de uso de uma USART, ou seja, um canal serial. Isto já foi inclusive detectado em [PON98] (canal serial utilizado entre o bloco gerencial e de execução). Neste mesmo trabalho, há também a utilização de memórias FIFO para a implementação de certos canais de comunicação (entre os blocos principal e de comunicação).A sua utilização exige que a leitura e a escrita sejam feitas de forma seqüencial, o que diminui a versatilidade de operação.

Surge então a possibilidade de se utilizar memórias Dupla-Porta. Este tipo proporciona uma maior versatilidade em sua utilização para a implementação de modos de comunicação. Em uma memória do tipo FIFO, um dado colocado nesta memória só poderá ser lido quando os primeiros, colocados anteriormente a este, já tiverem sido lidos. Já em uma memória dupla-porta, os dados podem ser lidos em ordem arbritária diferente daquela em que foram escritos.

Este conceito pode ser expandido para o envio de mensagens. Ou seja, se estas fossem escritas em uma memória dupla-porta, estas também não necessitam ser lidas na mesma ordem da escrita, permitindo assim que mensagens contidas na memória possuam prioridades atribuídas e que estas sejam tratadas conforme estas prioridades.

A capacidade da memória dupla-porta utilizada, está diretamente relacionada com a capacidade de processamento de cada um dos blocos que a utilizam, ou seja, o seu *throughput*. Se esta for muito pequena em tamanho, podem ocorrer ocasiões em que havendo a necessidade de escrita de mensagens de um dos blocos na memória dupla-porta, não haja mais espaço para tal. Isto ocasionaria uma degradação na capacidade do sistema. Por outro lado, se esta for muito grande, o custo financeiro em hardware aumentará, uma vez que este tipo de memória possui um custo elevado proporcional ao seu tamanho.

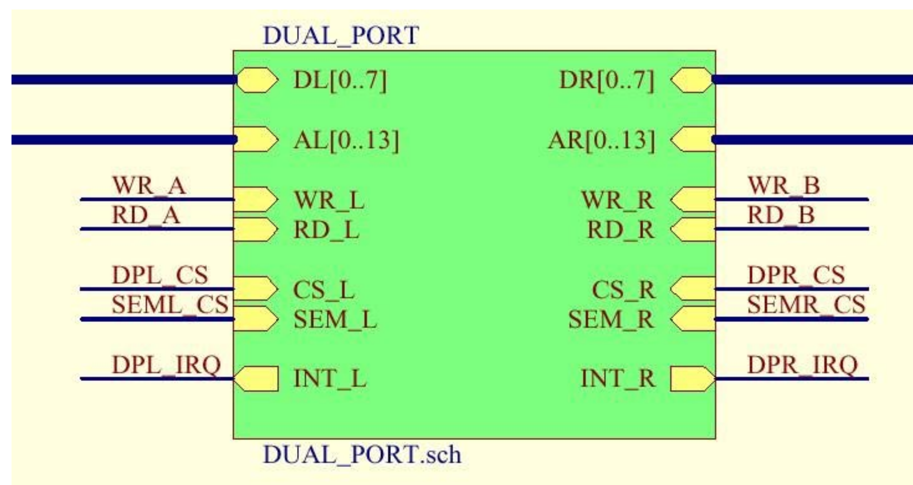


Figura 6.4 Conexão da memória Dupla-Porta

Mais uma vez, é difícil analisar teoricamente qual será a demanda de uso, por parte de cada bloco, da memória dupla-porta. Alguns recursos matemáticos podem ser utilizados, como apresentado em [LAP97], através de estudos estatísticos. A solução seria a utilização de uma memória de capacidade razoável levando em conta também as condições financeiras de se obtê-la. A oferta de mercado porém (na data de confecção do hardware), não permitia muitas escolhas, o que forçou o uso dos *chips* IDT7006, que é uma memória dupla-porta IDT7006 com capacidade de 16KB.

Outra característica destas memórias é a de oferecerem sinais de controle (semáforos[1]) para sincronização quando no acesso simultâneo de endereços. O uso destes é bastante versátil, uma vez que o usuário é que atribui as regiões de memória aos semáforos. Além disto, estas memórias oferecem possibilidade de geração de interrupções quando no acesso a um endereço específico (o que significa que o sistema pode responder rápido às

[1] No caso da memória Dupla Porta, o fabricante define semáforo como sendo um bit, disponivel no próprio CHIP, utilizado para sincronizar o acesso a mesma, sendo que este conceito será adotado durante o restante do texto.

mensagens recebidas), e o dado deste endereço pode servir como uma mensagem (de 8bits) utilizável em alguma implementação.

6.1.5 Comunicação com meio externo

A presente proposta, também tem a intenção de ser usada em ambientes de controle tempo-real distribuídos. Para tal deve ser disponibilizado um canal de comunicação que possibilite a implementação de protocolos de comunicação em tempo-real. Para tanto um canal de comunicação CAN é oferecido (como comentado em ítens anteriores), e que inclusive foi de grande influência na escolha do processador a ser usado no bloco secundário. Tendo em vista que o módulo CAN se encontra internamente à CPU, a sua velocidade e facilidade de uso ficam melhoradas.

Como não se deseja determinar aqui qual o meio físico utilizado neste barramento, pois a intenção é a de que isto seja determinada pela aplicação e por outros dispositivos, foram disponibilizados todos os sinais de controle necessários para a sua implementação física em um dos conectores previsto no hardware. Uma sugestão de uso de alguma interface CAN seria a mesma que foi utilizada em [BRU00], que utiliza-se de apenas um *chip* externo.

6.2 PROJETO DE SOFTWARE

Por se tratar de uma arquitetura para sistemas tempo-real distribuídos, o hardware necessita que seja utilizado um sistema operacional. Um pensamento natural de atribuição de uma arquitetura de software para o hardware proposto, seria a utilização de dois sistemas operacionais, um para cada um dos blocos descritos anteriormente. Ao invés disto, a proposta faz uso de um sistema operacional o qual será particionado entre os dois blocos. Com a intenção de isolar as atividades da aplicação dos serviços do sistema operacional, um dos processadores ficará totalmente dedicado para estas.

Aliado ao estudo do particionamento do sistema operacional, são propostas alterações neste para que adquira as características de um sistema tempo-real e que possa ser utilizado em ambientes de controle distribuídos, o que faz parte da proposição desta dissertação.

6.2.1 Escolha do sistema operacional

Para atender as necessidades colocadas no item anterior, um sistema operacional tempo-real comercial não seria uma escolha razoável. Além de seu custo elevado, não

possibilitaria a reestruturação deste para ser acomodado em um hardware com dois processadores, uma vez que os códigos-fonte dos sistemas operacionais comerciais são fechados e não possibilitam alterações.

Outra opção seria a do desenvolvimento de um sistema operacional novo para o hardware apresentado, o que demandaria muito esforço e impossibilitaria a reutilização de desenvolvimentos já realizados no próprio departamento. Senso assim, esta opção também não é a mais aconselhada.

Baseado nos requisitos desejados (sistema operacional com código fonte aberto, disponível para modificações, executável em microcontroladores de baixo custo, etc.) optou-se pelo uso do uClinux.

Como ponto de partida, optou-se pela utilização da versão 2.0.38 do uClinux (com o porte para o processador 68332), uma vez que esta se encontra estável e ter sido aplicada com sucesso em [BRU00].

6.2.2 Modificações no sistema operacional

Um sistema operacional, como visto no capítulo da revisão bibliográfica e mais especificamente o sistema Linux no capítulo anterior, é composto por vários sub-módulos responsáveis cada um por gerenciar funções específicas de hardware (dispositivos de entrada e saída, temporizadores) e de software (escalonamento de tarefas, gerenciamento de memória e arquivos). Todas estas atividades, incluindo as tarefas da aplicação, concorrem entre si na utilização dos recursos oferecidos. Seria interessante, e por razões já apresentadas em [PON98], que um processador estivesse inteiramente disponível para as tarefas da aplicação, como apresentado no capítulo anterior.

Para se alcançar tal objetivo, partindo-se de um sistema operacional, deve ser realizada a análise de quais são os pontos em que as tarefas da aplicação conseguem interagir com o resto do sistema e quais destas funções precisam necessariamente estarem co-alocadas (no mesmo processador de execução das tarefas). Com o particionamento definido, parte-se para a análise e alteração dos pontos críticos do sistema operacional para que este possa ser caracterizado como tempo-real.

Para uma arquitetura convencional, analisando-se o sistema Linux verifica-se que quando a CPU estiver executando algum processo (estando assim em nível de usuário), a única maneira de o controle passar para o nível de sistema é através de uma interrupção.

Estando em nível de sistema, um retorno de interrupção ou então o resultado da execução do algoritmo de escalonamento (se este for chamado) fazem com que a execução retorne a nível de usuário.

Estas interrupções podem ser originadas de algum dispositivo externo (quando hardware), algum módulo interno à CPU (*timer*) ou então gerada via software (como por exemplo a instrução *trap*[1] nos microcontroladores Motorola). Esta última é a maneira utilizada para a implementação das chamadas de sistema, que são serviços disponibilizados pelo sistema operacional aos processos.

Como na arquitetura proposta os níveis de usuário e de sistema estão separados entre os blocos Principal e Secundário, não há mais a necessidade de que haja a troca de controle da CPU entre estes níveis, pois cada bloco possui o seu microcontrolador. As interrupções de *timer* e de hardware são agora atendidas pelo bloco Secundário, onde estão os códigos dos *device drivers* (componentes do *File Systems*). No bloco Principal estarão sendo atendida as interrupções de software e a gerada pela memória dupla-porta (que permite a comunicação entre os blocos).

Como percebe-se, as chamadas de sistema são a interface com a qual os processos interagem com o núcleo. Estas agora deverão ser encaminhadas ao bloco Secundário via memória dupla-porta, assim como as respostas a estas chamadas. Assim, neste caso, ao invés de a mesma CPU da aplicação executar o serviço solicitado, esta será liberada para executar outro processo que esteja pronto, antes porém, um acesso a escrita em memória dupla-porta para a montagem da mensagem destinada ao bloco Secundário se fará necessária antes da troca de tarefa.

Neste primeiro passo, para o código do bloco Secundário, é necessário realizar mudanças na maneira de como as chamadas de sistema são atendidas quando efetuadas pelo processo no nível de usuário, pois nesta arquitetura ela chegará via mensagem na memória dupla-porta. E também a maneira de como a resposta do serviço solicitado é passado ao nível Principal, também deverá ser alterado. Porém o código que efetivamente realiza o serviço solicitado continuará inalterado, uma vez que este continuará fazendo parte do bloco Secundário.

A atividade de escalonamento, pertencente ao núcleo, também deverá sofrer alterações. No sistema convencional, quando o algoritmo de escalonamento é chamado, o

---

[1] A instrução “trap” é utilizada para causar uma interrupção de software

contexto do processo escolhido para ser executado pela CPU e selecionado e o controle passa a ser em nível de usuário. Na arquitetura proposta, o resultado do escalonamento deve ser comunicado ao bloco Principal via mensagem colocada na memória dupla-porta, e neste bloco será então realizado o chaveamento de contexto para que a tarefa selecionada seja executada.

Para o bloco Principal, as alterações no código necessária se concentram na maneira de como as chamadas de sistema são realizadas e de como a resposta a estas são recebidas. Também a decisão do chaveamento de contexto das tarefas para a troca do processo a ser executado pela CPU é agora motivado por uma mensagem recebida do escalonador no bloco Secundário.

Para uma melhor exemplificação das modificações realizadas no sistema operacional Linux, a Figura 6.5 apresenta um diagrama com os componentes que sofrerão modificações para serem colocados no bloco Principal. As partes que sofrerão modificação com o particionamento estão destacadas.

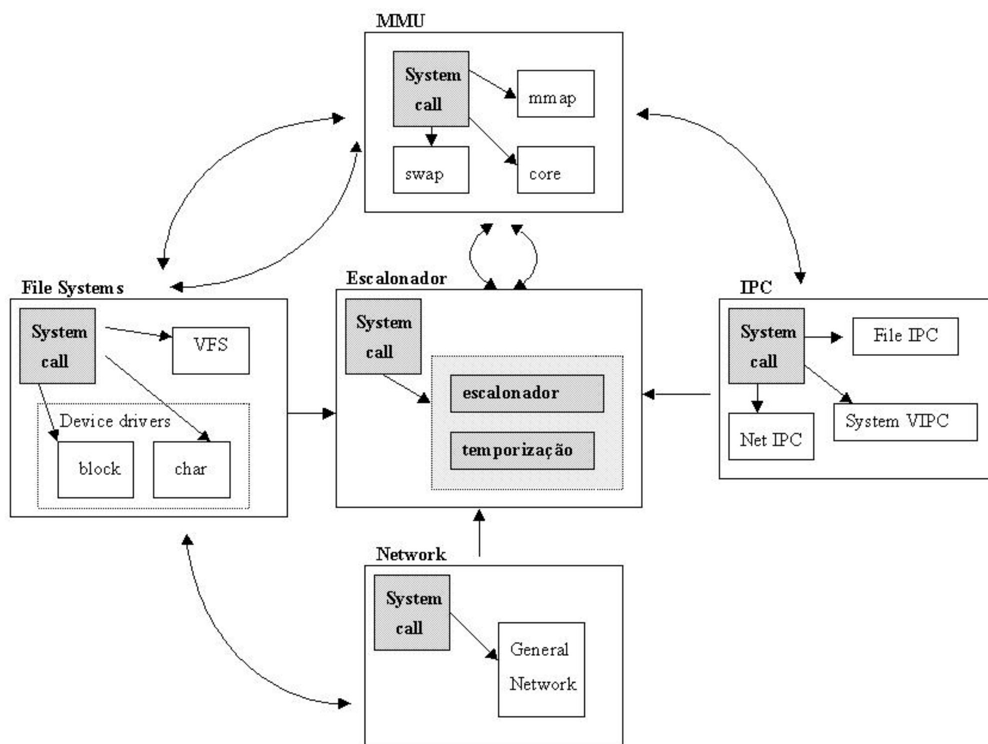


Figura 6.5 Diagrama com os vários subsistemas Linux

Com o particionamento do núcleo estabelecido, a primeira etapa do desenvolvimento encontra-se realizada. O trabalho seguinte concentra-se em realizar as modificações no sistema para que ele tenha características tempo-real. Como já mencionado em análises anteriores, as idéias elaboradas em RED-Linux são as mais adequadas a serem adotadas para a arquitetura proposta.

