Universidade do Minho
Escola de Engenharia

Bohdan Arzhanov

Kernel de Tempo Real assistido por Hardware

Bohdan Arzhanov Kernel de Tempo Real assistido por Hardwar

UMinho | 2013

outubro de 2013

Universidade do Minho
Escola de Engenharia

Bohdan Arzhanov

Kernel de Tempo Real assistido por Hardware

Tese de Mestrado
Ciclo de Estudos Integrados Conducentes ao Grau de
Mestre em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do
Professor Doutor Jorge Miguel Nunes Santos Cabral

outubro de 2013

DECLARAÇÃO

Nome: Bohdan Arzhanov.

Endereço eletrónico: a55694@alunos.uminho.pt

Tel./Tlm.: 919864914.

Número do Bilhete de Identidade: 31186726.

Título da Dissertação: Kernel de Tempo Real assistido por Hardware.

Ano de conclusão: 2013.

Orientador: Professor Doutor Jorge Miguel Nunes Santos Cabral.

Designação do Mestrado:

Ciclo de Estudos Integrados Conducentes ao Grau de Mestre em Engenharia Eletrónica Industrial e de Computadores.

Área de Especialização: Sistemas Embebidos.

Escola de Engenharia da Universidade de Minho.

Departamento de Eletrónica Industrial.

Universidade do Minho, ___/___/_____

Assinatura: _____

Agradecimentos

Gostaria de endereçar especial agradecimento às minhas avós, que me educaram e formaram durante os anos mais turbulentos da minha juventude. Foram elas que estavam ao meu lado durante a minha formação escolar na Ucrânia, sem elas esta dissertação certamente não existia: "Спасибо вам большое, мои дорогие бабушки!"

Aos meus pais, Serhiy Arzhanov e Oksana Arzhanova, pela amizade, compreensão e apoio, bem como pela sua coragem e determinação.

Ao meu orientador, Professor Doutor Jorge Cabral, pela excelente orientação e todo apoio no âmbito desta dissertação e além.

Um especial obrigado ao Professor Doutor Adriano Tavares, que teve o papel principal na minha formação como sendo "engenheiro do futuro", pelo seu profissionalismo como sendo professor e investigador e pela sua disponibilidade incondicional.

Ao grupo de investigação *Embedded Systems Research Group* do Departamento de Eletrónica Industrial da Universidade do Minho, que me proporcionou todas as condições necessárias para a conclusão desta dissertação.

Um especial obrigado ao Mestre Vitor Silva, que me servirá de exemplo de um engenheiro profissional, pela toda ajuda, partilha de conhecimento e amizade.

Ao meu companheiro, Pedro Oliveira, pela amizade, discussão e espírito da equipa, bem como aos meus amigos e colegas, que me acompanharam ao longo destes anos, em especial ao André Pereira, Domingos Martins, Simão Almeida, Hélder Vilas Boas, Fábio Cunha, Bruno Silva e Marco Azevedo.

À minha noiva, Ielyzaveta Trunova, pela paciência e compreensão.

Por fim, gostaria de endereçar um especial agradecimento ao meu grande amigo Carlos Sousa, cujo apoio e amizade eram cruciais para o meu sucesso.

"A todos, um muito obrigado!"

Abstract

Nowadays we live in the era when a large number of computer systems are making part of our life. Often, we don't realize the presence of these systems, due to its invisibility for the user through the integration into a larger system. These systems are called embedded and developed to perform a reduced set of operations, defined by a specific application which they aimed for. Often, the use of the operating system is crucial for viability of these computer systems. There are embedded systems that have real-time requirements, which mean that some of their activities may have a time restriction, existing deadline associated with its completion. It's from this context that real-time operating systems arise, with an enhanced focus on efficiency of scheduling algorithms, in order to achieve deterministic behavior. Determinism, as one of design metrics for computer system development, causes a constant search for improvements on reduction of computational overhead, consolidation of repeatability, cancellation of jitter and reduction of latency in the response to events. In this context, this thesis implements the migration of most features of real-time kernel to dedicated hardware, with the purpose of its potentiation by taking an advantage of hardware's deterministic nature and parallelism that it provides. Besides having a positive effect on reduction of the overhead related to execution of kernel services, the migration also has positive effects on the cancellation of jitter and reduction of latencies. An unification of tasks' and interruptions' priority spaces was also implemented. Also was possible to illustrate practical aspects such as: porting of an embedded operating system, implementation of common features of real-time kernel as software and as hardware, implementation of SoC (System-on-a-Chip) and custom peripherals by use of integrated development environment comprised for embedded systems based on FPGA devices manufactured by Xilinx company.

Resumo

Vivemos na era em que um elevado número de sistemas computacionais fazem parte do nosso dia-a-dia. Muitas são as vezes, quando nós nem apercebemos da presença destes sistemas, devido a sua invisibilidade para utilizador, através da integração num sistema maior. Estes sistemas são designados de sistemas embebidos e são desenhados para realizar um conjunto de operações reduzido, enquadrado numa aplicação específica que visam desempenhar. Muitas das vezes, a utilização do sistema operativo é crucial para viabilidade destes sistemas computacionais. Existem sistemas embebidos que apresentam requisitos de tempo real, o que significa, que algumas das suas atividades podem ter uma restrição temporal, existindo *deadline* associado a sua conclusão. É neste contexto que surgem os sistemas operativos de tempo real, com o foco realçado na eficiência dos algoritmos de escalonamento, com o intuito de alcançar um comportamento de natureza determinística. Determinismo, como a métrica do desenvolvimento de sistemas informáticos, provoca uma busca incessante por melhorias na redução do *overhead* computacional, consolidação da repetibilidade, cancelamento do *jitter* e redução das latências na resposta aos eventos. Neste âmbito, a presente dissertação implementa a migração da maioria dos recursos de um *kernel* de tempo real para *hardware* dedicado, com intuito de o potenciar, tirando proveito da natureza determinística do *hardware* e do paralelismo que este proporciona. Para além de ter um efeito positivo na redução do *overhead* da execução das tarefas do *kernel*, a referida migração também o tem no cancelamento do *jitter* e na redução das latências. Foi também implementada a unificação do espaço de prioridades das tarefas e interrupções. Foi possível exemplificar aspetos práticos como: *porting* do sistema operativo embebido, implementação dos recursos comuns de um *kernel* de tempo real em *software* e em *hardware*, caracterização de um sistema operativo embebido, implementação do SoC (*System-on-a-Chip*) e dos periféricos customizados no ambiente de *software* automatizado de projeto de sistemas embebidos baseados no dispositivo FPGA do fabricante Xilinx.

Conteúdo

1 Introdução **1**

1.1 Contextualização 1

1.2 Motivação e Objetivos 3

1.3 Contribuições 4

1.4 Organização da Dissertação 5

2 Estado da Arte **7**

2.1 Sistemas Embebidos 7

2.2 Sistemas Operativos e Tempo Real 9

2.3 Unificação de espaço de prioridades 15

2.4 Tecnologia FPGA 18

2.5 Linguagens de programação HDL 21

2.5.1 VHDL e Verilog 22

2.5.2 SystemVerilog 25

2.6 Diagramas SMChart 25

2.7 Conclusões 29

3 Ambiente de Desenvolvimento **31**

3.1 Plataforma de Desenvolvimento XUPV2 32

3.2 Xilinx *Embedded Development Kit* 10.1 33

3.2.1 Ciclo de Desenvolvimento 34

3.2.2 XPS *Xilinx Platform Studio* 34

3.3 Conclusões 43

4 Sistema Operativo de Suporte **45**

4.1 ADEOS 45

4.1.1 Visão global sobre organização 46

4.1.2 Tarefas 47

4.1.3 Listas de Tarefas 51

4.1.4 Escalonador e Pontos de Escalonamento 52
4.1.5 *Mutex* 55
4.2 FreeRTOS 58
4.2.1 Visão global sobre organização 59
4.2.2 Tarefas e a sua gestão 62
4.2.3 Listas de Tarefas 63
4.2.4 Escalonamento 65
4.2.5 Pontos de escalonamento e Comutação de contexto 66
4.3 Implementação da vertente *software* do sistema operativo híbrido . 71
4.3.1 *Porting* 71
4.3.2 *Upgrade* 82
4.3.3 Interface 89
4.4 Conclusões 91

5 *Hardware MicroKernel* 93
5.1 Requisitos 93
5.2 Visão global e configuração 94
5.3 Registos virtuais 97
5.4 Projeto genérico 98
5.5 Implementação 105
5.5.1 Escalonador 105
5.5.2 Temporizadores 114
5.5.3 Controlador das Interrupções 115

6 Resultados Experimentais 119
6.1 Ferramentas de medição 119
6.1.1 Contador dedicado PLB e registo TBL do PPC405 119
6.1.2 Contadores *ad hoc* dentro do periférico testado 121
6.1.3 Ferramenta *ChipScope Pro* 123
6.2 Testes 128
6.2.1 Validação das funcionalidades do *kernel* e testes de regressão 128
6.2.2 Tempos de Execução 131
6.2.3 Caracterização temporal da preempção baseada na interrupção 132
6.2.4 Versão *software* versus Versão híbrida 136

7 Discussão e Conclusões 139
7.1 Trabalho Desenvolvido 139

7.2 Trabalho Futuro 141

Bibliografia **142**

Lista de Figuras

2.1 Exemplos de sistemas emebebidos e domínios da sua aplicação . . . 8
2.2 Interseção dos sistemas embebidos com os sistemas de tempo real . 10
2.3 Componentes que constituem um diagrama SMChart 26
2.4 Entidade base do diagrama SMChart 27
2.5 Exemplo de um diagrama SMChart com a nomenclatura estendida . 27

3.1 Diagrama de blocos da plataforma do desenvolvimento XUPV2 . . . 32
3.2 Ciclo do desenvolvimento do *Embedded Development Kit*, adaptado de (Florida Internation University) 34
3.3 Ciclo do desenvolvimento do *Embedded Development Kit*, adaptado de (Florida Internation University) 35
3.4 Janela da configuração do sistema baseado no: (a) - processador *hard core* PPC, (b) processador *soft core* MicroBlaze 36
3.5 Organização das interfaces da memória do processador PPC405 . . 38
3.6 Representação *Bus Interfaces* do *System Assembly View* de um projeto XPS baseado no *hardware microkernel* concebido no âmbito desta dissertação 42
3.7 Representação *Ports* do *System Assembly View* de um projeto XPS baseado no *hardware microkernel* concebido no âmbito desta dissertação 42
3.8 Representação *Adresses* do *System Assembly View* de um projeto XPS baseado no *hardware microkernel* concebido no âmbito desta dissertação 43

4.1 Diagrama de classes do sistema operativo ADEOS 46
4.2 Transições do estado da tarefa no sistema operativo ADEOS 48
4.3 Lista ligada das tarefas prontas para execução 52
4.4 Transições do estado da tarefa no sistema operativo FreeRTOS . . . 62

4.5 Representação esquemática da lista *multi-FIFO* das tarefas prontas para execução 65
4.6 Estado da pilha após primeiras duas instruções do *contextInit()* . . 75
4.7 Estado da pilha da tarefa antes da execução da primeira instrução da função *run()* 77
4.8 O estado da pilha após primeiras duas instruções do *contextSwitch()* 79
4.9 Ilustração do mecanismo da comutação de contexto, no exemplo de uma tarefa A que é preemptida pela tarefa B que executa pela primeira vez e que, de seguida, é preemptida pela tarefa A que, desta forma, volta a executar 81

5.1 Arquitetura do *hardware microkernel* implementado 95
5.2 Diagrama SMChart genérico do mecanismo da interação com IPIC . 99
5.3 A composição do endereço do registo virtual 101
5.4 Componente escalonador do *hardware microkernel* 105
5.5 Ilustração do conceito utilizado no projeto do escalonador 106
5.6 Módulo do topo correspondente a um nó de comparação da árvore binária 107
5.7 Exemplificação do comportamento *round-robin* 109
5.8 Instanciação recursiva da árvore binária 110
5.9 Diagrama SMChart que descreve a interface do escalonador 113
5.10 Componente *Timers* do *hardware microkernel* 114
5.11 Diagrama SMChart que descreve a interface do componente *Timers* do *hardware microkernel* 114
5.12 Pseudo-código que define o estado dos *bits* do vetor `Timer_is_set` . 115
5.13 Componente SIC (*synchronous interrupt controller*) do *hardware microkernel* 115
5.14 O diagrama SMChart que descreve a interface do controlador das interrupções 116

6.1 Organização do sistema de depuração ChipScope Pro 123
6.2 Aspeto das janelas de visualização do *ChipScope Pro Analyzer* . . . 126
6.3 Comportamento esperado para o ensaio da funcionalidade *round-robin* 129
6.4 Comportamento esperado para o ensaio da funcionalidade de temporização 129
6.5 Comportamento esperado para o ensaio da funcionalidade exclusão mútua 130
6.6 Caracterização temporal da preempção de uma tarefa 134

Lista de Tabelas

2.1 Comparação dos objetivos do projeto de um sistema RTOS e um sistema GPOS 11

3.1 Capacidade do dispositivo XC2VP30 33
3.2 Opções da configuração do sistema baseado no PPC ou MicroBlaze 37

4.1 Registos da arquitetura 80x86 73

5.1 Parâmetros da configuração do *hardware microkernel* 96
5.2 Lista dos nomes *alias* aplicados no diagrama da Figura 5.2 99
5.3 Tabela de verdade da decisão do nó de comparação 108
5.4 Tabela de verdade para o caso da empate das prioridades 108
5.5 Os comandos 112

6.1 Tempos de execução em número de ciclos de relógio de certas tarefas do *kernel* híbrido 132
6.2 Resultados da comparação da versão híbrida com a versão em *software* 137

Lista de Listagens

2.1 Implementação genérica na linguagem VHDL baseada no SMChart da Figura 2.5 . . . 28
3.1 Exemplo da instanciação e configuração de um componente no ficheiro .mhs . . . 39
3.2 Exemplo da definição dos *drivers* para o processador e um periférico no ficheiro .mss . . . 39
3.3 Exemplo do mapeamento das ligações externas aos pinos I/O do dispositivo FPGA e da associação das respetivas restrições lógicas (ficheiro .ucf) . . . 40
4.1 Definição da classe *Task* . . . 47
4.2 Construtor da classe *Task* . . . 50
4.3 Função especial *run()* . . . 51
4.4 Definição da classe *Sched* . . . 53
4.5 Parte do ficheiro *sched.cpp* . . . 54
4.6 schedule(), método chave da classe *Sched* . . . 55
4.7 Definição da classe *Mutex* . . . 56
4.8 Implementação do método take() . . . 56
4.9 Implementação do método release() . . . 58
4.10 Ficheiro *FreeRTOSConfig.h* . . . 60
4.11 Listas de tarefas manuseadas pelo escalonador do FreeRTOS . . . 64
4.12 Assinatura dos *handlers* das exceções/interrupções no *porting* para processador PPC405 . . . 67
4.13 Definição da macro *taskYIELD()* que emite exceção *system call* . . . 68
4.14 Imlementação da rotina *vPortYield()* e *vPortTickISR()* . . . 68
4.15 Definição da macro *portSAVE_STACK_POINTER_AND_LR* . . . 69
4.16 Definição da macro *portRESTORE_STACK_POINTER_AND_LR* . . . 70
4.17 Versão original do ficheiro *bsp.h* . . . 72
4.18 Prototipo da função *contextInit()* . . . 74

4.19 Primeira metade da implementação da função *contextInit()* 74
4.20 Definição da estrutura *Context* 76
4.21 Segunda metade da implementação da função *contextInit()* 76
4.22 Protótipo da função especial *run()* 77
4.23 Protótipo da função *contextSwitch()* 78
4.24 Primeira metade da implementação da função *contextSwitch()* . . . 78
4.25 Segunda metade da implementação da função *contextSwitch()* . . . 80
4.26 Principais parâmetros da configuração do *kernel* concebido 84
4.27 Declaração da class Hw_Timer 86
4.28 Implementação da classe *Hw_Timer* 87
4.29 Macros para comunicação com o periférico geradas automaticamente 87
4.30 Protótipo do método *CreateTask()* da classe *Task* 88
4.31 Parte do código contido no ficheiro *hw_kernel_io.h* 89
4.32 Declaração da classe *HwKernel* 90
4.33 Construtor da classe *HwKernel* 91
5.1 Primeira metade da implementação interna da entidade *user_logic* . 102
5.2 Segunda metade da implementação interna da entidade *user_logic* . 103
5.3 Função VHDL set_records_field 104
6.1 Definição das macros para manipulação do registo TBL 120
6.2 Exemplo da utilização do registo *time base* na medição do tempo de execução 121
6.3 *Template* para implementação de contadores *ad hoc* dentro de um periférico 122
6.4 Exemplo da instanciação do componente ILA na linguagem VHDL . 127
6.5 Implementação do ensaio da funcionalidade *round-robin* 129
6.6 Implementação do ensaio da funcionalidade de temporização 130
6.7 Implementação do ensaio da funcionalidade exclusão mútua 131
6.8 Implementação de medição de tempos de execução das tarefas do *kernel* 132
6.9 Implementação do ensaio da caracterização temporal da preempção de uma tarefa 134

Capítulo 1

Introdução

Neste capítulo é contextualizado o âmbito desta dissertação, são definidos a motivação e objetivos, bem como apresentadas as contribuições do trabalho desenvolvido. No fim do capítulo é apresentada a organização da dissertação.

1.1 Contextualização

Vivemos na era em que um elevado número de sistemas computacionais fazem parte do nosso dia-a-dia. Muitas são as vezes, quando nós nem apercebemos da presença destes sistemas, devido a sua invisibilidade para utilizador através da integração num sistema maior. Estes sistemas computacionais são designados de sistemas embebidos e podem ser encontrados praticamente em todas as áreas e domínios da nossa vida atual. Ao contrário dos computadores pessoais, muitas vezes designados de sistemas computacionais de uso geral, um sistema embebido é desenhado para realizar um conjunto de operações reduzido, enquadrado numa aplicação específica que visa desempenhar. Com a diversidade das aplicações, varia também a complexidade dos sistemas embebidos, existindo muitos sistemas não triviais. Estes sistemas são muitas vezes encarregados com a execução de diversas atividades em simultâneo, bem como, dotados com elevado número de periféricos que proporcionam a interação com o mundo exterior. Neste contexto, a utilização do sistema operativo torna-se crucial para viabilidade dos sistemas deste tipo.

O componente central do sistema operativo é frequentemente denominado como núcleo ou *kernel* e normalmente consiste num pequeno e altamente otimizado conjunto de bibliotecas.

Existem sistemas embebidos que apresentam requisitos de tempo real, ou seja algumas das suas atividades podem ter uma restrição temporal, existindo *deadline* associado a sua conclusão. O incumprimento do dito *deadline* nestes sistemas é uma falha que pode comprometer a consistência do sistema ou a qualidade do serviço por este prestado.

Os sistemas de tempo real são dotados com sistemas operativos de tempo real, com o foco realçado na eficiência dos algoritmos de escalonamento, com o intuito de alcançar um comportamento de natureza determinística. A característica vital do sistema operativo de tempo real é a sua capacidade de resposta aos eventos internos e externos. Uma das métricas desta capacidade é a latência, o que significa: o tempo entre a ocorrência do evento e a execução da primeira instrução do código associado a dita ocorrência. Por exemplo, caso se trata de um sinal de pedido da interrupção (IRQ), pelo código associado a ocorrência do IRQ entende-se a rotina de serviço a interrupção (ISR), e a latência, neste caso, é denominada latência de interrupção. A outra métrica é o *jitter*, o que significa: a variação no período de um evento que, no entanto, é entendido como sendo um evento periódico de período constante.

Determinismo, como a métrica do desenvolvimento de sistemas informáticos, provoca uma busca incessante por melhorias na redução do *overhead* computacional, repetibilidade, cancelamento do *jitter* e redução das latências na resposta aos eventos. Várias são as soluções desenvolvidas, sendo que, cada uma afeta de um modo mais ou menos significativo o comportamento do sistema.

Um único processador não pode literalmente executar diversas tarefas em simultâneo, mas o sistema operativo cria a ilusão da execução simultânea de varias tarefas, gerindo o acesso destas ao CPU. Por sua vez, as rotinas que implementam os mecanismos desta gestão do acesso (mecanismos de escalonamento) também são executadas no mesmo CPU, introduzindo *overhead* computacional que, sendo elevado, pode comprometer as características temporais do sistema. Uma possível solução, que endereça a redução do dito *overhead*, consiste na migração de certas funcionalidades do *kernel* do sistema operativo para um coprocessador que irá executar verdadeiramente em paralelo com o CPU, cedendo às tarefas do utilizador o respetivo tempo do acesso poupado. Para além de ter um efeito positivo na redução do *overhead*, a referida migração para o *hardware* dedicado das funcionalidades cruciais do *kernel* do sistema operativo, também o pode ter no cancelamento do *jitter* e na redução das latências, devido a natureza determinística do *hardware*. Uma visão mais abrangente sobre sistemas operativos de tempo real, mecanismos do es-

calonamento e sincronização, bem como, desafios e soluções, associados a busca do determinismo nos atuais sistemas embebidos, será apresentada no Capítulo 2.

A tecnologia FPGA permanece em constante desenvolvimento, com especial relevância, no âmbito da sua utilização nos sistemas embebidos. É de alta importância referir que na última década as empresas desenvolvedoras da tecnologia apostaram na integração da área reprogramável FPGA junto com os microprocessadores embebidos e respetivos periféricos. Resultado da dita integração são dispositivos SoPC (*System-on-a-Programmable-Chip*). Estes dispositivos tornam-se cada vez mais apelativos para serem usados nos atuais sistemas embebidos de tempo real (Hall and Hamblen, 2006). As razões da adotação da tecnologia FPGA num crescente número de aplicações, bem como uma visão mais detalhada sobre questões associadas ao projeto de uma solução baseada nesta tecnologia serão apresentados no Capítulo 2.

1.2 Motivação e Objetivos

Como já foi referido na secção anterior, o alcance do comportamento de natureza determinística, nos sistemas operativos de tempo real, tem uma alta relevância no âmbito da sua utilização nos sistemas embebidos. A migração de certas funcionalidades para *hardware* dedicado poderá potenciar o *kernel* do sistema operativo, devido a natureza determinística do *hardware* e exploração do paralelismo que este proporciona. Ao mesmo tempo, muitas plataformas usadas atualmente nos sistemas embebidos são dotados com *hardware* que integra a área de silício reprogramável (FPGA) junto com microprocessadores embebidos e respetivos periféricos num só dispositivo. Nos sistemas com esta arquitetura é possível a integração de coprocessadores dedicados concebidos pelo desenvolvedor, que irão implementar as principais funcionalidades de um *kernel* de tempo real, possibilitando a sua execução determinística e em paralelo com a execução das tarefas do utilizador no CPU principal.

Nesta dissertação, pretende-se desenvolver um *kernel* de tempo real assistido por *hardware*. Trata-se um *kernel* simplista baseado no conceito de *microkernel*, com modelação através de componentes, desenvolvidos sobre os paradigmas das linguagens de programação C/C++. Este *kernel* deve implementar um conjunto mínimo dos recursos que compreendem um sistema operativo embebido do tempo real. A maioria destes recursos deve ser implementada também na forma de um conjunto

de coprocessadores dedicados (*hardware microkernel*). Por fim, o impacto da migração dos recursos do *kernel* para o *hardware* dedicado deve ser avaliado.

O sistema resultante será composto por entidades de naturezas distintas: *hardware* e *software*, neste contexto o sistema pode ser referido como o *kernel* híbrido ou sistema operativo híbrido.

Será ainda considerado o *refactoring* e integração de um sistema operativo existente, livre e adotado pela indústria de *software*, com o objetivo de potenciar a sua utilização e de servir como veículo de transporte do modelo no meio científico.

Sistemas operativos de tempo real e sistemas embebidos dotados com dispositivo FPGA são duas áreas que se apresentam como uma fonte incessante de aprendizagem e que são posicionadas pelo autor como sendo as suas áreas de interesse no âmbito de sistemas informáticos. O casamento destas duas vertentes num trabalho só requer domínio de tecnologias envolvidas e origina desafios motivadores.

1.3 Contribuições

No âmbito desta dissertação, foi implementado um *kernel* de tempo real assistido por *hardware* e o impacto da migração dos seus recursos para o *hardware* dedicado foi avaliado.

Ao longo deste documento são exploradas questões relacionadas com sistemas operativos de tempo real híbridos. Foi possível exemplificar aspetos como: *porting* do sistema operativo embebido, objetos da abstração que o sistema operativo implementa e a sua migração para *hardware* dedicado, unificação do espaço de prioridades das tarefas e interrupções, caracterização de um sistema operativo embebido.

O presente documento também é dotado com uma visão prática sobre a implementação do SoC (*System-on-a-Chip*), bem como dos periféricos customizados, no ambiente de *software* automatizado de projeto de sistemas embebidos baseados no dispositivo FPGA do fabricante Xilinx.

1.4 Organização da Dissertação

Este documento apresenta a implementação de um *kernel* de tempo real assistido por *hardware*.

Após uma breve contextualização e apresentação de motivação, objetivos e contribuições feitas no Capítulo 1, segue o Capítulo 2 que apresenta uma visão geral sobre as tecnologias e conceitos abordados nesta dissertação.

O Capítulo 3 apresenta a plataforma do desenvolvimento utilizada, bem como o ambiente de desenvolvimento em que esta se encontra integrada.

A descrição da implementação do *kernel* híbrido é feita em duas partes, sendo que o Capítulo 4 apresenta a implementação da vertente *software*, enquanto o Capítulo 5 foca na vertente *hardware*.

Os resultados experimentais, bem como as ferramentas de medição utilizadas na sua obtenção, são apresentados no Capítulo 6

O documento termina com o Capítulo 7 no qual é feita a discussão de resultados e conclusões acerca do trabalho desenvolvido. É também feita uma alusão a trabalho futuro sobre o tema.

Capítulo 2

Estado da Arte

Este capítulo apresenta uma visão geral sobre as tecnologias e conceitos abordados nesta dissertação.

Sendo o objeto de estudo desta dissertação os sistemas operativos de tempo real no âmbito da sua utilização nos sistemas embebidos dotados com dispositivo FPGA, torna-se importante a introdução e consolidação dos conceitos como: sistemas embebidos, sistemas operativos, tempo real, unificação de espaço de prioridades das tarefas e interrupções, tecnologia FPGA e metodologias do desenvolvimento *hardware*.

2.1 Sistemas Embebidos

Segundo a definição que pode ser encontrada no glossário dos sistemas embebidos (Barr, 2007), um sistema embebido é uma combinação do *software* e *hardware* computacional e, possivelmente, das partes adicionais mecânicas ou outras, desenhada para desempenhar uma função específica. Por norma, um sistema embebido faz parte de um sistema maior e não é visível para utilizador. Um exemplo clássico desta característica de sistemas embebidos é o sistema ABS (*antilock braking system*) a fazer parte de um veículo.

O mundo dos sistemas embebidos é muito diverso. Estes variam no seu tamanho, formato, complexidade, ambiente para qual foram concebidos, etc. Sistemas embebidos estão presentes em praticamente todas as áreas e domínios da nossa vida atual, desde uma simples calculadora do bolso, até a exploração espacial. A Figura 2.1 visa exemplificar a diversidade do mundo dos sistemas embebidos.

Figura 2.1: Exemplos de sistemas emebebidos e domínios da sua aplicação

Ao contrário dos computadores pessoais, muitas vezes designados de sistemas computacionais de uso geral, um sistema embebido é desenhado para realizar um conjunto de operações reduzido, enquadrado numa aplicação específica que visa desempenhar. Com a diversidade das aplicações, varia também a complexidade dos sistemas embebidos, existindo muitos sistemas não triviais. Estes sistemas são muitas vezes encarregados com a execução de diversas atividades em simultâneo, bem como, dotados com elevado número de periféricos que proporcionam a interação com o mundo exterior. Neste contexto, a utilização do sistema operativo torna-se crucial para viabilidade dos sistemas deste tipo.

Uma das características comuns de sistemas embebidos é a sua interação com o mundo exterior. O mundo real tem o comportamento de tempo real. Daí existem sistemas embebidos que apresentam requisitos de tempo real, ou seja algumas das suas atividades podem ter uma restrição temporal, existindo *deadline* associado à sua conclusão. O incumprimento do dito *deadline* nestes sistemas é uma falha que pode comprometer a consistência do sistema ou a qualidade do serviço por este prestado.

O processo de conceção de um sistema embebido de tempo real é uma área da engenharia que exige conhecimentos de diferentes campos tecnológicos como por exemplo: dispositivos *hardware*, sistemas operativos de tempo real, ferramentas *compiler/linker/loader*, técnicas de depuração e uso de *debugger*, desenvolvimento de aplicações *software*.

De seguida são apresentadas as características habituais dos sistemas embebidos:

- São orientados à aplicação. Durante o projeto, os recursos a serem utilizados

são definidos em função dos requisitos da aplicação endereçada. O sistema executa repetidamente uma aplicação *software* preinstalada pelo fabricante.

- Devem ser eficazes existindo varias métricas, muitas das vezes, contraditórias como: baixo consumo energético, baixo custo, pequeno tamanho e peso, pequeno consumo da memória, etc.
- Muitas vezes, devem promover a segurança dos dados pessoais e implementar meios de autenticação.
- Devem ser confiáveis. Existem os ambientes e aplicações nos quais a falha do sistema pode ter consequências catastróficas. Nestas circunstâncias, os sistemas devem endereçar a segurança através da garantia da disponibilidade da sua resposta e da confiança da sua execução e através da minimização da probabilidade do alcance do estado irrecuperável, mesmo em caso da falha.
- São usados nas aplicações autónomas. Na maioria dos casos os sistemas embebidos executam sem qualquer intervenção humana.
- Possuem uma interface dedicada com utilizador. Podendo existir ou não uma interface gráfica.
- Tipicamente ligados ao mundo exterior através dos sensores e atuadores.
- Implicam o desenvolvimento *cross-platform* na fase do seu projeto.
- Muitas das vezes são alimentados através de uso das baterias e consideram o baixo consumo energético no seu projeto, podendo também existir os meios do *harvesting* da energia associados.

2.2 Sistemas Operativos e Tempo Real

O objetivo de existência de um sistema operativo consiste na eficiente gestão dos recursos disponíveis no sistema, proporcionando um conjunto de serviços às aplicações no espaço utilizador, escalonando as tarefas no acesso ao recurso CPU, permitindo uma execução segura do *software*, virtualizando o acesso à memória e acrescentando uma camada de abstração do *hardware* subjacente (Silberschatz et al., 2012).

Uma das principais funções do sistema operativo consiste em proporcionar um ambiente de execução multitarefa. Ou seja, o sistema operativo introduz uma

abstração que é fluxo de execução independente, frequentemente denominado como tarefa, proporciona serviços de criação e utilização destas entidades abstratas, facilita, desta forma, a manutenção dos programas dos sistemas complexos, torna o projeto mais escalável e possibilita a execução paralela ou pseudo-paralela destas entidades (tarefas).

Para que uma tarefa faça o seu trabalho, esta deve executar. Num cenário em que o número de tarefas é maior do que o número de processadores, vão existir tarefas no estado da espera pela execução. A decisão na escolha das tarefas para execução recai sobre o escalonador do sistema operativo. Este deve dividir o tempo finito da execução do processador entre várias tarefas. As decisões do escalonador devem ir de encontro com a melhor utilização dos recursos do sistema, ao mesmo tempo com a satisfação dos requisitos temporais do sistema. É importante ter em mente que para efetuar o seu trabalho, o próprio sistema operativo também necessita do tempo de acesso ao CPU.

O núcleo do sistema operativo é denominado *kernel* e normalmente consiste num pequeno e altamente otimizado conjunto de bibliotecas.

Nem todos os sistemas embebidos apresentam requisitos de tempo real e nem todos os sistemas de tempo real são sistemas embebidos. O casamento desses dois domínios resulta no domínio dos sistemas embebidos de tempo real (Figura 2.2). Quando se fala num sistema de tempo real, ou seja, num sistema que tem restrições temporais sobre a conclusão das certas suas atividades, está praticamente implícito o ambiente de execução multitarefa que, por sua vez, requer a existência do sistema operativo. Os sistemas operativos empregues nestas circunstâncias são denominados sistemas operativos de tempo real ou RTOS (*real-time operating system*).

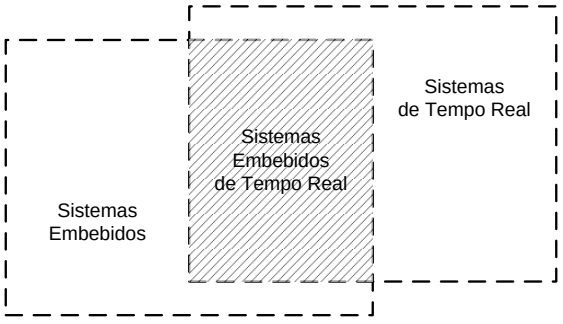


Figura 2.2: Interseção dos sistemas embebidos com os sistemas de tempo real

Os sistemas de tempo real são classificados de acordo com a gravidade das consequências do incumprimento dos requisitos temporais. Nos sistemas *"hard real-*

time" o incumprimento do *deadline* é uma falha crítica do sistema que tem consequências graves e às vezes irrecuperáveis. Nos sistemas *"soft real-time"* o incumprimento do *deadline* não tem consequências graves, mas a utilidade dos resultados computacionais diminui após *deadline* piorando a qualidade dos serviços prestados. Por sua vez, nos sistemas *"firm real-time"* o resultado computacional fora do tempo é inútil, no entanto, o incumprimento não frequente do *deadline* é tolerável mas faz degradar a qualidade dos serviços prestados. Muitas das vezes, a classificação limita-se a duas classes: *"hard real-time"* e *"soft real-time"*.

Os exemplos dos sistemas *"hard real-time"* são: o sistema ABS (*antilock braking system*) num veículo, *pacemaker* artificial, sistema do controlo do avião, etc. Uma falha de qual quer um desses sistemas pode ter consequências irreversíveis. Os exemplos dos sistemas *"soft real-time"* são: serviços VoIP (*Voice over Internet Protocol*), sistema de som, etc.

Os sistemas operativos de tempo real são os sistemas operativos que suportam a execução das aplicações de tempo real proporcionando respostas logicamente corretas nos tempos predefinidos. A estrutura destes sistemas é em tudo semelhante a dos sistemas operativos regulares, no entanto o objetivo principal endereçado aquando projeto é o determinismo da execução das suas tarefas em vez de maximização do desempenho global. Ao contrário dos sistemas operativos regulares, cujas otimizações endereçam o caso médio, os sistemas operativos de tempo real procuram distinguir e otimizar o pior caso, pois este é que define a conformidade com uma determinada categoria do tempo real. A Tabela 2.1 compara os objetivos do projeto de um sistema operativo de tempo real (RTOS) e um sistema operativo de propósito geral (GPOS).

Tabela 2.1: Comparação dos objetivos do projeto de um sistema RTOS e um sistema GPOS

RTOS	GPOS
Otimiza pior caso	Otimiza caso médio
Minimiza latência	Maximiza rendimento
Escalonamento previsível	Escalonamento eficiente
Simples executável	Amplo conjunto dos serviços

A característica vital do sistema operativo de tempo real é a sua capacidade de resposta aos eventos internos e externos. Uma das métricas desta capacidade é a latência, o que significa: o tempo entre a ocorrência do evento e a execução da primeira instrução do código associado a dita ocorrência. Por exemplo, caso se trata de um sinal de pedido da interrupção (IRQ), pelo código associado a

ocorrência do IRQ entende-se a rotina de serviço a interrupção (ISR), e a latência, neste caso, é denominada latência de interrupção. A outra métrica é o *jitter*, o que significa: a variação no período de um evento que, no entanto, é entendido como sendo um evento periódico de período constante.

Os desenvolvedores dos sistemas operativos de tempo real procuram ter o *kernel* do sistema desenhado de tal forma, que a execução das tarefas deste tivesse tempo constante, tornando o sistema mais previsível. A maior fonte do *jitter* nos sistemas operativos é o mecanismo de tratamento das interrupções. Os problemas surgem quando o *kernel* do sistema operativo não é preemptivel, ou seja, não pode ser interrompido durante a execução das suas tarefas. Se uma interrupção ocorrer durante a execução do *kernel*, não será atendida dentro do tempo correspondente à latência normal do sistema, pois será atendida só no fim da execução do *kernel* quando as interrupções voltam a ser habilitadas. Desta forma, outro objetivo do projeto do sistema operativo é a minimização do tamanho das secções críticas, que são partes do código executadas com as interrupções desabilitadas, visto que a interrupção durante esta secção do código poderá resultar na inconsistência do sistema.

Dito isto, as características comuns dos sistemas operativos de tempo real são:

- Pequeno tamanho;
- Funcionalidade multitarefa;
- Recursos de sincronização das tarefas;
- Baixa latência de interrupção;
- *Kernel* preemptivo;
- Minimização das secções críticas;
- Execução das suas tarefas no tempo constante;
- Comutação de contexto rápida.

O coração do sistema operativo é o seu escalonador. Esta entidade gere o acesso das tarefas ao CPU. A funcionalidade multitarefa no sistema pode ser preemptiva ou cooperativa. No cenário preemptivo, o escalonador é responsável pela decisão da promoção de uma determinada tarefa ao estado da execução e pela consequente preempção da tarefa em execução no momento. Ou seja, a interrupção involuntária da execução de uma determinada tarefa chama-se preempção da tarefa. Por sua

vez, no cenário cooperativo, a execução de uma tarefa não é interrompida até que esta toma uma decisão voluntaria de ceder o acesso ao CPU. Tipicamente, esta ação é denominada como *yield*. Os sistemas operativos de tempo real proporcionam o ambiente de execução multitarefa preemptivo, podendo existir a opção da escolha do mecanismo cooperativo.

Para poder escalonar as tarefas, o escalonador manuseia estas e o estado destas. Tipicamente, uma tarefa encontra-se num dos três estados: A executar, pronta para executar ou bloqueada. Na maior parte das vezes, o sistema embebido possui apenas uma unidade de processamento, sendo assim, apenas uma tarefa se encontra no estado a executar. As tarefas prontas para execução estão guardadas numa estrutura de dados adequada aos objetivos do projeto do sistema operativo em causa, como por exemplo, lista ligada, lista duplamente ligada, estrutura multi-FIFO das listas ligadas (pode ser vista na secção 4.2.3), etc. Por norma, as tarefas podem ficar bloqueadas à espera de um evento de sincronização ou um evento temporal. Na maioria dos casos, cada recurso de sincronização manuseia uma lista das tarefas bloqueadas própria.

Um algoritmo de escalonamento é um conjunto de regras que determinam a tarefa a ser executada num determinado momento (Liu and Layland, 1973). No âmbito de sistemas operativos de tempo real serão discutidos os algoritmos preemptivos baseados na atribuição das prioridades as tarefas. Um algoritmo é chamado estático quando as prioridades das tarefas são assinadas uma vez e para sempre. O algoritmo de escalonamento estático muitas vezes também é chamado como o algoritmo *fixed-priority*. Um algoritmo é chamado dinâmico quando a prioridade de uma determinada tarefa pode ser alterada (pelo escalonador) de um ponto de escalonamento para outro.

O algoritmo de escalonamento mais comum, utilizado nos sistemas operativos de tempo real *low-end*, é o escalonamento estático preemptivo baseado nas prioridades. Trata-se do cenário quando as tarefas a serem executadas e as suas restrições temporais são conhecidas previamente e conforme as suas necessidades são atribuídas as prioridades a cada uma destas. O escalonador assegura que seja sempre executada a tarefa mais prioritária das prontas para execução. No que diz respeito ao caso da empate de prioridades, algumas sistemas operativos limitam o uso do espaço de prioridades sendo proibido assinar várias tarefas com a mesma prioridade, enquanto outras implementam uma estratégia do desempate. A estratégia do desempate habitual é denominada *round-robin* e limita-se a execução intercalada das tarefas com a mesma prioridade, o período com o qual é feita a preempção

e reescalonamento é denominado *time slice*. Um sistema baseado na assinatura estática de prioridades tem capacidade de atingir eficácia e cumprir requisitos de um cenário de utilização particular, no entanto, maior parte da responsabilidade pelo sucesso é delegada ao desenvolvedor da aplicação (ou seja utilizador do sistema operativo), sendo que o sucesso do escalonamento depende totalmente das suas decisões na atribuição das prioridades e implementação das tarefas. Alguns sistemas operativos deste tipo implementam junto ao escalonador um conjunto de funcionalidades que visam minimizar o impacto das decisões erradas do desenvolvedor ou proporcionar-lhe mais ferramentas de influência sobre o comportamento global do sistema. Alguns exemplos destas extensões do algoritmo são: uma tarefa pode ter também um *time slice* associado e é interrompida mesmo se continuar a ser a tarefa mais prioritária mas exceder o seu tempo de execução predefinido; as tarefas de menor prioridade, que não podem executar devido a execução da tarefa mais prioritária, recebem um aumento gradual da sua prioridade; etc.

Segundo (Liu and Layland, 1973), a especificação de um algoritmo de escalonamento baseado nas prioridades equivale à especificação do algoritmo de atribuição das prioridades às tarefas. No caso do escalonador estático, uma das estratégias que o desenvolvedor da aplicação pode seguir, na decisão da atribuição das prioridades às tarefas, é o algoritmo do escalonamento RMS (*rate-monotonic scheduling*). O algoritmo simplesmente impõe que as tarefas cujo período é menor (cuja frequência é mais elevada) devem ser assinadas com uma prioridade maior. Este algoritmo é compreendido no cenário em que as tarefas *real-time* são periódicas com intervalos constantes entre respetivas solicitações, o tempo da sua execução é invariante, as solicitações das diferentes tarefas são independentes e é considerado insucesso quando a tarefa não acabar a sua execução antes de ocorrer a próxima solicitação desta. Para o cenário descrito, existe uma comprovação matemática que, para que um conjunto de n tarefas *real-time* seja escalonavel, a utilização do processador resultante deve ser menor ou igual a $n(2^{1/n} - 1)$

Ao contrário do escalonamento estático baseado nas prioridades, um escalonador dinâmico toma a decisão de assinatura de uma determinada prioridade a uma determinada tarefa durante o tempo da execução. Um exemplo, de utilização comum nos sistemas operativos do tempo real, é o EDF (*earliest deadline first*).

2.3 Unificação de espaço de prioridades

Os sistemas embebidos comuns apresentam elevado nível de interação com o mundo exterior, possuindo diversos dispositivos de I/O, tais como, sensores, atuadores, discos, interfaces de comunicação, etc. Estes dispositivos participam nas diferentes operações, que, na maioria dos casos, apresentam requisitos de tempo real rigorosos. Os eventos físicos originados pelos dispositivos periféricos são direcionados para o sistema computacional através das interrupções.

