Portando o EPOS para a Zedboard

Bruno Farias de Loreto[1]

[1]Departamento de Informática e Estatística (INE)
Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brazil

kira@lisha.ufsc.br

Abstract. *With the increasing demand for processing power from nowadays embedded systems, it became necessary for an embedded operating system to support multicore platforms. A embedded operating system having support for such platform enables new research lines, as well as new deployment scenarios, such as the integration of a multicore real time scheduler to manage critical tasks. The EPOS operating system does not have support for a multicore embedded platform, and such support is becoming increasingly necessary. This work aims on describing how the operating system EPOS was ported to the multicore embedded platform Zedboard.*

Resumo. *Com o aumento da demanda de poder de processamento dos sistemas embarcados atuais, tornou-se necessário que um sistema operacional embarcado tenha suporte para arquiteturas multicores embarcadas. Um sistema operacionanal embarcado tendo suporte para tal plataforma, possibilita novas linhas de pesquisa, e novos cenários de aplicação do SO, como por exemplo a integração de um escalonador* multicore *de tempo real para gerenciar tarefas críticas. O sistema operacional EPOS não possui suporte para um arquitetura embarcada* multicore*, e tem sido cada vez mais necessário a existência de tal suporte. Este trabalho visa descrever como o sistema operacional EPOS foi portado para a plataforma embarcada* multicore *Zedboard.*

1. Introdução

Sistemas embarcados estão instalados e controlam os mais diversos equipamentos hoje em dia, eles estão mais presentes do que nos damos conta. Conforme eles estão sendo cada vez mais empregados, sua demanda aumenta também, bem como seu poder de processamento. Máquinas que antes utilizavam vários pequenos microcontroladores trabalhando independentemente, hoje podem ser gerenciados com melhor custo-benefício por um sistema de maior desempenho [Gracioli 2014].

Sistemas embarcados estão empregados em tarefas que exigem a manipulação de grande volume de dados (muitas vezes em tempo real), como na área de comunicação e transmissão de dados. O nome "sistema embarcado" já foi sinônimo de pequenos microcontroladores dedicados, mas hoje também há grande demanda por sistemas embarcados de alto desempenho. Um meio de obter processamento de alto desempenho à baixo custo é com a implantação de processadores *multicores*.

Aplicações de tempo real necessitam um tempo de resposta previsível, e suas tarefas devem ser executadas com atraso controlável. Um exemplo são os sistemas de airbag

automotivo, que necessitam de uma resposta a um estímulo em um determinado intervalo de tempo [nat 2014]. Muitas destas aplicações são feitas diretamente em hardware para se adequar aos requisitos da aplicação (tempo real, gasto energético, etc). Entretanto aplicações feitas diretamente em hardware apresentam problemas em relação à manutenibilidade e possibilidade de atualização [Gracioli 2014]. Estes componentes de hardware poderiam ser centralizados em um sistema embarcado de maior processamento, e alocar tarefas de tempo real para cada uma das funções necessárias ao sistema. Deste modo, a manutenção e custo total do sistema pode ser melhorado, sem necessitar sacrificar as restrições temporais (usando um sistema operacional de tempo real que consiga escalonar o conjunto de tarefas).

É natural que estas plataformas de maior desempenho, que consequentemente possuem mais recursos e componentes à serem gerenciados, necessitem de um sistema operacional que se encarregue desta função, e que possam administrar eficientemente estes recursos. De acordo com um estudo recente [UBM 2013], a principal razão para usar um sistema operacional é ter garantias de tempo real.

De acordo com este mesmo estudo [UBM 2013], metade dos projetos de sistemas embarcados usaram mais que um microcontrolador/processador em seu projeto.

Dentre estas placas, podemos destacar a Zedboard, que é uma plataforma de bom desempenho[1] com um processador *dualcore* de 666 MHz e FPGA.

EPOS, que é um sistema operacional embarcado de tempo real (RTOS[2]) *multithread*, é o o primeiro RTOS de código aberto a suportar os escalonadores de tempo real global, particionado e agrupado [Gracioli 2014].

Estes fatos demonstram a necessidade de um sistema operacional de tempo real como o EPOS de ter suporte para uma plataforma embarcada multicore. A única arquitetura *multicore* suportada pelo EPOS é a IA32, que não é uma arquitetura que oferece previsibilidade, logo para todos os sistemas embarcados com restrições temporais uma arquitetura multicore com maior previsibilidade torna-se necessário.

O EPOS não possuia suporte para um processador embarcado *multicore*. Este trabalho trata de descrever como foi feito o porte [3] deste sistema operacional para a Zedboard, uma plataforma embarcada *multicore*.

O objetivo deste trabalho é portar o sistema operacional EPOS para a Zedboard. O porte consiste em adaptar os componentes que, dentro da arquitetura do EPOS, são chamados de mediadores de hardware. Cada mediador precisa ser refeito para cada plataforma[4].

---

[1] Comparado ao poder de processamento dos demais processadores embarcados.

[2] Real-Time Operating System

[3] Porte é um estrangeirismo da palavra *port*, que, no âmbito da computação, significa o ato de fazer um mesmo programa/sistema/SO funcionar em diferentes ambientes. Por exemplo, fazer um software que antes só funcionava no Linux passar a funcionar em um outro SO (sistema operacional) pode ser considerado um porte. Uma palavra alternativa que poderia ser usada é "suporte", entretanto acredito que esta palavra não expresse apropriadamente o que foi feito, já que essa palavra normalmente é associada com um serviço pago de assistência técnica, e o próprio fraseamento do que foi feito se tornaria de mais difícil compreensão e prolixo com esta palavra.

[4] Aqui chamamos de plataforma qualquer SoC (*System On-Chip*), ou processador. No capítulo 3 será explicado a diferença entre *machine* e *architecture*, e por enquanto estamos usando a palavra plataforma

2. Hardware-alvo

Zedboard é uma plataforma de desenvolvimento que suporta uma grande variedade de aplicações, visto que ela possui uma boa gama de interfaces e funções para habilitar isto. É dedicada à prototipação e *proof-of-concept*. Em seu interior ela possui um Xilinx Zynq 7000 (Z-7020), que é a arquitetura alvo deste porte. Zynq 7020 possui um processador *dual core* Cortex A9, cada core possui sua própria MMU e memória cache L1 (instruções e dados) privada.

3. EPOS

EPOS (Embedded Parallel Operating System) é um sistema operacional orientado a aplicação, cujo design chama-se ADESD (*Application-Driven Embedded System Design Method*), proposto por Fröhlich [Fröhlich 2001]. A ideia central do EPOS é prover um sistema operacional mínimo, de modo a minimizar o *overhead* da existência de um sistema operacional, deixando o processador livre para executar a aplicação do desenvolvedor [epo 2014].