Uma das causas da imprevisibilidade de tempo de execução das tarefas e na latência da ativação das tarefas no sistema Linux é o fato de o núcleo ser não-preemptivo quando executando alguma chamada de sistema. Usualmente, quando há uma interrupção do *timer*, o escalonador é chamado para possibilitar que outra função ocupe a CPU. Porém, quando executando alguma chamada de sistema, esta chamada é agendada e somente quando o núcleo terminar esta atividade, ou então quando ele próprio liberar a CPU, o escalonamento será chamado. Isto acarreta tempos de latência imprevisíveis nas respostas das tarefas.

Esta característica no Linux é uma das maneiras de permitir a sincronização das suas atividades, evitando o acesso simultâneo a regiões críticas do código do núcleo. Estes pedaços de código que são não-preemptivos definem a chamada resolução do núcleo. Levando a situação para o pior caso, tem-se que a resolução do núcleo é determinada pela região crítica que possui o maior tempo de execução. Para um sistema tempo-real, isto causa muita imprevisibilidade, uma vez que existem chamadas de sistema bastante rápidas na sua execução (quando ajustam a prioridade de algum processo, por exemplo) e outras maiores (quando, por exemplo, realizam a leitura de um conjunto muito grande de dados por exemplo). A figura 6.6 mostra um exemplo em que tais situações são comprometedoras para o caso de sistemas tempo-real:

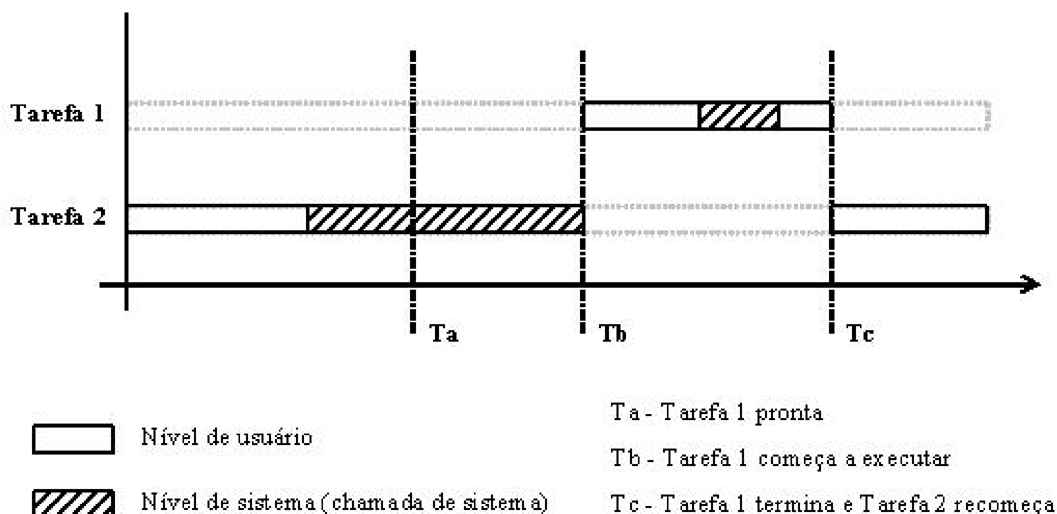


Figura 6.6 Atraso no atendimento de uma tarefa em Linux

Em RED-Linux, a solução adotada é a análise do núcleo para que sejam reduzidas as regiões críticas do núcleo, ou seja, as rotinas de serviço do núcleo devem ser rescritas para que sejam executadas em pequenos blocos. Após a execução de cada um destes pequenos blocos, o núcleo avalia se há a necessidade de executar o algoritmo de escalonamento (se assim sinalizada através de alguma interrupção de *timer*), e então interrompe a execução do

serviço e realiza a atividade pendente. Estes pontos de "quebra" dos blocos de execução são chamados, em RED-Linux, de pontos de preempção. A Figura 6.7 exemplifica a situação anterior usando-se os pontos de preempção:

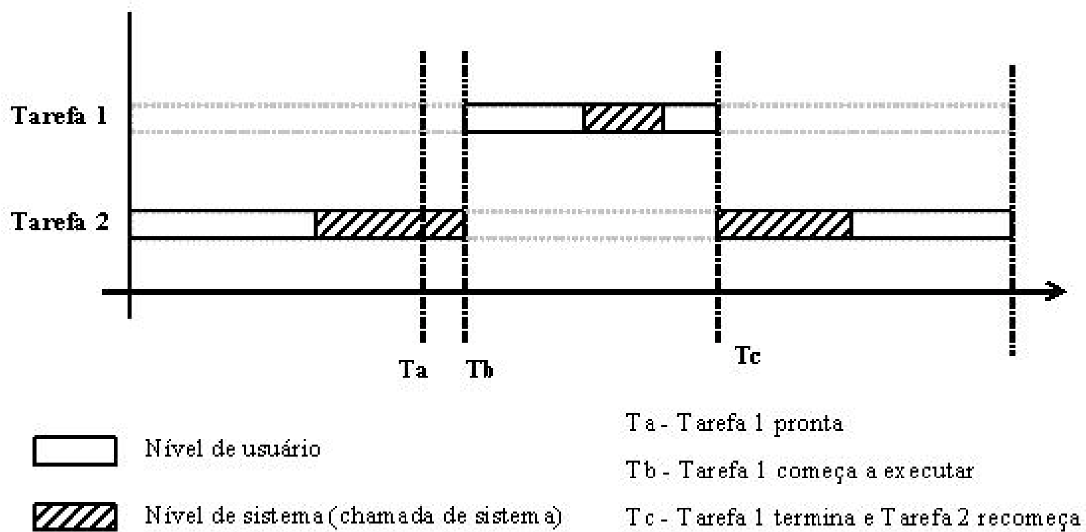


Figura 6.7 Redução do tempo de atendimento em RED-Linux

A arquitetura proposta traz uma grande vantagem neste aspecto pois minimiza este efeito. O bloco Principal pode trocar o processo em execução, iniciando outro que seja mais prioritário, e no bloco Secundário o serviço atual continuar a ser executado. O algoritmo de escalonamento não está mais, neste caso, impedido de executar quando na execução de uma região crítica do núcleo, pelo simples fato de as tarefas da aplicação não executarem na mesma CPU do sistema operacional. O resultado do algoritmo de escalonamento será enviado, via mensagem, para o bloco Principal que efetuará a troca de contexto e de tarefa, enquanto que o bloco Secundário poderá continuar a execução das atividades do núcleo.

Em uma etapa mais avançada da implementação pode-se avaliar procedimentos de execução destas atividades do sistema operacional de acordo com a prioridade da tarefa solicitante, a fim de que as tarefas de maior prioridade sejam atendidas preferencialmente.

Porém, podem existir situações em que o efeito seja percebido, mesmo que de maneira diferente. Se o bloco Principal trocar a tarefa em execução por outra de maior prioridade (a pedido do escalonador do bloco Secundário), e inicialmente esta tarefa requisitar algum serviço do núcleo, este ficará esperando até que o atual serviço do núcleo termine a sua atividade (ou então libere ele mesmo a execução) e então comece a executar a atividade solicitada pela tarefa de maior prioridade. Se o serviço em execução for muito longo, a tarefa mais prioritária levará um tempo maior para finalizar, causado pelo serviço de núcleo

solicitada por uma tarefa de menor prioridade. Esta situação causa indeterminismo no sistema e caracteriza uma situação de inversão de prioridade

Como colocado anteriormente, estes efeitos são atenuados na arquitetura proposta, pela divisão entre atividades do sistema operacional e aplicação. Porém, por outro lado, com a resolução temporal do sistema aumentada, os tempos de execução das atividades do sistema operacional começam a ser relevantes.

Assim, as rotinas do sistema operacional uClinux devem ser estudadas para sejam avaliados os tempos de execução dos blocos de execução não-preemptivos do código, verificando se estes não comprometem a previsibilidade de execução do sistema. Para tanto, a mesma orientação utilizada em RED-Linux pode ser utilizada.

Outra maneira que o sistema Linux utiliza para manter o sincronismo no núcleo é a de desabilitar as interrupções de hardware quando no acesso às regiões críticas. Isto evita que, quando o código do atendimento das interrupções possuir acesso a regiões críticas, este não seja feito no mesmo instante em que algum serviço de núcleo o estiver fazendo.

Para um sistema tempo-real esta situação não é desejável, uma vez que isto implica em tempos de latência imprevisíveis no atendimento das interrupções. Como solução, em RED-Linux é adotada a técnica apresentada em [BAR01], que consistem em realizar a emulação de interrupções via software. Quando ocorre alguma interrupção, o sistema simplesmente memoriza este evento e imediatamente retorna de onde foi interrompido, sendo o atendimento à interrupção colocado em uma fila e processado mais adiante. Isto permite também gerenciar o problema de atendimento de múltiplas interrupções, quando estas ocorrem praticamente ao mesmo tempo. Por exemplo, se houver a necessidade de que uma tarefa tempo-real retome a sua atividade na aplicação, esta pode ser efetuada antes de se executar o processamento das interrupções.

Para a arquitetura proposta, isto também se faz necessário. Com o escalonamento sendo executado no mesmo bloco em que as interrupções são processadas, a situação em que várias interrupções possam acontecer simultaneamente, é possível. Com a solução adotada, o escalonador pode avaliar a prioridade das interrupções para decidir em que ordem deverão ser executadas, e também avaliar se há a necessidade de enviar uma mensagem ao bloco Principal antes de efetuar os atendimentos às interrupções.

Outro aspecto avaliado em [WAN01b], está relacionado à granularidade do *timer* adotada pelo sistema Linux padrão, que adota um *tick* de 10ms. Para um grande número de

aplicações este intervalo de tempo não é suficiente. Com a arquitetura proposta, a idéia de aumentar a resolução temporal para o sistema fica facilitada. Enquanto que em um sistema convencional, o aumento da resolução de um *timer* periódico implica em aumentar o *overhead* de processamento, inclusive para a aplicação (pois usa-se uma única CPU), na arquitetura proposta este *overhead* no processamento deverá ser suportado apenas pelo bloco Secundário, onde é feito o gerenciamento da temporização

7 IMPLEMENTAÇÃO

Neste capítulo será apresentada a realização da separação das atividades funcionais do sistema operacional uClinux e de como ressaltar as suas características tempo-real.

Em virtude de várias dificuldades na execução do hardware proposto em capítulo anterior, dificuldades estas decorrentes principalmente da oferta de alguns componentes (as memórias) no mercado nacional e internacional, os estudos de particionamento apresentado são relativos a uma arquitetura de hardware provisória utilizada para realização de testes de implementação.

7.1 PROPOSTA ORIGINAL

Analisando-se o sistema tradicional, a CPU pode estar executando algum processo do usuário, estando assim em nível de usuário, ou então uma atividade do núcleo, estando assim a nível de núcleo. Sempre que a CPU estiver a nível de usuário, esta trocará de nível, quando houver alguma interrupção (sinal assíncrono, *timer*, *device driver*), ou quando houver alguma chamada de sistema pelo processo do usuário implementada pela instrução *trap*.

Percebe-se então que a ligação entre a camada de usuário e o sistema operacional é realizado através de chamadas de sistema. Se o núcleo não se encontra fisicamente alocado na mesma CPU onde são executadas as tarefas da aplicação, as chamadas de sistema devem ser implementadas através de um canal de comunicação com o núcleo. Este é o caso encontrado na presente arquitetura, que oferece uma memória dupla-porta como canal de comunicação.

Outro ponto de ligação que existe entre a camada de usuário e a de núcleo, está relacionada à atividade de escalonamento. Quando o algoritmo de escalonamento decide que é o momento de haver a troca de uma tarefa por outra, este deverá enviar uma mensagem ao bloco Principal para que coloque a tarefa atual em estado de espera, realize a troca de contexto para a tarefa que assumirá a CPU, e execute a próxima tarefa.

Para se aproveitar a idéia da implementação das chamadas de sistema pela instrução *trap*, em vez do bloco principal ser totalmente nível de usuário, uma pequena camada de nível de sistema deverá restar, para que o atendimento da interrupção de software gerada pelas chamadas de sistema seja atendida.

