SISTEMA DE AQUISIÇÃO DE DADOS EM TEMPO REAL

Ivo Daniel Gomes Correia Machado Alves

Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

2011

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Disciplina de Tese/Dissertação, do 2º ano, do Mestrado em Engenharia Electrotécnica e de Computadores

Candidato: Ivo Daniel Gomes Correia Machado Alves, Nº 1960576, 1960576@isep.ipp.pt

Orientação científica: Professor Doutor José Ricardo Teixeira Puga, jtp@isep.ipp.pt

Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

14 de Novembro de 2011

Agradecimentos

Ao **Professor Doutor José Ricardo Teixeira Puga** pela competência com que orientou a minha tese e pela motivação que me foi dando nas piores fases da tese, pelo tempo que me dedicou e pelos conhecimentos que me transmitiu. Muito obrigado.

Agradeço ao **Laboratório de Metrologia do DEE**, que com a colaboração do **GECAD** me disponibilizaram o acesso ao *hardware*, material e instrumentos necessários à realização da tese.

À Helena, família e amigos. Obrigado a todos!

Resumo

Neste trabalho propus-me realizar um Sistema de Aquisição de Dados em Tempo Real via Porta Paralela. Para atingir com sucesso este objectivo, foi realizado um levantamento bibliográfico sobre sistemas operativos de tempo real, salientando e exemplificando quais foram marcos mais importantes ao longo da sua evolução. Este levantamento permitiu perceber o porquê da proliferação destes sistemas face aos custos que envolvem, em função da sua aplicação, bem como as dificuldades, científicas e tecnológicas, que os investigadores foram tendo, e que foram ultrapassando com sucesso.

Para que Linux se comporte como um sistema de tempo real, é necessário configura-lo e adicionar um *patch*, como por exemplo o RTAI ou ADEOS. Como existem vários tipos de soluções que permitem aplicar as características inerentes aos sistemas de tempo real ao Linux, foi realizado um estudo, acompanhado de exemplos, sobre o tipo de arquitecturas de kernel mais utilizadas para o fazer. Nos sistemas operativos de tempo real existem determinados serviços, funcionalidades e restrições que os distinguem dos sistemas operativos de uso comum. Tendo em conta o objectivo do trabalho, e apoiado em exemplos, fizemos um pequeno estudo onde descrevemos, entre outros, o funcionamento escalonador, e os conceitos de latência e tempo de resposta. Mostramos que há apenas dois tipos de sistemas de tempo real o '*hard'* que tem restrições temporais rígidas e o '*soft*' que engloba as restrições temporais firmes e suaves. As tarefas foram classificadas em função dos tipos de eventos que as despoletam, e evidenciando as suas principais características.

O sistema de tempo real eleito para criar o sistema de aquisição de dados via porta paralela foi o RTAI/Linux. Para melhor percebermos o seu comportamento, estudamos os serviços e funções do RTAI. Foi dada especial atenção, aos serviços de comunicação entre tarefas e processos (memória partilhada e FIFOs), aos serviços de escalonamento (tipos de escalonadores e tarefas) e atendimento de interrupções (serviço de rotina de interrupção - ISR). O estudo destes serviços levou às opções tomadas quanto ao método de comunicação entre tarefas e serviços, bem como ao tipo de tarefa a utilizar (esporádica ou periódica).

Como neste trabalho, o meio físico de comunicação entre o meio ambiente externo e o *hardware* utilizado é a porta paralela, também tivemos necessidade de perceber como funciona este interface. Nomeadamente os registos de configuração da porta paralela. Assim, foi possível configura-lo ao nível de *hardware* (BIOS) e *software* (módulo do *kernel*) atendendo aos objectivos do presente trabalho, e optimizando a utilização da porta paralela, nomeadamente, aumentando o número de bits disponíveis para a leitura de dados.

No desenvolvimento da tarefa de *hard real-time*, foram tidas em atenção as várias considerações atrás referenciadas. Foi desenvolvida uma tarefa do tipo esporádica, pois era pretendido, ler dados pela porta paralela apenas quando houvesse necessidade (interrupção), ou seja, quando houvesse dados disponíveis para ler. Desenvolvemos também uma aplicação para permitir visualizar os dados recolhidos via porta paralela. A comunicação entre a tarefa e a aplicação é assegurada através de memória partilhada, pois garantindo a consistência de dados, a comunicação entre processos do Linux e as tarefas de tempo real (RTAI) que correm ao nível do kernel torna-se muito simples.

Para puder avaliar o desempenho do sistema desenvolvido, foi criada uma tarefa de *soft real-time* cujos tempos de resposta foram comparados com os da tarefa de *hard real-time*. As respostas temporais obtidas através do analisador lógico em conjunto com gráficos elaborados a partir destes dados, mostram e comprovam, os benefícios do sistema de aquisição de dados em tempo real via porta paralela, usando uma tarefa de *hard real-time.*

Palavras-Chave

Tempo real, aquisição de dados, porta paralela, RTAI, Linux, sistemas operativos, interfaces.

Abstract

In this work was decided to perform a data acquisition system in real time via parallel port. To successfully achieve this goal, was made a research on the literature related with real-time operating systems, creating a historical time-line of these systems, emphasizing and illustrating the most important milestones throughout its evolution. This summary allowed us to understand the proliferation of these systems taking in account the costs involved, depending on their application. As well as the difficulties, scientific and technological, those researchers had, and successfully overcome.

To set up Linux to work as a real-time system, a patch has to be added, such as ADEOS or RTAI, for example. Since there are several solutions to add real time features to Linux, was made a study, with examples, on the most used kernel architectures to achieve such behaviour.

The real-time operating systems have very specific features, and so, were made an overview of the most important features of these systems, such as scheduler, latency and response times. It was also classified in function of the utility of its outcome, i.e, and the temporal constraints. Concluding that there are only two types of real-time systems: the ‘hard’ who has 'strict' time constraints and the 'soft' which encompasses the ‘firm’ and ‘soft’ time constraints. The tasks were classified according to the types of events that they will trigger and deal, highlighting its main features.

The real-time system elected to create the system data acquisition via parallel port was the RTAI/Linux. So, a study was made on the main services and functions of RTAI, with special attention in the communication services between tasks and processes (FIFOs and shared memory), the scheduling services (types of schedulers and tasks) and interrupt service (service interrupt routine - ISR). The study of these services helped to make choices about the method of communication between tasks and services, as well as the type of task to use (periodic or sporadic).

The communication interface used between the external environment and the hardware was the parallel port. To use it right, a summary of this interface was done focusing primarily on the behaviour of the parallel port interface registers. To learn about their operation and how to set it (the hardware and software) according to the real-time data acquisition system needs, and also, to see if it was possible to read more bits than those that are normally available.

In the development of hard real-time task, was taken into account the various concepts referenced above. It was developed a sporadic task, because we had as a principle, 'read the parallel port' only when it's needed (by hardware interruption line), i.e, when there is data to read.

It was also developed an application that runs in user space that shows the data collected via the parallel port. Communication between the task and the application is ensured through the shared memory, because if data consistency is guaranteed, the communication between the Linux process and real time tasks (RTAI) running in the kernel is simpler.

To have a term of comparison, a soft real-time task was created to compare the response times with the hard real-time task. The responses obtained through the logic analyzer in conjunction with elaborate graphics, shows and prove the benefits of the real-time data acquisition system via the parallel port, using a hard real-time task.

Keywords

Real time, data acquisition, parallel port, RTAI, Linux, operating systems, interfaces.

Índice

Índice de Figuras

Índice de Tabelas

Acrónimos

ABS – Anti-lock Braking System

API – Application Programming Interface

ASCII – American Standard Code for Information Interchange

BIOS – Basic Input Ouput System

DMS – Dead Line Monotonic

DMA – Direct Memory Access

EDF – Earliest Deadline First

ECP – Extendend Capabilities Port

EPP – Enhanced Parallel Port

ECR – Extended Control Register

FIFO – First In First Out

HAL – Hardware Abstration Layer

ID – Identification

IEEE – Institute of Electrical and Electronics Engineers

IRQ – Interrupt Request Line

ISR – Interrupt Service Routine

LPT – Line Print Terminal

MUP – Multi-UniProcessor

POSIX – Portable Operating System Interface for Unix

PIC – Programmable Interrupt Controller

RAM – Random Access Memory

RMS – Rate Monotonic

RPC – Remote Procedure Call

RR – Round Robin

RTAI – Real Time Application Interface

RTHAL – Real Time Hardware Abstraction Layer

SATA – Serial Advanced Technology Attachment

SO – Sistema Operativo

STOR – Sistema Operativo de Tempo Real

SOUG – Sistema Operativo de Uso Geral

STR – Sistema de Tempo Real

SQR – System Request

SPP – Standard Parallel Port

SMP – Symmetric Multiprocessor

UP – Uniprocessor

VMM – Virtual Machine Monitor

1. INTRODUÇÃO

O sistema operativo Linux tem vindo a aumentar a sua popularidade, primeiro junto da comunidade estudantil e científica, e nos últimos tempos junto do público em geral e do meio empresarial. Pode-se então encontrá-lo em computadores portáteis, de secretária, em servidores e sistemas embebidos. Apesar de o Linux ter um bom comportamento, principalmente ao nível dos servidores, nunca foi capaz de garantir determinismo aos seus processos, ou seja, não havia garantias que as restrições temporais seriam cumpridas. Como se sabe, existem várias áreas onde o cumprimento das restrições temporais são fundamentais, tais como, robótica, equipamentos médicos computorizados, na aeronáutica militar e civil, e até em electrónica de consumo, onde funcionam como sistemas embebidos de vários dispositivos. Estes sistemas e respectivo *hardware* são bastante específicos, fazendo com que os custos das licenças de uso e o *hardware* dos sistemas operativos de tempo real sejam elevados. Estas restrições criaram a necessidade de desenvolver aplicações que permitissem adicionar ao Linux as funcionalidades de tempo real. Uma dessas aplicações é o RTAI (*Real Time Application Interface*) que foi desenvolvido no Politécnico de Milão, e é tida como uma das soluções mais activas e mantidas, capazes de garantir funcionalidades de tempo real ao Linux. O RTAI, entre outras características importantes, disponibiliza vários métodos de inter-comunicação entre processos que correm ao nível do utilizador, e as tarefas de tempo real que correm ao nível do kernel, tais

como *mailboxes*, *First In, First Out* (FIFOs) e memória partilhada. Estes métodos de intercomunicação tornam mais transparente e de fácil implementação, a comunicação entre processos do Linux e tarefas de tempo real.

O facto de ser possível juntar um sistema operativo gratuito, capaz de correr no *hardware* mais trivial, com uma aplicação capaz de garantir todos os requisitos de um sistema de tempo real faz com que o RTAI/Linux seja uma plataforma viável e económica para desenvolver o sistema de aquisição de dados em tempo real.

1.1. CONTEXTUALIZAÇÃO

Este projecto nasceu da necessidade de implementar um sistema de aquisição de dados em tempo real usando as funções e serviços disponibilizados pelo *Real Time Interface Application* (RTAI), em que o meio de comunicação físico, entre a máquina e o mundo exterior fosse a porta paralela. Os dados obtidos através do sistema de aquisição de dados em tempo real, que deve operar ao nível do núcleo (kernel) do sistema operativo, seriam disponibilizados através de uma aplicação que corre ao nível do utilizador, e que utiliza os serviços e funções para inter-comunicação entre tarefas e processos do RTAI.

1.2. OBJECTIVOS

O principal objectivo deste trabalho é, usando o RTAI, criar um sistema de aquisição de dados, tendo como interface de comunicação a porta paralela. Para atingir o objectivo, houve necessidade de subdividir em várias tarefas e objectivos parciais:

• Fazer uma introdução aos sistemas de tempo real, a sua evolução até aos dias de hoje, quer ao nível do *hardware*, *software* e ferramentas de desenvolvimento.

• Estudar as principais características dos sistemas operativos de tempo real (STOR), bem como as arquitecturas mais usadas para dotar o Linux com o determinismo que caracteriza os STOR;

• Estudo sobre os serviços e funções disponibilizados pelo RTAI, com especial atenção ao nível do escalonador, *Interrupt Service Routines* (ISR) e serviços de intercomunicação entre tarefas e processos;

• Estudo sobre a configuração e registos da porta paralela

1.3. CALENDARIZAÇÃO

	Task Name	Duration	Start	Finish
1	Planeamento da Tese	3 days	Mon 18-10-10	Thu 21-10-10
2	Requisitos bibliográficos	9 days	Fri 22-10-10	Mon 08-11-10
3	Analise da bibliografia	100 days	Tue 09-11-10	Fri 13-05-11
4	Instalação do RTAI	3 days	Tue 09-11-10	Fri 12-11-10
5	Pesquisa sobre serviços do RTAI	30 days	Mon 15-11-10	Fri 07-01-11
6	Pesquisa sobre porta paralela	15 days	Mon 15-11-10	Fri 10-12-10
7	Desenv. Do hardware para p. parale	7 days	Mon 06-12-10	Fri 17-12-10
8	Desenv. Da tarefa de hard real-time	15 days	Fri 17-12-10	Fri 14-01-11
9	Testes	15 days	Fri 17-12-10	Fri 14-01-11
10	Desen. Interface humana	8 days	Fri 21-01-11	Fri 04-02-11
11	Testes	9 days	Fri 21-01-11	Mon 07-02-11
12	Desen. tarefa soft real-time	7 days	Tue 08-02-11	Mon 21-02-11
13	Teste comparativos finais	15 days	Mon 21-02-11	Mon 21-03-11
14	Elaboração do relatório	45 days	Mon 21-03-11	Mon 13-06-11

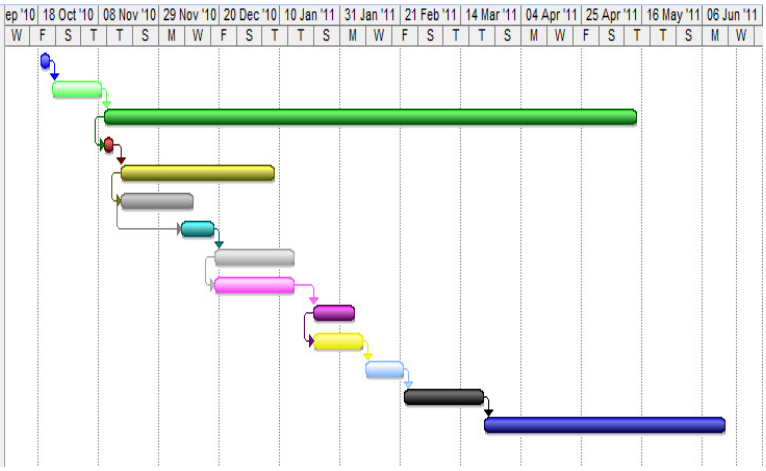

Figura 1 Calendarização da Tese

Para atingir o objectivo deste trabalho, o sistema de aquisição de dados em tempo real, foi adoptada a seguinte calendarização apresentada na Figura 1, onde se incluem tarefas como por exemplo, pesquisa sobre os serviços do *Real Time Applicattion Interface* (RTAI), desenvolvimento da tarefa de tempo real, desenvolvimento da aplicação de interface humana, estudo da documentação sobre porta paralela, testes, etc.

1.4. ORGANIZAÇÃO DO RELATÓRIO

No capítulo 1 é feita a introdução ao trabalho, contextualizando-o e apresentado os objectivos e calendarização da tese, bem como a organização do relatório. No capítulo seguinte, 2, faz-se uma abordagem à evolução tecnológica dos sistemas operativos de tempo real ao nível dos avanços mais marcantes nestes sistemas, quer ao nível do *software*, do *hardware* e das ferramentas de desenvolvimento. Ainda neste capítulo abordam-se também as principais características dos sistemas operativos de tempo real, e as arquitecturas usadas para permitir que o Linux se comporte como um sistema operativo de tempo real. No 3º Capítulo, reúnem-se as principais características do *Real Time application Interface* (RTAI), quer ao nível do escalonador, *Interrupt Service Routines*, comunicação entre tarefas e processos, bem como outros serviços. No Capítulo 4, de uma forma resumida, descreve-se a configuração e funcionamento da porta paralela e dos seus registos, quer ao nível do *hardware*, quer ao nível do *software*. No 5º Capitulo descreve-se a implementação e desenvolvimento do sistema de aquisição de dados em tempo real

usando como plataforma de tempo real o RTAI/Linux. No Capítulo 6, apresentam-se os resultados obtidos, quer ao nível dos dados lidos via porta paralela, quer dos tempos de resposta do sistema de aquisição de dados em tempo real. No 7º Capítulo, são reunidas as principais conclusões e sugestões para futuros desenvolvimentos.

2. SISTEMAS DE TEMPO REAL: INTRODUÇÃO

Apesar de os Sistemas de Tempo Real (STR) passarem despercebidos ao normal cidadão, eles encontram-se disseminados por toda a parte nos dias de hoje, e podem ser encontrados nas empresas, habitações, laboratórios e até em dispositivos de lazer. Têm tantas e tão variadas aplicações que muitas vezes nem se tem consciência das mais-valias que estes tipos de sistemas trazem ao nosso dia a dia. Por exemplo, quem hoje em dia dispensa o *Anti-lock Braking System* (ABS) ou os *airbags* num automóvel? No entanto os STR têm áreas de aplicação muito mais abrangentes, de tal modo que é fácil encontrá-los em áreas tão distintas como na medicina, aplicações militares, sistemas industriais, aeronáutica, aplicações de lazer, multimédia e indústria espacial.

Apesar dos STR que controlam, por exemplo, centrais nucleares, aviões civis e militares ou equipamento de monitorização médica que são extremamente complexos, estes, ainda apresentam algumas características dos sistemas desenvolvidos entre 1940 a 1960 [1] [2].

2.1. AVANÇOS TEÓRICOS IMPORTANTES

Grande parte dos avanços ao nível dos STR teve origem em áreas adjacentes [1], Figura 2, em particular da investigação operacional (*operations research*) que começou a surgir nos inícios de 1940, e de onde surgiu o conceito de escalonador (*scheduling*). Em 1950 surge o conceito de filas (*queueing*) que viria a influenciar a maior parte das teorias. No ano de 1967 Martin, J. escreve o livro *Design of Real-Time Computer Sistems* editado pela *Prentice Hall*, *Englewood Cliffs*. Neste livro, pode-se verificar a influência da investigação operacional (*scheduling*) e do conceito de filas (*queuing*) nos seus trabalhos e investigação em sistemas de tempo real.

Em 1973, Liu, C. L. e Layland, J. W. publicam o seu trabalho sobre a teoria *rate-monotonic*, teoria que viria a ser refinada até aos nossos dias, de modo a que pudesse ser facilmente utilizada nos projectos de STR. Nos anos 80 e 90 assistiu-se à publicação de muitos trabalhos teóricos sobre melhoramento na previsibilidade e fiabilidade dos STR, e também na resolução de problemas relacionados com sistemas de multi-processadores.

Em meados de 1995, grande parte dos problemas dos STR está relacionada com o escalonamento, conforme é referido e apresentado numa importante publicação [3], onde demonstram que a maior parte dos problemas relacionados com escalonamento são extremamente difíceis de resolver através de técnicas analíticas.

Actualmente apenas um conjunto restrito de investigadores e especialistas continuam a estudar problemas relacionados com escalonamento (*scheduling*) e análise de performance, enquanto um grupo mais alargado de engenheiros de sistemas generalistas tentam resolver questões relacionadas com a implementação prática destes sistemas em ambientes reais.

2.2. PRIMEIROS SISTEMAS DE TEMPO REAL

Apesar da origem do termo “Sistema de Tempo Real” ser incerta, foi muito provavelmente usado pela primeira vez no projecto da *Whirlwind* [1], um simulador de voo desenvolvido pela IBM para a Marinha dos EUA, ou no SAGE – *Semi-Automatic Ground Enviroment*, um sistema de defesa aérea desenvolvido para a Força Aérea dos EUA no inicio de 1950. Ambos os projectos, apesar de rudimentares, podem ser qualificados como sistemas de tempo real pelos padrões actuais.

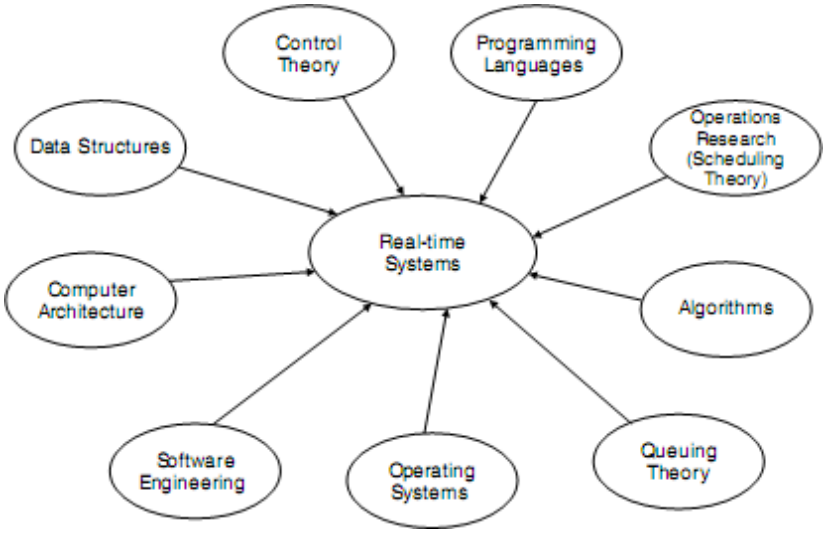

Figura 2 Áreas que influenciaram os Sistemas de Tempo Real [1]

Para além do contributo para os sistemas de tempo real, o projecto *Whirlwind*, usou pela primeira vez memórias com núcleo de ferrite, e um tipo de compilador de linguagem de alto nível, que foi o predecessor do Fortran. Outros Sistemas de Tempo Real (STR), como o SABRE, que tinha como função efectuar as reservas das linhas aéreas, foi desenvolvido para a *American Airlines* em 1959. No entanto, o maior impulso foi dado pelo programa espacial dos EUA, pois foi necessário desenvolver sistemas de tempo real para a telemetria, e também para controlar o veículo espacial.

Durante os anos 60 dá-se um rápido desenvolvimento dos STR, e só então, é que os interessados em soluções de tempo real generalistas, ou seja, que não estavam relacionados com projectos militares, ou instituições de grandes recursos, puderam ter acesso a equipamentos específicos para processamento em tempo real, conforme se pode ver na Tabela 1.

2.3. EVOLUÇÃO DO HARDWARE

Os primeiros Sistemas de Tempo Real (STR) tiveram como principais obstáculos a baixa eficiência dos processadores em termos de velocidade de processamento, e memórias lentas, e limitadas, em termos de armazenamento [1]. O projecto *Whirlwind* inovou ao introduzir as memórias de núcleo de ferrite, um grande avanço em relação ao seu antecessor, o tubo de vácuo.

No inicio de 1950, as interrupções assíncronas foram introduzidas e incorporadas como característica padrão no Univac *Scientific* 1103A. Em meados de 1950 deu-se um grande aumento na velocidade de processamento, resultante da evolução e maior complexidade dos processadores, sem que implicasse aumento de tamanho no *hardware* dos computadores, aumentando assim, a capacidade de processamento das máquinas usadas para fins científicos. Estas inovações levaram a uma maior proliferação dos computadores, e tornou possível que o processamento em tempo real fosse aplicado na área de sistemas de controlo. Tais características e avanços eram particularmente visíveis no desenvolvimento do SAGE por parte da IBM.

Nos anos 60 e 70, os avanços ao nível da integração e velocidade de processamento, levou ao aumento do espectro de problemas que poderiam ser resolvidos com os sistemas de tempo real. Em 1965 foi estimado que existiriam cerca de 350 processos de controlo em tempo real.

Entre 1980 e 1990, as aplicações tempo real, passam a poder correr em sistemas de multi-processador e outras arquitecturas *non-von Neumann*. Nos finais dos anos 90, e inicio de 2000 novas tendências surgem nas áreas dos sistemas de tempo real, principalmente em sistemas embebidos na área dos bens de consumo, e dispositivos *Web*. A evolução dos sistemas de tempo real associada à massificação dos equipamentos electrónicos, leva ao aparecimento de pequenos processadores com memória e funcionalidades limitadas, por exemplo, na electrónica de consumo, que voltam a despertar alguns dos desafios enfrentados pelos primeiros projectistas de sistemas de tempo real.

2.4. EVOLUÇÃO DO SOFTWARE

Os primeiros sistemas de tempo real eram escritos directamente em micro código e linguagem *assembly*, e só mais tarde foram usadas linguagens de programação de alto nível. O projecto *Whirlwind*, já referido anteriormente, usou um tipo de linguagem de programação, para simplificar o código, predecessora das linguagens de alto nível, denominada compilador algébrico [1]. Mais tarde os sistemas adoptaram o Fortran, CMS-2, e JOVIAL, as linguagens de programação que eram as preferidas dos Militares dos EUA. Em 1970 o departamento da defesa dos Estados Unidos ordenou o desenvolvimento

de uma única linguagem, tal que, todos os ramos, Marinha, Força Aérea e Exercito, pudessem usar, e que fosse de alto nível e especifica para sistemas de tempo real.

Após um processo de selecção, em 1983, apareceu a linguagem programação Ada como padrão. Em 1995 surgiu o Ada95 que corrigiu algumas falhas e problemas da versão original. No entanto, nos dias de hoje, apenas um pequeno número de sistemas são desenvolvidos em Ada95. A maior parte dos sistemas são desenvolvidos em C, C++, e até em *assembly* ou Fortran. No entanto nos últimos anos tem surgido um aumento do uso de metodologias e linguagens de programação orientadas a objectos, como por exemplo o Java.

2.5. PRIMEIROS SISTEMAS COMERCIAIS

Os primeiros sistemas operativos eram projectados para computadores *mainframe*[1]. A IBM desenvolveu o primeiro sistema *real-time executive*, o *Basic Executive*, em 1962, que fornecia *scheduling* em tempo real. Em 1963, o *Basic Executive II* já dispunha de programas de utilizador e sistema residentes em disco.

Por meados de 1970, os microcomputadores tornaram-se mais acessíveis financeiramente e já podiam ser encontrados em muitos ambientes de engenharia e desenvolvimento. Como consequência, houve um grande número de sistemas operativos de tempo real que foram desenvolvidos por empresas que fabricavam os microcomputadores. Entre outros podíamos encontrar o *Real Time Multitasking Executives* (RSX) para o PDP 11 da *Digital Equipement Corporation* (DEC), e o *Real Time Executive* (RTE) da *Hewlett-Packard* para a série HP 2000.

No fim dos anos 70 e inicio de 80 surgiu o primeiro sistemas operativos de tempo real projectados para microprocessadores, tais como, o RMX-80, MROS68K, VRTX entre outros. Nos últimos vinte anos muitos sistemas operativos de tempo real tem surgido e muitos também desapareceram. Dos que apareceram, e ainda se mantêm entre

[1] Termo usado para distinguir as máquinas de alto desempenho das máquinas de menor desempenho

nós, encontram-se o QNIX, o VxWorks, RTAI e o LynxOS, por exemplo. Outros como o DNIX, EROS, RT-11 encontram-se extintos, [1]. Na Tabela 1, podemos ver um resumo cronológico dos avanços mais importantes na área de Sistemas de Tempo Real

Tabela 1 Marcos históricos dos Sistemas de Tempo Real [1]

ANO	RESULTADO	DESENVOLVIMENTO	APLICAÇÃO	INOVAÇÕES
1947	Whirlwind	MIT/US Navy	Simulador de voo	Memórias de núcleo de ferrite; Tempos de resposta em "tempo real"
1957	Sage	IBM	Defesa Aérea	Propositadamente desenvolvido em "tempo real"
1958	Scientific 1103A	Univac	Generalista	Interrupções por hardware
1959	Sabre	IBM	Reservas das companhias aéreas	Politica *hub-go-ahead*
1962	Basic Executive	IBM	Generalista	Primeiro sistema a disponibilizar *hard real time*
1963	Basic Executive II	IBM	Generalista	Diversos *real time scheduling*; Disco Residente; Programas utilizador/sistema
1970	RSX, RTE	DEC, HP	Sistema Operativo de Tempo Real	Instalado em "mini computadores"
1973	Rate-Monotonic System	Liu e Layland	Teoria	Estabeleceu um novo paradigma nos sistemas de *schedulers*.
1980	RMX-80; MROS ; 68K; VRTX, etc	Vários	Sistemas Operativos de Tempo Real	Instalados em microcomputadores
1983	Ada 83	Departamento de Defesa dos USA	Linguagem de Programação	Previstos para *mission-critical*; Embebidos; Sistemas de Tempo Real
1995	Ada 95	Comunidade	Linguagem de Programação	Aperfeiçoamento da Ada 83

2.6. CARACTERÍSTICAS IMPORTANTES EM SISTEMAS DE TEMPO REAL

O *hardware* de um computador generalista processa ciclicamente macro-instruções, normalmente designadas por *software*, resolvendo assim os problemas propostos pelos utilizadores. O *software* está normalmente dividido em programas de sistema e programas de aplicação. Os programas de sistema, são responsáveis por fazer o interface com a camada de hardware do sistema, e são normalmente designados por *schedulers*, *device drivers*, *dispatchers*. Os programas de sistemas incluem também as ferramentas para o desenvolvimento aplicações. Entre estas ferramentas estão os compiladores, que são responsáveis por traduzir a linguagem de alto nível em código *assembly*, e os *assemblers* que traduzem a linguagem *assembly* para formatos binários, conhecido por código máquina, e ainda, pelos *linkers* que preparam o código para ser executado. Deste modo pode-se dizer que um sistema operativo é um conjunto de programas de sistema, específicos, que nos permite gerir os recursos físicos de um computador. Assim sendo um Sistema Operativo de Tempo Real (STR) também é um conjunto especifico de programas de sistema [1], mas com determinadas particularidades com iremos ver mais à frente.

Os programas de aplicação, também conhecidos simplesmente por aplicações, são um conjunto de instruções que descrevem tarefas que serão executadas nos sistemas operativos e executadas por um computador para resolver determinados problemas. Normalmente estes estão relacionados com as especificidades dos utilizadores. Por exemplo, num sistema operativo de uso geral: inventários, processamento de folhas de ordenados, aplicações multimédia.

2.6.1. DEFINIÇÃO DE SISTEMA DE TEMPO REAL

"*A real-time system is one in which the correctness of the computations not only depends upon the logical correctness of the computation but also upon the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred*."[4]

A definição anterior, em inglês, diz que os resultados não só dependem do correcto processamento das operações lógicas, mas também do espaço temporal em que elas são produzidas. Se o sistema não cumprir as exigências temporais, então diz que ocorreu uma

falha no sistema. Ou seja, o sistema tem de ser determinístico, para poder garantir que os requisitos temporais sejam cumpridos mesmo que haja variação das cargas a que o sistema está sujeito. Um outro aspecto importante, na definição citada, é que esta não refere a performance do sistema, pois em Sistemas de Tempo Real (STR) é a previsibilidade que é factor primordial e não a sua rapidez.

Por exemplo, um Sistema Operativo Linux que use um processador actual, tem um tempo de resposta a uma interrupção na ordem dos 20µs [4], mas ocasionalmente, o tempo de resposta pode demorar mais tempo. As respostas às interrupções que se prolongam para lá dos 20µs, não se devem ao facto de o Linux, e do *hardware* onde se encontra instalado, não serem céleres na resposta à interrupção, mas ao facto de o Linux não ser determinístico.

2.6.2. LATÊNCIA E TEMPO DE RESPOSTA

A definição anterior conduz a duas características importantes dos STR: a latência e tempo de resposta. Como podemos ver na Figura 3, o tempo entre a chegada da interrupção/inicio do evento e o envio da tarefa correspondente para lidar com o evento é designado por tempo de resposta. Em sistemas de tempo real este tempo deve ser determinístico, e inferior ao pior caso [4].

Sendo minucioso pode-se dizer que o tempo de resposta é um somatório das latências de interrupção e do escalonador, do tempo que a *handler* demora lidar com a interrupção, mais o tempo que o escalonador demora a realizar o *context switch*[2] [4] para que seja enviada uma tarefa para lidar com evento que gerou a interrupção.

[2] Inerente ao processo de enviar uma nova tarefa provocada por uma interrupção está o *context switch*. Este processo consiste em guardar o corrente estado do CPU aquando da interrupção, e de seguida restaurar o estado de uma determinada tarefa. O *context switch* é uma função tanto do sistema operativo como da arquitectura presente no processador.

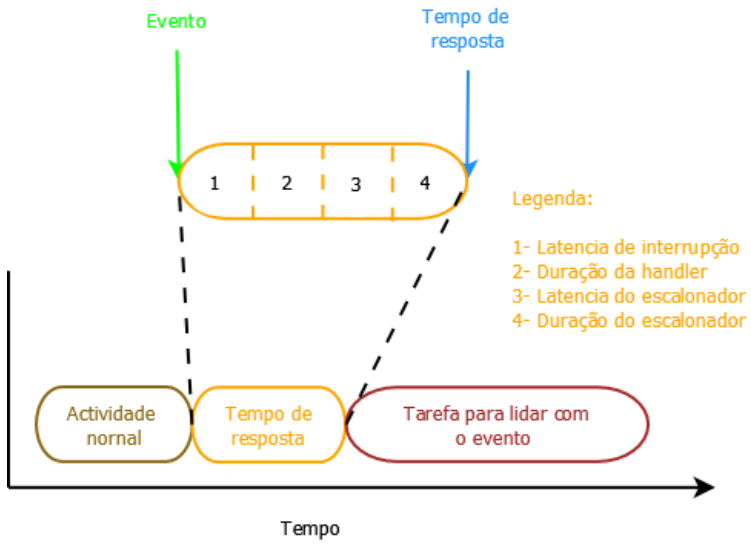

Figura 3 Tempo de resposta e latência

Um exemplo deste tipo de processo é o comportamento dos *airbags* actuais. Quando o sensor detecta a colisão do veículo, provoca uma interrupção no CPU. Esta interrupção está associada uma latência que corresponde ao tempo que passou desde que o sinal activou a interrupção até que a interrupção é atendida pela *handler*.

A partir deste momento o sistema operativo deve então responder rapidamente e despachar a tarefa que faz abrir o airbag, e não permitir que outras tarefas de menor prioridade, ou processos de não tempo real, interfiram no procedimento de abertura do *airbag*. No entanto, por muito rápido que seja a atender existem ainda latências e atrasos a considerar, como o tempo de duração da *handler*, que é o tempo que esta demora a processar a interrupção. Existe ainda o tempo decorrido entre o fim da *handler* e o inicio do escalonador, que é a latência do escalonador, e por fim a duração do escalonador, que corresponde ao tempo desde que escalonador é iniciado, executa a tarefa para abrir o airbag, e muda o contexto para uma nova tarefa ou para o que estava a fazer antes da interrupção.

A abertura do *airbag* quando se dá a colisão de um automóvel, é um bom exemplo para demonstrar como as latências dependem de um conjunto de factores, ver Figura 3, podem influenciar o tempo de resposta de um sistema de tempo real. Neste exemplo em particular,

saber o tempo de resposta é de extrema importância, pois um *airbag* que abre um segundo mais tarde é pior do que não ter *airbag* nenhum [4].

2.6.3. ESCALONADOR PERIÓDICO DE TAREFAS

Para além do determinismo no processamento de interrupções, é necessário um mecanismo capaz de escalonar tarefas em intervalos periódicos. Para gerir estas tarefas existe um serviço cujas características são importantes nos sistemas de tempo real: O escalonador de tarefas [4].

As tarefas periódicas são normalmente aplicadas em sistemas de aquisição de dados, ciclos de controlo, monitorização de sistemas, etc., devido a necessidade constante destes sistemas em saber o estado, e de actualizar os resultados, das variáveis dos sistemas que controlam. Para poder controlar as variáveis afectas a uma tarefa, o escalonador do Sistema de Tempo Real (STR) é responsável por atribuir um período de tempo, Figura 4 *Period* (p), no qual as tarefas têm de ser executadas, de acordo com a sua prioridade. O tempo de execução das tarefas, Figura 4 *Proc time* (t), deve ser inferior ao pior caso, ou seja, o tempo de execução não deve ultrapassar o limite, Figura 4 *Deadline* (d), em que o serviço a realizar atribuído a tarefa se degrade, de tal forma que o resultado da execução da tarefa perca a sua utilidade.