3.1. Arquitetura do EPOS

Linguagem e paradigma: O EPOS é escrito em C++, e não em C, como é tradicionalmente feito. Como paradigma, é usado orientação à objetos, então cada componente do EPOS está encapsulado por uma classe (como a heap, timer, thread, etc). No EPOS também é muito usado conceitos como herança e metaprogramação estática (que será abordado mais a frente).

Mediadores de Hardware: Dentro da arquitetura do EPOS há o conceito de mediadores de hardware, que são os componentes (ou classes) dependentes de plataforma. Idealmente, as únicas classes que precisam ser modificadas e/ou reimplementadas são os mediadores. Há mediadores específicos da placa, como para a Pandaboard, a Zedboard e etc; abstraídos sob o nome de *machine* e mediadores específicos de um processador, abstraídos sob o nome de *architecture*. No código do EPOS, estes mediadores encontram-se nas pastas *mach* e *arch*.

Mediadores de hardware são uma alternativa ao tradicional uso de VMs[5] e de HAL[6], proposta por Fröhlich em seu trabalho *Application-Oriented System Design* [Fröhlich 2001]. O problema do uso de VMs é o seu *overhead* causado devido à tradução das operações da VM em código nativo. Já o uso de um HAL incorre no problema da manutenibilidade e dificuldade de adapção à novas arquiteturas muito distintas entre si [Polpeta and Fröhlich 2004]. O HAL não conseguiu passar pela "prova do tempo", e já está sendo considerado obsoleto por distribuições GNU/Linux populares, como o Ubuntu [lin 2010], sendo chamado de "uma grande não-manutenível bagunça monolítica" [hal 2014].

3.2. Inicialização

O EPOS inicia executando um conjunto de códigos escritos em assembly, chamados de crts. No crt, a primeira tarefa feita na inicialização do EPOS é a configuração das pilhas

para nos referenciarmos a estes dois conceitos simultaneamente.

[5]Virtual Machines.

[6]Hardware Abstraction Layer

do sistema. Feito isto, o EPOS trata de limpar a seção .bss, para futuro reuso. Antes de ser chamado o construtor dos demais objetos globais, a UART e Display são inicializados, assim é possível mais facilmente depurar o código dos construtores. Então o EPOS começa a chamar o construtor global de cada componente do sistema. O crt consegue localizar onde está localizado cada construtor devido à forma como o EPOS foi compilado e ligado (*linked*). A figura 1 ilustra num diagrama de sequência a ordem de construção dos objetos globais.

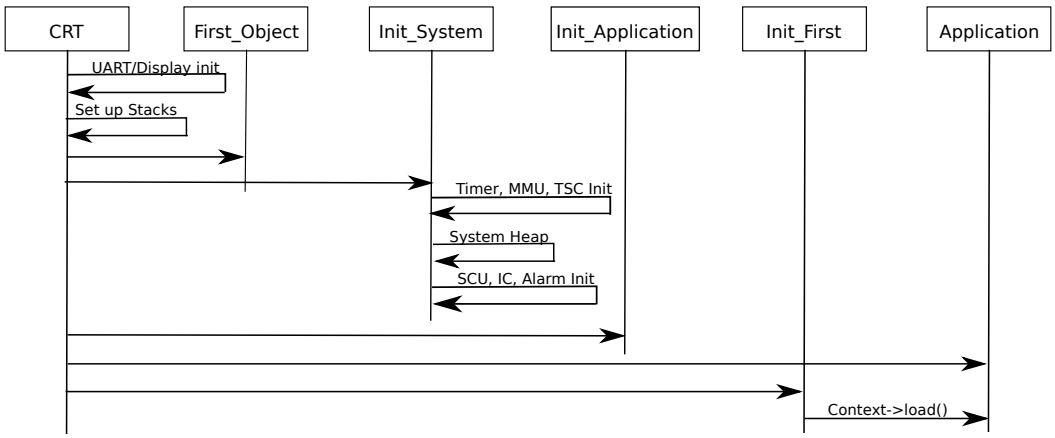

Figura 1. Diagrama de sequência da inicialização do EPOS

Primeiro é construído o `First_Object`, cuja principal função é apenas ser um ponto de entrada conhecido para o primeiro objeto do sistema. Em seguida, é construído `Init_System`. É neste construtor que todos os mediadores de hardware são construídos, com exceção da UART que já foi previamente construída no crt. Neste construtor, primeiramente é inicializado o *timer*, MMU, e, se disponível, o TSC (*Time Stamp Counter*). Em seguida o construtor aloca uma porção de memória para futuro uso da heap do sistema. Feito isto, o SCU (*Snoop Control Unit*), tratador de interrupções e alarme são também construídos.

`Init_Application` é responsável por alocar a heap da aplicação, através da requisição de páginas da MMU. `Init_First` chama o inicializador das *threads*, que então inicializa a *thread* da aplicação e a *idle thread*, que é uma *thread* especial que é executada quando o escalonador não encontra outra *thread* pronta para executar. Em seguida à criação destas *threads*, é chamado `context->load()` da *thread* da aplicação, fazendo com que o fluxo de execução seja finalmente transferido para a aplicação.

3.3. Relação Mediadores-Abstrações

Esta seção discute a relação entre os mediadores de hardware do EPOS (dependentes de arquitetura) com as abstrações do EPOS que usam estes mediadores (independente de arquitetura). O objetivo aqui é mostrar que estas abstrações não precisam ser alteradas entre as diferentes arquiteturas que o EPOS suporta, estando todas as diferenças arquiteturais isoladas em seus mediadores.

Cada mediador de hardware (que chamaremos apenas de "mediador" deste ponto em diante) possui uma classe genérica que o representa. Esta classe é a mesma para todas as arquiteturas, e cada novo mediador implementado deve estender esta classe, deste modo

pode-se manter a uniformidade das interfaces de cada novo mediador. Por exemplo, no caso da UART, cada novo mediador da UART precisa estender a classe `UART_Common`, como ilustra a figura 2. Note que o EPOS possui suporte para diversas arquiteturas, no diagrama são colocadas apenas duas por simplicidade.

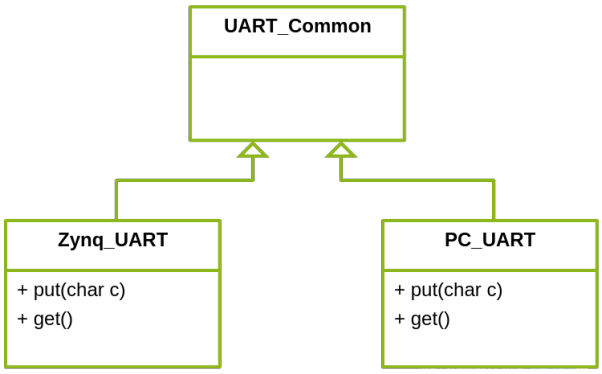

Figura 2. Diagrama de classes da UART.