Uma das principais tarefas do sistema operativo consiste na gestão eficaz do fluxo de execução, que deve ser conseguida na presença de tarefas executadas de forma síncrona e atendimento às interrupções, cuja ocorrência é assíncrona. Nos sistemas computacionais de uso geral, as tarefas não tem requisitos temporais rigorosos ao contrário das interrupções que, por norma, são associadas às operações restringidas temporalmente, exigindo eficácia e baixas latências. Sendo assim, nos sistemas operativos tradicionais, o escalonamento e a sincronização destes dois tipos de atividades (tarefas e interrupções) são abordados recorrendo aos mecanismos e as estratégias independentes e semanticamente e sintaticamente distintos. As tarefas, nestes sistemas, são geridas pelo *kernel* e escalonadas conforme a sua prioridade, enquanto as interrupções são geridas pelo *hardware* dedicado (controlador das interrupções) e escalonadas também conforme a sua prioridade. Tanto o escalonamento de tarefas como interrupções baseia-se nas prioridades, no entanto, devido à natureza distinta da implementação, trata-se de dois espaços de prioridades independentes, sendo que a precedência é sempre dada às interrupções. Esta arquitetura tradicional endereça de forma eficaz o requisito de baixas latências para eventos de I/O mas origina um conjunto de potenciais problemas que a tornam inadequada no contexto de sistemas de tempo real. Um dos potenciais problemas é a inversão das prioridades, que pode surgir no sistema de tempo real na presença de uma tarefa de alta prioridade que tem exigências temporais mais rigorosas do que algumas interrupções do mesmo sistema. Visto que, a arquitetura tradicional não permite atribuir a esta tarefa uma prioridade superior do que às interrupções de baixa prioridade, esta será inutilmente perturbada pelas interrupções relacionadas com as tarefas de baixa prioridade, comprometendo as garantias temporais do sistema. Um outro problema consiste na elevada complexidade e certas inconsistências do *kernel* causados pelas naturezas distintas da implementação da sincronização de tarefas e interrupções. Endereçamento destes problemas serviu de motivação para vários estudos, que propuseram a unificação de espaço de

prioridades e mecanismos de sincronização de tarefas e interrupções como sendo solução gratificante para os sistemas de tempo real (Leyva-del Foyo, 2006), (Hofer et al., 2009), (Scheler et al., 2009).

Nos sistemas tradicionais, o sinal enviado pelo controlador das interrupções com o intuito de interromper CPU é denominado IRQ (*interrupt request*), enquanto as rotinas que este desencadeia são denominadas ISRs (*interrupt service routines*). As interrupções são a maior fonte do *jitter* e imprevisibilidade nos sistemas computacionais. A razão para tal é a natureza assíncrona dos eventos externos e o requisito de baixas latências que é atingido comprometendo determinismo. Um outro problema surge quando a computação associada à interrupção é extensa. Visto que a maioria dos sistemas implementam ISRs como atividades não preemptivas, uma ISR demorada vai comprometer o atendimento das restantes interrupções que podem ocorrer durante a sua execução. Este problema é solucionado por maioria dos sistemas modernos atendendo a interrupção em duas etapas denominadas *"top-half"* e *"bottom-half"*. Onde a primeira etapa é efetivamente a rotina que atende a interrupção e prepara a segunda etapa para ser executada a seguir, dependendo da disponibilidade do CPU. Dependendo do sistema operativo, *"bottom-half"* pode ser implementado de diferentes maneiras, mas a principal ideia é que esta rotina já pode ser interrompida e se aproxima mais à tarefa. No trabalho (Leyva-del Foyo, 2006) a integração dos mecanismos de sincronização e espaços de prioridade é atingida convertendo as interrupções em eventos de sincronização (*"top-half"*) e implementando o corpo da ISR na forma de uma tarefa (denominada IST) igual às outras, escalonadas pelo *kernel* (*"bottom-half"*). Essa forma de abstração elimina as diferenças entre as interrupções e as tarefas. Sendo assim, ponto de vista do sistema, existem apenas tarefas (algumas dos quais desencadeadas pelas interrupções), o que simplifica e torna mais consistente o *kernel*, tornando obsoleto certo conjunto das suas funcionalidades e mecanismos originado pela anterior heterogeneidade. O trabalho (Leyva-del Foyo, 2006) compara o benefício de unificação do espaço de prioridades contra a desvantagem do aumento do *overhead* associado a atendimento às interrupções. Com base nos casos de utilização, que apresentam imprevisibilidade devido a inversão das prioridades no espaço de prioridades não integrado, foi demonstrado e provado matematicamente (para um subconjunto de casos) que a utilização do espaço de prioridades integrado, para além de tornar o sistema mais previsível, proporciona o aumento do limite de utilização deste. Por outro lado, o *jitter* associado ao processamento da etapa *"top-half"* do atendimento a interrupção permanece e não pode ser eliminado recorrendo às técnicas

em *software* que primeiro devem atender a interrupção antes de concluir se esta deve ser processada agora ou não. Para que esta fonte do *jitter* seja eliminada por completo os IRQs devem ser avaliados (*"top-half"*) numa entidade externa ao CPU.

Nos trabalhos associados ao projeto SLOTH (Hofer et al., 2009) a abordagem para integração do espaço de prioridades já é radicalmente oposta. Em vez de transformar interrupções em tarefas, todas as tarefas são implementadas como sendo ISRs tradicionais. A motivação dos autores consiste na delegação da responsabilidade pelo escalonamento tanto das tarefas como das interrupções ao controlador das interrupções existente nos dispositivos computacionais modernos. Foi estabelecido um conjunto dos requisitos que o controlador das interrupções deve cumprir para poder substituir quase por completo o escalonador em *software*. Estes requisitos são:

- Possibilidade de desencadear interrupção a partir de uma ISR ou tarefa (por *software*);
- Possibilidade de parcialmente desativar interrupções estipulando uma prioridade *threshold*;
- Possibilidade de acoplar *timer* em *hardware* ao *trigger* de uma interrupção específica para implementar desta forma uma tarefa temporizada;
- Suporte simultânea de um número suficiente de interrupções externas.

Assim, os autores selecionaram a plataforma Infineon-TriCore de uso comum na indústria automóvel e cujo controlador das interrupções suporta até 255 fontes de interrupções diferentes e cumpre os requisitos enumerados (Infieon Technologies, 2002). A implementação de todas as tarefas e interrupções como sendo ISRs que partilham uma única pilha tem as limitações em termos das funcionalidades comparando com os *kernels* tradicionais, no entanto, mantendo esta semântica, os autores conseguiram uma solução extremamente concisa (menos de 200 *source lines of code*), robusta e de fácil verificação. O sistema operativo por eles concebido está em conformidade com a classe BCC1 da especificação OSEK (OSEK Group, 2005), podendo ser estendida, ao custo de um *overhead* suportável (Hofer et al., 2011), para estar em conformidade com a classe ECC1. Especificação OSEK é uma norma que regula sistemas operativos embebidos e que é adotada pela indústria automóvel. A solução usufrui das vantagens da integração do espaço de prioridades e explora as potencialidades do *hardware*, superando assim, em termos de níveis de

latências e determinismo, as soluções comercializadas que estão em conformidade com a mesma norma.

Como já foi referido, quando a integração do espaço de prioridades vai pelo caminho de transformação das interrupções em tarefas, o *jitter* e indeterminismo associados à interrupção especulativa do CPU, necessária para processamento *"top-half"* dos IRQs, não podem ser eliminados por completo recorrendo as técnicas em *software*. Idealmente, a avaliação e escalonamento do IRQ deve ocorrer numa entidade independente do CPU e sem a sua interrupção. Para que seja possível integrar ISR no contexto do escalonador do *kernel* (que escalona tarefas) sem interromper CPU, o dito escalonador também deve encontrar se numa entidade desacoplada do CPU, ou seja, deve ser implementado como sendo coprocessador em *hardware*. Um exemplo do *microkernel* em *hardware* com a integração do espaço de prioridades pode ser encontrado no projeto hthreads (Andrews et al., 2005). O módulo do *microkernel* denominado CBIS (CPU *Bypass Interrupt Scheduler*) tem a capacidade de guardar em *hardware* a associação entre uma fonte de interrupção externa e uma tarefa do escalonador, o que é semelhante a assinatura do determinado *handler* ao determinado IRQ no sistema tradicional. Essa informação é posteriormente usada para poder traduzir um evento assíncrono da interrupção num evento assíncrono do escalonamento, direcionado para escalonador baseado em *hardware* sem passar pelo CPU. Isto possibilita o escalonamento das interrupções em paralelo com a execução da aplicação do sistema, eliminando assim por completo, a necessidade do processamento *"top-half"* (ponto de vista do CPU) e o respetivo *overhead* computacional. Ao mesmo tempo, a parte *"bottom-half"* do atendimento à interrupção é levada ao domínio do sistema operativo e pode usufruir das funcionalidades e flexibilidades que este proporciona às tarefas normais.

Visto que, o trabalho desenvolvido no âmbito desta dissertação implementa *hardware microkernel*, é possível integrar o espaço de prioridades das tarefas e interrupções seguindo abordagem apresentada no projeto hthreads. O modelo implementado é em tudo parecido mas, no entanto, tem as suas particularidades. Os detalhes da implementação são apresentados na Secção 5.5.3.

2.4 Tecnologia FPGA

A tecnologia FPGA (*Field-Programmable Gate Array*) na altura da sua invenção, no ano 1984, originou um novo mercado que desde aquele momento se encon-

tra em constante amadurecimento e crescimento. Um dispositivo FPGA é um circuito integrado constituído pelos blocos lógicos preconstruídos e que é dotado com a capacidade de roteamento reprogramável. Desta forma, o dispositivo pode ser configurado para implementar uma funcionalidade específica na forma de um circuito digital dedicado, já após o seu fabrico.

Para especificação do projeto pretendido são usadas as linguagens de programação, denominadas linguagens de descrição de *hardware* ou HDL (*Hardware Description Language*).

A tecnologia FPGA permanece em constante desenvolvimento, com especial relevância, no âmbito da sua utilização nos sistemas embebidos. As vantagens da tecnologia FPGA são apresentadas nos próximos parágrafos.

Desempenho

As soluções baseadas em FPGA exploram o paralelismo do *hardware*, quebrando o paradigma da programação sequencial. Os dispositivos FPGA modernos incluem na sua arquitetura recursos especiais preconstruidos (multiplicadores e DSP *Slices*) que os torna competitivos perante dispositivos DSP (*Digital Signal Processor*) e CPUs do uso geral. Em certas aplicações, as soluções baseadas em FPGA excedem as capacidades computacionais dos dispositivos DSP suficientemente para atingir uma relação custo-eficácia muito maior (Berkeley Design Technology Inc, 2007).

Tempo até o mercado

A tecnologia oferece flexibilidade e rapidez da prototipagem, não sendo dependente do demorado processo do fabrico de uma solução customizada ASIC (*Application-Specific Integrated Circuit*). É possível verificar o conceito e conceber o primeiro protótipo num prazo curto, tendo posteriormente a flexibilidade de implementar alterações incrementais nas futuras iterações do projeto. As empresas vendedoras dos dispositivos FPGA, em paralelo com a evolução das arquiteturas utilizadas nos dispositivos modernos, investem também fortemente no desenvolvimento de soluções que facilitam o projeto de sistemas complexos baseados em FPGA. *Software* automatizado de projeto de circuitos integrados torna-se cada vez mais sofisticado. Ao mesmo tempo que é feita uma forte aposta no desenvolvimento de bibliotecas das funções complexas predefinidas e circuitos predefinidos submetidos a validação e otimização rigorosas e que são acompanhados pelas ferramentas que

proporcionam a sua fácil integração no sistema final. Estas ferramentas e bibliotecas proporcionam um aumento de produtividade do desenvolvedor e potenciam a utilização da tecnologia. No entanto, ainda se prevê muito trabalho, no âmbito de diminuição do nível da complexidade do desenvolvimento, até o tornar próximo ao de cenários da utilização dos dispositivos DSP e CPUs de uso geral.

Custo

Em comparação com dispositivos ASIC cujo desenvolvimento apresenta custos NRE (*Non-recurring engineering*) muito elevados, fazendo viável a utilização desta tecnologia apenas para produção em massa, o desenvolvimento baseado na tecnologia FPGA tem custos não recorrentes muito baixos. Para além disso, no mercado atual, os requisitos que os produtos devem endereçar alteram com uma frequência elevada. O custo associado a alteração incremental de um projeto baseado em FPGA é desprezível em comparação com o custo associado ao redesenho de um dispositivo ASIC.

Confiabilidade

Neste aspeto a tecnologia FPGA pode ser comparada face as soluções tradicionais baseadas no CPU. Quando se trata de um ambiente de execução multitarefa, os sistemas baseados no CPU introduzem várias camadas de abstração, cujo intuito é facilitar o processo de escalonamento e virtualização dos recursos. Num determinado processador, a qualquer momento, apenas instruções de uma tarefa podem efetivamente executar, existindo sempre o risco de tarefas que apresentam requisitos temporais interromper umas às outras, não esquecendo também do facto, que as instruções dos próprios serviços do sistema operativo precisam de acesso ao CPU para executar. Uma solução baseada na tecnologia FPGA não lida com ambiente de programação *software* e é uma realização do programa diretamente em *hardware*. A ausência da necessidade do sistema operativo junto com a execução efetivamente paralela de tarefas distintas, bem como a natureza determinística do *hardware* potenciam a confiabilidade da solução resultante.

Manutenção

Ao contrario da tecnologia ASIC, os dispositivos FPGA são reprogramáveis. Desta forma, soluções baseadas na tecnologia FPGA têm capacidade de implementar

alterações possivelmente necessárias. É possível implementar melhorias funcionais com despesas temporais e monetárias baixas.

2.5 Linguagens de programação HDL

As linguagens HDL (*Hardware Description Language*) são as linguagens usadas para projeto de um circuito eletrónico, descrição formal da sua estrutura e funcionamento. Por norma, trata-se do projeto do circuito eletrónico digital. Ao contrário da maioria das linguagens de programação, as linguagens HDL incluem explicitamente a noção temporal. Depois de descrever a organização e a operação do circuito, este pode ser testado, com intuito de verificação, efetuando uma simulação. Ou seja, a linguagem HDL pode ser usada para implementação de uma especificação executável do *hardware*. O simulador neste contexto é um programa que implementa a semântica dos *statements* compreendidos na linguagem enquanto simula o progresso do tempo possibilitando, desta forma, a verificação de um determinado circuito antes da sua implementação física.

VHDL e Verilog são duas linguagens HDL padronizadas: IEEE 1076-2008 e IEEE 1364-2005, respetivamente. Estas linguagens são melhor suportadas e mais utilizadas na indústria.

As ditas linguagens se baseiam na abstração RTL (*Register Transfer Level*), que modela circuito digital síncrono em termos do fluxo de sinais digitais entre registos *hardware* e operações lógicas realizadas sobre estes sinais. É possível efetuar simulações, mas também é possível compilar e sintetizar o código, sendo o resultado de saída um *netlist*, que é uma especificação do circuito digital de um nível mais baixo que pode resultar numa implementação num dispositivo FPGA concreto, após recurso às ferramentas de mapeamento, posicionamento e roteamento associadas a este dispositivo. Nem todo o código HDL sintaticamente correto é sintetizável e, também, nem todo o código sintetizável pode ser implementado num dispositivo FPGA específico em função dos seus recursos. Também pode ocorrer situação em que o projeto é sintetizável e até, provavelmente, pode ser instanciado para o dispositivo em causa, mas as ferramentas automatizadas de mapeamento, posicionamento e roteamento resultam em insucesso, exigindo a intervenção manual do desenvolvedor.

2.5.1 VHDL e Verilog

Como já foi referido, VHDL e Verilog são duas linguagens de descrição de *hardware* que são melhor suportadas e mais utilizadas na industria.

A questão de adoção de uma linguagem ou outra é uma questão que emerge naturalmente para uma empresa que lida com o desenvolvimento do *hardware*. Existem óbvias vantagens logísticas em escolher apenas uma linguagem para ser adotada numa determinada empresa, mesmo existindo a possibilidade de usar a mistura de duas linguagens num projeto só. A escolha entre VHDL e Verilog é uma escolha controversa sem existir a possibilidade de afirmar que uma das linguagens é melhor que a outra. Mesmo assim, há possibilidade de comparação baseada num conjunto de aspetos que é apresentada nos próximos parágrafos, que referem a (Smith) e (Bailey). De notar que é feita a comparação da linguagem VHDL com a linguagem Verilog sem contabilizar as funcionalidades do *superset* da linguagem Verilog denominado SystemVerilog e cuja versão atual de padronização é IEEE 1800-2012.

Capacidade de modelação

Ao nível da abstração RTL as linguagens tem a capacidade equivalente e um projeto implementado em Verilog pode ser implementado em VHDL e vice-versa. Entretanto, no que diz respeito aos outros níveis de abstração, linguagem Verilog é mais apropriada para modelação aos níveis mais baixos como *gate level* e *switch level* tendo para isso primitivas embutidas, enquanto VHDL tem certas capacidades de alto nível que no caso da Verilog são muito limitadas. Falta de capacidades de modelação de baixo nível na linguagem VHDL foi endereçada com a introdução do padrão IEEE 1076.4 VITAL (VHDL *Initiative Towards ASIC Libraries*) que implementa pacotes que estendem a linguagem.

"Strong" and "weak" typing

Aquando classificação de uma linguagem de programação, pode ser dito que a linguagem é *strongly typed* ou *weakly typed*. Não existe uma definição precisa destes termos. A sua atribuição é ligada a presença ou ausência de certas funcionalidades nativas da linguagem ligadas ao seu sistema de tipos.

Segundo a definição do Microsoft Developer Network (MSDN, 2013):

- A linguagem de programação C é *weakly typed*, visto que os respetivos compiladores permitem operações como atribuição e comparação entre variáveis de tipos diferentes. Assim, por exemplo, a linguagem permite que seja feito o *cast* de uma variável para um tipo distinto. A possibilidade de utilizar variáveis de diferentes tipos na mesma expressão proporciona flexibilidade e eficiência.

- Uma linguagem *strongly typed* impõe restrições sobre as operações entre as variáveis de diferentes tipos. É emitido o erro durante o tempo de compilação que proíbe a operação. A rigidez do sistema de tipos é projetada para minimizar o número de possíveis erros ainda durante a compilação.

A linguagem VHDL é muito *strongly typed*, o que implica a codificação adicional relacionada com a conversão explicita de um tipo de dados para outro o que, inicialmente, pode prejudicar a produtividade do desenvolvedor. No entanto, a rigorosa verificação de tipos resulta na deteção de certos erros ainda na fase da análise do código fonte.

Ao contrário do VHDL, Verilog é um extremo oposto e é um exemplo de uma linguagem muito *weakly typed*. As representações de dados suportadas podem ser livremente misturadas. Assim, por exemplo, é possível a atribuição de um vetor de 8 *bits* a um vetor de 16 *bits*. Desta forma, o desempenho não é penalizado devido à verificação de tipos e pressupõe-se uma maior produtividade do desenvolvedor. No entanto, os erros não intencionais não são detetados pelo analisador.

Tipos de dados

VHDL permite a definição de tipos de dados não triviais customizados junto com a utilização de tipos de dados predefinidos na linguagem. As vezes, isso leva a necessidade de uso excessivo de funções de conversão. A escolha bem ponderada de tipos de dados a serem usados pode tornar o código mais legível e mais fácil de escrever. Ou seja, a linguagem VHDL pode ser preferida devido a um número elevado de tipos de dados, tanto definidos pela linguagem, como pelo utilizador.

Por sua vez, os tipos de dados da linguagem Verilog são muito simples, são também simples de utilizar e mais próximos à descrição estrutural do *hardware*. Ao contrário da linguagem VHDL, a funcionalidade da definição pelo utilizador dos tipos de dados é ausente, e todos os tipos compreendidos pela linguagem são definidos nesta.

No que diz respeito aos tipos de dados correspondentes ao registo e conexão, Verilog faz uma distinção clara e proporciona para o desenvolvedor tipos de dados `reg` e `wire`, respetivamente, enquanto no caso da linguagem VHDL tanto o objeto que resulta numa conexão, como o objeto com a capacidade de memória (registo) são declarados como `signal` e a sua distinção é feita com base no seu uso.

Facilidade de aprendizagem

A sintaxe da VHDL tem as suas raízes na linguagem ADA. Devido a consequente verbosidade e também divido a *strongly typing*, por norma, considera-se que esta linguagem exige mais tempo para aprendizagem. Por sua vez, a sintaxe da linguagem Verilog se baseia numa mistura da linguagem ADA com a linguagem C e revela-se sendo muito intuitiva para os utilizadores da linguagem C. Desta forma, em termos da facilidade de aprendizagem, a linguagem Verilog normalmente é considerada como sendo mais fácil das duas.

Capacidade de reutilização

A linguagem VHDL implementa o conceito de pacotes e a compilação separada, sendo que as funções e procedimentos podem ser colocados dentro de um pacote e ser reutilizados em vários projetos. Ao contrario da VHDL, Verilog não implementa pacotes e a capacidade de reutilização nesta linguagem é muito reduzida. No entanto, o conceito do pacote pode ser emulado na linguagem Verilog através da declaração e instanciação de um módulo *dummy* com as funções dentro.

Capacidade de modelação de alto nível

Enquanto Verilog possui apenas a possibilidade de parametrização dos seus módulos com os parâmetros que redefinem as constantes, VHDL proporciona um vasto número de funcionalidades de modelação de alto nível:

- tipos de dados abstratos;
- conceito de bibliotecas;
- pacotes (`package`) para reutilização dos modelos;
- configurações (`configuration`) para a configuração da estrutura do projeto;

- replicação das estuturas (`generate`);
- parametrização dos módulos genéricos (`generic`).

Capacidade de modelação de baixo nível

Na altura da sua criação, a linguagem Verilog foi projetada para ter o suporte a modelação de baixo nível embutido na própria linguagem, bem como as funcionalidades necessárias para modelação de primitivas das células ASIC e FPGA. Por sua vez, a linguagem VHDL tem embutido apenas operadores lógicos de duas entradas. Como já foi referido, a falta de capacidades de modelação de baixo nível na linguagem VHDL foi endereçada com a introdução do padrão IEEE 1076.4 VITAL (VHDL *Initiative Towards ASIC Libraries*) que implementa pacotes que estendem a linguagem.

2.5.2 SystemVerilog

SystemVerilog é uma linguagem HDL que surgiu a partir do Verilog, e beneficiou da extensão da Verilog conhecida como Superlog e das linguagens C/C++. Esta linguagem estende a Verilog através da inclusão de um sistema de tipos definidos pelo utilizador. Para este sistema de tipos é implementada a verificação forte (*strong typing*). No entanto, com intuito de compatibilidade com o código legado escrito na linguagem Verilog, foi mantida a verificação fraca dos tipos embutidos na linguagem Verilog (*weak typing*).

As extensões maioritariamente endereçam a falta da capacidade de modelação de alto nível tão evidente na comparação da Verilog com a VHDL.

2.6 Diagramas SMChart

Processo de projeto de engenharia exige rigor e método. Digna elaboração da fase de projeto (conceção da especificação do sistema a ser implementado) é crucial. É indiscutível que a implementação do *software* sem recurso às ferramentas de modelação é difícil, ineficiente e dispendiosa. O mesmo se aplica ao projeto do *hardware*. A implementação dos circuitos digitais no projeto associado a esta dissertação baseia-se na linguagem VHDL, com a utilização da descrição estrutural aquando blocos do topo e descrição comportamental aquando sub-blocos. Para

a modelação do sistema digital em termos do seu comportamento, foi adotada a utilização dos diagramas denominados *state machine flow chart* ou SMChart (Roth Jr., 1998).

Trata-se de uma ferramenta para modelação das maquinas de estado, que utiliza a nomenclatura do *flowchart*, habitual no projeto de *software*, mas que deve cumprir um conjunto de regras durante a sua construção. O diagrama resultante é equivalente ao *state graph* adequado, mas apresenta uma maior legibilidade e se transpõe com uma maior facilidade para os *statements* da linguagem de descrição do hardware. A Figura 2.3 mostra os componentes que constituem um diagrama SMChart.

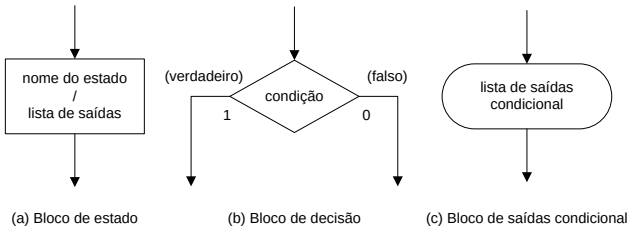


Figura 2.3: Componentes que constituem um diagrama SMChart

Cada estado da maquina de estados que está a ser modelada é representado por um bloco de estado (símbolo retangular). Este contém o nome do estado seguido pelo simbolo "/" e uma lista opcional de saídas. O símbolo na forma de diamante representa um bloco de decisão. Este tem uma condição no seu interior que é uma simples expressão booleana. O último componente utilizado na construção do diagrama SMChart é o bloco de saídas condicional (símbolo retangular com os vértices curvos). Este contem uma lista de saídas condicional. A lista é chamada condicional porque o respetivo bloco só é colocado após bloco(s) de condição, por sua vez colocado(s) após o bloco de estado. A Figura 2.4 apresenta a entidade base constituída por componentes enumerados.

Um diagrama SMChart para ser válido deve ser construido a partir das entidades base. Uma entidade base corresponde a um estado e a respetiva rede dos caminhos condicionais define o conjunto de saídas condicionais e o caminho de saída que, por sua vez, liga à próxima entidade base. Uma das saídas da entidade base pode realimentar a própria entrada, enquanto a realimentação dentro da rede de caminhos condicionais não é permitida. A cada ciclo do relógio, que alimenta a maquina de estados modelada pelo SMChart, ocorre a transição de uma entidade base para outra (de um estado para outro). As listas de saídas contêm os sinais

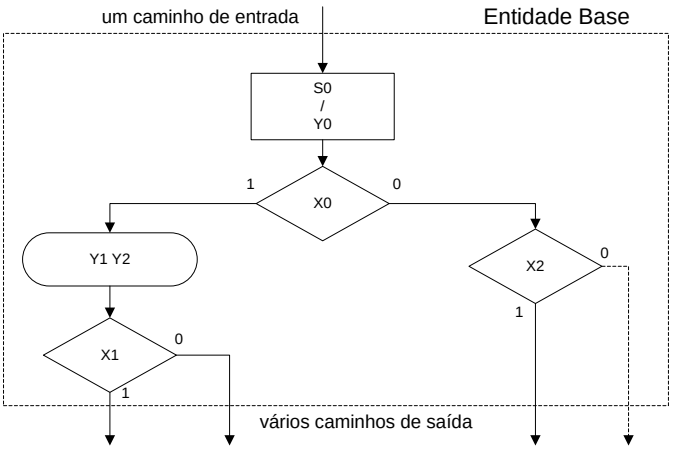


Figura 2.4: Entidade base do diagrama SMChart

que estão no nível lógico alto ("1") enquanto a maquina de estados se encontra no estado correspondente a respetiva entidade base.

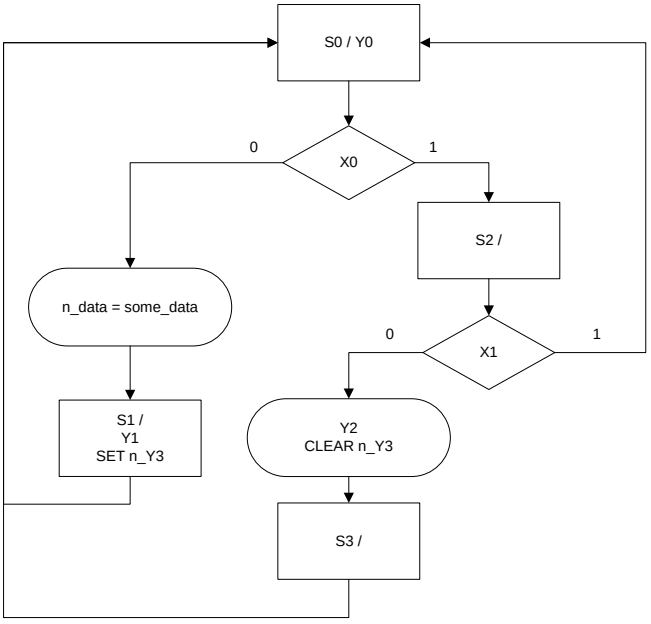


Figura 2.5: Exemplo de um diagrama SMChart com a nomenclatura estendida

O diagrama SMChart, como apresentado no livro acima referido, é utilizado para implementação de unidades de controlo com as saídas (palavra do controlo) a depender, através da lógica combinacional, do estado atual e das entradas. Único sinal com a capacidade de guardar o seu estado é o estado atual da maquina de estados. No âmbito desta dissertação, os diagramas SMChart não foram usadas no seu formato puro e foram estendidas com os sinais com a capacidade de memoria. Assim sendo, as listas de saídas do SMChart passaram a poder conter comandos

SET e *CLEAR* para sinais de um *bit*, bem como atribuições para sinais de vários *bits*. Ao contrario dos sinais combinacionais na lista de saídas, estes comandos tem efeito a partir do próximo ciclo, e que permanece devido a capacidade da memoria dos respetivos sinais.

A Figura 2.5 dá exemplo de um diagrama SMChart com a nomenclatura estendida, enquanto a Listagem 2.1 apresenta a respetiva implementação na linguagem VHDL.

```
entity Dummy_SMChart is
  port(X0_in, X1_in                    : in  std_logic;
       some_data_in                    : in  std_logic_vector(0 to 7));
       Y0_out, Y1_out, Y2_out, Y3_out  : out std_logic;
       data_out                        : out std_logic_vector(0 to 7));
end entity Dummy;

architecture behave of Dummy_SMChart is
  signal state, next_state        : integer range 0 to 3;
  signal Y0, Y1, Y2, Y3, next_Y3  : std_logic;
  signal data, next_data          : std_logic_vector(0 to 7);
begin
  Y0_out <= Y0; Y1_out <= Y1; Y2_out <= Y2; Y3_out <= Y3;
  data_out <= data;
  fsm: process(state, X0_in, X1_in, Y3, data)
  begin
    Y0 <= '0'; Y1 <= '0'; Y2 <= '0';
    next_Y3 <= Y3; next_data <= data; next_state <= state;
    case state is
      when 0 => Y0 <= '1'                                -- state 0
                if X0_in = '1' then next_state <= 2;
                else
                  next_data <= some_data_in;
                  next_state <= 1;
                end if;
      when 1 => Y1 <= '1'; next_Y3 <= '1';               -- state 1
                next_state <= 0;
      when 2 => if X1_in = '1' then next_state <= 0;     -- state 2
                else
                  Y2 <= '1';
                  next_Y3 <= '0';
                  next_state <= 3;
                end if;
      when 3 => next_state <= 0;                         -- state 3
    end case;
  end process fsm;
  state: process(clk)
  begin
    if (clk'event and (clk = '1')) then
      state <= next_state;
      data  <= next_data; Y3 <= next_Y3;
    end if;
  end process state;
end behave;
```

Listagem 2.1: Implementação genérica na linguagem VHDL baseada no SMChart da Figura 2.5

2.7 Conclusões

O objeto de estudo desta dissertação são os sistemas operativos de tempo real no âmbito da sua utilização nos sistemas embebidos dotados com dispositivo FPGA. Pretende-se implementar um *kernel* simplista de tempo real assistido por *hardware* recorrendo a tecnologia SoPC (*System-on-a-Programmable-Chip*) e recorrendo a tarefa de *refactoring* de um sistema operativo existente.

Este capítulo visa fundamentar e familiarizar a leitura do documento com os termos técnicos, conceitos e tecnologias relacionados com a presente dissertação e necessários para o sucesso do trabalho relacionado, tais como: sistemas embebidos, sistemas operativos, tempo real, unificação de espaço de prioridades das tarefas e interrupções, tecnologia FPGA e metodologias do desenvolvimento *hardware*.

Foi apresentada uma visão abrangente sobre sistemas operativos de tempo real, mecanismos do escalonamento e sincronização, bem como, desafios e soluções, associados a busca do determinismo nos atuais sistemas embebidos. Foram também discutidas as métricas de avaliação de um sistema operativo de tempo real.

Foi feita a contextualização no conceito de unificação de espaço de prioridades de tarefas e interrupções, através da apresentação da respetiva motivação e trabalhos efetuados.

Foi também feita uma breve introdução na tecnologia FPGA e metodologias do desenvolvimento *hardware* subjacentes.

Capítulo 3

Ambiente de Desenvolvimento

Este capítulo apresenta a plataforma do desenvolvimento utilizada, bem como o ambiente do desenvolvimento em que esta se encontra integrada.

O objetivo desta dissertação é a implementação de um sistema operativo de tempo real assistido por *hardware*. A tarefa chave é a conceção de um *hardware microkernel* na forma de um conjunto de coprocessadores e a sua implementação e integração junto com um CPU genérico. Esta ação baseia-se na utilização da tecnologia SoPC (*System-on-a-Programmable-Chip*) que consiste na integração da área reprogramável FPGA junto com os processadores genéricos num dispositivo só e no desenvolvimento de bibliotecas das funções complexas predefinidas e circuitos predefinidos, submetidos a validação e otimização rigorosas e que são acompanhados pelas ferramentas que proporcionam a sua fácil integração no sistema final.

As duas grandes empresas que dominam o mercado da tecnologia FPGA são Xilinx e Altera (Bacon et al., 2013). Estes produtores apresentam dispositivos que se enquadram na definição SoPC, bem como as ferramentas e bibliotecas que proporcionam um aumento de produtividade do desenvolvedor e potenciam a utilização da tecnologia. Para facilitar o desenvolvimento baseado nesta tecnologia, os produtores proporcionam as plataformas de desenvolvimento que integram o dispositivo FPGA rodeado por um conjunto abrangente de periféricos. A presença deste conjunto de periféricos permite desenvolvimento dos sistemas complexos e facilita a exploração das capacidades do dispositivo.

Trata-se de material dispendioso, sendo o custo comercial de uma plataforma de desenvolvimento do produtor Xilinx, endereçada para a construção de um SoC, está acima dos mil dólares americanos, existindo alternativas *community-based*,

como é o exemplo da plataforma ZedBoard (ZedBoard), avaliada em quatrocentos dólares.

O projeto relacionado com esta dissertação foi elaborado com a utilização da plataforma do desenvolvimento XUPV2 do produtor Xilinx que foi disponibilizada pelo grupo de investigação ESRG (*Embedded Systems Research Group*) que é integrado no centro de investigação Algoritmi (ESRG).

3.1 Plataforma de Desenvolvimento XUPV2

A plataforma do desenvolvimento do Xilinx, XUPV2, apresenta-se como uma plataforma de *hardware* avançada, dotada com o dispositivo FPGA de alto desempenho XC2VP30, da família Virtex-II Pro (Xilinx, 2009). Esta plataforma do desenvolvimento pertence ao programa académico do Xilinx, denominado *Xilinx University Program*, o que também consta no seu nome. É posicionada como suficientemente potente para ser utilizada nos projetos avançados de investigação, sendo ao mesmo tempo suficientemente acessível.

A plataforma integra um conjunto abrangente de periféricos ligados ao dispositivo FPGA que podem ser usados na construção de sistemas complexos. O diagrama de blocos que foi retirada do manual de referencia do *hardware* (Xilinx, 2009) e que resume a constituição da plataforma é apresentado na Figura 3.1.

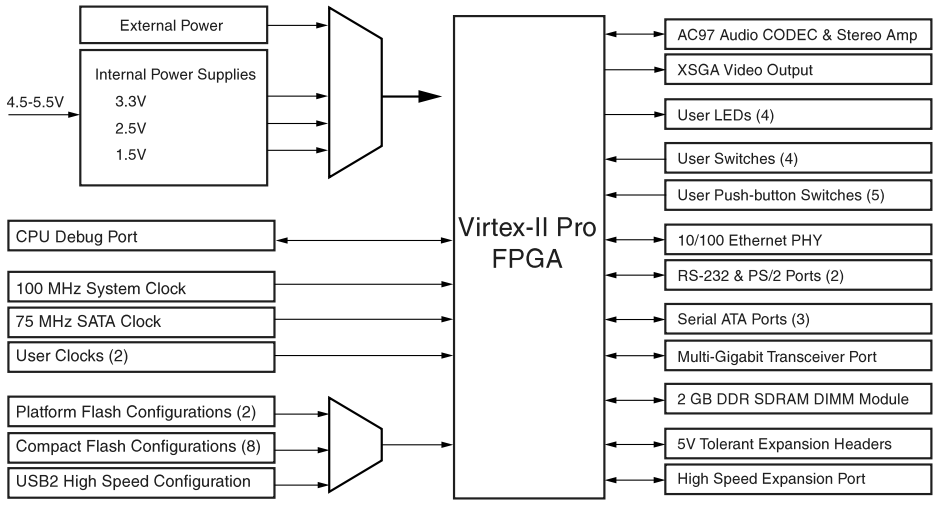


Figura 3.1: Diagrama de blocos da plataforma do desenvolvimento XUPV2

O dispositivo FPGA XC2VP30 pertence a família dos dispositivos Virtex-II Pro e em termos das suas capacidades encontra-se no meio da sua gama. A sua constituição em termos das quantidades dos determinados recursos está resumida na Tabela 3.1.

Tabela 3.1: Capacidade do dispositivo XC2VP30

RocketIO Transceiver Blocks	8
PowerPC Processor Blocks	2
Logic Cells	30 816
CLB	3 424
Slices	13 696
Max Distr RAM (Kb)	428
18 X 18 Bit Multiplier Blocks	136
BRAM 18 Kb Blocks	136
Max Block RAM (Kb)	2 448
DCMs	8
Maximum User I/O Pads	644

É de elevada importância referir, que o dispositivo integra no seu interior, junto à área reprogramável FPGA, dois blocos que implementam processadores PPC405. Ou seja, é possível a construção de um SoC baseado num processador RISC de elevada potência implementado em silício (*hard core*). Uma outra opção promovida pelo ambiente do desenvolvimento é a construção do SoC com base nos processadores *soft core* MicroBlaze (implementados na área reprogramável FPGA) desenvolvidos pelo Xilinx (Xilinx, 2008a).

3.2 Xilinx *Embedded Development Kit* 10.1

A plataforma XUPV2 encontra-se integrada no ambiente do desenvolvimento EDK 10.1. Esta secção apresenta as ferramentas que o constituem do ponto de vista geral, bem como do ponto de vista da sua utilização nesta dissertação.

O Xilinx *Embedded Development Kit* é um ambiente do desenvolvimento integrado do *software* e *hardware* para conceção completa dos sistemas embebidos baseados nos dispositivos FPGA. Consiste num conjunto de ferramentas, documentação e biblioteca de blocos reutilizáveis IP para projetar sistemas com processador embebido IBM PowerPC (*hard core*) e/ou processador embebido Xilinx Microblaze (*soft core*).

3.2.1 Ciclo de Desenvolvimento

O projeto embebido baseado no dispositivo FPGA engloba o desenvolvimento *software*, desenvolvimento *hardware*, bem como a gestão dos componentes e respetivas interligações que constituem o sistema . O diagrama que descreve o ciclo do desenvolvimento do sistema embebido baseado no *Embedded Development Kit* pode ser visto na Figura 3.2.

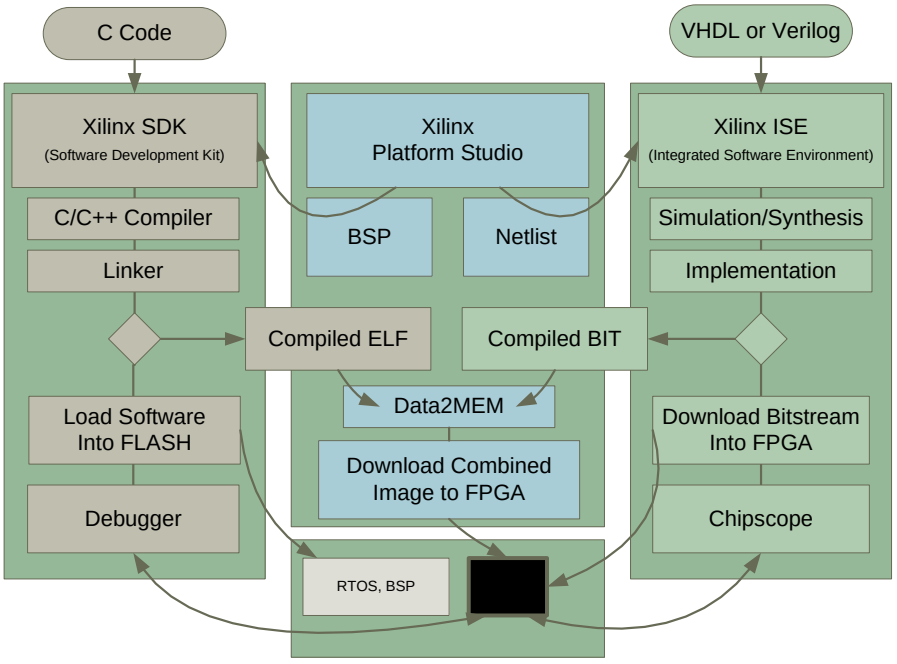


Figura 3.2: Ciclo do desenvolvimento do *Embedded Development Kit*, adaptado de (Florida Internation University)

No diagrama observam-se as três principais ferramentas que maioritariamentee constituem EDK: XPS (*Xilinx Platform Studio*), ISE (*Integrated Software Environment*) e SDK (*Software Development Kit*).

3.2.2 XPS *Xilinx Platform Studio*

XPS (*Xilinx Platform Studio*) é uma ferramenta IDE (*Integrated Design Environment*) gráfica, utilizada para desenvolvimento e verificação do projeto embebido baseado no FPGA. Esta ferramenta integra o número abrangente das ferramentas que constituem EDK e proporciona ao desenvolvedor um ambiente gráfico que facilita a gestão do projeto.

A Figura 3.3 apresenta o ciclo do desenvolvimento do projeto no ambiente EDK, desta vez com um maior foco nos ficheiros que compreendem o sistema completo e cuja gestão automatizada é proporcionada pelo XPS.