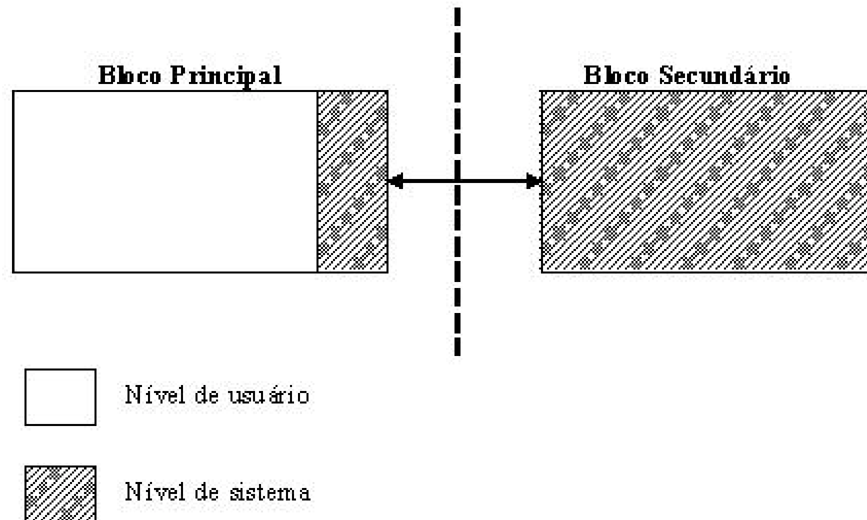


Figura 7.1 Abrangência dos níveis de sistema e de usuário

7.1.1 Bloco principal

Trazendo esta idéia para o hardware aqui proposto, temos que, quando a aplicação necessitar realizar alguma chamada de sistema, esta colocará uma mensagem na memória dupla-porta para ser lida no bloco secundário (que fará o processamento do serviço solicitado), colocará também a tarefa solicitante em estado de espera por serviço (WAIT_SERV) e aguardará pelo envio de uma mensagem do bloco Secundário especificando qual a próxima tarefa escolhida pelo algoritmo de escalonamento que deverá ocupar a CPU.

Analisando-se a forma de como esta atividade está implementada em uma arquitetura convencional, verifica-se que todo o seu código está escrita em linguagem de baixo nível (*assembly*). O código encontrado em <linux/include/asm-68knommu/unistd.h> mostra a implementação da primeira parte que prepara a chamada da rotina. Os argumentos de entrada para a realização da chamada, bem como o número que identifica a chamada, são colocados em registradores específicos e então a instrução "*trap*" é realizada, gerando assim uma interrupção de software.

A segunda parte do código está escrita em <linux/arch/m68knommu/entry.S>, onde é realizada a rotina de interrupção de software. Neste momento, este código já está sendo realizado no nível de sistema, ocasionado pela interrupção. São feitos alguns salvamentos de registradores, e a rotina correspondente ao número da chamada de sistema é então chamada. Esta rotina por sua vez, já é escrita em uma linguagem C e encontra-se no diretório correspondente ao módulo que se relaciona.

Para a implementação das chamadas na arquitetura proposta, pode-se aproveitar as rotinas criadas em *assembly* e modificar somente as rotinas escritas em C que encontram-se em seu específico diretório. A modificação consiste em substituir o código existente (o qual executa efetivamente o serviço) por outro que escreva a mensagem na memória dupla-porta para o bloco Secundário com todos os argumentos necessários para daí então executá-la.

Cada chamada de sistema deve ser avaliada individualmente, pois cada uma possui um conjunto único de parâmetros de entrada. E uma atenção especial deve ser dada àquelas rotinas que usam como parâmetros ponteiros para estruturas de dados que são utilizados, por exemplo, na escrita destes. Como o ponteiro faz referência a uma região de memória alocada no bloco Principal, este não fará sentido nenhum para o bloco Secundário. Ao invés de se enviar então o ponteiro como parâmetro, envia-se o conjunto de dados necessário. Aqui deve se fazer uma avaliação do tamanho máximo deste conjunto de dados, uma vez que será realizada uma cópia dos dados encontrados no nível de usuário para o nível de sistema (escritos na memória dupla-porta).

Para a leitura/escrita de grandes blocos de dados, que ultrapassem a capacidade reservada para tal na memória dupla-porta, ou que comprometa a temporização do sistema, o processo que está requisitando tal atividade deverá executar a transferência em etapas.

Como padrão para o envio da mensagem de requisição de serviço do sistema operacional para o bloco Secundário, foi escolhido o seguinte:

BUFF - Número da região da memória DP
ID - Identificador do processo, PID
NR - Número que identifica a chamada de sistema
DATA - Campo de dados, argumentos da chamada de sistema

Figura 7.2 Formato da mensagem de uma chamada de sistema

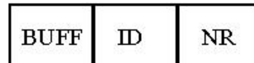
Como as mensagens são enviadas de forma bidirecional, do bloco Principal para o Secundário e vice-versa, e considerando-se que haverá a possibilidade de mais de uma mensagem a ser processada ao mesmo tempo (quando, por exemplo, a tarefa atual em execução envia um pedido de serviço para o bloco Primário sem que o anterior tenha sido processado), deverá haver um meio de sincronismo do uso da memória. Como solução foi feito o uso dos semáforos de hardware disponibilizados pela própria memória dupla-porta. Como são em número de 8, optou-se por dividir a memória em um número igual de regiões de mesmo tamanho:2K (16K / 8 sem.).

Para avisar o bloco Secundário que há uma mensagem nova colocada, é usado um recurso da memória dupla-porta utilizada que provê a geração de interrupção de uma lado para outro, quando acessado um endereço específico (um para cada lado da memória). Através deste endereço, responsável pela geração da interrupção, é utilizado o seu conteúdo para informar qual a região da memória em que a mensagem chegou.

A seqüência de envio é então a seguinte: verifica-se qual a região de memória da dupla-porta está livre (liberada pelo semáforo correspondente), sinaliza-se a ocupação desta no seu semáforo, preenche-se a região com os dados relativos à chamada de sistema, realiza-se a geração de interrupção através da escrita no endereço específico (juntamente com o índice referente à região de memória que possui a mensagem), e então libera-se o seu semáforo para permitir que o bloco Principal tenha acesso à mensagem.

Para a implementação das chamadas de sistema, é permitido o uso de 7 das 8 regiões possíveis da memória dupla-porta. A última é destinada à comunicação com o escalonador alocado no bloco Secundário. Atualmente está previsto apenas o envio de mensagens do escalonador para o bloco Principal, e não vice-versa, através deste canal. As mensagens enviadas do bloco Principal, com destino ao módulo de escalonamento, contido no bloco Secundário, será implementada através de novas chamadas de sistema, usando-se para tanto as 7 regiões de memória já mencionadas.

Como exemplo, o bloco Principal deverá informar ao módulo escalonador quando uma tarefa terminou a sua execução, para permitir que este retire esta das tarefas prontas e agende o novo instante de sua ativação, quando esta for periódica. Para tanto a nova chamada de sistema (“*rt_wait()*”) tem o seguinte formato:



BUFF - Número da região da memória DP
ID - Identificador do processo, PID
NR_rt_wait - identificador da chamada *rt_wait*

Figura 7.3 Mensagem do bloco Principal indicando fim de execução de tarefa

Quando o bloco Secundário terminar o processamento do serviço solicitado, este enviará o resultado também por uma mensagem na memória dupla-porta (que possuirá o mesmo formato da mensagem de envio) e, da mesma forma, este fará com que seja gerada uma interrupção (agora no bloco Principal) para o aviso da ocorrência do evento.

Na rotina de atendimento à interrupção da memória dupla-porta, o bloco Principal saberá qual região de memória encontra-se a mensagem recebida pelo índice escrito pelo bloco Secundário quando da geração da interrupção. Assim, a mensagem respectiva será lida da memória dupla-porta para retirar os parâmetros de retorno do serviço solicitado e reativar a tarefa que havia gerado a chamada de sistema.

Fica claro então que, de todo o código do sistema operacional, restará apenas uma pequena camada do núcleo onde as chamadas de sistema ainda estarão sendo efetuadas pela instrução *trap*. Porém, ao invés de executar efetivamente o serviço, um acesso à memória dupla-porta será feito, cuja execução é bem definida temporalmente e que implica em uma previsibilidade de acesso. Também há, nesta pequena camada, o gerenciamento da mensagem aguardada pela tarefa originaria da chamada, para que quando houver a resposta a esta, a tarefa correta seja invocada e os dados possam ser processados corretamente na aplicação.

As interrupções externas (excetuando a relacionada ao acesso a memória dupla-porta) não serão mais dirigidas ao bloco Principa,. pois todo o acesso a dispositivos e gerenciamento de entradas e saídas são realizados no bloco Secundário. Com esta implementação, o bloco Principal estará executando na maior parte do tempo, as tarefas da aplicação. Uma pequena parte do tempo será gasto em: atendimento da interrupção provocada pela memória dupla-porta, o gerenciamento do uso desta e a troca de contexto para as tarefas.

7.1.2 Bloco secundário

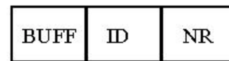
Basicamente todo o sistema operacional estará sendo realizado no bloco Secundário. Uma análise deve ser feita para averiguar quais as partes do núcleo necessitam ser modificadas para a realização da vinculação com o bloco Principal. Duas primeiras modificações são claras, devido às alterações já realizadas no outro bloco: as chamadas de sistema e o escalonamento.

No sistema convencional a chamada de sistema, provinda da tarefa do nível de usuário, realiza uma interrupção de software (comando *trap* para MC68XXX) para tal. Na proposta apresentada, a chamada de sistema também será realizada através de uma interrupção, mas esta originada pela memória dupla-porta. Daí surge a necessidade de modificação desta pequena camada que interpreta a chamada originada pelo bloco Principal.

Como proposto no item anterior, o byte enviado pela memória dupla-porta e que gerou a interrupção, indicará o bloco de memória onde a mensagem enviada está armazenada. Lendo a mensagem, o núcleo executará o serviço e quando estiver pronto, colocará o resultado em uma mensagem para a tarefa que originou o serviço, fazendo esta novamente voltar ao estado de execução.

As interrupções originadas pela memória dupla-porta substituem neste caso as interrupções de software, para as chamadas de sistema. A primeira atividade a ser realizada quando na execução da rotina de atendimento da interrupção (além do salvamento dos registradores obrigatórios) é sinalizar pelo semáforo correspondente a região da mensagem recebida, que agora esta está ocupada. Isto é necessário pois indica que o bloco Secundário não pode usar esta região para originar outro pedido.

Na seqüência o algoritmo de escalonamento é chamado para que escolha a nova tarefa a ser executada no bloco Principal e envia uma mensagem para este indicando qual foi a escolhida. O formato da mensagem é semelhante ao das mensagens utilizadas para a implementação das chamadas de sistema.



BUFF - Número da região da memória DP
ID - Identificador do processo, PID da nova tarefa
NR_rt_start - identificador da chamada

Figura 7.4 Mensagem do escalonador ao bloco Principal para reinicio de execução de tarefa

Após o envio desta mensagem para o bloco Principal, é feita a leitura dos parâmetros de entrada para a realização do serviço solicitado e então a rotina que realiza o serviço no núcleo é chamada. Antes de chamá-la porém, com os parâmetros de entrada todos lidos, a região de memória da qual proveio a mensagem é liberada para o acesso mediante sinalização no seu respectivo semáforo.

Quando o serviço solicitado estiver terminado, o bloco Secundário localiza uma região da memória dupla-porta livre na qual será enviada uma mensagem para a tarefa solicitante do respectivo serviço. Encontrando uma região livre, este bloqueia o seu acesso pelo bloco Principal através da sinalização do semáforo correspondente. É então construída a mensagem com os parâmetros de retorno da função e então sinalizada uma nova mensagem ao bloco Principal através da geração de uma interrupção neste escrevendo em um endereço respectivo o índice da região que contém a mensagem enviada.

O bloco Secundário ficará constantemente realizando os serviços solicitados, dando prioridade aqueles associados às tarefas mais prioritárias. Além disto também estará realizando o controle dos dispositivos através dos *devices drivers*, gerenciando a temporização e o módulo de escalonamento.

O bloco Secundário também enviará mensagens para a troca de tarefa na aplicação quando o algoritmo de escalonamento, baseado nos requisitos temporais das tarefas, decidir que uma tarefa mais prioritária do que a atual, está pronta para a execução.

Em relação ao escalonamento de tarefas, a informação necessária para que uma tarefa retorne à execução ou seja bloqueada (sobre a memória relacionada ao processo), está contida no bloco principal pois neste é efetuado o chaveamento de contexto. Porém no bloco Secundário, também deverá haver um conjunto de informações relacionadas com as tarefas da aplicação. Dentre estas informações estão, a prioridade da tarefa, o estado de execução,

valores relacionados à execução e seus requisitos temporais, tabela de arquivos, sinais e *status*.

Em uma visão mais geral, as modificações descritas até aqui adequam o tradicional sistema operacional executado em uma CPU para uma arquitetura com dois microcontroladores, analisado para um caso específico, o que atinge um dos objetivos iniciais. Outras modificações se fazem necessárias, e estas na grande maioria no bloco Secundário, para incorporar características necessárias a tornar o sistema operacional tempo-real.

Muitas das arquiteturas de software encontradas no sistema Linux poderão ser aproveitadas, como por exemplo, as filas de tarefas bloqueadas "*wait_queue*" (a tarefa é colocada nesta fila quando esperando por algum recurso de hardware ou que alguma atividade de hardware seja terminada). Um subsistema do Linux porém, deverá sofrer muitas alterações: o módulo de escalonamento de processos. Na versão tradicional, quando o escalonamento fosse chamado, o resultado do algoritmo de escalonamento já originava uma troca de contexto e o controle passaria a ser executado no nível de usuário, com a CPU executando a tarefa escolhida.

O enfoque do presente trabalho não é pesquisar tipos de escalonamento, ou implementá-los, mas sim prover meios para que novos algoritmos possam ser implementados em atividades futuras, para daí sim realizar a sua avaliação.

As implementações realizadas para realizar as reduções nos blocos de execução dos serviços do núcleo segue o roteiro implementado em RED-Linux. A diferença encontrada é a de que neste caso se estará trabalhando com um sistema operacional Linux para versões embarcadas. Neste aspecto, esta parte será facilitada, pois como a versão uClinux não possui o módulo de gerenciamento de memória, muitos problemas de atraso relacionados ao *swap* com o disco (encontrado na versão para desktop) não serão mais encontrados.

