César Augusto Marcelino dos Santos

Escalonador para baixo consumo de energia em sistemas de tempo real

10 de Agosto de 2013

César Augusto Marcelino dos Santos

Escalonador para baixo consumo de energia em sistemas de tempo real

Trabalho Final de Graduação apresentado ao Curso de Engenharia da Computação da Universidade Federal de Itajubá

Universidade Federal de Itajubá

Professor: Rodrigo Maximiano Antunes de Almeida

10 de Agosto de 2013

Lista de ilustrações

Lista de tabelas

Lista de códigos

Sumário

1 Introdução

Esta proposta apresenta um plano de trabalho para a elaboração do Trabalho Final de Graduação do curso de Engenharia de Computação da Universidade Federal de Itajubá (UNIFEI), cujo objetivo será a implementação de um escalonador de processos para sistemas de tempo real, baseado nas necessidades de processamento, o qual dinamicamente seleciona um modo de baixo consumo quando possível.

O hardware utilizado será um projeto fechado (INTERMEC..., 2013), no qual o foco estará sob o microcontrolador MSP430F5172.

1.1 Abstract

This proposal shows a working plan for the Final Thesis of Computer Engineering at Universidade Federal de Itajubá (UNIFEI), and the goal will be implementing a process scheduler for real time systems, based on load requirements, and then a low power mode will be dinamically selected based on this info.

The chosen hardware is a proprietary project (INTERMEC..., 2013), but the focus will be over the MSP430F5172 microcontroller.

1.2 Projeto de Pesquisa

Estudo e desenvolvimento de Sistemas Embarcados tem se expandido muito nos últimos anos e integram parte do cotidiano, desde relógios e câmeras até equipamentos móveis e smartphones (SANTOS; DEVERICK, 2013b). A programação para sistemas embarcados exige uma série de cuidados especiais, pois estes sistemas geralmente possuem restrições de memória e processamento. Por se tratar de sistemas com funções específicas, as rotinas e técnicas de programação diferem daquelas usadas para projetos de aplicativos para desktop's (BARROS; CAVALCANTE, 2002) (BARRETT; PACK, 2004). Além disso, Sistemas Embarcados possuem intrinsecamente outras preocupações como consumo de energia, dimensão, peso, rigidez, entre outros (JUANG et al., 2002).

Dentro deste vasto grupo encontram-se os Sistemas de Tempo Real, os quais exigem uma rápida resposta do sistema para determinadas tarefas (característica chamada de responsividade) (SANTOS; DEVERICK, 2013a).

Em termos de Ciência da Computação, o ramo de Sistemas Operacionais pode ser aliado à engenharia de Sistemas de Tempo Real, de modo que sejam desenvolvidos RTOS (Real Time Operating Systems, traduzido como Sistemas Operacionais de Tempo Real).

Tal termo significa que um SO é capaz de gerenciar diversos processos, porém alguns teriam uma prioridade em relação ao tempo de resposta do sistema, os quais são ditos processos de tempo real. Tal particularidade exige que recursos sejam direcionados a tal tarefa.

Sendo assim, supõe-se que há uma relação direta entre consumo de energia e responsividade, de modo que níveis aceitáveis são buscados em Sistemas Embarcados de Tempo Real. Para tal, pode-se modificar diversas características no seu projeto, desde a tecnologia de transistor utilizada, os modos de descanso de energia do microcontrolador, exigência de processamento das tarefas em fila, protocolos de comunicação entre os diversos periféricos, entre outros (SHAW, 2013).

1.3 Objetivo

Este projeto tem como objetivo desenvolver um escalonador de processos que permita diferentes níveis de baixo consumo de energia dependendo da necessidade de uso da CPU, baseado no SO de código aberto FreeRTOS (FREERTOS, 2013). O hardware utilizado possui design proprietário voltado a aplicações RFID, entretanto sua utilização se restringirá ao microcontrolador MSP430. O motivo pelo qual este circuito será utilizado é por questão de apoio de seu fabricante (INTERMEC. . . , 2013).

1.4 Metodologia

Num primeiro momento, será feita a implantação e modificação do sistema FreeRTOS, inserindo processos que permitam a medição do consumo de energia do microcontrolador para diferentes modos de energia. Em seguida, alteração e conclusão do sistema, de modo que o mesmo escolha o modo de energia que utilizará enquanto estiver ocioso.

A seguir estão apresentadas as ferramentas de hardware e software que serão utilizadas para o desenvolvimento do projeto.

1.4.1 Hardware

Como mencionado anteriormente, o hardware utilizado é proprietário da empresa Intermec (INTERMEC. . . , 2013), sendo simplificadamente composto de um circuito integrado especificado para atuar como front-end de uma antena em aplicações RFID, um microcontrolador MSP430F5172 e uma bateria 3 Volts. Como já comentado, não será utilizada a interface com a antena e quaisquer periféricos que este sistema já contenha. O microcontrolador se comunica com o circuito integrado e alimenta o mesmo, enquanto que a bateria somente permanece conectada ao microcontrolador, mas pode ser removida ou conectada manualmente ao circuito.