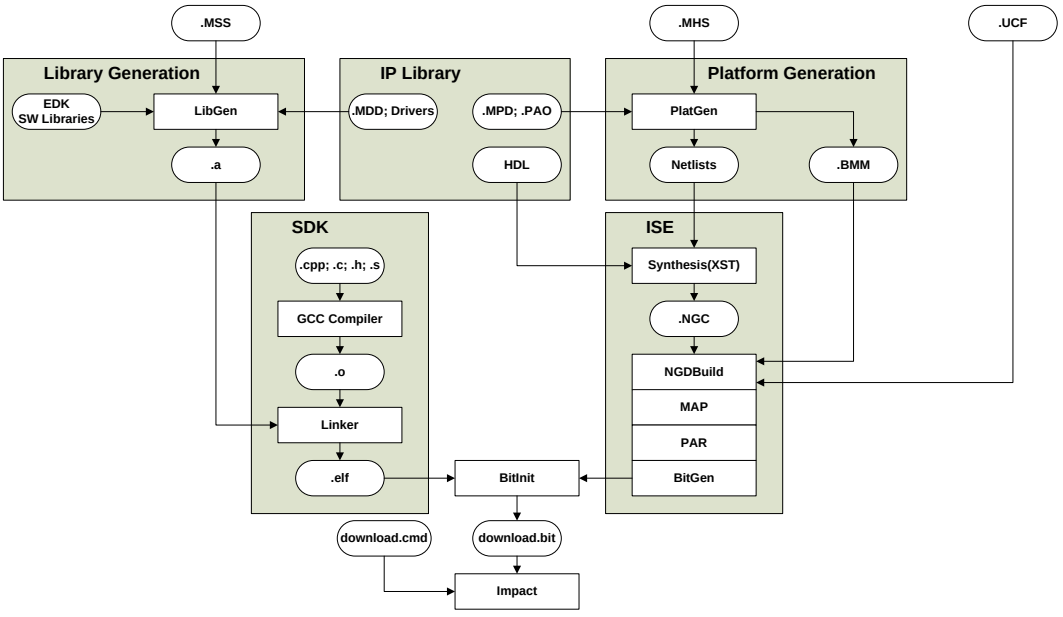


Figura 3.3: Ciclo do desenvolvimento do *Embedded Development Kit*, adaptado de (Florida Internation University)

BSB (*Base System Builder*)

Para conceção de um ponto de partida para um projeto baseado no processador, *Xilinx Platform Studio* integra a ferramenta denominada BSB (*Base System Builder*). Esta ferramenta automatiza a tarefa de configuração do *hardware* e *software* da plataforma. Quando é endereçada uma plataforma do desenvolvimento do produtor Xilinx ou uma outra plataforma oficial de algum parceiro deste, BSB permite instanciar processador ou processadores, configurar barramentos e memórias e escolher os periféricos pretendidos entre aqueles que a plataforma compreende. Com base nesta informação, introduzida pelo utilizador através de uma interface simplista e intuitiva, a ferramenta gere ficheiros .mss, .mhs e .ucf, explicados em maior detalhe na próxima secção. Para além disso, a ferramenta pode criar um projeto embebido de *software* que testa o correto funcionamento da memória instanciada, bem como o projeto que testa o correto funcionamento dos periféricos selecionados. Os ditos projetos são prontos para o *download* e execução na placa. A ferramenta tem a mesma utilidade para o caso de uma plataforma não oficial, desde que o dispositivo FPGA que esta integra seja suportado pela versão utilizada

do EDK. Para a conclusão da especificação do sistema completo, o desenvolvedor, neste caso, deverá introduzir as localizações dos pinos do dispositivo FPGA e as respetivas restrições no ficheiro UCF do projeto gerado.

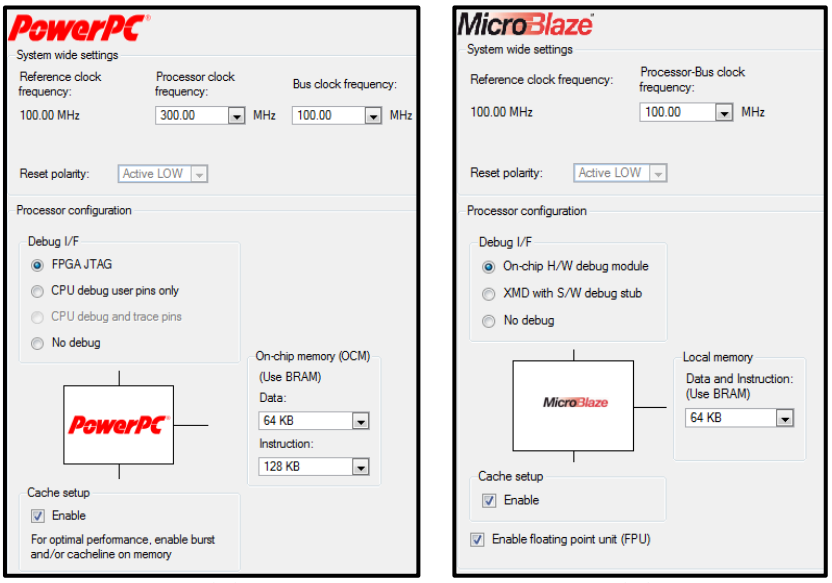


Figura 3.4: Janela da configuração do sistema baseado no: (a) - processador *hard core* PPC, (b) processador *soft core* MicroBlaze

A Figuras 3.4(a),(b) apresentam a primeira janela da configuração do BSB, após a escolha da plataforma XUPV2 e cenário da utilização do processador *hard core* PPC ou processador *soft core* MicroBlaze, respetivamente. A Tabela 3.2 resume todas as opções da configuração do projeto baseado no processador, possibilitadas pelo BSB no caso da plataforma XUPV2.

É possível observar, que a diferente natureza da implementação (*hard core* ou *soft core*) tem o impacto na frequência máxima da operação dos processadores. A frequência máxima do funcionamento para o processador *hard core* PPC é compreendida nos 300 MHz, enquanto para o processador *soft core* MicroBlaze - nos 100 MHz.

No caso do processador PPC, a frequência do barramento PLB (*Processor Local Bus*) pode ser configurada separadamente, enquanto no caso da utilização do MicroBlaze, esta é obrigatoriamente igual à frequência do processador.

Pode ser também observada a diferença, no que diz respeito à organização da interface da memória local e unidades *cache*. Ambos processadores implementam a arquitetura da memória *Harvard* (Xilinx, 2008a),(Xilinx, 2002), sendo os acessos aos dados e às instruções feitos nos espaços de memória separados.

Tabela 3.2: Opções da configuração do sistema baseado no PPC ou MicroBlaze

IBM PowerPC	
Processor Clock Frequency (MHz)	300 / 250 / 200 / 150 / 100 / 66,67
Bus Clock Frequency (MHz)	1 / $\frac{1}{2}$ / $\frac{1}{3}$ / $\frac{1}{4}$ of processor's; ≤ 100
Debug I/F	FPGA JTAG / CPU debug user pins / No
Cache Setup	Yes / No
OCM Data (KB)	from 8 to 64
OCM Instruction (KB)	from 4 to 128
Cache Data (KB)	fixed at 16
Cache Instruction (KB)	fixed at 16
Xilinx MicroBlaze	
Processor Clock Frequency (MHz)	100 / 75 / 66,67 / 50 / 33,33 / 25
Bus Clock Frequency (MHz)	same as processor's
Debug I/F	On-Chip HW module / XMD SW stubs
Cache Setup	Yes / No
Floating Point Unit	Yes / No
Local Memory Data and Instruction (KB)	from 8 to 64
Cache Data (*)	from 64B to 64KB
Cache Instruction (KB)	from 8 to 64

As unidades *cache* do processador *hard core* PPC405 têm um tamanho fixo de 16KB, enquanto as unidades *cache soft core* do MicroBlaze são extremamente configuráveis.

No que diz respeito a memória *non-cacheble*, que apresenta o acesso mais rápido, ambos processadores possuem interfaces dedicados para o propósito. No caso do PPC, dita interface é denominada OCM (*On-Chip Memory*), as particularidades da implementação das interfaces da memória do lado das instruções e do lado dos dados impõem que sejam utilizadas duas unidades BRAM fisicamente separadas. Estas mesmas particularidades impõem restrição sobre a utilização da memória do lado das instruções, não sendo possível a sua leitura, apenas *fetch* pelo processador. Esta limitação torna impossível o *debug* pelo *software*. A Figura 3.5 apresenta a organização das interfaces da memória do processador PPC405. Por sua vez, o processador *soft core* MicroBlaze possui a interface dedicada LMB (*Local Memory Bus*) para o acesso à memória local com a latência associada igual a um ciclo de relógio. Os controladores da memória do lado das instruções e do lado dos dados ligados a interface LMB podem partilhar o mesmo espaço de memória e ligar aos dois portos da mesma unidade BRAM, o que é útil para *debug* pelo *software*

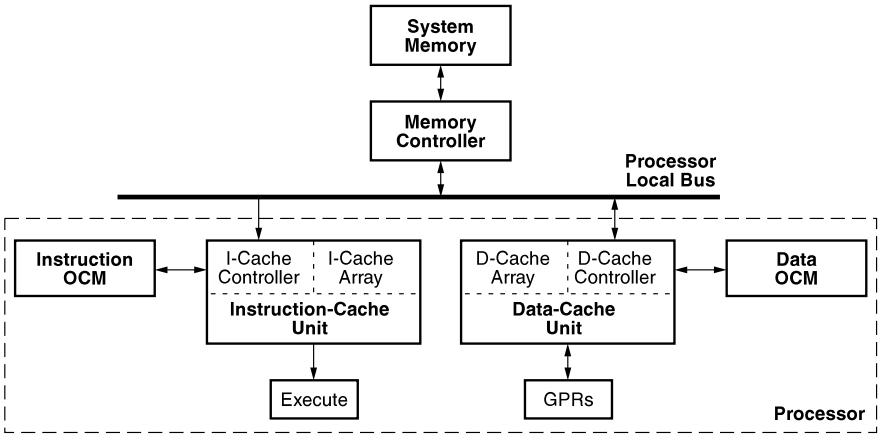


Figura 3.5: Organização das interfaces da memória do processador PPC405

Ficheiros do projeto XPS

Um projeto concebido e gerido pela ferramenta XPS é organizado numa pasta separada. O nome da pasta não tem significado e o seu conteúdo não é entendido para ser alterado manualmente. Nada impede efetuar a referida edição manual e em certos cenários esta pode ser útil e eficiente, no entanto, existe a possibilidade da corrupção do projeto, caso edição manual não apresentar consistência. De seguida, são enumerados ficheiros/pastas que constituem a pasta do projeto e que participam na definição do sistema.

- **system.xmp** - é o ficheiro XMP (*Xilinx Microprocessor Project*) que guarda a informação sobre o projeto XPS, sobre a sua constituição e respetivas opções. Trata-se de um ficheiro de texto que tem um formato simplista: cada sua linha define uma variável compreendida pelo XPS. O ficheiro define a versão da ferramenta, caminhos para bibliotecas, parâmetros do dispositivo FPGA, linguagem HDL preferida, configurações de simulação, caminhos e nomes para ficheiros que descrevem o sistema (.mhs, .mss, .ucf), processador, projetos software e respetivas opções e ficheiros fonte, bem como outras opções, como por exemplo o algoritmo de otimização do fluxo da implementação FPGA.

- **system.mhs** - este é o nome por defeito (pode ser alterado no system.xmp) para o ficheiro MHS (*Microprocessor Hardware Specification*). Este ficheiro define a vertente *hardware* do sistema, define a configuração dos barramentos, periféricos, processador(es), respetivas interligações e espaços de endereços. A Listagem 3.1 exemplifica a instanciação de um periférico no ficheiro .mhs efetuada automaticamente, como o resultado da utilização deste através da

interface gráfica do XPS. O periférico utilizado para o exemplo é o controlador das interrupções do *hardware microkernel* desenvolvido no âmbito desta dissertação. Para além de instanciar todos os componentes da forma apresentada na listagem, o ficheiro .mhs também declara as ligações externas. Estas devem ser mapeadas no ficheiro .ucf para os pinos I/O do dispositivo FPGA.

```
BEGIN soht_sic
 PARAMETER INSTANCE = soht_sic_0
 PARAMETER HW_VER = 1.00.a
 PARAMETER SOHT_EXT_INT_CONFIG = 2244
 PARAMETER SOHT_TASK_NUMBER = 16
 PARAMETER C_BASEADDR = 0xca000000
 PARAMETER C_HIGHADDR = 0xca00ffff
 BUS_INTERFACE SPLB = plb0
 BUS_INTERFACE MFSL = fsl_v20_0
 PORT t_valid_bit = prioritizer_n_addr_bfm_0_t_valid_bit
 PORT t_enable_bit = prioritizer_n_addr_bfm_0_t_enable_bit
 PORT irq_array = PB_UP&PB_RIGHT&PB_DOWN&PB_LEFT
END
```

Listagem 3.1: Exemplo da instanciação e configuração de um componente no ficheiro .mhs

- **system.mss** - este é o nome por defeito (pode ser alterado no system.xmp) para o ficheiro MSS (*Microprocessor Software Specification*). Este ficheiro define a vertente *software* do sistema, contem as diretivas para customização do sistema operativo, bibliotecas e *drivers* que compreendem o sistema. A Listagem 3.2 exemplifica a definição dos *drivers* para o processador e para o periférico referido no item anterior.

```
BEGIN PROCESSOR
 PARAMETER DRIVER_NAME = cpu_ppc405
 PARAMETER DRIVER_VER = 1.10.b
 PARAMETER HW_INSTANCE = ppc405_0
 PARAMETER COMPILER = powerpc-eabi-gcc
 PARAMETER ARCHIVER = powerpc-eabi-ar
 PARAMETER CORE_CLOCK_FREQ_HZ = 100000000
END
BEGIN DRIVER
 PARAMETER DRIVER_NAME = soht_sic
 PARAMETER DRIVER_VER = 1.00.a
 PARAMETER HW_INSTANCE = soht_sic_0
END
```

Listagem 3.2: Exemplo da definição dos *drivers* para o processador e um periférico no ficheiro .mss

- **data/system.ucf** - é a localização e nome por defeito (podem ser alterados no system.xmp) do ficheiro UCF (*User Constraints File*). Neste ficheiro é feito o mapeamento das ligações externas, definidas no ficheiro .mhs, aos pinos I/O do dispositivo FPGA, bem como a associação das respetivas restrições lógicas. A Listagem 3.3 exemplifica a parte do ficheiro .ucf correspondente às ligações externas instanciadas no controlador das interrupções do *hardware microkernel* desenvolvido no âmbito desta dissertação.

```
#### PUSH BUTTONS
NET "PB_UP" LOC = "AH4";
NET "PB_DOWN" LOC = "AG3";
NET "PB_LEFT" LOC = "AH1";
NET "PB_RIGHT" LOC = "AH2";

NET "PB_UP" IOSTANDARD = LVTTL;
NET "PB_DOWN" IOSTANDARD = LVTTL;
NET "PB_LEFT" IOSTANDARD = LVTTL;
NET "PB_RIGHT" IOSTANDARD = LVTTL;
```

Listagem 3.3: Exemplo do mapeamento das ligações externas aos pinos I/O do dispositivo FPGA e da associação das respetivas restrições lógicas (ficheiro .ucf)

- **SDK_projects** - este é o nome da pasta que é criada automaticamente aquando utilização do SDK (*Software Development Kit*) a partir do projeto XPS. A pasta inclui ficheiros de configuração utilizados pelas ferramentas do *download* da imagem para a plataforma, ficheiros que constituem *board support package* do SoC instanciado a partir do XPS, inclusivamente os ficheiros de saída da ferramenta LibGen (*Library Generator*), bem como os próprios projetos SDK.
- **pcores, drivers** - são os nomes de duas pastas criadas quando durante a criação de um periférico através da ferramenta CIP (*Create and Import Peripheral*) é escolhida a opção do armazenamento do *core* IP resultante de forma local no próprio projeto XPS. Outra opção consiste no armazenamento do *core* IP resultante no repositório do EDK, tornando possível a sua instanciação a partir de múltiplos projetos XPS.

Interface gráfica do XPS

A ferramenta *Xilinx Platform Studio* é um ambiente integrado de desenvolvimento (IDE) gráfico com suporte de linha de comandos. A interface gráfica facilita a manutenção e a interpretação dos ficheiros que compreendem o sistema, permite

construir e configurar SoC completo de uma forma eficiente e automatizada, poupando o esforço do desenvolvedor e potencializando a utilização da tecnologia. Esta secção foca nas funcionalidades suportadas pela interface gráfica que são relacionadas com a construção e configuração do SoC.

É aconselhado utilizar a ferramenta BSB *Base System Builder*, apresentada na secção 3.2.2, sempre que é concebido um novo sistema. O resultado da utilização da ferramenta BSB consiste num sistema funcional pronto para ser utilizado como um sistema embebido ou como um ponto de partida para uma customização mais avançada.

A principal janela da interface gráfica é denominada *System Assembly View*. O sistema pode ser consultado e personalizado em três representações, três pontos de vista: *Bus Interfaces*, *Ports* e *Adresses*. Em cada destas representações o sistema apresenta-se como a lista completa de componentes que o constituem. Os componentes integrados no sistema podem provir do catálogo dos *cores* IP *Xilinx Embedded IP* ou podem ser *cores* personalizados do utilizador. Em ambos casos, a implementação do componente é acompanhada pelo ficheiro MPD (*Microprocessor Peripheral Definition*) que define a existência no componente dos parâmetros configuráveis, das interfaces para com barramentos, bem como das saídas e das entradas. A informação contida nestes ficheiros é refletida no aspeto e nas funcionalidades da interface gráfica (janela *System Assembly View*) disponíveis para cada um destes componentes.

A Figura 3.6 mostra a representação *Bus Interfaces* do *System Assembly View* de um projeto XPS baseado no *hardware microkernel* concebido no âmbito desta dissertação. No conjunto dos componentes que constitui este projeto existem as seguintes interfaces: OCM, PLB, FSL e portas das unidades da memória BRAM. Do lado esquerda da representação, encontra-se a mapa do estado das ligações. Os símbolos de forma retangular representam as interfaces *master* PLB, OCM ou FSL, enquanto os símbolos em forma de círculo - as respetivas interfaces *slave*. Com recurso a esta representação gráfica do sistema, a configuração das interligações das interfaces torna-se intuitiva e fácil.

A Figura 3.7 mostra a representação *Ports* do *System Assembly View* do mesmo projeto XPS. Como pode ser observado, nesta representação é possível interligar as saídas e as entradas dos componentes, bem como declarar as ligações externas. Para cada *port* do cada componente a ferramenta visualiza a sua direção e a sua largura.

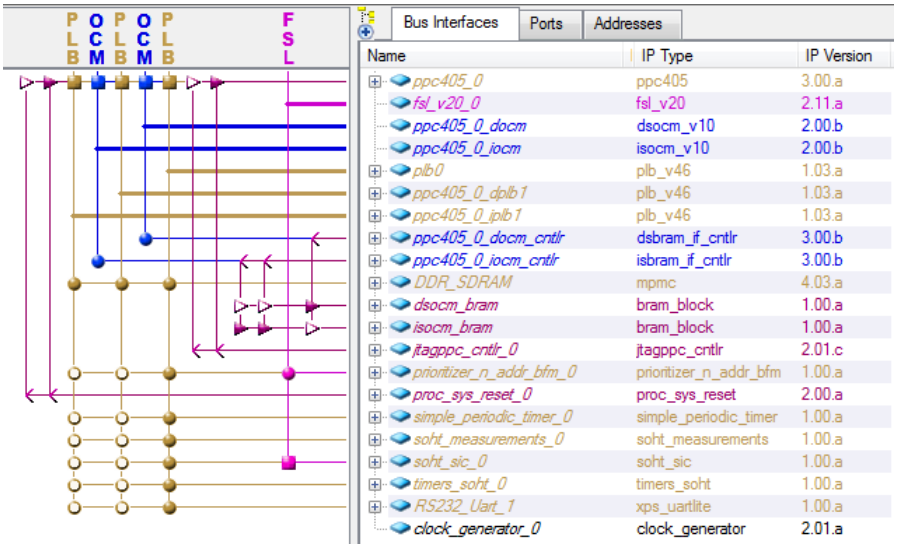


Figura 3.6: Representação *Bus Interfaces* do *System Assembly View* de um projeto XPS baseado no *hardware microkernel* concebido no âmbito desta dissertação

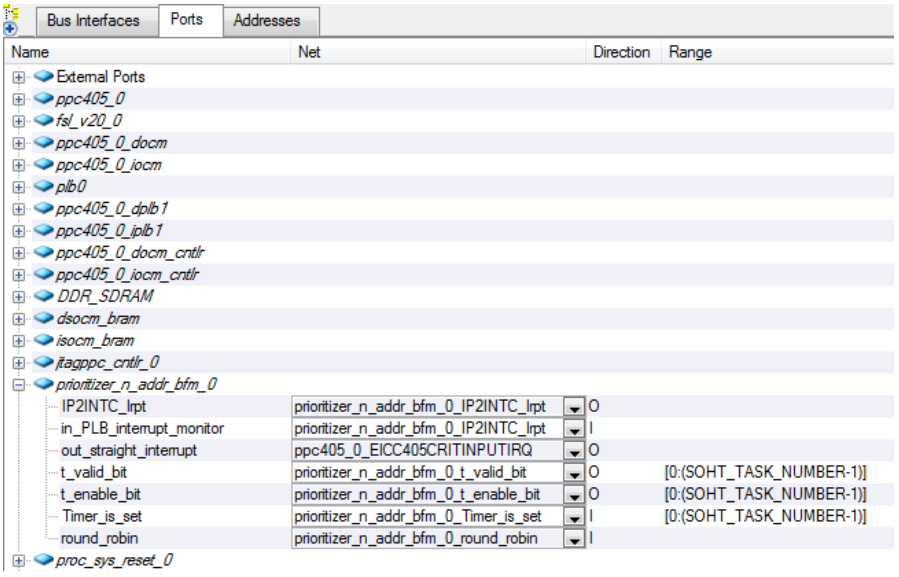
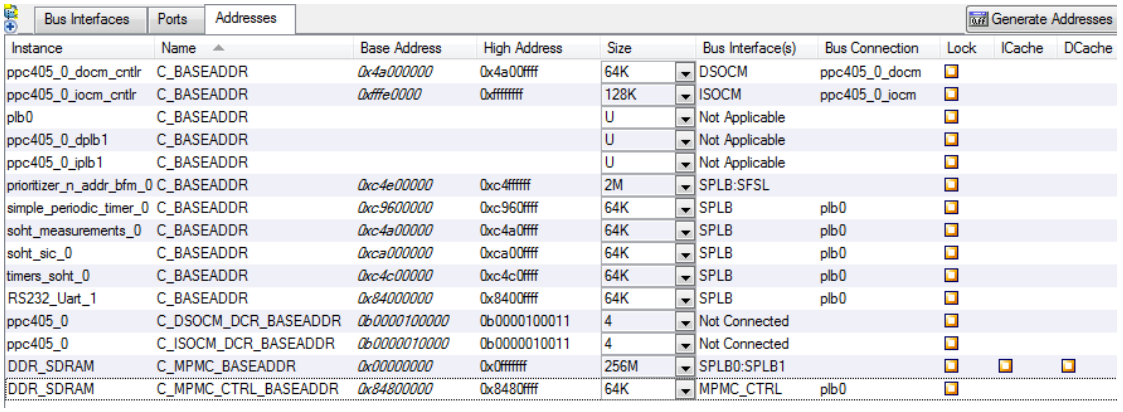


Figura 3.7: Representação *Ports* do *System Assembly View* de um projeto XPS baseado no *hardware microkernel* concebido no âmbito desta dissertação

A Figura 3.8 mostra a representação *Adresses* do *System Assembly View* do mesmo projeto XPS. Como pode ser observado, nesta representação são definidos os tamanhos do espaço de endereços da memória para cada um dos componentes. É aconselhável a utilização do *wizard Generate Adresses*, em vez da definição manual das margens dos espaços de endereços. No caso de definir manualmente um intervalo especifico para certo componente, é possível fixar respetivos valores recorrendo a funcionalidade *Lock*. Desta forma, a ferramenta *Generate Adresses* não vai gerar respetivo espaço mas vai considera-lo durante a geração dos restantes.



Instance	Name	Base Address	High Address	Size	Bus Interface(s)	Bus Connection	Lock	ICache	DCache
ppc405_0_docm_cntlr	C_BASEADDR	0x4a000000	0x4a00ffff	64K	DSOCM	ppc405_0_docm			
ppc405_0_iocm_cntlr	C_BASEADDR	0xfffe0000	0xffffffff	128K	ISOCM	ppc405_0_iocm			
plb0	C_BASEADDR			U	Not Applicable				
ppc405_0_dplb1	C_BASEADDR			U	Not Applicable				
ppc405_0_iplb1	C_BASEADDR			U	Not Applicable				
prioritizer_n_addr_bfm_0	C_BASEADDR	0xc4e00000	0xc4ffffff	2M	SPLB:SFSL				
simple_periodic_timer_0	C_BASEADDR	0xc9600000	0xc960ffff	64K	SPLB	plb0			
soht_measurements_0	C_BASEADDR	0xc4a00000	0xc4a0ffff	64K	SPLB	plb0			
soht_sic_0	C_BASEADDR	0xca000000	0xca00ffff	64K	SPLB	plb0			
timers_soht_0	C_BASEADDR	0xc4c00000	0xc4c0ffff	64K	SPLB	plb0			
RS232_Uart_1	C_BASEADDR	0x84000000	0x8400ffff	64K	SPLB	plb0			
ppc405_0	C_DSOCM_DCR_BASEADDR	0b0000100000	0b0000100011	4	Not Connected				
ppc405_0	C_ISOCM_DCR_BASEADDR	0b0000010000	0b0000010011	4	Not Connected				
DDR_SDRAM	C_MPMC_BASEADDR	0x00000000	0x0fffffff	256M	SPLB0:SPLB1				
DDR_SDRAM	C_MPMC_CTRL_BASEADDR	0x84800000	0x8480ffff	64K	MPMC_CTRL	plb0			

Figura 3.8: Representação *Adresses* do *System Assembly View* de um projeto XPS baseado no *hardware microkernel* concebido no âmbito desta dissertação

As configurações feitas através do *System Assembly View* definem os ficheiros MSS e MHS. O ficheiro UCF é gerado automaticamente pelo BSB (*Base System Builder*). A sua posterior customização deve ser feita manualmente. A documentação que acompanha a plataforma do desenvolvimento deve possuir um conjunto dos ficheiros .ucf que definem as localizações e restrições lógicas associadas aos pinos I/O do dispositivo FPGA que se encontram ligados aos periféricos, que rodeiam o dispositivo, ou aos conectores de expansão. Estes ficheiros podem ser usados como *templates* durante a customização do ficheiro .ucf do projeto XPS.

3.3 Conclusões

A plataforma de desenvolvimento XUPV2 do produtor Xilinx, utilizada nesta dissertação, é dotada com um dispositivo FPGA que se enquadra na definição da tecnologia SoPC (*System-on-a-Programmable-Chip*). Este integra a área reprogramável FPGA junto com os microprocessadores embebidos e respetivos periféricos.

A plataforma de desenvolvimento possui as capacidades necessárias para integração de um conjunto de coprocessadores customizados junto com um microprocessador embebido e que irão dar assistência em *hardware* ao *kernel*, cuja execução irá decorrer no referido microprocessador.

A plataforma é integrada no ambiente do desenvolvimento Xilinx EDK (*Embedded Development Kit*) 10.1. Neste capítulo foi feita a apresentação das ferramentas que o constituem do ponto de vista geral, bem como do ponto de vista da sua utilização nesta dissertação.

Capítulo 4

Sistema Operativo de Suporte

Este capítulo apresenta o sistema operativo de suporte que serviu de base para implementação do sistema operativo híbrido, bem como o sistema operativo considerado para futura integração do *hardware microkernel* implementado nesta dissertação. Na secção final do capítulo são apresentadas as questões relacionadas com *porting* do sistema, *upgrades* efetuados ao sistema, interface entre o sistema e *hardware microkernel* desenvolvido e que é apresentado em detalhe no Capítulo 5.

4.1 ADEOS

No ano 1999 foi lançado o livro "Programming Embedded Systems in C and C++" da autoria do Michael Barr (Barr, 1999). A motivação para a sua escrita consistia na latente necessidade de livros que cobrem o tema de particularidades do desenvolvimento de *software* para sistemas embebidos. Visto que, como já foi referido no Capítulo 2, todos, menos os mais triviais sistemas embebidos, beneficiam da inclusão do sistema operativo, este é um dos temas cobertas pelo livro. Os conceitos fundamentais de um sistema operativo embebido são apresentados com base na análise do código fonte de um *kernel* simplista escrito pelo autor. Esta opção é argumentada com a maior eficácia didática e com o intuito de desvendar a complexidade insuportável, que por norma se associa ao sistema operativo, aquando na mente de um engenheiro de *software* inexperiente na matéria. É no mesmo contexto que este sistema operativo se enquadra no âmbito da presente dissertação. O sistema operativo é denominado ADEOS, que é um acrónimo para *"A Decent Embedded Operating System"*. É resumido em menos de 1000 linhas de

código, três-quartos dos quais são escritas em linguagem C++ e não têm dependência da plataforma, sobrando um-quarto para o código dependente escrito para o processador Intel 16-*bit* 80x86.

Nas próximas secções será feita uma descrição das funcionalidades, características e da respetiva organização do sistema operativo ADEOS.

4.1.1 Visão global sobre organização

Como já foi referido no Capítulo 2, uma das principais funcionalidades do sistema operativo é proporcionar um ambiente com múltiplas unidades de execução e efetuar o escalonamento do acesso destas ao CPU. Na maioria dos casos, ditas unidades de execução são denominadas como tarefas, mesmo se aplica ao ADEOS. O sistema operativo implementa mínimo das funcionalidades. As tarefas são ordenadas no seu acesso ao CPU com base na prioridade. O escalonador garante que a qualquer momento seja executada tarefa com a maior prioridade, desde que pronta para execução. A prioridade é assinada pelo utilizador e não é alterada dinamicamente pelo escalonador. Por outras palavras, ADEOS implementa a estratégia do escalonamento do tipo *fixed-priority preemptive*. Ao nível de mecanismos de sincronização de tarefas é implementado *mutex* simplista. Uma outra forma de obter visão global sobre organização do sistema operativo passa pelo análise do diagrama de classes apresentado na Figura 4.1.

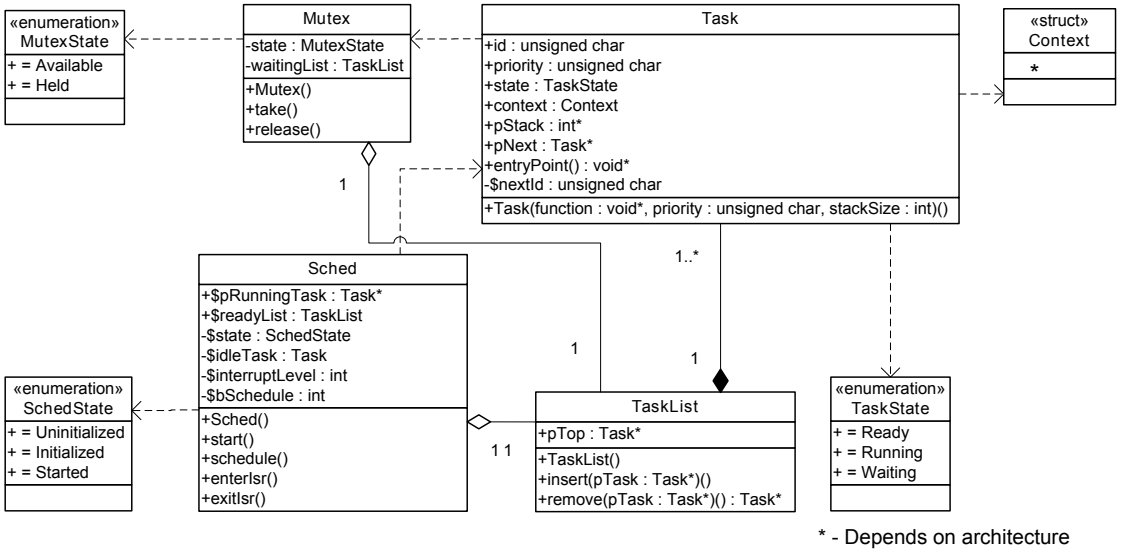


Figura 4.1: Diagrama de classes do sistema operativo ADEOS

É possível então distinguir três recursos, que juntos constituem sistema operativo ADEOS: conjunto de tarefas (objetos da classe *Task*), conjunto de objetos de sincronização *mutex* (objetos da classe *Mutex*), escalonador (objeto da classe *Sched*). As próximas secções irão dar uma breve descrição do propósito e implementação de cada um destes três recursos.

4.1.2 Tarefas

A execução intercalada de múltiplas tarefas implica a necessidade de virtualização dos recursos do sistema. O sistema operativo deve salvaguardar o estado do sistema em que cada uma das tarefas deixa a execução e restaura-lo devidamente aquando retorno da tarefa a execução, garantindo desta forma a consistência da execução de cada uma das tarefas. A informação que deve ser guardada para cada tarefa é denominada contexto da tarefa e consiste no estado do processador imediatamente antes ao ato da preempção. Para além do contexto, o sistema operativo necessita também de certa quantia de informação adicional necessária para gestão de tarefas.

A definição da classe Task se encontra na Listagem 4.1. Segue-se uma breve descrição dos atributos da classe *Task*.

```
typedef unsigned char TaskId;
typedef unsigned char Priority;
enum TaskState { Ready, Running, Waiting };

class Task
{
  public:

    Task(void (*function)(), Priority, int stackSize);

    TaskId         id;
    Context        context;
    int *          pStack;
    Task *         pNext;
    Priority       priority;
    TaskState      state;

    void  (*entryPoint)();

  private:

    static TaskId  nextId;
};
```

Listagem 4.1: Definição da classe *Task*

O *id* da tarefa é do tipo *unsigned char* (numero inteiro no intervalo entre 0 e 255) e é utilizado para identificação das tarefas. Identificadores são assinados pelo

construtor da classe. A tarefa *idle* é sempre criada em primeiro lugar (atributo estático da classe *Sched*), sendo lhe atribuído identificador 0. A partir daí, são atribuídos números sucessivos, com auxílio do atributo estático *nextId* manuseado pelo construtor da classe.

O atributo *context* é o contexto da tarefa, ou seja, estado do processador que é guardado pelo sistema operativo para impor a consistência na execução da tarefa aquando sua preempção e seu regresso a execução. A estrutura de dados que compreende o tipo deste atributo é dependente da arquitetura do processador que é 16-*bit* 80x86 no caso da versão original do ADEOS.

O atributo *pStack* é o apontador para a pilha da tarefa, pois, num ambiente de execução preemptivo, cada unidade de execução necessita da própria pilha. O atributo *pNext* é um simples apontador para objeto da mesma classe que é utilizado na construção da lista ligada com objetos desta classe.

O estado da tarefa (atributo *state*) é manuseado pelo escalonador e é um agente fundamental no processo do escalonamento. A partir do *enum TaskState* revela-se que a tarefa pode encontrar-se num dos três estados: *Ready* (pronta para execução), *Running* (a executar) e *Waiting* (a espera). O diagrama que apresenta possíveis transições do estado da tarefa do sistema operativo ADEOS pode ser visto na Figura 4.2. No ato da criação da tarefa o estado desta é inicializado como *Ready*.

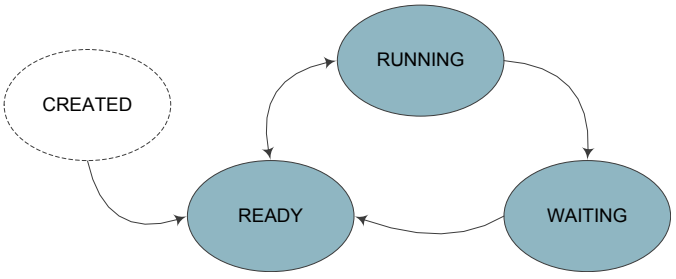


Figura 4.2: Transições do estado da tarefa no sistema operativo ADEOS

Apenas o escalonador pode promover a tarefa para execução (estado *Running*) e só a partir do estado *Ready*. O sistema operativo ADEOS é compreendido para arquiteturas que apresentam único CPU, ou seja, apenas uma tarefa se encontra no estado *Running* a qualquer momento. Após alcançar o estado *Running* a tarefa executa até a sua conclusão ou até o escalonador promover para execução uma outra tarefa, ou até surgir necessidade de esperar por algum evento antes de continuar a execução. O estado da tarefa transita do *Running* para *Waiting* no caso do insucesso da tentativa de adquirir *mutex*. Quando o recurso responsável pelo bloqueio da tarefa for libertado, esta passará para o estado *Ready*, sendo a partir daí, a

retoma da sua execução torna-se possível quando o escalonador reconhece-la como a tarefa mais prioritária das prontas para execução, promovendo-a para o estado *Running*. No caso particular do sistema operativo ADEOS não existem estados auxiliares que sinalizam que uma tarefa já terminou e não deve ser considerada pelo escalonador. Isto deve-se ao facto que as funções associadas às tarefas são chamadas a partir de uma função especial *run()* que acaba por apagar totalmente a tarefa quando a respetiva função associada retornar. Sendo assim, uma tarefa no sistema operativo ADEOS ou está num dos três estados (*Ready*, *Running* ou *Waiting*), ou não existe, pois é apagada automaticamente pelos internos do *kernel*, logo que conclui a sua execução.

Como já foi referido, o algoritmo do escalonamento em causa é baseado nas prioridades. O atributo *priority* pode tomar valores inteiros no intervalo de 0 a 255. Um numero mais elevado corresponde à maior prioridade e a prioridade da tarefa *idle* é igual ao 0.

Por último, o apontador para função *entryPoint* é utilizado pelos internos do sistema operativo para a chamada da função associada à tarefa, a partir da função especial *run()*.

Tudo acima referido pode ser também refletido na análise do código do construtor da classe *Task*, apresentado na Listagem 4.2. O construtor recebe apenas três argumentos: apontador para a tarefa associada *function* que é atribuído ao atributo *entryPoint*, prioridade escolhida pelo utilizador *p* que é atribuída ao atributo *priority* e o tamanho em *bytes* da pilha associada à tarefa. Como já foi referido, o construtor manuseia identificadores das tarefas com auxílio do atributo estático *nextId*, de forma a atribuir números sucessivos às tarefas criadas. O estado inicial de qualquer tarefa criada é *Ready*. Com base no tamanho escolhido pelo utilizador é alocada a memória para a pilha da tarefa. Através da função C *contextInit* implementada em *assembly* decorre a inicialização do contexto da tarefa, que consiste na preparação da pilha, de forma que quando esta tarefa assumir a execução, sendo figurante da rotina *contextSwitch()*, a consistência será assegurada e a execução começará a partir da primeira instrução da função especial *run()*. Após inicialização do contexto, o objeto que está a ser inicializado (tarefa) é inserido na lista ligada das tarefas prontas para execução. Sendo assim, esta tarefa já se encontra visível para o escalonador e será considerada no próximo ponto de escalonamento, que é invocado logo a seguir, por meio do método *schedule()* da classe *Sched*.

Como já foi referido no Capítulo 2, quando uma secção do código é denominada

```
TaskId    Task::nextId = 0;

Task::Task(void (*function)(), Priority p, int stackSize)
{
  stackSize /= sizeof(int);      // Convert bytes to words.
  enterCS();                  ////// Critical Section Begin
  //
  // Initialize the task-specific data.
  //
  id         = Task::nextId++;
  state      = Ready;
  priority   = p;
  entryPoint = function;
  pStack     = new int[stackSize];
  pNext      = NULL;
  //
  // Initialize the processor context.
  //
  contextInit(&context, run, this, pStack + stackSize);
  //
  // Insert the task into the ready list.
  //
  os.readyList.insert(this);

  os.schedule();                 // Scheduling Point

  exitCS();                   ////// Critical Section End
}   /* Task() */
```

Listagem 4.2: Construtor da classe *Task*

como sendo crítica, tal significa, que a sua execução deve decorrer de forma atómica, isto é, sem ser interrompida. As macros *enterCS()* e *exitCS()* são definidas no ficheiro *bsp.h* e consistem no conjunto de instruções (dependente da arquitetura do processador) que permitem desabilitar e habilitar interrupções, respetivamente.

Uma particularidade do ADEOS, em comparação com muitos outros sistemas operativos embebidos, está na implementação da função especial *run()*, responsável pela chamada da função associada à tarefa e a eliminação da tarefa caso esta concluir a sua execução (retorno da função associada). Para compreender melhor o mecanismo envolvido, pode ser analisado o código presente na Listagem 4.3. Quando uma tarefa é escolhida pelo escalonador para aceder CPU pela primeira vez, o contexto desta, previamente inicializado no construtor, é tal, que a execução começa dentro da função especial *run()*. O primeiro *statement* desta função chama a função associada a tarefa em causa. Quando a função associada retornar, o fluxo de execução voltará à função *run()*. Esta remove a respetiva tarefa da lista das prontas para execução, marca o apontador global para tarefa em execução como NULL (assim na rotina *contexSwitch()* não é salvaguardado o contexto da tarefa cuja pilha acabou de ser desalocada), desaloca a memória da pilha e invoca explicitamente o escalonador. Desta forma, as funções associadas às tarefas no ADEOS podem retornar e os recursos associados a estas são desalocados automaticamente,

```
void run(Task * pTask)
{
  //
  // Start the task, by executing the associated function.
  //
  pTask->entryPoint();
  enterCS();                  ////// Critical Section Begin
  //
  // Remove this task from the scheduler's data structures.
  //
  os.readyList.remove(pTask);
  os.pRunningTask = NULL;
  //
  // Free the task's stack space.
  //
  delete pTask->pStack;
  os.schedule();                  // Scheduling Point
  // This line will never be reached.
}   /* run() */
```

Listagem 4.3: Função especial *run()*

ao contrário da maioria dos sistemas operativos embebidos, que ditam que as suas tarefas devem ter um *loop* infinito dentro do corpo das respetivas funções.

4.1.3 Listas de Tarefas

Para além das três classes principais que constituem ADEOS (*Task*, *Sched* e *Mutex*), existe uma outra classe denominada *TaskList* que é apenas uma lista ligada dos objetos do tipo tarefa (*Task*), e que é ordenada pela prioridade destas. A classe apresenta apenas dois métodos (*insert* e *remove*), um dos quais já foi mencionado aquando análise do código do construtor da classe *Task*. O método *insert(Task * pTask)* insere a tarefa passada como argumento na lista ligada ordenada de acordo com a prioridade, enquanto o *remove(Task * pTask)* remove da lista a tarefa passada como argumento e devolve apontador para esta em caso do sucesso e *NULL* caso tarefa não ser encontrada.

As tarefas que estão no estado *Ready* e a tarefa que está no estado *Running*, juntas, compõem a lista ligada *readyList* (atributo estático da classe *Sched*), sendo a tarefa *Running* no topo da lista.

Visto que, a única causa do bloqueio de uma tarefa no sitema operativo ADEOS são recursos de sincronização *mutex*, todas as restantes tarefas (estado *Waiting*) se encontram nas listas associadas a estes, sendo que, cada um destes tem uma própria lista ligada *waitingList* de tarefas bloqueadas.

Para concluir esta subsecção segue-se a Figura 4.3 que ilustra uma lista ligada das

tarefas prontas para execução. O exemplo apresentado foi retirado do capítulo 8 do (Barr, 1999). A tarefa que se encontra no fim da lista é denominada *idle*, esta é criada no ato da criação do objeto da classe *Sched* (escalonador do sistema) e o seu identificador é 0, bem como a sua prioridade. A função associada a tarefa *idle* é definida no ficheiro *bsp.h* e é implementada em *assembly* no ficheiro *bsp.asm* resumindo-se a um simples *loop* infinito. Sendo assim, a tarefa *idle* nunca vai retornar ou bloquear e sempre estará no fim da lista das tarefas prontas para execução, garantindo desta forma, que o mecanismo da comutação de contexto não perderá consistência mesmo sem nenhuma tarefa do utilizador pronta para execução.