Uma das aplicações que tira partido do escalonador de tarefas é, por exemplo, o sistema de travagens ABS, que actualmente equipam todos os automóveis com a finalidade de evitar o bloqueio das rodas do automóvel quando se efectua uma travagem. Este sistema consiste numa tarefa de tempo real periódica que faz a amostragem da velocidade em cada roda da viatura, Figura 4 *Proc time* (t), com o objectivo de controlar a pressão exercida em cada travão para evitar que a roda bloqueie, Figura 4 *Deadline* (d), numa situação de travagem. Este controlo é conseguido através de uma amostragem que é feita cerca de 20 vezes, Figura 4 *Period* (p), por segundo. Para que o sistema funcione correctamente, a amostragem dos sensores e o controlo da pressão têm de ser feita em intervalos periódicos, e a tarefa tem de terminar a sua execução antes do limite para o qual o resultado do seu processamento se torne inútil, e provoque uma falha catastrófica, que neste exemplo seria o bloqueio das rodas do automóvel durante uma travagem.

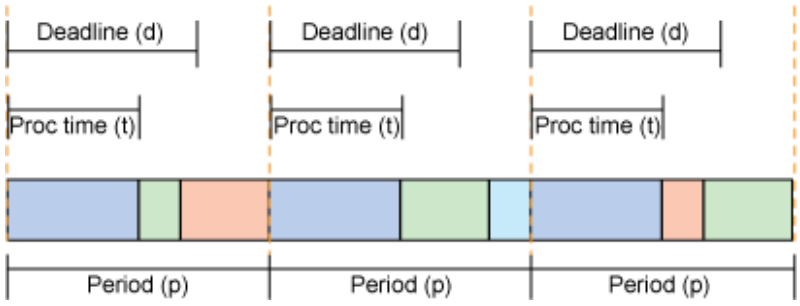

Figura 4 Escalonador de tarefas periódicas [4]

Além de determinar o período de execução das tarefas, o escalonador tem ainda de garantir que tarefas de menor importância não impeçam que a tarefa "ABS" seja executada nos períodos desejados, ou seja, evitar que a tarefa "ABS" perca a sua prioridade.

2.6.4. CLASSIFICAÇÃO DAS RESTRIÇÕES TEMPORAIS

Ao longo do texto, tem-se fazer referência à importância das tarefas de tempo real terminarem antes de atingir o seu prazo de execução, Figura 4 *Deadline* (d). O cumprimento das restrições temporais é importante pois, no mínimo, caso não sejam cumpridas, haverá perda na qualidade do serviço a prestar pela tarefa de tempo real. De acordo com a utilidade do resultado para a aplicação as restrições temporais podem-se classificar da seguinte forma [14], Figura 5:

- **Suave (*Soft*) -** Restrição temporal em que o resultado que a ela está associado mantém alguma utilidade para a aplicação, mesmo depois de um limite D, embora haja uma degradação da qualidade de serviço.
- **Firme (*Firm*) -** Restrição temporal em que o resultado que a ela está associado perde qualquer utilidade para a aplicação depois de um limite D.
- **Rígida (*Hard*) -** Restrição temporal que, quando não cumprida, pode originar uma falha catastrófica.

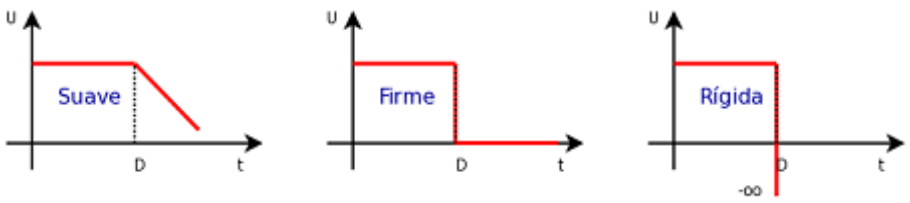

Figura 5 Restrições temporais [14]

2.6.5. CLASSIFICAÇÃO DOS SISTEMAS DE TEMPO REAL

Os sistemas de tempo real são classificados como *hard real-time* e *soft real-time.* Os primeiros são determinísticos, o que leva a que os sistemas cumpram as tarefas de tempo real dentro dos prazos de execução, mesmo sob as piores cargas de processamento. Normalmente a este tipo de sistemas estão associadas as restrições temporais rígidas, Figura 5.

Nos segundos, *soft real-time*, os tempos de execução continuam a ser fundamentais. No entanto neste tipo de sistemas de tempo real é aceitável que respeite, em média, os prazos de execução das tarefas. Os sistemas de *soft real-time* estão associados a restrições temporais firmes e suaves, conforme se pode ver na Figura 5.

Nos sistemas de *hard real time* o incumprimento do prazo de execução provoca normalmente resultados catastróficos, por exemplo, abertura tardia do airbag, ou permitir que o travão (ABS) esteja sob pressão tempo excessivo, bloqueando as rodas. Já os sistemas de *soft real time* podem falhar os prazos de execução sem causar uma falha crítica no sistema, por exemplo, uma aplicação multimédia, que perde algumas frames de vídeo durante uma vídeo-conferência, ou uma perda de pacotes por um dispositivo de rede, por exemplo um router [4].

2.6.6. CLASSIFICAÇÃO DAS TAREFAS

As tarefas são programas independentes que são executados com um determinado propósito. Esse propósito, por exemplo pode ser o controlo do sistema de travagem anti-bloqueio, conhecido por ABS, ou a activação do *airbag* após colisão do veículo. Estas tarefas, no entanto, não são activadas da mesma forma e também têm características ligeiramente diferentes. Tendo em conta estas diferenças, tarefas dos sistemas de tempo real podem ser classificadas em três tipos diferentes [14]:

- **Tarefas periódicas –** Caracterizadas por serem activadas regularmente em intervalos fixos. As tarefas têm as seguintes características:
    - C_i -Tempo de execução de pior caso
    - T_i - Período da tarefa
    - D_i - *Deadline* (relativa) da tarefa

Um exemplo deste tipo de tarefas é a amostragem periódica de um sensor, por exemplo ABS de um automóvel.

- **Tarefas Esporádicas –** Este tipo de tarefas tem como característica o facto de serem activadas em função de eventos (*event driven*) provocados por elementos externos ou alterações de ambiente, e são caracterizadas por:
    - C_i -Tempo de execução de pior caso
    - D_i - *Deadline* (relativa) da tarefa
    - mit_i - Representa o tempo mínimo entre activações

Um exemplo deste tipo de tarefas é a detecção de objectos em linhas de produção, ou detecção de uma colisão para abertura de *airbag*.

- **Tarefas aperiódicas** – Também são tarefas do tipo *event driven*, como a anterior, no entanto, tem propriedades desconhecidas. Podem ser reflexo de múltiplos eventos, geralmente muito próximos ou mesmo simultâneos.

2.7. ARQUITECTURAS PARA TORNAR O LINUX NUM SISTEMA OPERATIVO DE TEMPO REAL.

Hoje em dia o SO (Sistema Operativo) Linux pode ser uma excelente plataforma para implementar, estudar, experimentar e caracterizar algoritmos de tempo real, ou como ferramenta de desenvolvimento de aplicações de software. Com kernel 2.6 standard do Linux já é possível obter performances de *soft real-time*, e com um pouco mais de trabalho, ou seja, aplicando um *patch* no kernel, podemos criar um sistema capaz de garantir o determinismo que caracteriza as aplicações de *hard real-time*. De seguida, vamos descrever sucintamente, e acompanhando com exemplos, os métodos mais utilizados para transformar o Linux num sistema operativo capaz de processamento *hard real-time*.

2.7.1. MÉTODO *THIN* KERNEL

Foram tentadas inúmeras formas para que a abordagem *thin kernel* gerasse as tarefas de tempo real a partir do kernel standard do Linux. O *thin kernel* [4] também conhecido como *micro-kernel*, consiste num segundo kernel, que funciona como uma interface de abstracção entre o *hardware* e o kernel do Linux, ver Figura 6. Neste método, o kernel do Linux, não tempo real, corre em *background* como uma tarefa de baixa

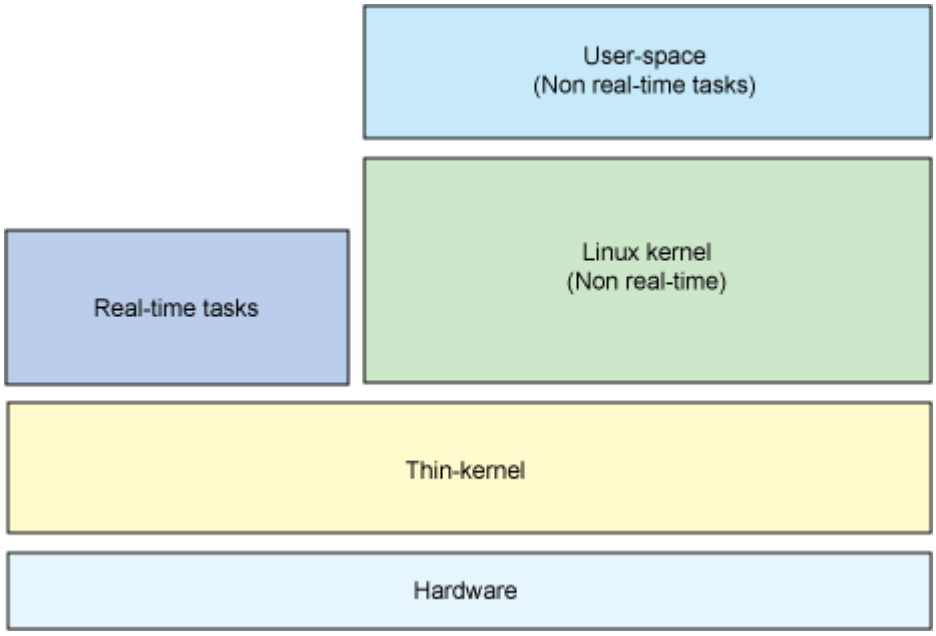

Figura 6 Estrutura do *Thin kernel* [4]

prioridade do *thin kernel* e suporta todas as tarefas que não sejam de tempo real, enquanto as tarefas de tempo real correm directamente no *thin kernel*.

A principal função do *thin kernel*, para além de suportar as tarefas de tempo real, é a gestão das interrupções. O *thin kernel* intercepta as interrupções para garantir que o kernel do Linux (base do sistema), que não é de tempo real, não se antecipe e atenda as interrupções, permitindo deste modo que o *thin kernel* forneça o suporte para *hard real-time* [4].

Apesar das vantagens de ter um sistema de tempo real a coexistir com o kernel Linux comum, este tipo de solução também acarreta alguns problemas. Por exemplo, as tarefas de tempo real e de não tempo real são independentes o que torna o *debugging* mais difícil. Outro problema é que as tarefas de não tempo real não são suportadas a cem por cento pela plataforma Linux. O *Real Time Application Interface* (RTAI), o RTLinux propriedade da *WindRiver Systems* e o Xenomai usam este tipo de abordagem.

2.7.2. MÉTODO NANO-KERNEL

O *nano-kernel* [4] consiste num kernel minúsculo que inclui um gestor de tarefas. Este tipo de abordagem vai um passo mais além, reduzindo ainda mais o tamanho do kernel.

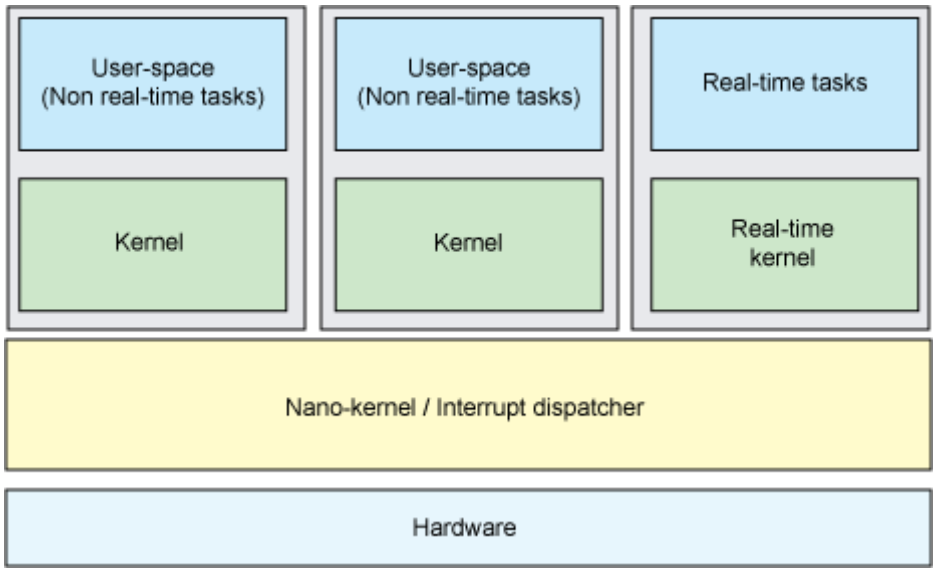

Figura 7 Estrutura do Nano-kernel [4]

Desta forma deixa de ser cada vez menos um kernel e mais uma camada de abstracção de *hardware*, HAL (*Hardware Abstration Layer*).

O *nano-kernel* permite a partilha de recursos do *hardware*, para que possibilite o suporte de vários sistemas operativos, os quais funcionam a um nível superior, ver Figura 7. Como o *nano-kernel* abstrai o *hardware*, este pode estabelecer prioridades para os sistemas operativos nas camadas superiores, e assim, obter desempenho de tempo real.

Na Figura 7, podemos ver as semelhanças entre esta abordagem e a abordagem à virtualização, processo que se usa para correr vários SO (Sistemas Operativos) no mesmo *hardware*. Neste caso, o *nano-kernel* abstrai o *hardware*, e separa o kernel de tempo real do kernel não tempo real. Isto é similar a forma como os *hypervisors* (também conhecido como VMM – *Virtual Machine Monitor*) abstraem o *bare hardware*[3] do SO anfitrião.

Um exemplo da aplicação do *nano-kernel* é o *Adaptive Domain Environment for Operating Systems* (ADEOS). O ADEOS suporta múltiplos sistemas operativos concorrentes em simultâneo, e quando um evento ocorre, O ADEOS interroga cada SO em cadeia para ver qual deles vai lidar com o evento.

---

[3] *Hardware* básico

2.7.3. MÉTODO *RESOURCE* KERNEL

O princípio do *resource* kernel, Figura 8, consiste em adicionar um módulo ao kernel que permite criar reservas para vários tipos de recursos [4]. Estas reservas garantem acesso a recursos do sistema multiplexados no tempo: CPU, rede ou largura de banda do disco. Estes recursos têm vários parâmetros, tais como: período de recorrência, tempo de processamento requisitado, ou seja, a quantidade de tempo necessário para o processamento, e a *deadline* (prazo de execução).

Esta abordagem fornece um conjunto de aplicações de interface (APIs – *Application Program Interface*), que permitem que as tarefas requisitem estas reservas. O *resource kernel* pode então agregar os pedidos (*requests*) de forma a definir o escalonamento que ira garantir o atendimento das tarefas dentro dos prazos de execução, usando para isso as condições e restrições definidas nas próprias tarefas, ou retornar erro, se as condições não forem garantidas. Para além de garantir o cumprimento dos prazos de execução, usando algoritmos como o EDF (*Earliest-Deadline-First*), o *resource kernel* pode lidar com *workloads* (cargas de trabalho) dinâmicos.

Este tipo de abordagem foi implementado no CMU's (*Carnegie's Mellon University*) Linux/RK, que integra no Linux um *resourse kernel* portável, como um módulo, que permite ser carregado ou descarregado do kernel. Esta aplicação evoluiu para uma versão comercial conhecida por TimeSys Linux/RT.

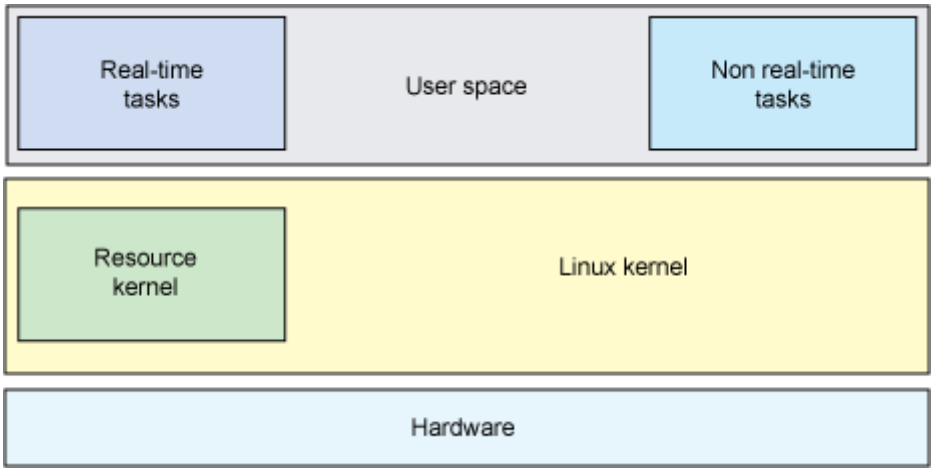

Figura 8 Estrutura *Resource Kernel* [4]

2.7.4. LINUX COM KERNEL 2.6 STANDARD PREENTIVO

Actualmente o Linux kernel 2.6 já permite performances de *soft-real-time* através de uma simples configuração do kernel para o tornar completamente preentivo, ver Figura 9 [4]. Numa configuração padrão do kernel Linux, quando um processo do espaço de utilizador faz uma chamada ao kernel, através de uma chamada do sistema, a chamada não pode ser interrompida. Ou seja, se um processo de baixa prioridade fizer uma chamada ao sistema, um outro processo de maior prioridade tem de esperar, que a tarefa com menor prioridade termine a sua execução, para poder aceder ao CPU (*Central Processor Unit*). A nova opção de configuração no kernel – CONFIG_PREEMPT – altera este comportamento. Assim, os processos podem ser interrompidos, caso haja alguma tarefa de maior prioridade a necessitar de ser executada, mesmo que os processos estejam a meio de uma chamada ao sistema.

Esta configuração também tem alguns aspectos negativos. Embora a opção permita o desempenho em tempo real, mesmo sob carga, o sistema operativo continua executar de forma célere, mas fá-lo com um custo. Esse custo é o rendimento um pouco menor, e uma pequena redução no desempenho do kernel por causa da sobrecarga adicional da opção CONFIG_PREEMPT. Esta opção é útil para sistemas desktop e sistemas embebidos, mas pode não ser a melhor opção em todos os cenários, como por exemplo, em servidores [4].

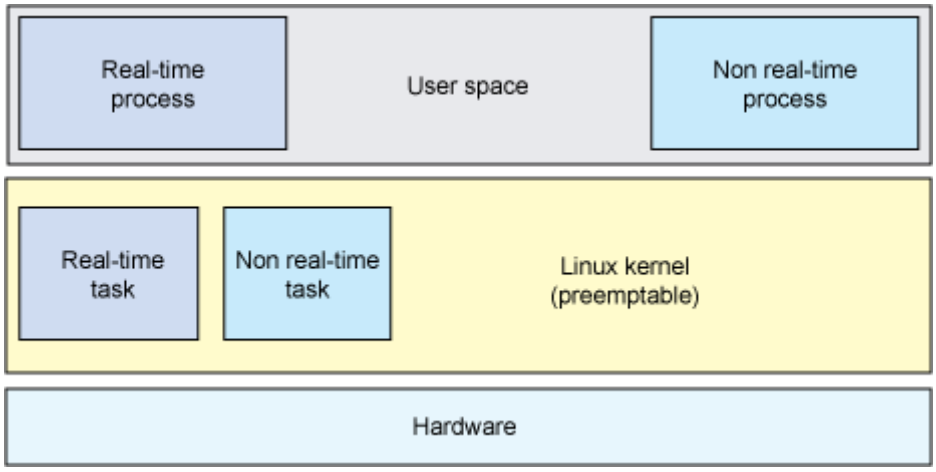

Figura 9 Estrutura Linux kernel 2.6 preentivo [4]

O kernel 2.6 incorpora um novo escalonador, "O(1)", que trás grandes benefícios para o desempenho dos sistema operativo de tempo real, mesmo quando há um grande número de tarefas, pois pode escalonar periodicamente, independentemente do número de tarefas a executar [4].

Outra das características de grande utilidade no kernel 2.6 é a existência de temporizadores/contadores de alta resolução. Esta nova opção permite que os temporizadores/contadores operem com uma resolução inferior a 1µs, depende do hardware da máquina, e implementa também um gestor dos contadores com *red-black tree*[4] para aumentar a eficiência. Deste modo, usando o algoritmo *red-black tree*, um grande número de temporizadores/contadores podem estar activos sem afectar a performance do subsistema do temporizador/contador.

Com um pouco mais de trabalho, podemos criar um sistema *hard-real-time* inserindo no kernel 2.6 o PREEMPT_RT *patch*. Este *patch* fornece várias modificações para permitir o desempenho de *hard real-time*. Algumas modificações incluem reimplantar as *locking primitives* no kernel, implementar herança de prioridade para os mutexes, e converter gestores de interrupções (*interrupt handlers*) em *threads* de kernel para estas ficarem totalmente preentivas [4].

---

[4] *Red-Black tree* [5] é um de muitos tipos de algoritmos de procura que são "balanceados" de modo a garantir que um conjunto básico de operações dinâmicas demora O(log *n*) no pior caso. Sendo *n* o numero de elementos da árvore.

3. O RTAI

O RTAI (*Real-Time Application Protocol*) devidamente configurado no SO (Sistema Operativo) Linux para além de garantir os requisitos que caracterizam os Sistemas de Tempo Real, disponibiliza ainda um conjunto de serviços e funções, tais como: FIFOS, semáforos, *mailbox*, mensagens, primitivas de comunicação RPC (*Remote Procedure Calls*), um subsistema POSIX compatível, escalonadores, gestão de interrupções, entre outros. Estes serviços garantem, para além do determinismo, abstracção e transparência, facilitando a programação e a configuração das tarefas de tempo real, bem como a comunicação entre tarefas de tempo real, ou entre tarefas de tempo real e os processos não determinísticos que correm no kernel do Linux. Para que a intrusão no kernel do Linux seja a menor possível, a ligação entre o Linux e o RTAI é feito através da camada de abstracção de *hardware*.

No RTAI, existe ainda um módulo, o LXRT, que fornece uma interface que permite, através do espaço de utilizador do Linux, aceder aos serviços e recursos do RTAI. Esta característica única e exclusiva do RTAI faz com que este módulo seja considerado um importante recurso do RTAI. Este recurso permite, por exemplo, que um programador que queira desenvolver com segurança uma tarefa de tempo real, o faça no espaço de utilizador do Linux usando as ferramentas de *debug* disponíveis neste espaço, ficando assim

protegido contra falhas de sistema originados por problemas gerados pelas tarefas que estão em desenvolvimento, graças à protecção de memória. Não esquecer que o espaço de kernel não está protegido contra acessos inválidos causados por módulos ou tarefas com *bugs*, e *trap*[5] *handling*.

No decorrer deste capítulo serão descritos os serviços e funções mais importantes disponibilizados pelo RTAI.

3.1. APARECIMENTO DO RTAI

Os Sistemas Operativos de Uso Geral (SOUG) são desenvolvidos para poderem disponibilizar à maior parte das suas aplicações uma "justa" parte dos recursos do sistema. Desta forma, vários e diferentes mecanismos são implementados para garantir que esta partilha de recursos seja feita com eficiência. De um modo geral, a partilha de recursos é feita através da utilização de esquemas de antiguidade e execução de precedências de partes importantes do código do sistema. Este tipo de abordagem é o mais adequado para ambientes de estações de trabalho e servidores, onde inclusive encontra a suas raízes. No entanto, para sistemas onde os tempos de resposta são determinísticos esta abordagem não é funcional pois não cumpre os requisitos, ou seja, não cumpre os prazos de execução.

Ao projectar os Sistemas Operativos de Tempo Real (SOTR) deve-se ter em atenção e garantir determinados propósitos tais como: tempo de resposta das interrupções, *context switch* e herança de prioridade, conforme referido no Capítulo 2. No entanto, a grande limitação nestes Sistemas Operativos (SO) é o facto de apenas suportarem um conjunto restrito de *hardware* e dispositivos em relação ao SOUG, além dos custos e das políticas restritivas no que respeita as licenças dos SOTR [7]. Com a chegada e disseminação dos sistemas operativos de 32-bits aumentou ainda mais a vontade de conjugar as características dos SOUG e dos STOR, ou seja, aliar a variedade do *hardware* compatível, dos SOUG, às características determinísticas dos STOR. À medida que o kernel do Linux se foi tornando cada vez mais popular, e o seu código fonte cada vez mais divulgado, começaram as primeiras tentativas para dotar o Linux de funcionalidades de tempo real.

---

[5] *Trap* é uma interrupção que é gerada pelo software após um erro, depois de um pedido específico de um programa de utilizador para executar um serviço do sistema.

No entanto, o kernel do Linux 2.0.xx ainda não tinha capacidade para suportar um RTHAL (*Real Time Hardware Abstration Layer*), que era fundamental para atingir o propósito de ter um sistema operativo onde se pudesse conciliar os benefícios dos SOUG com os SOTR. Mas estas adversidades foram ultrapassadas com o projecto RTLinux [6], onde se conseguiu desviar recursos do Linux de forma a garantir as necessidades de tempo real.

Com a disponibilização da nova versão do Linux kernel 2.2.xx, no inicio de 1999, tornou-se possível implementar o conceito RTHAL com sucesso, pois a camada de gestão de *hardware* já se encontrava devidamente implementada. Esta abordagem implica pequenas alterações ao nível do kernel, mas possibilita que as aplicações de tempo real sejam carregadas dinamicamente como um módulo do kernel. Tendo uma arquitectura similar ao RTlinux [8], o RTAI também trata o kernel standard do Linux como uma tarefa de tempo real, de baixa prioridade, e que pode realizar o seu normal trabalho, desde que nenhuma outra tarefa de tempo real[6] de maior prioridade esteja a decorrer. Em Março do mesmo ano a primeira versão do RTAI é lançada, e desde então, várias versões têm sido lançadas e vários melhoramentos têm sido feitos a vários níveis, tornando-se capaz de suportar uma grande variedade de projectos, e podendo ser executado num vasto número de arquitecturas, desde i386 até aos PowerPC, usado em vários sistemas de tempo real e suportado por uma largo número de distribuidores [7].

É neste contexto que nasce esta forma híbrida de sistemas operativos, que juntam as características dos SOUG, ao nível do suporte de uma grande gama de *hardware*, e dos STOR que fornece as características determinísticas típicas destes sistemas operativos, bem como as restritas políticas de escalonamento.

3.2. RTAI: CONTROLO E GESTÃO DOS RECURSOS DO LINUX

Como já referido anteriormente, ao adicionar ao kernel do Linux o *Real Time Application Interface* (RTAI), na realidade está-se a dotar o sistema de um mecanismo que permite controlar e gerir os eventos críticos despoletados pelos ambientes externos que se quer controlar. Para que os eventos críticos não se tornem catastróficos, o RTAI, tem de retirar o

[6] As tarefas de tempo real no RTAI são implementadas como módulos do kernel do Linux

controlo ao Linux, pois não sendo determinístico, não garante o determinismo típico dos sistemas de tempo real. O modo como o RTAI retira o controlo ao Linux, e a forma como os gere, são descritos nas secções seguintes.

3.2.1. RTHAL: TIRAR O CONTROLO AO LINUX

Num Sistema Operativo de Tempo Real (SOTR) os eventos críticos são despoletados por interrupções de *hardware* externas. Isto leva a que seja necessário que estas interrupções sejam desviadas do Linux sem afectar o seu normal comportamento, além disso, é necessário garantir que o Linux não estará numa posição de poder controlar a ocorrência destas interrupções. Deste modo, é necessário que o SOTR tenha meios que lhe permita desviar do Linux o controlo dos seguintes mecanismos: o fluxo de interrupções para o Linux e o controlo que o Linux tem sobre estas interrupções. No RTAI isto é feito através do *Real Time Hardware Abstration Layer* (RTHAL), que é um *patch* com cerca de cem linhas [7], que é adicionado ao kernel do Linux. O RTHAL é um conceito introduzido pela equipa de desenvolvimento do RTAI, e que consiste na intercepção e processamento das interrupções por *hardware*, e desta forma sobrepor-se ao Linux no controlo das interrupções. Para controlar as interrupções o RTHAL, que é uma estrutura instalada no kernel do Linux, reúne a informação disponível no kernel sobre o *hardware* interno e as funções nativas necessárias para o RTAI operar [8].

Para melhor entender o parágrafo anterior, sintetizou-se na Figura 10 o controlo de fluxo das interrupções. Esta figura apresenta duas situações: Primeiro, numa hipotética situação inicial, indicada como “1”, em que o RTAI não foi carregado no Linux, este comunica normalmente com a camada de *hardware*, ou seja, o Linux controla as interrupções. Assim que o RTAI é carregado, indicado como “2”, o controlo deixa de ser efectuado pelo Linux, e todas as interrupções fluem agora pelo núcleo do RTAI em ambas as direcções. Para que isto seja possível, o RTHAL, que é constituído por ponteiros de funções que inicialmente apontam para as funções e tabelas nativas do Linux, serão direccionadas para as funções e tabelas internas do RTAI, assumindo assim o controlo das interrupções.

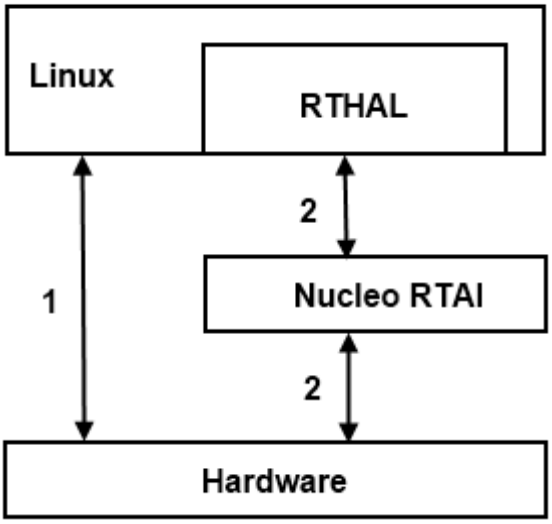

Figura 10 Controlo do fluxo das interrupções de e para o *hardware*

Uma outra função do RTHAL é de minimizar o número das alterações necessárias no código do kernel e assim melhorar a *maintainability*[7] do RTAI e do kernel do Linux. Com o RTHAL as operações, por exemplo gestão de interrupções (*interrupt handlers*) são fáceis de alterar ou modificar sem que se interfira com o SO Linux [8].

3.2.2. RTHAL: TIRAR O CONTROLO AO LINUX

No exacto momento em que o RTAI assume o controlo, passa de imediato a gerir as interrupções de, e para o Linux, e como consequência vai proporcionar às tarefas de tempo real abstracções e funcionalidades, onde se incluem a alocação das interrupções e o controlo do temporizador/contador (*timer*). O RTAI também vai fornecer um meio para disponibilizar as funcionalidades não determinísticas do Linux utilizando os SRQs (*System Requests*), que também é utilizada para alargar o conjunto de serviços disponibilizados aos programas que correm no espaço do utilizador.

Para obter o controlo e gerir as interrupções no Linux, o RTAI substitui o par de funções `cli()/sti()`[8]. Estas substituições, são na realidade funções do RTAI que irão activar *flags* na estrutura interna que é utilizada para registar se o Linux quer ser informado da

[7] Facilidade com que as aplicações podem ser modificadas para corrigir defeitos; atingir novos requisitos; facilitar futuras manutenções; ajustar a um ambiente diferente.

chegada de interrupções – `sti()` – , ou se as ignora – `cli()` –. Como numa situação normal, todas as interrupções seriam sinalizadas após a ocorrência de um *sti* do *hardware*, o RTAI implementa este mecanismo de substituição das funções `cli()/sti()` através do registo de todas as interrupções pendentes no Linux e activando-as sempre que for necessário [7]. Para perceber melhor o processo de atendimento de uma interrupção numa máquina Linux com o *patch* RTAI configurado, será descrito no parágrafo seguinte, tendo como suporte a Figura 11.

Quando surge um evento despoletado por uma interrupção global, através do Controlador de Interrupções Programável (PIC) 8259, a primeira função que encontra é `dispatch_global_irq()`, RTAI *Dispatcher* na Figura 11. Se existir uma *handler*, RT *Int Handler* na Figura 11, que esteja associada a esta interrupção, será chamada nesta altura. Após ter terminado, ou caso não existam mais *handlers*, irá ocorrer uma verificação para saber se o Linux estava a espera desta interrupção ou não, e verificar também, se tem as interrupções desabilitadas (`cli()/sti()`).

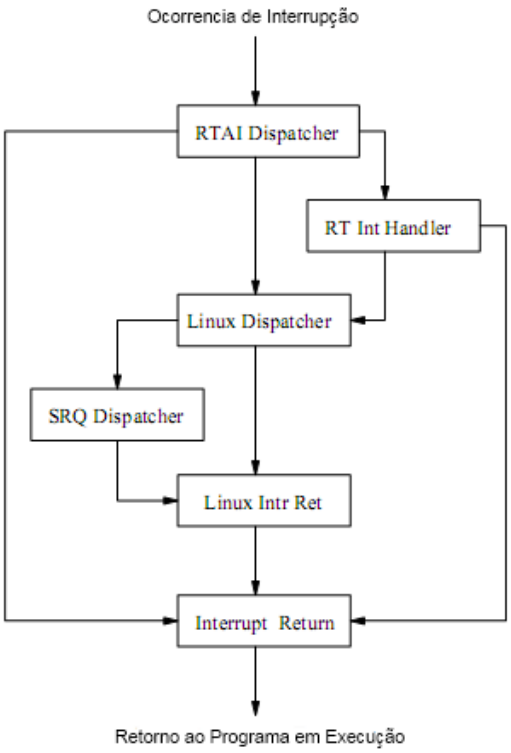

Figura 11 Caminho percorrido pela interrupção num sistema RTAI/Linux [7]

---

[8] CLI (Clear Interrupts); STI (Set Interrupts), ou seja, desactivar e activar interrupções.

No caso de as interrupções estarem habilitadas, o *Interrupt Dispatcher* do Linux, Linux *Dispatcher* na Figura 11, será chamado para lidar com a interrupção. Caso contrário, o RTAI sai do contexto de interrupção e retorna ao programa em execução, *Interrupt Return*, na Figura 11. Quando o *Interrupt Dispatcher* do Linux é chamado, Linux *Dispatcher* na Figura 11, poderá ser seguido pelo *System Request Dispacher*, SRQ *Dispatcher* na Figura 11, que irá chamar todas as SQR's (*System Requests*) que possam ter sido activadas pela *handler* de interrupção, e é usando esta característica que os módulos do RTAI têm acesso aos recursos não determinísticos do kernel. Isto leva a que, no caso de existirem, logo que as SRQ's sejam chamadas, o RTAI salta normalmente para o código (`ret_from_int`), Linux Intrt Ret na Figura 11, que permite fazer o retorno da interrupção e retornar ao programa em execução [7].

Conjugando o que foi descrito anteriormente, é possível representar a arquitectura do RTAI conforme é apresentada na Figura 12, onde é possível identificar as interrupções que têm origem no processador e nos periféricos. As interrupções que têm origem no processador, principalmente sinais de erro, do tipo divisões por zero, continuam a ser tratadas pelo kernel standard do Linux. Já as interrupções despoletadas pelos periféricos, como por exemplo a porta paralela, são tratadas pelo RTAI *Interrupt Dispatcher*. Quando não existem tarefas de tempo real activas o RTAI reencaminha as interrupções para os *handlers* do kernel standard Linux.

O escalonador (*scheduler*) também existe "separadamente" para o Linux e para o RTAI, sendo o escalonador do RTAI discutido mais adiante neste capítulo.