1.4.2 Software

O ambiente utilizado será o IAR Embedded Workbench, ferramenta proprietária para desenvolvimento de sistemas microcontrolados. A versão utilizada é especificamente para MSP430 da Texas Instruments, possui plugins para checar dados na memória RAM e Flash, variáveis internas, registros, inserção de breakpoints tanto no código fonte e na versão Assembly gerada, e também processos em fila no FreeRTOS.

2 Revisão Bibliográfica

2.1 Sistemas Embarcados

Apesar de ser amplamente utilizado, não há uma definição precisa. Segundo Gabriel Torres, são computadores para uma aplicação específica (TORRES, 2013), enquanto Silberchatz o define mais abrangentemente como dispositivos encontrados em tudo, de motores de carros a fornos microondas, que tendem a ter funções específicas (SILBERSCHATZ; GALVIN; GAGNE, 2013) . Além disso, podem ser utilizados para propósitos gerais, executando sistemas operacionais como Linux, porém com aplicações de propósitos especiais, ou então dispositivos de hardware os quais executam tarefas específicas sob a supervisão de um sistema operacional, ou até mesmo circuitos integrados de aplicação específica (ASIC) que executam sua função sem camadas de abstração de sistemas.

Neste setor, um dos destaques está na ascensão dos sistemas móveis, nas figuras de smartphones e tablets. Fisicamente, são projetados para serem portáteis e leves, e do ponto de vista histórico, se privam do tamanho de sua tela, capacidade de memória e funcionalidades em geral, que um computador desktop oferece normalmente, em troca de acesso a serviços de e-mail e navegação Web ao alcance das mãos (SILBERSCHATZ; GALVIN; GAGNE, 2013), como mostrado na Figura 1. Outro projeto que se popularizou nos últimos anos, especialmente para que pessoas que antes tinham a Eletrônica como hobby puderam desenvolver protótipos e projetos mais facilmente, foi a tecnologia de hardware livre Arduino, mostrado na Figura 2. Seus criadores a descrevem como uma plataforma de prototipação de eletrônica aberta, baseada em hardware e software flexíveis e de fácil uso (ARDUINO, 2006).

2.1.1 Arquitetura de Microcontroladores

Assim como os microprocessadores, em termos de conjunto de instruções, microcontroladores podem seguir arquiteturas do tipo RISC(*Reduced Instruction Set Computing*, também conhecida como arquitetura *load-store*, nas quais todas as instruções possuem operandos registro ou imediato, exceto as que acessam memória. Além disso, as instruções são de tamanho fixo do tipo aritmético-lógico, *load-store* ou controle) ou CISC(*Complex Instruction Set Computing* são um conjunto de instruções de tamanho variável, porém com aplicações mais específicas) (BAER, 2010).

Outra sub-categoria de arquitetura é quanto à informação armazenada na memória. Uma delas é a arquitetura Harvard, utilizada por alguns controladores PIC (SANTOS;

Figura 1 – Exemplo de sistema embarcado de alta capacidade de processamento

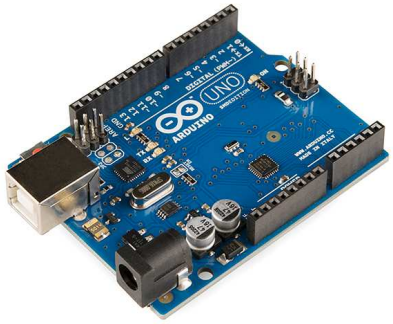

Figura 2 – Placa de desenvolvimento Arduino para prototipagem rápida de sistemas embarcados

ALMEIDA, 2012), na qual código e dados estão fisicamente separados, ou seja, em bancos de memória diferentes, não permitindo que o conteúdo de um programa modifique a si mesmo. Além desta, há o modelo de Von Neumann, definindo o conceito de que um programa fosse armazenado em memória, sendo também conhecido como arquitetura processador-memória. A diferença é que este último possui código e dados compartilhados no espaço de memória (TORRES, 2013). O modo como memória e entradas são tratados por estes diferentes modelos podem ser vistos nas Figuras 3 e 4.

Devido às semelhanças expostas, os termos processador, microprocessador, controlador e microcontrolador serão usados sem qualquer distinção. Quando a diferenciação for necessária, a discriminação será efetuada.