Como melhoria da granularidade da temporização, optou-se por utilizar-se emum primeiro momento, o módulo de interrupção periódica (interna à CPU) para geração da base de tempo, com um *tick* de 1ms como primeira aproximação. Comparativamente, em um sistema convencional, teríamos uma granularidade 10 vezes maior para a arquitetura proposta (para o uClinux, interrupção periódica de *timer* é de 10ms).

```
ENTRY(s_sti)
        andiw#0xf8ff,%sr
        pea   1f
        movew        %sr,%sp@-

 check_iret:
        /* disalbe interrupt, will be cleared when rte */
        /* make scratch register %d0,%d1 */
        oriw   #0x0700,%sr

        /* check if pending interrupt, SFIF is not
cleared yet */
        movel%d0,%sp@-
        movelSYMBOL_NAME(SFREQ),%d0
        andl   SYMBOL_NAME(SFMASK),%d0
        jeq    2f

        movel#0, SYMBOL_NAME(SFIF)
        andiw#0xf8ff,%sr
        bra    SYMBOL_NAME(SF_inthandler)

 1:     rts

 2:     movel#1, SYMBOL_NAME(SFIF)
        andiw#0xf8ff,%sr
        movel%sp@+,%d0
        rte
```

Figura 7.5 Código para STI na interrupção emulada

Para a implementação da emulação de software das interrupções, como mostrado em [BAR01], as rotinas "*sti()*" e "*cli()*" que realizam a habilitação e a desabilitação das interrupções no microcontrolador respectivamente, foram trocadas pelas macros *s_sti()* e *s_cli()*. A Figura 7.5 mostra como esta macro foi escrita para substituir a atual *sti()*. Foram criadas três novas variáveis para a realização da simulação: SFIF, SFREQ e SFMASK. A primeira, SFIF, que habilita ou desabilita as habilitações (habilitação geral), sinaliza para quando chegar a interrupção, esta seja atendida prontamente ou então seja memorizado o evento em SFREQ. Quando ocorrer a função *s_sti()*, a variável SFREQ é mascarada com SFMASK (habilitação individual das interrupções) e então se houver alguma atividade pendente esta será atendida.

Sempre que houver algum pedido para desabilitar as interrupções, através de *s_cli()*, e então ocorrer alguma interrupção, este evento será registrado para que, quando as interrupções forem novamente habilitadas, todas as rotinas de tratamento correspondentes possam então ser efetuadas. Além disto, se entre estas interrupções houver alguma que tenha

alta prioridade de atendimento (como uma mensagem do bloco Principal), esta pode ser avaliada e atendida para que o seu *deadline* não seja ultrapassado.

7.2 ARQUITETURA PARA TESTES

Em função da não disponibilidade do hardware projetado, buscou-se então alternativas para a implementação das idéias até aqui colocadas.

Devido ao fato de encontrar-se no laboratório apenas um hardware microcontrolado, usando o processador 68332, conhecido como MEGA332 (utilizado inclusive no trabalho apresentado em [BRU00]), e na inviabilidade de se encontrar outra placa semelhante, adotou-se a utilização de um software (chamado de *xcopilot*) para PC, ambiente Linux, responsável pela emulação de um terminal PalmTop, o qual executa uma versão uClinux com porte para o microcontrolador 68328. Entre estes dois blocos (PC x MEGA332) há disponível apenas um canal de comunicação: USART. Esta configuração permite o desenvolvimento de experimentos com as implementações/modificações do sistema operacional uClinux, tendo, entretanto, como maior restrição o comportamento temporal na comunicação entre os dois blocos (originalmente via memória dupla-porta e no caso em questão via canal serial).

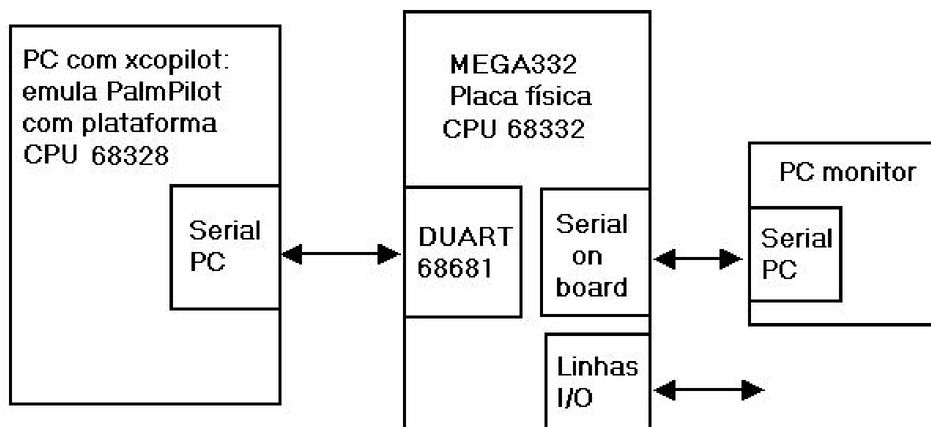


Figura 7.6 Hardware disponível

Desta parte do texto em diante, adotar-se-á a nomenclatura de bloco virtual e bloco físico, para a distinção entre o bloco emulado em PC e o bloco alocado na placa MEGA332, respectivamente.

Devido a esta variação no hardware disponível, modificações no particionamento do sistema foram avaliadas para que fosse possível a obtenção de alguns resultados práticos.

Para tanto, algumas considerações para com o software emulado no PC, ou bloco virtual se fazem necessárias:

- Este não efetua a emulação correta dos tempos de execução das instruções em um 68328, o que descaracteriza alguma tentativa de medição de tempos de latência neste bloco virtual;
- Existe apenas um canal serial físico disponível, que "mapeia" o canal virtual USART interno ao 68328, o que impede que seja usado o "*printk()*" como forma de *debug* nos teste;
- Não há disponível pinos físicos de I/O para a realização de testes.

7.2.1 Alternativa 1: Aplicação emulada

Visando manter a coerência com a proposta inicial, objetivou-se manter apenas um dos blocos responsáveis pela execução das tarefas e o outro dedicado às atividades do núcleo. Como o bloco responsável pelo escalonamento deve ter um relógio preciso, necessitando assim que o processador associado tenha um módulo de temporização, optou-se pelo uso da MEGA332 como bloco Secundário, e o PC como bloco Primário.

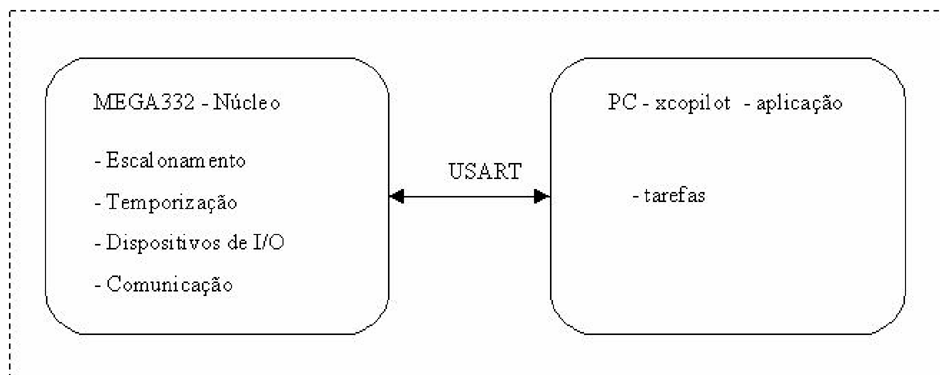


Figura 7.7 Diagrama em blocos da Alternativa I

Nesta situação também é possível analisar-se a adaptação do núcleo para um sistema tempo-real, pois como o trabalho do porte já é de conhecimento (realizado em [BRU00]), o trabalho torna-se menos árduo. Porém devido às condições de funcionamento do bloco virtual mencionadas acima, não temos condições de avaliar o desempenho das tarefas. Ou seja, não há como mensurá-las através de sinais de I/O e nem mesmo com *time-stamps.* Sendo assim, não seria útil a sua implementação e esta não será utilizada.

7.2.2 Alternativa 2: Núcleo emulado

Uma segunda tentativa de configuração de uso do hardware disponível é a de trocar o uso das funcionalidades dos blocos proposta no item anterior. Agora então o bloco virtual faria a implementação da maior parte do núcleo (bloco Secundário), e a MEGA332 responsável pela aplicação (bloco Principal). Com o bloco físico executando as tarefas, é possível usar sinais de I/O, juntamente com um osciloscópio, e realizar a mensuração na prática do desempenho do bloco Principal.

A grande dificuldade nesta versão de configuração é a de que o bloco secundário, responsável pelo escalonamento entre outras atividades, não possui uma temporização precisa, além de não possuir um módulo com temporizadores como na CPU 68332. Porém permite testar o conceito de que outro agente externo coordene, ou gerencie, a ordem de execução das tarefas defendido nesta dissertação.

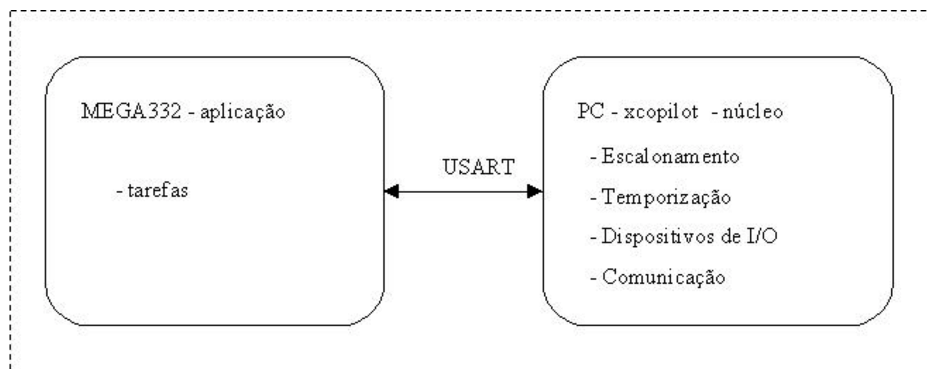


Figura 7.8 Diagrama em blocos da Alternativa II

Por ser esta configuração a que mais se assemelha com o proposto nesta dissertação, e por permitir que sejam realizadas algumas medições práticas no bloco Principal, esta arquitetura alternativa foi a adotada para a realização dos testes.

7.2.3 Alternativa 3: Emulação Mista

As configuração sugeridas nos dois itens anteriores não permitem o uso de chamadas de sistema, pois acarretaria em uma sobrecarga no canal serial disponível entre os dois blocos. Para que seja possível o estudo de tarefas realizando chamadas de sistema, e para não haver queda na performance do sistema (através do canal de comunicação), os serviços devem estar implementados no mesmo bloco onde as tarefas se encontram. Assim sendo, restaria para o bloco Secundário apenas a atividade do escalonamento.

Apesar de permitir que, para o hardware disponível, as tarefas possam realizar chamadas de sistema, esta configuração não vai de encontro a um dos objetivos apresentados

nesta dissertação. As tarefas da aplicação compartilham a CPU com a execução de boa parte do sistema operacional. E mesmo que o algoritmo de escalonamento seja efetuado em um bloco separado, o poder de processamento da CPU, disponível para a aplicação, é diminuído.

Outra implementação ainda possível seria, permanecendo este particionamento, alocar o bloco Principal no PC e o bloco Secundário na MEGA332, permitindo um escalonamento com um relógio mais preciso neste último. Porém, além de manter a contradição com a proposta da dissertação, não permitiria uma avaliação do desempenho das tarefas, uma vez que estas seriam executadas no bloco virtual (PC).

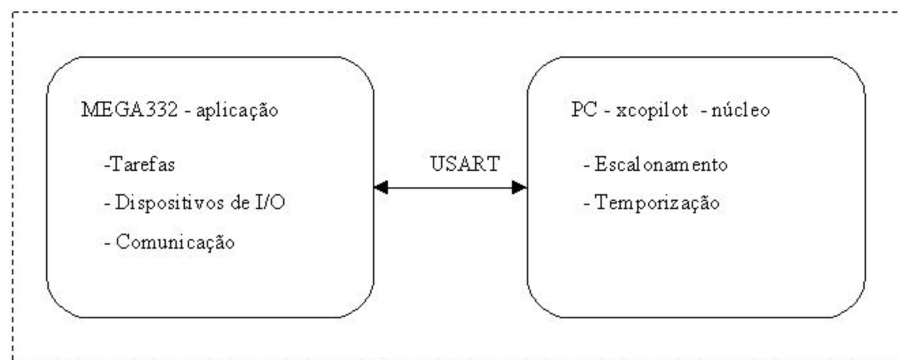


Figura 7.9 Diagrama em blocos da Alternativa III

Devido ao exposto acima, esta arquitetura não será implementada na prática por não permitir a avaliação de resultados conclusivos.

8 RESULTADOS OBTIDOS

Visando a avaliação de algumas idéias, implementou-se um dos particionamentos sugeridos. Usou-se então a segunda sugestão, Núcleo Emulado, proposta no capítulo anterior, em que o bloco Principal idealizado foi implementado na MEGA332 e o bloco secundário no *xcopilot*, executado no PC. Esta foi usada pois permite que as tarefas sejam avaliadas utilizando-se pinos de I/O disponíveis no hardware para a mensuração através de um osciloscópio.

Devido às condições limitadas de hardware disponível para a realização dos testes, não é possível se utilizar de estudos de casos mais elaborados, que seriam ideais, pois permitiriam avaliar todas as situações propostas. Foram utilizados então tarefas periódicas, as quais atuam em pinos de I/O, os quais permitem que sejam realizadas mensurações (com a ajuda de um osciloscópio digital). Com estas tarefas, pode-se avaliar, basicamente, a periodicidade e *jitter*, e comparar os comportamentos do sistema operacional com e sem características tempo-real.