A resolução do mediador da-se em tempo de compilação, através de metaprogramação estática e do uso de scripts automatizados. Para escolher a arquitetura alvo para compilar o EPOS, basta alterar o arquivo de traits `./include/system/traits.h`, trocando os valores das variáveis `ARCH` e `MACH`. Durante o processo de compilação, usando a ferramenta `sed`, o arquivo `./include/system/config.h` é alterado para alterar as variáveis `ARCH` e `MACH` para as escolhidas pelo usuário. Feito isto, macros são usadas para a resolução da inclusão dos *headers* corretos para a arquitetura escolhida. Segue abaixo algumas destas macros:

```
#define __HEADER_ARCH(X)          <arch/ARCH/X.h>
#define __HEADER_MACH(X)          <mach/MACH/X.h>
```

Para que o código independente de arquitetura do EPOS possa se referenciar às classes dos mediadores através de um nome genérico, para cada porte é feito um arquivo `config.h` onde são definidos tipos com nome padronizado que se referem aos mediadores daquela arquitetura. Por exemplo, um trecho do `config.h` do Zynq segue abaixo:

```
typedef ARMV7       CPU;
typedef ARMV7_MMU   MMU;
typedef Zynq        Machine;
typedef Zynq_IC     IC;
typedef Zynq_UART   UART;
typedef Zynq_Timer  Timer;
```

Com isto, todos os mediadores ficam abstraídos do ponto de vista do restante do sistema operacional, e podem ser trocados através da simples alteração de duas variáveis no arquivo de `traits`, seguida de nova compilação.

4. Implementação dos Mediadores de Hardware

Nesta seção será discutido como foi o porte de cada mediador de hardware, explicando as decisões tomadas. O processo de inicialização do sistema envolve as seguintes ações [ug5 2014, p. 110]:

1. Definir a tabela de vetores.
2. Invalidar *caches*, TLB, *branch predictor*
3. Preparar tabelas de tradução de página.
4. Configurar as pilhas dos diferentes modos de execução.
5. Carregar o endereço base da tabela de páginas no registrador apropriado para ser usado pela MMU.
6. Ativar a MMU.
7. Ativar a L2, ativar a L1.
8. Pular para o ponto de entrada a aplicação.

Estes e outros passos (como inicialização do GIC, timer, UART, CPU, etc) são explicados em detalhe nas seções que seguem. O primeiro passo a ser feito para iniciar um novo porte é incluir em `./include/system/types.h` os nomes das classes dos mediadores que serão implementados e criar as pastas necessárias para colocar estas classes, no caso deste porte, foram criadas as pastas `./include/mach/zynq/`, `./src/mach/zynq/`, `./include/arch/armv7/` e `./src/arch/armv7/`.

4.1. Mediador da UART

A inicialização da UART foi feita de acordo com o sugerido pelo manual em [ug5 2013, p. 554]. A primeira decisão que foi necessária é a de escabelecer qual será a taxa de transmissão (*Baud Rate*) da UART.

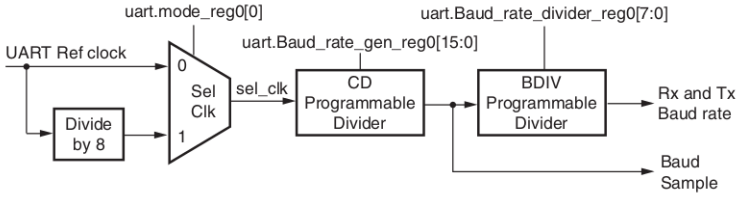

Figura 3. Esquemático de como é criada a taxa de transmissão.

Primeiramente foi necessário configurar o *clock* de referência da UART. Para isto, deve-se dividir o *clock* que vem do I/O PLL (que por sua vez deriva do PS_CLK, que é o *clock* geral do sistema). Recomenda-se dividir o *clock* da I/O PLL de modo a se obter 50 ou 33 MHz. O *clock* da I/O PLL, por padrão, multiplica o PS_CLK (de 33.33 MHz) por 26, resultando num *clock* de 866 MHz. No manual diversas vezes é usado como exemplo para o *clock* de referência (que é mostrado sob o nome de `UART Ref clock` na figura 3) da UART como 50 MHz, portanto, arbitrariamente, escolheu-se esse valor. Logo, deve-se configurar o registrador UART_CLK_CTRL, responsável por configurar o *clock* de entrada da UART, para dividir este *clock* vindo da I/O PLL por 17 ($866/17 = 50$).

Agora, com este *clock* estabelecido, que vamos chamar de sel_clk (de acordo com a nomenclatura do manual), devemos calcular quanto será a taxa de transmissão. Após alguma pesquisa em fóruns de desenvolvedores de software básico, foi observado que uma taxa de transmissão de 9600 bps é bastante comum, portanto, assumindo este valor, a próxima etapa é configurar os dois divisores de *clock* que ajustam a taxa de transmissão, como indicado na figura 3.

O primeiro divisor chama-se CD (*clock divider*), que configura a constante para se dividir o *clock*, e BDIV um segundo divisor usado para sobreamostragem, organizado

como mostrado na figura 3. O valor da taxa de transmissão final é calculado da seguinte forma:

$$\text{taxa de transmissão} = \frac{sel\_clk}{CD \times (BDIV + 1)} \quad (1)$$

O valor padrão de BDIV é 15, portanto, fixando-se este valor e resolvendo a equação por CD, temos que $CD = 325$. Configurando-se estes valores nos seus respectivos registradores, obtém-se a taxa de transmissão desejada de 9600 bps.

Após estas configurações, dentre outras que o manual descreve, a UART está pronta para ser usada. Os dois principais métodos usados da UART são o `put` e o `get`, o primeiro escreve um caractere na saída serial, sendo que este pode ser lido, por exemplo através de uma entrada USB, para imprimir estes caracteres numa tela; algo muito útil para depuração.

4.2. Mediador do *Timer*

Um *timer* permite contar um certo número de ciclos, e, ao final da contagem, ele emite uma interrupção ao processador, para que então o processador trate este evento. Entretanto note que é possível de existir mais *timers* sendo usados logicamente do que *timers* físicos disponíveis, significando que um mesmo *timer* deve conseguir servir a mais de uma requisição simultaneamente.