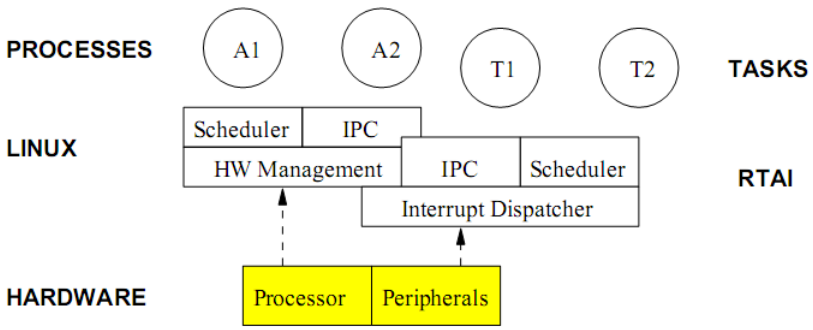

Figura 12 Arquitectura básica do RTAI [8]

3.3. SERVIÇOS E FUNÇÕES DO RTAI

O *Real Time Apllication Interface* (RTAI) disponibiliza um conjunto de serviços e funções de modo a garantir ao programador abstracção de alto nível na programação das tarefas, para que o desenvolvimento de aplicações seja o mais flexível possível. Estes serviços e funções incluem *Interrupt Service Routines* (ISR), escalonadores, mecanismos de comunicação e sincronização, entre outros [7][8]. No âmbito desta tese são descritos, seguidamente, os serviços e funções considerados mais relevantes para atingir o objectivo proposto.

3.3.1. *INTERRUPT SERVICE ROUTINES* NO RTAI

Sempre que uma interrupção é despoletada há necessidade de associar uma função/*handler* à interrupção para lidar com o evento que lhe deu origem. Este tipo de funções/*handlers* são normalmente designados por *Interrupt Service Routines* (ISR), e são usadas para processar as interrupções de *hardware* que têm origem em temporizadores/contadores, impressoras, disco rígido, controladores *Ehternet*, etc.

Para perceber melhor como usar esta funcionalidade do RTAI, será exemplificado como se atribui uma *Interrupt Service Routine* (ISR) a uma interrupção, para que seja executada sempre que essa interrupção ocorrer. Como a programação de uma ISR depende sobretudo do dispositivo do *hardrware* em si, o RTAI fornece apenas um conjunto de funções para ligar uma determinada interrupção à ISR.

A ISR não é mais do que uma função/*handler* sem argumentos e que não retorna nenhum valor, no entanto é responsável por mediar a interacção entre o dispositivo de *harware* e a aplicação de tempo real. A função ISR tem o seguinte formato:

```
static void isr_code ()
{
/*Escrever aqui o código que queremos executar*/
}
```

A associação da ISR à interrupção, neste caso IRQ7 correspondente à porta paralela, é bastante simples, e é realizada através do uso das funções disponibilizadas pelo RTAI. Neste caso, no '`init_module()`', teríamos de declarar as seguintes funções:

```
rt_free_global_irq(IRQ); /*desconectávamos outras
ISR*/
rt_request_global_irq(IRQ,handler);/*ligamos a
ISR pretendida ao IRQ*/
rt_startup_irq(7)/*informa ao RTAI para a usar*/
outb(inb(0x37A)|0x10,0x37A);/* diz ao hardware
par a habilitar a interrupção*/
```

Na segunda instrução, do excerto do código anterior, está definida a função que liga a *Interrupt Request Line* (IRQ) à ISR. O seu primeiro argumento corresponde às IRQ que se quer utilizar, e o segundo à ISR que irá processar a interrupção. A terceira instrução tem como finalidade iniciar e configurar a PIC para aceitar o IRQ que se pretende configurar.

Para remover a ISR convêm desassociá-la do IRQ. E isso faz-se no `cleanup_module()`', em que o processo é análogo, mas inverso ao da associação, conforme se pode ver no extracto de código seguinte.

```
outb(inb(0x37A)& 0xEF,0x37A);/* diz ao hardware
para desabilitar a interrupção*/
rt_shutdown_irq(IRQ)/*informa ao RTAI para a
ignorar*/
rt_free_global_irq(IRQ,handler);/*desconecta a
ISR do IRQ*/
```

3.3.2. ESCALONADOR DO RTAI

O escalonador do RTAI é inserido no kernel como um módulo. Esta abordagem tem como vantagem facilitar a utilização de outros escalonadores, caso seja necessário. Existem no RTAI três tipos de escalonadores diferentes, os quais serão abordados com mais pormenor mais adiante.

As unidades do escalonador do RTAI são denominadas como tarefas. Existe sempre pelo menos uma tarefa, nomeadamente o Linux, que é executada sempre como tarefa de mais baixa prioridade, assim sendo, sempre que uma tarefa de tempo real é criada, o escalonador dá-lhe prioridade sobre o kernel do Linux. O escalonador também é responsável por fornecer serviços tais como, *suspend*, *resume*, *yeld*, make *periodic*, *wait until*, que na realidade não passam de serviços que permitem o controlo do escalonador, ou seja: activar, suspender, ou desactivar escalonador. Estes serviços são equivalentes aos que são usados em vários sistemas operativos de tempo real [8].

O RTAI disponibiliza três tipos de escalonadores:

- ***UNIPROCESSOR* (UP) -** O escalonador *Uniprocessor* (UP) informa o algoritmo de escalonamento para que seleccione a tarefa a correr num único CPU. Este processo é muito transparente. O processo que tiver a maior prioridade fica com os recursos do CPU. Na verdade é um escalonador que tem como princípio a prioridade multi-lista, o que faz com que o Linux seja interpretado como uma tarefa de tempo real como outra qualquer, mas com a prioridade mais baixa de todas. Quanto a implementação propriamente dita, o escalonador é dividido em duas funções complementares: O `rt_schedule()` e `rt_timer_handler()`, em que a primeira pode ser chamada de diversas formas para alterar o escalonador, ou para alterar o estado de um processo, caso seja necessário. A segunda, é para lidar exclusivamente com o temporizador/contador (*timer*) de interrupção [7].

- ***SYMETRIC MULTIPROCESSOR* (SMP) -** O escalonador *Symetric MultiProcessor* (SMP) diferencia-se do UP pela sua capacidade de atribuir tarefas a mais do que um processador. Esta característica dos SMP é possível graças a diversos tipos de serviços, como por exemplo, a possibilidade de imputar uma tarefa a um determinado CPU, ou definir que uma IRQ irá ser tratada por um CPU específico. Convêm notar que este escalonador ao contrário do UP, não está limitado ao uso do temporizador /contador (*timer*) 8254 como única fonte de interrupções por tempo, pois a APIC (*Advanced Programmable Interrupt Controller*) possui o seu próprio temporizador /contador (*timer*). Tal como no escalonador *Uniprocessor*, o escalonador *Symetric MultiProcessor* continua orientado à prioridade [7].

- ***MULTI-UNIPROCESSOR* -** Este escalonador, MUP, vê as máquinas multiprocessador como um conjunto de uniprocessadores. Isto que significa que cada CPU pode ter o *timer* programado de forma independente dos outros. Esta característica faz com que possamos ter um CPU a correr em *periodic mode* e outro em a correr em *one-shot mode*. Ao criarmos a tarefa temos de indicar que processador a tarefa vai usar [7].

Qualquer um destes escalonadores permite utilizar, quer o modo *one-shot*, quer o modo *periodic* nas tarefas de tempo real. A diferença entre estes dois modos é a forma como estão relacionados com o processo de programação do temporizador/contador (*timer*). Estes processos de programação do *timer* têm origem nas arquitecturas dos PCs, onde o temporizador/contador (*timer/counter*) 8254[9] pode ser programado de várias formas. Em tais arquitecturas, o *timer/counter* pode ser programado para gerar interrupções com intervalos de tempo fixos, modo periódico (*periodic mode*), ou ser reprogramado a cada interrupção, modo *one-shot*. Os três micro segundos que poupa, por não ser necessário reprogramar o *timer* no modo periódico, pode levar a pensar que este modo seja o ideal para ser utilizado em qualquer situação. No entanto, o modo periódico pode não ser a solução que melhor se adapte a situações em que os intervalos de tempo possam variar. Neste caso, por exemplo, seria melhor usar o modo *one-shot*. Este modo, ao contrário do que acontece no modo periódico, permite também adicionar tarefas sem que haja preocupação com o período fundamental. O modo *one-shot* tem ainda a vantagem de reprogramar os intervalos de tempo usando a frequência do CPU, uma vez que do ponto de vista da arquitectura do PC, a frequência do *timer/counter* é consideravelmente mais baixa [7]. No entanto, como no modo *one-shot* a interrupção é recalculada a cada ciclo da tarefa, isto leva a que o hiato do tempo da reprogramação ocorra continuamente. O que é uma desvantagem face ao modo periódico.

Para implementar uma tarefa de tempo real em modo *periodic* deve-se, em primeiro lugar, configurar o período fundamental no *timer/counter*. De seguida, escrever a função-tarefa e definir a estrutura de tarefas, onde se especifica a prioridade da tarefa, o tamanho da *stack*, e a restante informação para a tarefa funcionar. Por fim, escalonar a tarefa, para que esta seja executada num determinado período, que é múltiplo do período do *timer/counter* [16].

Ao definir o período fundamental no *timer/counter* deve-se proceder da seguinte forma:

```
rt_set_periodic_mode();
RTIME start_rt_timer(RTIME period);
```

[9] O 8254 é um *timer/counter* desenvolvido para ser usado nos sistemas de microcomputadores Intel. É um elemento *multi-timing* de aplicação geral que pode ser utilizado como um vector de portos I/O no software do sistema. O 8254 resolve um dos mais comuns problemas nos sistemas de microprocessadores: A geração com precisão de atrasos temporais sob o controlo de software[12].

Com estas duas funções configura-se o temporizador/contador (*timer*) para funcionar em modo periódico puro. A primeira função, garante que o temporizador/contador não vai ser reprogramado após cada escalonamento da tarefa, ou seja, garante a periodicidade da tarefa. O valor que é passado para a segunda função, '`start_rt_timer()`', é o período, do tipo "unidades internas[10]". A função '`RTIME nano2counts(int nanoseconds)`' pode ser usada para converter o tempo, representado em nano segundos, para as "unidades internas". O período então requisitado é quantificado em função da frequência do temporizador/contador (*timer*) 8254 (1,193,180Hz), e qualquer temporização requisitada que não seja um múltiplo inteiro do período, é satisfeita pelo valor de período inteiro imediatamente mais próximo.

A função-tarefa (*task function*) é uma função que é chamada quando a tarefa de tempo real é escalonada para ser executada. Esta função é normalmente escrita em linguagem C, e tipicamente tem o seguinte comportamento: lê entradas, processa-as, e apresenta os resultados nas saídas e espera pelo próximo ciclo. O código típico de uma função deste género tem mais ao menos o seguinte aspecto:

```
void task_function(int arg)
{
   while(1){

/* Colocar aqui as instruções da tarefa*/

rt_task_wait_period();
   }
   return;
}
```

Como se pode ver no extracto do código anterior, a função-tarefa estará pronta a realizar um conjunto de instruções, em modo periódico, enquanto o módulo que corresponde à tarefa de tempo real estiver carregado no kernel. A função `void rt_task_wait_period(void)` tem como finalidade esperar pelo próximo ciclo do escalonador, para depois executar novamente as instruções definidas na função [16].

[10] Os valores definidos para a tarefa estão em milissegundos, mas o RTAI usa uma unidade interna (*count*) que é múltipla do TSC (Time Clock Stamp). Existem macros que permitem a conversão de valores nos dois sentidos, ou seja, de *count* para nanossegundos e vice-versa

Após definir a função-tarefa, é necessário configurar a estrutura responsável por guardar a informação sobre a tarefa de tempo real. Essa estrutura de dados, RT_TASK, é responsável por guardar a seguinte informação:

- A função-tarefa que irá realizar a tarefa.
- Um argumento inicial, que se lhe poderá passar.
- O tamanho da *stack* que é necessário para alocar as variáveis.
- A prioridade.
- Se usa vírgula flutuante ou não.
- Uma "*signal handler*" que é chamada na altura em que a tarefa se torna activa.

A estrutura em questão é então criada e inicializada através da seguinte função disponibilizada pelo RTAI [16]:

```
rt_task_init(RT_TASK *task,
             void *rt_thread, int data,
             int stack_size, int priority,
             int uses_fp, void *sig_handler);
```

Os argumentos desta função, que conforme se viu anteriormente, são responsáveis por guardar informação sobre a tarefa, e são descritos com mais pormenor nos pontos seguintes:

- '*task*' é um ponteiro para a estrutura do tipo RT_TASK, cujo espaço tem de ser fornecido pela aplicação. Tem de ser mantido durante todo o tempo de vida da tarefa.
- '*rt_thread*' é o ponto de entrada da nova função-tarefa.
- '*data*' é um único valor inteiro que é passado para a nova tarefa.
- '*stack_size*' é o tamanho da *stack*[11] a ser usada pela nova tarefa.

[11] Uma '*stack*' é uma lista ordenada ou uma estrutura de dados na qual o modo do acesso aos seus dados é do tipo LIFO (*Last In*, *First Out*) usada para colocar e restaurar dados. Esta estrutura é aplicada muitas vezes no campo da computação devido à sua simplicidade e ao seu funcionamento transparente.

- '*priority*' corresponde à prioridade de uma determinada tarefa. O valor mais alto é '0', e o mais baixo é 'RT_LOWEST_PRIORITY' (1,073,741,823).

- '*uses_fp*' é uma *flag*, em que um valor diferente de zero indica que a tarefa irá usar vírgula flutuante, e que o escalonador deverá efectuar um esforço extra para guardar e restaurar os registos de vírgula flutuante.

- '*sig_handler*' é uma função, que é chamada dentro do ambiente da tarefa e com as interrupções desactivadas, quando a tarefa se tornar a tarefa em execução após uma mudança de contexto.

Inicialmente, a tarefa de tempo real está num estado de suspensão, logo, é necessário escalonar a tarefa para que se torne activa e possa ser executada. Para a activar pode-se usar as seguintes funções disponibilizadas pelo RTAI: `rt_task_make_periodic()`, `rt_task_make_periodic_relative_ns()`,`rt_task_resume()`.

Apesar de ser possível usar qualquer uma das funções atrás indicadas para escalonar a tarefa, a função normalmente utilizada é:

```
int rt_task_make_periodic(RT_TASK *task,
                          RTIME start_time,
                          RTIME period);
```

onde:

- '*task*' é o endereço da estrutura RT_TASK.

- '*start_time*' é o tempo absoluto, em unidades RTIME, em que a tarefa deve iniciar a sua execução.

- '*period*' é o período da tarefa, em unidades RTIME, que é arredondado para o valor mais próximo do período fundamental do *timer* (temporizador/contador).

Ao declarar a função atrás descrita, esta-se na realidade a marcar a tarefa de tempo real, que foi previamente configurada na função `rt_task_init()`, como pronta para ser executada em modo periódico. O tempo da primeira execução da tarefa é dado por argumento da função *start_time*, que é valor absoluto medido em impulsos de relógio (*clock ticks*) [16].

Resumindo, para programar uma tarefa em tempo real, em modo periódico, existem quarto fases importantes. A definição do *timer*, a associação da função-tarefa à tarefa de tempo real, a definição da estrutura responsável por guardar a informação sobre a tarefa, e por fim, o escalonamento da tarefa.

Para além do modo periódico, o escalonador do RTAI permite ainda o uso do modo *one-shot*. A configuração do modo *one-shot* não difere muito, em termos estruturais, do modo periódico (*periodic mode*), ou seja, também é necessário definir o *timer*, a associação da função-tarefa à tarefa de tempo real, definir a estrutura responsável por guardar a informação sobre a tarefa, e por fim, escalonar a tarefa.

Como a configuração do modo *one-shot* não difere muito do modo periódico, e para não me tornar repetitivo, serão abordados apenas os pontos onde ambos os modos diferem. Assim sendo, tal como foi feito para o modo periódico, em primeiro lugar é necessário configurar o *timer* (temporizador/contador). O *timer*, em modo *one shot* é configurado utilizando a seguinte função do RTAI:

```
void rt_oneshot_mode(void);
```

Com esta instrução o *timer* 8254, vai ser reprogramado a cada ciclo, e usando a função:

```
RTIME start_rt_timer(RTIME count);
```

Inicia-se o *timer* no modo *one-shot*. Se não tiver um valor para o argumento '*count*', da função `start_rt_timer()`, pode-se usar '1', pois convêm usar um valor diferente de zero, porque em determinadas situações '0' pode significar desabilitar o *timer*.

A partir deste momento, a configuração da tarefa de tempo real em modo *one-shot* é idêntica ao modo *periodic*. Tem de se configurar a função '`rt_task_init()`' para guardar na estrutura, RT_TASK, a informação sobre a tarefa de tempo real, e também a função '`rt_task_make_periodic()`' que é responsável por iniciar a tarefa em modo *one-shot*.

Já na função-tarefa, ao contrário do modo periódico, podem surgir duas situações diferentes. Pode-se usar '`rt_sleep_delay(delay)`', em que o atraso (*delay*), é o atraso que se quer que a tarefa tenha. Ou usar '`rt_task_wait_period()`', como foi feito para o modo *periodic* que irá fazer com que tarefa fique adormecida durante o período de tempo que for definido, ou seja, até chegar o novo ciclo do escalonador [16].

A função-tarefa teria assim uma estrutura idêntica à que foi apresentada para o modo *periodic*, mudando apenas as instruções em função do tipo de utilização que se pretende.

3.3.3. POLITICAS DE ESCALONAMENTO

O *Real Time application Interface* (RTAI) fornece um conjunto de funções que permite definir as políticas de escalonamento das tarefas de tempo real. A política de escalonamento pode ser seleccionada por tarefa. Independentemente da política de escalonamento, o RTAI é preemptivo e baseado em prioridades. Ou seja, no RTAI a tarefa de maior prioridade é executada, e as de menor prioridades são bloqueadas.

As políticas de escalonamento suportadas pelo RTAI são as seguintes: *First In Fisrt Out* (FIFOS), *Round Robin* (RR), *Rate Monotonic* (RMS), *Deadline monotonic* (DMS), *Earliest Deadline First* (EDF). A politica de escalonamento padrão do RTAI é o FIFO, em que as tarefas de menor prioridade são "preemptadas" pelas de maior prioridade, e as tarefas com a mesma prioridade da tarefa em execução, só são executadas quando a tarefa em execução libertar o CPU. A política de escalonamento RR só está disponível caso esta tenha sido habilitada na configuração do RTAI. Neste tipo de política as tarefas podem ser "preemptadas" por tarefas de maior prioridade, ou por tarefas com a mesma prioridade cujo *quantum* expirou. Esta política pode ser atribuída a cada tarefa usando a função:

```
rt_set_sched_policy(RT_TASK *task, int policy, int rr_quantum_ns);
```

em que, a politica pode ser definida como `RT_SCHED_FIFO` ou `RT_SCHED_RR`, no segundo argumento. O *quantum*, é definido no terceiro argumento, e caso `rr_quantum_ns = 0`, a política de escalonamento RR usa o impulso (*tick*) do Linux. Vista como uma caso particular da política de escalonamento FIFO, a RMS tem apenas uma tarefa por cada nível de prioridade, em que a prioridade mais elevada pertence a tarefa com menor período de execução. No entanto, pode ser complicado usar esta política de escalonamento com tarefas dinâmicas, modo *one-shot*, porque neste tipo de tarefas é complicado garantir apenas uma tarefa em cada nível de prioridade. Para garantir que as prioridades estão correctamente atribuídas utiliza-se a função: `rt_spv_RMS(cpu)`. Uma outra política de escalonamento que pode ser vista como um caso particular das FIFO é a DMS. Na DMS, também só tem uma tarefa em cada nível de prioridade, no entanto, nas DMS, a maior prioridade é atribuída a tarefa com menor *deadline* [17]. Ao contrário da política de escalonamento da *Rate Monotonic* (RMS), não existe nenhuma função para

ajudar a escalonar correctamente as tarefas, visto que, na criação das tarefas o sistema não é informado sobre as *deadlines*. Por fim, o *Earliest Deadline First* (EDF), em que a tarefa com o *deadline* mais próximo é que é escalonada. Nesta politica, EDF, o escalonador precisa de saber quando a tarefa irá ser de novo escalonada e o seu *dedline*. Portanto, aquando da criação da tarefa, em vez de se usar a função `rt_task_wait_period()`, usa-se a função:

```
void rt_task_set_resume_end (RTIME resume_time, RTIME end_time);
```

Caso os argumentos da função anterior sejam negativos, estes são interpretados como relativos aos valores anteriores.

O RTAI, pode ainda, no mesmo sistema, escalonar e "misturar" tarefas com políticas de escalonamento diferente. As políticas de escalonamento FIFO, RR, RMS e DMS podem ser "misturadas" sem problemas, desde que o programador garanta que as prioridades são atribuídas de forma coerente em função do efeito desejado. A política de escalonamento EDF não foi incluída no grupo anterior, devido ao seu escalonamento não ser orientado à prioridade, logo a interacção com as outras políticas de escalonamento, FIFO, RR, RMS ou DMS, não é tão uniforme. Esta falta de uniformidade, deve-se ao facto do RTAI escalonar as tarefas com política de escalonamento EDF com maior prioridade que as outras. Na realidade não é que tenha maior prioridade. As EDF são executadas logo que o instante de libertação expira, independentemente do nível de prioridades das tarefas com políticas FIFO, RR, RMS ou DMS [17].

3.4. MEIOS DE COMUNICAÇÃO ENTRE TAREFAS E PROCESSOS

Uma das principais características do RTAI é a diversidade de meios de comunicação e serviços que disponibiliza aos programadores através dos seus escalonadores, mantendo no entanto interfaces idênticos em todos eles. Alguns destes serviços fazem parte dos módulos dos escalonadores, pois são simples de implementar, e permitem a utilização de meios de comunicação básicos. Existem, no entanto, outros meios de comunicação mais complexos e menos comuns, que têm de ser implementados com módulos próprios e independentes.

Para melhor entendimento dos serviços disponibilizados pelo RTAI, estes serão descritos nas subsecções seguintes.

3.4.1. COMUNICAÇÃO POR MAILBOXES

A *mailbox* é porventura o método de intercomunicação entre processos (IPC) mais flexível, pois é um *buffer* gerido pelo SO [15], que permite que um determinado número de mensagens com tamanhos diferentes sejam colocadas em fila. As características que tornam as *mailboxes* tão flexíveis são as seguintes:

- Uma tarefa pode enviar/receber mensagens para/de para uma *mailbox*, e múltiplas tarefas podem receber ou enviar mensagens de/para a mesma *mailbox*.

- Uma tarefa que esteja a receber mensagens vai lê-las por ordem de chegada (existem variações deste principio, como será mostrado mais à frente).

- Para usar comunicação bidireccional são necessárias duas *mailboxes*, uma para cada direcção.

A implementação original usa o princípio das FIFOs, no entanto foram desenvolvidas políticas de prioridade e gestão que aumentam a flexibilidade das *mailboxes*:

- Incondicionalidade: A tarefa aguarda até que toda a mensagem seja enviada (por defeito encontra-se nesta situação).

- *Best-effort*: Só são enviados os bytes que podem passar de imediato (usando a extensão '_wp' na função em questão).

- Condicionada à disponibilidade: Só envia a mensagem caso toda a mensagem possa ser enviada de imediato (usando a extensão '_if' na função em questão).

- Temporizada: Com *timeouts*, relativos ou absolutos (usando as extensões '_up' ou "_timed", conforme as necessidades, na função em questão).

Nos pontos seguintes é apresentada uma amostra dos serviços disponíveis nas *mailboxes*:

- `Rt_mbx_init()` Inicializa a *mailbox* com um determinado tamanho.

- `Rt_mbx_delete()` Apaga os recursos usados pela *mailbox*.

- `Rt_mbx_send()` Envia uma mensagem com tamanho definido para uma determinada *mailbox*.

- `Rt_mbx_recieve()` Recebe uma mensagem com tamanho definido para uma determinada *mailbox*.

Para criar uma *maibox* no espaço de kernel proceder-se-ia, por exemplo, da seguinte forma para inicializar a *mailbox*:

```
int rt_mbx_init (MBX *mbx, int size);
```

Onde se definia o apontador para a estrutura *mailbox* ("mbx"), que funciona como "nome" da *mailbox*, e o tamanho da *mailbox*, "*size*".

Pode-se também especificar a ordem com que as tarefas são postas em fila na mailbox para serem enviadas: FIFO *order* (FIFO_Q), *priority order* (PRIO_Q) ou *resource order* (RES_Q). Neste caso inicializaríamos a *maibox* da seguinte maneira:

```
int rt_typed_mbx_init (MBX *mbx, int size, int
type);
```

Os dois primeiros parâmetros são idênticos ao da função anterior, e o ultimo corresponde à forma como as tarefas serão ordenadas na *mailbox*.

Para enviar e receber uma mensagem, de forma incondicional, usa-se as seguintes funções:

```
int rt_mbx_send (MBX *mbx, void *msg, int
msg_size);
```

Neste caso, envio de dados, defini-se o apontador para a estrutura, ou seja, o "nome" da *mailbox*, a mensagem a enviar e o tamanho da mensagem, respectivamente.

A recepção é idêntica em termos de parâmetros, mudando apenas o nome da função:

```
int rt_mbx_receive(MBX *mbx,  void *msg, int
msg_size);
```

Convêm ainda salientar duas coisas. Primeiro, que o tamanho da mensagem (*"msg_size"*) na função de envio e de recepção tem de ser iguais. Segundo, que as duas funções acima descritas podem ser aplicadas as politicas de gestão e prioridade que foram descritas no inicio desta sub-secção *('_wp','_if','_up' or '_timed')*.

Para remover a *mailbox* usa-se a função:

```
rt_mbx_delete(MBX *mbx);
```

3.4.2. MENSAGENS E REMOTE PROCEDURE CALLS

Ao contrário das *mailboxes* as mensagens e RPCs (*Remote Procedure Calls*) são orientadas às tarefas, ou seja, ou envia ou recebe uma mensagem, de e para uma tarefa. Deste modo não é necessário inicializar estruturas ou identificadores, sendo apenas necessário garantir que o receptor ou emissor da mensagem seja uma tarefa activa, porque todas as chamadas para esta funcionalidade requerem a passagem de um ponteiro para a tarefa em questão. As mensagens têm tamanho fixo e são armazenadas em variáveis inteiras sem sinal.

De seguida indica-se uma amostra dos serviços disponibilizados pelo interface RPC:

- `rt_send()` Envia uma mensagem para uma determinada tarefa.
- `rt_recieve()` Recebe uma mensagem de uma determinada tarefa.
- `rt_rpc()` Envia uma mensagem para uma determinada tarefa e fica a espera de resposta.
- `rt_isrpc()` Determina se uma determinada tarefa está a espera de uma resposta de um RPC.
- `rt_return()` Resposta para um RPC de uma determinada tarefa.

Alguns destes serviços têm politicas condicionais e temporais, como por exemplo:

- `rt_send_timed()` Igual ao *rt_send()* mas com *timeout*.
- `rt_recieve_timed()` Igual ao *rt_recieved()* mas com *timeout*.
- Extensão `_if` Se usarmos esta extensão só passa a mensagem caso ela possa ser feita toda de uma vez.

Um exemplo simples do uso de mensagens para comunicar entre tarefas seria:

Se se quiser enviar uma mensagem para uma tarefa sincronizadamente. Bloqueia até que o receptor possa receber a mensagem:

```
RT_TASK * rt_send (RT_TASK *task, unsigned int
msg);
```

Onde os parâmetros da função são, um apontador para a tarefa e a mensagem.

Para receber a mensagem de uma tarefa. Bloqueia até que a mensagem esteja disponível:

```
RT_TASK * rt_receive (RT_TASK *task, unsigned int
*msg);
```

Os parâmetros são idênticos ao da função anterior, no entanto se no parâmetro *'task'* for definido como zero, esta função aceita mensagens de qualquer tarefa.

3.4.3. FISRT IN FIRST OUT – FIFOS

O mais antigo e básico mecanismo de comunicação entre processos do RTAI são as FIFOSs (*Fisrt In First Out*) [7], e para as poder usar é necessário carregar o respectivo modulo [8]. A FIFO é um canal de comunicação de um sentido, assíncrono, e sem bloqueio entre um processo do Linux e uma tarefa de tempo real, e cujo limite do tamanho é dado pelo utilizador. Na realidade, as FIFOs permitem que as tarefas coloquem dados num *buffer*, e que estes sejam lidos por ordem de chegada: os primeiros a entrar no *buffer* são os primeiros a ser lidos - *First In First Out* (FIFO) -. Na FIFO não é possível sobrepor dados, mas no entanto a FIFO pode ficar cheia, e neste caso os novos dados não podem ser escritos até que os antigos sejam processados. Os limites dos segmentos de dados a escrever na FIFO não são preservados e é da responsabilidade do leitor da FIFO resolve-lo caso seja necessário.

Quando é usada a partir do módulo do kernel a API da FIFO identifica a FIFO através da sua identificação (ID). No espaço do utilizador, esta ID corresponde a uma entrada na directoria `/dev` [7]. Por exemplo, se a FIFO fosse identificada no modulo de kernel como FIFO "1" esta corresponderia na directoria de *devices* a `/dev/rtf1` no espaço de utilizador.

A principal utilização das FIFOs é partilhar a informação de dados relativos às tarefas de tempo real entre o espaço do kernel e o espaço do utilizador. Ou seja, usando as FIFOs consegue-se transmitir os dados obtidos pelas tarefas de tempo real e torná-los acessíveis a uma aplicação normal do Linux, que não é dependente de exigências de tempo real. Por exemplo, pode ser usada na aquisição de dados, em que uma tarefa pode comunicar através deste processo usando as funções convencionais *open*, *read*, *write*, *close* e outros serviços fornecidos pelas APIs dos ficheiros do Unix, pois para a aplicação do utilizador, no espaço de utilizador, a FIFO comporta-se como um qualquer outro ficheiro do sistema.

Exemplo de algumas API disponíveis:

- `rtf_create()` Cria uma FIFO com respectivo nome e tamanho.
- `rtf_destroy()` Elimina a FIFO.
- `rtf_reset()` Limpa o conteúdo da FIFO.
- `rtf_put()` Coloca dados na FIFO.
- `rtf_get()` Obtem dados da FIFO.
- `rtf_create_handler()` Associa uma *handler*[12] para lidar com dados escritos ou lidos pela FIFO.

Para complementar estes serviços foram adicionadas primitivas de sincronização, semáforos, para sincronizar o acesso às FIFOs. Outra característica interessante é a possibilidade de atribuir nomes às FIFOs, aumentado assim a sua flexibilidade, pois usando esta característica, a sua identificação no espaço de utilizador, depende apenas do nome que lhe foi atribuído.

Tal como outras APIs, as FIFOs também dispõem de políticas condicionais e temporais, como por exemplo:

- `rtf_read_timed()` Lê os dados da FIFO no espaço de utilizador, com tempo de atraso.
- `rtf_write_timed()` Equivalente à anterior só que para escrita.
- `rtf_suspended_timed()` Suspende a função por um determinado período de tempo.
- `rtf_read_all_at_once()` Lê um bloco de dados de uma FIFO de tempo real.

Conjugando as normais leituras/escritas na FIFO, com os exemplos acima descritos, é fácil implementar I/Os (*Inputs*/*Outputs*) bloqueados (*blocked*), não bloqueados (*non-blocked*) e

temporizados (*timed*), ou seja, é possível gerir o fluxo de dados, mas como não estão padronizadas, não são portáveis, no entanto, é muito mais fácil de usar que um mecanismo *select/pull*[13].

Numa situação real em que fosse necessário usar uma FIFO para, por exemplo, comunicar entre as tarefas que correm ao nível do espaço do kernel, e os processos que correm ao nível do espaço do utilizador, primeiro seria necessário configurá-las ao nível do espaço do kernel e depois no espaço do utilizador:

```
int rtf_create (unsigned int fifo, int size);
```

A expressão anterior mostra como criar a FIFO ao nível utilizador. O primeiro parâmetro refere-se ao número com que se identifica a FIFO (de 0 a 63), e o segundo parâmetro ao tamanho da FIFO, em *bytes*.

No espaço de utilizador os números da FIFO de 0 a 63 estão associados a `/dev/rtf0` até `/dev/rtf63`, que são descritores criados quando se configura o RTAI. Desta forma para criar uma FIFO basta usar a função standard '`open()`' do Linux:

```
file_descriptor = open("/dev/rtf0", O_RDONLY);
```

É retornado um inteiro (*file_descriptor*) usado para identificar a FIFO no processo de leitura/escrita. Também é possível usar os parâmetros O_RDONLY (só de leitura), O_WRONLY (só para escrita) e O_RDWR (leitura e escrita), que são parâmetros conhecidos e usados no Linux.

No espaço do kernel, a leitura e escrita são realizadas da seguinte forma:

```
num_read = rtf_get(0, &buffer_in,
sizeof(buffer_in)); \\ Leitura
num_written = rtf_put(1, &buffer_out,
sizeof(buffer_out)); \\ Escrita
```

Na função de leitura o último parâmetro corresponde ao tamanho do *buffer* de entrada, que especifica o número de *bytes* que se podem ler. Caso seja necessário, é possível verificar os valores de retorno dos bytes lidos ou escritos, e caso haja valores negativos, significa que existem erros [15].

---

[12] O *Handler* é um apontador numa função que irá ser chamada sempre que um processo do Linux executar a FIFO

[13] As FIFOs permitem múltiplos leitores/escritores e usam este mecanismo para gerir a entrada/saída de dados na FIFO.

No lado do Linux, espaço de utilizador, a leitura e escrita são efectuadas usando as funções normais do Linux '`read()`' e '`write()`':

```
num_read = read(read_descriptor, &buffer_in,
sizeof(buffer_in));\\ Leituta
num_written = write(write_descriptor,
&buffer_out, sizeof(buffer_out));\\ Escrita
```

O '*write descriptor*' e '*read descriptor*' são os descritores retornados pela chamada '`open()`' que falamos atrás.

Para remover a FIFO utiliza-se a seguinte função:

```
rtf_destroy(unsigned int)
```

3.4.4. MEMORIA PARTILHADA

A memória partilhada permite um paradigma alternativo à FIFO[8] quando é necessário um modelo de comunicação alternativo. A memória partilhada é um bloco de memória que pode ser lido ou escrito por qualquer processo ou tarefa do sistema Linux/RTAI. Como os diferentes processos podem operar assincronamente na memória partilhada, o problema centra-se em garantir que os dados que estão na memória partilhada não são sobrepostos acidentalmente. Uma outra questão que se coloca é o facto do RTAI poder atender uma tarefa com maior prioridade e interromper um processo de leitura ou escrita, pondo em causa a integridade dos dados. Para evitar este problema é necessário garantir através de um mecanismo que o processo de leitura ou escrita não são feitos em simultâneo, nem que são interrompidos enquanto decorrem, garantindo assim a consistência dos dados.

Tal como nas FIFOs este é um módulo adicional, e uma vez carregado, tanto pode ser usado nas aplicações que correm no espaço do utilizador, como nas tarefas de tempo real, que correm ao nível do espaço do kernel. Este módulo permite a alocação e libertação de zonas de memória. Estas zonas de memória são identificadas através de um esquema de nomeação, que garante que na próxima alocação, com o mesmo nome, apenas irá resultar no mapeamento de uma determinada zona para o mapa de memória do processo, e fornece à *caller* o apontador para a zona em questão.

A memória partilhada apresenta as seguintes características:

- A memória partilhada é normalmente usada para guardar uma estrutura de dados em que os membros são lidos e escritos por dois ou mais processos.

- Ao contraio das FIFOs os dados da memoria partilhada não são postos em fila. Qualquer dado escrito para a memória partilhada sobrepõe o anterior.

- A memória partilhada também pode ser acedida de uma forma seccionada. Membros individuais de um conjunto total da memória partilhada podem ser escritos e lidos de forma independente.

- Como a memória partilhada é um recurso partilhado, tem de garantir a consistência de dados. Existem duas hipóteses para o par leitor-escritor:

    - A tarefa de tempo real é o leitor e interrompe o processo de Linux durante a sua escrita. Neste caso, a tarefa de tempo real vai ver uma estrutura de dados inconsistente, com dados novos no inicio e antigos no fim.