Outras avaliações em relação ao núcleo do sistema foram também avaliadas, como o tempo gasto com o "chaveamento de contexto" das tarefas. Esta medida tem influência direta na definição da resolução temporal a ser adotada no ambiente tempo-real, pois esta medida deverá ser considerada quando avaliado o tempo de latência na ativação das tarefas.

8.1 AMBIENTE DE REALIZAÇÃO DOS TESTES

O software emulador do sistema uCLinux para *PalmTop*, o *xcopilot*, está ainda com problemas na versão atual em relação ao mapeamento do canal serial disponível para a USART do PC. Não é possível realizar leituras através deste canal, o que tem sido constatado nas implementações dos testes aqui mencionados e no grupo de discussão sobre o uClinux [UCL01] na *internet*. Desta maneira não foi possível a implementação das chamadas de sistema para o núcleo (emulado).

Para o bloco Secundário utilizou-se então, do núcleo do sistema, apenas o módulo de gerenciamento do temporizador e o bloco escalonador, para gerar os pedidos de ativação das tarefas no bloco Principal. A camada de aplicação foi retirada.

No bloco Principal (Placa MEGA332), foi mantida a idéia utilizada em RED-Linux da existência de tarefas tempo-real e não tempo-real, permanecendo assim o escalonamento tradicional neste bloco para estas últimas. Também foram mantidas as funcionalidades do núcleo necessárias para manter os processos *shell* e *kbflush*, necessários para que fossem incluídas/canceladas outras tarefas para aumentar/diminuir a carga computacional do sistema. Porém os recursos utilizados pelas duas tarefas mencionadas, só serão utilizados no momento da inclusão/retirada das tarefas de carga. Em estado de regime, onde serão realizadas as medidas, estas não estarão ativas. As tarefas não tempo-real só serão executadas a partir do momento em que nenhuma tarefa tempo-real estiver ocupando a CPU e até que alguma tarefa tempo-real tiver de retomar a CPU (através de pedido do escalonador no bloco Secundário).

Com relação ao ciclo de vida das tarefas tempo-real, estas foram consideradas estáticas. Estas são criadas pelo próprio bloco Primário, assinaladas como tempo-real e permanecem assim durante a existência do sistema. No bloco Secundário, o escalonamento já considera estas tarefas como existentes e conhece os seus requisitos temporais. A ligação lógica existente entre os dois blocos para referenciar as tarefas entre estes é feita através de um índice (um identificador diferente do ID) atribuído a cada tarefa tempo-real.

Para permitir que as tarefas da aplicação tivessem acesso aos registros da CPU responsáveis pela configuração dos pinos de I/O (sendo esta atitude somente permitida no modo protegido, conhecido como nível de sistema), foi utilizada a mesma estrutura para geração de chamadas de sistema da arquitetura convencional. Para a sua implementação, foi escolhida uma chamada de sistema que não havia implementação no uClinux: "*ioperm*", a qual foi modificada e utilizada para o desejado fim (Figura 8.1).

Nas tarefas tempo-real, para indicar a finalização da atividade da tarefa e devolver o controle da CPU para o escalonador, foi utilizada a chamada de sistema "*pause*". Esta rotina coloca a tarefa corrente em estado de espera e realiza a chamada ao algoritmo de escalonamento. O seu código pode ser observado na Figura 8.2.

```
asmlinkage int sys_ioperm (unsigned char from,
                           unsigned char num, int on)

main ()
{
unsigned char i;
unsigned char byte = 0x01;


for (i=0;  i<num; i++) {
byte*=2;
}

if (on == 1) {
PORTQS |= byte;
}

else {
PORTQS &= (~byte);
}

return 0;
}
```

Figura 8.1 Chamada de sistema para acionamento de I/O

```
asmlinkage int sys_pause(void)
{
        current->state =
TASK_INTERRUPTIBLE;
        schedule();
        return -ERESTARTNOHAND;
}
```

Figura 8.2 Chamada de sistema "pause()", utilizada para sinalização fim de execução da tarefa tempo-real

As tarefas tempo-real utilizadas nos testes descritos, são todas periódicas, não possuem relação entre si, e possuem tempos de execução bastante rápidos a ponto de o seu valor não ser relevante para o escalonamento. Isto resulta em um algoritmo de escalonamento (utilizado pelo bloco secundário) bastante simples, e com o qual não serão realizados testes ou avaliações, pois não é alvo desta dissertação o estudo de tipos de algoritmos de escalonamento, mas sim prover uma arquitetura que facilite a implementação destes.

Em todos os testes realizados, estarão sempre presentes duas tarefas (não tempo-real) necessárias para a interação com o sistema pelo usuário: *shell* e *kbflush* Estas duas

tarefas agregam um custo computacional muito pequeno ao sistema e, como mencionado anteriormente, durante a mensuração não terão basicamente nenhuma atividade.

Os resultados foram obtidos a partir do bloco principal utilizando a placa MEGA332 (CPU 68332) operando com uma velocidade (*clock*) de 16MHz e o canal serial de comunicação entre os blocos a uma velocidade de 38400bps (máxima velocidade conseguida pelo emulador *xcopilot*).

8.2 ARQUITETURA TRADICIONAL

Com a intenção de realizar comparações posteriores e enfatizar a dificuldade de utilização de um sistema operacional como o uClinux em ambientes tempo-real, consideremos a existência de uma simples tarefa (Figura 8.4) que faça oscilar um pino de I/O na saída da MEGA332 (PORTQS,3). Se somente esta tarefa estiver sendo executada (excetuando as duas já mencionadas anteriormente), conseguiremos uma razoável estabilidade em relação ao período observado na saída do pino. Quando outras tarefas (Figura 8.3) são adicionadas ao sistema, com a função de "carregar" o sistema, nota-se um crescente distúrbio nos tempos mensurados no pino controlado pela primeira tarefa.

Para avaliarmos outros dados em relação à arquitetura convencional, foram medidos também alguns tempos relacionados com a execução de alguns blocos funcionais do núcleo do uClinux, a troca de contexto e a duração do escalonamento tradicional. Estes foram medidos através do tempo de ativação de uma saída física (pino de I/O) associada ao bloco alvo e com a ajuda de um osciloscópio digital. O escalonamento depende, dentre outros fatores, do número de tarefas ativas no sistema.

	Somente *shell*	1 tarefa de I/O	+5 tarefas carga	+10 tarefas carga
Período médio*	-	1,69 ms	1,69 ms	1,69 ms
Jitter	-	430µs	650µs	820µs
Tarefa ativa	-	100%	220ms(18,2%)	208ms(9,6%)
Tarefa inativa	-	0%	990ms(81,8%)	1960ms(90,4%)
Escalonamento	70µs	102µs	138us	180µs

Tabela 8.1 Medidas realizadas em ambiente não Tempo-Real

* Medido durante o intervalo "Tarefa ativa", com a ajuda do osciloscópio digital.

Outra medida realizada foi o tempo gasto para a realização da troca de contexto, o qual se manteve constante nas medidas:

- Troca de contexto: 95µs

```
main ()
{
int i;

while (1) {
        for (i = 0; i < 100; i++);
        }
}
```

Figura 8.3 Tarefa para "carregar" o sistema

```
#define        DELAY  1000

main ()
{
int i;

while (1) {
        for (i = 0;  i < DELAY;  i++);      // gera atraso
        ioperm (0, 4, 1);                   // coloca pino em "1"

        for (i = 0;  i< DELAY;  i++);       // gera atraso
        ioperm(0, 4, 0);                    // coloca pino em "0"

        }
}
```

Figura 8.4 Tarefa não tempo-real para acionamento cíclico

8.3 TAREFAS TEMPO-REAL

Para a avaliação de uso de tarefas tempo-real no bloco principal, foram criadas duas destas. Uma associada ao pino QS.1 com periodicidade de 2ms e outra com periodicidade de 4ms associada ao pino QS.3. Cada uma delas, quando ativada, troca o estado do pino relacionado e entra em estado de espera pela próxima ativação. Assim que a tarefa

executa a sua atividade e entra em estado de espera, o controle é passado ao escalonador do para a escolha de uma outra tarefa a ser executada.

Os resultados abaixo relacionados, foram feitos primeiramente com somente as duas tarefas tempo-real e após foram sendo incluídas outras tarefas para aumentar a carga computacional do sistema (mesma atitude adotada no teste anterior). Neste caso observou-se que as tarefas tempo-real praticamente não foram afetadas pelo incremento da carga computacional do sistema, mantendo as suas características temporais.

	2 tarefas tempo-real	+10 tarefas de carga
Período T1	2ms	2ms
Jitter T1	140µs - 280µs	130µs- 270µs
Período T2	4ms	4ms
Jitter T2	120µs – 200µs	120µs – 210µs

Tabela 8.2 Medidas realizadas em ambiente Tempo-Real

```
main ()
{
while (1) {
       ioperm(0, 1, 1);     // coloca pino em "1"
       pause();             // libera CPU

       ioperm(0, 1, 0);     // coloca pino em "0"
       pause();             // libera CPU
       }
}
```

Figura 8.5 Tarefa tempo-real para acionamento cíclico

Outros valores importantes de serem avaliados são os tempos associados a ativação das tarefas. Para se ativar uma tarefa tempo-real, a partir da chegada de um pedido pelo escalonador do bloco secundário, existem três tempos encadeados: Latência no atendimento da interrupção quando na chegada da mensagem, a procura da tarefa associada a mensagem chegada e também o tempo da troca de contexto.

Este último permanece o mesmo pois não é afetado pelas modificações realizadas.

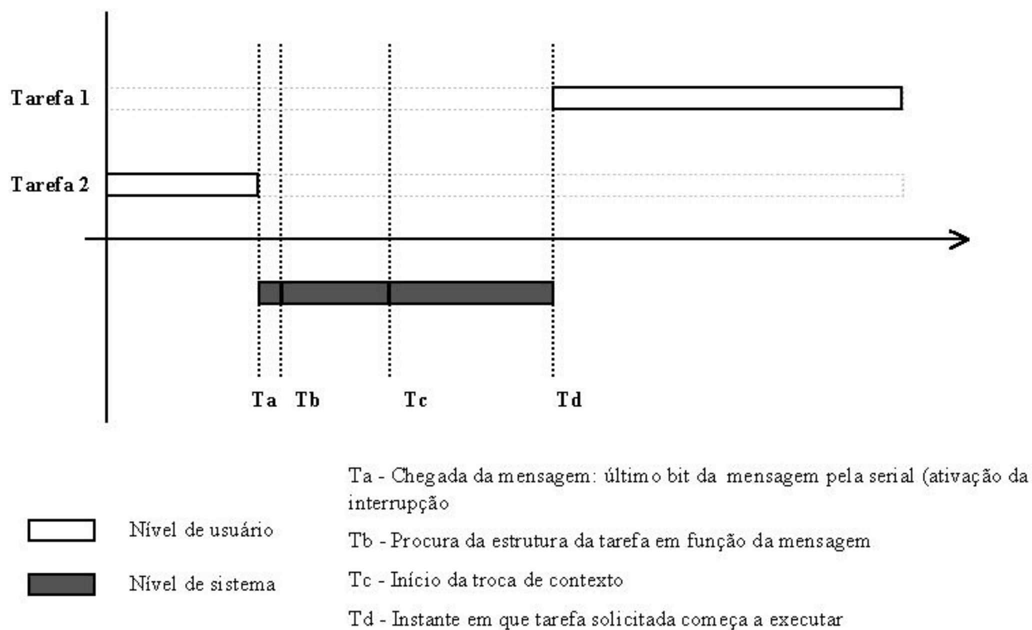


Figura 8.6 Tempos associados a ativação de tarefa

- Latência (englobando o tratamento da interrupção): 8,4µs ~ 11,6µs
- Procura da tarefa: 41,5µs
- Troca de contexto: 95µs

A medição do tempo de latência ocorre desde o instante em que o byte chega totalmente (via serial) para o bloco primário e o código de tratamento da interrupção é atendido. A procura da tarefa, ocorre desde o instante em que, com a referência à tarefa a ser chamada, a sua estrutura dentro de uma fila é procurada até antes de haver a troca de contexto.

Somando-se estes três valores, tem-se o tempo total entre a chegada da mensagem do escalonador e o instante em que a tarefa associada é iniciada: 148µs no pior caso.

Outra consideração que deve ser feita é a de que a decisão do algoritmo de escalonamento no bloco secundário está atrasada em pelo menos 300µs, aproximadamente, visto que este é o tempo que um byte leva para chegar ao bloco principal, via comunicação serial, utilizando uma velocidade de 38400bps.

8.4 AVALIAÇÃO DOS RESULTADOS OBTIDOS

Para a arquitetura não tempo-real, a utilizada pelo uClinux, a inclusão de mais tarefas ativas faz com que o tempo de execução do algoritmo de escalonamento começasse a

aumentar, ocupando a CPU com mais processamento que não o da aplicação (aumento do tempo de escalonamento, por exemplo). Levando-se em conta todas as restrições impostas nos testes e por não se tratar de um sistema tempo-real, estes resultados não são catastróficos.

Ainda se observa que, neste ambiente, há instantes consideravelmente grandes em que a tarefa deixa de ser executada (da ordem de segundos), observados pela tarefa na qual havia um pino de I/O associado. Isto se deve basicamente ao algoritmo de escalonamento utilizado pelo Linux (*time-sharing*) que tem como objetivo compartilhar o uso da CPU pelos processos existentes proporcionando um tempo médio para estes.

Um ponto que pode ser estudado, para avaliar a redução do tempo de troca de contexto, que permitiria um aumento do tempo de resposta na ativação das tarefas. Como neste bloco (primário) não haverá mais tanta informação associada à tarefa que necessita ser salva quando na troca por outra, este tempo pode ser reduzido.