Portanto não podemos apenas configurar um *timer* para contar initerruptamente até passar o tempo que desejamos, do contrário novas requisições sobreescreveriam a anterior. Para ilustrar, suponha que se queira contar por 10 segundos, como o *clock* do *timer* é de 333 MHz (periodo $\frac{1}{333\times10^6}$s), bastaria configurar o *timer* para contar por $10 \times 333 \times 10^6$ ciclos e então chamar o *handler* associado à interrupção gerada quando o *timer* chegar em zero.

Agora imagine que, no cenário acima, enquanto o *timer* ainda está servido àquela solicitação de contagem, apareça outra solicitação, de um alarme por exemplo, e queria contar por 20 segundos. Se esta solicitação sobrescrever o registrador de configuração do *timer*, a solicitação anterior não terá seu pedido atendido a tempo. Note também que o escalonador de processos também estará usando este *timer*.

Para resolver este problema, na arquitetura do EPOS existe o conceito de ticks (algo parecido com o que se faz no Linux), onde se configura um *timer* para gerar interrupções em um intervalo regular, intervalo este que deve ser pequeno o suficiente para poder atender a demanda de contagens de pequenos valores, assim como não ser pequeno demais ao ponto de gastar mais processamento tratando as interrupções geradas pelo *timer* do que servindo à outras funções. Assim, cada objeto que instancia (ou usa) um *timer*, como o Alarm, Scheduler e Chronometer, nunca realmente tocam em algum registrador do *timer* (portanto esses componentes são independentes de arquitetura), e, no lugar disso, computam quantos ticks, isto é, quantas interrupções de *timer* aconteceram.

O construtor do `Zynq_Timer` recebe como parâmetro a frequência que o contador deve contar, assim como o *handler* que deve ser chamado quando esta contagem terminar (isto é, a função chamada quando acontecer uma interrupção devido ao *timer* ter chego a zero), e um número chamado `channel`, que serve para demultiplexar qual *handler* deve ser chamado.

A classe `Zynq_Timer` possui um atributo estático (e portanto único para todas as instâncias) definido como `Zynq_Timer* _channels[CHANNELS]`, onde `CHANNELS` é uma constante com o número de diferentes classes usando o *timer* (Scheduler e Alarm). Este vetor é necessário pois, quando uma interrupção de *timer* acontece e o *handler* do *timer* é chamado, este *handler* pode iterar sobre ele, chamando todos os respectivos *handlers* daquelas classes.

Os 4 principais registradores a se trabalhar para configurar o *timer* são o `load`, registrador onde se escreve por quantos ciclos se deve contar; o `counter`, que é o registrador que contém o atual valor contado, sendo decrementado a cada ciclo até chegar em zero, chegando em zero o é gerada uma interrupção número 29; registrador `control`, que permite configurar certos comportamentos do *timer*, como o de ativa-lo, ativar modo cíclico, ativar interrupções e atribuir um valor para o `prescale`; e finalmente o registrador `interrupt status`, que, como o nome indica, permite que se leia o status das interrupções de timer. Todos os timers trabalham à metade do *clock* do sistema, ou seja, usando o *clock* CPU_3x2x.

Durante a inicialização do sistema, o *timer* é configurado para gerar interrupções periodicamente, e esta configuração não é alterada durante a execução da aplicação. Como o construtor do `Zynq_Timer` recebe uma frequência como parâmetro, é necessário converter esta frequência para um número de ciclos a se contar. Para isto, é necessário levar em conta a frequência com que o contador é decrementado, para então se definir um valor a ser decrementado periodicamente de modo a fornecer a frequência desejada.

Como sabemos que o *clock* ao qual o *timer* está submetido é metade do *clock* do sistema, e que antes dele chegar ao contador, este mesmo *clock* é dividido por um divisor chamado `prescaler` (que divide pelo valor configurado nele mais 1), podemos dizer a frequência F_dec de decremento do contador é de:

$$\text{F\_dec} = \frac{\text{CPU\_6x4x}}{2 \times (PRESCALER + 1)} \tag{2}$$

Logo, usando a mesma linha de racioncínio exposta no exemplo de cálculo de ticks, temos que o valor a ser carregado no `load_register` (que será chamado de `load_value`), sendo F a frequência que se deseja configurar o *timer*, é:

$$\text{load\_value} = \frac{\text{CPU\_6x4x}}{2 \times (PRESCALER + 1) \times F} \tag{3}$$

Ou de maneira mais simples:

$$\text{load\_value} = \frac{\text{F\_dec}}{F} \tag{4}$$

Precisamos agora definir o `prescaler`. Definimos ele como a razão entre o *clock* do sistema pela frequência desejada (prescaler $= \frac{\text{clock}}{2 \times F}$). Há a premissa de que a frequência desejada não será maior que a do *clock* do *timer*, pois é impossível contar mais rápido que isto. Normalmente esta razão ($\frac{\text{clock}}{2 \times F}$) será um número maior que 255, já que o clock costuma ser muito mais rápido, e como o campo onde se registra o valor do prescaler possui apenas 8 bits, frequentemente o prescaler será 255.

4.3. Mapeamento de Memória

Por padrão, as 8 primeiras palavras da memória (ou seja, os primeiros $8 \times 4 = 32$ bytes) devem possuir instruções específicas. A primeira palavra (memória posição 0) contém a primeira instrução a ser executada, e, nas 7 palavras seguintes, fica a tabela de vetores (*vector table*). Como abaixo da instrução inicial há uma tabela que não se deseja executar no momento de inicialização do sistema, esta primeira instrução necessariamente é um *jump* para uma outra região, para aí então se iniciar o processo de *boot*. As demais 7 palavras, pertencentes à tabela de vetores possuem, similarmente, *jumps* para o código onde o tratador da exceção se localiza. A primeira instrução da tabela (posição 0x4) deve conter um *jump* o tratador de uma exceção do tipo undefined instruction, depois, nas próximas palavras, a software interruption, prefectch abort, data abort, reserved, irq e, finalmente, fiq, nesta ordem.

Também foi necessário definir as pilhas (*stacks*) do sistema, assim como reservar um espaço para a pilha dos tratadores de interrupção. Lembrando que pilhas, num sistema operacional, tradicionalmente crescem em direção à posições menores da memória, a pilha do usuário, portanto, localiza-se na última posição da memória (512 MB neste caso) e cresce para "baixo" (posições menores de memória) a partir de lá.

Para a MMU, são necessários 3600 KBs para mapear todo o espaço de endereçamento virtual (vide a seção 4.5.2 para entender porque apenas 3600 KBs).

Tabela 1. Mapeamento de memória.