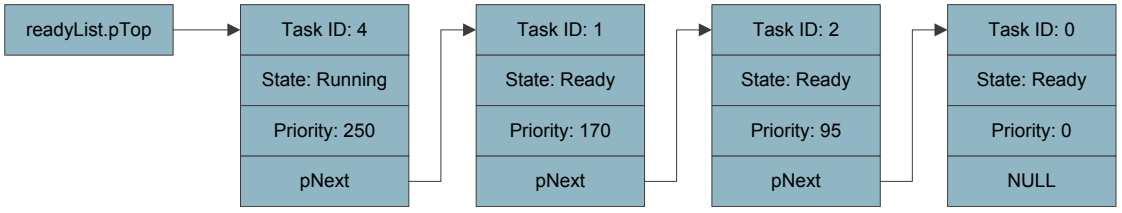


Figura 4.3: Lista ligada das tarefas prontas para execução

4.1.4 Escalonador e Pontos de Escalonamento

O "coração" de qualquer sistema operativo é o seu escalonador. Em traços gerais, este gere a informação das tarefas que constituem o ambiente de execução e escalona o seu acesso ao CPU (ou CPUs) de acordo com uma estratégia de escalonamento (ou um conjunto destas). No caso do sistema operativo ADEOS, a estratégia usada é baseada em prioridades e preempção é suportada. Nos pontos de escalonamento o escalonador sempre escolha para execução a tarefa mais prioritária das prontas para execução. No caso da existência de varias tarefas com a mesma prioridade é escolhida aquela que entrou mais cedo na lista das tarefas prontas para execução (no método inserir da lista ligada tarefa é inserida sempre "depois" da ultima das aquelas com a prioridade maior ou igual). Sendo assim, pode ser afirmado que o sistema operativo ADEOS implementa FIFO (*First in, first out*) como a estratégia do desempate de duas ou mais tarefas com a mesma prioridade. A tarefa em execução pode ser preemptida pelo escalonador caso deixar de ser mais prioritária, ou seja, se entretanto surgir uma tarefa pronta para execução com uma prioridade maior. O escalonador é implementado como sendo objeto da classe *Sched* cuja definição pode ser vista na Listagem 4.4.

```
class Sched
{
  public:

    Sched();

    void  start();
    void  schedule();

    void  enterIsr();
    void  exitIsr();

    static Task *    pRunningTask;
    static TaskList  readyList;

    enum SchedState { Uninitialized, Initialized, Started };
  private:

    static SchedState  state;
    static Task        idleTask;
    static int         interruptLevel;
    static int         bSchedule;
};
extern Sched os;
```

Listagem 4.4: Definição da classe *Sched*

Todos os atributos da classe são estáticos, isto é, o valor destes é o mesmo para todas as instâncias dessa classe. Isto faz todo o sentido, visto que compreende-se existência de apenas um escalonador e a definição dos atributos da classe como sendo estáticos reforça esta compreensão. O escalonador pode encontrar-se num dos três estados: *Uninitialized*, *Initialized* e *Started*. O atributo *state* é utilizado para guardar o dito estado. O atributo *interruptLevel* indica o nível atual do aninhamento da interrupção, ou seja, um valor nulo indica que a execução se encontra num contexto normal, enquanto um valor não-nulo indica que a execução ocorre no contexto de uma interrupção, sendo que um valor acima de um indica que tinham ocorrido várias interrupções e cujo atendimento está a ser feito de forma aninhada. Este atributo é incrementado com recurso ao método *enterIsr()* e decrementado com recurso ao método *exitIsr()* aquando do atendimento de uma interrupção e da conclusão desta, respetivamente. O atributo *bSchedule* é utilizado para adiar um ponto de escalonamento que ocorreu enquanto o nível de aninhamento da interrupção era não-nulo para altura quando este voltar ao zero. O objeto escalonador possui também uma tarefa *idle* e uma lista ligada de tarefas prontas para execução (*readyList*) e ordenadas pela sua prioridade. É manuseado também o atributo *pRunningTask* que aponta para a tarefa em execução. Logicamente, este, a maior parte do tempo, aponta para o topo da *readyList*, no entanto, assume em certos casos o valor nulo e no próximo ponto de escalonamento, dentro da rotina *schedule()* isso faz com que o contexto da tarefa "atual" não é salvaguardado. Como já

foi referido, os métodos *enterIsr()* e *exitIsr()* são utilizados nos mecanismos relacionados com *interruptLevel* e *bSchedule*. O método *enterIsr()* apenas incrementa *interruptLevel*, já o método *exitIsr()* decrementa *interruptLevel* e caso este voltar ao zero verifica se houve um ponto de escalonamento adiado (*bSchedule* não-nulo), procedendo com escalonamento em caso afirmativo. Para concluir a análise de todos os atributos e métodos, com excepção do *schedule()*, pode ser consultado o código da Listagem 4.5.

```
Sched::SchedState  Sched::state = Uninitialized;
int        Sched::interruptLevel = 0;
int        Sched::bSchedule      = 0;

Task *     Sched::pRunningTask = NULL;
TaskList   Sched::readyList;

Sched      os;
Task       Sched::idleTask(idle, 0, 128);

Sched::Sched(void)
{
  if (state != Uninitialized) return;

  state = Initialized;
}   /* Sched() */
void Sched::start(void)
{
  if (state != Initialized) return;

  state = Started;

  schedule();           // Scheduling Point
  // This line will never be executed.
}   /* start() */
```

Listagem 4.5: Parte do ficheiro *sched.cpp*

O código da Listagem contém a inicialização de todos os atributos (estes são estáticos), bem como construtor e método *start()* que revelam a forma de manuseamento do atributo *state* e impõem a existência de apenas um objeto da classe *Sched* (denominadao *os*).

Ao longo da presente discussão da implementação do sistema operativo ADEOS, enumeras vezes foram mencionados pontos de escalonamento e a invocação do método *schedule()*. Este é o método chave do escalonador e a sua implementação pode ser vista na Listagem 4.6.

A comutação de contexto (preempção da tarefa em execução) deve tomar lugar quando o apontador para a tarefa que está atualmente em execução (*pRunningTask*) não coincide com o apontador para a tarefa mais prioritária das prontas para execução (*readyList.pTop*). Distinguem-se então três cenários: o ponto de

```
void Sched::schedule(void)
{
  Task *  pOldTask;
  Task *  pNewTask;

  if (state != Started) return;
  // Postpone rescheduling until interrupts are completed.
  if (interruptLevel != 0){
    bSchedule = 1;
    return;
  }
  // If there is a higher-priority ready task, switch to it.
  if (pRunningTask != readyList.pTop){
    pOldTask = pRunningTask;
    pNewTask = readyList.pTop;

    pNewTask->state = Running;
    pRunningTask = pNewTask;

    if (pOldTask == NULL){
      contextSwitch(NULL, &pNewTask->context);
    }
    else{
      pOldTask->state = Ready;
      contextSwitch(&pOldTask->context, &pNewTask->context);
    }
  }
}   /* schedule() */
```

Listagem 4.6: schedule(), método chave da classe *Sched*

escalonamento não terá lugar, haverá comutação de contexto com salvaguarda do contexto e haverá comutação de contexto sem salvaguarda deste (*pRunningTask* é nulo). A rotina *contextSwitch()* que recebe como argumentos os apontadores para antigo e novo contextos e efetua a salvaguarda e restauro destes, respetivamente, será discutida na secção 4.3.1.

Pontos de escalonamento no sistema operativo ADEOS ocorrem exclusivamente com a chamada explicita do método *schedule()* que é colocada no código do *kernel* após todas as operações que podem resultar na alteração do topo da lista ligada das tarefas prontas para execução. No caso quando a chamada teve efeito e foi verificada a alteração da tarefa mais prioritária das prontas para execução, ocorre a consequente chamada explicita do método *contextSwitch()*.

4.1.5 *Mutex*

Como o recurso de sincronização de tarefas, o sistema operativo ADEOS implementa a exclusão mútua numa forma simplista mas suficientemente eficaz. Os objetos deste mecanismo de sincronização são denominados *mutex* e podem ser vistos como uma *flag* binária, por norma associada a algum recurso partilhado por

conjunto de tarefas, que assegura que o dito recurso é acedido por apenas uma tarefa a qualquer momento. Antes de aceder um recurso partilhado, a tarefa tenta tomar o respetivo *mutex* e sucede, caso este estiver disponível, caso contrário, a tarefa bloqueia a espera de *mutex* ser libertado.

A definição da classe *Mutex*, bem como a implementação do respetivo construtor podem ser encontradas na Listagem 4.7.

```
class Mutex
{
  public:
    Mutex();
    void  take(void);
    void  release(void);

  private:
    TaskList  waitingList;
    enum { Available, Held } state;
};
Mutex::Mutex()
{
  enterCS(); //// Critical Section Begin

  state = Available;
  waitingList.pTop = NULL;

  exitCS();  //// Critical Section End
}   /* Mutex() */
```

Listagem 4.7: Definição da classe *Mutex*

Mutex pode encontrar-se num dos dois estados: disponível (*Available*) ou segurado (*Held*), sendo que é sempre disponível quando criado. Cada *mutex* tem uma própria lista das tarefas bloqueadas por este e que é vazia aquando da criação. O código que implementa o método *take()* pode ser visto na Listagem 4.8.

```
void Mutex::take(void)
{
  Task *  pCallingTask;
  enterCS();                  ////// Critical Section Begin
  if (state == Available){ // The mutex is available. Simply take it.
      state = Held;
      waitingList.pTop = NULL;
  }
  else{ // The mutex is taken. Add the calling task to the waiting list.
      pCallingTask = os.pRunningTask;
      pCallingTask->state = Waiting;
      os.readyList.remove(pCallingTask);
      waitingList.insert(pCallingTask);
      os.schedule();       // Scheduling Point
      // When the mutex is released, the caller begins executing here.
  }
  exitCS();                   ////// Critical Section End
}   /* take() */
```

Listagem 4.8: Implementação do método take()

Como já foi referido, quando *mutex* está disponível, *take()* resulta em apenas alteração do estado do *mutex* de *Available* para *Held*. Já quando *take()* é feito sobre um *mutex* que está segurado por alguma tarefa, a tarefa que chamou o método é retirada da lista ligada das tarefas prontas para execução e é inserida na lista da espera associada a este *mutex*. A chamada do método *schedule()* neste contexto sempre resultará na comutação efetiva de contexto.

É de notar que a versão original do código fonte apresenta um conjunto de *bugs* um dos quais se revela precisamente no cenário acima referido. O método *remove()* da classe *TaskList* deixa inalterado o apontador para a próxima tarefa da tarefa removida. O método *insert()* da classe *TaskList* também deixa inalterado o apontador para a próxima tarefa da tarefa inserida, quando se trata da inserção numa lista vazia. Tendo em conta estes dois fatos segue-se a análise da situação quando uma tarefa é bloqueada a espera de um *mutex*, sendo a *waitingList* deste por enquanto vazia. Em qualquer momento, a tarefa que chama *mutex()* tem pela menos uma tarefa com a prioridade menor (pela menos a tarefa *idle*) e que, sendo assim, encontra-se na lista ligada das prontas para execução na direção do apontador *pNext*. Ou seja, quando a tarefa é removida da *readyList* o seu apontador *pNext* não é nulo e aponta para a parte sucessiva da *readyList*. Ao inserir esta tarefa na *waitngList* do *mutex*, quando ultima ainda é vazia, o apontador *pNext* continua ter o valor não nulo e, basicamente, a lista *waitingList* acaba por conter a mesma sequencia das tarefas que a *readyList*. Por enquanto, não ocorreu qualquer erro, mas as sucessivas chamadas do método *release()* vão comprometer a consistência da *readyList* do escalonador, porque vai ocorrer a situação quando *release()* vai chegar a libertar as tarefas que não deviam estar na *waitingList* e vai coloca-las "de volta" na *readyList*. Isto, por sua vez, vai resultar na duplicação destas tarefas ponto de vista do escalonador (isto é na *readyList*). Este problema pode ser atacado através da modificação do método *insert()* ou atráves da atribuição do valor nulo ao apontador *pNext* entre passos *remove* e *insert* aquando da colocação de uma tarefa na *waitingList*.

O código que implementa o método *release()* pode ser encontrado na Listagem 4.9.

A chamada do método *release()* sobre um *mutex* disponível não tem qualquer efeito. Caso *mutex* esteja segurado por alguma tarefa, dois cenários são possíveis: sua lista da espera é vazia ou não. Caso existe tarefa ou tarefas bloqueadas pelo *mutex*, será libertada e colocada de volta para a *readyList* do escalonador a tarefa mais prioritária destas. Quando *release()* é invocado sobre um *mutex* que se en-

```
void Mutex::release(void)
{
    Task *  pWaitingTask;


    enterCS();                          ////// Critical Section Begins

    if (state == Held)
    {
        pWaitingTask = waitingList.pTop;

        if (pWaitingTask != NULL)
        {
            //
            // Wake the first task on the waiting list.
            //
            waitingList.pTop = pWaitingTask->pNext;
            pWaitingTask->state = Ready;
            os.readyList.insert(pWaitingTask);

            os.schedule();                  // Scheduling Point
        }
        else
        {
            state = Available;
        }
    }

    exitCS();                           ////// Critical Section End

}   /* release() */
```

Listagem 4.9: Implementação do método release()

contra no estado *Held* mas de momento não tem qualquer tarefa a sua espera este mudará o seu estado para *Available*.

4.2 FreeRTOS

Sistema operativo FreeRTOS do *"Real Time Engineers Ltd"* se apresenta como líder do mercado (entra no *top* 5) de sistemas operativos de tempo real *low-end*, com referência aos estudos anuais do mercado de soluções na área de sistemas embebidos, promovidos pela empresa de marketing UBM.

Ao nível de licenciamento são oferecidas três opções: FreeRTOS, OpenRTOS e SafeRTOS. FreeRTOS é livre e licenciado pela versão modificada da GNU GPL (*General Public License*), pode ser usado para propósitos comerciais, desde que as regras da GPL sejam cumpridas. OpenRTOS é a versão comercial do FreeRTOS que não tem qualquer referência ao GPL, oferece garantia e suporte técnico profissional. SafeRTOS é uma versão paga, derivada da principal, e cujo *refactoring* levou-a ao cumprimento de requisitos do padrão de segurança IEC 61508.

O *download* do código fonte pode ser feito a partir do *website* oficial do sistema operativo (FreeRTOS, a), que apresenta uma grande quantidade do material de apoio e documentação. O conteúdo desta secção baseia-se sobretudo no referido *website* (FreeRTOS, a), guia prática de uso do FreeRTOS (Barry, 2009) e a experiência do autor desta dissertação na integração de um *hardware microkernel* no FreeRTOS que, no entanto, ocorreu fora do âmbito desta dissertação.

4.2.1 Visão global sobre organização

O código fonte do sistema operativo FreeRTOS é estruturado de uma forma que potencia a sua portabilidade. Escrito predominantemente na linguagem de programação C, apresenta um conjunto de rotinas que devem ser reimplementadas na linguagem *assembly* específica da arquitetura alvo aquando do *porting*. De momento (versão 7.5.2), são suportadas 34 arquiteturas inclusivamente PPC405, no âmbito da sua presença nos dispositivos FPGA do produtor *Xilinx*.

O núcleo do sistema operativo é implementado em apenas três ficheiros *".c"*: *tasks.c*, *queue.c* e *list.c*, também existem ficheiros *croutine.c* e *timers.c*, usados quando são pretendidas funcionalidades de corotinas e temporizadores por *software*, respetivamente. Estes cinco ficheiros são comuns para todos os *portings* e não implementam o código dependente do processador. Para além desses ficheiros *".c"* a distribuição atual do FreeRTOS é constituída também por onze ficheiros cabeçalhos *".h"*, também comuns para todos os *portings* do sistema. A parte do código que é dependente do processador é implementada nos ficheiros *port.c* e *portmacro.h*. Cada *porting* tem a sua versão desses ficheiros, podendo conter também um conjunto dos ficheiros adicionais como é o caso, por exemplo, do *porting* para processador PPC405 que, para além desses dois ficheiros, contém também ficheiros *portasm.S* e *FPU_Macros.h*. O sistema operativo FreeRTOS é configurável e escalável. Para customização do FreeRTOS para os propósitos da aplicação é usado ficheiro da configuração *FreeRTOSConfig.h*. Cada aplicação baseada neste sistema operativo deve conter o seu ficheiro *FreeRTOSConfig.h* no caminho conhecido pelo pré-processador. Na Listagem 4.10 está apresentado o ficheiro *FreeRTOSConfig.h* da aplicação *Demo*, distribuída junto com o núcleo do sistema operativo, e que corresponde ao *porting* para processador PPC405.

Uma breve análise do código apresentado é suficiente para ganhar uma noção acerca da granularidade da configuração do sistema. A configuração ocorre no tempo da compilação e é implementada com recurso à técnica da compilação con-

```
#ifndef FREERTOS_CONFIG_H
#define FREERTOS_CONFIG_H

/*-----------------------------------------------------------
 * Application specific definitions.
 *
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *
 * See http://www.freertos.org/a00110.html.
 *----------------------------------------------------------*/

#define configUSE_PREEMPTION            1
#define configUSE_IDLE_HOOK             0
#define configUSE_TICK_HOOK             0
#define configMINIMAL_STACK_SIZE        ( ( unsigned short ) 250 )
/* Clock setup from start.asm in the demo application. */
#define configCPU_CLOCK_HZ              ( ( unsigned long ) 200000000 )
#define configTICK_RATE_HZ              ( (portTickType) 1000 )
#define configMAX_PRIORITIES            ( ( unsigned portBASE_TYPE ) 6 )
#define configTOTAL_HEAP_SIZE           ( (size_t) (80 * 1024) )
#define configMAX_TASK_NAME_LEN         ( 20 )
#define configUSE_16_BIT_TICKS          0
#define configIDLE_SHOULD_YIELD         1
#define configUSE_MUTEXES               1
#define configUSE_TRACE_FACILITY        0
#define configCHECK_FOR_STACK_OVERFLOW  2
#define configUSE_COUNTING_SEMAPHORES   1
#define configUSE_APPLICATION_TASK_TAG  1
#define configUSE_FPU                   1


/* Co-routine definitions. */
#define configUSE_CO_ROUTINES           0
#define configMAX_CO_ROUTINE_PRIORITIES ( 4 )

/* Set the following definitions to 1 to include the API function, or zero
to exclude the API function. */
#define INCLUDE_vTaskPrioritySet            1
#define INCLUDE_uxTaskPriorityGet           1
#define INCLUDE_vTaskDelete                 1
#define INCLUDE_vTaskCleanUpResources       1
#define INCLUDE_vTaskSuspend                1
#define INCLUDE_vResumeFromISR              1
#define INCLUDE_vTaskDelayUntil             1
#define INCLUDE_vTaskDelay                  1
#define INCLUDE_xTaskGetSchedulerState      1
#define INCLUDE_xTaskGetCurrentTaskHandle   1
#define INCLUDE_uxTaskGetStackHighWaterMark 1
#define configUSE_RECURSIVE_MUTEXES         1


#if configUSE_FPU == 1
    /* Include the header that define the traceTASK_SWITCHED_IN() and
    traceTASK_SWITCHED_OUT() macros to save and restore the floating
    point registers for tasks that have requested this behaviour. */
    #include "FPU_Macros.h"
#endif

#endif /* FREERTOS_CONFIG_H */
```

Listagem 4.10: Ficheiro *FreeRTOSConfig.h*

dicional que usa diretivas *#if*, *#ifdef*, *#ifndef*, *#else*, *#elif* e *#endif*. Para uma visão mais completa sobre as funcionalidades configuráveis, deve ser consultado o ficheiro *FreeRTOS.h*, que faz parte do núcleo do sistema e que define com o valor predefinido os parâmetros que foram omitidos pelo utilizador no ficheiro da configuração ou interrompe compilação, com emissão da mensagem de erro (diretiva *#error*), para o caso dos parâmetros que não podem ser omitidos. Para finalizar a visão global sobre organização do FreeRTOS, ao nível dos ficheiros que o constituem, falta referir apenas que a gestão do *heap* é um ponto que é parcialmente dependente da arquitetura em causa. Por um lado, a biblioteca de funções *standard C* implementa para este propósito as funções *malloc()* e *free()*; por outro lado, ditas funções não são adequadas para o âmbito dos sistemas embebidos, pois nem sempre são presentes, apresentam indeterminismo, não apresentam segurança de *thread* e são dispendiosas em termos da memória do código. Dito isto, a implementação customizada da gestão da memória é um cenário comum nos sistemas embebidos. É por isso que a distribuição do FreeRTOS coloca a respetiva implementação na camada portável, fornecendo para o uso quatro modelos exemplo de implementação da dita gestão, colocando os nos ficheiros *heap_1.c*, *heap_2.c*, *heap_3.c* e *heap_4.c*, respetivamente. As implementações variam na sua complexidade e nos respetivos prós e contras. É apenas um ficheiro que deve ser escolhido para o uso em cada caso específico, podendo ser implementada uma outra solução para contemplar da melhor forma os requisitos impostos pela aplicação.

No que diz respeito ao padrão de codificação, o código comum que constitui o núcleo do FreeRTOS é apresentado como sendo em conformidade com padrão MISRA (*Motor Industry Software Reliability Association*) com quatro divergências deste, enumeradas em (Barry). Em termos do estilo de codificação, são usadas convenções de nomes que facilitam a orientação no código fonte. Os prefixos dos nomes das variáveis indicam o seu tipo, enquanto das funções - o tipo do retorno. As macros e as funções têm também no seu prefixo o nome do ficheiro no qual são definidas.

Ao contrario do exemplo didático, que é ADEOS apresentado na secção 4.1, sistema operativo FreeRTOS proporciona ao utilizador uma vasta gama de funcionalidades e ferramentas adicionais como, por exemplo, ferramentas de *trace*, úteis na fase de desenvolvimento e depuração, com elevada importância no contexto de cumprimento de requisitos do tempo real. Visto que, no âmbito desta dissertação pretende-se desenvolver *hardware microkernel* que implementa funcionalidades e recursos básicos de um sistema operativo do tempo real, nas próximas secções será

dado o foco apenas às funcionalidades fundamentais, como aquelas que compreendem o sistema operativo ADEOS.

4.2.2 Tarefas e a sua gestão

Na maioria dos casos, num ambiente de múltiplas unidades de execução, estas são denominadas como tarefas. Esta afirmação continua ser válida, da mesma forma como no caso do ADEOS. O sistema operativo FreeRTOS implementa esta abstração numa forma semelhante ao ADEOS mas que, no entanto, tem as suas particularidades.

As funções associadas às tarefas são na mesma implementadas como sendo simples funções C, mas o seu protótipo deve ser obrigatoriamente declarado para retornar *void* e receber como único argumento o apontador para o *void*. Ao contrário da implementação do ADEOS, que, com recurso ao uso da função especial *run()*, encoraja o retorno da função associada à tarefa quando esta concluiu a necessidade da sua execução, as funções associadas às tarefas no FreeRTOS são proibidas de retornar. Se uma tarefa não é mais necessária, esta deve ser apagada recorrendo à chamada explicita da função *vTaskDelete()* da API do sistema.

A qualquer momento, apenas uma tarefa está em execução. Diz-se então que esta se encontra no estado *Running*. As restantes tarefas podem estar prontas para execução ou não, sendo que primeiras se encontram no estado *Ready* e as últimas num dos dois estados *Suspended* ou *Blocked*. O diagrama completo dos estados das tarefas e possíveis transições entre estes pode ser consultado na Figura 4.4.

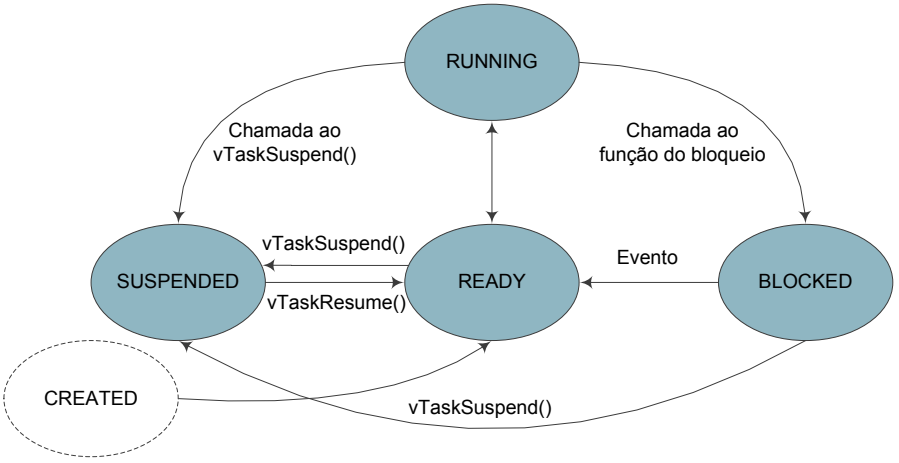


Figura 4.4: Transições do estado da tarefa no sistema operativo FreeRTOS

Mais uma vez, o escalonador do sistema é a única entidade que pode promover uma tarefa para o estado *Running*. E mais uma vez, uma tarefa só pode ser promovida para o estado *Running* a partir do estado *Ready*. Ao promover uma tarefa, o escalonador despromove a que está a executar, mas a última também pode abandonar a execução por ela própria. Se a tarefa chamar *vTaskSuspend(NULL)* (o argumento que esta função recebe é o apontador para tarefa a ser suspensa, *NULL* indica que deve ser suspensa a própria tarefa que chamou a função) o seu estado mudará para *Suspended* e o escalonador irá escolher para a promoção a tarefa mais prioritária das prontas para execução (estado *Ready*). A tarefa também pode chamar uma função que a colocará no estado da espera pelo evento. Este estado é denominado *Blocked*. Os eventos podem ser de dois tipos: eventos de temporização e eventos de sincronização. Uma tarefa pode atrasar a sua execução por uma quantia de tempo ou pode ficar a espera do certo instante do tempo em termos absolutos. Em termos dos eventos de sincronização, FreeRTOS oferece um vasto numero de recursos que os podem criar como, por exemplo, filas (*queues*), semáforos binários, semáforos de contagem, semáforos recursivos, recursos da exclusão mutua (*mutexes*). Ao bloquear a espera pelo evento de sincronização pode ser também definido um tempo do *timeout*, sendo que a tarefa sairá do estado *Blocked* quando ocorrer o respetivo evento ou quando expirar o tempo estipulado para *timeout*.

No que diz respeito à prioridade de uma tarefa, FreeRTOS compreende um valor maior como correspondente a uma prioridade maior, sendo o número total das prioridades disponíveis definido através do parâmetro *configMAX_PRIORITIES*, não existindo limite imposto pela arquitetura do sistema operativo, mas sim pela memória RAM, cujo consumo aumenta com aumento do *configMAX_PRIORITIES*. Várias tarefas podem partilhar a mesma prioridade.

4.2.3 Listas de Tarefas

O mecanismo do escalonamento em si baseia-se no manuseamento de um conjunto de listas das tarefas. Uma tarefa existindo encontra-se guardada numa e só numa destas listas. No ficheiro *list.h* está definida a estrutura de dados *xList*, sendo que no ficheiro *list.c* está implementado um conjunto de funções que operam sobre as listas. Dependendo do uso da função *vListInsertEnd()* ou *vListInsert()* a lista é manuseada, respetivamente, como sendo FIFO ou como sendo ordenada por algum valor (prioridade ou tempo de atraso). O escalonador do FreeRTOS inicializa e manuseia as listas das tarefas prontas para execução, tarefas que estão a espera do

evento temporal (*delayed tasks*) e tarefas que foram reconhecidas pelo escalonador enquanto este estava suspenso e que, sendo assim, serão movidas para lista das prontas para execução aquando escalonador for resumido. Também são manuseadas pelo escalonador as listas de tarefas que foram apagadas mas cuja memoria associada ainda não foi libertada, caso na configuração do núcleo a função API *vTaskDelete()* for habilitada, e tarefas suspensas, caso for habilitada a função API *vTaskSuspend()*. As linhas do código fonte que ilustram referidas afirmações e que foram retiradas do ficheiro *task.c* da distribuição do FreeRTOS são apresentadas na Listagem 4.11.

```
PRIVILEGED_DATA static xList pxReadyTasksLists[ configMAX_PRIORITIES ];
PRIVILEGED_DATA static xList xDelayedTaskList1;
PRIVILEGED_DATA static xList xDelayedTaskList2;
PRIVILEGED_DATA static xList xPendingReadyList;
#if ( INCLUDE_vTaskDelete == 1 )
    PRIVILEGED_DATA static xList xTasksWaitingTermination;
    PRIVILEGED_DATA static volatile unsigned uxTasksDeleted = 0U;
#endif
#if ( INCLUDE_vTaskSuspend == 1 )
    PRIVILEGED_DATA static xList xSuspendedTaskList;
#endif
```

Listagem 4.11: Listas de tarefas manuseadas pelo escalonador do FreeRTOS

No que diz respeito às tarefas prontas para execução, estas são organizadas num *array* das listas, onde o índice do *array* corresponde a uma certa prioridade e cada lista é tratada como sendo FIFO. Ou seja, pode ser dito que as tarefas prontas para execução são organizadas numa estrutura, frequentemente denominada como estrutura *multi-FIFO* (Jeffrey Liu, Insop Song, 2007). Uma representação esquemática da referida estrutura de dados pode ser vista na Figura 4.5.

As tarefas que estão a espera de um evento temporal são guardadas na lista *xDelayedTaskList* ordenada pelo valor correspondente ao número de *ticks*. Esta, por sua vez, é separada em duas listas idênticas: *xDelayedTaskList1* e *xDelayedTaskList2*. Tal separação resulta de uma solução não trivial do fenómeno associado ao *overflow* do temporizador. Existe então apontador *pxDelayedTaskList* que aponta para a lista que está atualmente em uso e um outro apontador *pxOverflowDelayedTaskList* que aponta para a lista que guarda as tarefas para quais ocorreu *overflow* aquando a sua inserção. Esta última lista passará a ser a atual e vice-versa (os apontadores vão trocar das listas) quando ocorrer o *overflow* do temporizador.

As outras três listas: *xPendingReadyList*, *xTasksWaitingTermination* e *xSuspendedTaskList* não apresentam curiosidades e o seu propósito já foi referido acima.

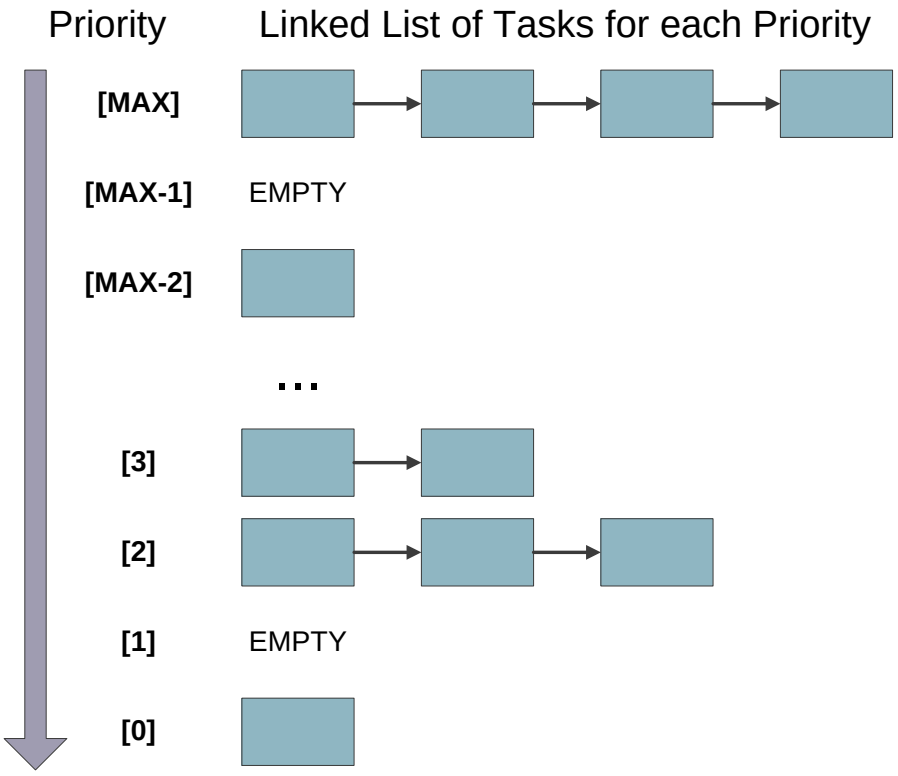


Figura 4.5: Representação esquemática da lista *multi-FIFO* das tarefas prontas para execução

O escalonador possui e manuseia as listas de tarefas referidas acima, sobrando apenas um tipo de tarefas: tarefas bloqueadas a espera de um recurso de sincronização. Da mesma forma como no sistema operativo ADEOS, cada recurso de sincronização possui e manuseia uma própria lista de tarefas por este bloqueadas. Como já foi referido na secção 4.2.2, FreeRTOS implementa um vasto número de recursos da sincronização: filas (*queue*), semáforos binários, semáforos de contagem, semáforos recursivos, recursos da exclusão mutua (*mutex*). O mecanismo complexo associado ao recurso da sincronização denominado fila (*queue*) é implementado no ficheiro *queue.c*, sendo que os restantes recursos de sincronização derivam deste e são implementados como sendo casos particulares de uma fila.

4.2.4 Escalonamento

O escalonador do sistema operativo FreeRTOS possui dois modos de funcionamento: escalonamento preemptivo baseado nas prioridades e escalonamento cooperativo. O modo do escalonamento deve ser escolhido pelo utilizador através da definição obrigatória do parâmetro *configUSE_PREEMPTION* no ficheiro *FreeR-*

TOSConfig.h e é inalterável após compilação. Se o dito parâmetro for definido com valor "1" o escalonamento preemptivo será usado, caso contrario, será usado escalonamento cooperativo. Algoritmo do escalonamento preemptivo baseado nas prioridades é relativamente simples e já foi discutido ao longo desta dissertação. Seguindo (Barry, 2009) este algoritmo pode ser resumido nos quatro pontos:

- Cada tarefa tem uma prioridade associada.
- Cada tarefa existe num dos estados (ver Figura 4.4).
- A qualquer momento apenas uma tarefa se encontra em execução (estado *Runnning*).
- O escalonador escolha para execução sempre a tarefa mais prioritária das prontas para execução (cujo estado é *Ready*).

Da mesma forma como no caso do ADEOS, pode ser dito que estamos perante algoritmo *fixed-priority preemptive scheduling*, onde *fixed-priority* significa que as prioridades assinadas para as tarefas não são alteradas pelo *kernel* e que podem ser alteradas apenas a partir do contexto de execução de uma tarefa (esta última afirmação é inaplicável no caso do ADEOS, pois este não implementa função API que permita alteração da prioridade da tarefa após a sua criação). Como já foi referido na secção 4.2.2, FreeRTOS não impõe limitações na forma como prioridades são assinadas às tarefas, permitindo que várias tarefas partilham a mesma prioridade. Quando ocorrer o consequente empate das prioridades, o escalonador resolve-o recorrendo à estratégia *round-robin* com *time slice*, fazendo com que as tarefas com a mesma prioridade partilham o acesso ao CPU ao executar alternadamente. A duração do *time slice* é inversamente proporcional à frequência do *system tick* que é definida através do parâmetro *configTICK_RATE_HZ*.

No caso da utilização do modo de escalonamento cooperativo, uma tarefa deixa o estado *Running* e ocorre uma comutação de contexto só se esta tarefa fica a espera de um evento (entra no estado *Blocked*) ou chama explicitamente a função API *taskYIELD()* (ou acaba a sua existência com a chamada do *vTaskDelete()*). Sendo assim, no modo cooperativo uma tarefa nunca é preemptida.

4.2.5 Pontos de escalonamento e Comutação de contexto

Ao contrário do caso do sistema operativo ADEOS, os pontos do escalonamento no FreeRTOS não consistem numa chamada explicita de uma rotina nos pontos

relevantes do código do *kernel*, mas sim baseiam-se na salvaguarda e restauro do contexto da tarefa aquando atendimento a um pedido de interrupção.

A explicação do mecanismo utilizado pode ser feita em termos genéricos mas, para tornar a descrição mais apelativa e informativa, esta será feita com base no exemplo do *porting* do FreeRTOS para o processador PPC405 presente no dispositivo FPGA da família Virtex4 do produtor Xilinx (FreeRTOS, b).

Recorrendo às macros definidas no BSP (board support package) gerado pelas ferramentas Xilinx aquando definição do SoC em causa, no ficheiro *port.c* as rotinas *vPortYield()*, *vPortTickISR()* e *vPortISRWrapper()* são assinadas como sendo *handlers* (rotinas de atendimento) para os casos de exceção *system call*, interrupção do temporizador programável e uma interrupção externa não crítica, respetivamente. As linhas do código que correspondem a afirmação anterior e que foram retiradas do código associado ao ficheiro portável *port.c* relativo ao *porting* para o processodor PPC405 podem ser encontradas na Listagem 4.12.

```
    //...
XExc_RegisterHandler(
    XEXC_ID_SYSTEM_CALL, (XExceptionHandler) vPortYield, (void*) 0);
    //...
XExc_RegisterHandler(
    XEXC_ID_PIT_INT, (XExceptionHandler) vPortTickISR, (void*) 0);
    //...
XExc_RegisterHandler(
    XEXC_ID_NON_CRITICAL_INT, (XExceptionHandler) vPortISRWrapper, NULL);
```

Listagem 4.12: Assinatura dos *handlers* das exceções/interrupções no *porting* para processador PPC405

As rotinas *vPortYield()*, *vPortTickISR()* e *vPortISRWrapper()* são implementadas na linguagem *assembly* no ficheiro *portasm.S*. Em três casos, o corpo da rotina é delimitado com as macros *portSAVE_STACK_POINTER_AND_LR* e *portRESTORE_STACK_POINTER_AND_LR*. No caso das rotinas *vPortYield()* e *vPortTickISR()*, dentro do corpo da rotina ocorre uma chamada explicita da rotina *vTaskSwitchContext()* (implementada na linguagem C no ficheiro *task.c*), enquanto no caso da rotina *vPortISRWrapper()* fica a responsabilidade do utilizador a utilização ou não de uma chamada explicita da comutação de contexto.

Uma exceção *system call* pode ser emitida explicitamente e, sendo assim, é possível efetuar uma chamada explicita do ponto de escalonamento. As linhas de código que definem a rotina que emita uma exceção *system call* foram recolhidas dos ficheiros *task.h* e *portmacro.h* e podem ser vistas na Listagem 4.13

```
#define taskYIELD()     portYIELD()                     //from task.h
//...
#define portYIELD()     asm volatile ("SC \n\t NOP")    //from portmacro.h
```

Listagem 4.13: Definição da macro *taskYIELD()* que emite exceção *system call*

A rotina *vPortTickISR()* é assinada como sendo o *handler* da interrupção do PIT (*Programmable Interval Timer*) do processador. Este temporizador programável é usado pelo *porting* como a entidade responsável pelo *system tick*. Ou seja, a referida interrupção ocorre com a frequência definida pelo parâmetro *configTICK_RATE_HZ* e resulta num ponto de escalonamento, que pode efetivamente acabar com a comutação de contexto nos seguintes casos: se desde o último *system tick* tinha ocorrido uma alteração na tarefa mais prioritária sem uma chamada explicita de comutação de contexto; no caso da empate da prioridade que resulta na execução da próxima tarefa da fila FIFO da respetiva prioridade.

A implementação das rotinas *vPortYield()* e *vPortTickISR()* pode ser consultada na Listagem 4.14.

```
vPortYield:
    portSAVE_STACK_POINTER_AND_LR
    bl vTaskSwitchContext
    portRESTORE_STACK_POINTER_AND_LR
    blr

vPortTickISR:
    portSAVE_STACK_POINTER_AND_LR
    bl xTaskIncrementTick

    #if configUSE_PREEMPTION == 1
        bl vTaskSwitchContext
    #endif

    /* Clear the interrupt */
    lis     R0, 2048
    mttsr   R0
    portRESTORE_STACK_POINTER_AND_LR
    blr
```

Listagem 4.14: Imlementação da rotina *vPortYield()* e *vPortTickISR()*

Segue-se uma análise das etapas que ocorrem aquando comutação de contexto perante uma chamada explicita do *taskYIELD()* ou ocorrência do *system tick*. Uma tarefa se encontra na execução, tendo uma pilha associada. Quando ocorre a interrupção é executado o prólogo da interrupção que é da responsabilidade do compilador, pode ser dito que o referido prólogo acaba por guardar o contexto atual do processador dentro da pilha da tarefa a executar. A seguir são executadas instruções que constituem a macro *portSAVE_STACK_POINTER_AND_LR* que pode ser consultada na Listagem 4.15.

```
.macro portSAVE_STACK_POINTER_AND_LR

    /* Get the address of the TCB. */
    xor     R0, R0, R0
    addis   R2, R0, pxCurrentTCB@ha
    lwz     R2, pxCurrentTCB@l( R2 )

    /* Store the stack pointer into the TCB */
    stw     SP,   0( R2 )

    /* Save the link register */
    stwu    R1, -24( R1 )
    mflr    R0
    stw     R31, 20( R1 )
    stw     R0,  28( R1 )
    mr      R31, R1

.endm
```

Listagem 4.15: Definição da macro *portSAVE_STACK_POINTER_AND_LR*

As primeiras três instruções servem para compor no registo `R2` o valor da variável global *pxCurrentTCB* que é apontador para o TCB (*Task Control Block*) da tarefa que está atualmente em execução, ou seja, da tarefa cuja pilha está sob operação de momento. De seguida, o atual apontador para a pilha é guardado na posição da memória apontada pelo *pxCurrentTCB* (a primeira posição na estrutura de dados *tskTaskControlBlock* é exatamente *pxTopOfStack*). As restantes instruções servem para preservar *link register*.

Se estamos perante *system tick* com escalonador no modo preemptivo ou perante exceção *system call*, a rotina *vTaskSwitchContext* será invocada. Esta é implementada na linguagem C no ficheiro *task.c*. Ao seu cargo fica a atualização do apontador *pxCurrentTCB* para o TCB da tarefa mais prioritária ou da próxima tarefa da prioridade atual, em caso de esta ser a máxima e houver mais do que uma tarefa associada. Esta atualização pode resultar ou não na alteração efetiva do apontador.

A seguir, são executadas instruções que constituem a macro *portRESTORE_STACK_POINTER_AND_LR* que pode ser consultada na Listagem 4.16.

As primeiras cinco instruções restauram *linker register* previamente preservado. As próximas três instruções compõem no registo `R1` (cujo *alias* é *SP*) o valor da variável global *pxCurrentTCB* , previamente atualizada na rotina *vTaskSwitchContext()*. De seguida, o apontador da pilha assume o valor guardado na primeira posição do TCB da tarefa escolhida no *vTaskSwitchContext()* como sendo a próxima para executar.

```
.macro portRESTORE_STACK_POINTER_AND_LR

    /* Restore the link register */
    lwz     R11,  0( R1 )
    lwz     R0,   4( R11 )
    mtlr    R0
    lwz     R31, -4( R11 )
    mr      R1,   R11

    /* Get the address of the TCB. */
    xor     R0, R0, R0
    addis   SP, R0, pxCurrentTCB@ha
    lwz     SP, pxCurrentTCB@l( R1 )

    /* Get the task stack pointer from the TCB. */
    lwz     SP, 0( SP )

.endm
```

Listagem 4.16: Definição da macro *portRESTORE_STACK_POINTER_AND_LR*

O processo do atendimento à interrupção acaba com o epílogo da interrupção, que é da responsabilidade do compilador e que, a partir da pilha para qual aponta o *stack pointer*, recupera o estado do processador correspondente a ultima vez que a tarefa proprietária desta pilha foi interrompida.

Pode ser resumido, que a comutação de contexto no sistema operativo FreeRTOS se baseia no prólogo e epílogo do atendimento à interrupção (que é da responsabilidade do compilador) e na manipulação do *stack pointer* entre o acontecimento destes. Existem três cenários para a comutação de contexto: a chamada explicita do *taskYIELD()* a partir de uma tarefa, isto, por sua vez, provoca *system call*; ocorrência do *system tick*; a chamada explicita da macro *taskYIELD_FROM_ISR()*, a partir da interrupção do utilizador, que nem é mais nem menos do que a redefinição do *vTaskSwitchContext()* .