Nestes ensaios, fica evidente que, se não houvesse o escalonamento presente no mesmo bloco onde está havendo a execução das tarefas, algoritmos bastante aprimorados podem ser elaborados sem praticamente acarretar sobrecarga para a aplicação. Na arquitetura convencional, quanto mais tarefas estiverem sendo executadas, maior o tempo de escalonamento e mais difícil a manutenção do determinismo temporal, especialmente se chamadas de sistema estiverem sendo realizadas por estas tarefas.

9 CONCLUSÕES E TRABALHOS FUTUROS

Uma arquitetura de hardware de baixo custo capaz de suportar um sistema operacional e particioná-lo a fim de proporcionar características como determinismo temporal, comportamento previsível (indispensáveis em um sistema tempo-real) e o uso de um sistema operacional confiável, com código aberto (também implicando em um baixo custo) foram apresentadas nesta dissertação.

Os testes realizados na prática mostraram que o software utilizado pode ser utilizado para a proposta apresentada, e capaz de suportar as idéias apresentadas em RED-Linux quanto a redução dos blocos de execução do núcleo do uClinux para ressaltar as suas características tempo-real. Mostra ainda o tempo gasto pela CPU apenas com o processamento do algoritmo de escalonamento funcionalidade esta que ficaria associada ao bloco secundário, e não o da aplicação.

O estudo realizado mostra também que o uso de um canal serial para a comunicação entre os dois módulos "estrangula" a capacidade do sistema. Ainda mais considerando-se que por este canal uma razoável quantidade de dados poderá trafegar, especialmente quando houver a leitura ou escrita de um dispositivo de entrada/saída. E mesmo com o uso de uma memória dupla-porta na sua implementação, um cuidado especial se deve ter no momento de se escrever o código de acesso e comunicação entre os dois blocos, para que este não se torne uma grande limitação na velocidade do sistema.

A utilização de um sistema operacional já estável e confiável e que está em constante evolução, e que tenha disponível o seu código fonte facilita em muito o desenvolvimento, permitindo um grande ganho de tempo. Associado a isto, a sua ampla documentação e estudos realizados neste ambiente possibilitam um aprofundamento do conhecimento do seu funcionamento. Se forem utilizadas outros processadores, já existe o porte para outras plataformas como o i960 da Intel ou a linha Coldfire da Motorola, componentes com maior capacidade de processamento.

9.1 TRABALHOS FUTUROS

Como seqüência aos trabalhos desenvolvidos tem-se a confecção da placa de circuito impresso proposta nesta dissertação. O seu diagrama eletrônico está realizado no programa Protel99, e pode assim que definidos os "*packages*" das memórias a serem utilizadas, ser realizado o roteamento no mesmo aplicativo e então confeccionar a placa de circuito impresso. Um cuidado deve haver quanto às memórias FLASH, pois deve-se prever um adaptador que facilite o seu manejo no processo de gravação, ou então prover um sistema de gravação *in-circuit*, o que necessitaria de um programa monitor (*firmware*) residente no primeiro bloco da memória que gerenciaria o processo de gravação através de uma interface com o PC.

A nível de software deve-se aproveitar a filosofia de redução dos blocos de execução do núcleo do uClinux, tomando-se como base a implementação realizada no Linux pelo REDLinux (propostas no capítulo 5), uma vez que o desenvolvimento de outro núcleo totalmente preemptivo seria em demasiado oneroso e abandonaria a idéia de uso de um software já confiável. O desenvolvimento das idéias de implementação apresentadas no mesmo capítulo, relacionadas a interface entre os dois blocos definidos: Principal e Secundário.

E, como conseqüência destes trabalhos, a criação de um ambiente que facilite o desenvolvimento das aplicações e a sua integração com ferramentas desenvolvidas pelo grupo de pesquisa do GCAR, assim como colocado em [BRU00], possibilitando assim que poderosas ferramentas de desenvolvimento que incorporem o estado da arte em termos de sistemas tempo-real sejam utilizadas.

REFERÊNCIAS BIBLIOGRÁFICAS

[BAR 01] BARABANOV, M; Yodaiken, V. **A Real Time Linux**. URL: http://www.rtlinux.org/documents/papers/usenix.pdf., 2001

[BAR 97a] BARABANOV, M. **A Linux Based Real-Time Operating Systems**. Socorro, New Mexico, 1997. Dissertação de Mestrado, New Mexico Institute of Mining and Technology

[BAR 97b] BARABANOV, N. **A Linux-based Real-Time Operating System**. M.S. Thesis, June 1997. URL: http://www.rtlinux.org/documents/papers/thesis.ps

[BAY 96] BARABANOV, M; Yodaiken, V. **Real Time Linux**. Linux Journal. March, 1996

[BEC 99] BECKER, L. B. **Ambiente de modelagem e implementação de sistemas tempo-real usando o paradigma de orientação a objetos**. Porto Alegre:CPGCC da UFRGS, 1999. Dissertação de Mestrado.

[BRU 00] BRUDNA, Cristiano. **Desenvolvimento de Sistemas de Automação Industrial Baseado em Objetos Distribuídos e Barramento CAN**. Porto Alegre: CGPEE da UFRGS, 2000. Dissertação de Mestrado

[DOE 98] BECK, M.; BÖHME, h.; DZIADZKA, M.; KUNITZ, U.; MAGNUS, R.; VERWORNER, D. **Linux Kernel Internals**. 2.ed., Addison Wesley, 1998, 480p.

[ELL 94] ELLISON, K. S. **Developing Real-time Embedded Software in a Market-driven Company**. 1.ed., Wiley, 1994, 351p.

[EPP 97] EPPLIN, Jerry **Linux as an Embedded Operating System**. Embedded Systems Programming, October, 1997

[HAL 92] HALANG, W. A.; Colnaric, M. **Architectural Support for Predictability in Hard Real Time Systems**. Control Engineering Practice, 1 (1993).

[HAL 94] HALANG, W. A.; Colnaric, M.; Tol R. M.; **A Hardware Supported Operating System Kernel for Embedded Hard Real Time Applications**. Microprocessors and Microsystems, 18 (1994).

[KIR 88] KIRNER, C.; Mendes, S.B.T. **Sistemas Operacionais Distribuídos**. Editora Campos LTDA, 1988, 165p.

[LAP 97] LAPLANTE, P. A. **Real-time Systems Design and Analysis: an engineer's handbook**. 2.ed., IEEE PRESS, 1997, 360p.

[LAW 92] LAWSON, H. W. **Parallel Processing in Industrial Real-Time Applications**, Pertince Hall, 1992.

[LIN 01a] LINUXCE. **Linux ce**. URL: http://www.linuxce.org, 2001.

[LIN 01b] LINEO. **Embedix sdk**. URL: http://www.lineo.com, 2001.

[LTD 01] LTDA, Q. R. S. **Qnx neutrino**. URL: http://www.qnx.com, 2001.

[LUN 00] LUNDQVIST, Kristina. **Distributed Computing and Safety Critical Systems in Ada**. University Printers, Uppsala 2000

[MIL 92] MILENKOVIC, M. **Operating Systems**, McGraw-Hill, second edition, 1992

[MOT 93] MC68332 **Users's Manual**, @Motorola, Inc, 1993

[MOT 96] MC68336/376 **Users's Manual**, @Motorola, Inc, 1996

[NIE 92] NIEHAUS, D.; Nahum, E. M.; Stankovic, J. A., Ramamritham, K**.; Architecture and OS Support for Predictable Real-Time Systems**. 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 1992.

[PON 98] PONTREMOLI, M. M. B. **Arquitetura de Hardware de Baixo Custo para Sistemas Tempo-Real Distribuídos**. Porto Alegre, 1998. 105p. Dissertação

de Mestrado. PPGEMM, UFRGS.

[RAM 89] RAMMAMRITHAM, K.; Stankovic, J. A. Zhao, W. **Distributed scheduling of Tasks with Deadlines and Resource Requirements**, IEEE Transactions on Computers, Vol. 38, No.8, 1989.

[RTL 00] RTLINUX. **Real-time linux**. URL: http://www.rtlinux.org, 2001.

[RTL 01a] RTAI. Real-Time Application Interface. URL: http://www.rtai.org, 2001.

[RUB 98] RUBINI, A. **Linux Device Drivers**. O'Reilly & associates, Inc., first edition, 1998.

[SAL 88] SALKIND, L. **UNIX for Real-time Control: problems and solutions.** Tech Report 400, Robotics Report 171, New York University, Tech. Report, September 1988.

[SIL 98] SILBERSCHATZ, A.; GALVIN, P. B. **Operating System Concepts**. 5ª ed., Addison-Wesley, 1998, 888p.

[SRI 95] SRINIVASAN, B. **A Firm Real-time System Implementation using Commercial Off-The-Self Hardware and Free Software**. Kansas, USA, 1995. 66p. Dissertação de Mestrado, 1995. University of Kansas

[SRL 01] SRL, P. Etlinux. URL: http://www.prosa.it/etlinux, 2001

[STA 88] STANKOVIC, J. **Misconceptions about real-time computing: a seius problem for next-generation sytems.** IEEE Computers, v.21, n.10, October 1988.

[STA 90] STANKOVIC, J.; RAMAMRITHAN, K. **What is Predictability for Real-Time Systems?.** J. Real-time Systems, Vol 2, December 1990.

[STA 92] STANKOVIC, J.; Niehaus, D.; Ramamritham, K.; **SpringNet: A Scalable Architecture for High Performance Predictable, and Distributed Real-Time Computing**. Workshop on Architectural Aspects of Real-Time Systems, Dec. 1992.

[STA 00] STANKOVIC, J. VEST: **A Toolset for Constructing and Analyzing Component Base Operating System For Embedded and Real Time Systems**. Technical Report, Computer Science Departament of Virginia University, July, 2000.

[TAN 92] TANEBAUM, A. S. **Modern Operating Sytems,**. Prentice Hall International, New Jersey, 1992.

[TOS 00] TOSCANI, S.S.; Oliveira, R.S.l; Carissimi, A.S. **Sistemas Operacionais II**, Sagra Luzzato, 2000, 150p.

[UCL 01] UCLINUX. **The linux/microcontroller project**. URL: http://www.uClinux.org, 2001.

[VIS 01] VISTA, M. **Hard hat linux**. URL: http://www.mvista.com, 2001.

[WAN 01a] WANG, Y; LIN, K. **Implementing a General Real-time Sheduling Framework in the RED-Linux Real-time Kernel**. RTSS Real Time Systems Symposium. 2001.

[WAN 01b] WANG, Y; LIN, K. **Enchancing the Real-time Capability of the Linux Kernel**. RTSS Real Time Systems Symposium, 1998.

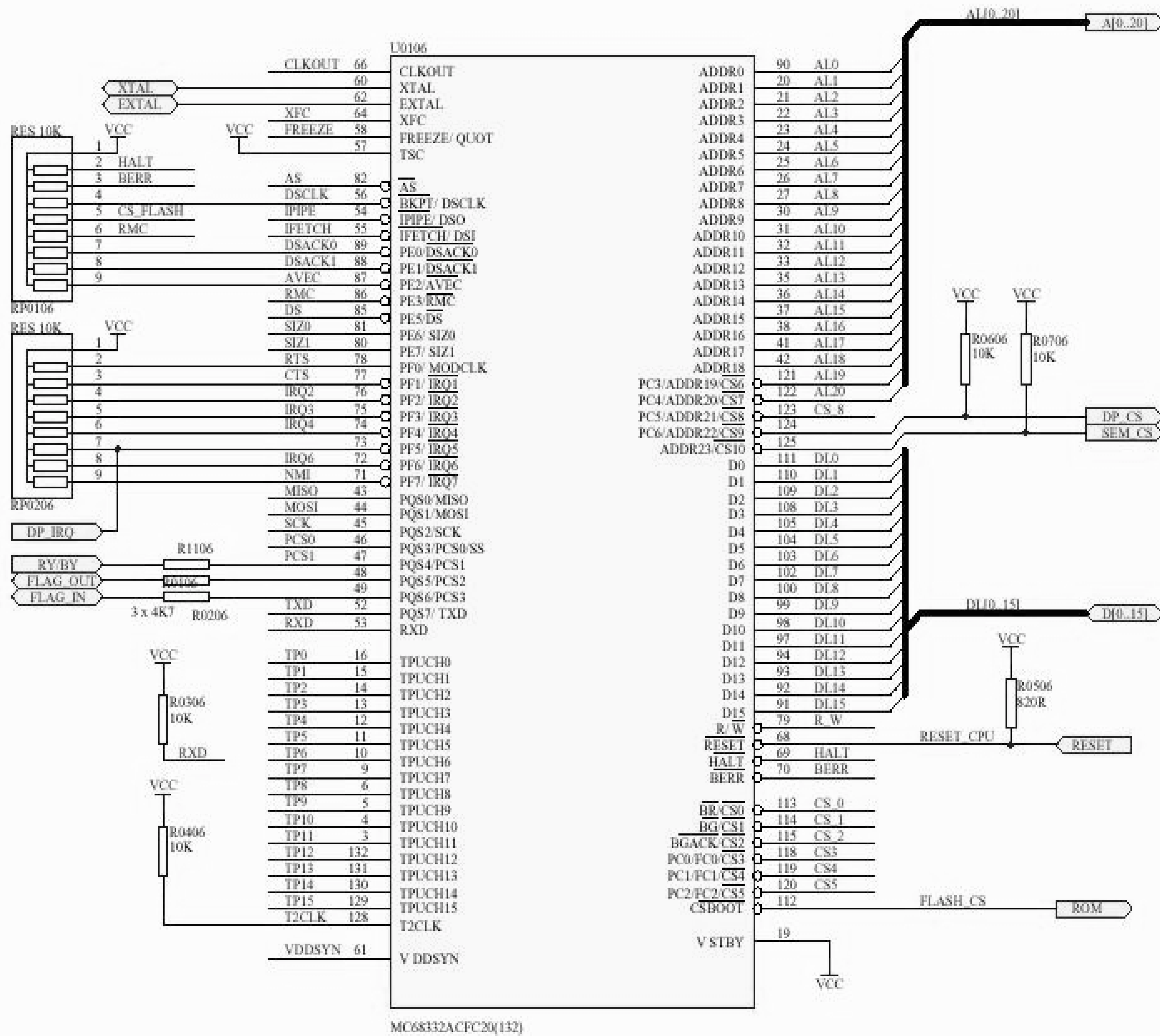
[WOR 01] WORKS, L. **The Blue Cat Linux** . URL: http://www.lynx,com, 2001.