    - A tarefa de tempo real é o escritor, e interrompe o processo de Linux durante uma leitura. Neste caso, é processo de Linux irá ver inconsistência na estrutura de dados, com dados antigos no inicio e novos no fim.

3.4.5. GESTÃO DE MEMÓRIA PARTILHADA

Em versões anteriores do RTAI/Linux a memória tinha de ser alocada estaticamente, pois a alocação em tempo real não era possível. No entanto, nas versões actuais do RTAI/linux já vem incluído um módulo de gestão de memória, que permite alocação dinâmica de memória em tempo real usando um interface similar à *standard* C *library* [8].

O RTAI pré-aloca fragmentos de memória, configuráveis em número e tamanho, antes da execução da tarefa de tempo real. Quando uma tarefa de tempo real chama a função `rt_malloc()`, a memória requisitada é fornecida pelo bloco de memória pré-alocado. Quando a quantidade de memória livre no bloco de memória é inferior ao valor limite (*threshold*), um novo bloco de memória é reservado para futuras alocações. Do mesmo modo quando a função `rt_free()` é chamada, o espaço ocupado pela memória libertada fica livre no "pedaço" de memória alocado, e fica à espera para reservas futuras. Quando a quantidade de memória libertada é superior a um limite pré-definido, o bloco de memória é libertado.

Para implementar a comunicação através de memória partilhada o RTAI fornece um conjunto de funções que permite faze-lo de uma forma simples e transparente como se pode ver de seguida:

```
struct mystruct *ptr;
ptr = rtai_kmalloc (101, sizeof(struct
mystruct));
```

No extracto de código anterior declararam-se às instruções para alocar um bloco de memória partilhada no espaço de kernel (RTAI). O primeiro argumento da função '`rtai_kalloc()`' é uma "chave" que identifica a zona de memória partilhada, ou seja, um outro processo que queira aceder a esta zona de memória tem de usar esta chave. O segundo argumento é o tamanho da estrutura, para saber o espaço a reservar.

Para aceder a esta zona de memória no espaço de utilizador (Linux), usa-se a instrução '`rtai_malloc()`' que faz a conexão com a memória partilhada, e é declarada da seguinte forma:

```
struct mystruct *ptr;
ptr = rtai_malloc (101, sizeof(struct mystruct));
```

Comparando a ultima expressão com a anterior verifica-se que os parâmetros são os mesmos, pois caso contrário, não se conseguia aceder ao pedaço de memória pretendido. A função '`rtai_malloc()`' retorna um apontador para o pedaço de memória partilhada anteriormente por '`rtai_kalloc()`'. Uma vez criada, o ponteiro da memória partilhada pode ser referenciado como qualquer outro ponteiro, fazendo com que os dados estejam imediatamente disponíveis para todos os processos que partilhem este pedaço de memória.

Para libertar o pedaço de memória alocado, é necessário chamar a função `rt_kfree()` ao nível do kernel, ou seja na tarefa de tempo real:

```
Rt_kfree(101);
```

Em que o parâmetro '101' é a chave que foi usada quando para criar a zona de memoria partilhada.

A zona de memória também tem de ser libertada no espaço de utilizador. Neste caso a função seria declarada da seguinte forma:

```
Rt_kfree(101, ptr);
```

Em que o primeiro argumento é idêntico ao da instrução anterior. E corresponde a chave do bloco de memória, e o segundo corresponde ao ponteiro para a zona de memória.

Estas instruções devem ser executadas na ordem inversa à criação da memória partilhada, ou seja, primeiro deve-se libertar a memória partilhada no processo de Linux (espaço de utilizador) e depois ao nível do kernel (tarefa de tempo real).

3.5. OUTROS SERVIÇOS

3.5.1. O WATCHDOG

Para que o RTAI se tornasse um ambiente de programação mais seguro, foi implementado um *watchdog*[14]. Esta garante que uma tarefa não irá bloquear o sistema devido a tarefas com comportamentos inesperados. Usando o *watchdog*, garante-se que a repetição infinita (*loops*) e os *scheduling overuns*[15] das tarefas não afectem a capacidade do sistema de continuar a realizar as suas operações, pois é implementado um meio de reacção a tais ocorrências que permite suspender tarefas problemáticas, ou mesmo eliminá-las [7].

O *watchdog*, tem então, como principal função, monitorizar as tarefas para evitar que estas ultrapassem o seu período. Logo, o *watchdog*, deve ser programado com um período menor e com prioridade mais alta que as outras tarefas. O modo do *timer* utilizado pelo *watchdog* deve ser o mesmo das tarefas, e deve ser inicializado pelo *watchdog*. Existe ainda um ficheiro onde são registadas todas a tarefas com problemas. Esse ficheiro, encontra-se em `/proc/rtai/watchdog`.

Conforme já visto noutros serviços, também existem para o *watchdog* várias políticas de funcionamento. Estas políticas são [17]:

- *Nothing*: Faz apenas o registo das tarefas que não cumpriram o período. Não recomendado.
- *Resync* a tarefa: Para tarefas que possam estar sujeitas a *overrun*. Recomendado para *overrruns* ocasionais.

---

[14] É um hardware ou software temporizador que acciona uma reinicialização do sistema, ou uma acção correctiva numa determinada aplicação.

[15] Quando uma tarefa é de novo chamada sem que tenha tido tempo de terminar o seu serviço.

- *Debug*: Caso especial da politica de resincronizar. Tem o *limit* e o *safety* desabilitados. Recomendado para *debug* de tarefas *oneshot*.

- *Stretch*: Incrementa o período da tarefa até deixar de haver *overun*. Recomendado para tarefas em constante *overrruning*.

- *Slip*: Suspende a tarefa por uma fracção de tempo. O tempo de suspensão e o número máximo de suspensões pode ser configurado. Recomendado para *loops* infinitos.

- *Suspend*: Suspende a tarefa. A tarefa não volta a ser executada, no entanto, continua a existir no escalonador, pois pode voltar a ser "acordada". Recomendado para a maior parte das situações.

- *Kill*: Remove a tarefa. A tarefa deixa de existir no escalonador. Sentença de morte.

Para configurar o *watchdog* existe um conjunto de parâmetros que são usados pelas APIs [17]:

- *TickPeriod*: Período em nanosegundos.

- *Wd_Oneshot*: Coloca em modo *one shot*.

- *Grace*: Quantidade de períodos de *overrun* permitidos.

- *GraceDiv*: O mesmos que o anterior, mas para valores menores que 1

- *Safety*: Limite de segurança para suspender qualquer tarefa. Normalmente tarefas que superem cem vezes o período são suspensas.

- Policy: *Nothing* = 0; *Resync* = 1; *Debug* = 2; *Streatch* = 3; *Slip* = 4; *Suspend* = 5; *Kill* = 6.

- *Streatch*: Percentagem do período a ser incrementado.

- *Slip*: Percentagem do período pelo qual a tarefa será suspensa.

- *Limit*: Quantidade de períodos de *overrrun* que são permitidos, antes que a tarefa seja forçosamente suspensa.

As API, disponibilizadas pela libraria rtai_wd.h, que permite atribuir os parâmetros são as seguintes:

```
wd_policy rt_wdset_policy(wd_policy new_value);
/*Onde "wd_policy" pode ter um dos seguintes
parâmetros NOTHING, RESYNC, DEBUG, STRETCH, SLIP,
SUSPEND, KILL*/

int rt_wdset_grace(int new_value);

int rt_wdset_gracediv(int new_value);

int rt_wdset_slip(int new_value);

int rt_wdset_stretch(int new_value);

int rt_wdset_limit(int new_value);

int rt_wdset_safety(int new_value);
```

3.5.2. SINCRONIZAÇÃO: SEMÁFOROS

Os semáforos são mecanismos de sincronização simples, e permitem que várias tarefas coordenem o seu trabalho de forma coerente [7]. No RTAI os semáforos são identificados usando as suas estruturas. No entanto, as tarefas que aguardam a sincronização usando um semáforo, vão necessitar de um ponteiro para essa estrutura de semáforo. Nas restantes características, o comportamento dos semáforos do RTAI é idêntico aos semáforos convencionais.

Apresenta-se de seguida uma amostra das APIs (*Application Program Interface*) do semáforo:

- `rt_sem_init()` Inicializa o semáforo num determinado valor.
- `rt_sem_delete()` Elimina o semáforo.
- `rt_sem_signal()` Sinaliza o semáforo.
- `rt_sem_wait()` Espera no semáforo.

A API `rt_sem_wait()`têm políticas condicionais e temporais que podem ser de grande utilidade em algumas situações. De seguida fica um exemplo dessas APIs:

- `rt_sem_wait_if()` Toma o semáforo "se" possível. É similar ao `rt_sem_wait()` mas nunca bloqueia a *caller*.

- `rt_sem_wait_until()`, `rt_sem_wait_timed()`, Espera no semáforo com *timeout*, ou temporizado .

3.5.3. POSIX PTHREADS

O RTAI contém um módulo de *pthreads* que implementa as *threads* segundo a norma POSIX 1003.1c e parte da norma 10031.b, a fila de mensagens. Utilizando as operações especificadas nas normas, os utilizadores podem gerir as *threads* com propriedades idênticas às *threads* POSIX convencionais. Deve-se ter em conta que a implementação das *pthreads* no RTAI ainda não suporta os conceitos de *joining* e *detaching* [8].

As *threads* sob o mesmo processo partilham o mesmo espaço de memória, tornando fácil a troca de informação entre elas. No entanto, a área de memória partilhada utilizada pelas *threads* para trocar informação tem de estar sincronizada. No que diz respeito à sincronização, para além dos processos descritos anteriormente, é possível o uso de *mutexes* nas *pthreads* implementadas pelo RTAI de forma a garantir exclusão mútua no código da aplicação, bem como o uso variáveis condicionais para ajudar à sincronização.

3.6. INTERFACE LXRT

Entre todos os serviços e abstracções disponibilizadas pelos módulos do RTAI, o LXRT é o mais flexível e também o mais complicado. Ao tornar disponível para o programador um interface de programação simétrica, faz com que o LXRT reúna as melhores características dos SOUG e dos SOTR, permitindo que se comunique de uma forma transparente com as tarefas de tempo real, e vice-versa, através do espaço de utilizador.

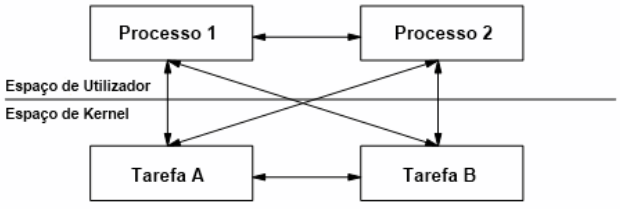

Figura 13 Comunicação entre tarefas e processos em diferentes domínios

A Figura 13 mostra as possibilidades de comunicação entre as várias tarefas que pertencem a diferentes domínios. É de salientar que se efectuaram testes que demonstram que a comunicação no LXRT é bastante rápida [7].

3.6.1. SERVIÇOS AO NÍVEL DO UTILIZADOR DO LXRT

A comunicação entre o espaço de utilizador e o kernel é possível através de interrupções por software geridas através do LXRT. Utilizando este tipo de interrupção, é possível criar as tarefas ao nível do utilizador, que podem chamar serviços próprios do RTAI da mesma forma como o faz para as chamadas ao sistema[16] Linux. De entre os serviços disponibilizados no espaço de utilizador pelo LXRT, que anteriormente apenas estavam disponíveis como módulos carregáveis, encontram-se os RPCs, *mailboxes*, semáforos e mensagens. Actualmente a API do LXRT é de tal forma transparente que é possível usar as mesmas funções e semântica, quer no espaço de utilizador, quer ao nível do espaço de kernel (*kernel space*) com o mesmo efeito. A única diferença é o uso das funções `main()` em vez das interfaces `init_module()` `clean_up module()` tão características dos módulos de kernel. Esta flexibilidade do LXRT faz com que seja possível testar aplicações de tempo real antes de as escrever como módulos [7].

No entanto convém recordar que as tarefas do espaço de utilizador que usam o mecanismo LXRT para aceder aos serviços do RTAI não são tarefas de *hard real-time*, mas sim de *soft real-time*. Logo, antes usar qualquer serviço do RTAI, as aplicações ao nível do utilizador necessitam criar uma tarefa que irá manter a coerência das estruturas de dados do RTAI enquanto lidam com as comunicações entre tarefas e o escalonador.

3.6.2. LXRT: DO SOFT AO HARD REAL TIME

Como foi referido na subsecção anterior as tarefas do LXRT são do tipo *soft real-time*, no entanto podem-se transformar em tarefas de *hard real-time* como será demonstrado de seguida.

---

[16] Em computação, a chamada ao sistema (*system call*) é um mecanismo usado por uma aplicação para requisitar um serviço do sistema operativo, ou mais especificamente, do núcleo do sistema operativo.

O conceito geral nesta área, era que as tarefas em sistemas híbridos SOUG/SOTR ou pertenciam a um domínio ou a outro, ou seja, ao nível do utilizador ou do kernel, e teriam de ser programadas em função do domínio onde iriam ser executadas. Os serviços disponibilizados no espaço de utilizador pelo LXRT vieram criar uma faixa "cinzenta" neste conceito. O acréscimo de uma rotina capaz de tornar as tarefas normais de um SOUG em tarefas de tempo real de um RTOS faz com que processos normais do Linux sejam transformados em tarefas de *hard real-time* através do uso da chamada `rt_make_hard_real_time()`. Ao contrário dos módulos carregáveis, estas tarefas de tempo real correm no seu próprio espaço de memória isolado, protegendo assim as tarefas de tempo real. Da mesma forma que é possível tornar uma tarefa de *soft* em *hard real-time* também é possível fazer o inverso usando a chamada `rt_make_soft_real_time()` [7].

De seguida, apresenta-se um pequeno exemplo para melhor perceber o funcionamento do LXRT. Não se define nenhuma acção para a tarefa de tempo real, apenas se mostra como as tarefas podem ser criadas a partir do espaço do utilizador. Pois apenas se pretende mostrar as funcionalidades da API RTAI e LXRT [8].

Está esquematizado na Figura 14 o espaço de memória de um sistema operativo *Linux*, em que o espaço kernel e o espaço do utilizador estão separados. Uma tarefa de tempo real que seja criada no espaço de utilizador, irá certamente, criar duas *threads* adicionais, uma *controller thread* de *hard real-time* e uma Linux *server thread* que é executada em *soft real-time*, que é capaz de usar os serviços do Linux. Estas *threads* são capazes de comunicar entre si usando áreas de memória comuns.

No início, a *main thread* reserva memória, usando o modulo de gestão de memória do RTAI (`rt_malloc()`) para os dados que são comuns às diferentes *threads*. O módulo principal chama então o `rt_task_init()` para criar um *real-time agent* no espaço de kernel. Este *real-time agent* é necessário para a execução das tarefas de *hard real-time* no espaço de utilizador e para o uso dos serviços do RTAI. A função, `rt_task_init()`, indica também a prioridade da tarefa, quando esta é escalonada em *hard real-time*. Mesmo que a tarefa não necessite de ser executada em *hard real-time*, a função `rt_task_init()` terá de ser sempre chamada, pois é necessária para ter acesso aos demais serviços do RTAI.

As *threads*, *controller e Linux server*, são criadas usando a implementação de *threads* Posix do RTAI, e são responsáveis pela maior parte do trabalho. Após a criação das *threads* a *main thread* pode ser suspensa até que a *threads*, *controller* e *Linux server* terminem. Quando a *main thread* terminar é chamada a função `rt_task_delete()`, que tem como função limpar os dados internos da tarefa e apagá-la. Por fim, a *main thread* liberta a memória que tinha previamente reservado [8].

A tarefa *Linux server* é responsável por definir a politica de escalonamento FIFO, através da função `sched_set_scheduler()`, no escalonador do Linux, conforme é recomendado para processos temporais críticos de forma a ter prioridade sobre os processos normais do Linux. Tal como na *main thread*, o *soft real-time Linux server* também necessita de executar a função `rt_task_init()`para poder ter acesso aos serviços do Linux. Por fim, antes que comece a esperar por pedidos do *controller thread*, o *Linux server thread* chama a função de sistema `mlockall()` para "prender" o código às páginas de memória de dados do Linux à RAM. Este bloqueio, em adição à política de escalonamento, tem como finalidade evitar atrasos inesperados.

A *thread controller* é a única tarefa com comportamento *hard real-time*. No início também define a política de escalonamento como FIFO no escalonador do Linux, para que seja atendida "a tempo" enquanto estiver a operar *soft real-time*. De seguida o *controller thread* inicializa a informação da tarefa RTAI e cria um *real-time agent* com a função `rt_task_init()`, num processo similar as *threads main* e *Linux server*. Antes de entrar em modo *hard real-time*, é novamente chamada a função de sistema `mlockall()` para "prender" o código e as páginas de memória de dados do Linux à RAM [8].

A *controller thread* entra em modo *hard real-time* chamando a função `rt_make_hard_real_time()`, que faz suspender a *Linux server thread* e activar a tarefa *controller agent* no escalonador RTAI. Na próxima vez que a tarefa for activada já está em modo *hard real-time*, e retorna da função `rt_make_hard_real_time()` para o programa que activou a chamada. É de salientar que neste momento as interrupções do Linux encontram-se desabilitadas, embora o RTAI os possa interceptar. Nesta situação, o *controller thread* não é "preemptado" por nenhum dos processos do Linux.

A primeira função que é chamada após o *controller thread* entrar em modo *hard real-time* é a função `rt_make_periodic()`, que faz com que a tarefa seja escalonada em

intervalos periódicos. Após definir um período de escalonamento, o *controller thread* entra no ciclo principal, e a cada ciclo de repetição, o *controller* espera pelo próximo período que foi definido na função `rt_task_make periodic()` [8].

Quando o ciclo principal do *controller* termina, o *controller thread* invoca a função `soft_real_time()` para suspender o escalonador do RTAI. As interrupções do Linux ficam novamente disponíveis e a *thread* é activada no escalonador do Linux. A *thread* neste momento encontra-se em *soft real-time*. Por fim, o *controller thread* liberta do *real time-agent*, a informação relacionada com o RTAI e também desbloqueia a memória anteriormente bloqueada, para permitir que seja de novo paginada [8].

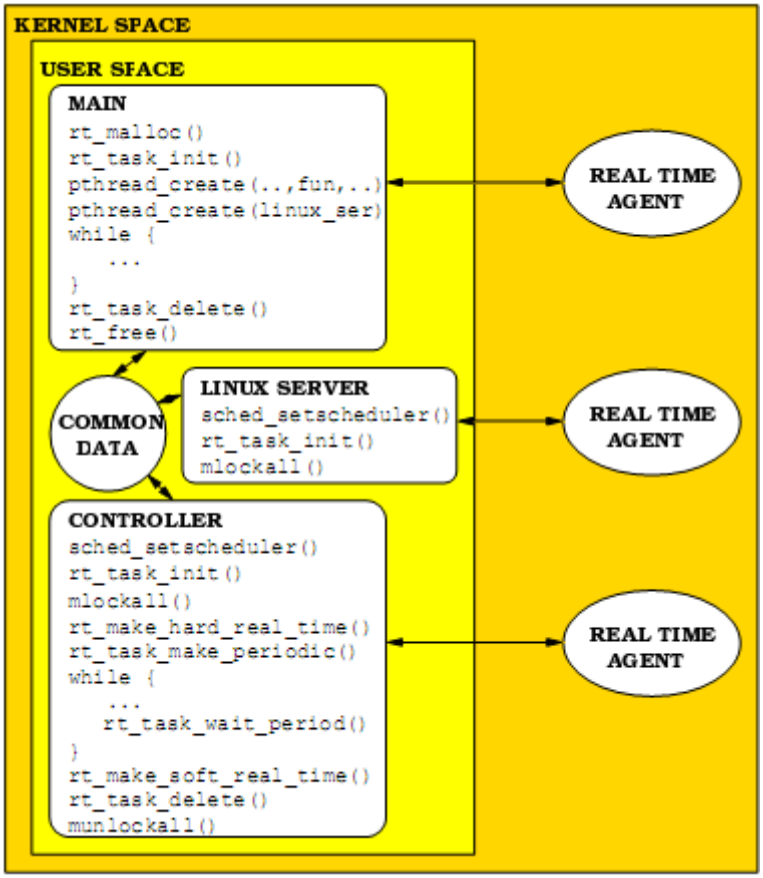

Figura 14 Arquitectura de uma tarefa de tempo real (LXRT) [8]

Convêm referir que as tarefas de tempo real criadas através do LXRT têm latência de escalonamento adicional [18]. Isto deve-se ao facto, de o escalonador de tempo real que opera ao nível espaço kernel, ao ter de reescalonar uma tarefa criada em LXRT, no espaço de utilizador, para uma segunda LXRT do espaço de utilizador. Esta troca entre tarefas

LXRT faz com que seja necessário realizar uma operação de ida e volta, *user*-kernel-*user*, além da gestão de memória que irá ser feita sempre que houver troca de tarefa. A somar à latência provocada pelo reescalonamento, existem ainda latências ao nível das chamadas das APIs. A chamada de uma APIs do RTAI, por exemplo um semáforo, será sempre mais rápido no espaço kernel, do que via LXRT no espaço de utilizador. Isto acontece porque o LXRT necessita fazer uma chamada ao kernel, e esperar que o resultado seja enviado de volta, perdendo assim alguns micro segundos [18].

4. INTERFACE COM A PORTA PARALELA

As portas paralelas são um interface que permite comunicar entre o computador e um periférico, e foi apresentado pela primeira vez por volta de 1970, quando a Centronics mostrou um interface electrónico, e o respectivo protocolo, que permitia a comunicação paralela para as suas impressoras [9]. Como era uma abordagem extremamente simples, muitos fabricantes de computadores adoptaram este interface, de tal forma, que se tornou padrão até 1981. Neste ano a IBM fez uma alteração ao protocolo, alegando que estava a introduzir novas funcionalidades, e criou um novo interface, fazendo com que apenas as suas impressoras (EPSON) funcionassem nos seus computadores. Mais tarde, a IBM criou um meio de manter a compatibilidade entre ambos os interfaces, sem grande resultado de inicio, pois os vários fabricantes de computadores ignoraram o interface da IBM, e continuaram a vender computadores com o interface Centronics. Em 1988, praticamente todos os fabricantes de PC já se tinham rendido ao interface da IBM. Esta mudança deu-se, sem que se tivessem apercebido, que um ano antes, a IBM tinha introduzido no seu interface a possibilidade de transmissão de dados bidireccional, característica que apenas foi utilizada por alguns fornecedores de *hardware* para poderem comunicar entre computadores.

As portas paralelas actuais estão padronizadas desde 1994 sob o padrão IEEE1284, que define cinco modos de funcionamento [10]:

1. Modo de Compatibilidade
2. *Nibble Mode*
3. *Byte Mode*
4. EPP *Mode.*
5. ECP *Mode.*

O objectivo destes modos é possibilitar a criação de novos *devices drivers* que fossem compatíveis entre si e que sirvam de interface de acordo com o padrão da porta paralela. Os modos de Compatibilidade, *Nibble* e *Byte* estão disponíveis em todas as placas de porta paralela originais, enquanto os modos EPP e ECP, só se encontram nas portas paralelas mais recentes pois necessitam de *hardware* adicional para atingirem uma maior velocidade de transmissão de dados, sendo no entanto compatível com os padrões da porta paralela.

O modo de compatibilidade, ou "Centronics", como também é conhecido, só pode transmitir dados num sentido, ou seja, enviar para um periférico. Esta transmissão de dados é efectuada a uma velocidade de 50 kbytes/s, no entanto pode atingir valores na ordem dos 150 kbytes/s.

Para poder usar este interface para receber dados de um periférico, tem que se usar o modo *Nibble* ou o *Byte*. No modo *Nibble* o que se recebe no computador é um *nibble*, ou seja 4 bits, proveniente da porta paralela. Já no modo *Byte* recebe-se no computador, via porta paralela, 8 bits, ou seja, 1 Byte. Para receber este Byte, o sistema de aquisição de dados em tempo real, vai recorrer a uma característica da porta paralela anteriormente descrita, e que só existe em algumas placas: A bidireccionalidade da porta paralela.

Os modos *Extended* e *Enhanced* usam *hardware* adicional para gerir o *handshake*. Para enviar um byte para um periférico no modo de compatibilidade o software tem de fazer o seguinte:

1. Escrever o byte para o porto *Data*

2. Verificar se o periférico está ocupado. Caso esteja não recebe dados, logo os dados serão perdidos.

3. Pôr a linha de *Strobe,* pino 1 da Figura 15, em *low*. Isto indica ao periférico que os dados correctos estão nas linhas de dados, pinos 2-9 da Figura 15.

4. Pôr o *Strobe* em *high* de novo após aguardar cerca de 5µs depois de o *Strobe* ter sido posto em *low*.

Este processo limita a velocidade a que a porta pode operar. O modo EPP e ECP contornam esta limitação permitindo que o *hardware* verifique se o periférico está ou não ocupado e assim gerar o *Strobe* e o respectivo *handshake*. Isto significa que só é necessário executar uma instrução de I/O, aumentando assim a velocidade de transmissão, que para estas portas, podem atingir uma velocidade de transmissão na ordem de 1 a 2 megabytes/s. O ECP usa canais DMA e *buffers* FIFO que é uma vantagem quando se quer deslocar (*"shiftar"*) dados sem que seja necessário usar instruções de I/O [10].

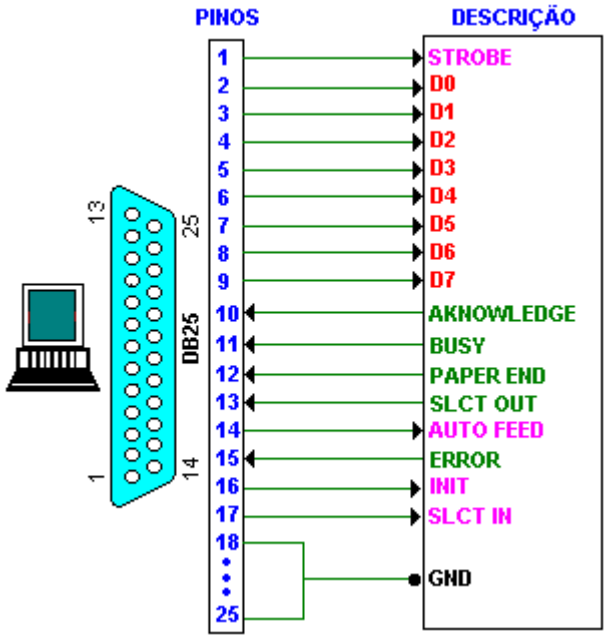

Figura 15 Ficha DB-25 -Pinos e descrição (Vermelho: Registo DATA, Verde: Registo STATUS, Rosa: Registo CONTROL)

4.1. PROPRIEDADES DA PORTA PARALELA

O padrão IEEE 1284 especifica três conectores diferentes para serem usados na porta paralela. O IEEE 1284 Type A que é conector D-Type 25 que normalmente é encontrado na parte de trás dos PCs. O 1284 Type B que é o conector de 36 pinos, também conhecido por conector Centronics, normalmente encontrado nas impressoras. Existe ainda o IEEE1284 Type C que é idêntico ao conector Centronics, mas mais pequeno e com melhor fixação ao periférico, possuindo ainda melhores características eléctricas e é também mais fácil de montar. Este conector disponibiliza mais dois pinos para sinais, que podem ser usados para determinar se o dispositivo conectado tem alimentação [10].

Tabela 2 Numeração do conector D-Type 25 para porta paralela

Nº. Pino (D-Type 25)	Nº. Pino (Centronics)	Sinal SPP	Transmissão In/Out	Registo	Invertido Por *Hardware*
1	1	nStrobe	In/Out	CONTROL	SIM
2	2	Data 0	Out	DATA	
3	3	Data 1	Out	DATA	
4	4	Data 2	Out	DATA	
5	5	Data 3	Out	DATA	
6	6	Data 4	Out	DATA	
7	7	Data 5	Out	DATA	
8	8	Data 6	Out	DATA	
9	9	Data 7	Out	DATA	
10	10	nAck	In	STATUS	
11	11	Busy	In	STATUS	SIM
12	12	Paper-Out / Paper-End	In	STATUS	
13	13	Select	In	STATUS	
14	14	nAuto-Linefeed	In/Out	CONTROL	SIM
15	32	nError / nFault	In/Out	STATUS	
16	31	nInitialize	In/Out	CONTROL	
17	36	nSelect-Printer / nSelect-In	In/Out	CONTROL	SIM
18-25	19-30	Ground	Gnd	-	

Na Tabela 2, os nomes de sinais que são precedidos de "n" estão activos ao nível lógico baixo, por exemplo nError/nFault. No caso de uma impressora, por exemplo, estar a funcionar correctamente, este sinal (nError/nFault) está sempre no nível lógico alto, e se porventura ocorrer um problema, o estado do sinal passa para nível lógico baixo. Na mesma tabela pode-se verificar que existe uma coluna que indica quais os sinais que estão invertidos por *hardware*. Ou seja, se o pino 11 for colocado no nível lógico '1', +5 volts, e o registo status for lido, o Bit 7 do Registo Status estaria a '0'. Esta noção é bastante importante, pois mais à frente vai ser necessário aceder a mais que um registo para se conseguir ler 12 bits da porta paralela. Mais concretamente, irão ser lidos 8 Bits através linhas de dados do Registo de Data e os restantes quatro irão ser lidos através do Registo de Status. Como o registo de Status tem sinais que são invertidos por *hardware*, será necessário inverter novamente, por software, os bits em questão.

As saídas e entradas da porta paralela são normalmente níveis lógicos TTL. Se os níveis de voltagem não são grande problema, já a corrente que pode ser gerada, ou aterrada pode variar. Apesar de os manuais das portas paralelas referirem que estas podem gerar/aterrar correntes a volta de 12mA, estas podem no entanto variar. Se essa variação existir, e for significativa, usa-se um *buffer* de modo a que a porta paralela consuma o mínimo de corrente possível, e evitar danos no *hardware* .

O facto da porta paralela utilizar níveis lógicos TTL faz com que tenha de trabalhar em baixo nível, o que neste caso é vantajoso, pois permite manipular os Bits de acordo com as necessidades, e estabelecer facilmente o protocolo de leitura desejada.

4.2. ENDEREÇOS DOS PORTOS

O porto paralelo tem três endereços base (*base adresses*). O endereço 3BCh foi originalmente usado nas portas paralelas das primeiras placas de vídeo. Este endereço desapareceu posteriormente quando retiraram as portas paralelas das placas de vídeo, para voltar a aparecer como opção para as portas paralelas integradas nas *motherboards*, sendo que a sua configuração pode ser alterada no *Basic Input/Output System* (BIOS).

O *device label* LPT1 está normalmente associado ao endereço base 378h, e o LPT2 é associado ao endereço base 278h. No entanto, no arranque do computador, a BIOS ao

associar o endereço dos portos aos *device labels* LPT1, LPT2 e LPT3 pode atribuir LPT1 ao endereço 3BCh, e o 378h e 278h aos *devices labels* LPT2 e LPT3, respectivamente.

Existem ainda algumas placas de portas paralelas que permitem definir, ao nível do *hardware*, através de *jumpers*, o porto a atribuir a LPT1, LPT2 ou LPT3. No entanto para quem deseja interligar dispositivos com o PC através da porta paralela, estas questões não devem trazer grandes problemas, pois deve-se utilizar o endereço base (*base address*) para ligar ao porto. Por exemplo, em vez de "LPT1" usar 378h [10].

4.3. REGISTOS DE SOFTWARE

As portas paralelas têm três endereços base (*base addresses*). Estes endereços são conhecidos por portos ou registos de dados (*data port*), de estado (*status port*) e de controlo (*control port*). Como o interface usado para ler um dispositivo externo é a porta paralela, perceber a arquitectura destes portos é fundamental, pois vai permitir perceber como inverter o porto de dados para o poder usar como porto de leitura, ou, como detectar uma interrupção para efectuar uma leitura de um periférico, ou mesmo aumentar a capacidade de leitura via porta paralela, de oito para doze bits, usando para esse efeito quatro bits do registo de estado (*status register*).

Os endereços base (*base addresses*) também conhecidos por porto de dados (*data port*) ou de registo de dados (*data register*) estão definidos (*offset*) como BASE+0. As portas paralelas convencionais, inicialmente, serviam só como porto de escrita. No entanto, as portas paralelas actuais são normalmente bidireccionais, o que permite ler 8 bits através deste porto, ver Tabela 3.

O porto de estado (*status port*), ver Tabela 4, é definido como BASE+1. Este porto é apenas de leitura e qualquer tipo de dados que sejam enviados para este porto são ignorados. É constituído por cinco linhas de entrada (pinos 10, 11, 12, 13 e 15) e um registo de estado para o IRQ (*Interrupt Request Line*). Tem o bit 7 (Busy) invertido por *hardware*, ou seja, caso este tenha o valor lógico '0', quer dizer que tem +5 volts no pino 11. Já o bit 2 que é responsável por detectar as interrupções é designado por nACK em que o "n" informa que o bit é activo a zero, ou seja, no bit 2, enquanto não houver uma interrupção, está no estado lógico '1', caso haja uma interrupção, o estado lógico passa a ser '0' [10].

Tabela 3 Porto de dados

Offset	Nome	Leitura/Escrita	Nº. Bit	Propriedades
Base + 0	Porto de dados	Escrita. Nota: Caso a porta seja bidireccional também permite ler	Bit 7	Dado 7 (Pin 9)
			Bit 6	Dado 6 (Pin 8)
			Bit 5	Dado 5 (Pin 7)
			Bit 4	Dado 4 (Pin 6)
			Bit 3	Dado 3 (Pin 5)
			Bit 2	Dado 2 (Pin 4)
			Bit 1	Dado 1 (Pin 3)
			Bit 0	Dado 0 (Pin 2)

Tabela 4 Porto de estado

Offset	Nome	Leitura/Escrita	Nº. Bit	Propriedades
Base + 1	Porto de estado	Leitura	Bit 7	Busy
			Bit 6	Ack
			Bit 5	Paper Out
			Bit 4	Select In
			Bit 3	Error
			Bit 2	nIRQ
			Bit 1	Reservado
			Bit 0	Reservado

O porto de controlo (*control port*), BASE+2, é supostamente apenas de escrita, consultar Tabela 5. Numa situação normal, em que uma impressora é acoplada à porta paralela são usados quarto bits de controlo: O *Strobe*, *Auto Linefeed*, *Initialize* e *Select Printer*. Estes bits são todos invertidos excepto o *Initialize*. No entanto, este porto possui uma característica importante, que é o facto das saídas serem do tipo colector aberto, ou seja, funcionam com um estado lógico baixo (0 volts) e em alta impedância. O estado de alta impedância é necessário para evitar danos no hardware e conflitos, pois se o computador colocar um destes pinos num nível lógico alto (+5 volts), e o dispositivo tiver necessidade de o colocar a um nível baixo (0 volts), pode curto circuitar o porto. No entanto, esta característica é importante caso se pretenda usar estes quatro bits como entradas [10].

Para colocar os pinos num estado lógico alto pode-se usar uma resistência de *pull up*, no mínimo de 4.7kΩ. [10]. Pois caso a porta paralela tenha *pull ups* internos, a resistência exterior iria portar-se como o paralelo da resistência de *pull up* interna, resultando numa resistência de valor inferior. Deste modo, quando um pino do porto de controlo da porta paralela estiver em alta impedância, do ponto de vista do sistema de aquisição de dados, o pino encontra-se no estado lógico alto (+5 volts).

Tabela 5 Porto de controlo

Offset	Nome	Leitura/Escrita	Nº. Bit	Propriedades
Base + 2	Porto de controlo	Escrita/Leitura	Bit 7	Não usado
			Bit 6	Não usado
			Bit 5	Activa porto bidireccional
			Bit 4	Activa o IRQ via a Linha de ack
			Bit 3	Select Printer
			Bit 2	Initialize Printer (Reset)
			Bit 1	Auto Linefeed
			Bit 0	Strobe

Deste modo, os quatro pinos do porto de controlo podem ser utilizados para ler os dados de um dispositivo exterior via porta paralela. No entanto, para poder ler os dados, através da porta paralela, é necessário colocar o porto de controlo em xxxx0100 [10]. O bit 3 é posto no nível lógico '1' porque este bit, Initialize Printer, é activo quando detecta o nível lógico '0', conforme se pode ver na Tabela 2. Como os outros portos são invertidos por *hardware* será necessário coloca-los a "0" para obter o estado lógico alto (+5 volts). Enquanto o porto estiver configurado desta forma, o nosso dispositivo externo pode colocar o pino num estado lógico baixo (0 volts), ligando o pino a massa, e assim mudar o estado do porto.