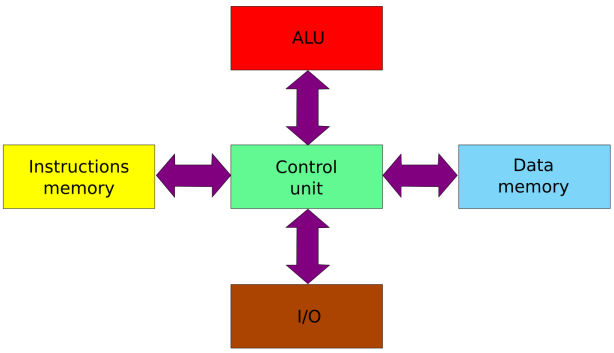

Figura 3 – Arquitetura Harvard

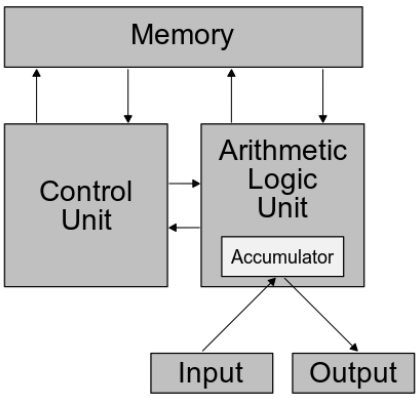

Figura 4 – Arquitetura Von Neumann

Para o microcontrolador utilizado neste projeto, MSP430F5172 é do tipo RISC de 16 bits, possui Memória Extendida (apesar de manipular simultaneamente 16 bits de dados, seu barramento de endereços é de 20 bits, então possui múltiplos modos de endereçamento e instruções especiais para lidar com esse desemparelhamento) (T.I., 2010). Suas instruções operam, em grande parte, sobre registros internos, utilizando 7 diferentes modos

de endereçamento para operandos fonte e 4 modos de endereçamento para operandos destino. Possui 16 registros, dos quais 12 são de propósito geral.

2.1.2 Modos de Baixo Consumo

Características gerais de sistemas embarcados e sistemas móveis são aplicabilidade (prático e de fácil uso), resistência, resposta à interação do usuário em tempo hábil (sem atrasos), portabilidade (pequenas dimensões e pouco peso) e baixo consumo de energia (SHAW, 2013). Para isso, fabricantes de processadores e controladores buscam otimizar o consumo de energia na execução de suas instruções. Outro meio é permitir que o próprio programador indique ao circuito quando deve diminuir suas atividades e baixar seu consumo.

Uma das técnicas é alterar os valores dos registros internos do microcontrolador para que ele diminua a frequência de seus temporizadores internos ou sua taxa de *clock* - *clock rate*. A segunda opção é, quando o processador estiver ocioso, que ele entre em modo de economia de energia ou de baixo consumo (dependendo do fabricante, pode ser chamado *Low Power Mode* ou *Power Saving Mode*). Estas duas técnicas podem ser misturadas, mas não há necessariamente uma regra geral para quando cada um deva ser usada, dependendo da especificação do projeto (KING, 2008) (em outras palavras: é melhor aumentar a frequência para que o controlador processe mais rapidamente e haja mais tempo para que "durma", ou deve diminuir sua frequência para economizar energia e permanecer mais tempo deste modo?).

O microcontrolador utilizado possui 6 diferentes modos de operação: ativo, LMP0 (*Low Power Mode 0*, LPM1, LPM2, LPM3 e LPM4. A diferença entre eles define o número de clocks internos trabalhando, assim como se a CPU está ativa ou não. Além de variar a quantidade de corrente consumida em cada um destes modos, também muda o tempo que é necessário sair de um modo LPMx para o modo ativo. Há, ainda, um outro modo denominado LPM4.5, porém o controlador por si só não é capaz de restaurar sua execução, dependendo de algum sinal ou interrupção externa.

2.2 Sistemas Operacionais de Tempo Real

2.2.1 Sistemas Operacionais

Assim como Sistemas Embarcados, não há uma definição precisa para SO(Sistema Operacional), porém, segundo Silberschatz (SILBERSCHATZ; GALVIN; GAGNE, 2013), é um programa que gerencia o hardware do computador. Caso sejam definidos grupos maiores dessa gerência, seriam três: processos, memória e dispositivos de entrada/saída de dados, ou *I/O*, como pode ser visto na Figura 5. Um Sistema Operacional serve não somente para facilitar a programação e a comunicação com os componentes físicos do sistema, mas

também para proteger o sistema de possíveis violações, seja ela material ou a nível de software.

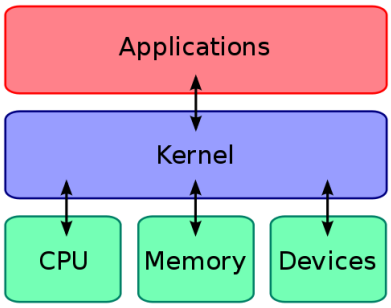

Figura 5 – Interação entre as camadas de um SO

Gerência de Processos

Em termos de processos, um importante componente é o chamado Escalonador, entidade responsável por colocar uma ordem de execução dos processos em fila. Eles podem ser executados na ordem em que foram inseridos até o seu fim, cada processo pode ter uma parcela de tempo de processamento (chamado *timeslice*), os processos podem ter diferentes prioridades de execução, ou então um processo pode ser interrompido no meio e os recursos de processamento se dedicam a outro. Esses tipos de escalonadores se chamam cooperativo, *round-robin*, de prioridade e preemptivo, respectivamente. Estes podem ser misturados para que o sistema se comporte de uma forma mais adequada para o que foi projetado.

Um detalhe importante é que, a cada momento que um processo é trocado por outro na execução, é necessário garantir que os dados que eram utilizados por ele (dados locais) se mantenham íntegros, assim há consistência na execução. Este método é chamado *Troca de Contexto* ou *Context Switch*, e os dados que são salvos por cada processo a cada troca constituem um PCB - *Process Control Block*.