4.3 Implementação da vertente *software* do sistema operativo híbrido

Como já foi referido no Capítulo 1, o principal objetivo do trabalho associado a esta dissertação passa pela implementação de um sistema operativo com a maioria de objetos do seu *kernel* implementados em *hardware*, com o intuito de beneficiar o comportamento de natureza determinística.

Está secção resume as etapas da implementação relacionadas com a adaptação do sistema operativo de suporte ADEOS e a sua integração junto com o *hardware microkernel* desenvolvido ao longo desta dissertação e apresentado em detalhe no Capítulo 5.

4.3.1 *Porting*

Após a escolha do sistema operativo de suporte, a primeira preocupação foi a sua correta execução numa arquitetura diferente ao da apontada pela versão original do código portável do sistema. O microprocessador em causa foi PPC405 embutido no dispositivo XC2VP30 da família *Virtex-II Pro* e presente na plataforma do desenvolvimento XUPV2. A plataforma do desenvolvimento, bem como o microprocessador são discutidos no Capítulo 3.

Como foi referido na secção anterior, a maior parte do código fonte do sistema operativo ADEOS é escrito em linguagem C++, o que o torna portável. Uma das particularidades do desenvolvimento do *kernel* de um sistema operativo em comparação com o desenvolvimento habitual do *software* consiste no facto de ser inevitável o recurso ao código dependente do processador na implementação de certos mecanismos, como por exemplo comutação de contexto. No âmbito desta dissertação, foi necessário efetuar o *porting* do sistema operativo ADEOS da arquitetura 80x86 para a arquitetura PPC405. Foi preciso adaptar e reescrever o código dependente do processador presente nos ficheiros *bsp.h* e *bsp.asm*.

O código do ficheiro *bsp.h* pode ser visto na Listagem 4.17.

No ficheiro encontra-se definida a estrutura de dados *Context*, definidas as macros de entrada e saída das secções críticas, bem como os protótipos das três funções C implementadas em *assembly* no ficheiro *bsp.asm*. Cada tarefa possui a própria variável do tipo *struct Context* no qual é salvaguardado o estado do processador

```
struct Context
{
  int IP;
  int CS;
  int Flags;
  int SP;
  int SS;
  int SI;
  int DS;
};
#include "task.h"
#define enterCS() asm { pushf; cli }
#define exitCS()  asm { popf }
extern "C"
{
  void contextInit(Context *, void (*run)(Task *), Task *, int * pStackTop);
  void contextSwitch(Context * pOldContext, Context * pNewContext);
  void idle();
};
```

Listagem 4.17: Versão original do ficheiro *bsp.h*

80x86 aquando comutação de contexto para uma outra tarefa. As secções críticas do código do *kernel* (código não preemptivo) são delimitadas com as macros *enterCS()* e *exitCS()* cujo efeito é desabilitação e habilitação de interrupções, respetivamente. É de notar que as rotinas a ser implementadas na linguagem *assembly* foram declaradas sob a diretiva *extern "C"*, que força o compilador a usar convenção de chamada da linguagem C e não utilizar a técnica de decoração de nome (*name mangling*), característica para linguagem C++ devido à funcionalidade de sobrecarga de método (*name overloading*).

Para efetuar o *porting*, mantendo a abordagem original aquando da comutação de contexto, é necessário, então, redefinir a estrutura *Context* para esta representar o contexto do processador PPC405, reescrever as rotinas *contextInit()*, *contextSwitch()* e *idle()*, bem como redefinir as macros *enterCS()* e *exitCS()*.

Introdução aos registos visíveis da arquitetura 80x86

Para ter uma maior facilidade na interpretação do código *assembly* respetivo às rotinas *contextInit()* , *contextSwitch()* e *idle()* , segue-se uma pequena *overview* dos registos compreendidos pela arquitetura 80x86. A informação relacionada com a arquitetura 80x86 presente nesta dissertação inicialmente foi compilada com base num conjunto de *websites*, sendo posteriormente consolidada através da consulta do (Brey, 1997). Os 14 registos que podem ser manipulados pelo utilizador, bem como a descrição do seu propósito habitual, são enumerados na Tabela 4.1.

O endereçamento da memória no caso da série de processadores 80x86 é segmen-

Tabela 4.1: Registos da arquitetura 80x86

Grupo	Nome	Acrónimo	Propósito
Uso Geral	Accumulator	AX	Multiplicação, divisão, I/O
	Base	BX	Apontador para memória
	Count	CX	Instruções REP: *shift*, *loop*, etc.
	Data	DX	Multiplicação, divisão, I/O
Apontador e Índice	Stack Pointer	SP	Apontador para o topo da pilha
	Base Pointer	BP	Apontador para memória
	Source Index	SI	Índice de origem nas operações *string*
	Destination Index	DI	Índex de destino nas operações *string*
Segmento	Code Segment	CS	Secção que guarda código
	Data Segment	DS	Secção que guarda dados
	Stack Segment	SS	Define memória da pilha
	Extra Segment	ES	Data Segment (DS) alternativo
Outros	Flags	-	Indica/controla estado do processador
	Instruction Pointer	IP	Aponta para a próxima instrução

tado. Os endereços são compostos por duas partes: segmento e *offset*. Ambos são números sem sinal de 16 *bits*. Para compor o endereço físico o segmento é deslocado 4 *bits* para esquerda e é somado com o *offset*, resultando, desta forma, num endereço de 20 *bits* de largura.

De seguida são enumerados os propósitos de alguns dos registos, no contexto da sua participação no mecanismo de endereçamento da memória. O processador efetua *fetch* das instruções a partir do IP (*Instruction Pointer*) e em relação ao segmento de código, definido pelo registo CS (*Code Segment*). O *fetch* de dados é efetuado em relação ao segmento de dados, definido pelo registo DS (*Data Segment*). As operações sobre a pilha são efetuados com utilização dos apontadores SP (*Stack Pointer*) e BP (*Base Pointer*) em relação ao segmento de pilha que é definido pelo registo SS (*Stack Segment*). O registo ES (*Extra Segment*) é apenas uma alternativa ao registo DS (*Data Segment*).

Quatro registos de uso geral: AX, BX, CX e DX são de 16 *bits* de largura, mas também são divididos nos *bytes high* e *low*. As partes mais e menos significativas de cada um destes registos podem ser referidos através de AH e AL, BH e BL, CH e CL, DH e DL, respetivamente. As operações de 8 *bits* sobre estes registos são suportadas por completo no conjunto de instruções.

O estado do processador é compreendido pelo registo IP, que indica a próxima instrução a ser executada, e num registo que contém um conjunto de *flags*.

Análise da rotina *conextInit()*

Segue-se a análise da rotina *conextInit()*. Como já foi referido, o protótipo da função é definido no ficheiro *bsp.h* sob a diretiva *extern "C"* e implementado na linguagem *assembly* no ficheiro *bsp.asm*, tendo em conta a convenção de chamada C. Dita convenção de chamada é padrão para interface das rotinas C/C++ aquando compilação recorrendo aos compiladores GNU *gcc* ou *Microsoft Visual C++*. No ato da chamada, os argumentos são empurrados (*pushed*) para a pilha da direita para esquerda. A Listagem 4.18 volta a mostrar o protótipo da função, enquanto a Listagem 4.19 mostra a primeira metade da implementação da dita função na linguagem *assembly*.

```
void contextInit(Context *, void (*run)(Task *), Task *, int * pStackTop);
```

Listagem 4.18: Prototipo da função *contextInit()*

```
EVEN
_contextInit   PROC    FAR
    push    bp
    mov     bp, sp
    les     di, dword ptr ss:[bp+6]     ; Get pContext from the caller.

    push    ds
    lds     bx, dword ptr ss:[bp+10]    ; Get pFunc from the caller.
    mov     dx, ds
    mov     es:[di], bx
    mov     es:[di+2], dx

    pushf                               ; Initialize the processor flags.
    pop     ax
    or      ax, 0000001000000000b       ; Enable interrupts by default.
    mov     es:[di+4], ax
;...
```

Listagem 4.19: Primeira metade da implementação da função *contextInit()*

A palavra reservada `FAR` indica que estamos perante a chamada *Far Call* o que signfica que, ao guardar na pilha o endereço de retorno, será guardado não sou o *offset* mas também o segmento. Sendo o segmento guardado numa posição superior da memória e o *offset* - inferior, o que vai de encontro com *endianess* da arquitetura x86 que é *little-endian*.

A combinação de instruções `push bp` e `mov bp, sp` é a primeira etapa das três que é da responsabilidade da rotina chamador antes da execução do seu corpo. A segunda etapa é a alocação do espaço para variáveis locais e que não é aplicável neste caso particular. E a terceira etapa é a salvaguarda dos registos que irão ser alterados durante a execução.

A Figura 4.6 apresenta o estado da pilha (cuja largura é de 16 *bits*) após a chamada e a execução das primeiras duas instruções da rotina. A instrução `push` faz decrementar o SP, guardar o *byte* superior do argumento, de seguida decrementar mais uma vez o SP e guardar o *byte* inferior do argumento. Assim, o SP passa a apontar para a posição onde foi salvaguardado BP e, após a execução da instrução `mov bp, sp`, o BP passa a ter também o mesmo valor.

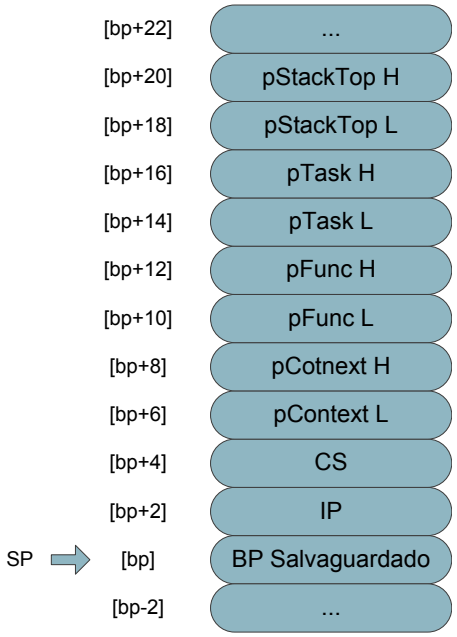


Figura 4.6: Estado da pilha após primeiras duas instruções do *contextInit()*

A partir da terceira instrução da rotina, os argumentos da função: *pContext*, *pFunc*, *pTask* e *pStackTop* podem ser acedidos através de `[bp+6]`, `[bp+10]`, `[bp+14]` e `[bp+18]`, respetivamente.

A instrução `les di, dword ptr ss:[bp+6]` faz a palavra (16 *bits*) referida pelo endereço `ss:[bp+6]` ser guardada no registo DI, enquanto a palavra referida pelo endereço `[bp+8]` - no registo ES. Ou seja, o endereço `es:[di]` passa a ser igual ao valor do primeiro argumento da função, que é o apontador para o contexto da tarefa em causa *pContext*.

A próxima instrução `push ds` salvaguarda o registo DS, pois este vai ser "perdido" após a execução da instrução `lds bx, dword ptr ss:[bp+10]` cujo sintaxe é em tudo igual à instrução `les`, ou seja, o endereço `ds:[bx]` passa a ser igual ao segundo argumento da função - *pFunc*. As três próximas instruções fazem guardar o *offset* do endereço *pFunc* no primeiro campo da estrutura *Context* (campo IP) da tarefa em causa, e que é referido pelo `es:[di]`, bem como guardar o segmento

do endereço *pFunc* no campo CS da estrutura *Context* referido pelo `es:[di+2]`. A Listagem 4.20 volta a mostrar a definição da estrutura *Context*, com intuito de reforçar as afirmações feitas acima.

```
struct Context
{
    int IP;     //es:[di]
    int CS;     //es:[di+2]
    int Flags;  //es:[di+4]
    int SP;     //es:[di+6]
    int SS;     //es:[di+8]
    int SI;     //es:[di+10]
    int DS;     //es:[di+12]
};
```

Listagem 4.20: Definição da estrutura *Context*

As quatro últimas instruções da Listagem 4.19 resultam no seguinte: o conteúdo do registo *flags* é passado pela pilha para o registo AX; é feito *set* ao *bit* correspondente a habilitação das interrupções; a palavra resultante é guardada no campo *Flags* da estrutura *Context*.

A Listagem 4.21 mostra a segunda metade da implementação da função *contextInit()* na linguagem *assembly*.

```
;...
    les     di, dword ptr ss:[bp+18]    ; Point to the task's stack.
    lds     bx, dword ptr ss:[bp+14]    ; Get pTask from the caller.
    mov     dx, ds
    mov     es:[di-4], bx               ; Place pTask onto the stack.
    mov     es:[di-2], dx

    les     di, dword ptr ss:[bp+6]     ; Point to the task's context.
    lds     bx, dword ptr ss:[bp+18]    ; Get pStack from the caller.
    mov     dx, ds
    sub     bx, 8                       ; Arrange value of future stack pointer.
    mov     es:[di+6], bx               ; Place future stack pointer onto SS:[SP].
    mov     es:[di+8], dx

    pop     ds
    mov     dx, ds
    mov     es:[di+10], si
    mov     es:[di+12], dx

    pop     bp
    ret
_contextInit    ENDP
```

Listagem 4.21: Segunda metade da implementação da função *contextInit()*

A combinação das cinco primeiras instruções pode ser vista como um *template*, que pode ser aplicado durante a programação da arquitetura 80x86, nos cenários quando uma variável de 32 *bits*, referida pela sua posição na pilha, deve ser guardada numa posição da memória, cujo endereço é relativo a uma outra variável

(apontador), também acessível pela sua posição na pilha. Neste caso concreto, o terceiro argumento da função (*pTask*) é guardado nas duas posições de memória sucessivas ao quarto argumento da função (*pStackTop*), que é o apontador para a pilha reservada para a execução da tarefa. A ordenação *little-endian* é assegurada.

As seis próximas instruções seguem o *template* referido acima e resultam na salvaguarda do valor `pStackTop-8` na combinação de campos SS:[SP] da estrutura *Context*. SS:[SP] é o valor do *stack pointer* que a rotina especial *run()* irá ver nos seus internos. A Listagem 4.22 volta a apresentar o protótipo da função especial *run()*, enquanto a Figura 4.7 mostra a pilha da tarefa após a execução das instruções acima referidas.

```
void run(Task * pTask);
```

Listagem 4.22: Protótipo da função especial *run()*

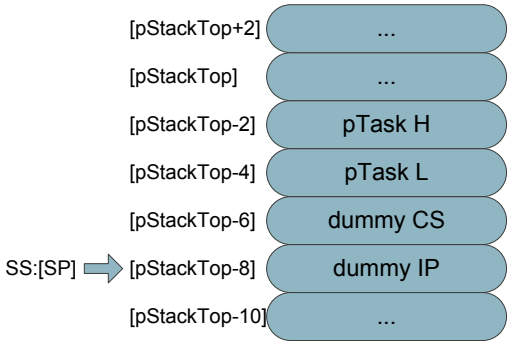


Figura 4.7: Estado da pilha da tarefa antes da execução da primeira instrução da função *run()*

De notar que após a atribuição do `pStackTop-8` ao *stack pointer* do contexto da tarefa (SS:[SP]) a pilha se encontra em conformidade com a convenção de chamada do tipo `FAR`. É importante também compreender que, visto que as posições da memória denominadas na figura pelo *dummy CS* e *dummy IP* não foram inicializadas, o retorno da função *run()* é inadmissível e levará à inconsistência do fluxo da execução.

As últimas seis instruções da Listagem 4.21 colocam o valor do previamente salvaguardado registo DS no respetivo campo do contexto da tarefa, bem como o registo SI que não foi alterado ao longo do corpo da rotina.

Análise da rotina *contextSwitch()*

Segue-se a análise da implementação da rotina *contextSwitch()*. A Listagem 4.23 apresenta o protótipo da função, enquanto a Listagem 4.24 e a Listagem 4.25 apresentam a primeira e a segunda metade da implementação da função na linguagem *assembly*, respetivamente.

```
void contextSwitch(Context * pOldContext, Context * pNewContext);
```

Listagem 4.23: Protótipo da função *contextSwitch()*

A função recebe como primeiro argumento o apontador para o contexto da tarefa que está a ser preemptida (*pOldContext*) e como segundo argumento o apontador para o contexto da tarefa que entrará em execução (*pNewContext*). Apenas no caso quando é preemptida a tarefa *idle*, em vez do valor real do apontador para o contexto da tarefa é passado o valor nulo (*NULL*) no argumento *pOldContext*.

```
EVEN
_contextSwitch   PROC    FAR
    push    bp
    mov     bp, sp

    les     di, dword ptr ss:[bp+6] ; Get pOldContext from the stack
    mov     dx, es
    mov     ax, di
    or      ax, dx                  ; if (pOldContext == NULL) goto fromIdle
    jz      fromIdle

    mov     dx, cs                  ; Save the address of the end of this routine
    lea     ax, switchComplete
    mov     es:[di], ax
    mov     es:[di+2], dx

    pushf                           ; Save the processor flags
    pop     es:[di+4]

    mov     dx, ss
    mov     es:[di+6], sp           ; Save the stack pointer
    mov     es:[di+8], dx           ; Save the stack segment
    mov     dx, ds
    mov     es:[di+10], si          ; Save the source index
    mov     es:[di+12], dx          ; Save the data segment
;...
```

Listagem 4.24: Primeira metade da implementação da função *contextSwitch()*

O modelo de programação da rotina é em tudo igual ao do caso da rotina *contextInit()* e o estado da pilha após a execução das instruções `push pb` e `mov bp, sp` pode ser visto na Figura 4.8.

As primeiras cinco instruções, depois da salvaguarda e atualização do *base pointer* (BP), resultam no mesmo que o seguinte *statement* da linguagem C:
`if (pOldContext == NULL) goto fromIdle;`.

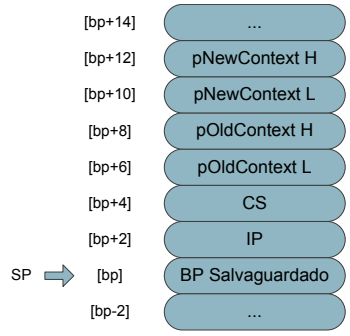


Figura 4.8: O estado da pilha após primeiras duas instruções do *contextSwitch()*

O apontador *pOldContext* é acedido através de `ss:[bp+6]` e é verificado se todos os *bits* do segmento e *offset* deste apontador são nulos. Caso verdade, conclui-se que está a ser preemptida a tarefa *idle* e que a etapa da salvaguarda do contexto deve ser ignorada, continuando a execução a partir da *label fromidle* (ver Listagem 4.25).

As restantes instruções desta metade da implementação, que corresponde a salvaguarda do contexto, simplesmente guardam os valores atuais dos registos nos campos da estrutura *Context* na ordem correta e a partir do primeiro campo (IP). O endereço guardado no CS:[IP] corresponde à *label switchComplete* que se encontra no fim da rotina. Este é o endereço a partir do qual será retomado o fluxo de execução da tarefa que está a ser interrompida aquando sua posterior volta à execução. Este pormenor do mecanismo da comutação de contexto será realçado no fim desta subsecção.

Segue-se agora a análise da segunda metade da rotina que é responsável pela restauração do contexto. O apontador para o contexto da tarefa que vai entrar em execução é acedido a partir da pilha (ss:[bp+10]). As próximas duas instruções parecem ser redundantes e têm aspeto de um erro da origem *copy-paste*. De seguida os registos são restaurados a partir dos campos da estrutura *Context* de forma ordenada e a começar dos últimos dois campos, que são *data segment* (DS) e *source index* (SI). Durante a execução das instruções correspondentes à restauração do SS e SP as interrupções são desabilitadas. Antes de desabilitar as interrupções, recorrendo a instrução `cli`, o registo inteiro de *flags* do processador é guardado no registo CX para ser recuperado depois da manipulação, desta forma, atómica do SS:[SP]. Aquando restantes três campos da estrutura *Context*, estes são empurrados para a pilha na ordem: *Flags*, IP, CS. De seguida é invocada a instrução `iret` que recupera da pilha CS, IP e *Flags* e continua a execução a partir de *CS:[IP]*, passando, desta forma, o controlo à tarefa que entrou em execução e cujo contexto era apontado pelo *pNewContext*.

```
;...
fromIdle:
    les     di, dword ptr ss:[bp+10]; Get pNewContext from the stack
    mov     dx, es
    mov     ax, di

    lds     si, dword ptr [di+10]   ; Restore the data segment and source index
    mov     dx, es:[di+8]           ; Restore the stack segment (part 1)
    mov     ax, es:[di+6]           ; Restore the stack pointer (part 1)
    pushf                           ; Save the current interrupt state
    pop     cx
    cli                             ; Disable interrupts
    mov     ss, dx                  ; Restore the stack segment (part 2)
    mov     sp, ax                  ; Restore the stack pointer (part 1)
    push    cx
    popf                            ; Restore the saved interrupt state

    push    es:[di+4]               ; Restore the processor flags
    push    es:[di+2]               ; Restore the return address
    push    es:[di]
    iret                            ; Now return, taking the saved flags with us
switchComplete:
    pop     bp
    ret
_contextSwitch   ENDP
```

Listagem 4.25: Segunda metade da implementação da função *contextSwitch()*

A execução da rotina acaba com a instrução `iret` e a nova tarefa entra em execução. *Label switchComplete* e mais duas instruções se encontram após instrução `iret`, trata-se da combinação `pop bp` e `ret` habitual para o retorno da rotina. A rotina *contextSwitch()* foi chamada a partir do contexto de uma tarefa cuja execução consequentemente foi interrompida. No endereço CS:[IP] do contexto da dita tarefa é colocado o endereço da *label switchCoplete*. Desta forma, quando a tarefa em causa voltar a ser promovida para execução e a instrução `iret` será executada no fim da rotina *contextSwitch()* (desta vez invocada a partir do contexto de uma outra tarefa), a consistência do fluxo da execução da primeira tarefa será mantido e este retornará da rotina *contextSwitch()*, continuando a sua execução. A Figura 4.9 ilustra a evolução do fluxo da execução do sistema durante o cenário em que uma tarefa A é preemptida pela tarefa B, que entra em execução pela primeira vez, e que de seguida é preemptida pela tarefa A que, desta forma, volta a executar.

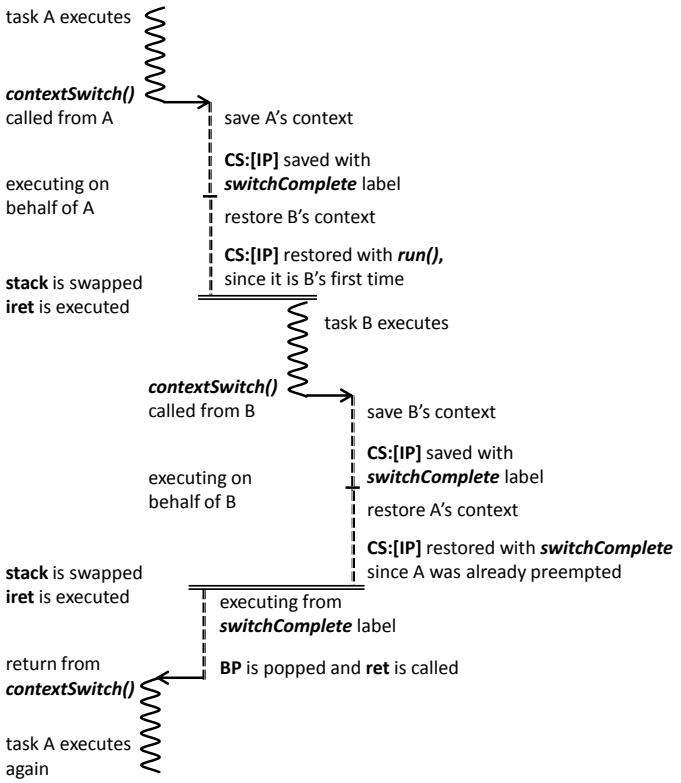


Figura 4.9: Ilustração do mecanismo da comutação de contexto, no exemplo de uma tarefa A que é preemptida pela tarefa B que executa pela primeira vez e que, de seguida, é preemptida pela tarefa A que, desta forma, volta a executar

Adaptação ao PPC405

Antes de proceder ao desenvolvimento do *hardware microkernel*, na fase inicial, foi necessário obter um sistema operativo de suporte, que executa sobre a plataforma de desenvolvimento adotada para o projeto, com intuito de familiarização com as ferramentas relacionadas e obtenção das noções práticas aquando arquitetura e utilização de um sistema operativo embebido do tempo real. O *demo* do FreeRTOS concebido para a plataforma do desenvolvimento ML403 dotada com o dispositivo Virtex-4 que, por sua vez, integra processador PPC405, foi adaptado para a plataforma do desenvolvimento XUPV2. Sendo assim, e com base no (Barry, 2009), o sistema foi experimentado do ponto de vista do utilizador. Aquando a decisão de adotar, como sendo o sistema operativo de suporte, um sistema ainda mais didático e fácil de depurar (ADEOS), foi necessário efetuar a reescrita do seu código dependente do processador. A análise detalhada do código subjacente, apresentada na subsecção anterior, foi feita em paralelo com a análise dos internos do FreeRTOS, mais propriamente, do seu *porting* para o processador PPC405. O código dependente do processador do sistema operativo ADEOS foi reescrito tendo

em conta a arquitetura PPC e respetiva linguagem *assembly*. A versão resultante do sistema permitiu a execução e teste de diferentes cenários, cuja respetiva depuração permitiu consolidar a compreensão do mecanismo de comutação de contexto implementado na versão original do ADEOS, bem como efetuar a correção dos *bugs* presentes no respetivo código fonte. Essa versão, no entanto, não chegou a ser depurada devidamente, no que diz respeito à correta gestão das pilhas, tornando-se obsoleta após a introdução dos *upgrades* enumerados na próxima secção.

4.3.2 *Upgrade*

O desenvolvimento do sistema operativo híbrido, baseado no *hardware microkernel*, teve como o ponto de partida a conceção do *porting* do sistema operativo de suporte. Após o sucesso do dito *porting*, que deu origem à possibilidade de aprendizagem, depuração e sensibilização, procedeu-se ao processo iterativo do *hardware/software codesign* que consistia na instanciação do SoC no ambiente de desenvolvimento Xilinx, o desenho do *hardware microkernel* na forma de um conjunto de periféricos mapeados na memoria e acedidos a partir do barramento PLB (*Processor Local Bus*) e a implementação da interface entre sistema operativo de suporte e *hardware microkernel*. Este processo, em certos pontos, abriu o caminho ao aumento do número de funcionalidades suportadas pelo sistema, em comparação com a sua versão original. Esta secção visa expor a informação relativa à reorganização, alteração e expansão do sistema operativo de suporte, que tem por base o código fonte do sistema operativo ADEOS.

Software versus Hardware

O *hardware microkernel*, desenhado no âmbito desta dissertação e apresentado em detalhe no Capítulo 5, implementa em *hardware* um conjunto de recursos comuns aos sistemas operativos embebidos. Os recursos que eram implementados em *software* passam a residir em *hardware*. Esta migração implica a remoção destes recursos do *software* e a implementação da *interface* ente a parte do sistema que continua em *software* e os recursos que agora se encontram implementados em *hardware* e em certos casos numa forma diferente.

Durante o processo do estabelecimento da dita interface, foi decidido manter a versão baseada no *hardware microkernel* e a versão puramente em *software*. A resultante variabilidade é gerida recorrendo a técnica da compilação condicional

baseada no uso das diretivas de pré-processamento: *#if*, *#ifdef*, *#else*, *#elif*, *#endif*, etc. HW_SCHED é a macro que define o uso da versão, sendo que, quando é definida com o valor "1", o sistema resultante da compilação se baseia no *hardware microkernel*, enquanto o valor "0" resulta na versão puramente em *software*.

Durante a evolução do *hardware microkernel* e a alteração das funcionalidades suportadas pelo sistema, a sustentação da versão puramente *software* foi suspensa, até voltar a ser recuperada na fase final do projeto, com intuito de avaliação dos potenciais benefícios que resultaram do uso do *hardware microkernel*.

Utilidades de *trace* e *timing*

Durante a fase da depuração, como auxílio na validação de certos cenários, várias vezes, em certos pontos estratégicos do código foram impressas mensagens informativas com recurso à função *xil_printf()* (versão mais leve do *printf()*). Algumas destas informações revelaram-se ser úteis na depuração do vasto número de cenários e permaneceram no código ao longo das várias versões, encapsuladas nas secções de compilação condicional, resultando numa utilidade simplista de *trace*. Ditas mensagens foram agrupadas em três categorias, sendo que as macros *IDLE_TRACE*, *BASIC_TRACE* e *MUTEX_TRACE* definem a compilação condicional de cada uma destas.

Foram também concebidas algumas utilidades simplistas da medição dos tempos que serão referidas no Capítulo 6. Mas ao contrário das referidas utilidades de *trace*, estas só se adequam aos cenários concretos e a sua presença na parte do código referente ao núcleo do sistema não é justificada.

adeos_config.h

Para conter as macros da configuração do sistema, como por exemplo as referidas na secção anterior, foi criado o ficheiro cabeçalho *adeos_config.h*. Para além das macros que configuram as utilidades de *trace* e *timing*, o ficheiro contém também parâmetros da configuração, enumerados na Listagem 4.26.

```
#define HW_SCHED         0
#define ROUND_ROBIN      0
#define RR_RATE_HZ       1
#define HW_SCHED_N_TASK 16

#define SYS_TICK_RATE 1000

#define HW_PTR   1
#define SW_PTR   0

#define BUS_CLOCK_HZ 100000000
#define CPU_CLOCK_HZ 100000000
```

Listagem 4.26: Principais parâmetros da configuração do *kernel* concebido

O primeiro parâmetro HW_SCHED já foi referido numa das secções anteriores e o seu propósito é a escolha entre o sistema que utiliza recursos em *hardware* (quando é definido como "1") e o sistema puramente em *software* (quando é definido como "0").

O parâmetro ROUND_ROBIN indica a utilização ("1") ou não ("0") da estratégia *round-robin* com *time slice* como o método do desempate das prioridades.

O parâmetro HW_SCHED_N_TASK deve ser definido com o número igual ao do número de tarefas suportado pela versão do escalonador em *hardware* utilizado. O *nibble* menos significativo da palavra devolvida pelo escalonador em *hardware*, como a resposta ao comando da leitura, cujo *opcode* é definido pelo parâmetro C_GET_VERSION, contem o número de níveis da árvore binária instanciada em *hardware*, que será discutida na secção 5.5.1. Por sua vez, o número de tarefas é igual ao 2^n, onde `n` é o número de níveis da árvore. No arranque do sistema operativo, o número de tarefas suportadas pelo *escalonador* (quando em *hardware*) é consultado ao *periférico* e, caso este for diferente ao do definido no ficheiro de configuração, é emitida uma mensagem de aviso.

O parâmetro RR_RATE_HZ, tanto no cenário híbrido como no puramente *software*, define a frequência da preempção no algoritmo *round-robin* (em Hertz), quando este é utilizado como o método do desempate das prioridades. No cenário híbrido, este define a configuração do temporizador dedicado, que é ligado através da interface dedicada ao escalonador em *hardware* e que o força ao reescalonamento do estilo *round-robin*. No cenário puramente *software*, a relação entre este parâmetro e o parâmetro SYS_TICK_RATE define o número dos *system-ticks* que devem ocorrer entre cada chamada da rotina *sw_rr()* que força o reescalonamento do estilo *round-robin*.

O parâmetro SYS_TICK_RATE só é compreendido no cenário do sistema em *software* e define em Hertz a frequência da ocorrência do *system-tick*. No caso particular do processador PPC405 é utilizado o temporizador PIT (*Programmable Interval Timer*) configurado com base na frequência definida. Dentro da rotina que implementa o *handler* da respetiva interrupção é feita a atualização dos temporizadores em *software*, bem como o reescalonamento em *software* com a frequência definida pelo RR_RATE_HZ que logicamente deve ser menor ou igual ao do SYS_TICK_RATE. No cenário híbrido, devido a ausência do *system-tick*, discutida em detalhe no Capítulo 5, este parâmetro não tem qualquer significado.

Os parâmetros HW_PTR e SW_PTR apresentam redundância, visto que, quando um destes é definido com o valor "1", o outro deve ser obrigatoriamente definido com o valor "0". A remoção desta redundância é considerada como uma melhoria a fazer e é enumerada nos rascunhos relacionados com a manutenção do código. A configuração destes parâmetros define a solução utilizada para a manutenção dos apontadores para as tarefas aquando utilização do escalonador em *hardware*. As duas soluções são: manter em *software* um *array* dos apontadores indexado pelo identificador da tarefa (numero inteiro de 0 a HW_SCHED_N_TASK-1) ou manter os apontadores para tarefas diretamente no periférico em *hardware*. A primeira solução é compilada quando o parâmetro SW_PTR é definido com o valor "1" e a segunda, respetivamente, quando com "1" é definido o parâmetro HW_PTR. A configuração se baseia no *trade-off* entre a diminuição do tempo despendido durante a comutação de contexto (HW_PTR = "1") e a diminuição do consumo dos recurso FPGA (SW_PTR = "1"). No entanto, visto que o *hardware* é sintetizado e programado na FPGA de forma independente do pré-processador do compilador C, a configuração do *hardware microkernel* não é alterada com a configuração dos referidos parâmetros e este tem alocado os recursos para suporte de apontadores em *hardware* independentemente da configuração da parte *software* do sistema operativo.

O parâmetro BUS_CLOCK_HZ deve ser definido com a frequência do barramento PLB do sistema. Este parâmetro é utilizado no cálculo dos valores absolutos para temporizadores dedicados das tarefas e para temporizador do *round-robin*, no caso do sistema ser híbrido. O parâmetro CPU_CLOCK_HZ, por sua vez, deve ser definido com a frequência do funcionamento da unidade do processamento, pois esta será também a frequência do funcionamento do PIT (*Programmable Interval Timer*), que é configurado tendo em conta este parâmetro e usado na versão do sistema puramente *software*.

Hardware timer

A existência do temporizador simplista em *hardware*, desenhado para o controlo da funcionalidade *round-robin* do *hardware microkernel*, se refletiu sobre a parte do *software* com a criação da classe *Hw_Timer*, cuja declaração pode ser encontrada na Listagem 4.27.

```
class Hw_Timer
{
    public:
        Hw_Timer(unsigned int * addr);

        void set_period(int);
        void run();
        void reset();
        void run_autoreload();
    private:
        unsigned int *timer_addr;
};
```

Listagem 4.27: Declaração da class Hw_Timer

A variável privada *timer_addr* guarda o endereço base do periférico do tipo *simple periodic timer*. Os comandos da escrita, resultantes das chamadas dos métodos da classe, usam este endereço. O periférico em causa foi desenhado recorrendo ao *wizard* CIP (*Create and Import Peripheral*) configurado para produzir um *template* que possui dois registos físicos acessíveis a partir de *software*. O primeiro dos dois guarda o período do temporizador, enquanto o segundo guarda a palavra do controlo que define o comportamento do circuito digital. A Listagem 4.28 mostra a implementação do construtor da classe, do método *run_autoreload()*, do método *set_period()*, bem como a definição das macros que refletem o significado dos *bits* do registo do controlo.

Quando é criado um periférico com recurso ao *wizard* CIP, este cria também o *template* do *driver* correspondente, para ser enquadrado no *board support package* gerado automaticamente pelo *Xilinx Platform Studio*. Este *template* do *driver* limita-se aos três ficheiros: ficheiro *".c"* que vem vazio, ficheiro cabeçalho *".h"* que define um conjunto de macros para construção dos comandos de leitura e escrita e ficheiro *".c"* denominado *selftest* que é um ficheiro exemplo que contém rotinas da verificação da correta resposta do periférico e que na versão automaticamente gerada verificam apenas a funcionalidade de escrever e ler os valores dos registos acessíveis a partir do *software*. A Listagem 4.29 apresenta as macros definidas no ficheiro *simple_periodic_timer.h*, gerado automaticamente, relevantes para com-

```
#define TIMER_RESET          0x00000000
#define TIMER_AUTORELOAD     0x20000000
#define TIMER_RUN            0x40000000
//...
Hw_Timer::Hw_Timer(unsigned int * addr) {
  timer_addr=addr;
}

void Hw_Timer::set_period(int Period) {
  SIMPLE_PERIODIC_TIMER_mWriteSlaveReg0((Xuint32) timer_addr, 0, Period);
}
//...
void Hw_Timer::run_autoreload() {
  SIMPLE_PERIODIC_TIMER_mWriteSlaveReg1((Xuint32) timer_addr, \
                                        0, TIMER_RUN | TIMER_AUTORELOAD);
}
```

Listagem 4.28: Implementação da classe *Hw_Timer*

preensão dos comandos utilizados nos métodos da classe *Hw_Timer* (Os prefixos SIMPLE_PERIODIC_TIMER nesta listagem foram substituídos pelo SPT por razões de "compactidade").

```
#define SPT_USER_SLV_SPACE_OFFSET (0x00000000)
#define SPT_SLV_REG0_OFFSET (SPT_USER_SLV_SPACE_OFFSET + 0x00000000)
#define SPT_SLV_REG1_OFFSET (SPT_USER_SLV_SPACE_OFFSET + 0x00000004)

#define SPT_mWriteSlaveReg1(BaseAddress, RegOffset, Value) \
      XIo_Out32((BaseAddress)+(SPT_SLV_REG1_OFFSET)+(RegOffset), (Xuint32)(Value))
```

Listagem 4.29: Macros para comunicação com o periférico geradas automaticamente

Comutação de contexto baseada no atendimento às interrupções

A arquitetura do *hardware microkernel* concebido pressupõe a geração da interrupção quando e só quando ocorre uma alteração efetiva do identificador da tarefa que deve ser promovida para execução, sendo esta interrupção a única interface de obtenção do *feedback* da manipulação sobre os recursos do *hardware microkernel*. Este facto, junto com a consideração do FreeRTOS para futura integração do *hardware microkernel* desenhado, levaram à decisão de adotar a abordagem utilizada para implementação da comutação de contexto no sistema operativo FreeRTOS. Dito mecanismo já foi apresentado na secção 4.2.5 e pode ser resumido, que a comutação de contexto baseia-se no prólogo e epílogo do atendimento à interrupção (que é da responsabilidade do compilador) e na manipulação do *stack pointer* entre o acontecimento destes.

Função *CreateTask()*

A versão original do *kernel* ADEOS apresenta apenas um mínimo de funcionalidades. Com a integração do *hardware microkernel* aumentou a gama das funcionalidades e consequentemente a variedade de possíveis cenários de utilização. Aquando implementação destes cenários de utilização do ponto de vista do utilizador do *kernel*, sentiu-se a falta de flexibilidade no que diz respeito à criação da tarefa e seu ciclo da vida. Para endereçar essa questão, foram feitas alterações na função especial *run()* e no construtor da classe *Task*, bem como foi criado o método público *CreateTask()* na classe *Task*. A Listagem 4.30 mostra o protótipo do método *CreateTask()*. Apenas esse método deve ser usado pelo utilizador aquando criação de uma tarefa. Assim sendo, o construtor da classe *Task* passou a ser declarado como privado em vez de público.

```
static Task* CreateTask(void (*function)(void *), int priority, int stackSize,
            void * arg, int irq_id, int destroy_on_return, int start_on_create);
```

Listagem 4.30: Protótipo do método *CreateTask()* da classe *Task*

Na altura da criação da tarefa, para além de definir a função associada, a prioridade e o tamanho da pilha, passam a ser também definidos os argumentos:

- *void * arg* - É utilizado para passagem do argumento à função associada à tarefa na forma de um apontador para *void*. O verdadeiro tipo deve ser conhecido pela função associada e o respetivo *cast* deve ser feito enquanto utilização do argumento. Este *upgrade* aos funcionalidades se refletiu também na declaração da função associada e na chamada desta dentro da função especial *run()*.

- *int irq_id* - Define o identificador da fonte da interrupção externa a ser associada no controlador das interrupções dedicado, apresentado em detalhe na secção 5.5.3. É inicializado com o valor "-1" quando não se aplica.

- *int destroy_on_return* - Este argumento é avaliado dentro da função especial *run()* depois da devolução da função associada à tarefa. Se for igual a "1", a tarefa é destruída e a sua memória é desalocada, caso contrário a tarefa é apenas desabilitada e passa a estar pronta para chamar de novo a sua função associada quando for habilitada ("acordada").

- *int start_on_create* - Define se a tarefa é habilitada aquando a sua criação (inicializado com o valor "1") ou é criada no estado desabilitado.

4.3.3 Interface

Após *porting*, depuração e alguns dos *upgrades* realizados, procedeu-se a integração do *hardware microkernel* no *kernel* em *software* resultante das tarefas anteriores.

A arquitetura do *hardware microkernel* é apresentada em detalhe no Capítulo 5. Este é constituído por três periféricos mapeados na memória. A interface entre CPU (que executa *kernel* em *software*) e o *hardware microkernel* concebido limita-se às operações de leitura e escrita das posições da memória que correspondem aos registos físicos ou virtuais dos periféricos em causa. Foi criado o ficheiro *hw_kernel_io.h* para resumir as macros usados como auxílio na composição de endereços, comandos, argumentos e operações, bem como na decomposição do *feedback* recebido dos periféricos. A Listagem 4.31 mostra uma parte do código contido no ficheiro *hw_kernel_io.h* que, no exemplo de algumas funcionalidades do escalonador, demonstra a lógica da interface através do mapeamento na memória.

```
#define SCHED_BASEADDR          XPAR_PRIORITIZER_N_ADDR_BFM_0_BASEADDR

#define read_reg(cmd)      (*(volatile unsigned int*)(cmd))
#define write_reg(reg,val) ((*(volatile unsigned int*)(reg)) = (unsigned int)val)
#define read_sched_cmd(cmd,id)         read_reg(encode_sched_cmd(cmd,id))
#define write_sched_cmd(cmd,id,value)  write_reg(encode_sched_cmd(cmd,id),value)

#define C_DISABLE_TASK              0x09
#define C_AWAKE_TASK                0x0A
#define C_RELEASE_MUTEX             0x0C
#define C_GET_PTR_ACK               0x0F

#define COMMAND_SHIFT               12
#define COMMAND_MASK                0x000000FF
#define ID_SHIFT                    4
#define ID_MASK                     0x00000FF

#define encode_sched_cmd(cmd,id)    (SCHED_BASEADDR|\
                                    (((cmd)&COMMAND_MASK)<<COMMAND_SHIFT)|\
                                    (((id)&ID_MASK)<<ID_SHIFT))

#define soht_disable_id(id)         write_sched_cmd(C_DISABLE_TASK,id,0)
#define soht_awake_id(id)           write_sched_cmd(C_AWAKE_TASK,id,0)
#define soht_release_mutex_mid(mid) write_sched_cmd(C_RELEASE_MUTEX,0,mid)
#define soht_get_ptr_ack()          read_sched_cmd(C_GET_PTR_ACK,0)
```

Listagem 4.31: Parte do código contido no ficheiro *hw_kernel_io.h*

Todos os métodos que manuseiam as estruturas de dados contidas no *hardware mircokernel* ou com este relacionadas foram encapsulados na classe *HwKernel*, criada para o propósito. O código da Listagem 4.32 representa a declaração da classe mencionada. Os atributos da classe limitam-se a dois apontadores privados: primeiro é o endereço base do componente que implementa o escalonador (*hwkernel_p*), enquanto o segundo (*sys_timer*) é o apontador para objeto que serve de

abstração para interface com um simples temporizador em *hardware* que é utilizado para gerar o pulso periódico no sinal ligado ao escalonador em *hardware* e que provoca o reescalonamento *round-robin* para as tarefas com a mesma prioridade. No que diz respeito aos métodos, foi tomada a decisão que as funcionalidades que interagem diretamente com o escalonador em *hardware*e que na maioria dos casos se limitam a uma macro devem ser implementados como sendo funções *inline*. Para possibilitar *inline* na ausência das ferramentas de compilação e *link* invulgarmente inteligentes, a definição e não apenas a declaração da função *inline* deve encontrar-se no escopo (Stroustrup, 1997).

```
class HwKernel
{
    public:
        HwKernel(unsigned int * timer_addr, unsigned int * sched_addr);

        void start();
        void print_version();

        inline int  request_task(int priority, int start_on_create){ /*...*/ }
        inline void set_task_pointer(int id, Task * pointer){ /*...*/ }
        inline void disable_running_task(){
            soht_disable_id(os.pRunningTask->id);
            yieldPPC();}
        inline void disable_task(int id){ /*...*/ }
        inline void remove_task(int id, int irqid){ /*...*/ }
        inline void delay_running_task(int delay_time){ /*...*/ }
        inline void delay_task(int id, int delay_time){ /*...*/ }
        inline void awake_task(int id){ /*...*/ }
        inline void nops(){ /*...*/ }
    private:
        unsigned int *hwkernel_p;
        Hw_Timer *sys_timer;

        void enable_interrupt();
        void enable_preemption();
};
```

Listagem 4.32: Declaração da classe *HwKernel*

Como pode ser visto na listagem, os métodos *inline* foram definidos e não apenas declarados dentro da definição da respetiva classe. As definições de quase todos os métodos *inline* foram omitidas nesta listagem com intuito de "compactidade". Para além de encapsular as operações de leitura e escrita definidas no ficheiro *hw_kernel_io.h* esta classe também proporciona a abstração na vertente da arquitetura de interrupções associada a existência do *hardware microkernel*. A Listagem 4.33 apresenta a definição do construtor da classe *HwKernel*.