Dado	Endereço base	Tamanho Alocado
APP_DATA	`0x00100140`	1KB
System Info	`0x00100540`	260 bytes
Livre	`0x00100644`	14KB
Tabela MMU	`0x00104000`	3600KB
SYS_HEAP	`0x00488000`	128MB

Estes valores são editáveis nos `traits`, entretanto também foi necessário atualizar o arquivo `./include/mach/zynq/memory_map.h`.

4.4. Controlador de Interrupções

Para se usar o controlador de interrupções (que será referenciado como GIC no restante desta seção, de *Generic Interrupt Controller*), é necessário antes inicializar o distribuidor de interrupções e as interfaces dos processadores.

É no distribuidor que é determinada a prioridade de cada interrupção, e onde é decidido se determinada interrupção deve ou não ser encaminhada para a interface de um determinado processador. Todas as interrupções passam por ele.

A interface dos processadores é por onde os processadores se comunicam com o GIC. Nela o processador pode confirmar que recebeu a interrupção (*acknowledge*), indicar que terminou de tratar a mesma, definir prioridades entre diferentes interrupções, indicar uma política de preempção de interrupções, ou mesmo desligar esta interface.

4.4.1. Inicialização

Na inicialização do distribuidor, através dos registradores mapeados em memória de configuração do mesmo, é definido, para cada uma das possíveis interrupções, se elas são *level-sensitive* ou *edge-triggered*. Em seguida configura-se a prioridade de cada interrupção. A princípio todas as interrupções foram definidas como tendo a mesma prioridade, mas isto é configurável caso necessário. É configurado então o processador-alvo de cada interrupção, isto é, para quais interfaces de processador uma determinada interrupção será encaminhada. Finalmente então são ativadas as interrupções [gic 2008].

A inicialização da interface do processador é mais simples. Primeiro se configura a máscara de prioridade da CPU, isto é, qual é o nível de prioridade mínimo que uma interrupção precisa ter para interromper aquele processador. Na implementação, esta máscara está desativada. Depois configura-se a política de grupos de preempção. No GIC é possível separar interrupções em grupos de preempção, onde é definido se determinado grupo pode preemptar determinado outro grupo. Todas as interrupções foram colocadas no mesmo grupo e interrupções podem ser preemptadas.

Adicionalmente a estas configurações, é possível de mascarar as interrupções FIQ e IRQ através do CPSR (*Current Program Status Register*), alterando-se os bits 6 e 7 dele, para mascarar interrupções FIQ e IRQ, respectivamente. Normalmente é o que é feito quando é necessário mascarar as interrupções temporariamente, enquanto se mantém as configurações do GIC.

4.4.2. Fluxo de execução ao se receber uma interrupção

O processador, ao receber uma interrupção, (portanto as interrupções estão ativas e não mascaradas pela interface ou pelo CPSR), para a execução do código que estava executando e então executa a instrução contida na tabela de vetores (mostrada na página **??**) correspondente ao tipo de interrupção recebida. Esta instrução é um *jump* para um tratador (*handler*) daquele tipo de interrupção.

Após selecionar qual handler chamar do vetor de exceções, o processador muda de modo, indo, no caso de uma interrupção IRQ, para o modo de execução IRQ. Neste modo há 3 registradores banqueados: O SPSR, que contém o valor do registrador CPSR imediatamente antes da interrupção, sendo necessário para que seja possível restaurar o valor original do CPSR após tratar a interupção; o LR (*link register*), que contém o endereço da próxima instrução que seria executada imediatamente antes da interrupção mais 4; e, finalmente, o SP (*stack pointer*), que aponta para a pilha própria desde modo (cada modo pode possuir sua própria pilha).

Agora será discutido como que as interrupções são tratadas individualmente. O corpo do `int_handler` é bastante curto, então vale a pena escreve-lo aqui:

```
void Zynq_IC::int_handler()
{
    unsigned int icciar_value = CPU::in32(IC::GIC_PROC_INTERFACE
        | IC::ICCIAR);
    IC::Interrupt_Id id = icciar_value & IC::INTERRUPT_MASK;
```

```
    if(id == 1023){
        kout << "Spurious interruption received\n";
        return;
    }
    _vector[id](id);
    CPU::out32(IC::GIC_PROC_INTERFACE | IC::ICCEOI, icciar_value);
}
```

A primeira coisa que o tratador faz é descobrir qual é o número da interrupção que foi gerada, para assim saber como tratar aquela interrupção. Isto é feito lendo-se o registrador ICCIAR (*Interrupt Acknowledge Register*), que provê o número da interrupção e também o processador endereçado. É possível que uma interrupção já tenha sido tratada por outro processador, quando isto acontece, o GIC emite uma *Spurious Interruption* para indicar isto. Quando se detecta isto, o tratador não precisa tomar nenhuma outra ação, basta retornar a execução normal.

Com o número da interrupção em mãos, pode-se então chamar o tratador daquele tipo de interrupção. O `_vector` é um vetor de *handlers*, onde para cada posição i, existe o tratador da interrupção número i. Para sinalizar que uma interrupção foi tratada, deve-se escrever no registrador ICCEOI (*End of Interruption*) o número lido no ICCIAR (ou seja, o número da interrupção e processador de destino).

Normalmente, interrupções de *timer* são o tipo mais frequente de interrupção durante a execução do sistema. Dele dependem o escalonador de processos (ou *threads*), Delay, Chronometer e Alarm. Como mencionado na seção do porte do *timer*, uma mesma interrupção pode gerar a chamada de mais de um handler, como a do *timer* que chama a do Alarm e do escalonador. Com isto fica ilustrado que uma única interrupção pode gerar a chamada de vários tratadores para esta mesma interrupção.

4.5. MMU

As principais funções de uma MMU (*Memory Managemend Unit*) são proteção de memória, ou seja, não permitir acesso a certas regiões da memória por processos não autorizados, e de fazer o mapeamento de memória virtual para memória física, mapeamento este que facilita a escrita de aplicações já que o desenvolvedor dela não precisará estar ciente de como a memória é mapeada internamente pelo sistema operacional.

A MMU consegue cumprir estes dois objetivos (e outros mais) através do uso de uma Tabela de Tradução de Páginas (que chamaremos de TTP), onde, a grosso modo, cada linha desta tabela é indexada pelo valor do endereço virtual, e na entrada correspondente há o endereço físico assim como *flags* que indicam, dentre outras coisas, se aquela região pode ou não ser acessada pela aplicação em execução.

Esta tabela é salva na memória principal (RAM) do sistema, e configurar a MMU significa especialmente criar métodos para gerenciar e popular esta tabela. Para entender como isto é feito, será explicado agora como é estruturada esta tabela, como que ocorre a tradução de memória virtual para física, e o que cada entrada da tabela deve ter.