Gerência de Memória

Em termos de memória, informações podem ser compartilhadas entre processos utilizando a memória RAM como intermediária, ou então uma certa região da memória pode ser bloqueada quando um processo a está acessando. Caso a restrição de acesso seja de um processo por vez, este contexto é chamado de *Região Crítica*. Se um processo se encontra em região crítica, é necessário assegurar o que é chamado *Exclusão Mútua*, ou seja, um mesmo recurso não pode ser acessado por dois processos simultaneamente. Técnicas como semáforos e *mutexes* podem ser utilizadas, ou seja, simplificadamente um detentor de um semáforo, ou uma entidade externa ao processo garante acesso único em um instante de tempo a um recurso (que pode ser memória ou dispositivo de I/O).

Outra função importante é definir a alocação de memória para os processos, ou seja, se eles poderão acessar outras posições de memória externas ao seu escopo, ou então quanto poderão utilizar de memória local (podendo esta ser alocada dinamicamente, determinando uma região chamada *Heap*, ou então de armazenamento temporário, através da *Pilha*). Tais segmentos são exemplificados na Figura 6.

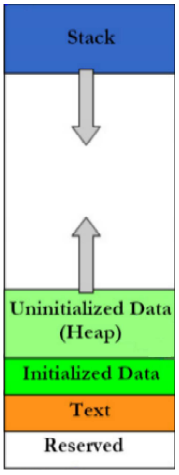

Figura 6 – Os diferentes segmentos existentes em um programa

Outro meio de permitir que processos troquem informações é a utilização de troca de mensagens ou IPC(*Inter Process Communication*).

Gerência de Dispositivos de Entrada/Saída

Em termos de dispositivos de I/O, uma camada de abstração de software pode ser desenvolvida, de modo que os processos do sistema tenham uma interface de fácil acesso ao hardware (ou mesmo, impedir que o acesso aos recursos sejam ilimitados). Para tal, uma *Controladora de Drivers* fornece as permissões e as camadas de abstração para cada periférico disponível, e esses periféricos são enxergados como abstrações em software, chamados de *Drivers*.

Sistemas de Tempo Real

Entretanto, quando se trata de um SO de Tempo Real, tem-se a impressão enganosa de que o mesmo executará suas tarefas com alta velocidade em relação ao tempo humano. O fato de ser dito *Real Time* significa que se torna determinístico o momento em que a execução ou processamento de uma certa tarefa se iniciará.

Os meios de medir a performance de um sistema de tempo real é através de duas características: responsividade e *throughput* (CHRISTOFFERSON, 2013). O primeiro significa o quão rápido o sistema responde a um pedido de um processo, ou seja, o quão determinístico é o processamento de uma tarefa. Já o segundo é a quantidade de dados que

o sistema consegue enviar à medida que os processos são executados. Uma característica pode levar a outra, mas não necessariamente.

Para clarificar esta situação, há três classificações de sistemas de tempo real:

- *Soft Real Time*: erros no determinismo do sistema são toleráveis, e sua frequência determina a qualidade do serviço(*QoS - Quality of Service*, quantidade de dados entregues ao usuário). Comum para aplicações multimídia, em que a alta precisão de transmissão de cada bit não é necessária para compreensão da informação, ou seja, busca-se um equilíbrio entre *throughput* e responsividade;
- *Firm Real Time*: erros no determinismo do sistema são toleráveis, mas sua frequência pode torná-lo inutilizável. Um exemplo são a transmissão de dados em radio-frequência, como conexões 3G e *Wireless*. Nesta situação, o importante não é responsividade, mas *throughput*.
- *Hard Real Time*: erros no determinismo do sistema causam falhas consideráveis ou graves no sistema. Exemplo são aplicações médicas, como marca-passo ou então controle do sistema de direção de um veículo. Neste caso, responsividade é o foco.

2.2.2 FreeRTOS

FreeRTOS é um dos destaques do mercado em termos de Sistemas Operacionais de Tempo Real(também chamados RTOS, acrônimo de *Real Time Operating Systems*) com código aberto e permissão GPL, portável para mais de 30 arquiteturas. A missão do grupo que o mantém é "disponibilizar um produto gratuito que supere a qualidade e os serviços exigidos por usuários de alternativas comerciais": Figura 7 (FREERTOS, 2013).

Figura 7 – Logo do sistema FreeRTOS

O fato de ser GPL permite que seu código possa ser copiado na íntegra, e ainda assim, proteger a propriedade intelectual do sistema. Entretanto, qualquer modificação feita nos arquivos originais, no *kernel* do sistema, estes arquivos não podem possuir código fechado, e devem fazer, ao menos, referência ao site de distribuição do FreeRTOS.

Quando foi criado seu projeto, a intenção é que se tornasse um kernel de tempo real que atuasse em pequenos sistemas embarcados. Dentre as características que possui:

- escalonador preemptivo, cooperativo e possibilidade híbridas;

- implementações voltadas para baixo consumo, como o modo *tickless*: não há um clock interno que pede a troca de contexto;
- desenvolvido para ser pequeno, simples e de fácil uso, de modo que o arquivo binário do kernel tenha entre 4K e 9Kbytes;
- Código-fonte altamente portável, predominantemente escrito em C;
- Suporte a *tasks* e co-rotinas;
- Opção de detecção de estouro de pilha;
- Aplicações demo pré-configuradas, permitindo rápido aprendizado;
- Sem restrições quanto ao número de tasks de tempo real, número de prioridades e é livre de royalties;
- Entre outros.