Como pode ser visto na listagem, construtor recebe como argumentos dois endereços usados na inicialização dos únicos atributos da classe. No que diz respeito as exceções do PPC405, no âmbito da sua utilização nas ferramentas integradas do Xilinx, o respetivo *board support package* proporciona um conjunto de funções

```
HwKernel::HwKernel(unsigned int * timer_addr, unsigned int * sched_addr)
{
    hwkernel_p=sched_addr;
    sys_timer=new Hw_Timer(timer_addr);
    XExc_Init();
    XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
                         (XExceptionHandler)asmISRWrapper, (void *)0);
    XExc_RegisterHandler( XEXC_ID_SYSTEM_CALL,
                         (XExceptionHandler)asmISRWrapper, (void *)0);
    enable_interrupt();
}
```

Listagem 4.33: Construtor da classe *HwKernel*

para a sua gestão (Glover, 2005). A declaração destas funções pode ser encontrada no ficheiro *xexception_l.h* localizado na pasta *include* do *board support package* integrado automaticamente no projeto SDK (*Software Development Kit*) quando inicializado a partir do XPS *Xilinx Platform Studio*. A função *XExc_Init()* inicializa os mecanismos de tratamento de exceções para o sistema baseado no PPC. Para todas as exceções o vetor é configurado com um *stub handler*. A função *XExc_RegisterHandler()* faz a ligação entre o identificador da fonte de exceção e o *handler* associado que é executado quando a exceção é detetada. O último argumento nesta função (é nulo na listagem) é usado como argumento para o *handler* quando este é chamado.

Do ponto de vista das funcionalidades da versão original do *kernel*, foi feita uma integração transparente do suporte por *hardware*, sendo invisível para o utilizador a natureza da execução das tarefas do *kernel* e a existência da classe *HwKernel*. No entanto, com a integração do *hardware microkernel*, aumentou a gama das funcionalidades, sendo novas funcionalidades acedidas pelo utilizador através da invocação dos métodos sobre o objeto *hwkernel*.

4.4 Conclusões

Ao longo deste capítulo foi feita uma apresentação detalhada do sistema operativo ADEOS e o sistema operativo FreeRTOS. Com base nesta apresentação, que englobou a descrição da organização, funcionalidades e características, bem como a enumeração dos recursos que estes sistemas compreendem e a análise da respetiva implementação, foi possível a introdução nos conceitos fundamentais de um sistema operativo embebido.

O principal objetivo do trabalho associado a esta dissertação passa pela implementação de um sistema operativo com um numero significante de objetos do seu *kernel* implementados em *hardware*, com o intuito de beneficiar o comportamento de natureza determinística.

Foi decidido basear o projeto num sistema operativo simplista e de poucas funcionalidades, como o apresentado ADEOS. Ao contrário da abordagem que passa pela implementação e integração transparente do *hardware microkernel* num sistema operativo relativamente mais complexo, neste cenário, o autor pôde usufruir de um maior controlo sobre os internos do sistema. Dito maior controlo, por sua vez, resultou na aceleração da curva de aprendizagem do autor, abriu possibilidade de experienciar as particularidades de desenvolvimento de sistemas operativos, através da implementação de *upgrades* que modificam ou acrescentam as funcionalidades no sistema, potenciou a elaboração dos testes de avaliação relevantes, bem como a correta interpretação dos seus resultados. O *refactoring* e integração de um sistema operativo existente, livre e adotado pela indústria de *software*, como o apresentado FreeRTOS, também foi considerado. A implementação do sistema operativo e *hardware microkernel* levou em conta resultados desta consideração e procurou proporcionar a possibilidade para fácil integração do *hardware microkernel* nestes sistemas.

Foram apresentadas as questões relacionadas com *porting* do sistema, *upgrades* efetuados ao sistema, interface entre o sistema e *hardware microkernel*.

Os dois sistemas apresentados são muito semelhantes do ponto de vista da sua organização geral, dos recursos base que os compreendem e da respetiva estratégia do escalonamento. Ambos sistemas implementam a estratégia do escalonamento do tipo *fixed-priority preemptive*. No entanto, seguem a abordagem radicalmente oposta na implementação dos pontos de escalonamento e comutação de contexto. No caso do sistema operativo ADEOS, os pontos de escalonamento consistem numa chamada explicita de uma rotina nos pontos relevantes do código do *kernel*, enquanto no FreeRTOS a mudança do contexto baseia-se na salvaguarda e restauro do contexto da tarefa aquando atendimento a um pedido de interrupção. A última abordagem se revelou como a mais adequada para a integração com o *hardware microkernel* concebido, cuja arquitetura pressupõe a utilização do sinal da interrupção como a única forma do *feedback* para com o CPU e onde a geração da interrupção é feita quando e só quando ocorre uma alteração efetiva do identificador da tarefa que deve ser promovida para execução.

Capítulo 5

Hardware MicroKernel

O núcleo desta dissertação consiste no desenho e conceção de um *hardware microkernel* que implementa, na forma de um conjunto de periféricos, os recursos comuns de um sistema operativo embebido. Este *hardware microkernel* pode ser integrado num sistema operativo existente ou pode servir de base para conceção de um sistema operativo híbrido. Este capitulo é dedicado a descrição do *hardware microkernel* concebido do ponto de vista da sua interface e funcionalidades, bem como do ponto de vista da sua implementação e *design flow* associado.

As noções que são inerentes ao ambiente e a plataforma de desenvolvimento utilizados e que complementam o *background* necessário para melhor orientação neste capítulo são cobertas no Capítulo 3.

5.1 Requisitos

Após levantamento da organização dos sistemas operativos embebidos de tempo real e análise dos recursos que estes implementam, foram estipulados os requisitos para o *hardware microkernel* a ser implementado. Estes requisitos foram divididos em dois grupos: requisitos base e funcionalidades extra.

Os requisitos base enumeram um conjunto mínimo dos recursos que compreendem um sistema operativo embebido de tempo real com a estratégia de escalonamento baseada nas prioridades. Desta forma, pretende-se uma entidade que manuseia a informação das tarefas, inclusivamente as suas prioridades, e que permite a extração da tarefa com a maior prioridade das prontas para execução a qualquer momento. Pretende-se a alocação dos recursos de temporização dedicados a cada

uma das tarefas. E, por fim, pretende-se também a existência dos recursos simplistas de sincronização que implementam a exclusão mútua.

Como as funcionalidades extra, que irão potencializar a utilização do *hardware microkernel*, foram estipuladas: o mecanismo *round-robin*, para servir da estratégia do desempate das prioridades, e a unificação do espaço de prioridades de tarefas e interrupções, discutida em detalhe na secção 2.3.

As próximas secções apresentam a arquitetura implementada, que endereça os estipulados requisitos base, bem como as funcionalidades extra. A apresentação é feita na abordagem *top-down*, dando primeiro o foco à interface e funcionalidades do *hardware microkernel* como um todo e, de seguida, fazendo a explicação mais detalhada de cada um dos aspetos do desenho e implementação de cada componente.

5.2 Visão global e configuração

A plataforma do desenvolvimento, utilizada neste projeto e apresentada no Capítulo 3, definiu o ambiente do desenvolvimento e as ferramentas disponíveis para a conceção do projeto. A plataforma de desenvolvimento é denominada XUPV2, é dotada com o dispositivo FPGA de alto desempenho XC2VP30 da família Virtex-II Pro do produtor Xilinx e é integrada nas ferramentas do desenvolvimento EDK 10.1. Para o dispositivo em causa, as ferramentas do desenvolvimento possibilitam a construção de um SoC baseado num dos dois núcleos PPC405 embutidos no dispositivo ou no *soft* processador *MicroBlaze* desenvolvido pelo Xilinx.

O projeto implementado no âmbito desta dissertação se baseia no microprocessador PPC405. O *hardware microkernel* é implementado na forma de um conjunto de periféricos que possuem interface *slave* PLB (*Processor Local Bus*). Desta forma, os registos físicos ou virtuais destes periféricos são acedidos a partir do processador através das instruções de leitura ou escrita das respetivas posições da memória. A Figura 5.1 apresenta o diagrama de blocos que mostra a organização do *hardware microkernel*.

O *hardware microkernel* é composto por três componentes: escalonador (*Scheduler*), controlador das interrupções que em conjunto com o escalonador proporciona a unificação do espaço de prioridades das tarefas e interrupções (SIC) e o componente que contêm temporizadores dedicados para cada uma das tarefas do

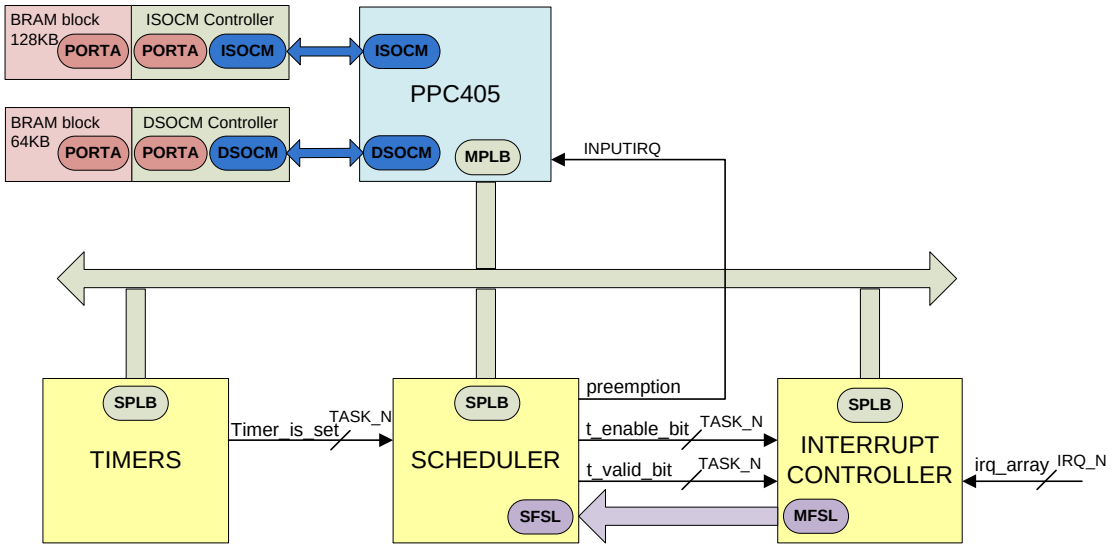


Figura 5.1: Arquitetura do *hardware microkernel* implementado

escalonador (Timers).

Como já foi referido, os requisitos estipulados endereçam a estratégia de escalonamento *fixed-priority preemptive*.

Os periféricos são criados com recurso ao *wizard* CIP e os respetivos procedimentos e particularidades serão apresentados nas secções posteriores. O *wizard* possibilita configurar IPIF que irá atuar sobre o barramento PLB (normalmente no papel do *slave*) e selecionar os sinais do IPIC pretendidos (interface visível do lado da lógica customizavel do utilizador). Os três componentes do *hardware microkernel* apresentam interface *slave*, acoplado ao barramento PLB, no qual o processador PPC405 atua como *master*. Cada um dos componentes tem o seu próprio intervalo dos endereços. O processador comunica com os componentes através das instruções de escrita e leitura das respetivas posições da memória.

Existe também um meio de interrupção do processador quando o *hardware microkernel* requerer. A saída *preemption* do componente escalonador é ligada diretamente ao pino da interrupção externa do processador.

É utilizada a linguagem de descrição do *hardware* VHDL na definição da lógica customizada dos periféricos. Foi tirado o proveito das funcionalidades da modelação de alto nível, existentes nessa linguagem, para poder implementar os periféricos de forma genérica e parametrizável. Foram utilizados *statements generate* para replicação das estruturas e *statements generics* para caracterização individual da largura em *bits* de certos sinais, bem como do número de certas estruturas durante

a sua replicação. Os *generics* foram propagados ao longo da hierarquia das entidades do projeto estrutural, sendo presentes também nas entidades do topo. Desta forma, e com o uso correto da funcionalidade de importação da ferramenta CIP, ditos parâmetros tornam-se visíveis e configuráveis a partir da interface do XPS durante a construção do SoC.

A Tabela 5.1 enumera os parâmetros visíveis a partir do XPS e que devem ser definidos pelo utilizador aquando integração do *hardware microkernel* num SoC.

Tabela 5.1: Parâmetros da configuração do *hardware microkernel*

Generic	Scheduler	SIC	Timers	Exemplo
TASK_NUMBER	X	X	X	16
TIMERS_WIDTH			X	32
EXT_INT_NUMBER		X		4
EXT_INT_CONFIG		X		3344

O parâmetro TASK_NUMBER é presente nos todos componentes e configura o número de tarefas que o *microkernel* irá suportar. Desta forma, no escalonador, este parâmetro define o numero de posições da estrutura que manuseia a informação das tarefas necessária para o escalonamento e indexada pelo identificador da tarefa, o que, por sua vez, define o número de níveis da árvore binária utilizada para extração da tarefa mais prioritária. No componente *timers*, este parâmetro define o número de temporizadores dedicados, indexados mais uma vez pelo identificador da tarefa. Já no caso do controlador das interrupções, este parâmetro não altera significativamente a estrutura interna deste, resumindo-se a definição da largura dos vetores das entradas *t_enable_bit* e *t_valid_bit* utilizados para *feedback* do escalonador para controlador. Este parâmetro deve ser sempre definido com o valor igual à potencia de dois e os exemplos são: 8, 16, 32 (tarefas). Obviamente que o valor definido deve ser o mesmo para todos os componentes, resultando num erro do XPS no caso contrário.

O parâmetro TIMERS_WIDTH define a largura em *bits* dos temporizadores dedicados. Sendo o valor por defeito igual ao 32.

O parâmetro EXT_INT_NUMBER define o número de entradas do controlador das interrupções, isto é, o número de possíveis fontes de interrupção que podem ser ligados ao controlador das interrupções. Este parâmetro não tem restrição na sua definição, sendo a única limitação a área da FPGA, que não será o problema para um número comum de interrupções externos nos sistemas embebidos.

O parâmetro EXT_INT_CONFIG configura o modo da interrupção das respetivas entradas. Este parâmetro é compreendido num número inteiro, em que cada digito corresponde ao modo da interrupção de uma entrada (o número dos dígitos deve corresponder ao número das entradas). O digito mais a esquerda (mais significativo) corresponde ao fonte da interrupção com índice 0, o próximo digito - fonte com índice 1, e por aí em diante. O controlador das interrupções implementa quatro modos da interrupção: "1" - nível alto, "2" - nível baixo, "3" - flanco ascendente, "4" - flanco descendente. Assim sendo, para o caso exemplificado na tabela, o controlador das interrupções irá ter quatro fontes de interrupção (entradas), entre quais: *irq_array(0)* e *irq_array(1)* seráo configurados com o modo flanco ascendente, enquanto *irq_array(2)* e *irq_array(3)* - flanco descendente.

5.3 Registos virtuais

Os componentes desenhados, que integram *hardware microkernel*, seguem todos a mesma arquitetura. O seu espaço de endereços pode ser visto como um conjunto dos registos virtuais, cujas leituras ou escritas desencadeiam operações específicas sobre as estruturas de dados que compreendem estes periféricos. O endereço e o modo de operação (leitura ou escrita) codificam certa operação sobre certa entidade. Assim sendo, os *bits* mais significativos do registo virtual definem o periférico ao qual este pertence (endereço base do periférico), enquanto os restantes *bits* do endereço podem codificar o comando e o índice de certa entidade, caso este comando se aplique a um conjunto de entidades indexados. Os comandos de escrita podem ainda passar ao periférico um argumento de 32 *bits*, enquanto os comandos da leitura para além de desencadear um processo específico podem também receber um *feedback* do lado do periférico.

Para conseguir um *template* que servirá de base para conceção dos componentes que implementam a abordagem descrita acima, foi usada a ferramenta CIP. No que diz respeito à configuração do IPIF, podem ser desmarcadas todas as configurações adicionais. Já na configuração do IPIC, obrigatoriamente deve ser selecionado o sinal Bus2IP_Addr, que por defeito é desabilitado.

5.4 Projeto genérico

Esta secção apresenta o projeto genérico do periférico que possui interface *slave* PLB, usa o mecanismo IPIF + IPIC, gerado automaticamente pela ferramenta CIP, e que, a partir dos endereços das instruções da leitura e escrita emitidas pelo *master* PLB, descodifica os comandos que desencadeiam operações específicas sobre a sua estrutura de dados. Basicamente, este periférico deve implementar uma maquina de estados que deteta a presença do evento da leitura ou da escrita, descodifica o comando presente no endereço, dependendo do comando atua sobre os sinais do controlo da lógica associada e, por fim, sinaliza o barramento sobre o fim da transação.

A interação com barramento é feita através dos sinais de entrada e saída da interface IPIC que é mais simplista do que a interface PLB e que permite que a lógica do utilizador interage com barramento com uma maior facilidade. Os sinais do IPIC usados nos componentes do *hardware microkernel* são: Bus2IP_Clk, Bus2IP_Reset, Bus2IP_Addr, Bus2IP_Data, Bus2IP_RdCE, Bus2IP_WrCE, IP2Bus_Data, IP2Bus_RdAck e IP2Bus_WrAck. Os nomes destes sinais são auto-explicativos, apenas letras CE no Bus2IP_RdCE e Bus2IP_WrCE podem levantar alguma questão e significam *chip enable*. Bus2IP_RdCE e Bus2IP_WrCE são vetores do tamanho do número de *chip enables* que o IPIF descodifica para o espaço de endereços do periférico. Quando na configuração do IPIF, durante a utilização do CIP, é selecionado o serviço dos registos acessíveis a partir do *software*, a largura destes vetores é igual ao número dos ditos registos. Visto que durante a configuração o serviço dos registos acessíveis a partir do *software* não foi selecionado, a largura dos vetores Bus2IP_RdCE e Bus2IP_WrCE é igual a um *bit* (valor por defeito), existindo apenas um *chip enable* para a transação da escrita (Bus2IP_WrCE(0)) e um outro para a transação da leitura (Bus2IP_RdCE(0)).

A Figura 5.2 apresenta um diagrama SMChart genérico desenhado para endereçar o problema estipulado, enquanto a Tabela 5.2 enumera os nomes *alias* utilizados no dito diagrama por razões de "compactidade" e legibilidade. As propriedades deste tipo do diagrama são descritas na secção 2.6. Trata-se de uma ferramenta para modelação das maquinas de estado, que utiliza a nomenclatura do *flowchart*, habitual no projeto de *software*, mas que deve cumprir um conjunto de regras durante a sua construção. O diagrama resultante é equivalente ao *state graph* adequado, mas apresenta uma maior legibilidade e se transpõe com uma maior facilidade para os *statements* da linguagem de descrição do hardware. Apenas três

tipos dos componentes constituem SMChart: bloco do estado (símbolo retangular), bloco de decisão (símbolo com a forma do diamante), e bloco condicional de saídas (símbolo retangular com os vértices curvos). No diagrama apresentado é utilizada a nomenclatura estendida e que é coberta em detalhe na secção 2.6.

Tabela 5.2: Lista dos nomes *alias* aplicados no diagrama da Figura 5.2

Alias	Referencia regular
cmd	Bus2IP_Addr(I_CMD_L to I_CMD_H)
id is	Bus2IP_Addr(I_ID_L to I_ID_H)
Write_PLB	Bus2IP_WrCE(0)
Read_PLB	Bus2IP_WrCE(0)
data_PLB	Bus2IP_Data(32-REC_WIDTH to 31)
data32_PLB	Bus2IP_Data
no_PLB	Bus2IP_RdCE(0)='0' and Bus2IP_WrCE(0)='0'

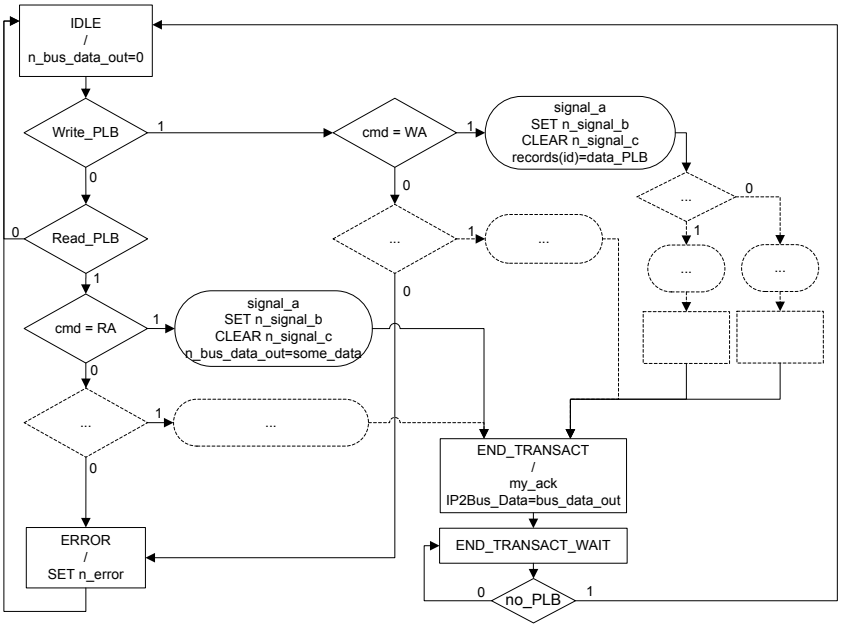


Figura 5.2: Diagrama SMChart genérico do mecanismo da interação com IPIC

Como pode ser observado no diagrama, a maquina de estados desenhada, que implementa o mecanismo de interação com o barramento PLB com recurso a interface IPIC, apresenta apenas quatro estados por defeito: IDLE, END_TRANSACT, END_TRANSACT_WAIT e ERROR. Um periférico específico pode implementar os seus estados auxiliares particulares e desencadeados devido ao certo comando da escrita ou leitura. Para os efeitos da exemplificação, ditos possíveis estados auxiliares, desencadeados por um comando da escrita, são representados no diagrama através dos retângulos tracejados.

A maquina de estados permanece no estado IDLE até surgir a transação da escrita ou leitura. Quando uma das duas condições é satisfeita, o comando é descodificado através da comparação sucessiva dos comandos reconhecidos para a operação em causa com o respetivo conjunto dos *bits* do endereço. No diagrama é apresentada a descodificação de apenas dois comandos: um reconhecido para operação da escrita e outro da leitura. Os blocos de decisão tracejados (figuras em forma de diamante) visam mostrar o lugar no diagrama onde é feita a descodificação de outros comandos, caso estes existem no periférico. Um comando descodificado pode resultar na ativação de um sinal do controlo, e/ou na operação SET/CLEAR sobre um sinal do controlo que tem memória, e/ou na atribuição do argumento colocado no barramento de dados à certa estrutura de dados do periférico, sendo o próximo estado END_TRANSACT ou um estado especifico do periférico em causa que posteriormente irá transitar para o END_TRANSACT.

Quando for detetada a presença da escrita ou leitura, mas o comando codificado no endereço da operação não corresponder a nenhum comando reconhecido, ocorre a transição para o estado ERROR, a partir do qual, por sua vez, ocorre a transição para o estado IDLE.

Como foi estipulado no enunciado do problema, o periférico deve sinalizar o barramento sobre o fim da transação. Isto é feito nos estados END_TRANSACT e END_TRANSACT_WAIT. Primeiro é feito um pulso no sinal my_ack que é traduzido no IP2Bus_RdAck(0) ou IP2Bus_WrAck(0), recorrendo a lógica auxiliar. E de seguida, o periférico espera que o barramento responde ao *acknowledge* com a desativação do sinal *chip enable*, para poder transitar para o estado IDLE que irá esperar pela nova transação.

As listagens 5.1 e 5.2 apresentam, respetivamente, a primeira e a segunda metade do código VHDL que implementa o periférico estipulado, a partir do diagrama presente na Figura 5.2. Está presente quase por inteiro a implementação interna (*architecture body*) da entidade user_logic presente no ficheiro user_logic, automaticamente gerado pela ferramenta CIP. Por defeito, esta entidade possui a interface IPIC e é instanciada num módulo superior que implementa o mecanismo IPIF e possui a interface *slave* PLB.

Os lugares no código que pressupõem a extensão, representada no diagrama por blocos com contornos tracejados, são marcados com os comentários VHDL que começam por `--MORE`.

Para os efeitos de exemplificação são usadas constantes ID_WIDTH,

CMD_WIDTH e REC_WIDTH que podiam ser declarados através dos *statements generic* na definição da entidade e depois atribuídos durante instanciação da entidade. A constante ID_WIDTH indica o número de *bits* que constituem o identificador para os comandos que operam sobre entidades indexadas, enquanto a constante CMD_WIDTH indica o número de *bits* que codificam o comando. Por sua vez, a constante REC_WIDTH define a largura em *bits* de uma posição do vetor *records*.

As constantes I_ID_H, I_ID_L, I_CMD_H e I_CMD_L definem a codificação do endereço das operações sobre registos virtuais. A composição do endereço definido por estas constantes é ilustrada na Figura 5.3.

0	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
Endereço Base do IP																C0	C1	C2	C3	-	-	-	-	-	I0	I1	I2	-	-	-	-

Figura 5.3: A composição do endereço do registo virtual

As quatro linhas do código que atribuem valores a estas constantes possuem dois *magic numbers* (27 e 8) que merecem ser explicados. Antes de mais, no código VHDL produzido pelo Xilinx os vetores são normalmente declarados com a ordem crescente da numeração dos respetivos elementos (é usada a palavra reservada `to`), sendo que no caso do Bus2IP_Addr std_logic_vector o *bit* mais significativo tem o índice "0", enquanto o menos significativo - "31".

A constante I_ID_L não é definida com o valor "31" porque o argumento das instruções `stw` e `lwz`, que vão ser utilizados para interface com os periféricos, deve ser um endereço alinhado em 4 *bytes*, ou seja, os dois *bits* menos significativos devem ser nulos. A constante I_ID_L também não é definida com o valor "29", mas sim com o valor "27", para uma maior facilidade na implementação e rápida interpretação das simulações durante a fase de desenvolvimento, porque desta forma o índice é alinhado ao nível dos *nibbles* e o endereço pode ser facilmente composto "à mão" e interpretado, quando no formato hexadecimal.

O número 8 na diferença entre I_ID_L e I_CMD_L faz com que, independentemente do uso, estão sempre reservados 8 *bits* para o índice (um máximo de 256 elementos indexados). O comando, desta forma, também é alinhado ao *nibble*.

Com está definição das constantes o espaço de endereços que o periférico ocupa é definido por 16 *bits*, o que corresponde a 64KB. De notar que, quando o serviço controlador das interrupções do IPIF é selecionado durante a configuração, mais um *bit* é utilizado para distinguir entre o espaço de endereços do periférico e do correspondente controlador das interrupções.

```
architecture IMP of user_logic is
  --generics --> G_CMD_WIDTH = 4, G_ID_WIDTH = 3, G_REC_WIDTH = 8
  constant I_ID_H   : integer := 27;
  constant I_ID_L   : integer := I_ID_H-G_ID_WIDTH+1;
  constant I_CMD_H  : integer := I_ID_H-8;
  constant I_CMD_L  : integer := I_CMD_H-G_CMD_WIDTH+1;
  --"0000" is reserved by now
  constant C_WA       : std_logic_vector(0 to (I_CMD_H-I_CMD_L)) := "0001";
  constant C_RA       : std_logic_vector(0 to (I_CMD_H-I_CMD_L)) := "0010";
  --MORE commands here
  type fsm_state_type is
  (
    IDLE_STATE, END_TRANSACT_STATE, END_TRANSACT_WAIT_STATE,  ERROR_STATE
  --MORE states here
  );
  signal state, n_state                 : fsm_state_type;
  signal error, n_error                 : std_logic;
  signal bus_data_out, n_bus_data_out   : std_logic_vector(0 to 31);

  alias command : std_logic_vector(0 to (I_CMD_H-I_CMD_L))
                  is Bus2IP_Addr(I_CMD_L to I_CMD_H);
  alias id      : std_logic_vector(0 to (I_ID_H-I_ID_L))
                  is Bus2IP_Addr(I_ID_L to I_ID_H);
  alias dataPLB : std_logic_vector(0 to (REC_WIDTH-1))
                  is Bus2IP_Data(32-REC_WIDTH to 31)

  signal my_ack : std_logic;

  signal a                  : std_logic;
  signal b, n_b, c, n_c     : std_logic;
  signal records, n_records :std_logic_vector(0 to (2**G_ID_WIDTH)*G_REC_WIDTH-1);
  signal enable_next <= set_enable_field(FSL_S_Data(32-h to 31), enable, '1');
  --MORE signals here
begin
  ACK_PROC : process(my_ack, Bus2IP_RdCE, Bus2IP_WrCE)
  begin
    if (Bus2IP_RdCE(0) = '1') then
      IP2Bus_RdAck <= my_ack;
    else
      IP2Bus_RdAck <= '0';
    end if;
    if (Bus2IP_WrCE(0) = '1') then
      IP2Bus_WrAck <= my_ack;
    else
      IP2Bus_WrAck <= '0';
    end if;
  end process ACK_PROC;

  FSM_STATE_PROC : process (Bus2IP_Clk, Bus2IP_Reset) is
  begin
    if (Bus2IP_Clk'event and (Bus2IP_Clk = '1')) then
      if( Bus2IP_Reset = '1' ) then
        b                   <= '0';
        c                   <= '0';
        records             <= (others => '0');
        --MORE signals here
        bus_data_out        <= (others => '0');
        error               <= '0';
        state               <= IDLE_STATE;
      else
        b                   <= n_b;
        c                   <= n_c;
        records             <= n_records;
        --MORE signals here
        bus_data_out        <= n_bus_data_out;
        error               <= n_error;
        state               <= n_state;
      end if;
    end if;
  end process FSM_STATE_PROC;
--...
```

Listagem 5.1: Primeira metade da implementação interna da entidade *user_logic*

```
--...
FSM_LOGIC_PROC : process (
    Bus2IP_WrCE(0), Bus2IP_RdCE(0), state, bus_data_out,
    error, command, Bus2IP_Data, Bus2IP_Addr,
    b, c --MORE signals here
    ) is
  begin
    my_ack                <= '0';
    IP2Bus_Data           <= (others => '0');

    a                     <= '0';
    n_b                   <= b;
    n_c                   <= c;
    --MORE signals here

    n_state               <= state;
    n_bus_data_out        <= bus_data_out;
    n_error               <= error;
    case state is
      when IDLE_STATE =>
        n_bus_data_out <= (others => '0');
        if (Bus2IP_WrCE(0) = '1') then
          case command is
            when  C_WA =>
              a      <= '1';
              n_b    <= '1';
              n_c    <= '0';
              n_records   <= set_records_field(id, records, dataPLB);
              n_state     <= END_TRANSACT_STATE;

            -- MORE write commands here
            when others =>
              n_state     <= ERROR_STATE;
          end case;
        elsif (Bus2IP_RdCE(0) = '1') then
          case command is
            when C_RA =>
              a      <= '1';
              n_b    <= '1';
              n_c    <= '0';
              n_bus_data_out  <= X"DEADBEEF";
              n_state         <= END_TRANSACT_STATE;

            --MORE read commands here
            when others =>
              n_state        <= ERROR_STATE;
          end case;
        end if;

      when END_TRANSACT_STATE =>
        IP2Bus_Data <= bus_data_out;
        my_ack      <= '1';
        n_state     <= END_TRANSACT_WAIT_STATE;

      when END_TRANSACT_WAIT_STATE =>
        if( Bus2IP_RdCE(0)='0' and Bus2IP_WrCE(0)='0' ) then
          n_state <= IDLE_STATE;
        else
          n_state <= state;
        end if;
      --MORE states here
      when ERROR_STATE =>
        n_error   <='1';
        n_state   <= IDLE_STATE;
      when others =>
        n_state   <= ERROR_STATE;
    end case;
  end process FSM_LOGIC_PROC;
end IMP;
```

Listagem 5.2: Segunda metade da implementação interna da entidade *user_logic*

O processo ACK_PROC implementa a lógica auxiliar, já referida nesta secção, que traduz o sinal *my_ack* no *acknowledgement* da transação da leitura ou da transação da escrita. Assim, na presença da transação da leitura, o pulso no sinal my_ack que ocorre no estado END_TRANSACT será traduzido num pulso no sinal IP2Bus_RdAck.

Mais um aspeto de referir, em relação a primeira metade da implementação da entidade, é a definição do vetor *records*. Aqui trata-se de um vetor dos *records*, cuja dimensão é definida pelo ID_WIDTH, e em que cada *record* tem a largura de REC_WIDTH *bits*. Ou seja, é um vetor bidimensional, em que cada dimensão é definida pelo parâmetro *generic*. Este cenário levanta problemas na compilação e a solução adotada nesta dissertação para este tipo de problema resume-se na declaração deste vetor bidimensional na forma do vetor unidimensional, com a largura igual ao produto das duas dimensões, e implementação da função de acesso aos seus campos a partir do índice. A dita função não está presente nas listagens 5.1 e 5.2 mas pode ser vista na Listagem 5.3

```
function set_records_field(
  index       : std_logic_vector (0 to G_ID_WIDTH-1);
  records     : std_logic_vector (0 to (2**G_ID_WIDTH)*G_REC_WIDTH-1);
  input_value : std_logic_vector (0 to G_REC_WIDTH-1))
return std_logic_vector is

  variable records_res : std_logic_vector (0 to (2**G_ID_WIDTH)*G_REC_WIDTH-1);
  variable index_int   : integer range 0 to (2**G_ID_WIDTH)-1;

begin
  index_int := CONV_INTEGER(index);
  records_res := records;
  records_res(index_int*G_REC_WIDTH to (index_int+1)*G_REC_WIDTH-1):=input_value;
  return records_res;
end function set_records_field;
```

Listagem 5.3: Função VHDL set_records_field

A segunda metade da implementação da entidade *user_logic* é uma tradução direta do SMChart desenhado. O *template* resultante é um periférico pronto para interagir com processador e que dá também o exemplo da operação sobre um sinal sem memória (a), dois sinais com memoria (b e c), um vetor bidimensional (*records*), um comando de escrita e um comando de leitura. Para alem da personalização destes sinais e comandos a extensão deste *template* pode ser feita nas posições de código indicados com o comentários que começam por `--MORE`.

O projeto dos componentes do *hardware microkernel* baseou-se neste *template*. O código resultante é de fácil manuseio e extensão. A extensão do periférico com um novo comando tem um custo do desenvolvimento baixo.

5.5 Implementação

Após a apresentação da visão global sobre a composição do *hardware microkernel* e apresentação detalhada do *template* especialmente projetado e usado no desenho dos componentes, nesta secção, será dado o foco sobre as funcionalidades e particularidade da implementação de cada componente.

5.5.1 Escalonador

O componente do *hardware microkernel* que implementa as funcionalidades do escalonador é o mais complexo dos três. A Figura 5.4 ilustra o seu módulo do topo junto com as respetivas interfaces.

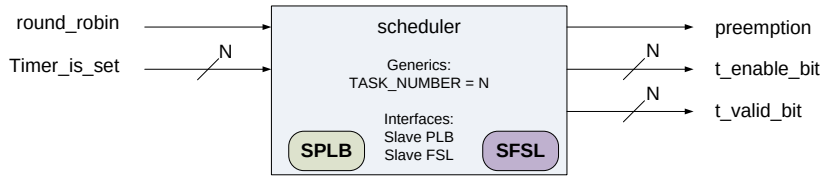


Figura 5.4: Componente escalonador do *hardware microkernel*

Da mesma forma como no caso dos outros componentes, o projeto deste se baseou no *template* apresentado na secção anterior e o seu módulo do topo possui interface *slave* PLB para receção dos comandos enviados a partir do processador.

Do ponto de vista do utilizador, o periférico possui apenas um parâmetro configurável, que é o número de tarefas (TASK_NUMBER) e que obrigatoriamente deve ser igual a potência de dois (2^n).

Seguindo os requisitos estipulados, o escalonador implementa a estratégia do escalonamento baseada nas prioridades. Possui também a possibilidade de emular o comportamento *round-robin*, como a estratégia do desempate. Para isso, apresenta uma entrada (`round_robin`), que deve ser ligada a um temporizador externo, caso dita funcionalidade for pretendida.

A entrada `Timer_is_set` é um vetor com a largura igual ao número de tarefas do escalonador. Os sinais deste vetor podem ser ligados aos *timers* externos, permitindo assim, a temporização das diferentes tarefas de forma independente. O vetor pode ser ligado diretamente ao componente *Timers* do *hardware microkernel* concebido para efeito e que possui a respetiva interface.

O componente também é dotado com a interface *slave* FSL. Uma palavra recebida através deste canal corresponde ao identificador da tarefa que deve ser acordada (habilitada).

Esse canal pode ser ligado ao controlador das interrupções do *hardware microkernel*, caso for pretendida a unificação do espaço de prioridades das tarefas e interrupções. As saídas do periférico `t_enable_bit` e `t_valid_bit` correspondem aos vetores `enable` e `valid` da informação das tarefas. Cada tarefa tem um *bit* `enable` e um *bit* `valid` que indicam, respetivamente, se a tarefa está pronta para executar e se a tarefa existe. Ditos vetores também devem ser ligados ao controlador das interrupções do *hardware microkernel* se este for usado.

Por fim, o sinal de saída *preemption* serve para sinalizar o CPU acerca da necessidade da preempção (comutação de contexto). Sempre que a preempção é habilitada e de um ciclo para outro é detetada uma alteração a saída do extrator da tarefa mais prioritária, o dito sinal passa a ter o valor "1". Este deve ser ligado a uma interrupção externa do processador e o seu flanco ascendente irá informar o processador sobre a necessidade de efetuar a comutação de contexto.

Árvore binária

A principal tarefa do escalonador é guardar a informação das tarefas presentes no sistema operativo e extrair a qualquer momento a tarefa com a maior prioridade das prontas para execução. Como a arquitetura em *hardware*, que pode ser utilizada no escalonamento, foi decidido explorar a árvore binária. Um possível enquadramento desta arquitetura no suporte ao sistema operativo foi apresentado em (Kohout et al., 2003) e serviu de base para conceção do "coração" do escalonador do *hardware microkernel*. O conceito em causa pode ser visto na Figura 5.5.

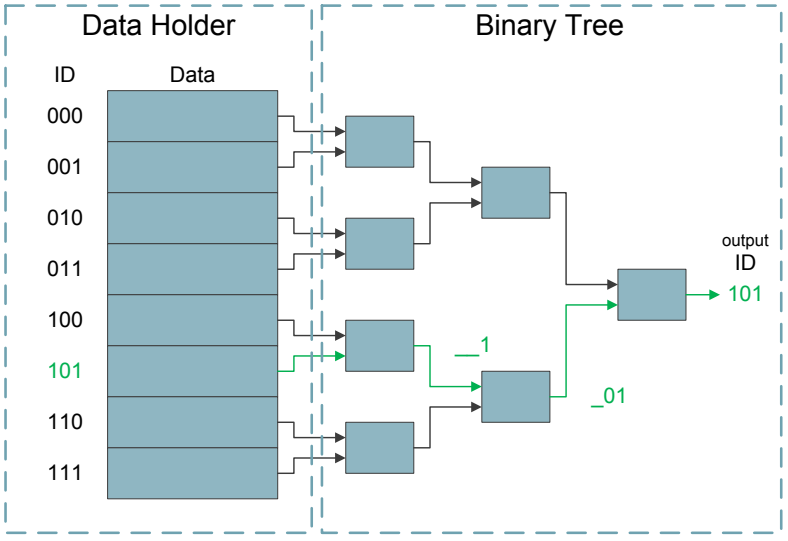


Figura 5.5: Ilustração do conceito utilizado no projeto do escalonador

O escalonador implementado pode ser visto como constituído por duas partes: *Data Holder* e *Binary Tree*. O *data holder* está intrinsecamente ligado a maquina de estados de descodificação dos comandos PLB, enquanto a árvore binária foi implementa como sendo uma entidade separada. O número de tarefas suportadas é fixo, igual ao 2^n e definido pelo utilizador aquando instanciação do *hardware microkernel*. As estruturas em *hardware* que guardam a informação das tarefas são indexadas pelo identificador da tarefa. A dimensão da árvore binária depende do numero de tarefas suportadas. No exemplo apresentado na figura, o escalonador suporta um máximo de 8 tarefas, sendo que o numero de níveis da árvore binária é igual a 3 ($log_2 N$), enquanto o número de nós de comparação é igual a 7 ($N - 1$). Para além de guardar a informação das tarefas, o *data holder* também a acondiciona com recurso à lógica auxiliar, proporcionando à árvore binária a informação necessária, que se resume a prioridade da tarefa, a sua admissão para escalonamento (tarefa existente, habilitada, não bloqueada pelo evento temporal, nem pelo evento da sincronização) e mais um *bit* relacionado com a emulação do comportamento *round-robin* e que será coberto posteriormente.

De notar que os identificadores das tarefas não são fornecidos à árvore a partida e o identificador a saída da árvore é formado ao longo da propagação dos resultados dos nós de comparação. A interface de um nó de comparação pode ser vista na Figura 5.6.