4.5.1. Tradução

Vamos agora entender como se traduz de um endereço virtual, o endereço de uma TTP2. No coprocessador CP15, existe um registrador chamado TTBR (dois na realidade, TTBR0 e TTBR1), sigla de *Translate Table Base Register*, onde é guardado o endereço da primeira posição da TTP1. Com a MMU ligada, quando há um acesso à memória, automaticamente a MMU pulará para a posição indicada no TTBR, e indexará esta tabela usando os 12 bits mais significativos do endereço virtual. A maneira exata de como é feito este cálculo é mostrada na figura 4.

Na posição de memória encontrada no passo anterior, estará uma entrada da TTP1 em um formado determinado. No caso de uma entrada que indica uma outra TTP, os primeiros 22 bits indicam o endereço onde estará esta tabela, com os demais bits em 0, com exceção do último e os do campo de domínio (que será explicado a seguir). Com estas informações, a MMU já pode localizar a posição da TTP2. Para saber qual das entradas da TTP2 deve ser selecionada, é usado os bits [19:12] do endereço virtual para indexar a TTP2.

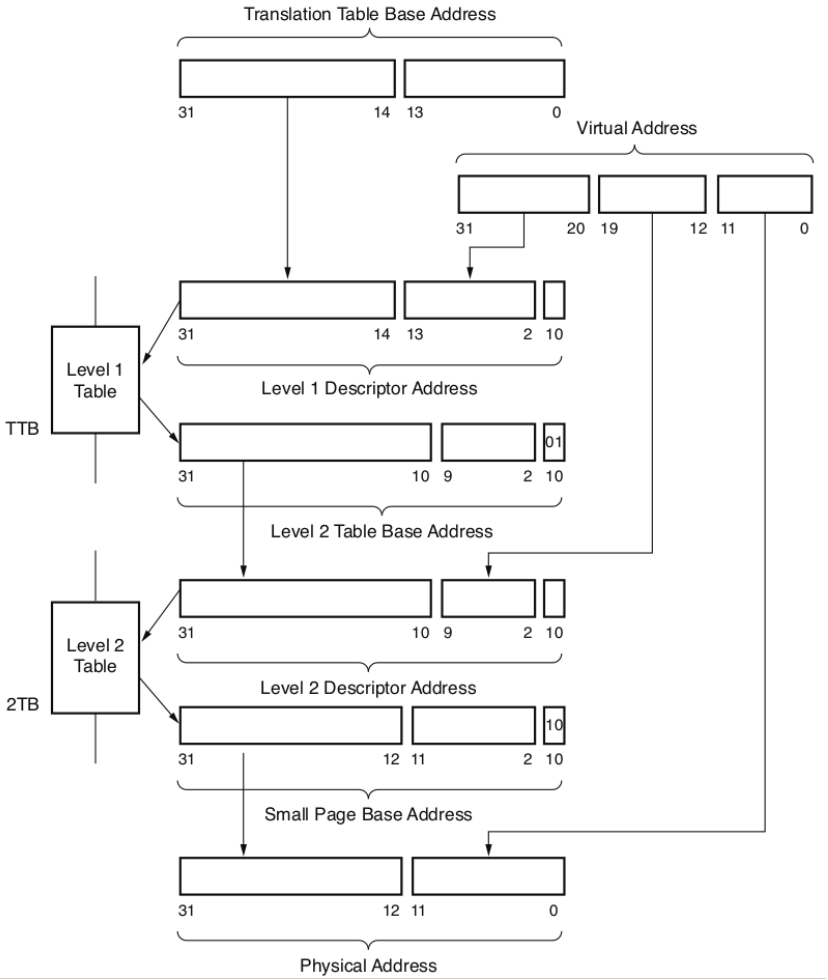

Figura 4. Processo de tradução de memória virtual para física.

A TTP2 possui 256 entradas, cada uma com o tamanho de uma palavra, logo,

cada TTP2 precisa de 1 KB de memória. Note que como uma entrada da TTP1 provê 22 bits para endereçar uma TTP2, esta TTP2 pode efetivamente estar localizada em qualquer região da memória, já que 22 bits são suficientes para indexar qualquer região de 1 KB de memória.

4.5.2. Inicializando a Tabela de Páginas

Agora que sabemos como as TTPs são estruturadas, bem como o formato de cada entrada dela, podemos finalmente popular esta tabela. Há várias estratégias para fazer isto [mmu 2014], dependendo da forma desejada de mapeamento.

No mapeamento adodotado, a princípio mapeia-se em em uma relação 1:1 os primeiros 512MB de memória virtual para os 512MB de memória física disponíveis, ou seja, o endereço virtual será o mesmo que o endereço físico. Entretanto note que com o uso de `Chunks` e `Address_space` é possível alocar uma porção de memória em um mapeamento próprio, como explicado na seção **??**.

Suponha que a MMU esteja ativa, e deseja-se salvar determinada informação numa determinada posição de memória física P, cujo endereço virtual correspondente seja V. Se apontarmos um ponteiro para a posição P, e escrever nesta posição, a MMU automaticamente irá interpretar P como sendo memória virtual, e irá traduzir P para uma outra posição de memória física qualquer e imprevisível.

Agora suponha que o mapeamento de memória esteja dividido no meio, de tal modo que a segunda metade faz uma relação 1:1 com a memória física. Deste modo, dado um endereço físico P, é possível calcular qual é seu endereço virtual V.

No EPOS foi feito um mapeamento semelhante, o intervalo de memória virtual [0x20000000 : 0x3FFFFFFF] mapeia para [0x0 : 0x1FFFFFFF]. Portanto `P | 0x20000000 == V`, onde | é o operador *or* de bits. Veja o seguinte exemplo de código:

```
static Log_Addr phy2log(Phy_Addr phy){
    return phy | PHY_MEM;
}
static Phy_Addr calloc(unsigned int frames = 1) {
    Phy_Addr phy = alloc(frames);
    memset(phy2log(phy), 0, sizeof(Frame) * frames);
    return phy;
}
```

`memset` irá tentar escrever no endereço enviado como parâmetro, que será interpretado como um endereço virtual, que então será traduzido para um físico correspondente. Entretanto no lugar de um endereço físico, é enviado o endereço lógico no lugar, para que este seja então traduzido para o endereço físico que desejamos. Esta é uma maneira do SO internamente burlar a tradução da MMU.

O restante da memória virtual (1GB-4GB) é traduzido numa relação 1:1. Esta região de memória corresponde principalmente a registradores mapeados em memória, não sendo, portanto, memória RAM. A figura 5 ilustra o mapeamento adotado.