[YOD 97] YODAIKEN, V. **The RT-Linux approach to hard Real-Time**. URL: http://www.rtlinux.org/documents/papers/whitepaper.html, October 1997.

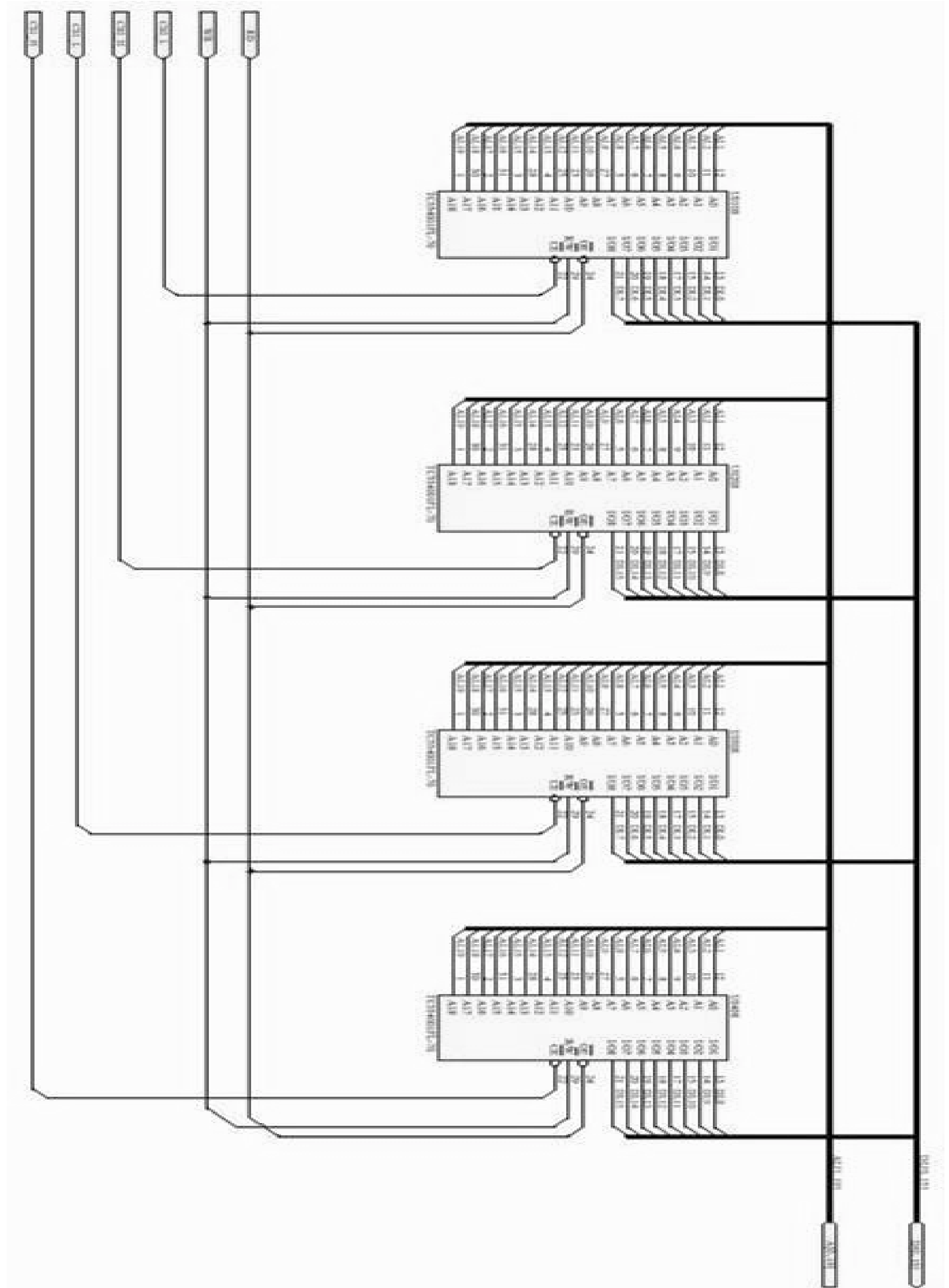
ANEXOS

Em anexo são apresentados os esquemas eletrônicos do Hardware proposto nesta dissertação nas suas partes mais relevantes.