O registo de controlo tem ainda dois bits importantes, os bits 4 e 5 de controlo interno, que são necessários para detectar as interrupções que vão activar o processo de leitura da porta paralela em tempo real. O bit 4 activa o IRQ e o bit 5 torna o porto bidireccional, ou seja, permite que o porto de dados (Tabela 3), possa ser usado para entrada de dados. Os bits 6 e 7 são reservados e qualquer escrita nestes bits é ignorada.

4.4. CONFIGURAÇÕES DA PORTA PARALELA NA BIOS

Hoje em dia as portas paralelas permitem várias configurações através da BIOS. Os modos mais correntes são:

- *Printer Mode*: Este modo também é conhecido por *default mode* ou *normal mode* é o modo mais básico e apenas permite a transmissão de dados para um periférico, e

mesmo que seja activado o bit 5 do porto de controlo que permite bidireccionalidade, o porto de dados não irá ser capaz de receber dados.

- *Standard and Bi-directional mode (SPP)*: Neste modo, se for activado o bit 5 do do porto de controlo, a direcção do porto é invertida, podendo assim ler dados através das linhas de dados.

- *Extendend Capabilities Port* (EPP) 1.7 *e SPP mode*: Este modo de operação permite o acesso aos registos do modo SPP (Dados, Controlos, Estado) e aos registos do EPP. Neste modo é possível inverter a direcção das linhas de dados usando o bit 5.

- *EPP1.9 e SPP mode*: É idêntico ao anterior, mas usa a versão 1.9 do EPP. Tal como no modo anterior, permite o acesso aos registos do SPP e ao bit 5 do porto de controlo. No entanto nesta versão de EPP tem-se acesso a um bit de *timeout*.

- *ECP Mode*: Este modo permite o acesso ao *Extended Capabilities Port* (ECP), O modo de funcionamento do porto será então definido no *Extended Control Register* (ECR). Os modos de utilização podem ser consultados na Tabela 6. Neste modo o EPP não está disponível no ECR.

- *ECP e EPP1.7 Mode & ECP e EPP1.9 Mode*: permite fazer o mesmo que o modo anterior, no entanto, o modo EPP vai estar disponível no registo ECR.

Tabela 6 Extended Control Register (continua)

Bit	**Função**	
7:5	Selecciona o modo de funcionamento	
	000	*Standard* Mode – Este modo faz com que o porto ECP se comporte com o SPP sem bidireccionalidade
	001	Byte *Mode*/PS2Mode – Funciona como o SPP no modo bidireccional. O bit 5 activa a inversão da porta para se poder ler dados.
	010	*Parallel Port* FIFO *Mode* - Neste modos a DATA FIFO serão enviados para o periférico usando o *handshake* do SPP. O *Hardware* irá gerar o *handshaking* necessário. Útil em dispositivos não ECP, tipo impressoras.

Bit	Função	
7:5	Selecciona o modo de funcionamento	
	011	ECP FIFO *Mode* – Modo padrão do ECP
	100	EPP *Mode*– Activa o modo EPP, caso esteja disponível.
	101	Reserved
	110	FIFO *Test Mode*
	111	*Configuration Mode*
4	ECP *Interrupt Bit*	
3	DMA *Enable* Bit	
2	ECP *Service* Bit	
1	FIFO *Full*	
0	FIFO *Empty*	

5. IMPLEMENTAÇÃO

Neste capítulo será descrita a implementação do módulo de kernel carregável, que permite a aquisição de dados em tempo real via porta paralela, conforme foi proposto em tese. Para implementar o sistema de aquisição de dados em tempo real, foram considerados todos os conhecimentos adquiridos durante o estudo e a pesquisa desenvolvida nos capítulos anteriores. Esse estudo permitiu fundamentar, neste capítulo, as estratégias, as opções e as decisões que levaram ao desenvolvimento do Sistema de Aquisição de Dados em Tempo Real.

Como a implementação e configuração do RTAI no Linux se encontra muito bem documentada nos manuais de utilizador do RTAI [11], e em vários artigos, nomeadamente, o artigo de Cristóvão Sousa [13], optou-se por não dar muita ênfase ao processo de implementação do RTAI. Optando-se antes por fazer alguns comentários ou chamadas de atenção sempre que necessário.

5.1. AMBIENTE DE DESENVOLVIMENTO

O ambiente de desenvolvimento é constituído por computador DELL, conforme se pode ver na Figura 16, equipado com um processador Pentium D 3.40 GHz com 1 Gbyte de

memória RAM e um Disco SATA de 80 Gbytes, e com o respectivo interface para porta paralela, bidireccional.

Figura 16 PC DELL

O equipamento usado para simular as interrupções na porta paralela foi um gerador de sinal HAMEG 10 MHz Function Generator, modelo: HM8030-6, que podemos ver representado na Figura 17.

Figura 17 Gerador de Sinal HAMEG

Figura 18 Analisador lógico

Para poder visualizar e registar os tempos de resposta à interrupção foi disponibilizado o analisador lógico, representado na Figura 18. O INTRONIX LOGIC PORT 34 CHANNEL LOGIC ANALYZER. A informação colectada pelo analisador lógico é disponibilizada pelo software Intronix LogicPort Logic Analyser.

5.2. PREPARAÇÃO DO AMBIENTE DE DESENVOLVIMENTO.

O *patch* do RTAI e respectivas configurações foram implementados na distribuição *Scientific* Linux 5.1 previamente instalada no computador disponibilizado, de acordo com o manual de utilizador do RTAI [11], e um artigo de Cristóvão Sousa, do Departamento de Engenharia Electrotécnica e Computadores, Faculdade de Ciências e Tecnologia da Universidade de Coimbra [13], onde se descreve, em português, a instalação e configuração do RTAI no Linux.

Para simular o ambiente de aquisição, ligou-se a porta paralela do PC a uma placa de ensaio, Figura 19. Nesta placa de ensaio, simulou-se as doze linhas de entrada correspondentes aos doze bits da porta paralela que foram usados para entrada de dados. O estado lógico das entradas era definido conforme o bit, ou bits, estivessem ligados ou não à massa. Os bits colocados no nível lógico ‘1’ acenderiam um LED vermelho, e logicamente, os que estivessem no nível lógico ‘0’ estariam apagados. Os restantes quatro bits serviram para sinalizar os acontecimentos importantes que acorreram durante os testes do sistema de aquisição, ajudando na depuração do sistema. Adiante, será abordado com mais detalhe este assunto, descrevendo como se procedeu à leitura dos bits na porta paralela.

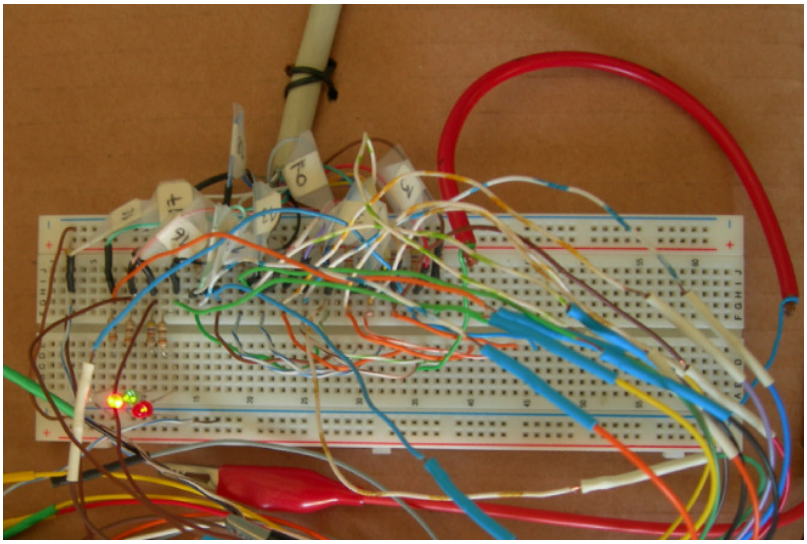

Figura 19 ***Hardware*** **para simular aquisição de dados**

O ambiente de desenvolvimento não estaria completo sem ter um gerador de sinal, HAMEG, para simular interrupções, e um analisador lógico INTRONIX, que permitisse confirmar os dados lidos e os tempos de resposta às interrupções.

Para desenvolver o módulo responsável pela aquisição de dados em tempo real e a aplicação que permite visualizar os valores obtidos, utilizou-se o software de desenvolvimento KDevelop. Esta aplicação não é um compilador, pois usa o *gcc* para criar código executável. No entanto, reúne um conjunto de ligações a vários tipos de ferramentas que agiliza o desenvolvimento de aplicações, bem como na depuração do código desenvolvido.

Para representar graficamente os valores registados pelo analisador lógico foi usada a linguagem de *scripting*/programação *Python* com a sua biblioteca *matplotlib* especifica para a criação de gráficos.

5.3. FORMULAÇÃO DO SISTEMA DE AQUISIÇÃO DE DADOS

Inicialmente foram consideradas duas hipóteses para implementar o Sistema de Aquisição de Dados em Tempo Real: Uma tarefa periódica que permitisse ler a porta paralela periodicamente, ou, uma rotina de interrupção (ISR), que lesse a porta paralela sempre que fosse despoletada uma interrupção.

Após alguns testes optou-se pela ISR. Esta decisão deveu-se ao facto do RTAI tratar o LINUX como a tarefa com mais baixa prioridade do sistema. Ao tratar o Linux como a tarefa com mais baixa prioridade, sempre que a tarefa de tempo real é chamada para ser executada, o RTAI impede o Linux de fazer as operações que estiver a realizar para executar a referida tarefa de tempo real. Este comportamento levantou alguns problemas, porque, caso os períodos de execução entre as tarefas de tempo real seja muito curto, apesar de a tarefa de tempo real ser religiosamente cumprida, o PC fica de tal forma com o processador sobrecarregado que torna difícil trabalhar com outras ferramentas e aplicações, pois são constantemente postas em segundo plano (perde prioridade) para que a tarefa de tempo real se possam realizar. No entanto, ao usar a ISR, a função (*handler*) responsável por executar o código que permite a leitura da porta paralela, só é despoletada caso haja uma interrupção. Ao chamar esta *handler* apenas quando é estritamente necessário, faz com que a sobrecarga do processador seja menor, pois não está constantemente a ler a porta paralela, libertando assim recursos para outras aplicações e ferramentas do sistema.

Para passar os dados obtidos na porta paralela, do espaço kernel para o espaço utilizador optou-se por o fazer através da memória partilhada. Usou-se a memória partilhada porque se pretendia usar uma estrutura de dados que pudesse ser lida e escrita por dois processos diferentes, a tarefa de tempo real e a aplicação a executar no espaço de utilizador responsável por apresentar os dados obtidos pelo sistema desenvolvido. A memória partilhada, tem a vantagem de permitir aceder só a determinadas áreas de estrutura de dados, ou seja, membros individuais da estrutura da memória partilhada podem ser acedidos independentemente para leitura ou escrita. Este método é o mais adequado nesta situação, mesmo tendo em conta que ao contrário das FIFOs, os dados na memória partilhada não são colocados em fila. Logo qualquer dado que seja escrito na memória partilhada sobrepõe o existente. No entanto, a memória partilhada, não corre o risco de ficar cheia e ter de aguardar que os dados escritos sejam lidos para se proceder a uma nova escrita de dados.

Para implementar este recurso tem de se assegurar a consistência de dados, ou seja, no RTAI/Linux, um processo do Linux pode ser interrompido por uma tarefa de tempo real. Como o Linux é a tarefa de mais baixa prioridade o contrário não se verifica. Esta interrupção do Linux por parte da tarefa de tempo real verifica-se em duas situações: quando o tarefa de tempo real executa uma leitura e interrompe o Linux durante uma

processo de escrita provocando uma inconsistência de dados, ou, quando uma tarefa de tempo real executa um escrita e interrompe o Linux durante um processo de leitura provocando, mais uma vez, inconsistência de dados. Isto pode acontecer pois a tarefa de tempo real, neste caso uma ISR, irá ter sempre prioridade sobre a aplicação responsável por apresentar os dados obtidos no espaço de utilizador, pois esta aplicação é um processo do Linux. No entanto, esta questão foi resolvida como será demonstrado mais adiante neste capítulo.

5.4. O MÓDULO DE KERNEL: ESTRUTURA

O conceito e o funcionamento básico dos módulos são fundamentais para se puder criar as tarefas de tempo real. De uma maneira simplista pode-se dizer que um módulo é um conjunto de binários que permite adicionar novas funcionalidades ao kernel.

Na Figura 20, apresenta-se o exemplo de uma estrutura simples de um módulo de kernel onde estão identificadas quarto zonas: livrarias, macros, construtora/desconstrutora e macros de inicialização e saída do módulo.

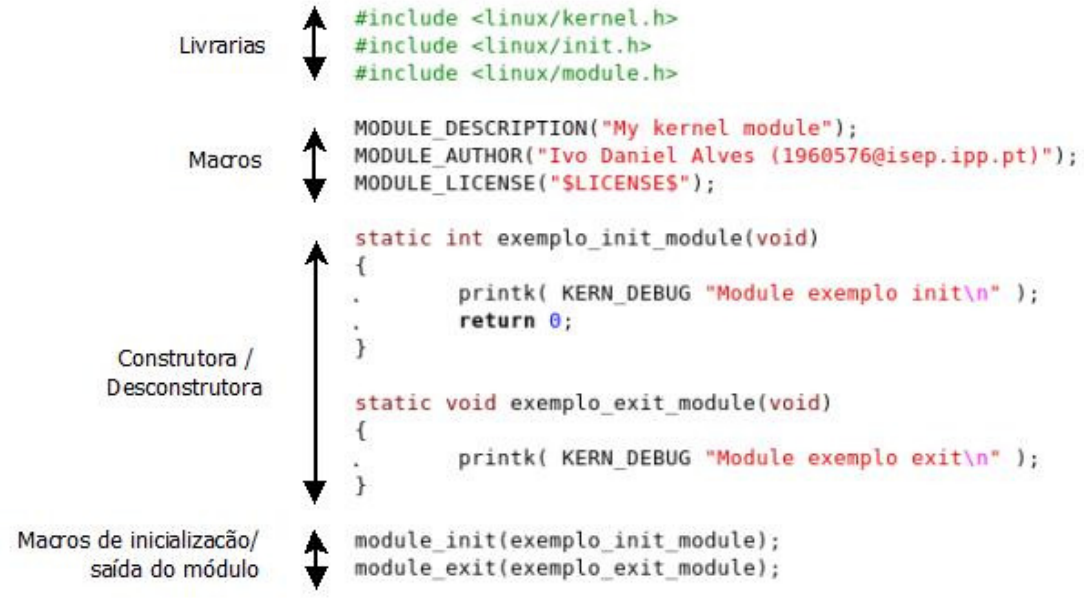

Figura 20 Estrutura do módulo de kernel

As livrarias dizem ao compilador para inserir a informação nelas contida, por exemplo, as livrarias típicas do RTAI que permitem que a utilização das funções de tempo real, FIFOs, memoria partilhada etc. A zona delimitada pelas setas, na Figura 20, que está identificada

como macros permite a identificação do autor, do tipo de licença, a descrição do módulo, e outros tipos de informação sobre o módulo. Não é obrigatório disponibilizar este tipo de informação, no entanto, caso não seja definido o tipo de licença, por exemplo GPL, ao carregar o módulo será enviado um aviso de *tainted kernel*. Desde o kernel 2.6 que é possível definir duas macros no fim do módulo, fazendo com que seja possível dar o nome que mais convier à função construtora e desconstrutora. A função construtora que está associada à macro `module_init` é chamada quando o módulo é carregado no kernel através do comando `insmod` executando as instruções nela contidas. A função desconstrutora associada à macro `module_exit` funciona do modo oposto, ou seja, quando o comando `rmmod` é executado esta função executa as instruções necessárias para parar o que o módulo estiver a executar, e remove o módulo do kernel.

Para inserir ou remover a tarefa de tempo real do kernel, usam-se as instruções do Linux '`ismod rt_lpt_task-driver_rt.ko`' e '`rmmod rt_lpt_task-driver_rt.ko`' respectivamente. No entanto, estes módulos não são os únicos a serem introduzidos no kernel. Também é necessário inserir os módulos que disponibilizam os serviços e funções necessárias para que a tarefa de tempo real seja executada. Neste caso é necessário inserir os módulos '`rtai_hal.ko`' que inicializam o RTAI (o "famoso" RTHAL descrito no Capitulo 2 e **Erro! A origem da referência não foi encontrada.**), '`rtai_sched.ko`' necessário para o escalonador, e o '`rtai_shm.ko`' para se poder usufruir dos serviços de memória partilhada. Estes módulos têm dependências entre eles e têm de ser carregados numa ordem específica. Para que o processo de implementar, inserir e remover os módulos fosse mais transparente, automatizado e menos sujeito a erros foi criado um pequeno *script* para os inserir e outro para os remover do kernel. O *script* de inserção, denominado '`start.sh`' insere desta forma os módulos:

```
#!/bin/sh
TMP=$PWD
cd /usr/realtime/modules/
insmod rtai_hal.ko
insmod rtai_sched.ko
insmod rtai_shm.ko
cd $TMP
insmod rt_lpt_task-driver_rt.ko
```

Como se pode verificar o último modulo a ser inserido é o que corresponde à tarefa de tempo real.

No caso de querer remover a tarefa de tempo real do kernel, usa-se o *script*, '`stop.sh`', em tudo idêntico ao anterior, mas neste caso retiraram-se os módulos pela ordem inversa a que foram inseridos:

```
#!/bin/sh
rmmod rt_lpt_task-driver_rt.ko
rmmod rtai_shm.ko
rmmod rtai_sched.ko
rmmod rtai_hal.ko
```

5.5. CONFIGURAÇÃO DA PORTA PARALELA

Com base no capítulo quatro, para ler a porta paralela é necessário fazer as seguintes configurações:

- Colocar a porta paralela em modo PS2/*Byte Mode*. Este modo é escolhido via BIOS. No entanto optamos por faze-lo directamente no kernel através do módulo/tarefa do RTAI.
- Tornar a porta bidireccional. Através do bit 5 do registo CONTROL.
- Habilitar a interrupção. Através do bit 4 do registo de CONTROL.

Definidas as condições, foi relativamente fácil implementar as instruções necessárias que permitem configurar a porta paralela de modo a torná-la bidireccional, bem como habilitar a interrupção. É apenas necessário declarar, na função construtora do módulo/tarefa de tempo real, as instruções para configurar os registos em questão, conforme se pode ver no extracto de código que se segue.

```
/*Habilitar as interrupções da porta paralela ao
nível do hardware*/

outb(inb(ECR)&0x1f|0x20, ECR); /*Activa o modo
PS2*/

outb(inb(CONT)|0x20, CONT); /*Activa o
bidirecional*/

outb(inb(CONT)|0x10,CONT);/*Activa o IRQ */

/* Fim das configurações para habilitar as
interrupções ao nível do hardware*/
```

5.6. ASSOCIAÇÃO DO IRQ À ISR/*HANDLER*

Na sub-secção anterior descreveu-se como configurar o periférico de modo a que este seja capaz de gerar as interrupções, bem como tornar o registo DATA da porta paralela bidireccional. No entanto, é necessário configurar a PIC (*Programable Interrupt Controller*) para que a máquina atenda a interrupção gerada pela porta paralela.

Para atender e processar algo que foi despoletado por uma interrupção, tem de se associar o pedido de interrupção feito a PIC, a uma ISR (*Interrupt Service Routine*), ou seja, uma função *handler* que neste caso irá realizar a leitura da porta paralela. Essa associação é feita na função construtora do módulo, ver Figura 20, através da função `rt_request_global_irq(IRQ, isr_ler_lpt)`, onde o primeiro parâmetro da função é a que está definida como uma macro, `#define IRQ 7`, que corresponde à interrupção da porta paralela na PIC e o segundo parâmetro é a *handler* que irá ler a porta paralela. De seguida, é necessário dizer à PIC que esta vai ter de honrar as interrupções geradas pela porta paralela. Isto é feito a através das função `rt_enable_irq(IRQ)`.

5.7. CRIAR, ACEDER E ELIMINAR A MEMÓRIA PARTILHADA

Para alocar um segmento de memória partilhada é necessário declarar a função `rtai_kmalloc()` na construtora do módulo/tarefa de tempo real, ver Figura 20. No extracto de código seguinte, é exemplificado a alocação de um segmento de memória que é identificado com a chave 35.

```
/* Macros na header file param.h
#define SHM_KEY 35
#define SHM_HOWMANY 1024
Fim da macro*/
.
.
/* Instruções na função construtora do módulo
struct dados *datashm;

datashm = rtai_kmalloc(SHM_KEY,SHM_HOWMANY *
sizeof(dados));
```

A chave 35 é usada para identificar este segmento de memória, para que outros processos que queiram usar este segmento de memória o possam fazer através do uso desta chave. O tamanho do segmento de memoria a alocar é dado por:

```
SHM_HOWMANY*sizeof(dados);
```

em que SHM_HOWMANY é o tamanho de um vector que irá guardar os dados lidos na porta paralela. Multiplicando o tamanho do vector pelo tamanho do espaço ocupado pela estrutura de dados tem-se o valor final do segmento. Do lado do Linux, o acesso à memória partilhada é feito usando a função 'rtai_malloc()'. Está função não faz alocação de memória, pois a alocação de memória é feita pela tarefa de tempo real. Ou seja, esta função é apenas uma ligação à memória partilhada. É devido a esta particularidade que a alocação de memória tem de ser feita pela tarefa de tempo real antes que o processo de Linux lhe possa aceder.

No espaço de utilizador, a ligação ao segmento de memória previamente alocado pela tarefa de tempo real é declarada da seguinte forma:

```
struct dados *datashm;

datashm = rtai_malloc (SHM_KEY, SHM_HOWMANY *
sizeof(dados));
```

em que a função rtai_malloc () retorna um apontador para a memória partilhada previamente reservada pela função 'rt_kmalloc' da tarefa de tempo real. A função rtai_malloc () usa a mesma chave, SHM_KEY. A chave permite ter vários segmentos de memória partilhados, cada um com a sua ID. Esta característica permite partilhar a informação contida nestes segmentos com vários processos, desde que tenham a chave correcta. Uma vez criado, o apontador para a memória partilhada, pode ser referenciado como outro apontador qualquer, e os resultados ficam imediatamente disponíveis para todos os que partilham a memória.

A memória partilhada é apagada chamando a função 'rtai_kfree()' (que é a versão do RTAI da função do kernel 'free()') usando a mesma chave que foi usada para a criar:

```
rtai_kfree(SHM_KEY);
```

Do lado do Linux, chama-se a função 'rtai_free()'na qual se indica a chave e o ponteiro:

```
rtai_free(SHM_KEY,datashm);
```

O método para libertar a memória partilhada é o inverso da sua alocação. Ou seja, tem de ser libertada primeiro no processo de Linux e só depois na tarefa de tempo real.

5.8. IMPLEMENTAR O SERVIÇO DE ROTINA DE INTERRUPÇÃO

A ISR (*Interrupt service Routine*) é uma função sem argumento e não retorna nenhum valor, e gere a interacção do dispositivo de *hardware* com a aplicação de tempo real. A ISR vai ser invocada quando a interrupção for gerada e irá ler a porta paralela, e de seguida, envia os respectivos dados para a aplicação que corre no espaço de utilizador onde se poderá observar os dados adquiridos através da porta paralela.

A porta paralela permite, com as configurações descritas no ponto anterior, ler facilmente os 8 bits do registo DATA da porta paralela. No entanto, com um pouco de trabalho pode-se ler mais do que os 8 bits disponíveis no registo DATA. Nos outros dois registos, quer o CONTROL, quer o STATUS, é possível ler os respectivos bits, no entanto deve-se ter cuidado em não alterar o estado de determinados bits que são responsáveis pelo modo de funcionamento da porta paralela. Por exemplo, o bit que habilita a interrupções no registo de STATUS, ou o bit 5 da porta de controlo que torna a porta paralela bidireccional. Após consultar a arquitectura dos registos conclui-se que os registo que convinha ler para aumentar o tamanho da palavra de oito para doze bits, seria o STATUS, por este ser um registo apenas de leitura, onde qualquer tentativa de escrita é ignorada, logo não podia ser utilizado para enviar informação para o exterior. Tendo em conta as limitações do registo de STATUS no que respeita à escrita de dados, e tendo a intenção de ter alguma informação visual sobre o que se passa no sistema de aquisição de dados, foi usado para esse efeito o registo CONTROL, ver Figura 19 (led aceso), nomeadamente os bits ‘0’,‘1’,‘2’ e ‘3’ deste registo cujo sinal SPP corresponde ao Strobe, Auto LineFeed, Initialize Printer e Reset. Na Figura 21, mostra-se como se “rearranjaram” os registos para que os sistema de aquisição de dados em tempo real fosse capaz de ler uma palavra de 12 bits.

Como se decidiu usar interrupções para detectar a presença de informação a ler pelo sistema de aquisição de dados, o bit 6 do registo de STATUS vai ficar reservado para detectar as interrupções externas via porta paralela. Assim sendo, sobram os bits 7 do registo de STATUS, o *busy*, que é invertido por *hardware*, ou seja, o valor lido é contrário ao estado em que se encontra. Já o bit 4 e 6 têm a característica de estarem activos ao nível lógico baixo. Isto quer dizer que o bit 6, por exemplo, que corresponde ao sinal SPP nACK, ligado ao pino 10 da ficha DB25 (ver Figura 15) responsável por detectar interrupções, só está activo ao estado lógico baixo. Quando está a nível lógico 1 não existe

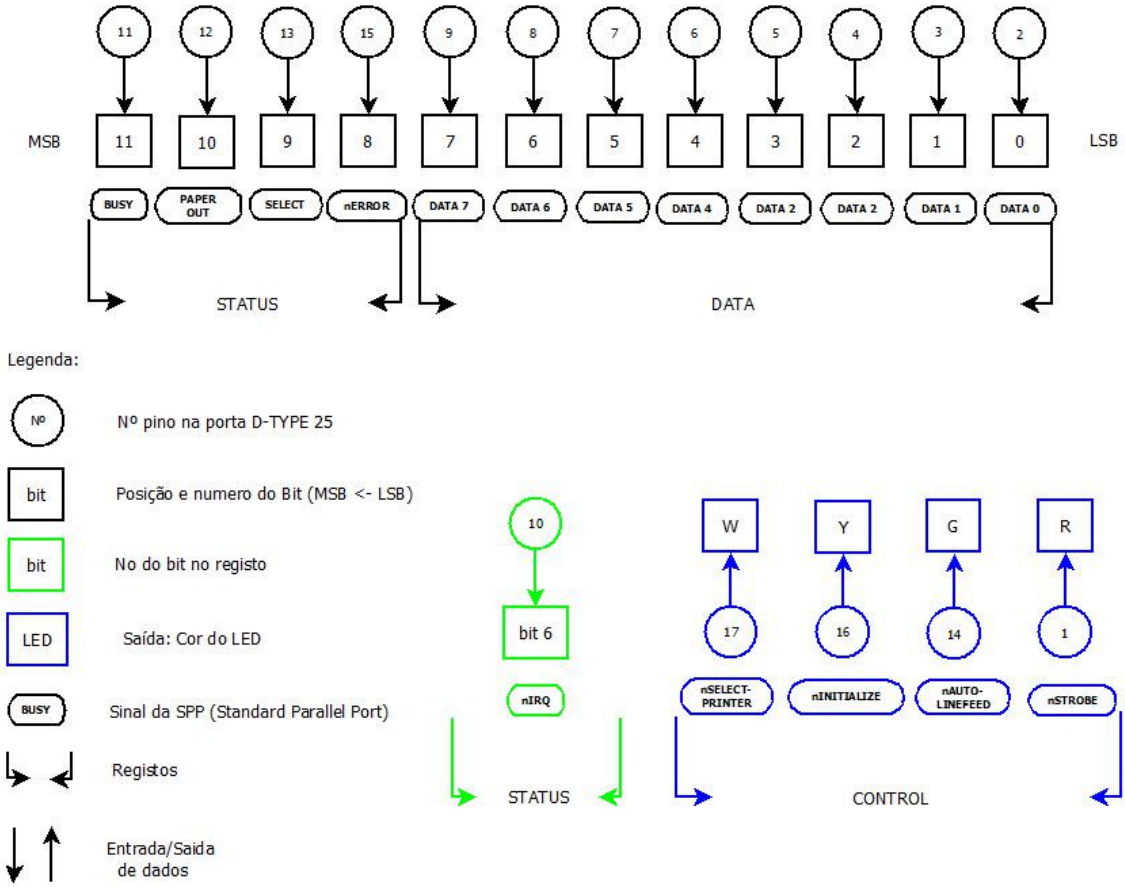

Figura 21 Protocolo de Leitura da Porta Paralela.

interrupção, quando tem nível lógico 0 detecta a interrupção. Isto faz com que haja a necessidade de entrar com estas condições quando se proceder a leitura do registo STATUS, bem como trocar (*"shiftar"*) a sua posição do bit 7 com o bit 6 para obter as posições dos bits conforme a Figura 21. Tendo em conta todas estas condições foi implementado um algoritmo conforme é mostrado pela Figura 22.

Após criar a ISR (*Interrupt* Service *Routine*) era necessário colocar na memória partilhada os dados obtidos, para serem apresentados numa aplicação a executar no espaço de utilizador. Conforme já foi referido, ao idealizar o sistema de aquisição de dados, há que garantir a consistência de dados, e impedir que a tarefa de tempo real e os processos de Linux nas operações de escrita e leitura não causem inconsistência de dados. No entanto surge também outro problema, caso a ISR tenha um tempo de execução superior ao tempo a que são geradas a interrupções, pode acontecer que uma interrupção seja despoletada antes de a ISR terminar, o que faz com que a ISR fosse interrompida e chamada de novo

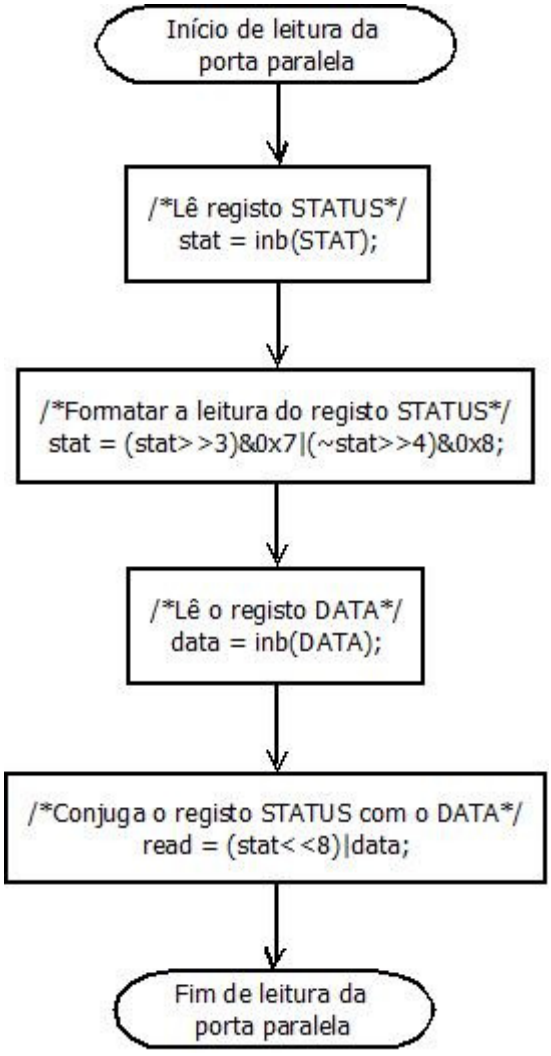

Figura 22 Rotina de leitura da porta paralela

para atender essa nova interrupção. Ou seja, a rotina ISR é reentrante. Este conceito é similar à recursão, em que uma função se pode chamar a si mesmo.

A reentrância torna o processo de leitura da porta paralela em tempo real ainda mais crítico, visto que estas situações devem ser evitadas quando se pretende escrever para recursos que não possam ser escritos atomicamente, como por exemplo: estruturas globais e registos de hardware, que é o que se pretende fazer.

Para ultrapassar o problema da ISR reentrante, é necessário desactivar as interrupções o mais cedo possível, isto é logo após a *handler* ser chamada. No entanto, é necessário assegurar que a bidireccionalidade da porta paralela não é desconfigurada. Por fim, antes de sair da *handler*, as interrupções têm de ser novamente activadas, como é exemplificado de seguida:

```
static void isr_ler_lpt (void){

/* Para não ocorrer reentrância na função de ISR
devemos desactivar as interrupção quando entramos
na função e restaurar a interrupção quando antes
de sair*/


outb(inb(CONT) & 0xEF, CONT); /* desactivar IRQ
/* código para ler as porta paralela deverá ser
escrito entre estas duas expressões/

outb(inb(CONT) & 0xEF, CONT); /* activar IRQ
}
```

Caso fosse necessário, era possível adicionar funções que desactivassem o escalonador e desta forma evitar que uma possível tarefa de maior prioridade interrompesse a interrupção. Usando como exemplo o excerto de código anterior, a desactivação do escalonador seria feita com a função `rt_sched_lock()` logo após a desactivação do IRQ, e seria activado imediatamente antes da função que activa o IRQ com a função `rt_sched_unlock()`.

5.9. PARTILHAR OS DADOS COM O ESPAÇO DE UTILIZADOR DO LINUX

Como já foi mencionado varias vezes, as tarefas de RTAI, correm como módulos do kernel. Para que se possa visualizar no monitor do PC os dados adquiridos via porta paralela, foi criada uma aplicação, no espaço de utilizador, capaz de ler a informação colectada pela tarefa de tempo real.

Conforme descrito no capítulo 3, existem vários mecanismos para comunicar entre tarefas que correm ao nível do kernel e os processos ao nível de utilizador. Inicialmente houve uma certa indecisão do mecanismo a utilizar: FIFOs ou memória partilhada. Mas como existem algumas inconveniências no uso das FIFOs optou-se pela memória partilhada.

A utilização de memória partilhada também requer alguns cuidados, como por exemplo, garantir exclusão mútua, e assim evitar que um processo do Linux leia ou escreva na memória partilhada quando a tarefa de tempo real estiver a fazer uma operação, de leitura ou escrita, na memória partilhada. Outro problema é garantir a integridade dos dados, pois na memória partilhada, existe a possibilidade de os dados mais recentes poderem ser escritos em cima de dados que já se encontram na memória partilhada. Ou seja, escrever

numa variável que está ser partilhada entre dois ou mais processos, em que o valor actual corresponde sempre ao da última escrita.

Na memória partilhada não é necessário abrir um canal para escrita e outro para leitura. Apenas é necessário ler ou escrever para a zona partilhada, evitando as preocupações relativas aos limites da mensagem como acontece nas FIFOs. Por exemplo, no caso de escritas sucessivas seria necessário inserir um identificador para saber onde começa e termina a mensagem. Ao ler a mensagem seria necessário reconstrui-la tendo em conta os identificadores. Outro problema é o facto de as FIFOs poderem ficar cheias, e não permitir que novos dados sejam enviados da tarefa para aplicação ao nível do utilizador responsável por apresentar os valores.

Para que os dados pudessem estar disponíveis para ambas as aplicações, tarefa de tempo real e aplicação de visualização ao nível do utilizador, foi idealizado um vector circular que vai armazenando os valores lidos pela tarefa de tempo real. Este vector é disponibilizado via memória partilhada, e logo que a aplicação que corre no espaço utilizador é executada, os dados colectados pela tarefa de tempo real, são apresentados no monitor do PC para que o utilizador os possa ver.