Os processos do FreeRTOS podem ser tasks ou co-rotinas. O primeiro permite maior segurança para o contexto (aqui chamado de TCB) de cada task, e também limita seu escopo, tornando uma task FreeRTOS simples, sem restrições, com opções de preempção e prioridade. Já as co-rotinas foram desenvolvidadas para os casos em que a memória RAM é limitada, e como consequência a região de memória entre todas as co-rotinas é compartilhada. Em outras palavras, implementar co-rotinas permite compartilhamento de resultados com muito menos uso de memória primária, escalonador cooperativo permite reentrância de processos, e é facilmente portável para outras arquiteturas.

Os diferentes estados que tasks e co-rotinas podem ter estão representados nas Figuras 8 e 9. Os estados são descritos como Disponível (*Ready*), Execução (*Running*), Bloqueado (*Blocked*) e Suspenso (*Suspend*). Assim que uma task é criada ou quando está pronta para ser executada, vai para o estado Disponível; se estiver em execução pelo sistema estará em modo Execução; caso esteja indisponível indeterminadamente, a não ser que seja explicitamente chamada pelo sistema para o modo Disponível é definido como estado Suspenso; se uma task está indisponível aguardando algum evento (normalmente relacionado a temporização, como por exemplo esperar 2 segundos para ser re-executada) está em modo Bloqueado.

Este sistema já define padrões próprios para tipos de variáveis e funções, assim como convenções de chamada de funções, criar tasks, alocar memória, etc.

Além disso, quando não há mais task alguma para ser executada, uma pseudo-task chamada IDLE passa a ser executada, sem implementação. Com isso, é possível disparar uma tarefa que seja específica quando a fila de processos estiver vazia.

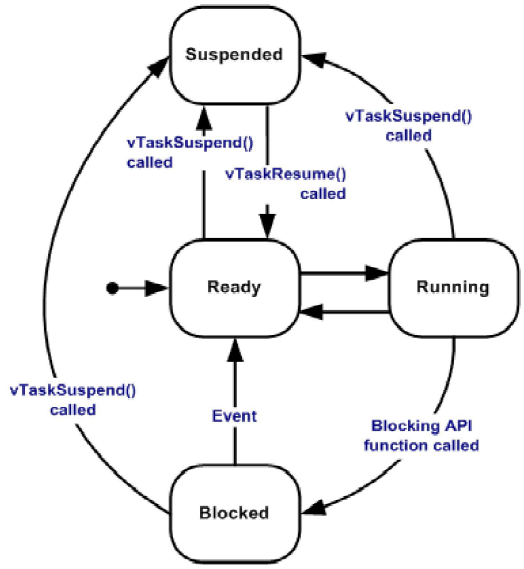

Figura 8 – Os possíveis estados de uma task

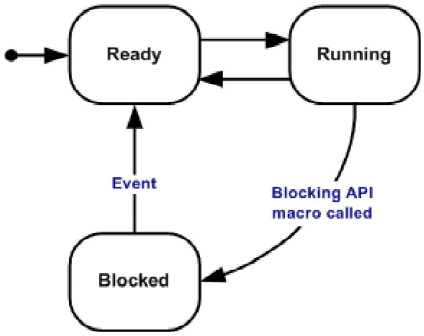

Figura 9 – Os possíveis estados de uma co-rotina

2.2.3 Consumo de Recursos

Aplicações em sistemas embarcados necessitam estar a par dos recursos disponíveis. Quando se está desenvolvendo para sistemas de pequeno porte, recursos como memória, processamento, número de portas de I/O, e mesmo consumo de energia, se tornam limitados. Deste modo, inserir camadas de abstração como um sistema operacional implica nisso. Porém, interfaces para proteção e distribuição destes recursos são fornecidas.

Sendo assim, o número de tarefas que o sistema irá executar, ou o volume de dados que deve manipular implicará na quantidade necessária de processamento e gasto energético (DICK et al., 2000) (KING, 2008) utilizado. Concluindo, uma tarefa com alta necessidade de recursos, sejam esses *CPU-bound* ou *I/O-bound* (SILBERSCHATZ; GALVIN; GAGNE, 2013) potencialmente pode ser dividida em várias sub-tarefas, ou então, a necessidade de manipular a frequência de processamento dinamicamente e os modos de economia de energia afetam os recursos do sistema como um todo.

3 Desenvolvimento

3.1 Portando o FreeRTOS

Os microcontroladores MSP430 possuem dois diferentes núcleos: 430 e 430X (T.I., 2010). Os núcleos 430 são tipicamente controladores de 16 bits, mas os de núcleo 430X possuem 20 bits para endereçamento de dados. Deste modo, o conjunto de instruções foi expandido para facilitar a manipulação das informações para este novo grupo.

Devido a este detalhe, as instruções utilizadas para, por exemplo, troca de contexto entre tarefas não é a mesma para os dois núcleos. Parte do código de salvamento de contexto (guardar valores atuais dos registros na pilha, assim como a janela de pilha - também chamada de *stack frame* (PINTO, 2006)) é mostrado nos Códigos 1 e 2.