Note que num mapeamento completo 1:1, a TTP1 tomaria 4×4096 bytes (4KB),

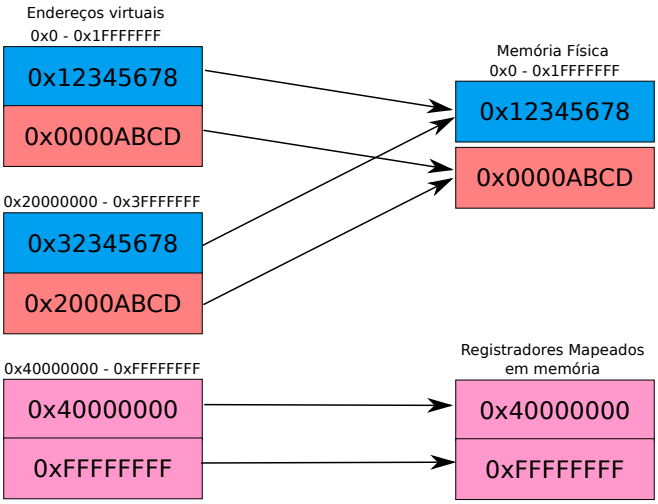

Figura 5. Mapeamento de memória adotado. Os primeiros 512MB de memória virtual são mapeados 1:1 para a memória física, enquanto os 512MB de memória virtual seguintes mapeiam para os os mesmos 512MB de memória física. De 1GB à 4GB é um mapeamento simples 1:1, referentes aos registradores mapeados em memória.

e as 4096 TTP2 (uma para cada posição da TTP1) tomariam $4 \times 256 \times 4096$ bytes (4MB). Entretanto como o mapeamento virtual do intervalo 512MB-1GB pode usar as mesmas tabelas TTP2 que foram criadas com o mapeamento do intervalo 0-512MB, pode-se economizar $4 \times 256 \times 512$ bytes. Portanto, no total, as tabelas de tradução de páginas usadas pela MMU usam 3600KB de memória RAM.

Em um mapeamento 1 para 1, para popular as TTPs, primeiramente, em um loop, itera-se 512 vezes (já que há 512 de RAM), salvando em posições sucessivas de uma determinada região da memória os endereços das TTP2s, levando-se em conta as *flags* acima mencionadas, após as 512 posições, itera-se pelas por mais 512 posições, mapeando novamente os primeiros 512 MB de memória física. Após isto, mapeia-se as posições restantes em relação 1:1. Feito isto, é necessário também popular cada TTP2 que foi apontada pela TTP1, em um processo semelhante. Em um mapeamento por demanda, marca-se as regiões da memória virtual como não mapeadas, com a diferença que, ao invés de apenas abortar o acesso, cria-se dinamicamente aquele mapeamento.

Após feita as tabelas, a MMU está pronta para ser ativada. A MMU é controlada pelo coprocessador CP15, é nele que são guardados os registradores de configuração, endereço base para a TTP, dentre outras funções. Escrevendo-se 1 no bit menos significativo no registrador c1 (SCTLR) deste coprocessador, a MMU é ativada.

4.6. Mediador da CPU

O mediador da CPU encapsula uma série de funções/rotinas usadas pelo sistema. A maior parte destas funções precisam ser escritas diretamente em assembly, ou usam informações dependente de arquitetura. Este mediador não precisa ser instanciado nem inicializado, ele é apenas um conjunto de funções necessárias por outros componentes.

Entre as funções deste mediador estão:

- Ativar/desativar máscara de interrupções de um dado processador.

- Rotinas para desligar, suspender e reiniciar o sistema.
- Função usada pelo escalonador para fazer a troca de contexto.
- Salvar/alterar registrador que guarda o endereço base da tabela de páginas.
- Funções primitivas usadas em semáforos e mutexes, como tsl (*test and set lock*), finc e fdec.
- Inicializar CPU 1.
- Funções para escrita/leitura de registradores.

Nesta classe também há uma classe interna chamada `Context`. Ela possui funções para salvar e carregar o contexto do processador, e atributos para salvar cada registrador de propósito geral da arquitetura (incluindo sp e pc e cpsr). Esta classe é usada no momento da criação de uma *thread*, e durante a troca de contexto.

4.7. Multicore

Ao se ligar a placa, apenas a CPU0 executa código, e é função dela fazer as inicializações necessárias para então ligar o segundo processador (CPU1). Inicialmente, a CPU1 executa uma instrução (`wfe`) que o coloca em modo de espera por evento (*Wait for Event Mode*). Para sair deste modo, a CPU0 precisa emitir uma instrução do tipo evento de sistema, `sev` (*System Event*). Quando a CPU1 recebe este sinal, ele imediatamente executa a instrução que está no endereço 0xffffffff0, localização onde a CPU0 deve previamente ter escrito uma instrução que faça a CPU1 fazer um `jump` para o ponto de entrada do que se deseja executar com a CPU1. Note que a posição 0xffffffff0 é uma região reservada da memória, e não corresponde à uma posição da RAM (é o mesmo princípio de registradores mapeados em memória).

Isto é suficiente para fazer o segundo processador passar a executar código, entretanto esta é a parte simples, ter dois processadores rodando tem várias implicações sobre como deve ser o *design* de cada componente do sistema, pois agora há a preocupação com coerência e consistência de memória, condições de corrida *racing conditions*) e etc.

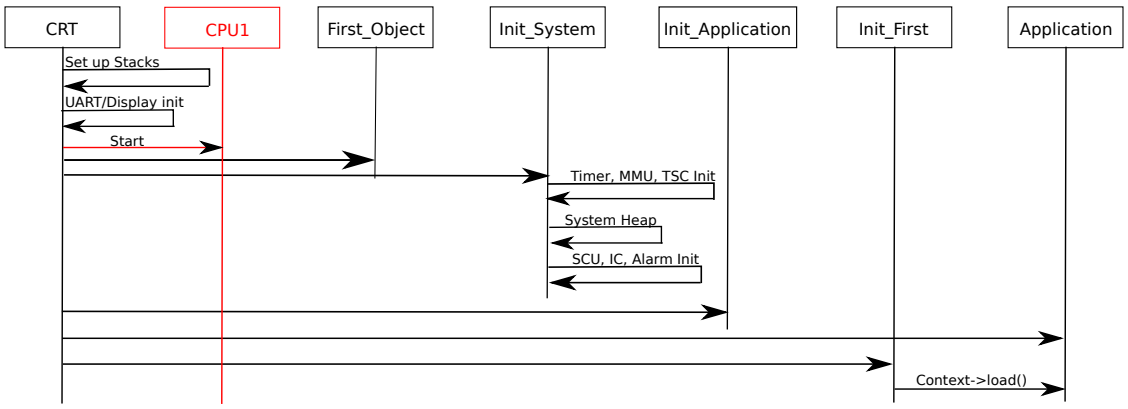

Figura 6. Ponto de entrada da CPU 1 durante a inicialização.