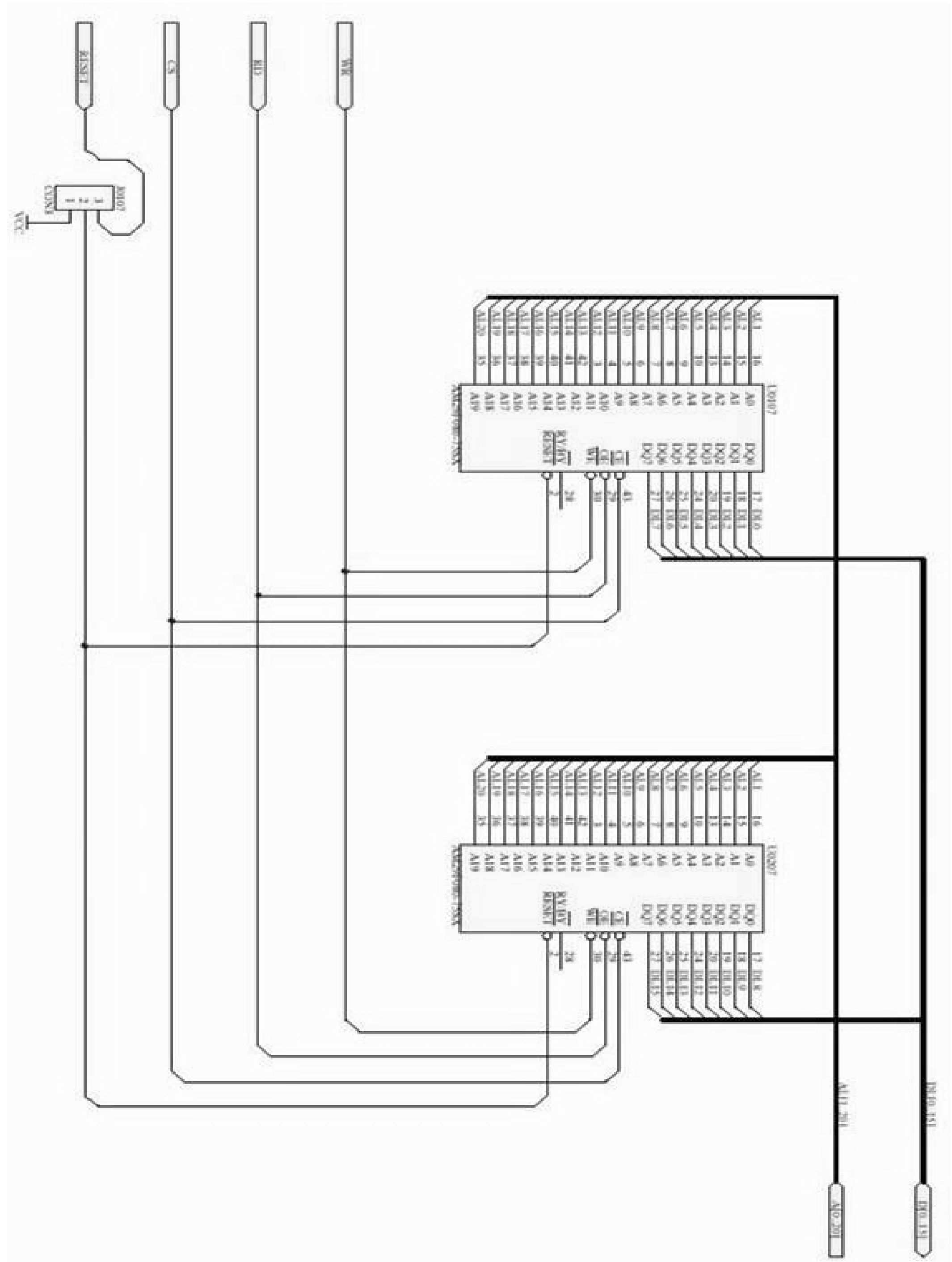
ANEXO 1 - CPU bloco principal



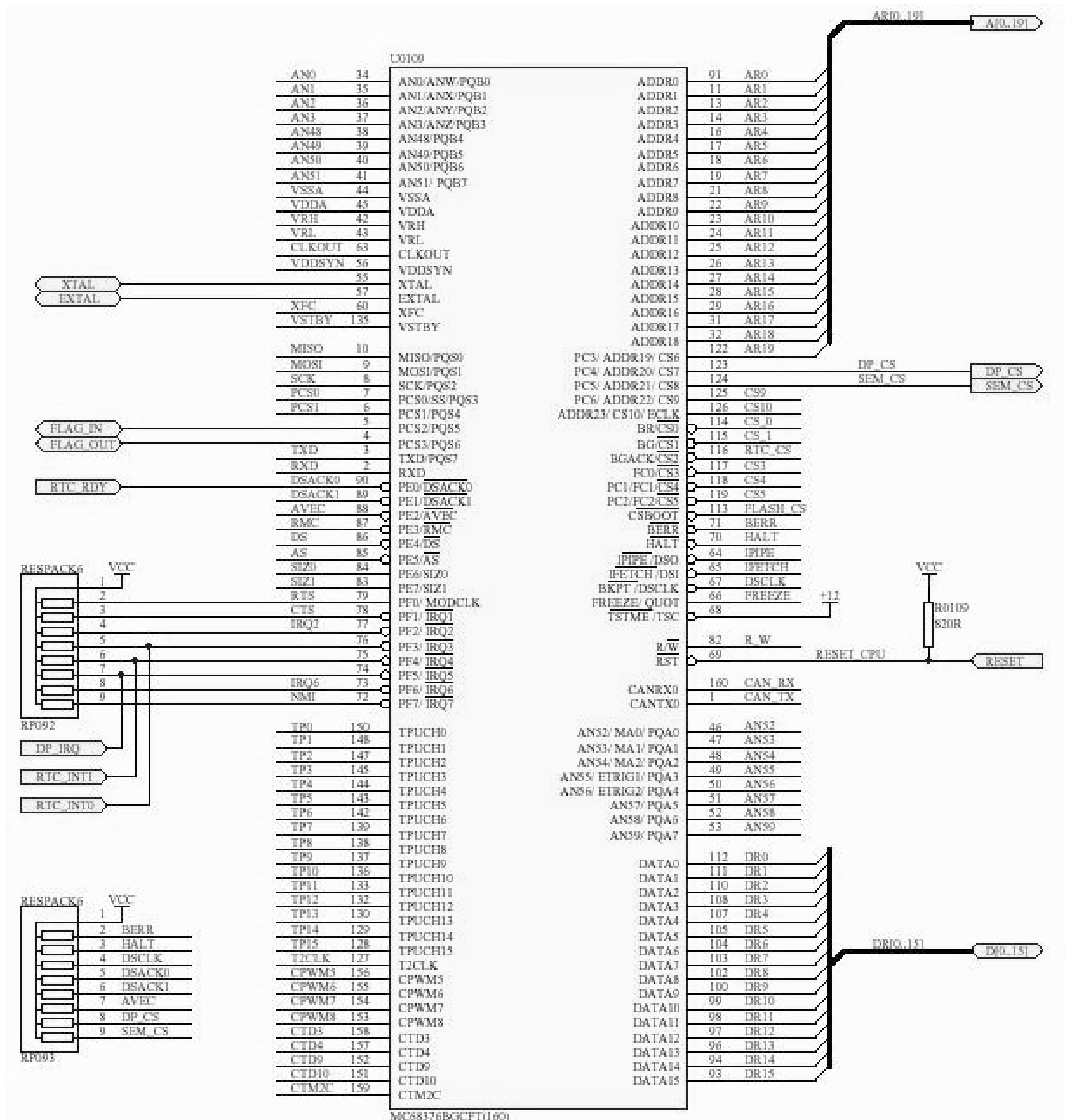
ANEXO 2 – Memória RAM bloco principal



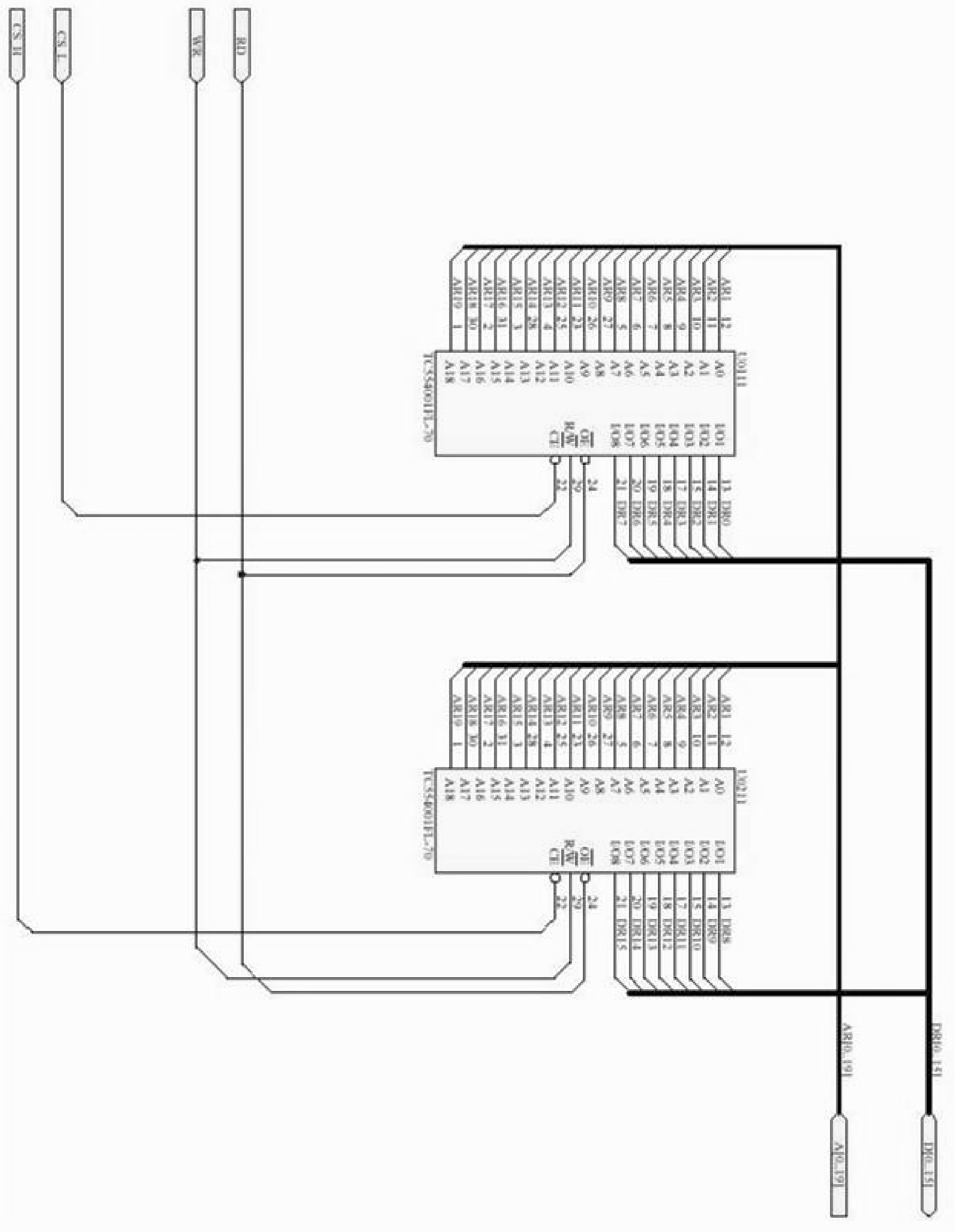
ANEXO 3 – Memória FLASH bloco principal



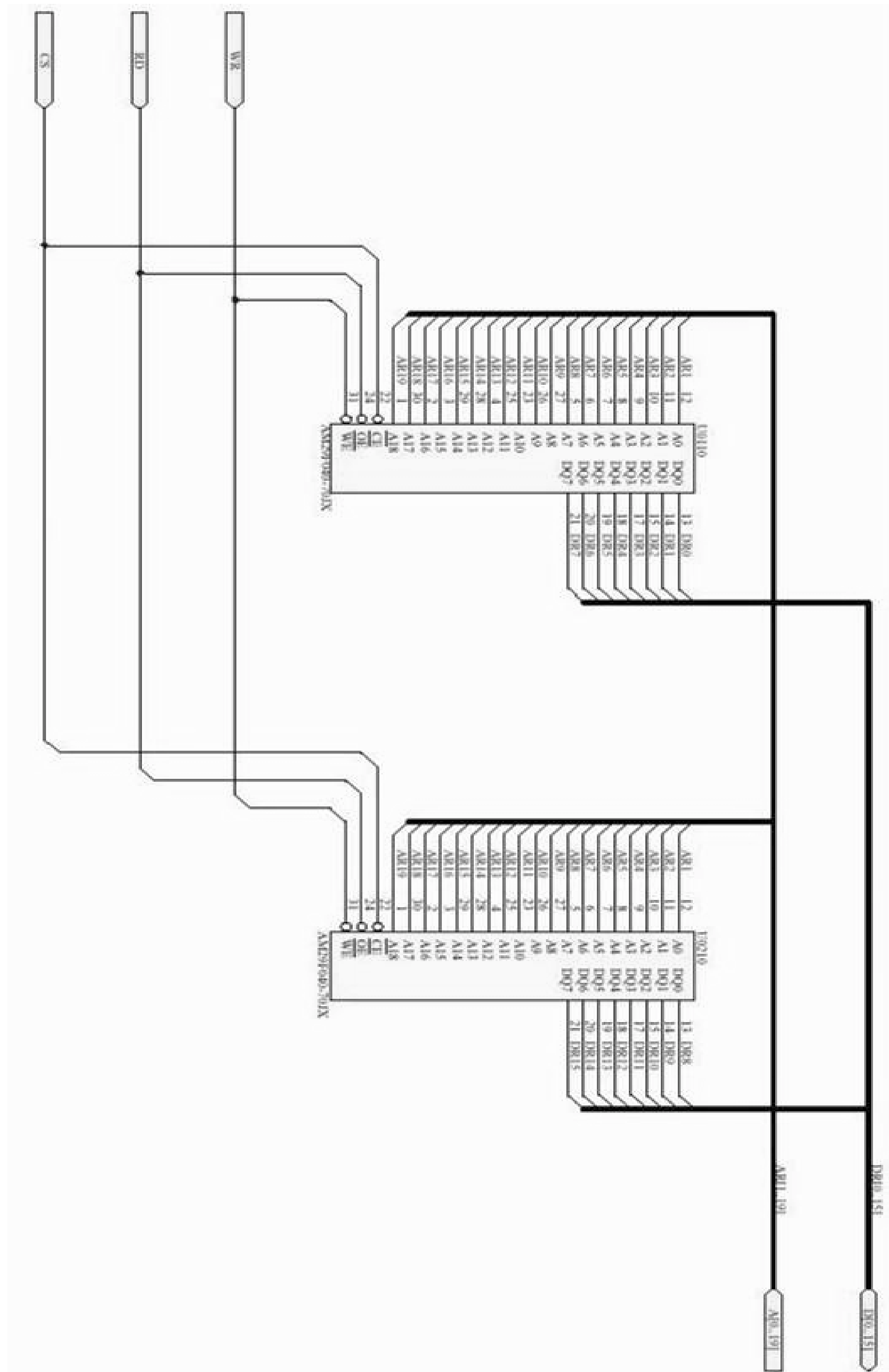
ANEXO 4 - CPU bloco secundário



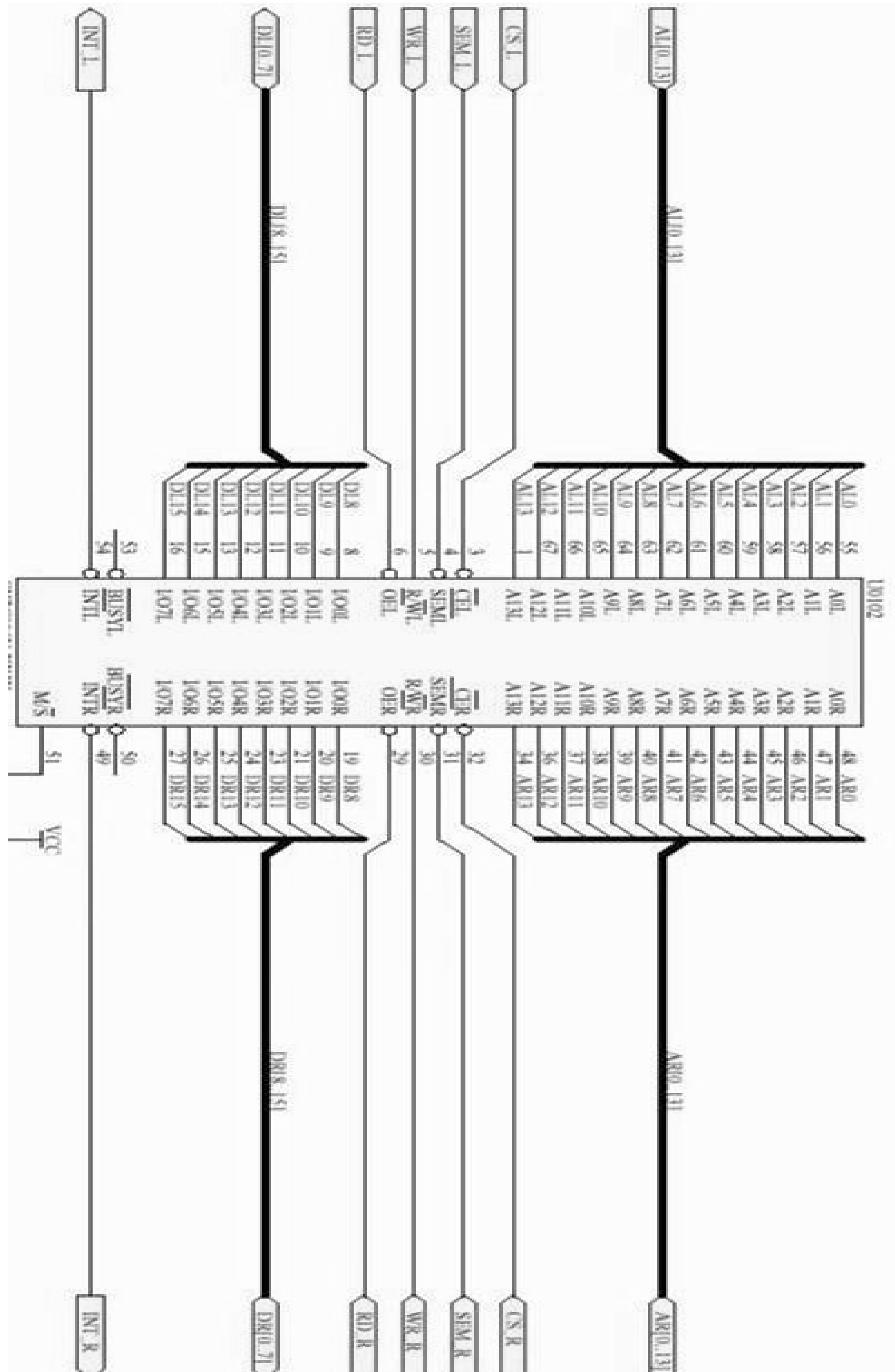
ANEXO 5 - Memória RAM bloco secundário



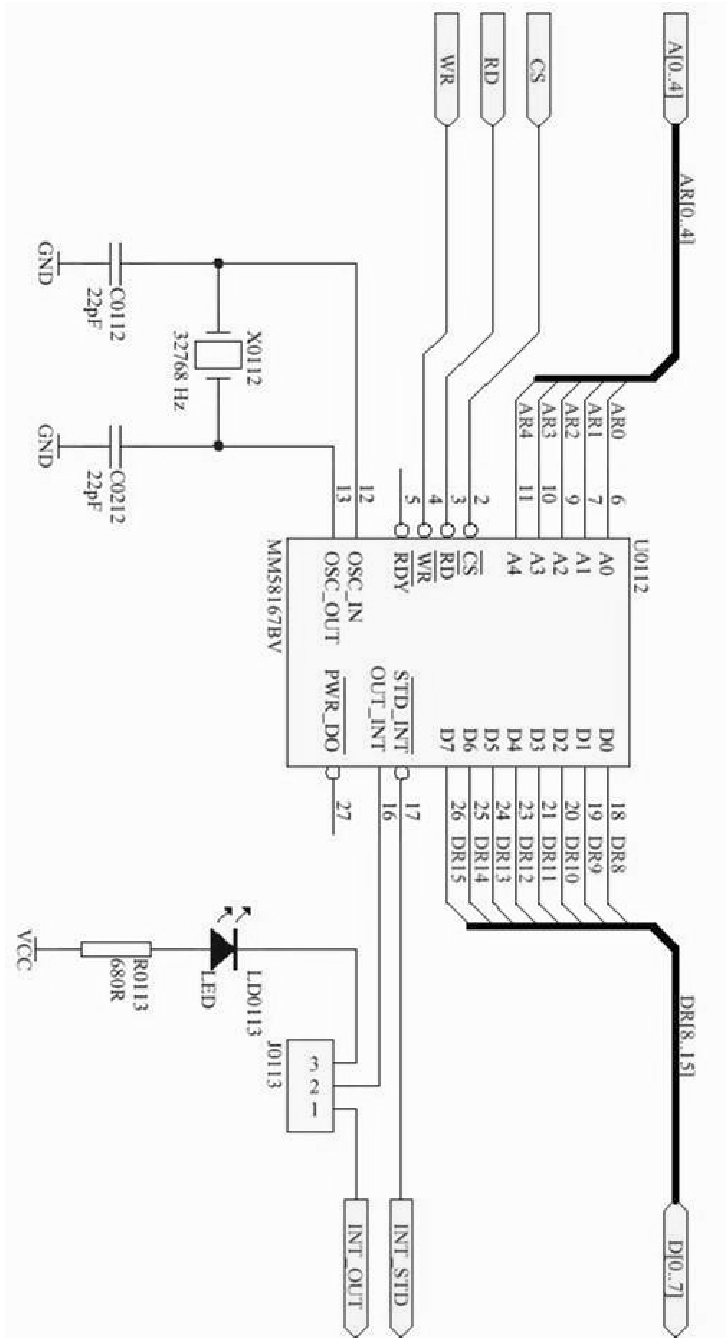
ANEXO 6 - Memória FLASH bloco principal



ANEXO 7 - Memória Dupla Porta



ANEXO 8 – Relógio externo



ANEXO 9 – Diagrama em blocos Geral