Deste modo, deve-se conhecer bem o hardware utilizado para que o sistema seja o mais modular possível. Entretanto, o fato de depender da arquitetura invariavelmente atinge um ponto em que pedaços do código são criados especificamente para um determinado processador ou controlador.

Código 1 – Versão do código de salvamento de contexto para o núcleo 430

```
1  portSAVE_CONTEXT macro
2
3      IMPORT pxCurrentTCB
4      IMPORT usCriticalNesting
5
6      /* Save the remaining registers. */
7      push  r4
8      push  r5
9      push  r6
10     push  r7
11     push  r8
12     push  r9
13     push  r10
14     push  r11
15     push  r12
16     push  r13
17     push  r14
18     push  r15
19     mov.w &usCriticalNesting, r14
20     push  r14
21     mov.w &pxCurrentTCB, r12
22     mov.w r1, 0(r12)
23     endm
```

Código 2 – Versão do código de salvamento de contexto para o núcleo 430X

```
1 portSAVE_CONTEXT macro
2
3   /* Save the remaining registers. */
4   pushm_x #12, r15
5   mov.w &usCriticalNesting, r14
6   push_x r14
7   mov_x &pxCurrentTCB, r12
8   mov_x sp, 0( r12 )
9   endm
```

3.2 Arquitetura do FreeRTOS

O código completo do FreeRTOS está disponível em seu site (FREERTOS, 2013), e é subdividido em uma hierarquia de pastas e módulos opcionais a serem incluídos no projeto. Para este trabalho, as partes opcionais não serão utilizadas, e só algumas funções do *core* serão aprofundadas, como explicado mais adiante. As subdivisões serão estudadas como *Portable*(individual para cada arquitetura), *Memory Management*(tratando a respeito de alocação de memória), *Includes*(gerando várias definições e macros básicas para operar o resto do *kernel* do sistema), *Lists*(definindo listas encadeadas, as quais serão usadas para o escalonador do sistema) e *Tasks*(trata e manipula as tarefas do sistema), sendo estes os macrocomponentes básicos para que o sistema funcione apropriadamente para o estudo desejado.

3.2.1 Arquivos específicos da arquitetura (Portable)

Grupo específico de arquivos para a arquitetura desejada, implementa definições e macros próximos ao nível de máquina.

data_model.h

Para a arquitetura MSP430X, há três tipos de modelos de dados: *Small*(todas operações são baseadas em 16 bits),*Medium*(operações envolvendo pilha utilizam 20 bits) e *Large*(operações envolvendo memória são de 20 bits).

port.c

Define um ponteiro para a *task* corrente (que pode ser acessada globalmente), assim como uma variável de controle de região crítica (a todo momento que atingir o valor 0, significa que as interrupções estão desabilitadas). Além disso, possui uma sub-rotina de inicialização da pilha de cada task.

Código 3 – Rotina de Tratamento de Interrupção do Timer de tick do sistema

```
1  vPortTickISR:
2
3    /* The sr is not saved in portSAVE_CONTEXT() because vPortYield() needs
4    to save it manually before it gets modified (interrupts get disabled).
5    Entering through this interrupt means the SR is already on the stack, but
6    this keeps the stack frames identical. */
7    push.w sr
8    portSAVE_CONTEXT
9
10   calla #xTaskIncrementTick
11   cmp.w   #0x0, R12
12   jeq     SkipContextSwitch
13   calla   #vTaskSwitchContext
14 SkipContextSwitch:
15   portRESTORE_CONTEXT
```

portmacro.h

Define os tipos de variáveis recomendados, de modo que uma mesma aplicação pode ser portada a diversas arquiteturas (por exemplo, um *int* em uma arquitetura 16 bits pode ser de 16 bits, enquanto que em uma arquitetura 32 bits será de 32 bits. Logo, uma compatibilidade para variáveis sinalizadas de 16 e 32 bits pode ser definida como *portSHORT* e *portBASE_TYPE*).

Também possui macros de seção crítica (entrada e saída) e uma assinatura genérica de tasks que deve ser utilizada como padrão.

portext.s43

Este arquivo define as instruções a nível de máquina, ou seja, implementadas em Assembly. Macros de salvar contexto, recuperar contexto, incrementar o *tick* do sistema, e realizar trocas de contexto são exibidas. O Código 3 exemplifica-se a rotina de tratamento de interrupção do timer de tick.

3.2.2 Gerenciamento de memória

Composto por diferentes opções de alocação e desalocação de memória, para este projeto, o foco se dá no esquema *heap_1*. Para este, só há alocação de memória, mas não há como desalocar. Isso implica que cada task criada será mantida no sistema enquanto ele estiver em execução.

Deste modo, define-se na memória RAM do microcontrolador uma área reservada (*static*) que atuará como *heap*: para o FreeRTOS, a Heap é uma região de memória na qual recursos para as tasks são alocados dinamicamente. Tal processo checa o quanto a task deseja de memória para si, e se há espaço suficiente.

Um trecho de código do arquivo é mostrado no Código 4.