O maior problema ao optar por esta solução era o facto de se escrever no vector circular através de uma tarefa de tempo real que corre ao nível do kernel, e ler através de uma aplicação que corre no espaço de utilizador. Esta situação fez com que houvesse a necessidade de criar uma *flag* que indicasse se determinada posição no vector tinha sido lida ou escrita, e desta forma evitar escrever em cima de dados que não tinham sido lidos, ou mesmo, ler posições onde não existe informação, ou mesmo "lixo". Assim sendo, o vector é composto por uma estrutura com dois campos importantes. O campo `data`, responsável por guardar os dados lidos via porta paralela, e o campo `nova_leitura`, que caso tenha o valor "0" sinaliza se determinada posição foi lida e aguarda escrita, e se estiver a "1" foi escrita e aguarda leitura. Conforme se pode ver no estrato de código seguinte.

```
typedef struct
{

   unsigned int data;    /* vamos escrever para
   este vector a leitura dos 12bits*/
   unsigned int nova_leitura; /* "0" lido, "1"
   escrito*/
} dados;
```

O vector propriamente dito é declarado no inicio do modulo, após a declaração das livrarias, das *heather files* e descrição do módulo, conforme se pode ver a seguir.

```
dados DATA_SRT[SHM_HOWMANY];
```

O número de posições do vector é dado por SHM_HOWMANY, e pode ser definido na *heather file* `param.h`.

Deste modo, ficou relativamente fácil escrever os dados lidos via porta paralela no vector, sendo apenas necessário garantir que quando este chegue à ultima posição volte automaticamente ao inicio do mesmo. É do conhecimento geral que este processo de voltar ao inicio do vector é relativamente fácil de implementar. No entanto, convêm lembrar que a *handler* só é chamada quando existe uma interrupção. Logo, antes que a tarefa de tempo real teste a posição do vector circular, onde vai escrever, para ver se está livre, tem que ser incrementado um contador, que é responsável pela posição onde a tarefa de tempo real vai escrever no vector. Este contador tem de ser declarado como uma variável global iniciada a zero, ou como *static*, pois caso contrário irá ser iniciada a zero sempre que a função seja chamada, impedindo o funcionamento pretendido para o vector circular. O conjunto das seguintes expressões mostra como são preenchidas as posições do vector circular a cada interrupção.

```
static unsigned int t=0; /* declarado fora da
função handler*/
static void isr_ler_lpt (void){ /*função
handler*/
.
.
t++; /* incrementar vector    circular*/

  if(datashm[n].nova_leitura = = 0) /* pode-se
escrever?*/
  {
   n=t%SHM_HOWMANY; //incrementa o vector
   datashm[n].data = read; /*coloca os dados na
estrutura*/

   datashm[n].nova_leitura = 1; /* flag leitura
dos dados*/


   outb((0x02^0x0b)|inb(CONT), CONT); //Led verde
“on” - escrita OK
  }
  else{
  outb((0x04^0x0b)|inb(CONT), CONT); // led
vermelho amarelo “on” - Escrita não OK
  }
```

Como se pode verificar neste extracto de código os dados lidos são colocados na memória partilhada através de uma estrutura. Posteriormente a memória partilhada é acedida por um processo de leitura que corre em Linux ao nível do utilizador e disponibiliza os dados obtidos na porta paralela.

Como não é possível usar os semáforos entre tarefas de tempo real e processos do Linux, é necessário aplicar algum tipo de técnica que permita garantir a exclusão mútua e a integridade dos dados. Existem várias técnicas para implementar a exclusão mútua e garantir a integridade, como a *head to tail* ou usando *flags* de leitura e escrita entre outras. Neste caso, para garantir a exclusão mútua e integridade de dados entre o processo de escrita e o processo de leitura, ou seja, entre a tarefa de tempo real que corre ao nível do kernel e a aplicação de Linux para visualizar as leituras da porta paralela, decidiu-se usar a *flag* de leitura de dados para impedir que o vector circular fosse lido pela aplicação Linux sem que a tarefa de tempo real lá tivesse colocado qualquer dado. Isto consegue-se definindo que a tarefa de tempo real vai ser o escritor na memória partilhada e que o processo/aplicação Linux irá ser o leitor.

Definindo estes pressupostos, e inicializando o vector circular a zero sempre que a tarefa de tempo real é carregada no kernel, e com o uso da *flag* `datashm[n].nova_leitura` garantiu-se a exclusão mútua e a integridade dos dados. Na tarefa de tempo real esta *flag* é testada antes da escrita na memória partilhada para verificar se o vector circular está cheio, e não reescrever sobre dados que ainda não foram lidos. Caso seja possível escrever no vector circular, o estado da *flag* é actualizado de forma a permitir que o processo/aplicação Linux o possa ler. Quando o processo de Linux o for ler também vai testar se existe algum valor a ser lido nessa posição, e em caso afirmativo lê-o, e actualiza a *flag* para que se possa escrever nessa posição novamente. Os processos de leitura e escrita estão esquematizados na Figura 23.

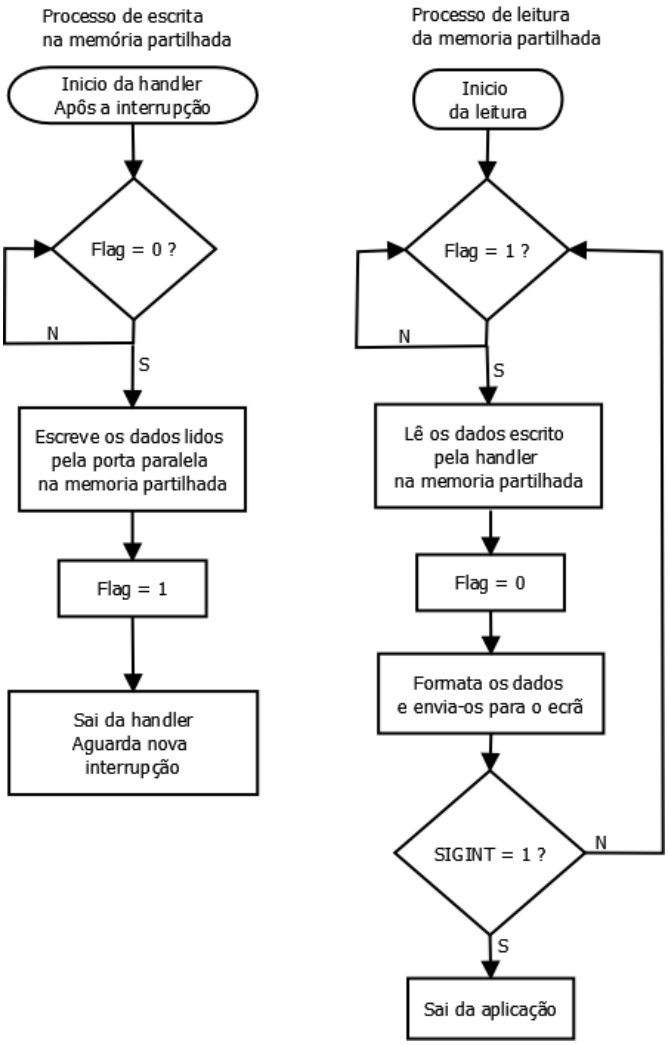

Figura 23 Rotina de escrita/leitura na memória partilhada

5.10. INTERFACE COM UTILIZADOR

O interface com utilizador é uma aplicação que corre na consola do Linux e que tem como objectivo apresentar os dados obtidos via porta paralela. Conforme já foi referido em capítulos anteriores os dados obtidos através da porta paralela são disponibilizados via memória partilhada para que outras aplicações lhe possam aceder. Assim sendo, a aplicação terá de aceder ao pedaço de memória que é criado pelo módulo de kernel/tarefa de tempo real quando este é carregado no kernel. Este acesso é feito usando a função '`rtai_malloc()`' como se pode ver no estrato de código seguinte:

```
static dados *datashm;

datashm = rtai_malloc(SHM_KEY,
SHM_HOWMANY*sizeof(dados));
```

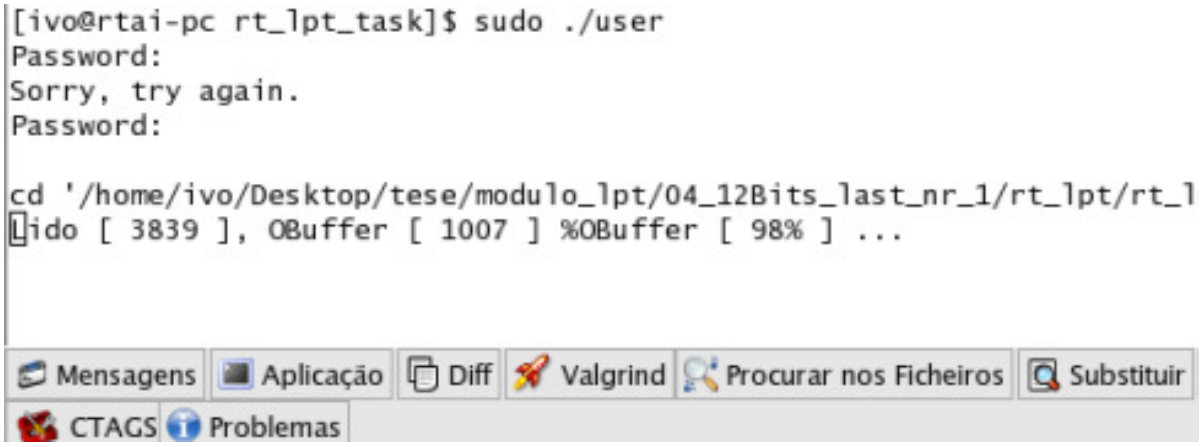

Figura 24 Aplicação interface com utilizador

Desta forma é retornado para o apontador a memoria partilhada reservada previamente pela tarefa de tempo real através da função '`rt_kmalloc()`', ficando os dados imediatamente disponíveis para aplicação que corre em Linux.

A aplicação que corre em Linux disponibiliza os seguintes dados: Os dados lidos pela porta, a posição no buffer circular e a percentagem de ocupação do buffer.

A Figura 24, mostra a aplicação "`user`" a ser executada, e a mostrar os dados lidos pela tarefa de tempo real via porta paralela.

5.11. PLACA DE ENSAIO PARA SIMULAR A AQUISIÇÃO DE DADOS

Para se simular a aquisição de dados em tempo real criou-se uma placa de desenvolvimento como se pode ver na Figura 19, onde se usou um antigo cabo de impressora de porta paralela ao qual foi cortada a ponta que liga à impressora. De seguida identificaram-se todos os condutores através da sua condutividade com um multímetro.

Foram também usadas resistências, de 390Ω nas linhas de saída e 4,7kΩ nas linhas de entrada, ver Figura 19, de forma a evitar consumos de corrente excessivos que pudessem danificar a *motherboard*, e que permitissem a utilização dos portos de entrada e saída de dados do computador, conforme é descrito no quarto capítulo.

As linhas da porta paralela foram depois ligadas a um analisador lógico de forma a obter o tempo entre o instante em que se dá a interrupção e o inicio da leitura dos dados da porta paralela, ou seja, a diferença entre o evento e inicio da nossa tarefa de tempo real, obtendo

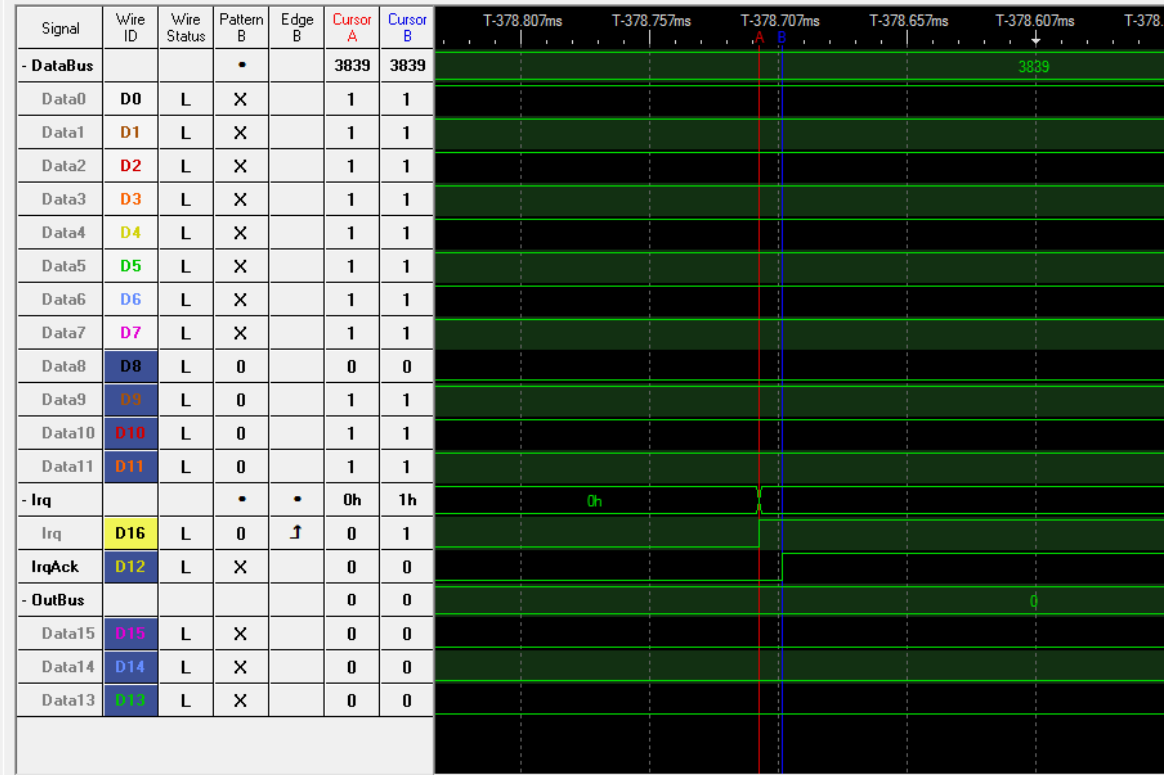

Figura 25 Analisador lógico: tempo de resposta.

As linhas da porta paralela foram depois ligadas a um analisador lógico de forma a obter o tempo entre o instante em que se dá a interrupção e o inicio da leitura dos dados da porta paralela, ou seja, a diferença entre o evento e inicio da nossa tarefa de tempo real, obtendo assim o tempo de resposta, que é a diferença entre a linha vermelha (A) e a azul (B) conforme podemos ver na Figura 25.

5.12. O AMBIENTE DE TESTE

Para comparar o comportamento da tarefa de tempo real desenvolvida, foi criada uma pequena tarefa de *soft real-time* sem usar funções do RTAI. Para simular a tarefa de *soft real-time* usou-se a função '`request_irq()`' conforme se mostra de seguida:

```
ret = request_irq(7, handler, SA_INTERRUPT,
"parallelport", NULL);
```

Esta função é muito semelhante à usada na tarefa de tempo real:

```
ret = rt_request_global_irq(IRQ, isr_ler_lpt)
```

Como se pode ver, as funções são idênticas e a sua função similar. Ambas ligam a linha de interrupção da porta paralela (IRQ 7) a rotina de serviço de interrupção (ISR) definidas nos dois primeiros argumentos da função.

No entanto a função '`request_irq()`' contem mais argumentos. O quarto e quinto argumento da função não são muito importantes pois apenas se referem ao nome do dispositivo reivindicado e a uma cookie que é passada para a *handler*. Já o terceiro argumento, a *flag* SA_INTERRUPT, é importante, pois faz com que após atendida a interrupção, esta não possa ser interrompida enquanto o processamento da mesma não acabar, porque desactiva as interrupções locais.

Estando as duas tarefas, a de *hard* e a de *soft real-time* prontas, faltava algo para "stressar" a máquina, de modo a averiguar o seu desempenho. Para isso, foram executadas no Linux as seguintes instruções:

```
a=0; while "true"; do dd if=/dev/hda1
of=/dev/null bs=1M count=1000; a=`expr $a + 1`;
date; echo $a; done
```

Este conjunto de instruções cria um ciclo infinito que copia um bloco de dados do disco rígido para o *device null*, além disso informa o numero de instruções realizadas, data e hora. Com estas instruções conseguiu-se stressar a máquina, como se pode verificar nas imagens seguintes, Figura 26 e Figura 27. Para avaliar se o stress provocado na máquina pelas instruções referidas era significativo, foi usado o comando '`top`' do Linux, que permite visualizar a actividade do processador em tempo real, bem como outros parâmetros, tais como a carga média (*load average*), número de tarefas, memória, processos (a correr, a dormir, parados ou zombie), etc.

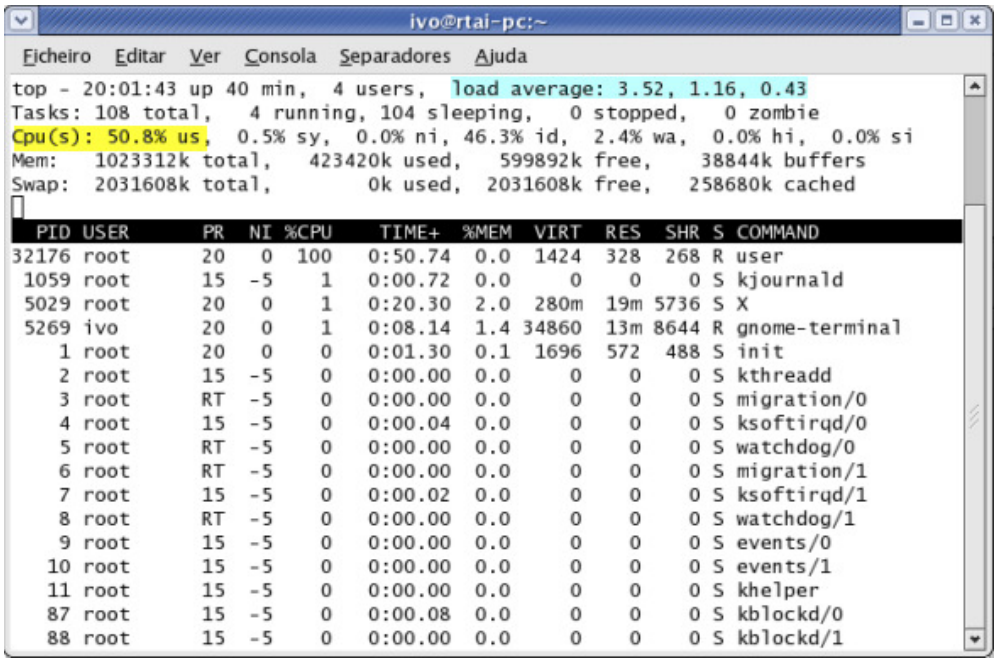

Figura 26 Informação estado do CPU (amarelo) e carga média (azul) sem carga

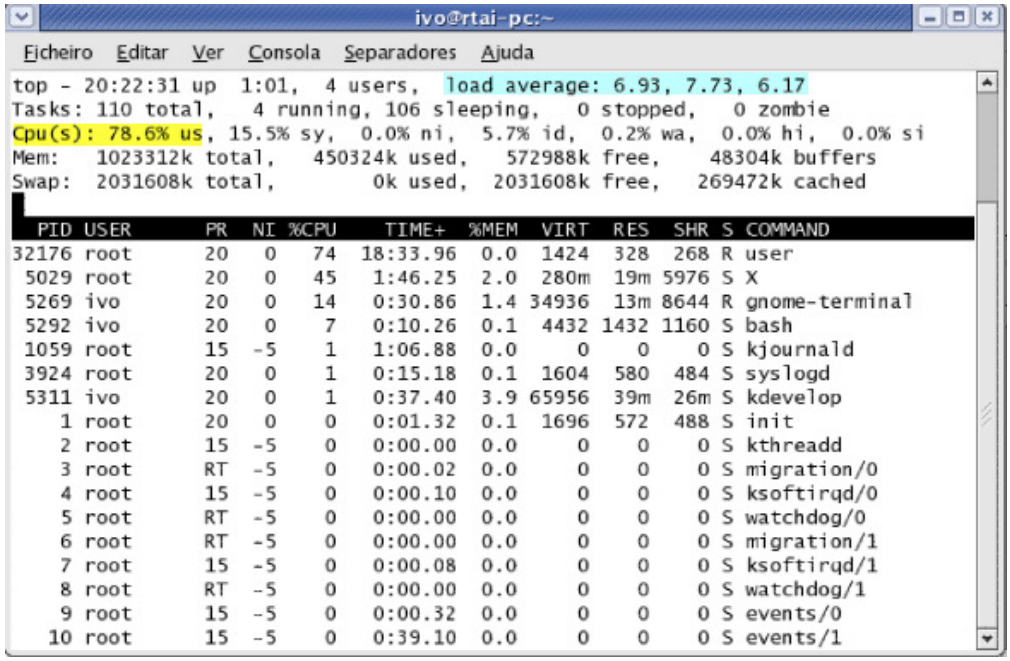

Figura 27 Informação estado do CPU (amarelo) e carga média (azul) com carga

Nos exemplos da Figura 26 e da Figura 27 pode-se verificar que há um aumento de cerca de vinte por cento na utilização de CPU quando se submete a tarefa de *hard real time* a interrupções com frequência de 5Hz.

Após ter tudo definido e pronto a testar, com a ajuda do analisador lógico registaram-se os tempos de resposta às interrupções das tarefas de *hard real-time* e *soft real-time*, com e sem as instruções que provocam aumento de "stress" (carga) na máquina. Foi inicialmente estabelecido a recolha de dados para as seguintes frequências: 5Hz, 50Hz, 500Hz, 5kHz, 50kHz, 100kHz, 150kHz. No entanto, só foram colectados valores até 50kHz, pois a esta frequência a tarefa *soft real-time* deixa de responder às interrupções falhando leituras, como se mostra mais adiante. Outro problema que surgiu, foi quando a tarefa de *hard real-time* ficou sujeita a interrupções geradas a frequências de 150kHz. A esta frequência o PC bloqueava, pois a elevada frequência fazia com que o Linux fosse interrompido para que as interrupções geradas, que são de tempo real, fossem atendidas (maior prioridade). A elevada frequência das interrupções faz com que o Linux e todos os seus processos sejam ignorados pois têm menor prioridade, e fiquem sem a sua fatia de tempo de processamento, causando o *crash* da máquina. Esta subjugação do Linux por parte da tarefa de tempo real é verificável a 100kHz, pois a aplicação que corre em Linux, e que permite visualizar os dados obtidos pela porta paralela, só é actualizada quando a tarefa de tempo real não está a atender nenhuma interrupção, fornecendo o tempo de CPU necessário para o

processamento dos processos do Linux. Ou seja, quando a tarefa de tempo real não está a atender interrupções permite que outras aplicações possam correr em Linux.

Esta comutação com frequências superiores a 100kHz faz com o refrescamento da imagem disponibilizada no monitor seja menor, parecendo mais um movimento “arrastado” frame a frame, sendo que para quem está a tentar observar os dados parece que os dados só estão a ser lidos entre frames. Pela mesma razão, para valores de frequência na ordem dos 150kHz a taxa de refrescamento do monitor é muito menor e a imagem fica bloqueada num frame e não se conseguem ver os dados que vão sendo colectados via porta paralela a serem actualizados no ecrã. Assim sendo, se não é possível ver os dados colectados não faz sentido tentar colectar dados para estas frequências. No entanto se for reduzida a frequência vê-se que, pelo aumento de dados no buffer, que os dados foram registados. Para valores acima destes a máquina bloqueia e não recupera. Devido a estes factos, foi decidido ter em conta só os dados obtidos até a frequência de 50kHz.

6. RESULTADOS

Neste capítulo são apresentados e analisados os resultados dos testes a que o sistema desenvolvido foi submetido. Foram recolhidas quarenta amostras para o sistema de *hard real-time* com e sem carga e também para o sistema de *soft real-time*. Com estes dados foram criados gráficos para possibilitar uma clara percepção e análise de resultados.

6.1. ROTINA DE LEITURA

Nas duas figuras seguintes são apresentados os resultados das leituras de dados quando o sistema desenvolvido foi testado a uma frequência de 50kHz.

Figura 28 Consola do Kdevelop a executar a aplicação de leitura de dados

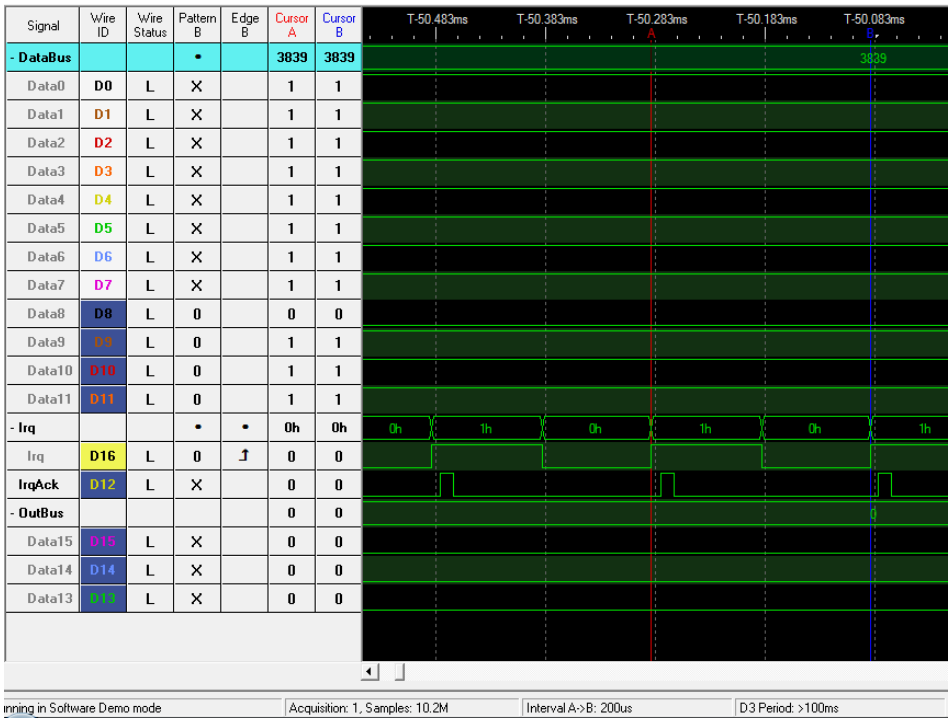

Figura 29 Aplicação gráfica do analisador lógico

Comparado o valor obtido através da rotina de leitura, na Figura 28, sublinhado a azul, com o valor obtido através do analisador lógico, Figura 29 também a azul, é claro que o valor obtido é o mesmo. Assim confirma-se que a aplicação de leitura de dados, no espaço de utilizador, está a apresentar correctamente os dados obtidos pela tarefa de *hard real-time* que corre ao nível do kernel e é responsável pela aquisição de dados via porta paralela.

Tabela 7 Tempos de resposta à interrupção em *hard real-time*

	Tempos de resposta à interrupção da Porta Paralela									
	Frequência/Período									
	5Hz/200ms		50Hz/20ms		500Hz/2ms		5kHz/200us		50kHz/20us	
	Tipo de dados									
Num. Amostras	**HRT s/c**	**HRT c/c**	**HRT s/c**	**HRT c/c**	**HRT s/c**	**HRT c/c**	**HRT s/c**	**HRT c/c**	**HRT s/c**	**HRT c/c**
1	12,9	13,4	12,7	12,4	13	7,5	8,7	8,7	8,6	13,3
2	13,9	9	12,9	8,3	13,7	7,1	8,8	6,7	7,2	13,8
3	11,6	15	13	11,9	7	7,9	12,1	7,2	7,6	12
4	12,8	9,1	13,6	8	12,2	7,8	14,6	7,9	8,4	12,7

Continua na página seguinte

Continuação da tabela da página anterior

	Tempos de resposta à interrupção da Porta Paralela									
	Frequência/Período									
	5Hz/200ms		50Hz/20ms		500Hz/2ms		5kHz/200us		50kHz/20us	
	Tipo de dados									
Num. Amostras	**HRT s/c**	**HRT c/c**	**HRT s/c**	**HRT c/c**	**HRT s/c**	**HRT c/c**	**HRT s/c**	**HRT c/c**	**HRT s/c**	**HRT c/c**
5	14,2	10	12,1	9,8	13,9	8,5	13,7	8,7	8,8	13,1
6	12	9,6	12,8	9,8	6,7	8,4	7,4	7,8	7,2	13,7
7	11,6	9,9	13,2	10	13,4	8	12,6	8,4	7,6	12,2
8	13,3	10,1	13,2	8,3	6,9	8,2	9,1	9,1	8,4	12,7
9	13,4	9,8	13,1	9,7	12	9,1	7,3	7,8	6,8	13,2
10	12,5	8,4	13,3	9,3	13,5	9,6	12,6	7,6	7,2	11,5
11	12,9	9,6	13	9,2	13,5	9,1	9	8,6	7,8	12,2
12	12,5	8,3	12,4	8,9	13,4	6,9	8,1	7,7	8,6	12,3
13	12	9,3	12,4	9,2	13,1	7,1	12,74	8,1	7	13,3
14	13,2	9,6	12,8	8,6	13,3	7,5	8,74	8,5	7,8	11,7
15	13,3	8,9	11,9	8,3	13	8,2	6,9	6,9	6,4	12,2
16	13,4	8,5	12	8,5	8,9	7	12,6	7,7	7,2	12,7
17	11,8	9,9	11,9	7,8	8,7	8,1	13,1	8,4	8,6	13,3
18	13	9,5	12,7	8,4	13	7,8	12,9	7,1	8,8	11,6
19	14,2	13,2	13,5	8,1	12,9	8,4	7,4	7,9	7,4	12,2
20	12,5	8,1	13,7	8	15,4	8,1	12,3	8,1	8,2	13,3
21	14.2	8.1	13.3	9,5	13.2	8,2	7.7	7,5	7.6	12.6
22	12.6	13.5	12.8	9	12.1	8,6	13.5	8,1	5.8	13.2
23	14	7.8	12.4	8,6	11.8	9,7	12.1	8,8	6.8	11.6
24	13.6	9.2	13.8	8,9	7.2	7	7.9	7,6	7.2	12.2
25	12.5	9.5	12.9	12,9	11.8	7,2	8.9	8	5.8	12.8
26	14.4	8.7	13.7	7,8	7.5	7	7.1	8,5	6.8	14
27	13.9	9.7	13.8	10,3	8.1	7,3	12.7	7,2	7.6	11.4
28	13	10.7	13.3	9,4	12.2	7	13.2	7,4	5.8	12.6
29	12.4	9.2	14.1	9,8	12.6	7,5	6.9	8,3	6.6	13.4
30	12.4	9.3	13.1	9,3	13.1	7,8	8.3	9,4	8	12
31	12.8	9.15	12.2	8,8	8.1	8,6	8.1	8,1	5.6	12.4
32	12.4	9.5	12.6	8,6	14.6	8	11.5	7,7	6.6	13
33	16.7	8.8	12.7	8,3	12.7	8,7	12.7	8,7	7.4	13.6
34	12.8	10.2	13.5	8,6	8.1	8,7	8	7,4	5.4	12
35	11.6	9	12.9	10,3	8.5	9,1	6.7	7,9	6.6	12.6
36	12.6	8	11.8	10,3	13.2	7,4	7.7	8,6	7	13.6
37	12.7	9.4	13.2	13,9	8.5	7,2	8.5	7,2	5.8	13.8
38	13.3	8.8	13.6	8,9	13.8	7,3	8.7	7,7	7.2	11.8
39	12.2	8.3	11.9	9,3	13.1	7	7.1	8,9	7.4	12.6
40	13.9	9.6	12	8,7	13.2	9	7.1	7,3	5.4	13.3

Legenda: HRT s/c: Hard Real Time sem carga

HRT c/c :Hard Real Time com carga

Na Tabela 7 mostra-se o tempo que decorreu desde que foi despoletada uma interrupção até que esta é atendida pela *handler*. Assim, foi obtido o tempo de resposta do sistema de aquisição de dados de tempo real. Ao todo foram recolhidas as quarenta amostras geradas correspondentes a um igual número de interrupções registadas pelo analisador lógico.

Tabela 8 Tempos de resposta à interrupção em *soft real-time*

	Tempos de resposta à interrupção da Porta Paralela							
	Frequência/Período							
	5Hz/200ms		50Hz/20ms		500Hz/2ms		5kHz/200us	
	Tipo de dados							
Num. Amostras	**SRT s/c**	**SRT c/c**	**SRT s/c**	**SRT c/c**	**SRT s/c**	**SRT c/c**	**SRT s/c**	**SRT c/c**
1	7,2	7	8	7	8,3	6	25,6	9,6
2	9,7	7	7,5	6,8	7,7	6,2	16,9	10,5
3	8,2	8	6,9	7	5,9	11,5	17,5	11,8
4	7,4	6,5	9	5,8	6,1	5,6	7,5	6,5
5	8,8	6,2	7,9	11	6,1	6,1	8,5	16,7
6	7,7	6,4	8,7	5,3	7,7	6,1	6,2	17,4
7	8	6,1	7,7	6,3	7,3	6,5	5,8	4,8
8	8,1	6,8	7,1	7	7,3	6	27,2	5,6
9	9,2	6,7	7,9	10,8	7,7	6,1	7,7	6,2
10	9,1	7	9,3	5,5	7,7	6,7	10,1	4,8
11	6	6,8	8,6	5,4	9,9	32,2	7,3	28
12	7,6	7,2	8,8	5,7	7,8	4,2	6,9	6,2
13	6,2	5,8	9,3	6,2	7,1	26	10	4,6
14	8,8	6,7	9,1	6,5	7,1	4,4	7,8	5,4
15	9,3	5,8	8,5	5	44,3	17,8	12,34	5,8
16	7,6	6,7	8,3	6	6,2	7,3	7,3	4,3
17	8,2	6,5	7,9	5,7	6,8	4,3	8,1	5,1
18	8,6	6,2	9,4	4,8	6,7	44,5	6,1	6,3
19	8,3	6,4	9,3	4,7	16,5	28,3	8,3	5,1
20	8,6	6,9	8,9	4,7	19,7	32,6	7,7	5,6
21	7.5	5.8	7	5.8	6.4	5.3	20.6	4.8
22	8.2	6.8	7	6.6	6.2	6.1	6.6	6.3
23	8.6	6.4	8.7	7.3	37.8	5.6	8	4.7
24	8	6.8	7.2	6.4	20.6	5	30	5.6
25	8.6	7.4	8	7.6	5.3	5.8	22.8	6.5
26	7.2	8.1	7.2	12.5	7.1	5.3	12.5	4.6
27	7.1	5.5	7.9	5.6	6.1	5.2	11.9	5.2

Continua na página seguinte

Continuação da tabela da página anterior

	Tempos de resposta à interrupção da Porta Paralela							
	Frequência/Período							
	5Hz/200ms		50Hz/20ms		500Hz/2ms		5kHz/200us	
	Tipo de dados							
Num. Amostras	**SRT s/c**	**SRT c/c**	**SRT s/c**	**SRT c/c**	**SRT s/c**	**SRT c/c**	**SRT s/c**	**SRT c/c**
28	7.1	5.9	9.4	6.2	6.9	4.7	6.9	5.9
29	6.5	6.2	7.3	6.9	6.9	4.5	8.8	4.2
30	7.8	6	8	8.4	6.8	4.5	6.3	5
31	9.3	6.7	6.4	7.1	7.8	5	4.9	5.9
32	8	5.7	9	5.7	5.6	6.6	14.3	4
33	8	6.9	9.4	6.2	7.8	6.2	5.8	4.9
34	6.6	6.2	8.6	7.3	6.9	6.1	6.8	6.4
35	8.5	8	9.7	6.7	6.8	6.3	6.5	4.8
36	8.6	6.8	8.3	7.2	27.4	5.9	6.1	5.2
37	7.1	5.7	7.8	7.2	7.9	42.4	7.2	5.9
38	8.6	7	7.1	7.6	7.3	6	7.2	4.3
39	8.2	5.8	7.5	6.4	33.4	6.5	30.2	5
40	7	8.2	9	7.6	41	45.1	25.3	5.7

Legenda: SRT s/c: *Soft Real-Time* sem carga
SRT c/c: *Soft Real-Time* com carga

Na Tabela 8 apresenta uma situação análoga à da Tabela 7, mas neste caso os tempos de resposta correspondem à tarefa de *soft real-time* que foi criada para ser termo de comparação com a tarefa de *hard real-time*.