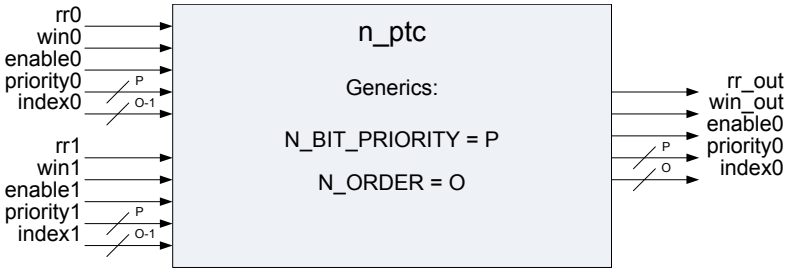


Figura 5.6: Módulo do topo correspondente a um nó de comparação da árvore binária

Os parâmetros configuráveis de um nó é o seu nível na árvore binária (define a largura do índice de entrada e índice de saída), e o número de *bits* que compreendem prioridade. O primeiro parâmetro é gerido pelo processo da instanciação automática recursiva aquando construção da árvore, enquanto o segundo parâmetro é inicializado a partir do escalonador com um valor fixo e igual a 8. Os sinais *rr* e *win* são sinais auxiliares que participam na emulação do comportamento *round-robin*, onde o sinal *rr* é fornecido pelo *data holder*, enquanto o *win* é sempre nulo do ponto de vista do primeiro nível da árvore binária. Sem contar com a emulação do *round-robin*, a tabela de verdade que define a decisão da propagação de uma tarefa ou outra é apresentada na Tabela 5.3. O sufixo "0" ou "1" indica, respetivamente, que o sinal provem da tarefa com índice inferior ou superior, ou seja, duma tarefa que graficamente se encontra acima ou abaixo nas Figuras 5.5 e

5.6. O valor do `output` corresponde a um sinal interno da entidade que é ligado aos multiplexadores, propagando para saída os sinais correspondentes à tarefa "0" ou "1".

Tabela 5.3: Tabela de verdade da decisão do nó de comparação

en0	en1	pr1>pr2	pr2>pr1	output
x	0	x	x	0
0	1	x	x	1
1	1	1	x	0
1	1	0	1	1
1	1	0	0	0

Para concluir a análise do funcionamento dos nós de comparação e consequentemente da árvore binária, segue-se a explicação do mecanismo da emulação do comportamento *round-robin*. De forma resumida, o *round-robin* é uma técnica aplicada no caso da empate das prioridades, que faz as tarefas que partilham a maior prioridade executar alternadamente e com um *time slice* associado. A noção da ordenação FIFO não existe na arquitetura árvore binária, o que não impede a implementação do principal intuito do *round-robin* que é a execução alternada. A solução utilizada nesta dissertação para emulação do comportamento *round-robin* na arquitetura árvore binária resume-se a seguinte: um temporizador externo emita um pulso ao *data holder* com período igual ao *time slice*, quando este sinal é recebido a tarefa cujo identificador naquele momento se encontra a saída da árvore é marcada, pondo o seu *bit* `rr` ao "1" e pondo os *bits* `rr` das restantes tarefas que partilham a prioridade com esta ao "0". Pretende-se agora que a árvore escolhe para execução a tarefa com a maior prioridade e que está "abaixo" (graficamente nas Figuras 5.5 e 5.6) daquela que acabou de ser marcada. Caso não existir nenhuma tarefa empatada "abaixo", deve ser escolhida a tarefa que se encontra mais perto para o "topo". Este problema foi solucionado com acréscimo de mais um *bit* propagado entre os nós. Este *bit* é denominado `win`, indica que a tarefa deve ser escolhida e torna-se "1" quando num nó de decisão aparecem tarefas empatadas e a tarefa "acima" é marcada com `rr`. A Tabela 5.4 resume as regras de propagação e decisão com base nos sinais `rr` e `win`. Esta tabela é uma extensão da Tabela 5.3 para o caso da empate das prioridades de duas tarefas habilitadas.

Tabela 5.4: Tabela de verdade para o caso da empate das prioridades

rr0	rr1	win0	win1	output	rr_out	win_out
1	x	x	x	1	0	1
0	1	x	x	0	1	0
0	0	1	x	0	0	1
0	0	0	1	1	0	1
0	0	0	0	0	0	0

A Figura 5.7 exemplifica um cenário no qual quatro tarefas partilham a maior prioridade (realçados com a cor verde). As setas curvas indicam a ordem de execução alternada que se pretenda conseguir com a utilização dos *bits* auxiliares `rr` e `win`. No exemplo a tarefa "001" está marcada com *bit* `rr` a "1". Isto resulta na escolha da tarefa "011" para execução, que será marcada com o *bit* `rr` na próxima chegada do sinal *round-robin* cedendo o acesso ao CPU à tarefa "110", e por aí em diante.

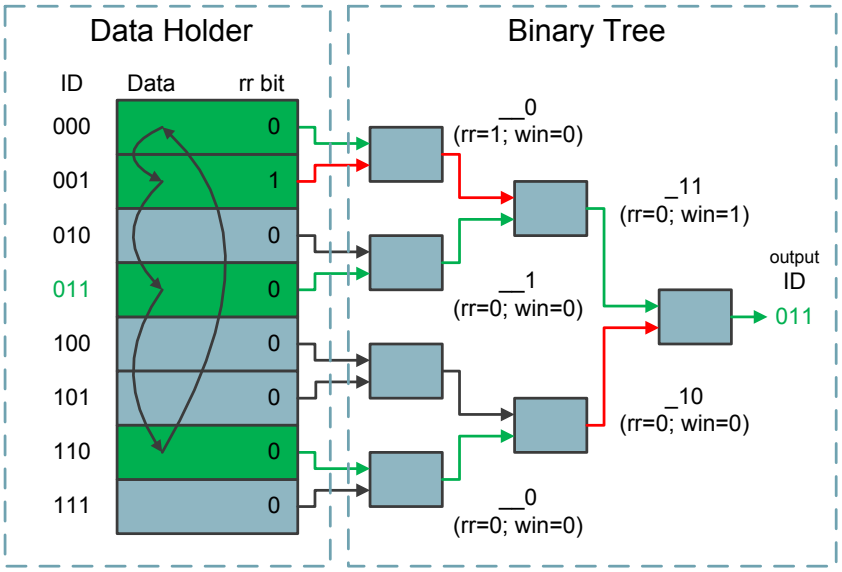


Figura 5.7: Exemplificação do comportamento *round-robin*

A entidade árvore binária é implementada de forma genérica, sendo que um dos parâmetros da sua configuração é o número dos seus níveis. A construção da árvore de forma genérica não é trivial e implica a instanciação recursiva. As técnicas do desenvolvimento VHDL das estruturas recursivas foram consultadas em (Ashenden, 1994). Basicamente, a instanciação recursiva da árvore binária pode ser resumida em duas frases: uma árvore da ordem *n* é constituída por duas árvores da ordem *n-1*, cujas saídas são ligadas a um nó de comparação; uma árvore da ordem 0 tem as entradas só para uma tarefa e que são ligadas diretamente às correspondentes saídas. A Figura 5.8 visa exemplificar a dita recursividade. Lado a lado se encontram uma representação ilustrativa da árvore binária de 4 níveis (suporta 16 tarefas) e o *output* do RTL *viewer* integrado nas ferramentas do desenvolvimento Quartus II do produtor Altera (Altera, 2008). Para realçar as correspondências das partes de uma representação com a outra são usados retângulos de diferentes cores. Pode ser visto, que uma árvore de nível 4 é constituída por um nó de comparação (vermelho) e duas árvores de nível 3 (azul), que por sua vez são constituídas por um nó de comparação (verde) e duas árvores de nível 2 (laranja). Durante a fase de aprendizagem de técnicas do desenvolvimento das estruturas recursivas na linguagem VHDL, foi observada uma falta de eficiência das ferramentas Xilinx no que diz respeito a visualização dos esquemáticos RTL. Aquando validação da instanciação recursiva da árvore

binária, o *output* do RTL *viewer* do Xilinx mostrava apenas o módulo do topo com ligações no seu interior a flutuar ou ligadas a massa. Após inúmeras tentativas de localizar o erro no projeto, foi decidido instalar as ferramentas do desenvolvimento do Altera Quartus II para experimentar o respetivo RTL *viewer*. Desta vez o *output* correspondia às expetativas, o que deu para concluir que a instanciação recursiva teve sucesso.

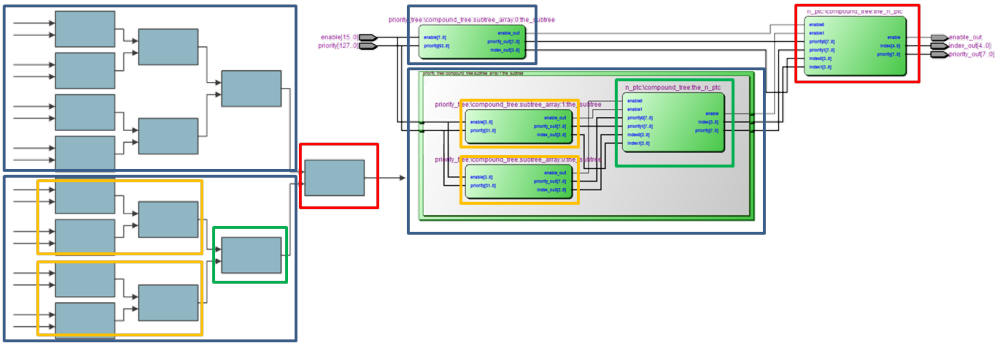


Figura 5.8: Instanciação recursiva da árvore binária

Pipeline na árvore binária

Com aumento do número de tarefas que escalonador suporta, aumenta o numero de níveis da árvore binária o que, por sua vez, leva ao aumento do caminho crítico do circuito digital. Já com 16 tarefas o caminho crítico do circuito projetado ultrapassou os 10ns que representam o limiar para correto funcionamento à frequência do barramento PLB configurada para ser igual ao 100MHz. Este fato levou a necessidade da aplicação da prática *pipeline* na árvore binária. Em todos os níveis da árvore, os nós de comparação deixaram de ser circuitos de lógica puramente combinacional. Esta sequencialização do circuito reduziu o caminho crítico, mas teve o impacto de introduzir um atraso de `n` ciclos (onde `n` igual ao número de níveis da árvore) entre a atualização de uma entrada da árvore e a reflexão desta alteração na saída.

Exclusão mútua

O único recurso de sincronização que foi estipulado nos requisitos do *hardware microkernel* é a exclusão mútua. Este recurso foi implementado de forma trivial e intrinsecamente ligado ao *data holder* do escalonador. O seu projeto definiu a necessidade de duas funcionalidades: desabilitar uma dada tarefa com um dado evento de sincronização; habilitar a tarefa mais prioritária das aquelas que estão desabilitadas por um dado evento de sincronização.

A funcionalidade da desabilitação da tarefa foi implementada através da criação do vetor `events`, que para cada tarefa guarda o identificador do evento associado, caso existir. Quando pelo menos um *bit* do identificador do evento correspondente a uma tarefa é não-nulo, a lógica auxiliar do *data holder* coloca ao zero o sinal *enable* desta tarefa na árvore binária.

O processo de habilitação da tarefa a partir do identificador do recurso de sincronização já é mais complexo. Primeiro, a árvore binária é utilizada para detetar a tarefa mais prioritária entre aquelas que tem assinado o identificador em causa. Para isso, a preempção é desligada e são habilitadas as tarefas cujo identificador do evento é igual ao identificador em causa e desabilitadas todas as outras. Deve ser assegurado o número dos ciclos necessário para atualização da saída da árvore devido ao *pipeline*. No fim destes ciclos a árvore tem na sua saída o identificador da tarefa que deve ser habilitada. Este identificador da tarefa é utilizado para limpar o respetivo identificador do evento. Por fim, a árvore deixa de operar no modo especial, as tarefas voltam ao seu estado normal e a preempção volta a ser habilitada depois de assegurar que a saída da árvore foi de novo atualizada.

Sinal da preempção

O escalonador implementa um pequeno controlador das interrupções. Sempre que a preempção é habilitada e de um ciclo para outro é detetada a alteração do identificador a saída da árvore binária, o sinal de saída `preemption` passa a ter o valor "1". Este sinal deve ser ligado a uma interrupção externa do processador e o seu flanco ascendente irá informar o processador sobre a necessidade de efetuar a comutação de contexto. A rotina do atendimento da dita interrupção consulta, através de um comando de leitura, a tarefa que está a saída da árvore binária e ao mesmo tempo efetua o *acknowledgement* da preempção, fazendo com que o sinal *preemption* volta a ter o valor "0".

Interface

Para poder usufruir do *hardware microkernel*, a parte do sistema operativo que permanece em *software* deve ter uma forma de comunicação com os componentes que o constituem. O escalonador fornece uma interface constituída por 15 comandos que são enumerados na Tabela 5.5

A segunda coluna da tabela indica o modo da operação para qual este comando é reconhecido. Sendo que, a letra *L* significa *Leitura* e a letra *E* - *Escrita*. A terceira coluna indica se o comando é ou não é indexado, sendo com o sinal *X* marcados os comandos cuja codificação no endereço deve ser acompanhada com o identificador da tarefa.

Tabela 5.5: Os comandos

Macro do comando	M*	I*	Descrição
GET_VERSION	L		Lê o numero dos níveis da árvore binária e a versão
EN_INTERRUPT	E		Habilita interrupção
GET_STATE	L		Lê uma palavra com um conjunto de informação, inclusivamente id a saída da árvore binária
GET_STATE_PR_ACK	L		Faz GET_STATE e limpa o sinal da interrupção (preempção)
CR_TASK	E	X	Valida a tarefa com o respetivo id e escreve a sua prioridade no campo correspondente
CR_TASK_EN	E	X	Cria a tarefa como CREATE_TASK e a habilita
REMOVE_TASK	E	X	Invalida a tarefa com o respetivo id
DISABLE_TASK	E	X	Desabilita a tarefa com o respetivo id
AWAKE_TASK	E	X	Habilita a tarefa com o respetivo id
TAKE_EVENT	E	X	Bloqueia tarefa, assinando o identificador do evento
RELEASE_MUTEX	E		Através do identificador do evento desbloqueia uma tarefa
CR_TASK_PTR	L		Cria a tarefa e lê o id desta no caso do sucesso; no lugar da codificação do id passa a prioridade
CR_TASK_EN_PTR	L		Cria a tarefa como CR_TASK_PTR e a habilita
GET_PTR_ACK	L		Lê o apontador da tarefa que está a saída da árvore e binária e limpa a interrupção
SET_PTR	E	X	Atribui apontador à tarefa com o respetivo id

São apresentados 15 comandos diferentes e que cada um é compreendido apenas para leitura ou para escrita. Visto que o modo de operação é codificado de forma implícita pelo barramento, os dois grupos de comandos podem compreender o mesmo espaço de endereços, por exemplo, as macros GET_VERSION e EN_INTERRUPT podem ser definidos com o mesmo valor, sendo distintas pelo facto de ser uma operação de leitura ou escrita. A tabela não inclui um conjunto de comandos relacionados com a ferramenta de medição *ad-hoc* discutida no Capítulo 6.

O diagrama SMChart que implementa o *data holder* do escalonador e a sua interface com o barramento PLB atráves da interface IPIC pode ser consultado na Figura 5.9.

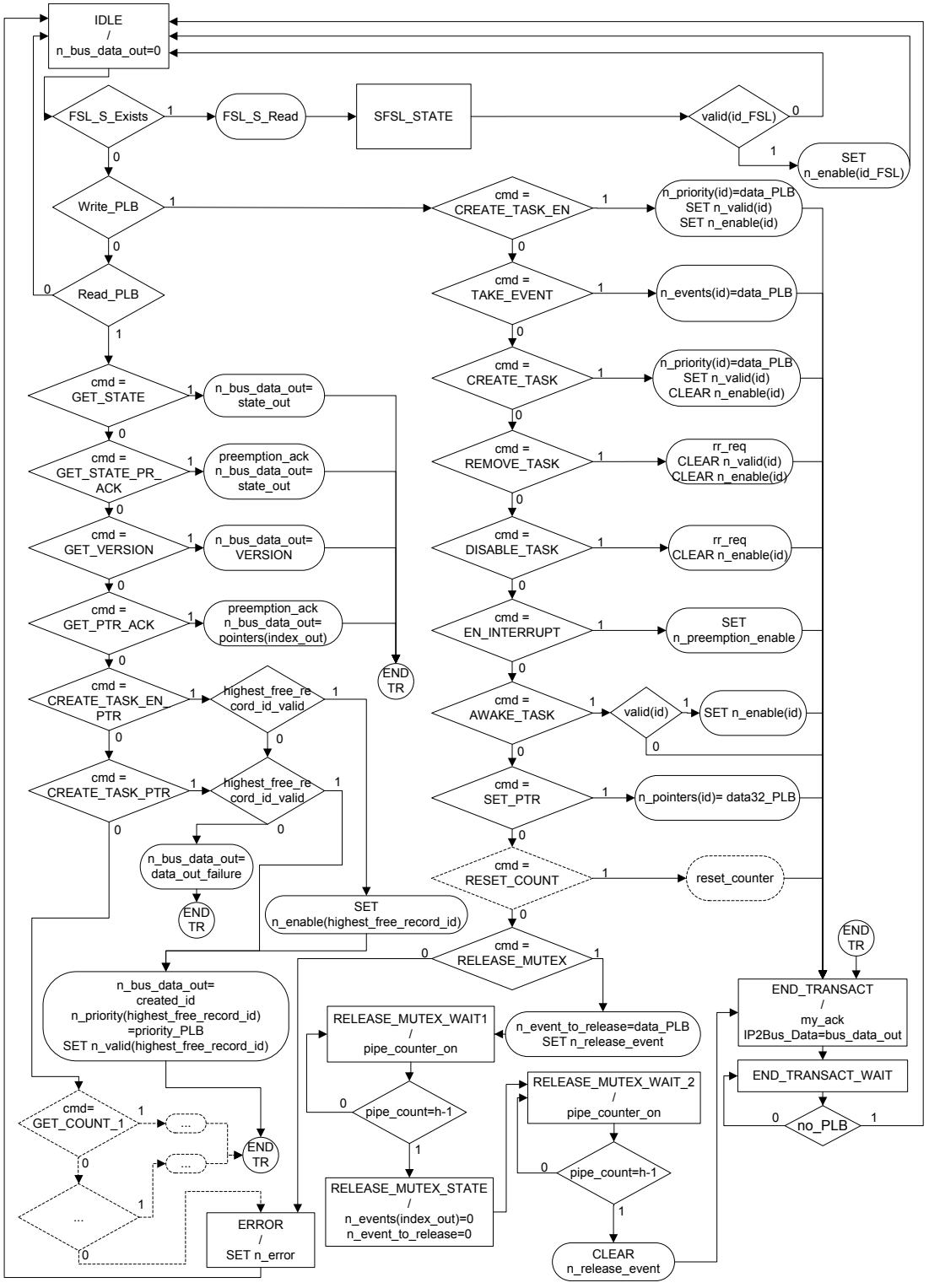


Figura 5.9: Diagrama SMChart que descreve a interface do escalonador

5.5.2 Temporizadores

O componente do *hardware microkernel* que é denominado *Timers* implementa temporizadores dedicados para as tarefas que se encontram no escalonador. O seu módulo de topo pode ser visto na Figura 5.10.

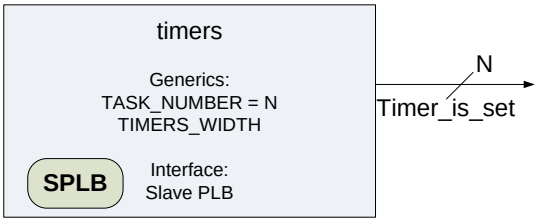


Figura 5.10: Componente *Timers* do *hardware microkernel*

No que diz respeito à interface com o barramento PLB, o componente tem uma única funcionalidade que é colocar o valor enviado via PLB no *timer* correspondente à tarefa, cujo identificador é codificado no endereço do registo virtual. A maquina de estados descrita na forma do diagrama SMChart e que implementa a funcionalidade acima referida pode ser consultada na Figura 5.11.

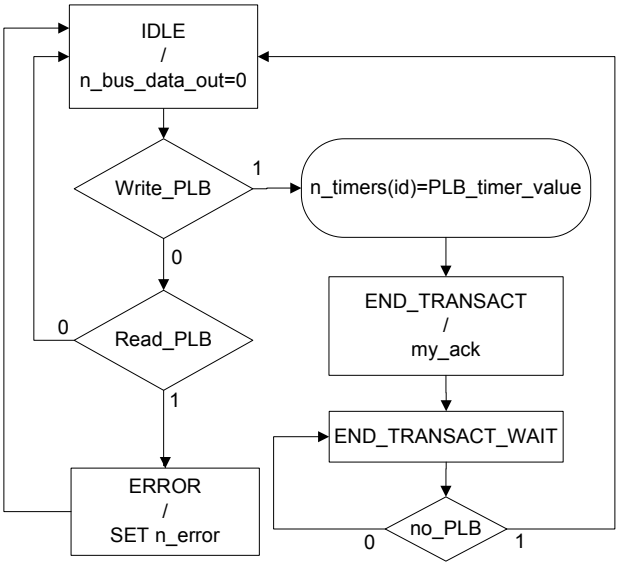


Figura 5.11: Diagrama SMChart que descreve a interface do componente *Timers* do *hardware microkernel*

Os temporizadores são decrementados a cada ciclo do relógio, permanecendo no valor nulo quando o atingem. Para interface dedicado com o escalonador do *hardware microkernel* o componente possui o vetor `Timer_is_set`. Cada *bit* deste vetor é ativo quando pelo menos um *bit* do valor do temporizador correspondente é ativo. A Figura 5.12 mostra o pseudo-código que define o estado dos *bits* do vetor `Timer_is_set`.

Timer_is_set *(a cada ciclo do relogio):*
Para i no intervalo entre 0 e TASK_NUMBER-1:
Se algum bit do Timer(i) é ativo:
Timer_is_set(i)=1,
Senão:
Timer_is_set(i)=0.

Figura 5.12: Pseudo-código que define o estado dos *bits* do vetor `Timer_is_set`

5.5.3 Controlador das Interrupções

O *hardware microkernel* desenhado implementa a unificação de espaço de prioridades das tarefas e interrupções. A contextualização e a motivação para isso foram apresentadas na secção 2.3. O controlador das interrupções SIC (*synchronous interrupt controller*) acondiciona os eventos da interrupção externos dependendo do modo de configuração e, em vez de interromper processador com pedido de interrupção, faz um pedido ao escalonador do *hardware microkernel* para habilitar a tarefa previamente assinada como sendo o *handler* para a fonte de interrupção em causa. O módulo de topo deste componente pode ser visto na Figura 5.13

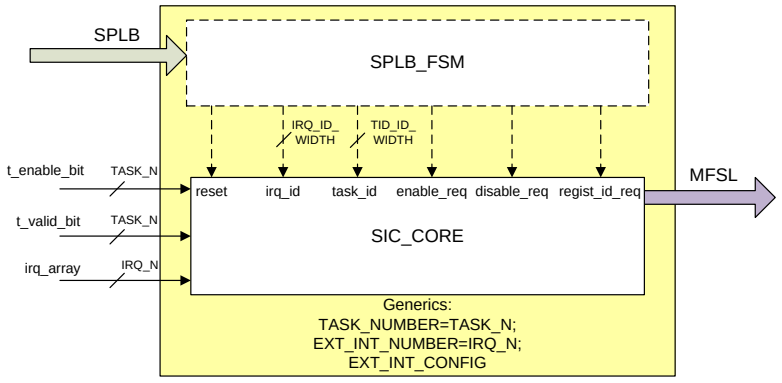


Figura 5.13: Componente SIC (*synchronous interrupt controller*) do *hardware microkernel*

Como já foi referido na secção 5.2, o controlador tem três parâmetros da configuração: número de tarefas no escalonador, número de interrupções externas e a configuração das interrupções externas.

Um projeto notável que num contexto idêntico implementa a unificação do espaço de prioridades é o projeto hthreads (Agron, 2010). Tanto no caso do hthreads, como no caso desta dissertação, o controlador das interrupções, bem como os outros módulos do *microkernel*, são implementados como sendo periféricos ligados ao barramento PLB. O CPU figura no barramento como sendo *master*, enquanto os periféricos assumem o papel do *slave*. Para a comunicação entre o controlador das interrupções e o escalonador, no projeto hthreads optou-se por dotar controlador das interrupções também com a

interface *master* PLB e fazer com que os eventos de escalonamento sejam comunicados ao escalonador através do mesmo barramento. Trata-se de uma opção prática e que não exige qualquer alteração no escalonador, visto que este é desenhado para poder receber os eventos da sincronização enviados pelo CPU e o controlador das interrupções pode utilizar a mesma "linguagem". Ao contrário do projeto hthreads, nesta dissertação foi decidido evitar o aumento do número dos *masters* no barramento, minimizando assim o impacto da natureza indeterminística deste. Foi então considerado que a comunicação entre o controlador das interrupções e o escalonador "merece" a implementação de uma interface dedicada. Sendo assim, foi utilizado um canal FSL (*Fast Simplex Link*) com o controlador das interrupções a atuar do seu lado *master* e o escalonador do lado *slave*.

Como pode ser visto no diagrama que representa o módulo do topo, o núcleo do controlador foi implementado como uma entidade independente, controlada por um conjunto de sinais provenientes do descodificador dos comandos enviados via barramento PLB. O digrama SMChart que implementa a maquina de estados correspondente ao dito descodificador pode ser visto na Figura 5.14.

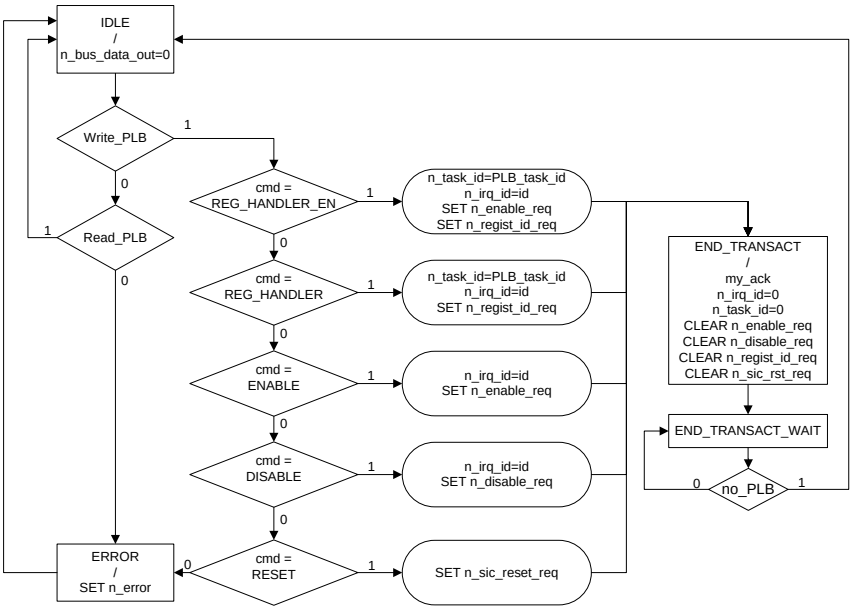


Figura 5.14: O diagrama SMChart que descreve a interface do controlador das interrupções

O núcleo do controlador implementa e manuseia, com base nos sinais do controlo, as tabelas `sic_tid(0:IRQ_NUM-1)(0:TID_WIDTH)` e `sic_enable(0:IRQ_NUM-1)`. Para cada interrupção externa, a primeira tabela guarda o identificador da tarefa assinada como o *handler*, enquanto a segunda - o respetivo *bit enable*.

Quando uma interrupção externa habilitada é detetada, o controlador das interrupções envia via FSL o identificador da tarefa (posição na árvore binária) previamente registada

no escalonador e registada também no controlador das interrupções como sendo o *handler* dessa interrupção. O escalonador recebe identificador e habilita a respetiva tarefa se o seu registo for valido. O CPU é interrompido apenas quando a saída da árvore binária do escalonador mudar.

O modo de operação de cada uma das interrupções externas pode ser configurado da forma independente. Esta configuração é feita através do parâmetro EXT_INT_CONFIG, coberto em detalhe na secção 5.2.

Capítulo 6

Resultados Experimentais

Este capítulo descreve os ensaios efetuados no *kernel* do tempo real assistido por *hardware*, bem como apresenta as ferramentas de medição utilizadas durante esta fase do desenvolvimento.

6.1 Ferramentas de medição

Antes de apresentar os resultados experimentais é feita uma breve descrição das ferramentas de medição usadas:

- Contador dedicado implementado como periférico PLB;
- Registo *time base* do processador PPC405;
- Implementação intrusiva de contadores *ad hoc* dentro do periférico testado;
- Ferramenta *ChipScope Pro*.

6.1.1 Contador dedicado PLB e registo TBL do PPC405

Alguns dos testes limitam-se a medição do tempo que o CPU demora a executar uma sequência de instruções que não contém instruções de salto. Para este tipo de medição, a ferramenta pode basear-se num contador que incrementa com frequência do relógio do processador e duas funções: uma que dá inicio a contagem e que é colocada antes da zona do código avaliado e outra que consulta o valor do contador e que é colocada no fim da zona do código avaliado.

Inicialmente, o referido contador foi implementado em *hardware* e ligado ao processador através do barramento PLB. A escrita (operação *store*) de qualquer valor para endereço base deste periférico colocava contador a zero, enquanto a leitura (operação *load*) deste endereço resultava no acesso ao valor atual do contador. Esta solução foi abandonada devido a natureza indeterminística do barramento PLB e que pode conduzir aos resultados inconsistentes caso o barramento esteja congestionado com outras transações no momento da interação com contador.

A solução adotada baseia-se na utilização do registo *time base* que é proporcionado pela arquitetura *PowerPC embedded-environment*, sendo também implementado no PPC405 (Xilinx, 2002). Trata-se de um registo 64 *bit* que é incrementado durante cada ciclo do relógio do processador. Só foi usada a sua metade menos significativa, denominada TBL, um registo de 32 *bits* que pode ser manipulado com instruções `mttb`, `mftb` ou `mtspr` e `mfspr`. No caso da frequência do processador ser 100MHz, o *overflow* do contador de 32 *bits* só ocorre passando 42 segundos, uma quantia de tempo cuja magnitude está muito acima do aquilo que se pretende medir. Os comandos do início e do fim da medição foram implementados na forma de macros e usando *inline assembly*. A definição destas macros pode ser vista na Listagem 6.1.

```
#define start_time_base()   asm volatile ("ori 0,0,0\n\tli 0,0\n\tmttbl 0\n\t":::)
#define get_time_base(time) asm volatile ("mftb %["#time"]":[time]"=r" (time)::)
```

Listagem 6.1: Definição das macros para manipulação do registo TBL

A instrução que coloca zero no registo TBL (`mttbl 0`) é exatamente a ultima instrução antes da sequência das instruções avaliada e a instrução que lê o valor do TBL (`mftb %["#time"]`) é exatamente a primeira depois da sequência avaliada, pelo que, no caso da execução determinística, está medição apresenta um *offset* nulo e constante. O resultado das medições não é influenciado pelo alinhamento da instrução `mttbl` na memória, nem pelo código que precede ou sucede a medição. Durante a fase da validação desta ferramenta da medição denotou-se que no caso quando a macro *start_time_base()* se encontrava logo após *statement for* da linguagem C, ver Listagem 6.2, o resultado da medição era um ciclo do relógio a mais em relação ao esperado. Instrução `ori 0,0,0` é uma instrução NOP (*no operation*) e o tempo da sua execução é igual ao 1 ciclo do relógio do processador. Assim, para o exemplo apresentado na listagem. o resultado esperado é 6 ciclos, enquanto o resultado da medição era 7. Isto deve ser relacionado com a execução especulativa não acertada do *branch predictor* e que ocorre no inicio do corpo do *statement for*.

Foi devido a esse fenómeno que foi adicionada uma instrução NOP (*no operation*) no inicio da macro *start_time_base()*. Esta instrução acaba por afastar instrução `mttbl 0` mais longe do código precedente, suprimindo assim o efeito colateral na medição.

```
int time;
for (int i=0; i<N_GET_TIME; i++)
{
    start_time_base();
    asm volatile ("ori 0,0,0"); asm volatile ("ori 0,0,0");
    asm volatile ("ori 0,0,0"); asm volatile ("ori 0,0,0");
    asm volatile ("ori 0,0,0"); asm volatile ("ori 0,0,0");
    get_time_base(time);
    xil_printf("%3d    %10d\r\n", i+1, time);
}
```

Listagem 6.2: Exemplo da utilização do registo *time base* na medição do tempo de execução

6.1.2 Contadores *ad hoc* dentro do periférico testado

Os componentes que constituem o *hardware microckernel* são todos implementados com base no projeto genérico apresentado na secção 5.4. Dito projeto especifica a implementação da maquina de estados que, a partir dos endereços das instruções da leitura e escrita emitidas pelo *master* PLB, descodifica os comandos que desencadeiam operações específicas sobre estruturas de dados que compreendem o respetivo periférico. Este mecanismo é facilmente extensível com os novos comandos e respetivas operações. Desta forma, a integração da interface com os contadores *ad hoc* intrinsecamente ligados às estruturas de dados do periférico é de fácil implementação.

A arquitetura proposta para a dita ferramenta de medição consiste num contador que incrementa a cada ciclo do relógio do barramento e um conjunto dos registos de amostragem implementados para guardar o valor atual do contador quando ocorrer certa condição. O número destes registos de amostragem depende do número de acontecimentos independentes cujo instante de ocorrência se pretende registar (durante o mesmo ensaio ou vários) em relação ao instante do *reset* do contador. Ao nível da interface com utilizador são necessários: um comando para o *reset* do contador e, para cada um dos registos, um comando para leitura do valor neste guardado. A integração destes comandos na interface com o periférico é de fácil realização e é exemplificada com os blocos tracejados no diagrama SMChart apresentado na Figura 5.9.

O *template* para implementação da dita arquitetura pode ser visto na Listagem 6.3. Este apresenta implementação do contador e um registo de amostragem, sendo necessária, para implementação de um número de registos de amostragem pretendido, a replicação dos *statements* nos espaços indicados com comentário.

```
signal reset_counter          : std_logic;
signal counter, next_counter  : STD_LOGIC_VECTOR (0 to 31);
signal count1, next_count1    : STD_LOGIC_VECTOR (0 to 31);
signal count1_used, next_count1_used : STD_LOGIC;
--MAIS REGISTOS AQUI
COUNTER_STATE_PROC : process (Bus2IP_Clk) is
begin
  if (Bus2IP_Clk'event and (Bus2IP_Clk = '1')) then
      counter     <= next_counter;
      count1      <= next_count1;
      --MAIS REGISTOS AQUI
  end if;
end process COUNTER_STATE_PROC;
COUNTER_PROC: process (reset_counter, counter, count1_used, count1[, ...]) is
begin
  next_counter  <= counter+1;
  next_count1   <= count1;
  next_count1_used  <= count1_used;
  --MAIS REGISTOS AQUI
  if (reset_counter='1' [ou OUTRA CONDIÇÃO]) then
    next_counter <= (others => '0');
    next_count1_used <= '0';
    --MAIS REGISTOS AQUI
  end if;
  if (ALGUMA CONDIÇÃO and count1_used='0') then
    next_count1 <= counter;
    next_count1_used <= '1';
  end if;
  --MAIS REGISTOS AQUI
end process COUNTER_PROC;
```

Listagem 6.3: *Template* para implementação de contadores *ad hoc* dentro de um periférico

6.1.3 Ferramenta *ChipScope Pro*

ChipScope Pro é um conjunto das ferramentas capazes de configurar, gerir, monitorizar e inserir os analisadores lógicos integrados (ILA) diretamente no *bitstream* do projeto. É uma utilidade muito poderosa que permite efetuar depuração e validação do *hardware* que constitui SoC (*System-on-a-Chip*) desenvolvido durante o funcionamento real do dispositivo FPGA.

A presente secção faz uma breve introdução às potencialidades da ferramenta, bem como apresenta algumas observações práticas da sua utilização retiradas durante a respetiva exploração.

Visão Geral

A Figura 6.1 apresenta a visão geral sobre a organização dos componentes que integram o sistema de depuração.

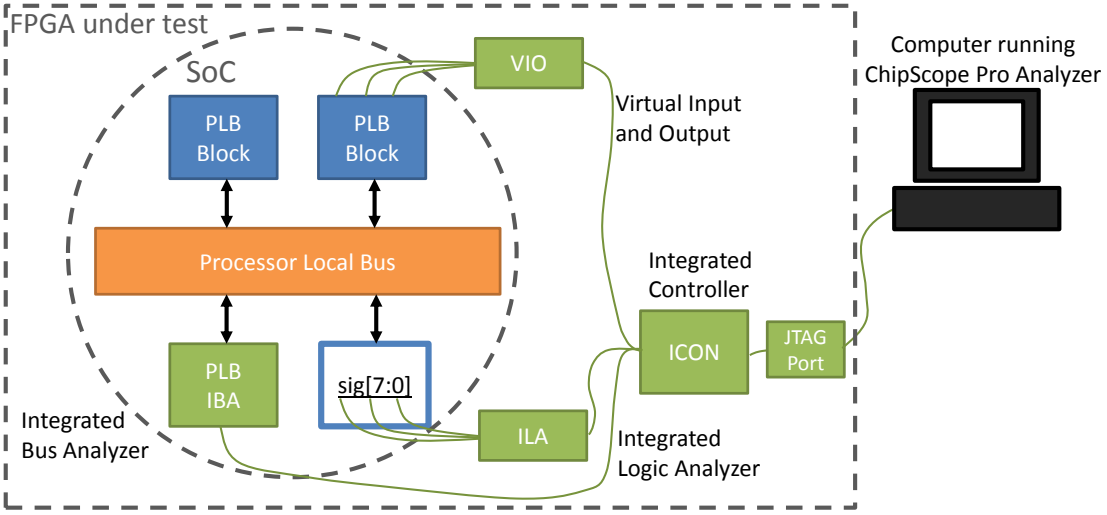


Figura 6.1: Organização do sistema de depuração ChipScope Pro

O sistema de depuração pode ser vista nas três vertentes:

- Os componentes em *hardware* integrados diretamente no SoC depurado:
  - ICON - *Integrated Controller* é o componente que cria a ponte entre a porta JTAG (e consequentemente aplicação em *software ChipScope Pro Analyzer* a executar num computador) e até 15 componentes de depuração: ILA, IBA, VIO, ATC2. O componente pode ser instanciado diretamente no código VHDL ou Verilog ou pode ser integrado num projeto embebido através da utilização das ferramentas embutidas no Xilinx EDK (Xilinx, 2008b).

- ILA - *Integrated Logic Analyzer* é um componente extremamente customizavel que implementa funcionalidades de um analisador lógico, que pode ser usado para monitorização dos sinais internos do projeto. É possível a configuração da largura do porto do *trigger* e porto de dados, bem como a configuração do tamanho da memória para retenção das amostras. Pode ter até 16 unidades de *trigger* independentes, que podem ser combinados numa única condição ou sequência de *trigger*. Configuração das respetivas condições é feita no tempo da execução, a partir da interface do *ChipScope Pro Analyzer*. Inclui também a funcionalidade de qualificador de amostras. Apresenta a saída *trig_out* que pode ser usada para propagação do evento *trigger* para um outro componente ou para um porto externo do dispositivo FPGA (Xilinx, 2008c).
- IBA - *Integrated Bus Analyzer* é um analisador do barramento especializado e desenhado para a depuração dos sistemas que utilizam o barramento PLB. Separa sinais *master*, *slave* e *error* nas unidades de comparação independentes e configuráveis. Proporciona também um porto *trigger* de entrada separado, que possibilita a ligação de sinais do utilizador. Apresenta a saída *trig_out*, da mesma forma como um componente ILA (Xilinx, 2008d).
- VIO - *Virtual Input/Output* é um periférico customizavel que tanto pode monitorizar sinais internos, como atuar sobre estes no tempo da execução. Proporciona botões virtuais e outras utilidades para o controlo das suas saídas de forma síncrona e assíncrona. Proporciona LEDs virtuais e outras utilidades para a monitorização das suas entradas síncronas e assíncronas. Para o caso das saídas síncronas, suporta a funcionalidade de definição de uma trama de 16 *bits* a ser enviada a frequência do projeto (Xilinx, 2008e).

- As ferramentas de configuração e geração integradas no EDK:
  - *XPS –> Debug –> Debug Configuration* - É uma ferramenta embutida no *Xilinx Platform Studio* que permite adicionar e configurar os componentes de depuração no ambito da sua integração num projeto embebido. No ato da adição do componente ILA, IBA ou VIO, o *wizard* ajuda na configuração e ligação dos respetivos portos proporcionando a listagem de todos os sinais que podem ser acedidos. A instanciação e configuração do componente ICON, bem como a ligação dos respetivos portos do controlo aos respetivos componentes de depuração é feita de forma automática.
  - *IP Core Generator* - é uma ferramenta poderosa e integrada no Xilinx ISE que permite acelerar o processo de projeto, fornecendo acesso aos componentes IP (*Intellectual Property*) altamente parametrizáveis. O gerador proporciona um catalogo de componentes divididos em diferentes categorias. Sob

a categoria *"Debug & Verfication"* podem ser encontrados componentes do sistema de depuração *ChipScope Pro*. A ferramenta fornece uma interface simplista de configuração e gere o respetivo ficheiro .ngc no fim do processo da customização.

- *ChipScope Pro Analyzer* - é uma aplicação em *software* que executa num computador e interage com os componentes de depuração através da cadeia JTAG, por meio do controlador integrado (ICON). A aplicação permite configurar o dispositivo, configurar os *triggers*, configurar consola e ver os resultados de amostragem durante a execução real. Os *triggers* e a visualização de amostras podem ser manipulados em diferentes maneiras através de uma interface intuitiva e de fácil uso.

Observações práticas da utilização

O conjunto de ferramentas *ChipScope Pro* foi explorado durante a fase final da validação do correto funcionamento do sistema, consultando o *tutorial* apresentado em (Asifuzzaman, 2010).

No caso da versão 10.1 do EDK, as ferramentas *ChipScope Pro* não se encontram embutidas por defeito, sendo necessária a sua instalação independente.

Para instanciação e configuração dos componentes de depuração foi utilizada a ferramenta *Debug Configuration* integrada no XPS. Foram instanciados um componente ILA e um componente IBA.

Ao configurar o componente IBA, o barramento PLB utilizado pelo *hardware microkernel* foi selecionado, deixando maioria das opções da configuração por defeito. Para possibilitar posterior sincronização dos diferentes analisadores entre si, foi habilitado o sinal de saída *trig_out*. Foi também habilitado o porto *trigger* destinado aos sinais do utilizador com largura de 1 bit. Ao qual depois, através da interface gráfica do XPS, foi ligado o sinal *trig_out* do mesmo componente o que permitiu registar a latência deste. Verificou-se o valor fixo de 10 ciclos do relógio que pode ser também encontrado na documentação.