Código 4 – Alocação dinâmica de uma task na memória RAM

```
1  void *pvPortMalloc(size_t xWantedSize) {
2      void *pvReturn = NULL;
3      static unsigned char *pucAlignedHeap = NULL;
4
5      /* Ensure that blocks are always aligned to the required number of bytes. */
6  #if portBYTE_ALIGNMENT != 1
7      if (xWantedSize & portBYTE_ALIGNMENT_MASK) {
8          /* Byte alignment required. */
9          xWantedSize += (portBYTE_ALIGNMENT - (xWantedSize & ↩
               portBYTE_ALIGNMENT_MASK));
10     }
11 #endif
12
13     vTaskSuspendAll();
14     {
15         if (pucAlignedHeap == NULL) {
16             /* Ensure the heap starts on a correctly aligned boundary. */
17             pucAlignedHeap = (unsigned char *) (((portPOINTER_SIZE_TYPE) & ↩
                   ucHeap[ portBYTE_ALIGNMENT ]) & ((portPOINTER_SIZE_TYPE) ↩
                   ~portBYTE_ALIGNMENT_MASK));
18         }
19
20         /* Check there is enough room left for the allocation. */
21         if (((xNextFreeByte + xWantedSize) < configADJUSTED_HEAP_SIZE) &&
22                 ((xNextFreeByte + xWantedSize) > xNextFreeByte))/* Check for ↩
                    overflow. */ {
23             /* Return the next free byte then increment the index past this
24             block. */
25             pvReturn = pucAlignedHeap + xNextFreeByte;
26             xNextFreeByte += xWantedSize;
27         }
28     }
29     xTaskResumeAll();
30
31 #if( configUSE_MALLOC_FAILED_HOOK == 1 )
32     {
33         if (pvReturn == NULL) {
34             extern void vApplicationMallocFailedHook(void);
35             vApplicationMallocFailedHook();
36         }
37     }
38 #endif
39
40     return pvReturn;
41 }
42
43
44 }
```

Basicamente, o funcionamento checa se *ucHeap* ainda possui capacidade para uma nova task ser criada, de acordo com o tamanho exigido pela mesma (*xWantedSize*). O processo de alocação ocorre com o SO em pausa (*vTaskSuspendAll()*), para garantir que o processo não será corrompido por uma troca de contexto. A mesma retorna um ponteiro para o início do contexto da task recém criada.

3.2.3 Configuração e definições (Includes)

Possui módulos para semáforos, filas de mensagem, entre outros. Para este projeto, se trata de um grupo de definições de pré-compilação e algumas macros de verificação de estouro de faixa (como *stack overflow*), para permitir que o sistema não fique sem os módulos requeridos. Por exemplo, se for selecionada a opção de fila de mensagens, porém o arquivo não pertence ao projeto, não será permitida a sua compilação.

Os arquivos são *FreeRTOS.h*, *mpu_wrappers.h*, *portable.h*, *projdefs.h* e *StackMacros.h*.

3.2.4 Implementação de lista encadeada

Este sistema é baseado na estrutura de lista encadeada, de modo que cada elemento é inserido após o outro. Como o escalonador e o *tick* manipulam tasks, estes são os elementos, e as ordenações são feitas em relação às suas prioridades. As definições de lista e elemento de lista podem ser vistos no Código 5.

Entretanto, as soluções-padrão utilizadas nas bibliotecas C possuem muito mais recursos do que o FreeRTOS necessita, e não são otimizadas para trabalhar com tasks, e esta é outra razão pela qual existem arquivos neste Sistema Operacional para tal. As funções de checar se a lista está vazia e inserção de elemento são bons exemplos, mostradas no Código 6.

3.2.5 Definição e implementação das Tasks

Esta parte é o foco deste projeto. As estruturas e os métodos que manipulam tasks no FreeRTOS são descritos e implementados nos arquivos *task.h* e *tasks.c*. Somente alguns elementos e funções serão melhor descritos a seguir, com o objetivo de oferecer uma melhor compreensão do funcionamento geral do sistema.

tskTaskControlBlock

O código a seguir descreve o bloco de controle de dados de uma task (aqui chamado de *Task Control Block* ou TCB, o que em teoria de Sistemas Operacionais equivale ao *Process Control Block* - PCB). Basicamente define-se os dados necessários para que cada task continue sua operação de onde parou, inclusive identificadores únicos, diferenciando cada tarefa do sistema. Tal estrutura é salva na pilha do sistema, mas para separar os *stack*

Código 5 – Definição de lista e de elemento de lista

```
1  /*
2   * Definition of the only type of object that a list can contain.
3   */
4  struct xLIST_ITEM {
5      configLIST_VOLATILE portTickType xItemValue; /*< The value being listed.  ↩
           In most cases this is used to sort the list in descending order. */
6      struct xLIST_ITEM * configLIST_VOLATILE pxNext; /*< Pointer to the next ↩
           xListItem in the list. */
7      struct xLIST_ITEM * configLIST_VOLATILE pxPrevious; /*< Pointer to the ↩
           previous xListItem in the list. */
8      void * pvOwner; /*< Pointer to the object (normally a TCB) that contains ↩
           the list item.  There is therefore a two way link between the object ↩
           containing the list item and the list item itself. */
9      void * configLIST_VOLATILE pvContainer; /*< Pointer to the list in which ↩
           this list item is placed (if any). */
10 };
11 typedef struct xLIST_ITEM xListItem; /* For some reason lint wants this as two ↩
       separate definitions. */
12
13 /*
14  * Definition of the type of queue used by the scheduler.
15  */
16 typedef struct xLIST {
17     configLIST_VOLATILE unsigned portBASE_TYPE uxNumberOfItems;
18     xListItem * configLIST_VOLATILE pxIndex; /*< Used to walk through the list. ↩
           Points to the last item returned by a call to ↩
           pvListGetOwnerOfNextEntry (). */
19     xMiniListItem xListEnd; /*< List item that contains the maximum possible ↩
           item value meaning it is always at the end of the list and is therefore ↩
           used as a marker. */
20 } xList;
```