É necessário também definir um ponto de entrada para o segundo processador. Foi escolhido iniciar o segundo processador já desde os primeiros estágios da inicialização do sistema, como é ilustrado na figura 6. Dentro de classes como `Init_System`, onde são inicializado os mediadores do sistema, há desvios de fluxo (`if's`) para inicializar

apenas o que for pertinente para cada processador inicializar. Por exemplo, o *timer* pode ser inicializado para cada *core*, já a MMU não precisa ser inicializada duas vezes.

A decisão sobre qual é o melhor design depende muito do tipo da aplicação, e para as aplicações atuais do EPOS, o modelo simétrico é adequado.

4.8. Testes

Os testes foram feitos em dois ambientes: Simulação com o virtualizador de sistemas operacionais `qemu`, e com uma Zedboard física.

MMU: Como explicado na seção 4.3, e ilustrado na figura 5, os primeiros 512 MB do espaço de endereçamento virtual são mapeados numa relação 1:1 com a memória física, e então os 512 MB de memória seguintes são mapeados também para os primeiros 512 MB de memória física.

Logo, se a MMU estiver ativa, o endereço virtual X (para X $\in [0..512MB]$) e X+512 MB apontam para a mesma posição de memória física. Portanto, o teste consistiu em, antes de ligar a MMU, escrever um número numa posição de memória X, e então ler as posições X e X+512MB, e notar que evidentemente estas posições possuem valores diferentes. Então a MMU é ligada, e é lido novamente as posições X e X+512 MB, resultando agora no mesmo número, mostrando que a MMU esta ativa e traduzindo corretamente os endereços de memória virtuais.

UART: A UART funcionou sem problemas no `qemu`. Na placa, por via de um cabo usb, leu-se os dados pelo `cutecom`. Os caracteres são enviados com sucesso pelo cabo serial. Pode ocorrer a perda de alguns caracteres dependendo da taxa de transmissão escolhida.

GIC: No porte do GIC, o `qemu` mostrou diferentes comportamentos quando usado diferentes versões, fazendo com que certas configurações do GIC funcionassem apenas em algumas versões. Foi possível analisar o comportamento das exceções e dos seus tratadores, entretanto as interrupções não pareciam estar sendo geradas nos intervalos corretos configurados no `timer`, o que levou à conclusão que este componente só pode ser validado com precisão na placa física. Entretanto o GIC se mostrou não responsível na Zedboard, independente da configuração tentada. Foi feito um relatório a respeito, onde é exposto como, dentro das instruções do manual, este comportamento não pode corresponder ao esperado de acordo com o manual, indicando a possibilidade dele estar incompleto. Será necessário analisar outros códigos que usam a Zedboard.

Timer: O `timer` foi validado usando um `Chronometer`, para contar por uma certa quantidade de tempo, onde após isto foi lido quanto tempo passou. Devido aos problemas descritos acima, na placa física o `timer` foi testado lendo-se o registrador mapeado em memória referente ao atual número que está sendo contado no `timer`, onde constatou-se que ele de fato está operante e fazendo suas contagens de forma periódica.

Multicore: No `qemu` o segundo core foi ligado de forma bem sucedida, sincronizando sua inicialização nas barreiras que já existiam no EPOS com o primeiro *core*, inicializando apenas os mediadores que eram necessários, e finalmente criando e executando suas *threads* (thread `main` para o processador 0, e uma `idle` no caso do processador 1). Como o segundo *core* é ativado através de uma interrupção vinda da instrução `sev`, as dificuldades com o GIC impediram a análise do comportamento na placa.

5. Conclusão

A necessidade de um sistema operacional de tempo real de possuir suporte para uma plataforma embarcada *multicore* motivou este trabalho, que objetivou implementar o suporte do EPOS para uma plataforma embarcada *multicore*, a Zedboard. Neste trabalho foi discutido como se dá a interação software-hardware do sistema operacional com a placa, através do uso de mediadores de hardware, tanto no projeto do EPOS, quanto do interfaceamento fornecido pela placa. Em particular, a implementação dos mediadores de hardware do EPOS foi descrita.

O EPOS é o o primeiro RTOS de código aberto a suportar os escalonadores de tempo real global, particionado e agrupado [Gracioli 2014]. Assim sendo, este porte abre várias novas linhas de pesquisa de aplicação para este sistema operacional, em particular na área de escalonadores de tempo real. Uma possível nova aplicação do EPOS é o controle de um quadcóptero. Com os algoritmos de tempo real que estão implementados no EPOS, e com os recursos da Zedboard, ampla pesquisa pode ser feita, e novos horizontes de aplicação se abrem.

Como trabalho futuro, este porte pode servir para uma maior validação do escalonador feito em [Gracioli 2014], pois a Zedboard possui os recursos de hardware necessários para sua implementação (*lock/unlock* de linhas de cache, contadores de desempenho e capacidade de configurar a CPU-alvo de uma interrupção), e naquele trabalho foi usado um IA32 como plataforma de validação, que é uma arquitetura de menor previsibilidade que a Zedboard, e portanto essa pesquisa teria uma sustentação melhor sendo aplicada nesta placa.

Este trabalho descreveu como foi feito o porte do EPOS para a Zedboard. Este porte possibilita novos cenários de aplicação do EPOS e novas linhas de pesquisas, principalmente na área de escalonadores.

Referências

(2008). *ARM Generic Interrupt Controller, Architecture Specification*, version 1.0 edition.

(2010). Linux magazine.

(2013). Ubm.

(2013). *Zynq-7000 Technical Reference Manual*, ug585 (v1.6.1) edition.

(2014). Embedded bits.

(2014). Epos user guide.

(2014). National instruments.

(2014). Ubuntu wiki.

(2014). *Zynq-7000 Technical Reference Manual*, ug585 (v1.7) edition.

Fröhlich, A. A. M. (2001). *Application-Oriented Operating Systems*. PhD thesis, Technische Universität Berlin.

Gracioli, G. (2014). *Real-Time Operating System Support for Multicore Applications*. PhD thesis, Universidade Federal de Santa Catarina.

Polpeta, F. V. and Fröhlich, A. A. (2004). Hardware mediators: A portability artifact for component-based systems. In *Embedded and Ubiquitous Computing*.