6.2. REPRESENTAÇÃO GRÁFICA DOS DADOS OBTIDOS E COMPARAÇÃO ATRAVÉS DE BOXPLOTS

Para obter uma melhor perspectiva sobre os dados registados, na Tabela 7 e Tabela 8, representou-se graficamente os dados e compararam-se os tempos de resposta obtidos em *hard real-time* e *soft real-time*.

Para possibilitar uma melhor percepção da distribuição dos valores em função das frequências foram construídos gráficos *boxplot* conforme é exemplificado na Figura 30.

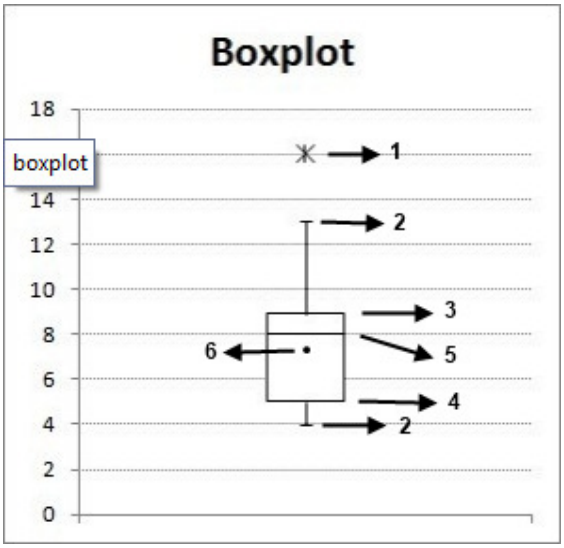

Figura 30 Boxplot : 1-*Outlier*; 2-Limite máximo e mínimo; 3-1º quartil; 4- 3º quartil; 5-Mediana; 6-Média

De uma forma sucinta, visto que não é objectivo da tese, pode-se dizer que o *boxplot* é constituído por uma caixa delimitada pelo primeiro e terceiro quartil, a linha horizontal dentro da caixa é a mediana e o ponto a média, conforme indicado na Figura 30. Existem ainda os "bigodes" que, para além de indicarem a dispersão, também informam sobre a simetria da distribuição dos valores obtidos para os tempos de resposta à interrupção. Os "bigodes" também indicam os valores máximos e mínimos caso não existam *outliers*[17]. Neste caso os "bigodes" identificam o maior e menor valor não *outlier*.

A apresentação de um gráfico *boxplot* sozinho como se encontra na Figura 30 não parece uma grande ajuda. No entanto, representando dois ou mais conjuntos de dados estes gráficos facultam um excelente meio de identificação visual das tarefas (*soft real-time* ou *hard real-time*) com melhor desempenho ao nível dos tempos de resposta do sistema de aquisição de dados em tempo real.

No *boxplot* para cada frequência/período de interrupção foi colocado um conjunto de dados obtidos: tarefa *soft real-time* (STR), tarefa *hard real-time* (HRT), tarefa *soft real-time* com carga (SRT_CC) e tarefa *hard real-time* com carga (HRT_CC).

[17] *Outlier* é uma observação que parece desviar-se acentuadamente dos outros membros da amostra em que ocorre

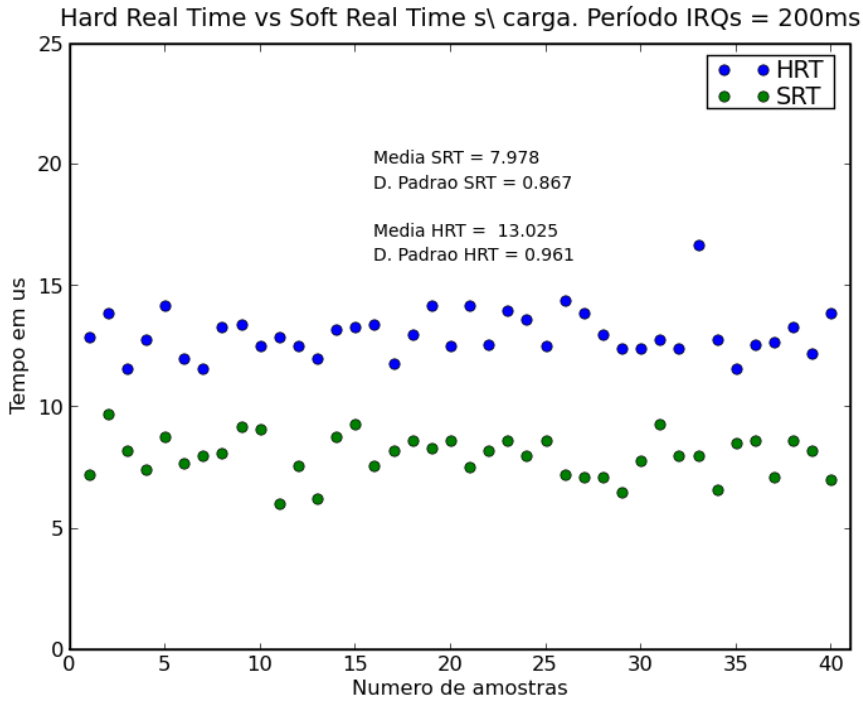

Figura 31 Gráfico de resultados para frequência/período 5Hz/200ms (sem carga)

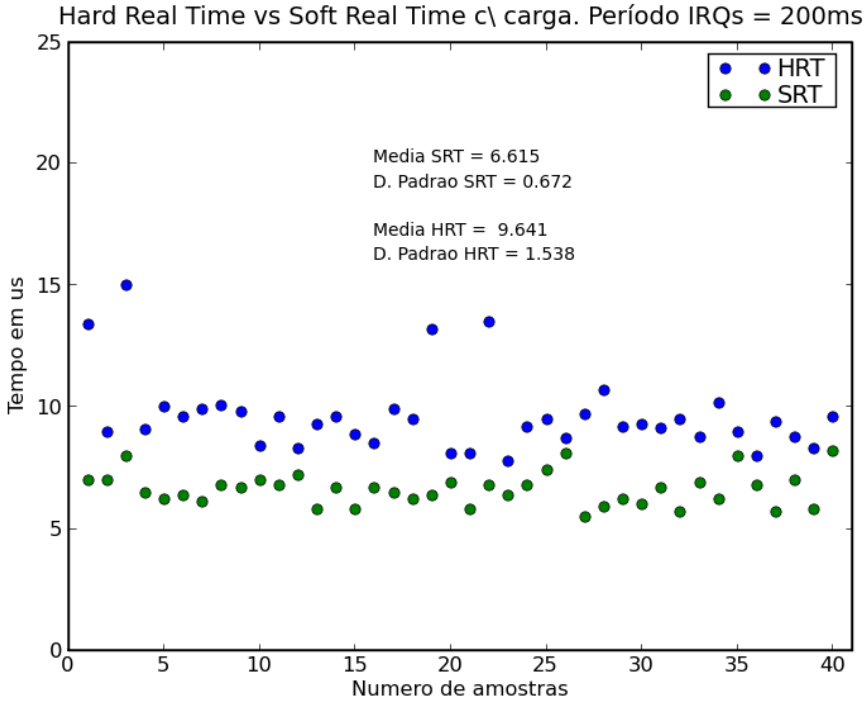

Figura 32 Gráfico de resultados para frequência/período 5Hz/200ms (com carga)

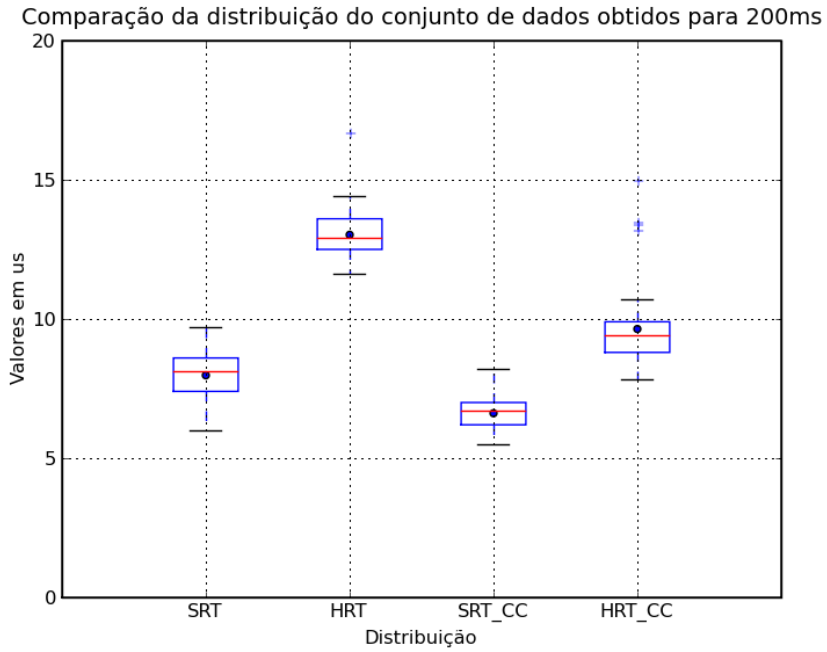

Figura 33 Distribuição dos conjuntos de dados para 5Hz/200ms

Nos gráficos representados nas Figura 31 e Figura 32 nota-se, que em termos médios, o tempo de resposta às interrupções por parte da tarefa de *hard real-time* (HRT) é superior à tarefa de *soft real-time* (SRT). É também visível que os valores obtidos com ambas as tarefas quando o computador está sujeito a carga, Figura 32, são inferiores às mesmas quando o computador se encontra sem carga. E isso é bem explícito quando estes resultados são apresentados sobre a forma de *boxplot* na Figura 33.

Ainda sobre o *boxplot* representado na Figura 33 também é visível que existem *outliers* para o conjunto de valores obtidos com a tarefa *hard real-time*, com e sem carga (HRT e HRT_CC). Estes valores indicam que há dados que se destacam negativamente pois representam tempos de resposta altos. A dispersão de valores neste conjunto de dados não é muito elevada, conforme indica o desvio padrão e os "bigodes" do *boxplot*. No entanto, para os conjuntos de dados representados a simetria da distribuição é melhor para os dados de *hard real-time* do que para os dados de *soft real-time*.

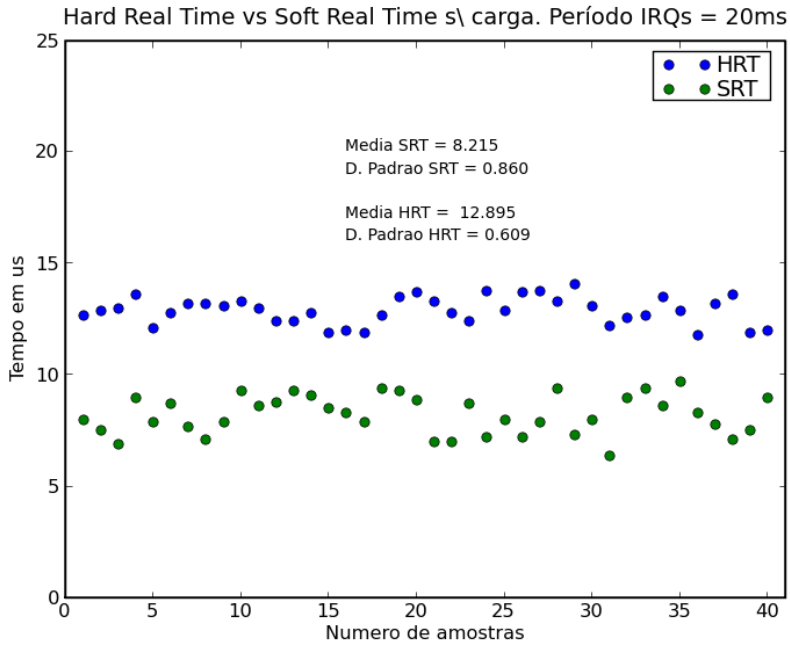

Figura 34 Gráfico de resultados para frequência/período de 50Hz/20ms (sem carga)

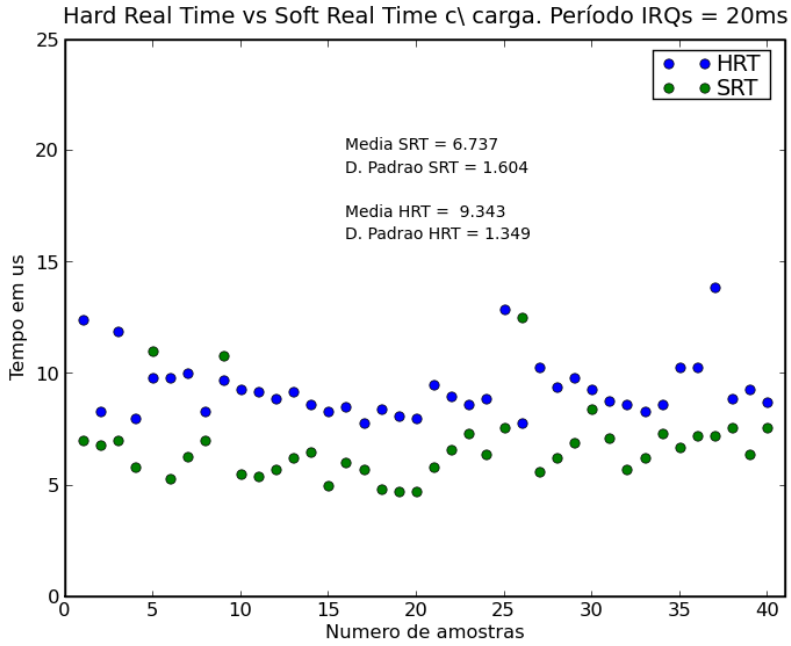

Figura 35 Gráfico de resultados para frequência/período de 50Hz/20ms (com carga)

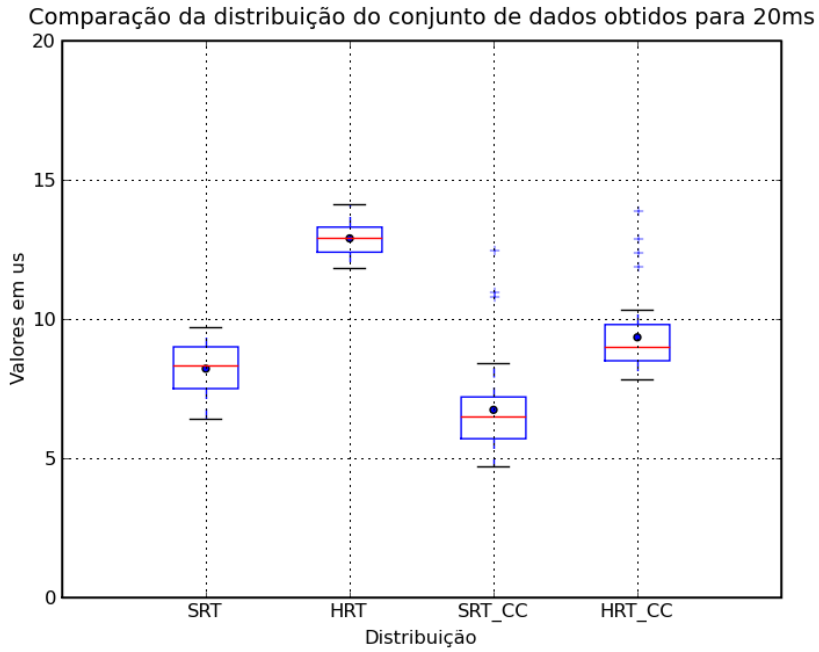

Figura 36 Distribuição dos conjuntos de dados para frequência/período de 50Hz/20ms

Para interrupções com um período de 20ms, registaram-se tempos de resposta às interrupções mais baixos para as tarefas de *soft real-time*. Os tempos de resposta das tarefas quando a máquina não está em carga são superiores aos valores verificados quando a máquina é sobrecarregada. No entanto, pode-se dizer que a dispersão, e os valores máximos dos tempos de resposta são inferiores aos obtidos para períodos de 200ms. Os *Ouliers* aparecem apenas para os conjuntos dos tempos de reposta que foram sujeitos a carga (SRT_CC e HRT_CC). Sendo a dispersão das respostas temporais da tarefa de *soft real-time* superior à tarefa de *hard real-time* conforme indica o desvio padrão na Figura 34 e na Figura 35. A dispersão é um pouco inferior em relação aos tempos de resposta obtidos para 200ms. É ainda visível que há pequenas assimetrias nas distribuições de *soft real-time*, conforme se pode ver nos "bigodes" do *boxplot* da Figura 36.

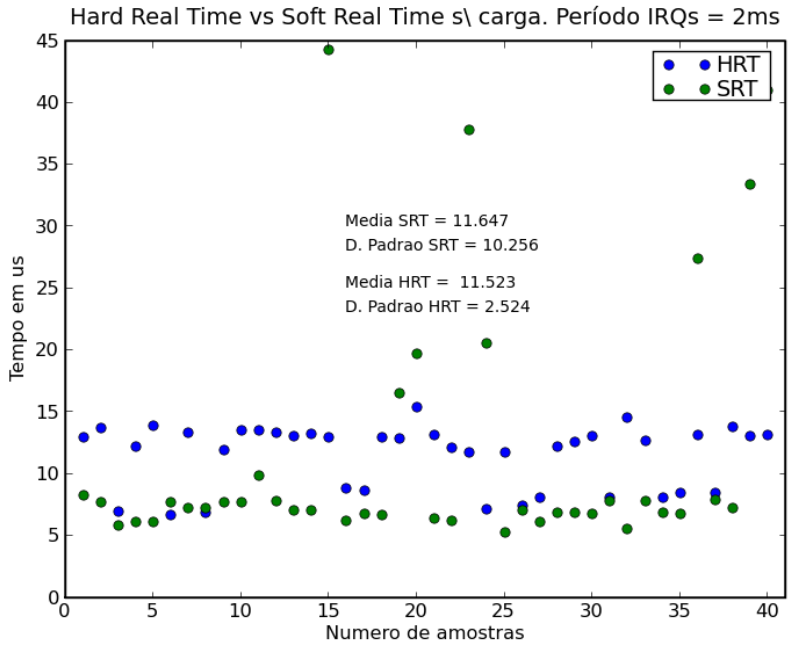

Figura 37 Gráfico de resultados para frequência/período de 500Hz/2ms (sem carga)

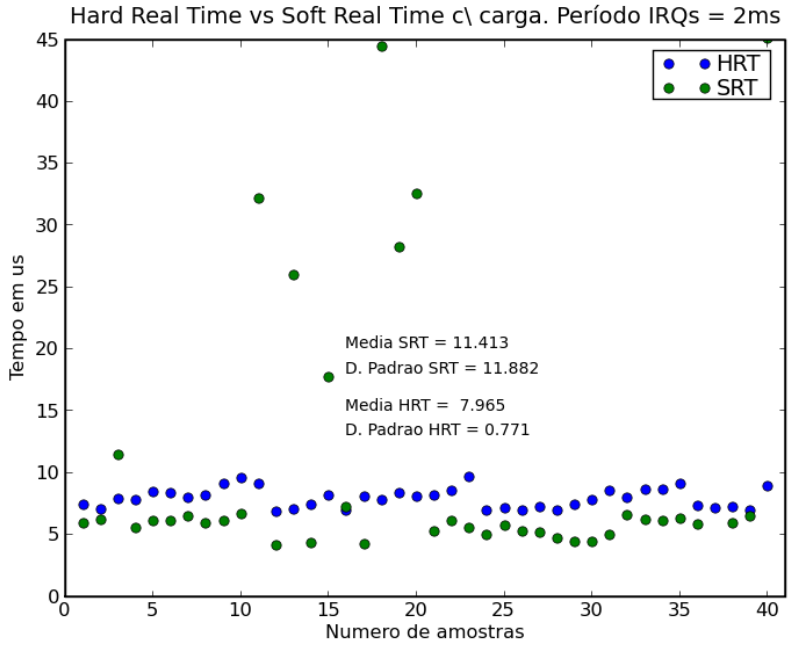

Figura 38 Gráfico de resultados para frequência/período de 500Hz/2ms (com carga)

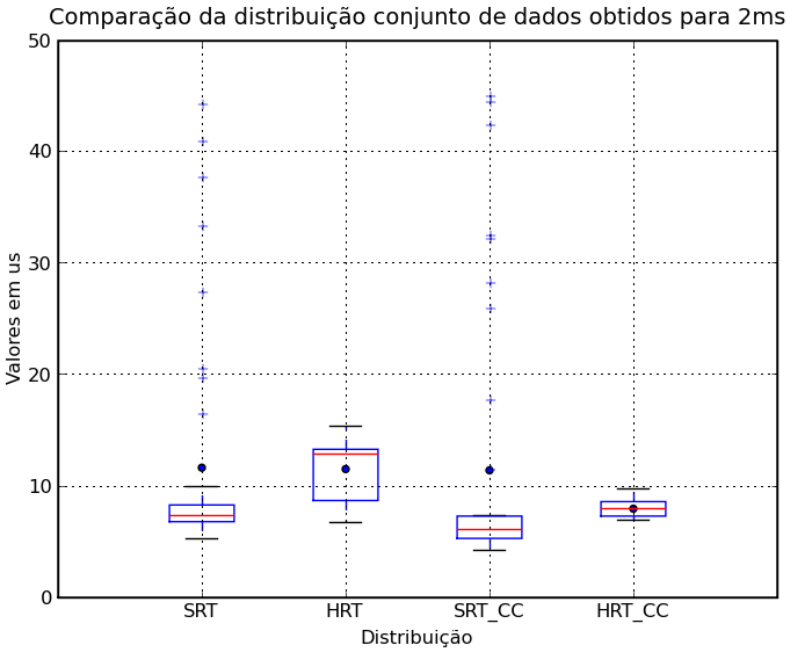

Figura 39 Distribuição dos conjuntos de dados para frequência/período de 500Hz/2ms

Como é visível neste conjunto de gráficos representados pelas Figura 37, Figura 38 e Figura 39, o panorama alterou-se bastante. Verificam-se *outliers* com valores muito altos para os tempos de resposta da tarefa de *soft real-time* (SRT e SRT_CC), chegando a ser quatro vezes superior ao maior valor não *outlier*. Ao contrário do que se verifica para a tarefa de *hard real-time*, onde se verificou uma melhoria significativa, quer devido ao facto de deixarem de existir *ouliers*, como também devido ao valor médio dos tempos de resposta à interrupção ter baixado, quer esteja o computador em carga ou não. Exactamente o oposto do que acontece para a tarefa de *soft real-time*.

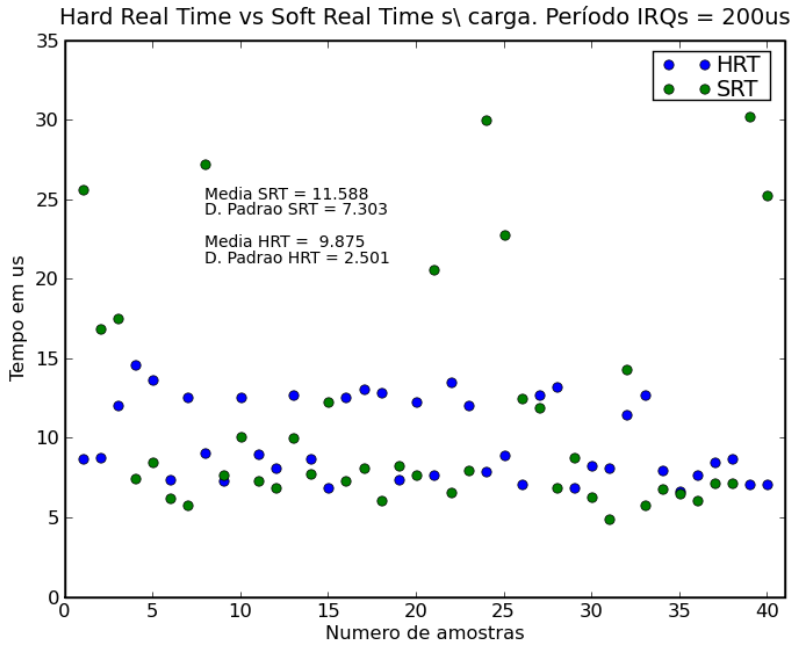

Figura 40 Gráfico de resultados para frequência/período de 5kHz/200us (sem carga)

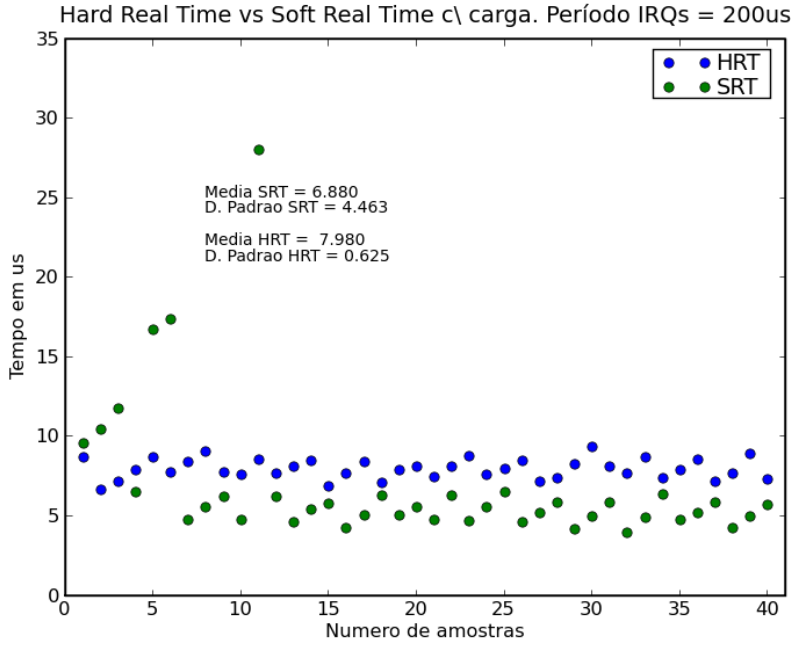

Figura 41 Gráfico de resultados para frequência/período de 5kHz/200us (com carga)

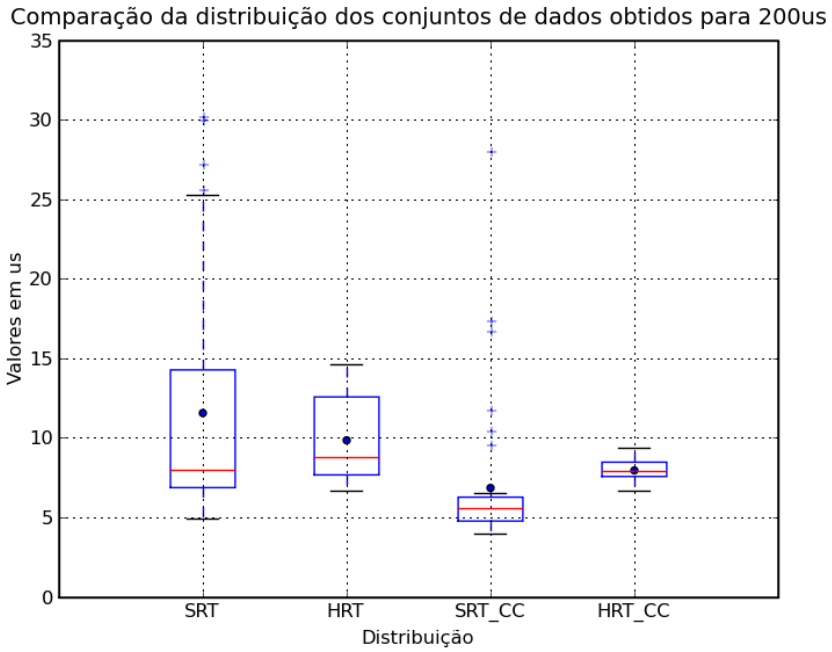

Figura 42 Distribuição dos conjuntos de dados para frequência/período de 5kHz/200us

Nas Figura 40, Figura 41 e Figura 42, para o período de interrupções de 200µs, é de salientar o elevado número de *outliers* nas tarefas de *soft real-time* e o elevado valor que possuem os tempos de resposta à interrupção. É observada uma grande dispersão de dados para os valores obtidos pelas tarefas de *soft e hard real-time* quando estes não se encontram em carga, confirmados pelo valor do desvio padrão, bem como pelos "bigodes" dos respectivos *boxplots*. No entanto, a assimetria da distribuição é mais significativa nos dados obtidos com a tarefa de *soft real-time* do que na tarefa de *hard real-time*. Nas tarefas de *soft real-time* continuam-se a verificar valores muito altos em relação à média dos tempos de resposta de *soft real-time*, que estão sinalizados nos *boxplot* como *outliers*. Facto que já não acontece nas tarefas de *hard real-time*, conforme se pode verificar no gráfico da Figura 42, que apresenta melhores valores, quer ao nível da dispersão, quer ao nível da simetria dos tempos de resposta, bem como na ausência de *outliers*. Confirmando assim o melhor desempenho das tarefa de *hard real-time*.

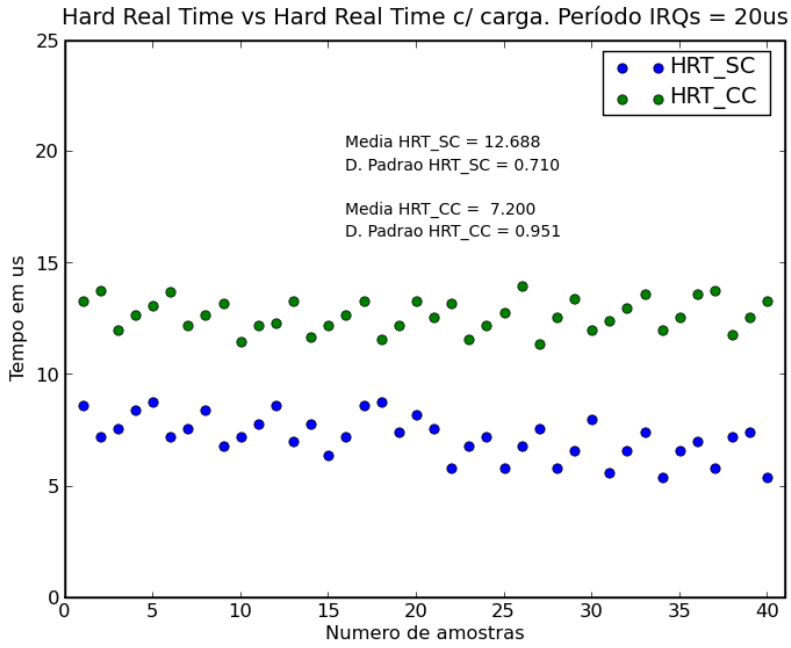

Figura 43 Gráfico de resultados para frequência/período de 50kHz/20us (com carga)

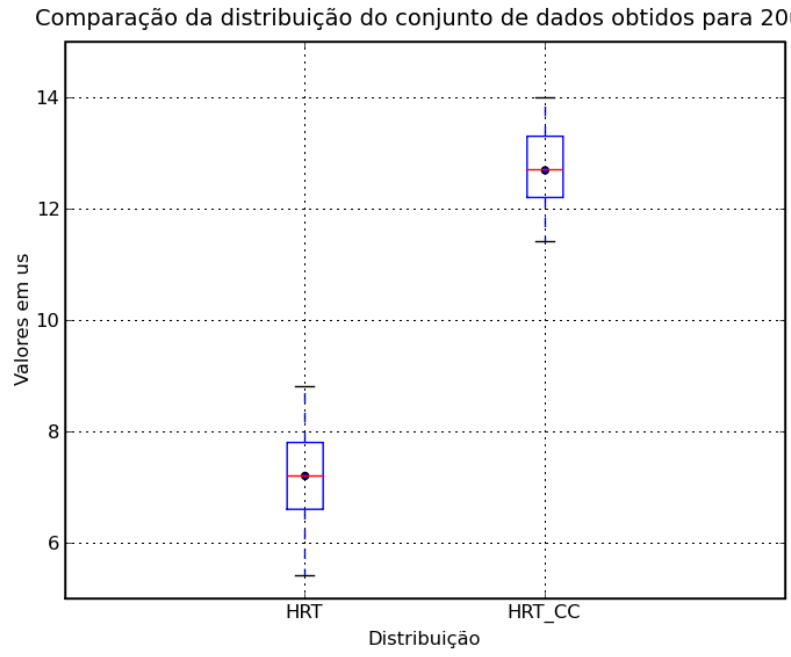

Figura 44 Distribuição dos conjuntos de dados para frequência/período de 50kHz/20us

Nas Figura 43 e Figura 44 são representados graficamente os resultados da tarefa de *hard real-time*, com ou sem carga, para um período de 20µs. Os resultados mostram um desempenho muito bom para o período em questão, não existindo *outliers* que indiciam valores que se distanciam acentuadamente da média de todos os outros. Verifica-se também uma distribuição de valores praticamente simétrica, confirmado pelos "bigodes" do *boxplot* e pela média ser praticamente idêntica à mediana, e uma baixa dispersão dos valores, como se pode ver pelo valor do desvio padrão, e também pelos "bigodes" dos *boxplots*. Para estes valores de frequência de interrupção a tarefa de *soft real-time* começa a falhar o atendimento das interrupções de forma aleatória.

As Figura 45 e Figura 47 mostram o comportamento da tarefa de *soft real-time* versus *hard real-time*, face às interrupções geradas pela porta paralela a 50kHz. A tarefa de *soft real-time* perde a sua capacidade de atender as interrupções, falhando assim a aquisição de dados.