Ao configurar componente ILA, no separador *"Basic"* os portos *trigger* foram complementados com os sinais ou vetores de sinais pretendidos. É de notar que o componente é instanciado ao mesmo nível que os periféricos do sistema e assim só tem acesso aos portos dos periféricos e não aos seus sinais internos. Desta forma, sem alterações intrusivas nos componentes que constituem *hardware microkernel*, foi possível a monitorização dos seguintes sinais: as interfaces dedicadas entre os componentes, as interrupções externas ligadas ao controlador de interrupções e o sinal de preempção do CPU. É de notar que

as larguras dos portos *trigger* completados no separador *"Basic"* tornam-se automaticamente preenchidas no separador *"Advanced"*. No separador *"Advanced"*, por defeito, todos os sinais dos portos *trigger* fazem parte do porto de dados e vice-versa. Tirar essa opção torna o porto de dados configurável separadamente. O sinal de saída *trig_out* foi habilitado da mesma forma como durante a configuração do componente IBA.

Visto que a condição *trigger* mais comum para os testes seria o endereço específico de uma transação do barramento PLB, a sincronização entre os componentes IBA e ILA foi feita ligando o *trig_out* do IBA a um dos portos *trigger* do ILA.

Após a finalização da configuração e a atualização do *bitstream* do SoC, foi ligado o *ChipScope Pro Analyzer* que através da cadeia do JTAG e controlador integrado (ICON) detetou a presença do componente ILA e componente IBA.

A importação do ficheiro *chipscope_plbv46_iba_0.cdc* que contem os nomes para sinais do porto de dados do IBA é crucial para a navegação na visualização de amostragem e na configuração das condições de *trigger*. O *trigger* do IBA foi configurado para detetar um endereço específico durante uma transação PLB. O seu sinal *trig_out* foi configurado para o nível lógico alto, enquanto o *trigger* do componente ILA foi configurado para detetar o respetivo flanco ascendente. É de notar a importância do *trigger* do ILA ser armado antes do *trigger* do IBA.

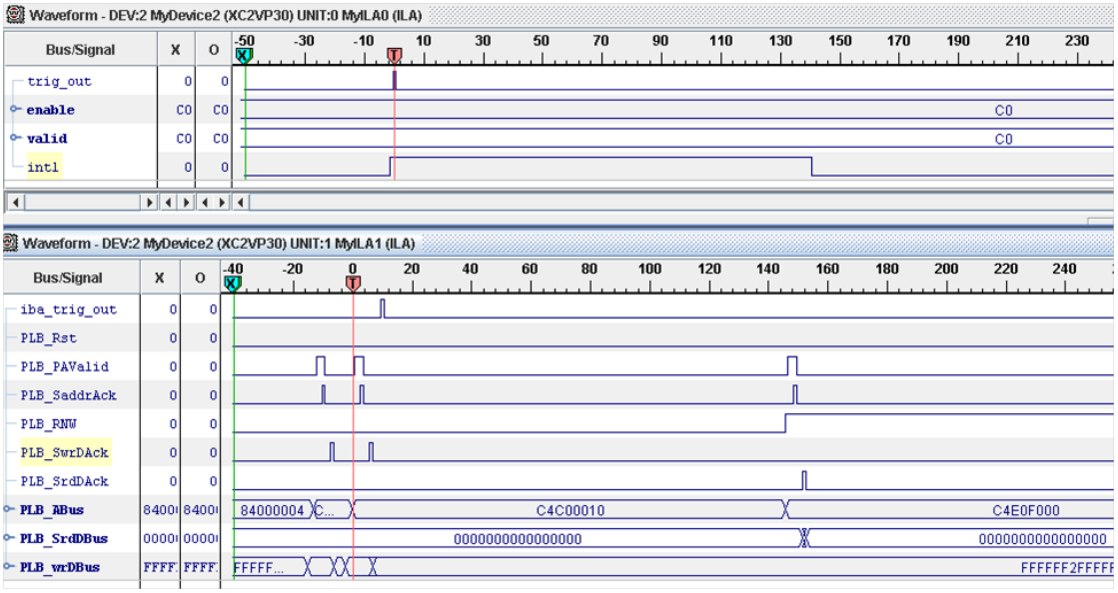


Figura 6.2: Aspeto das janelas de visualização do *ChipScope Pro Analyzer*

A Figura 6.2 apresenta o aspeto visual das janelas com os sinais do IBA (parte inferior) e ILA (parte superior) amostrados. É de notar que as janelas foram configuradas para ter a mesma proporção das escalas e uma diferença de 10 ciclos de relógio no seu *offset*, sendo assim sincronizadas. As riscas verticais vermelhas indicam a altura do disparo do

trigger, sendo do componente ILA 10 ciclos mais atrasado em relação ao do componente IBA.

Foram também efetuadas experiências com a integração do componente ILA dentro da lógica do utilizador do periférico PLB que implementa o escalonador do *hardware micro-kernel* concebido. Desta forma, foi possibilitada a monitorização dos seus sinais internos resultando numa depuração mais profunda. Para o propósito, foi criado um projeto ISE vazio com único intuito de geração do ficheiro .ngc correspondente ao componente ILA customizado através da utilização da ferramenta *IP Core Generator*.

No ficheiro VHDL *user_logic* do periférico PLB o componente ILA foi instanciado com a configuração da interface igual a feita no *Core Generator*. A referida instanciação pode ser consultada na Listagem 6.4, é de notar a maneira pouca intuitiva do manuseio dos portos *trigger* cuja largura é de 1 *bit*. Foram também necessárias alterações na lista dos portos da entidade *user_logic* e da entidade do topo do periférico, para tornar o porto CONTROL acessível a partir do XPS com intuito de o ligar ao controlador integrado (ICON). O ficheiro *ila.ngc* criado pelo ISE foi copiado para a pasta *pcores/<plb_core>/netlist* do repositório local do projeto XPS. Ao fazer reimportação do periférico com utilização da ferramenta CIP (*Create and Import Peripheral*), foi indicado o facto da presença dos ficheiros .ngc e apontado também o caminho para *ila.ngc*.

```
component ila IS
port(
  CONTROL: in std_logic_vector(35 downto 0);
  CLK: in std_logic;
  TRIG_OUT : out std_logic;
  TRIG0: in std_logic_vector(0 downto 0);
  TRIG1: in std_logic_vector(0 downto 0);
  TRIG2: in std_logic_vector(7 downto 0);
  TRIG3: in std_logic_vector(1 downto 0);
  TRIG4: in std_logic_vector(7 downto 0);
  TRIG5: in std_logic_vector(3 downto 0);
  TRIG6: in std_logic_vector(4 downto 0);
  TRIG7: in std_logic_vector(3 downto 0);
  TRIG8: in std_logic_vector(7 downto 0);
  TRIG9: in std_logic_vector(7 downto 0);
  TRIG10: in std_logic_vector(2 downto 0);
  TRIG11: in std_logic_vector(0 downto 0);
  TRIG12: in std_logic_vector(7 downto 0);
  TRIG13: in std_logic_vector(0 downto 0));
END component;
```

```
i_ila: ila
port map(
  CONTROL=>CONTROL,
  CLK=>Bus2IP_Clk,
  TRIG_OUT=> TRIG_OUT,
  TRIG0(0)=> TRIG_IN,
  TRIG1(0)=> round_robin,
  TRIG2=> Timer_is_set,
  TRIG3=> ila_FSL_concat,
  TRIG4=> FSL_S_Data(24 to 31),
  TRIG5=> state_sel,
  TRIG6=> ila_pipe_counter_concat,
  TRIG7=> ila_highest_free_concat,
  TRIG8=> final_enable,
  TRIG9=> priority_out,
  TRIG10=> index_out_MSB,
  TRIG11(0)=> release_event,
  TRIG12=> event_to_release,
  TRIG13(0)=> preemption);
```

Listagem 6.4: Exemplo da instanciação do componente ILA na linguagem VHDL

6.2 Testes

Os ensaios elaborados podem ser divididos em quatro grupos: validação das características funcionais do *kernel* concebido, avaliação dos tempos da execução das tarefas do *hardware microkernel* junto com a respetiva interface, caracterização temporal do processo da preempção baseada na interrupção e a avaliação do impacto da migração dos recursos para o *hardware* dedicado, através da comparação da versão do *kernel* em *software* com a versão híbrida.

Todos os testes foram efetuados em execução real em *hardware*. As medições foram feitas em ciclos de relógio do processador, que é uma grandeza independente da frequência do mesmo. Os testes aqui discutidos foram efetuados no sistema em que PPC405 liga à memória de código e dados com a utilização da interface OCM (*On-Chip Memory*), sendo a memória de código de tamanho 128KB e a memória de dados 64KB. A frequência do relógio do processador e da memória foi mesma e igual a 100MHz.

6.2.1 Validação das funcionalidades do *kernel* e testes de regressão

Os ensaios descritos nesta secção consistem na elaboração dos cenários de utilização simplistas e comparação do comportamento resultante com o comportamento esperado. Estes ensaios foram utilizados para validação do correto funcionamento das funcionalidades do *kernel* aquando sua migração para o *hardware* ou aquando sua implementação (para o caso das funcionalidades ausentes na versão original do *kernel*). Os ditos ensaios também foram utilizados como sendo testes de regressão durante a fase de implementação, com objetivo de verificar se o sistema continua a demonstrar o comportamento esperado após cada alteração significativa do *software* ou *hardware*.

Funcionalidade *round-robin*

Este ensaio consiste num numero máximo de tarefas (depende da configuração do escalonador) com a mesma prioridade, e cujas funções associadas imprimam o identificador da tarefa dentro do *statement while(1)* (ver Listagem 6.5).

De notar que o parâmetro de configuração ROUND_ROBIN deve ser definido no ficheiro *adeos_config.h* com o valor "1", caso contrário só irá executar a primeira tarefa. Caso parâmetro RR_RATE_HZ for configurado com 1, cada tarefa deve ter um *slice* de um segundo para executar. O comportamento esperado para este ensaio pode ser visto na Figura 6.3.

```
int main(){
    for(int i=1; i<HW_SCHED_N_TASK; i++)
        Task::CreateTask(FuncWhile1PrintId, 255, 1024, NULL, -1, 1, 1);
    os.start();
}

void FuncWhile1PrintId(void * arg){
    int id=os.pRunningTask->id;
    while(1){
        xil_printf("%2d\r\n", id);
        for(long int j=0; j<2000000; j++)  nop();
    }
}
```

Listagem 6.5: Implementação do ensaio da funcionalidade *round-robin*

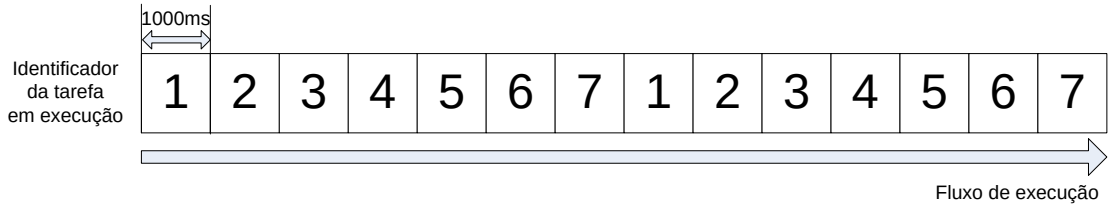


Figura 6.3: Comportamento esperado para o ensaio da funcionalidade *round-robin*

A figura mostra o comportamento esperado para o caso quando o escalonador em *hardware* é configurado (parâmetro HW_SCHED_N_TASK) para um máximo de 8 tarefas. A tarefa *idle* sempre possui identificador 0 e não aparece na figura, pois neste cenário nunca irá executar.

Funcionalidade de temporização

Este ensaio, mais uma vez, consiste num numero máximo de tarefas (depende da configuração do escalonador) com a mesma prioridade. As funções associadas, mais uma vez executam dentro do *statement while(1)*, mas desta vez suspendem a execução da tarefa por um intervalo de 1 segundo depois de imprimir o seu identificador(ver Listagem 6.6).

O comportamento esperado para este ensaio pode ser visto na Figura 6.4.

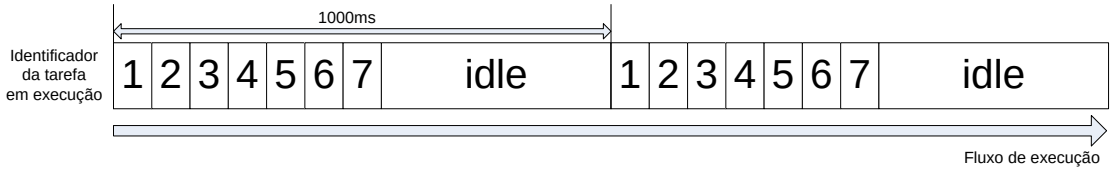


Figura 6.4: Comportamento esperado para o ensaio da funcionalidade de temporização

```
int main(){
    for(int i=1; i<HW_SCHED_N_TASK; i++)
        Task::CreateTask(FuncAutoDelay, 255, 1024, (void *) 1000, -1, 1, 1);
    os.start();
}

void FuncAutoDelay(void * arg){
    int id=os.pRunningTask->id;
    while(1){
        xil_printf("%2d\r\n", id);
#if (HW_SCHED==1)
        hwkernel.delay_running_task((int)arg);
#else
        Timer timer;
        timer.start((int)arg, OneShot);
        timer.waitfor();
#endif
    }
}
```

Listagem 6.6: Implementação do ensaio da funcionalidade de temporização

Funcionalidade exclusão mútua

Este ensaio visa verificar o correto funcionamento do único recurso de sincronização que o *kernel* implementa e que é exclusão mútua (*mutex*). O ensaio consiste numa tarefa de baixa prioridade (denominada como tarefa A) que é criada antes do arranque do sistema operativo. Esta tarefa, dentro da sua função associada, adquira o *mutex*, depois de que cria a tarefa B com a prioridade alta. No fim da criação da tarefa B ocorre a preempção da tarefa A, visto que esta deixa de ser a tarefa mais prioritária. Dentro da função associada a tarefa B ocorre uma tentativa de adquirição do *mutex* que acaba no insucesso. A tarefa B é bloqueada e é preemptida a favor da tarefa A, que volta a ser a tarefa mais prioritária das prontas para execução. A tarefa A retoma a sua execução e faz a libertação do *mutex* que resulta no desbloqueio da tarefa B e consequente retoma da sua execução. Por fim, a tarefa B liberta por completo o *mutex* e acaba a sua existência, pois a sua função associada chega ao fim da execução e devolve, por sua vez, a tarefa A retoma a execução, deixando também de existir logo a seguir.

A implementação do ensaio pode ser vista na Listagem 6.7, enquanto o comportamento esperado é representado na Figura 6.5.

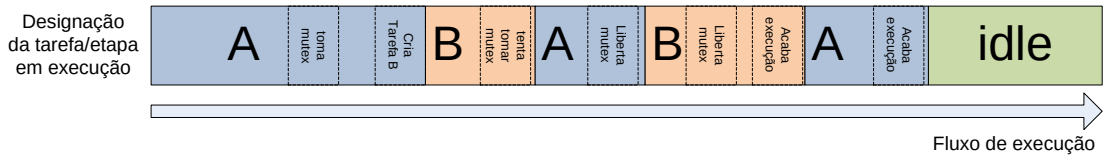


Figura 6.5: Comportamento esperado para o ensaio da funcionalidade exclusão mútua

```
int main(){
    Mutex mineMutex;
    Task::CreateTask(FuncA, 240, 1024, (void *)&mineMutex, -1, 1, 1);
    os.start();
}
void FuncA(void * arg){
    xil_printf("A: Enter\r\n");
    xil_printf("A: Take\r\n");
    ((Mutex*)arg)->take();
    xil_printf("A: Create B\r\n");
    Task::CreateTask(FuncB, 255, 1024, arg, -1, 1, 1);
    xil_printf("A: Release\r\n");
    ((Mutex*)arg)->release();
    xil_printf("A: Return\r\n");
}
void FuncB(void * arg){
    xil_printf("B: Enter\r\n");
    xil_printf("B: Take\r\n");
    ((Mutex*)arg)->take();
    xil_printf("B: Release\r\n");
    ((Mutex*)arg)->release();
    xil_printf("B: Return\r\n");
}
```

Listagem 6.7: Implementação do ensaio da funcionalidade exclusão mútua

6.2.2 Tempos de Execução

Uma parte dos testes efetuados consistia na medição de tempo de execução das certas tarefas do *kernel* híbrido concebido com a utilização da ferramenta de medição baseada no registo *time base* do processador PPC405 e que foi apresentada na secção 6.1.1. A Listagem 6.8 apresenta a implementação do respetivo teste.

Como pode ser visto, foi criada uma tarefa de alta prioridade (250), cuja função associada é *FuncTimeBase()*, tamanho de pilha 1024 *bytes*, que passa o argumento nulo para a função associada, que não é associada à nenhuma fonte de interrupção externa, que não é apagada aquando devolução da sua função associada e que é desabilitada no ato da criação. Uma explicação detalhada da utilização do método *CreateTask()* pode ser encontrada na secção 4.3.2.

Ao mesmo tempo, é criada uma tarefa de prioridade média que irá acordar a tarefa anterior TEST_NUM vezes. A função associada a esta recebe como argumento o identificador da tarefa anterior e é configurada para ser habilitada no ato da criação.

Ao recorrer a este mecanismo foram efetuados testes com diferentes tarefas do *kernel* a serem avaliadas e os resultados destes testes são compilados na Tabela 6.1.

```
#define TEST_NUM 20
int main(){
    //Tarefa de alta prioridade, que não é apagada ao retornar
    //e que é desabilitada no ato da criação
    int idTaskTB=Task::CreateTask(FuncTimeBase, 255, 1024, NULL, -1, 0, 0)->id;
    //Tarefa de prioridade média que irá acordar a tarefa de alta prioridade
    //e que recebe o respetivo identificador como argumento
    Task::CreateTask(FuncAwakeTask, 200, 1024,(void *)idTaskTB, -1, 1, 1);
    os.start();
}
void FuncTimeBase(void * arg){
    int time;
    start_time_base();
    /*...*/ //Aqui rotina para ser avaliada
    get_time_base(time);
    xil_printf("%5d\r\n", time);
}
void FuncAwakeTask(void * arg){
    for(int i=0; i<TEST_NUM; i++){
        xil_printf("%3d:", i+1);
        hwkernel.awake_task((int)arg);
    }
}
```

Listagem 6.8: Implementação de medição de tempos de execução das tarefas do *kernel*

Tabela 6.1: Tempos de execução em número de ciclos de relógio de certas tarefas do *kernel* híbrido

Tarefa	NCR
nop()	1
request_task(priority,start_on_create)	3
set_task_pointer(id,Task_p)	5
disable_task(id)	4
remove_task(id,irq_id)	7
delay_task(id,time)	5
awake_task(id)	4
enterCS()	5
exitCS()	4
get_time_base(time); global_time[id][i++]=time;	7

6.2.3 Caracterização temporal da preempção baseada na interrupção

Foi também feito um ensaio com intuito de caracterização da quantia do tempo despendida durante o ato da preempção de uma tarefa no caso do *kernel* híbrido. O mecanismo utilizado na preempção foi apresentado na secção 4.2.5 e pode ser resumido que a comutação de contexto se baseia no prólogo e epílogo do atendimento à interrupção (que é da responsabilidade do compilador) e na manipulação do *stack pointer* entre o acon-

tecimento destes.

Este ensaio não se baseia num cenário de utilização prático, mas sim numa utilização inapropriada da funcionalidade do *kernel* que faz o *delay* da tarefa em execução. A implementação do ensaio pode ser encontrada na Listagem 6.9 e o seu intuito pode ser resumido nos seguintes pontos:

- A medição do tempo é feita recorrendo ao mesmo tempo ao registo TBL do PPC405 e ao contador *ad hoc* em *hardware* intrinsecamente ligado ao componente que implementa escalonador no *hardware microkernel*. Estas ferramentas de medição são apresentadas em detalhe nas secções 6.1.1 e 6.1.2, respetivamente.

- No início é feito *reset* dos contadores das duas ferramentas de medição: `start_time_base()` e `soht_reset_count()`. O consequente *offset* entre os resultados destas duas ferramentas deve ser considerado durante a respetiva análise.

- A tarefa suspende a si própria escrevendo para o respetivo temporizador (`write_delay(os.pRunningTask->id,1)`), no entanto o tempo requerido é de apenas 1 ciclo do relógio do CPU. A interrupção vai ocorrer na mesma, mas ainda antes de esta ser atendida, a tarefa volta a ser mais prioritária das prontas para execução e por isso continua a sua execução após interrupção.

- O contador *ad hoc*, que se encontra dentro do escalonador, copia o seu valor atual para registos `count1`, `count2` e `count3` quando o sinal `Timer_is_set` da tarefa muda de 0 para 1, quando a linha de interrupção do periférico muda de 0 para 1 e quando o sinal `Timer_is_set` volta mudar para 0, respetivamente.

- Foram também colocadas macros `get_time_base()` (que registam o estado atual do registo TBL) logo antes e logo depois da leitura do escalonador (e consequente *acknowledge* da interrupção) dentro da rotina *schedule()* invocada a partir da interrupção (não aparece na listagem). Os respetivos resultados são guardados nas variáveis globais `global_time1` e `global_time2`.

- Para além de efetuar as medições com as ferramentas referidas, foi também feita a validação mútua dos resultados destas ferramentas com as da ferramenta *ChipScope Pro* apresentada em detalhe na secção 6.1.3. Ao controlador integrado (ICON) foi ligado um componente ILA, configurado para amostrar a linha de interrupção do escalonador e o sinal `Timer_is_set(1)` da interface dedicada entre componente de temporização, e um componente IBA, ligado ao mesmo barramento PLB que o *hardware microkernel*. O *trigger* do IBA foi configurado para detetar o inicio da transação causada pelo `write_delay()`, enquanto o *trigger* do componente ILA foi ligado ao sinal propogado pelo *trigger* do IBA.

```
int main(){
    Task::CreateTask(FuncDelayOneCycle, 255, 1024, NULL, -1, 1, 1);
    os.start();
}
void FuncDelayOneCycle(void * arg){
    declare_20_times();
    int hw_time1, hw_time2, hw_time3, hw_time4;

    start_time_base();
    soht_reset_count();
    write_delay(os.pRunningTask->id,1);
    get_time_base(time1);
    get_time_base(time2);
    /*...*/
    get_time_base(time20);

    hw_time1=(unsigned int)soht_get_count1();
    hw_time2=(unsigned int)soht_get_count2();
    hw_time3=(unsigned int)soht_get_count3();

    xil_printf("%10d %10d %10d %10d\r\n", time1, time2, time3, time4);
    /*...*/
    xil_printf("%10d %10d %10d %10d\r\n", time17, time18, time19, time20);

    xil_printf("%10d %10d %10d\r\n", hw_time1, hw_time2, hw_time3);
    xil_printf("%10d %10d\r\n", global_time1, global_time2);
}
```

Listagem 6.9: Implementação do ensaio da caracterização temporal da preempção de uma tarefa

Após a análise dos resultados das três ferramentas foi possível produzir diagrama que se encontra na Figura 6.6

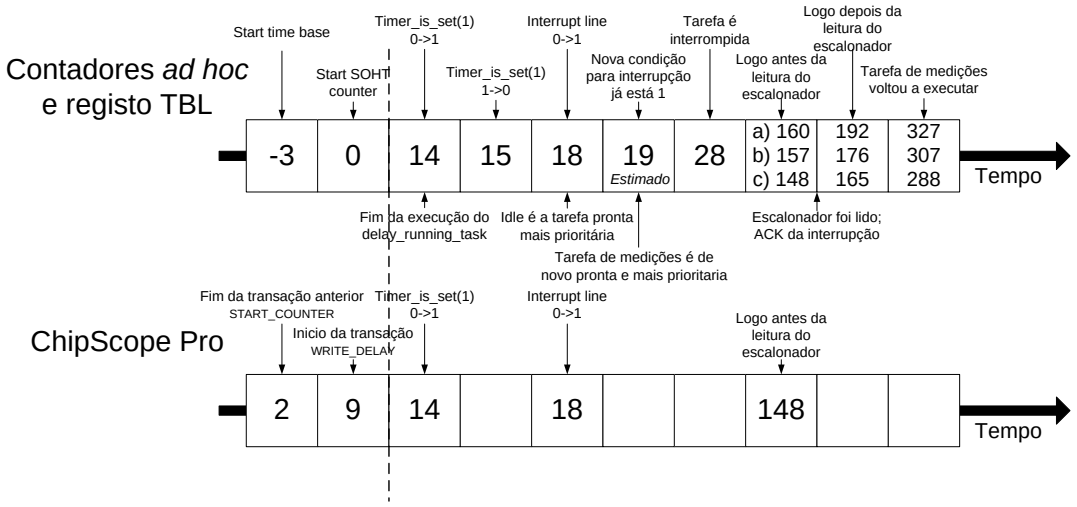


Figura 6.6: Caracterização temporal da preempção de uma tarefa

A análise dos resultados compilados no diagrama pode ser resumida nos seguintes pontos:

- O *offset* de 3 ciclos, existente entre início da contagem do contador *ad hoc* no escalonador e *reset* do registo *time base*, foi retirado de todas as medições feitas com recurso ao registo *time base*.
- O tempo entre fim da execução do `write_delay()` e o momento em que a linha de interrupção mudar de 0 para 1 é de 4 ciclos. Isto deve-se ao facto de testes serem feitos com o escalonador em *hardware* configurado para 16 tarefas, o que significa que a sua árvore binária apresenta 4 níveis e uma alteração a sua entrada é propagada até a saída em 4 ciclos do relógio. Também foram efetuados testes com o escalonador preparado para 8 tarefas e o referido intervalo foi de 3 ciclos do relógio, respetivamente.
- Inicialmente pensou-se que o intervalo de 10 ciclos entre o flanco ascendente do sinal da preempção (instante 18) e o momento da interrupção da execução da tarefa (instante 28) é da responsabilidade do controlador de interrupções, que faz parte dos serviços do controlador IPIF utilizado durante o projeto do periférico PLB e que foi selecionado durante a conceção do *template* para o escalonador do *hardware microkernel*. Esta suspeita foi desvendada, visto que, mesmo com a ligação do sinal da preempção, que muda para o nível lógico alto no instante 18, diretamente ao porto da interrupção externa do processador, a tarefa só é interrompida no instante 28.
- Para o caso das medições feitas com contadores *ad hoc* e registo TLB, são apresentados resultados relativos às três versões da implementação. Inicialmente, o escalonador em *hardware* tinha capacidade de fornecer apenas o identificador da tarefa que se encontra a sua saída, enquanto em *software* estava a ser gerida uma tabela que fazia a tradução do dito identificador da tarefa para o apontador do respetivo objeto (versão "a"). Foi depois decidido de efetuar também a migração para o *hardware* da dita tabela. Atualmente, o escalonador guarda o apontador para objeto da tarefa aquando a sua criação e o fornece durante o processo da preempção. Esta melhoria resultou na redução do respetivo *overhead* (versão "b"). Por sua vez, a versão "c" é uma melhoria da versão "b" conseguida através do *inline* do método *schedule()* invocado a partir do serviço do atendimento à interrupção causada pelo sinal da preempção do escalonador em *hardware*.
- No que diz respeito aos resultados obtidos com a utilização da ferramenta *ChipScope Pro*, foi necessário recorrer a sincronização do início da sua escala temporal com a das medições anteriores, sendo esta feita pelo instante em que o sinal `Timer_is_set` muda o seu estado de 0 para 1. Após esta sincronização, revelou-se coincidência no resultado da medição do instante da leitura do escalonador

(instante 148), que permitiu validar mutuamente as ferramentas de medição e os resultados obtidos.

6.2.4 Versão *software* versus Versão híbrida

O objetivo conclusivo da presente dissertação apontado no Capítulo 1 é a avaliação do impacto da migração dos recursos do *kernel* resultante para o *hardware* dedicado. Para poder efetuar a dita avaliação, o *kernel* resultante proporciona um parâmetro que permite configurar se a assistência por *hardware* proporcionada pelo *hardware microkernel* é usada ou não. Como já foi referido na secção 4.3.2, esta variabilidade foi gerida recorrendo a técnica da compilação condicional baseada no uso das diretivas de pré-processamento: *#if*, *#ifdef*, *#else*, *#elif*, *#endif*, etc.

Esta secção apresenta a comparação da versão *software* com a versão híbrida em termos de latências e *jitter* associados. O cenário utilizado para esta comparação é resumido nos seguintes tópicos:

- Ao mesmo tempo executam 10 tarefas periódicas com a mesma prioridade (200) e com os períodos múltiplos de 50ms. Sendo a primeira tarefa com o período 50ms, segunda tarefa com o período 100ms e por aí em diante, acabando com a décima tarefa com o período 500ms.
- Uma tarefa de prioridade máxima é suspensa durante 5 segundos, permitindo desta forma a execução das referidas tarefas periódicas de prioridade média. Durante este tempo, a primeira tarefa (maior frequência) executa 100 vezes, enquanto a décima - 10 vezes.
- Sempre que uma das tarefas periódicas começa a sua execução depois de ser suspensa, o número de ciclos do relógio que passaram desde o arranque do sistema é registado num vetor associado a tarefa em causa (sendo *overhead* da execução das respetivas instruções é de apenas 7 ciclos, ver Tabela 6.1).
- Ao fim de 5 segundos, a tarefa de prioridade máxima volta a executar e para cada tarefa imprime os instante do tempo nos quais esta entrou em execução.
- Para cada evento de reentrada da tarefa é depois calculado o atraso em número de ciclo de relógios entre o tempo da entrada em execução e o tempo esperado em relação à última execução. Um atraso negativo indica que a execução ocorreu mais cedo do que era previsto.

Este cenário foi executado sob controlo da versão híbrida do *kernel* (parâmetro HW_SCHED definido com "1") e sob controlo da versão em *software*. Os respetivos resultados (nú-

meros de ciclos de relógio do processador), depois de serem acondicionados numa folha do cálculo *excel*, são resumidos na Tabela 6.2.

Tabela 6.2: Resultados da comparação da versão híbrida com a versão em *software*

Versão	Atraso Máximo	Atraso Mínimo	*Jitter*	Desvio Padrão
Híbrida	1268	302	966	179,85
Software	3640	-5855	9495	1474,18

Um atraso negativo indica que a tarefa entrou em execução mais cedo do que foi pretendido.

Para este cenário de utilização pode ser concluído que, com a migração dos recursos do *kernel* para o *hardware*, a redução do *jitter* (diferença entre atraso máximo e atraso mínimo) foi na ordem de 10 vezes.

Capítulo 7

Discussão e Conclusões

O trabalho desenvolvido no âmbito desta dissertação foi resultado da junção de duas áreas que se apresentam como uma fonte incessante de aprendizagem: sistemas operativos de tempo real e sistemas embebidos dotados com dispositivo FPGA. O casamento destas duas vertentes num trabalho só requereu domínio de tecnologias envolvidas e originou desafios motivadores.

A variedade e profundidade de conhecimentos adquiridos, bem como resultados obtidos, conduzem à conclusão que os objetivos, quer pedagógicos, quer do projeto, foram alcançados.

Neste último capítulo da dissertação, é feito um resumo do trabalho desenvolvido e apresentadas as conclusões retiradas pelo autor, com base no que foi implementado. São também feitas considerações relativas ao trabalho futuro.

7.1 Trabalho Desenvolvido

De acordo com os objetivos estipulados aquando inicio do trabalho, foi implementado um *kernel* de tempo real assistido por *hardware*. O *kernel* concebido implementa um conjunto mínimo dos recursos que compreendem um sistema operativo embebido de tempo real. A maioria destes recursos foi também implementada na forma de um conjunto de periféricos dedicados e o impacto da migração dos recursos do *kernel* para o *hardware* dedicado foi avaliado.

O processo da implementação foi conduzido nas duas vertentes em paralelo: vertente *software*, que envolveu a análise detalhada e o *porting* de um sistema existente, a implementação dos *upgrades* neste, bem como a implementação da interface entre este e coprocessadores em *hardware*; e vertente *hardware*, que envolveu a familiarização com os

conceitos e as ferramentas envolvidos na conceção de um sistema embebido baseado na tecnologia SoPC (*System-on-a-programmable-Chip*), bem como o processo da conceção de um *hardware microkernel* implementado na forma de um conjunto de periféricos mapeados na memória e integrados junto com o CPU no qual executa a referida vertente *software*.

Partindo de um *kernel* simplista, foi feita a sua adaptação e expansão, chegando à integração transparente do *hardware microkernel* concebido. O último pode ser integrado num sistema operativo existente ou pode servir de base para conceção de um sistema operativo híbrido.

O *hardware microkernel* concebido tem a capacidade de escalonar, seguindo a estratégia *fixed-priority preemptive*, 8, 16 ou 32 tarefas. A situação da empate de prioridades é endereçada através da implementação da funcionalidade *round-robin* com *timeslice*. A cada tarefa é associada uma entidade de temporização dedicada e a funcionalidade de exclusão mútua é implementada de forma intrinsecamente ligada ao gestor da informação de tarefas. Com a migração de todos estes recursos para o *hardware* dedicado, foi eliminada a necessidade do *system-tick* e das execuções especulativas dos pontos de escalonamento. O CPU é interrompido quando e só quando o escalonador em *hardware* detetar a necessidade efetiva da preempção, sendo o prólogo e o epílogo do respetivo atendimento à interrupção aproveitados na implementação da mudança do contexto, resumindo o *overhead* computacional a uma simples operação de leitura que devolve o apontador para a tarefa que deve entrar em execução.

Para além disso foi também implementada a unificação de espaço de prioridades das tarefas e interrupções, através da conceção de um controlador de interrupções dedicado, que acondiciona os eventos da interrupção externos dependendo do modo de configuração e, em vez de interromper processador com pedido de interrupção, faz um pedido ao escalonador do *hardware microkernel* para habilitar a tarefa previamente assinada como sendo o *handler* para a fonte de interrupção em causa.

O correto funcionamento dos periféricos foi verificado e o processo de aperfeiçoamento está terminado. O impacto da migração das funcionalidades do *kernel* para o *hardware* dedicado foi avaliado através da comparação da versão puramente *software* com a versão híbrida.

Foi então concluído que a estrutura *binary tree* desenhada, utilizada como o extrator da tarefa mais prioritária no escalonador em *hardware*, apresenta uma ocupação da área do silício insustentável, visto que o síntese do escalonador parametrizado para 64 tarefas não foi conseguido pela ferramenta XST *Xilinx Synthesis Technology*, ou seja a dita arquitetura é pouco escalável. A decisão da implementação da funcionalidade exclusão mútua de forma intrinsecamente ligada à árvore binária foi errada e dificultou o desenvolvimento e

manutenção do escalonador. A migração das funcionalidades do *kernel* para o *hardware* dedicado teve um efeito positivo no sentido da consolidação do determinismo: *jitter* reduzido e latência baixas na execução dos cenários comuns da utilização de um sistema operativo de tempo real *low-end* com a estratégia do escalonamento *fixed-priority preemptive*. A execução de tarefas do *kernel* apresenta um *overhead* computacional baixo e constante. Para um cenário particular da utilização, *kernel* híbrido apresentou uma redução do *jitter* na ordem de dez vezes em relação a versão puramente *software* do mesmo.

7.2 Trabalho Futuro

Durante a elaboração do presente trabalho, foi adquirido um conjunto de *hard skills* cujo domínio profundo torna agora possível a consideração da elaboração de um projeto mais ambicioso e com a relevância indiscutível, como por exemplo a implementação em *hardware* de uma versão simplificada do escalonador CFS *Completely Fair Scheduler*, utilizado atualmente no *kernel* do sistema operativo Linux, cuja adopção em sistemas embebidos tornou-se numa tendência crescente e cada vez mais generalizada.

Com intuito de tornar o *hardware microkernel* elaborado independente do barramento presente no SoC (*System-on-a-Chip*) particular, no qual este pode ser integrado, pode ser considerada a implementação do descodificador do barramento com recurso a técnica do microcódigo.

No que diz respeito às melhorias que podem ser implementadas no *kernel* híbrido concebido, é óbvia a necessidade de adoção de uma arquitetura diferente, como por exemplo *multi-FIFO*, na conceção em *hardware* do gestor de tarefas e respetivo extrator da tarefa mais prioritária. A implementação de recursos de sincronização deve ser desacoplada do gestor de tarefas e feita de uma forma independente, num componente dedicado.

Bibliografia

J. Agron, "Hardware microkernels-a novel method for constructing operating systems for heterogeneous multi-core platforms," Ph.D. dissertation, University of Arkansas, 2010.

Altera, "Analyzing Designs with Quartus II Netlist Viewers," Altera, Tech. Rep. May, 2008.

D. Andrews, W. Peck, and J. Agron, "hthreads: a hardware/software co-designed multithreaded RTOS kernel," *Emerging Technologies and Factory Automation, 2005. ETFA 2005. 10th IEEE Conference on*, vol. 2, pp. 331–338, 2005.

P. J. Ashenden, "Recursive and Repetitive Hardware Models in VHDL," Department of Computer Science, The University of Adelaide, Tech. Rep., 1994.

K. Asifuzzaman, "Using ChipScope with Xilinx Platform Studio (XPS)," 2010. [Online]. Available: http://fileadmin.cs.lth.se/cs/Education/EDA385/HT10/documents/KaziChipScopeTutorial.pdf

D. Bacon, R. Rabbah, and S. Shukla, "FPGA Programming for the Masses," *Communications of the ACM*, pp. 1–13, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2436271

S. Bailey, "Comparison of VHDL, Verilog and SystemVerilog." [Online]. Available: http://boydtechinc.com/btf/archive/att-1977/01-LanguageWhitePaper.pdf

M. Barr, *Programming Embedded Systems in C and C++*. O Reilly, 1999.

——, "Embedded Systems Glossary," 2007. [Online]. Available: http://www.barrgroup.com/embedded-systems/glossary

M. Barr and A. Massa, *Programming Embedded Systems: With C and GNU Development Tools, 2nd Edition*. O'Reilly, 2006.

R. Barry, "FreeRTOS, Coding Standard and Style." [Online]. Available: http://www.freertos.org/FreeRTOS-Coding-Standard-and-Style-Guide.html

——, *Using The FreeRTOS Real Time Kernel*. Richard Barry, 2009.

Berkeley Design Technology Inc, "The Evolving Role of FPGAs in DSP Applications," Berkeley Design Technology, Inc, Tech. Rep., 2007.

B. B. Brey, *Embedded Controllers: 80186, 80188, and 80386EX*. Prentice Hall, 1997. [Online]. Available: http://www.amazon.com/Embedded-Controllers-80186-80188-80386EX/dp/0134001362

ESRG, "Embedded System Research Group." [Online]. Available: http://algoritmi.uminho.pt/research-teams/esrg/

Florida Internation University, "PowerPC based embedded design in FPGA using Xilinx EDK," Florida Internation University, Tech. Rep. [Online]. Available: http://web.eng.fiu.edu/~gaquan/teaching/ccli/EDK.pdf

FreeRTOS, "Xilinx PowerPC (PPC405) Port on a Virtex-4 FPGA." [Online]. Available: http://www.freertos.org/Free-RTOS-PPC405-Xilinx-Virtex4.html

——, "FreeRTOS." [Online]. Available: www.freertos.org

D. Gallegos, B. Welch, and J. Jarosz, "Soft-core processor study for node-based architectures," Sandia National Laboratories, Tech. Rep. September, 2008. [Online]. Available: http://prod.sandia.gov/techlib/access-control.cgi/2008/086015.pdf

P. Glover, "Using and Creating Interrupt-Based Systems," Xilinx, Tech. Rep., 2005. [Online]. Available: http://www.xilinx.com/support/documentation/application_notes/xapp778.pdf

T. Hall and J. Hamblen, "Using System-on-a-Programmable-Chip Technology to Design Embedded Systems," *International Journal of Computer Applications*, vol. 13, no. 3, pp. 1–11, 2006. [Online]. Available: http://knowledge.e.southern.edu/cgi/viewcontent.cgi?article=1004&context=facworks_comp

W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, "Sloth: Threads as Interrupts," *2009 30th IEEE Real-Time Systems Symposium*, pp. 204–213, Dec. 2009. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5368879

W. Hofer, D. Lohmann, and W. Schröder-Preikschat, "Sleepy Sloth: Threads as Interrupts as Threads," *2011 IEEE 32nd Real-Time Systems Symposium*, pp. 67–77, Nov. 2011. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6121427

Infieon Technologies, "TriCore 1.3 32-bit Unified Processor Core," Infineon Technologies, Tech. Rep., 2002.

A. M. Jeffrey Liu, Insop Song, "Efficient Priority-Queue Data Structure for Hardware

Implementation," *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pp. 476 – 479, 2007.

P. Kohout, B. Ganesh, and B. Jacob, "Hardware support for real-time operating systems," *First IEEE/ACM/IFIP International Conference on Hardware/ Software Codesign and Systems Synthesis (IEEE Cat. No.03TH8721)*, pp. 45–51, 2003. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1275254

L. Leyva-del Foyo, "Predictable Interrupt Management for Real Time Kernels over conventional PC Hardware," in *Real-Time and Embedded Technology and Applications Symposium (RTASâ€™06)*, 2006. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1613319

C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, no. 1, pp. 46–61, 1973. [Online]. Available: http://dl.acm.org/citation.cfm?id=321743

MSDN, "Strong Typing," 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa378693(v=vs.85).aspx

OSEK Group, "OSEK/VDX Operating System Specification 2.2.3," OSEK Group, Tech. Rep., 2005. [Online]. Available: http://portal.osek-vdx.org/files/pdf/specs/os223.pdf

C. H. Roth Jr., *Digital Systems Design Using VHDL*, 1st ed. CL Enginering, 1998.

F. Scheler, W. Hofer, B. Oechslein, R. Pfister, W. Schröder-Preikschat, and D. Lohmann, "Parallel, hardware-supported interrupt handling in an event-triggered real-time operating system," *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems - CASES '09*, p. 167, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1629395.1629419

A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 9th ed. Wiley, 2012.

D. J. Smith, "VHDL & Verilog Compared & Contrasted Plus Modeled Example Written in VHDL, Verilog and CNo Title." [Online]. Available: http://www.bawankule.com/verilogcenter/verilogvhdl.html

B. Stroustrup, *The C++ Programming Language*, 3rd ed. Addison-Wesley Longman Publishing Co., 1997.

Xilinx, "PPC405 User Manual Virtex-II Pro â„¢ Platform FPGA," Xilinx, Tech. Rep. March, 2002. [Online]. Available: http://www.cs.york.ac.uk/rts/docs/Xilinx-datasource-2003-q1/userguides/V2pro_handbook/userguide/ug010v2a.pdf

——, “MicroBlaze Processor Reference Guide,” Xilinx, Tech. Rep., 2008. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

——, “Platform Specification Format Reference Manual,” Xilinx, Tech. Rep., 2008. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/edk10_psf_rm.pdf

——, “ChipScope Pro ICON (v. 1.00a, 1.01a, 1.02a),” Xilinx, Tech. Rep., 2008.

——, “ChipScope Pro ILA (v. 1.00a, 1.01a, 1.02a),” Xilinx, Tech. Rep., 2008.

——, “ChipScope PLBv46 Integrated Bus Analyzer (v1.00a,1.01a),” Xilinx, Tech. Rep., 2008.

——, “ChipScope Pro VIO (v. 1.00a, 1.01a, 1.02a),” Xilinx, Tech. Rep., 2008.

——, “Virtex-II Pro Development,” Xilinx, Tech. Rep., 2009.

——, “Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet,” Xilinx, Tech. Rep., 2011. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf

ZedBoard, “ZedBoard.” [Online]. Available: http://www.zedboard.org/