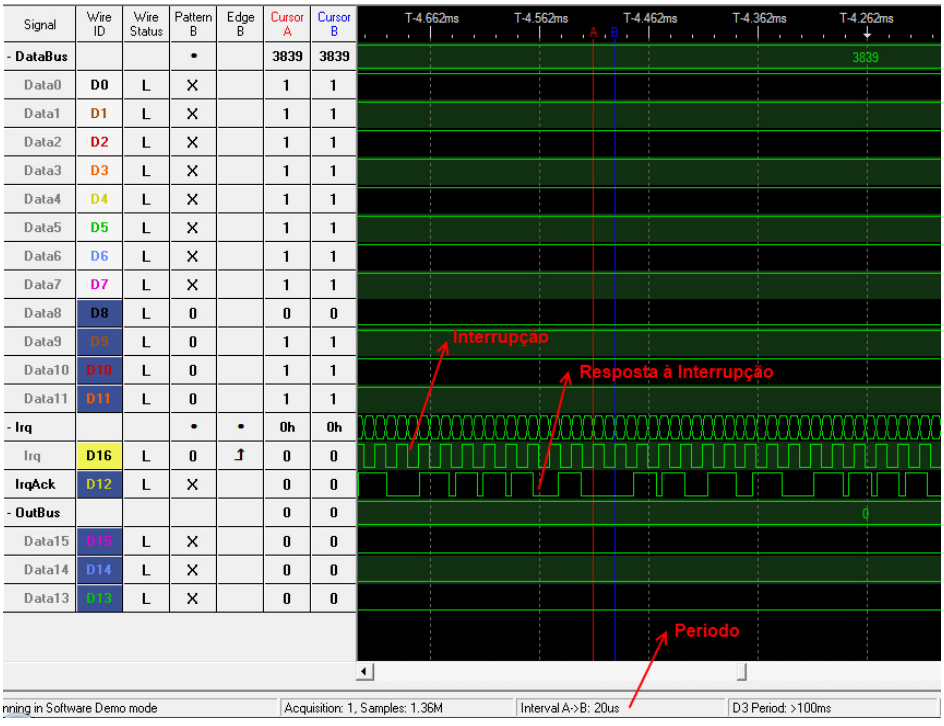

Figura 45 Resposta à interrupção em soft *real-time* sem carga para frequência 50kHz

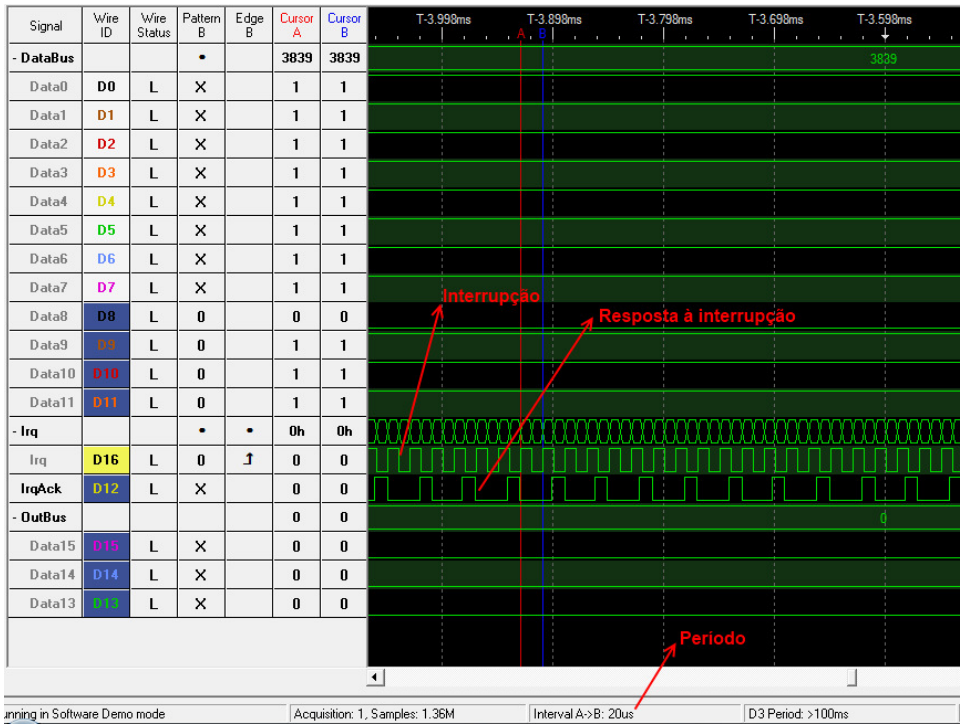

Figura 46 **Resposta à interrupção em *hard real-time* sem carga para frequência 50kHz**

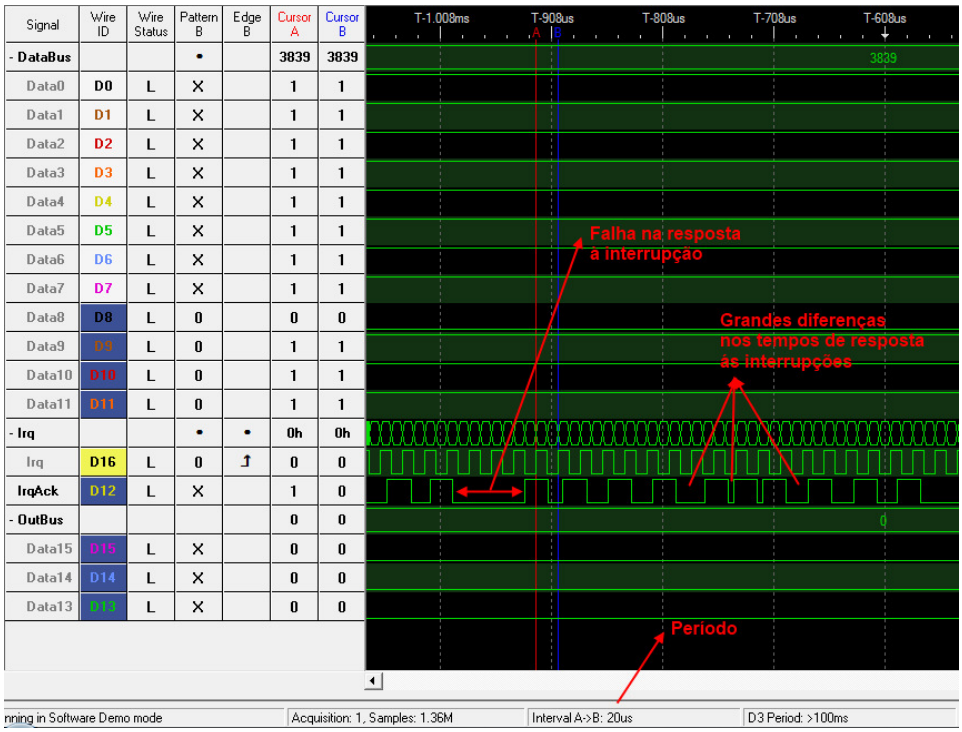

Figura 47 **Resposta à interrupção em *soft real-time* com carga para frequência 50kHz**

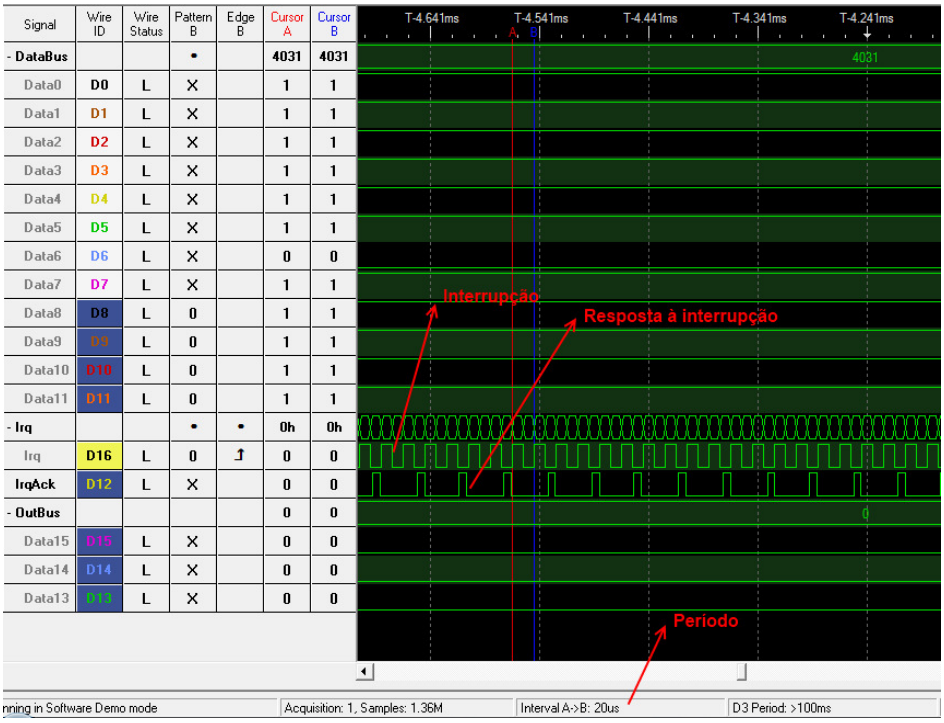

Figura 48 Resposta à interrupção em *hard real-time* com carga para frequência 50kHz

Comparando as imagens anteriores, obtidas com o analisador de sinais, é visível que para a tarefa de *soft real-time*, Figura 45 e Figura 47, as respostas às interrupções falham. Em consequência, falham uma série de leituras da porta paralela não preenchendo assim os pressupostos dos sistemas *hard real-time*. No entanto, a tarefa de *hard real-time* continua a cumprir e a responder às interrupções, conforme se pode verificar nas Figura 46 e Figura 48.

De notar que entre duas leituras da porta paralela (resposta à interrupção) realizadas pela tarefa de *hard real-time*, por exemplo na Figura 46, existe uma interrupção (leitura) que não é atendida. Isto deve-se, ao facto de a *Interrupt Service Routine* (ISR) ser reentrante, como explicado no quinto capítulo. Ou seja, como a periodicidade a que são geradas as interrupções é mais curta que o tempo de execução da tarefa de leitura da porta paralela (ISR), ira ser despoletada uma interrupção durante o processo a leitura da porta paralela. Tendo este problema em conta, sempre que se entra na ISR as interrupções são desactivadas e mais tarde activadas, conforme é descrito no quinto capítulo. Ao aplicar esta medida é garantido que não irão ocorrer tarefas aperiódicas, evitando assim possíveis problemas de consistência de dados.

6.3. DISCUSSÃO DE RESULTADOS

No capítulo dois foi dito que um Sistema Operativo Linux que use um processador actual tem um tempo de resposta a uma interrupção na ordem dos 20µs [4], mas ocasionalmente o tempo de resposta pode ser superior. Isto não se deve ao facto do Linux e do *hardware* em questão não serem rápidos ou eficientes, mas sim ao facto de não ser determinístico.

Em sistemas de tempo real este tempo deve ser determinístico, e inferior ao pior caso [4]. Os dados e gráficos representados neste capítulo confirmam esta afirmação. Pode-se ver que os valores estão abaixo dos tais 20µs que é o tempo de resposta à interrupção no Linux. No entanto, não deixa de ser verdade que a tarefa de *hard real-time* desenvolvida em Linux/RTAI, mesmo tendo um tempo de resposta maior que a tarefa de *soft real-time*, respondeu sempre às interrupções, comportando-se sempre de forma determinística durante os ensaios para os períodos/frequências estabelecidos. Os valores obtidos confirmam isso. Mesmo na situação em que foram colocadas ambas as tarefas sobre carga, o comportamento da tarefa de *hard real-time*, de um modo geral, foi sempre superior.

Tendo em conta que o comportamento da tarefa de *soft real-time* tenha tido melhores resultados nas frequências mais baixas (inferiores a 500Hz), ou seja, garantiu determinismo, e ao mesmo tempo um baixo tempo de resposta. Não deixa de ser evidente que estas características se vão degradando com o aumento da frequência a que a interrupção é gerada. Por exemplo, o pior caso em termos de tempo de resposta para a tarefa de *hard real-time* é de 16,7µs, para um impulso gerado a uma frequência de 5Hz (Ver Tabela 7 e Figura 31) e sem carga. No caso da tarefa de *soft real-time*, o pior caso acontece a frequências de 500Hz (ver Figura 32 e Figura 37), também sem carga, e tem um tempo de resposta de 44,3µs. Tendo em atenção que a média dos valores neste exemplo é de 13,0µs (tarefa de *hard-real time*) e 11,6µs (tarefa de *soft real-time*), surge uma diferença em relação à média de 3,7µs e 32,7µs, respectivamente. Em termos percentuais, o tempo de resposta é superior à média para a tarefa de *hard real-time* de 28,5%, e de 281,9% para a de *soft real-time*. O que é uma diferença enorme e muito significativa.

As diferenças de comportamento ainda se tornam mais evidentes a 50kHz. A tarefa de *soft real-time* com ou sem carga, falha o atendimento às interrupções, ver Figura 45 e Figura 47, para além de se verificarem enormes diferenças nos tempos de resposta (Figura 47). Se

este comportamento acontecesse com uma tarefa de *hard real-time* poderia originar uma falha catastrófica.

Os resultados apresentados mostram, no seu conjunto, que a tarefa de *hard real-time* desenvolvida consegue realizar aquisição de dados via porta paralela, garantindo determinismo, até frequências de 50kHz. Validando assim as opções tomadas durante a elaboração da tarefa de *hard real-time* e da aplicação de interface humana, que foram descritas no Capitulo 5.

7. CONCLUSÕES

Conforme demonstrado no sexto Capítulo, usando o RTAI/Linux, foi atingido o objectivo de realizar um sistema de aquisição de dados em tempo real e criar uma tarefa de *hard real-time*, que funciona ao nível do kernel, capaz de adquirir dados, via porta paralela, e enviá-los para uma aplicação que funciona ao nível de utilizador (LINUX) para que esses mesmos dados pudessem ser visualizados pelo utilizador.

Foi desenvolvido com sucesso uma interface *hardware* capaz de adquirir uma palavra de 12 bits de um ambiente externo. A interface foi implementada usando a porta paralela do PC. Nesta interface, para além de se simular a leitura dos doze bits da porta paralela, também foram reservados quatro bits que foram usados para informar o utilizador sobre a existência de interrupções, ou se o *buffer* circular estava cheio, por exemplo. Este interface teve um papel importante, pois permitiu à tarefa de tempo real comunicar com o mundo exterior e deu a perceber que, apesar de estar em desuso, a porta paralela pode ter aplicações extremamente válidas.

A par do objectivo principal, sistema de aquisição de dados em tempo real, foi realizada uma comparação com uma tarefa de *soft real-time*. Conforme se pode ver no Capítulo 6, foram obtidos melhores resultados, especialmente quando as interrupções eram despoletadas a frequências na ordem dos 50kHz. O tempo de resposta da tarefa de *hard*

real-time é superior ao tempo de resposta da tarefa *soft real-time*, no entanto este não é o factor primordial. Importante é que o sistema seja determinístico, para poder garantir que os requisitos temporais sejam cumpridos em função da variação das cargas a que o sistema está sujeito. Situação que só aconteceu, conforme pode ser visto no Capitulo 6, para a tarefa de *hard real-time* usando o interface e módulos do RTAI.

Um outro objectivo atingido foi o estudo de serviços e funções disponibilizados pelo RTAI. O estudo de comunicação entre tarefas e processos num sistema RTAI/Linux era fundamental, visto que era necessário trocar informação entre tarefas e processos. Neste estudo ficou claro que os métodos mais utilizados de comunicação entre tarefas e processos é a memoria partilhada, FIFOs e *mailboxes*, sendo as últimas as mais flexíveis. O entendimento do funcionamento do escalonador do RTAI, na forma como processa as ISR (*Interrupt Service Routines*), e como é feita a gestão entre o RTAI e o Linux para as interrupções geradas por *hardware* também foi da maior importância, pois as interrupções são normalmente usadas para lidar com eventos similares à tarefa de aquisição de dados em tempo real desenvolvida. Outros serviços complementares, como os semáforos, *watchdog*, e o LXRT, foram também pesquisados e abordados, embora com menos profundidade. Esta pesquisa influenciou substancialmente a forma como mais tarde foi desenvolvida a tarefa de *hard real-time*.

Também foi objectivo identificar outros sistemas operativos de tempo real para além do RTAI. Estes sistemas foram identificados e classificados em função da sua arquitectura, como pode ser visto na subsecção 2.7, ou seja em função da sua integração com o kernel do sistema operativo e com o *hardware* da máquina. Neste sentido foram identificados vários sistemas alternativos como o ADEOS, o XENOMAI, o RTLinux e TimeSYS entre outros.

Durante o trabalho realizado ficou claro que existem pressupostos e restrições que devem ser respeitadas quando se projecta um sistema de tempo real. Estes pressupostos têm efeito sobre o tipo de sistema a utilizar, um pressuposto "*soft"* que cumpre em média os requisitos temporais, ou um pressuposto "*hard"* em que uma única falha pode ser catastrófica. Neste caso, deve ser dada atenção às restrições temporais do sistema: *soft, firm* ou *hard*. Foi identificado claramente como definir que tipo de tarefa a utilizar em cada situação: periódicas ou orientadas a eventos (*event driven*). Isto é, conforme seja requerido em intervalos de tempo pré definidos ou despoletadas através de um evento.

Para desenvolvimentos futuros, seria interessante estudar com mais pormenor o módulo do LXRT do RTAI e também o serviço de comunicação entre tarefas e processos *mailbox*. O estudo de outros interfaces para além da porta paralela também seria interessante. Futuramente poderia ser desenvolvida uma interface gráfica, onde para além de visualizar os resultados obtidos, também fosse possível visualizar graficamente resultados temporais que caracterizam os sistemas de tempo real.

Referências Documentais

[1] Phillip A. Laplante — *Real-time systems design and analysis : an engineer's handbook* — 3rd ed, IEEE PRESS. ISBN: 0-471-22855-9

[2] Qing Li; Carolyn Yao — *Real-Time Concepts for Embedded Systems*, CMP Books 2003. ISBN:1578201241

[3] Stankovic, J.A., Spuri, M., Dinatale, M., and Buttanazzo, G. — *Implications of Classical Scheduling Results for Real Time Systems* — IEEE Computer Volume 28, Number 6, June 1995, pp 16-21.

[4] M. Tim Jones — Anatomy of real-time Linux architectures. IBM Technical library, http://www.ibm.com/developerworks/linux/library/l-real-time-linux/.

[5] Thomas H. Corman, Charles E. Leiserson, Ronald L. Livest, Clifford Stein — *Introduction to Algorithms Second Edition* — MIT PRESS 2001, pp 273. ISBN: 0-262-03293-7

[6] Victor Yodaiken, Michael Barabanov — A Real Time Linux — New Mexico Institute of Technology. http://users.soe.ucsc.edu/~sbrandt/courses/Winter00/290S/rtlinux.pdf

[7] Karim Yaghmour — The Real-Time Application Interface — Linux Symposium 2001. http://www.linuxsymposium.org/archives/OLS/Reprints-2001/yaghmour.pdf

[8] Pasi Sarolahti, — Real-Time Application Interface, Research seminar on Real-Time Linux and Java — University of Helsinki, Department of Computer Science, 26th February 2001.

[9] Durda IV, Frank — *Centronics and IBM Compatible Parallel Printer Interface Pin Assignment Reference*. http://nemesis.lonestar.org/reference/computers/interfaces/centronics.html.

[10] Craig Peacock — *Interfacing the Standard Parallel Port.* 15th June 2005. http://www.beyondlogic.org/spp/parallel.htm

[11] Giovanni Racciu, Paolo Mantegazza — *RTAI User Manual 3.4*. — 2006 – rev 0.3.

[12] Intel — 8254 PROGRAMMABLE INTERVAL TIMER Data sheet, September 1993

[13] Sousa, Cristóvão — *How to Install Rtai - openSuse way* — 2007 – v0.1. http://www.isr.uc.pt/~rui/str/howto_install_rtai.html.

[14] Pedreiras, Paulo — Conceitos básicos de Tempo-Real — Departamento de Electrónica, Telecomunicações e Informática Universidade de Aveiro, V1.1 Outubro/2009

[15] Harco Kuppens — FIFOs & *Messages* & *Mailboxes* — Radboud Uneversity Nijmegen, Computer Science Department, 19 September 2007. http://www.cs.ru.nl/lab/rtai/exercises/5.FIFOs-Messages-Mailboxes/Exercise-5.html

[16] Fred Proctor – *Real-Time Linux Tutorial* – National Institute of Standards and Tecnology, Control Systems Group, 03 January 2006. http://www.isd.mel.nist.gov/projects/rtlinux/rtutorial-2.0/doc/ack.html

[17] Walter Fetter Lages, – *RTAI Políticas de Escalonamento* – Universidade do Rio Grande do Sul, Escola de Engenharia, Departamento de Engenharia Electrica, ENG04008 Sistemas de Tempo Real. www.ece.ufrgs.br/~fetter/eng04008

[18] Harco Kuppens – LinuX RealTime – Radboud Uneversity Nijmegen, Computer Science Department, 19 September 2007. http://www.cs.ru.nl/lab/rtai/experiments/3.LXRT/Experiment-3.html

Anexo A. Tarefa de tempo real para leitura de porta paralela: rt_lpt_task-driver.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <rtai.h>
#include <rtai_sched.h>
#include <rtai_fifos.h>
#include <rtai_nam2num.h>
#include <rtai_shm.h>
#include "param.h"



// EndereÃ§o base da porta paralela.
#define BASE 0x378
// Em modo ECP o registo ECR permite escolher o modo de funcionamento PS/2 (Byte
Mode) e todos os outros.
// Caso a porta esteja configurada na bios para PS/2 não é necessá¡rio fazer mais
nada.
#define ECR BASE + 0x402

#define DATA BASE + 0
#define STAT BASE + 1
#define CONT BASE + 2



#define Nb12 0x800


#define IRQ 7



MODULE_DESCRIPTION("My kernel module");
MODULE_AUTHOR("Ivo Daniel Alves (1960576@isep.ipp.pt)");
MODULE_LICENSE("$LICENSE$");

dados DATA_SRT[SHM_HOWMANY];

static RT_TASK send_task;
static RTIME tick_period;
static dados *datashm=0;


static unsigned int t=0;
unsigned int i;

void printb(unsigned int n, unsigned int nb) {
       unsigned int t,i=0;

       for(t=nb; t>0; t=t>>1)
       {
              if (n & t) rt_printk("1");
              else rt_printk("0");
              i++; if (i%4==0) rt_printk(" ");

       }
}
```

```
/*--------FUNÇÃO/HANDLER,O QUE IRÁ SER DESPOLETADA PELA INTERRUPÇÃO PARA LER A
PORTA PARALELA(HANDLER)-----------------------------------------------------*/

static void isr_ler_lpt (void){
 /* Para nÃ£o ocorrer reentrÃ¢ncia na funÃ§Ã£o de ISR devemos desactivar as
interrupção quando entramos na funçãoo e restaurar a interrupçãoo quando ante de
sai*/

unsigned char data, stat, cont, ret;
unsigned int read, n=0;

outb(inb(CONT) & 0xEF, CONT); /*desliga as interrupÃ§Ãµes enquanto atende uma
interrupÃ§Ã£o*/

outb((0x1^0x0b)|inb(CONT), CONT); /*Escreve no led vermelho*/


/*-------------------------------------------------------------------------------
-------------------------------------------------------------------------------*/

/*  Rotina para ler os 12 bits em no registo de DATA e STATUS e 0rdena-los  */

       stat = inb(STAT);
       stat = (stat>>3)&0x7|(~stat>>4)&0x8;
       data = inb(DATA);
       read = (stat<<8)|data;
/*-------------------------------------------------------------------------------
-------------------------------------------------------------------------------*/


printb(read, Nb12); /* função para printar o valor lido em binÃ¡rio e organizar
os 12 bits de 4 em 4 bits*/

t++; // contador que vais ser usado no vector circular

if(datashm[n].nova_leitura == 0) /*verifica se a posição no vector está livre*/
       {
n=t%SHM_HOWMANY; /*incrementa o vector*/

datashm[n].data = read; /* coloca o que foi lido na porta paralela no campo da
estrutura a passar a o nivel de utilizador*/
datashm[n].nova_leitura = 1; /*após ler, marca a posição no vector a indicar que
tem informação a ser lida*/

rt_printk("Valor do contador [t], (%d)\n",n);
rt_printk("Valor contido em ******* (%d)\n",datashm[n].data);
rt_printk("nova leitura(%d)\n", datashm[n].nova_leitura);

outb((0x02^0x0b)|inb(CONT), CONT); /*Led verde on*/
}
else{
outb((0x04^0x0b)|inb(CONT), CONT); /* led vermelho amarelo on*/
}
cont = inb(CONT);
rt_printk("this was write activated by irq %d\n", cont);

/*rt_busy_sleep (10000000);  temporiza para vermos o led apagar e acender (ajuda
no debug)*/
outb((0x00^0x0b)|inb(CONT), CONT); /* apaga os leds*/

outb(inb(CONT) | 0x10, CONT);  /*voltar a activar a interrupção*/
rt_ack_irq(7);
}
```

```
/*..............CONFIGURAÇÃO DA FUNÇÂO INIT DO MODULO.....................*/


static int rt_lpt_task_init_module(void)
{

unsigned char cont;
unsigned int ret;



printk( KERN_DEBUG "Module rt_lpt_task init\n" );

/*-------------------------------------------------------------------------------
-------------------------------------------------------------------------------*/

/* alocar memória partilhada para passar os dados de adquiridos na rotina de
serviço de interrupção para o utilizador*/

datashm = rtai_kmalloc(SHM_KEY,SHM_HOWMANY*sizeof(dados));
       if (0 == datashm){ // NÃO consigo Alocar memória. Talvez muito grande!
              rt_printk ("Erro a alocar a memória\n");
              return -ENOMEM;
              }


/*inicialização dos campos da estrutura que vai passar por memoria partilhada*/

for (i=0;i<SHM_HOWMANY;i++)
{
datashm[i].nova_leitura=0;
datashm[i].irq_ack_counter=0;
datashm[i].data=0;
}
//Fim da inicialização campos de estrutura

/*-------------------------------------------------------------------------------
-------------------------------------------------------------------------------*/


/* Ligar a rotina de serviço de interrupção, também conhecida por ISR, à  porta
paralela (lpt).*/
/* O IRQ esta definido com varialvel global "#define IRQ 7"*/

ret = rt_free_global_irq(IRQ); /* Se a interrupção está em uso liberta-a*/
if (ret){
       rt_printk("problemas a libertar interrupção\n", ret);
       }

ret = rt_request_global_irq(IRQ, isr_ler_lpt);/* liga a rotina de serviço de
interrupção (isr_ler_lpt) à  interrupção da porta paralela */

if (ret){
       if(ret== -EINVAL){ /* O irq não tem um valorou o handler é nulo*/
              printk("IRQ não valido\n");
              }
       else if (ret == -EBUSY) { /* O handler já está a ser utilizado*/
              printk("Este IRQ já está em uso pelo RTAI");
       }
}

rt_enable_irq(IRQ); /*Em vez de rt_enable_irq(IRQ) Também se pode usar
rt_startup_irq*/

/* Fim da configuração da rotina se serviço de interrupção*/
```

```
/*-------------------------------------------------------------------------------
-------------------------------------------------------------------------------*/


/*habilitar as interrupções da porta paralela ao ní-vel do hardware*/

outb(inb(ECR)&0x1f|0x20, ECR); //Activa o modo PS/2
cont= inb(CONT);
printk("Estado do registo de controlo após activar o PS2: %d\n", cont);

outb(inb(CONT)|0x20, CONT); //Activa o bidirecional (0x20);
cont= inb(CONT);
printk("Estado do registo de controlo após activar a bidirecionalidade: %d\n",
cont);

outb(inb(CONT)|0x10,CONT);//Activa IRQ (0x10)
cont= inb(CONT);
printk("Estado do registo de controlo após activar o IRQ: %d\n", cont);

/* Fim das configurações para habilitar as interrupções ao nivel do hardware*/

}

/*------------------INICIO DA FUNÇÃO EXIT  DO MODULO-----------------------------
-------------------------------------------------------------------------------*/


static void rt_lpt_task_exit_module(void)
{

printk( KERN_DEBUG "Module rt_lpt_task start exiting ...\n" );

/*Desactivar ao nivel do hardware*/
/*tenho de desabilitar o PS2 e IRQ e o bidireccional*/

outb(inb(CONT)& 0xEF,CONT);/*Desactiva IRQ. 0xEF é 0x10 negado*/
printk ("Desabilitar IRQ ao nivel da porta paralela ao nível do hardware--> [OK]
\n");
outb(inb(CONT)& ~0x20,CONT); /*Desactiva o bidirecional*/
printk ("Desabilitar Birecionalidade ao nível da porta paralela ao nivel do
hardware--> [OK] \n");
outb((inb(ECR)|~0x1f)& ~0x20, ECR); /*Desativa Activa o modo PS/2*/

       /* PS2 e IRQ e o bidireccional desactivados*/
       /* Fim de desactivaçãoo ao nível do hardware*/

/*-------------------------------------------------------------------------------
-------------------------------------------------------------------------------*/

       /*Desactivar o RTAI*/
rt_disable_irq(IRQ); /*desabililar as interrupções no RTAI*/
rt_printk ("Desabilitar IRQ --> [OK] \n");
rt_free_global_irq(IRQ); /* Liberta a rotina de serviço de interrupção, aka ISR*/
rt_printk ("rt_free_global_irq[IRQ] --> [OK] \n");
stop_rt_timer(); // para o timer da tarefa de tempo real
rt_printk ("Para rt_timer --> [OK] \n");
rtai_kfree(SHM_KEY); /*liberta a memoria partilhada*/
rt_printk ("Libertar Memoria --> [OK] \n");
rt_busy_sleep (10000000); /*espera 1 milisegundo*/
rt_printk ("Bye Bye \n"); /*tchau*/
       return 0;
}

module_init(rt_lpt_task_init_module);
module_exit(rt_lpt_task_exit_module);
```

Anexo B. Aplicação de visualização de dados ao nível do utilizador: user.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <signal.h>
#include <rtai_shm.h>
#include <rtai_nam2num.h>
#include "param.h"

dados DATA_SRT[SHM_HOWMANY];

static int end;
static void endme(int dummy) { end=1; }

int main (void)
{

printf ("\n"); // saltar uma linha

unsigned int t=0,n=0;

static dados *datashm;


datashm = rtai_malloc (SHM_KEY, SHM_HOWMANY*sizeof(dados));

 if (0 == datashm){
       fprintf(stderr, "Impossivel ALOCAR a memoria partilhada \n");
       return 1;
       }

signal(SIGINT, endme);

t++;
while (!end){

if (datashm[t%SHM_HOWMANY].nova_leitura == 1)
{

printf ("Lido [ %d ], OBuffer [ %d ]"
       " %%OBuffer [ %d%% ] ... \r",     /*IrqTime - CPUTime [ %d ] se for preciso
usar pôr atrás das reticencias*/
             datashm[n].data, n, (n*100)/SHM_HOWMANY); /*, datashm[n].time-
datashm[n].time_cpu (ver o comentário anterior)*/

fflush(stdout);
datashm[n].nova_leitura = 0;
n=t%SHM_HOWMANY;
}
else if (datashm[t%SHM_HOWMANY].nova_leitura == 0){

       printf("Posição vazia ... provavelmente não há interrupções");
}

else
```

```
{ printf ("Outro problema ...\n");
                                    }
}

rtai_free (SHM_KEY, &datashm);
printf(" \n Bye \n");
return 0;
}
```

Anexo C. Tarefa de Soft Real-Time: lpt_module_no_rt-driver.c

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/interrupt.h>
#include <asm/io.h>


#define BASE 0x378

#define ECR BASE + 0x402
#define BASE 0x378
#define DATA BASE + 0
#define STAT BASE + 1
#define CONT BASE + 2
#define IRQ 7

#define Nb12 0x800

MODULE_DESCRIPTION("soft real-time module");
MODULE_AUTHOR("Ivo Daniel Alves (1960576@isep.ipp.pt)");
MODULE_LICENSE("$LICENSE$");

void printb(unsigned int n, unsigned int nb) {
       unsigned int t,i=0;

       for(t=nb; t>0; t=t>>1)
       {
              if (n & t) printk("1");
              else printk("0");
              i++; if (i%4==0) printk(" ");

       }
}


static int handler(void)
{
 unsigned char data, stat,cont,read, ret;

       outb((0x0)^0xb|(0x20|~0x10), CONT);


       stat = inb(STAT);
       stat = (stat>>3)&0x7|(~stat>>4)&0x8;
       data = inb(DATA);
       read = (stat<<8)|data;
       printb(read, Nb12);


       outb((0x1)^0xb|0x20|~0x10, CONT); /*ligar led */

       outb((0x1)^0xb|0x20|0x10, CONT);  /*voltar a activar a interrupção*/

       printk(">>> PARALLEL PORT INT HANDLED\n");
       return IRQ_HANDLED;
}
```

```
static int lpt_module_no_rt_init_module(void)
{
	int ret,cont;
	printk( KERN_DEBUG "Module lpt_module_no_rt init\n" );



	ret = request_irq(7, handler, IRQF_DISABLED, "parallelport", NULL);
/*habilitar vector de interrupção*/
	enable_irq(7);

	outb(inb(ECR)&0x1f|0x20, ECR); /*Activa o modo PS/2*/

	cont= inb(CONT);
	printk("control: %d\n", cont);

	outb((0x0)^0xb|(0x20|0x10), CONT); /*Activa o bidirecional (0x20); Activa
IRQ (0x10); e coloca os 4 bits de (leds) "off"*/

	cont= inb(CONT);
	printk("control: %d\n", cont);

	// Fim das configurações


	printk("Gerar interrupções e adquirir dados (intr/ACK = pin 10)\n");


	return 0;
}

static void lpt_module_no_rt_exit_module(void)
{

	disable_irq(7);
	free_irq(7, NULL);
	printk( KERN_DEBUG "Module lpt_module_no_rt exit\n" );
}

module_init(lpt_module_no_rt_init_module);
module_exit(lpt_module_no_rt_exit_module);
```

Anexo D. Código dos gráficos Hard Real-Time vs Soft Real-Time (exemplo para 500Hz sem carga)

```
import numpy
import pylab
import decimal

irq = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40]

srt = [8.3,7.7,5.9,6.1,6.1,7.7,7.3,7.3,7.7,7.7,9.9,7.8,7.1,7.1,44.3,6.2,6.8
,6.7,16.5,19.7,6.4,6.2,37.8,20.6,5.3,7.1,6.1,6.9,6.9,6.8,7.8,5.6,7.8,6.9,6.8,27.4
,7.9,7.3,33.4,41]

rt = [13,13.7,7,12.2,13.9,6.7,13.4,6.9,12,13.5,13.5,13.4,13.1,13.3,13,8.9,8.7,
13,12.9,15.4,13.2,12.1,11.8,7.2,11.8,7.5,8.1,12.2,12.6,13.1,8.1,14.6,12.7,8.1,8.5
,13.2,8.5,13.8,13.1,13.2]



Msrt = (numpy.mean(srt))
DPsrt = (numpy.std(srt))

Mrt = (numpy.mean(rt))
DPrt = (numpy.std(rt))



pylab.text(16,30, 'Media SRT = %.3f' %Msrt, fontsize=10, va='bottom')
pylab.text(16,28, 'D. Padrao SRT = %.3f' %DPsrt, fontsize=10, va='bottom')
pylab.text(16,25, 'Media HRT =  %.3f' %Mrt, fontsize=10, va='bottom')
pylab.text(16,23, 'D. Padrao HRT = %.3f' %DPrt, fontsize=10, va='bottom')



pylab.subplot(111)
pylab.plot(irq,rt, 'bo')
pylab.plot(irq,srt, 'go')
pylab.xlabel('Numero de amostras')
pylab.ylabel('Tempo em us')



pylab.title(u'Hard Real Time vs Soft Real Time s\ carga. Período IRQs = 2ms')
pylab.legend( ('HRT', 'SRT') )



pylab.axis([0, 41, 0, 45])
#guardar imagem
pylab.savefig('/home/ivo/Desktop/py/imagens_graficos/grafico_de_amostras/500hzSC.
png')
pylab.show()
```

Anexo E. Código dos gráficos Boxplot (exemplo para 500Hz)

```
#!/usr/bin/python

#
# 500hz boxplot code
#

from pylab import *

import numpy as np


srt = [8.3,7.7,5.9,6.1,6.1,7.7,7.3,7.3,7.7,7.7,9.9,7.8,7.1,7.1,44.3,6.2,6.8,
6.7,16.5,19.7,6.4,6.2,37.8,20.6,5.3,7.1,6.1,6.9,6.9,6.8,7.8,5.6,7.8,6.9,6.8,27.4,
7.9,7.3,33.4,41]


rt = [13,13.7,7,12.2,13.9,6.7,13.4,6.9,12,13.5,13.5,13.4,13.1,13.3,13,8.9,8.7,
13,12.9,15.4,13.2,12.1,11.8,7.2,11.8,7.5,8.1,12.2,12.6,13.1,8.1,14.6,12.7,8.1,8.5
,13.2,8.5,13.8,13.1,13.2]

srt_c = [6,6.2,11.5,5.6,6.1,6.1,6.5,6,6.1,6.7,32.2,4.2,26,4.4,17.8,7.3,4.3,
   44.5,28.3,32.6,5.3,6.1,5.6,5,5.8,5.3,5.2,4.7,4.5,4.5,5,6.6,6.2,6.1,6.3,5.9,42.
   4,6,6.5,45.1]

   rt_c = [7.5,7.1,7.9,7.8,8.5,8.4,8,8.2,9.1,9.6,9.1,6.9,7.1,7.5,8.2,7,8.1,
   7.8,8.4,8.1,8.2,8.6,9.7,7,7.2,7,7.3,7,7.5,7.8,8.6,8,8.7,8.7,9.1,7.4,7.2,7.3,7,
   9]

data = [srt, rt,srt_c, rt_c]
# multiple box plots on one figure
figure()
boxplot(data)

media = [np.mean(x) for x in data]


xticks([1,2,3,4], ['SRT', 'HRT', 'SRT_CC', 'HRT_CC'])
scatter([1, 2, 3, 4], media)

axis([0, 5, 0, 50])

grid()
title(u'Comparação da distribuição conjunto de dados obtidos para 2ms')
xlabel(u'Distribuição')
ylabel('Valores em us')

savefig('/home/ivo/Desktop/py/imagens_graficos/grafico_de_amostras/500hzBox.png')

show()
```

Histórico

- 26 de Outubro de 2011, Versão Entrega, mailto:1960576@isep.ipp.pt
- 14 de Novembro de 2011, Versão Final, mailto:1960576@isep.ipp.pt

$Id:MEEC - Sistema de Aquisição de Dados em Tempo Real.docx versão Final Date:14-11-2011$