frames, devem ser descritos os endereços de início e fim desta estrutura, para que não haja sobreposição, e consequentemente, perda de dados. Outros dados, como o nome e sua prioridade são também salvos, tal como pode ser visto no Código 7.

Estruturas relativas ao TCB

Sendo um sistema preemptivo, as tasks em execução estão constantemente se alterando, e muitas vezes, não são executadas até seu término. Com isso, acessar estruturas como lista para executá-las pode se tornar custoso, então uma estrutura global é utilizada para apontar na memória aonde se encontra a tarefa corrente em execução.

Além disso, como mostrado no diagrama da Figura 8, as tasks possuem diferentes estados. Para não manter uma lista global e facilitar a separação das que podem ou não serem executadas, listas são descritas para cada possível estado, como pode ser visto no Código 8.

O termo *PRIVILEGED_DATA* é uma palavra reservada reconhecida por alguns microprocessadores, os quais vão reservar uma região especial na memória na qual somente usuários com privilégios (em sistemas *Desktop* seria o equivalente ao Administrador ou Root) podem acessar, evitando violação de dados essenciais ao bom funcionamento do SO.

Código 6 – Exemplo de funções da biblioteca de listas do FreeRTOS

```
1  /*
2   * Access macro to determine if a list contains any items.  The macro will
3   * only have the value true if the list is empty.
4   *
5   */
6  #define listLIST_IS_EMPTY( pxList )        ( ( portBASE_TYPE ) ( ( pxList ↩
       )->uxNumberOfItems == ( unsigned portBASE_TYPE ) 0 ) )
7
8  void vListInsert(xList * const pxList, xListItem * const pxNewListItem) {
9      xListItem *pxIterator;
10     portTickType xValueOfInsertion;
11
12     /* Insert the new list item into the list, sorted in ulListItem order. */
13     xValueOfInsertion = pxNewListItem->xItemValue;
14
15     /* If the list already contains a list item with the same item value then
16     the new list item should be placed after it.  This ensures that TCB's which
17     are stored in ready lists (all of which have the same ulListItem value)
18     get an equal share of the CPU.  However, if the xItemValue is the same as
19     the back marker the iteration loop below will not end.  This means we need
20     to guard against this by checking the value first and modifying the
21     algorithm slightly if necessary. */
22     if (xValueOfInsertion == portMAX_DELAY) {
23         pxIterator = pxList->xListEnd.pxPrevious;
24     } else {
25         for (pxIterator = (xListItem *) &(pxList->xListEnd); ↩
               pxIterator->pxNext->xItemValue <= xValueOfInsertion; pxIterator = ↩
               pxIterator->pxNext) /*lint !e826 !e740 The mini list structure is ↩
               used as the list end to save RAM.  This is checked and valid. */ {
26             /* There is nothing to do here, we are just iterating to the
27             wanted insertion position. */
28         }
29     }
30
31     pxNewListItem->pxNext = pxIterator->pxNext;
32     pxNewListItem->pxNext->pxPrevious = pxNewListItem;
33     pxNewListItem->pxPrevious = pxIterator;
34     pxIterator->pxNext = pxNewListItem;
35
36     /* Remember which list the item is in.  This allows fast removal of the
37     item later. */
38     pxNewListItem->pvContainer = (void *) pxList;
39
40     (pxList->uxNumberOfItems)++;
41 }
```

Código 7 – Descrição de um Task Control Block

```
1  typedef struct tskTaskControlBlock {
2      volatile portSTACK_TYPE *pxTopOfStack; /*< Points to the location of the ↩
           last item placed on the tasks stack.  THIS MUST BE THE FIRST MEMBER OF ↩
           THE TCB STRUCT. */
3
4      xListItem xGenericListItem; /*< The list that the state list item of a task ↩
           is reference from denotes the state of that task (Ready, Blocked, ↩
           Suspended ). */
5      xListItem xEventListItem; /*< Used to reference a task from an event list. */
6      unsigned portBASE_TYPE uxPriority; /*< The priority of the task.  0 is the ↩
           lowest priority. */
7      portSTACK_TYPE *pxStack; /*< Points to the start of the stack. */
8      signed char pcTaskName[ configMAX_TASK_NAME_LEN ]; /*< Descriptive name ↩
           given to the task when created.  Facilitates debugging only. */
9
10 #if ( portSTACK_GROWTH > 0 )
11     portSTACK_TYPE *pxEndOfStack; /*< Points to the end of the stack on ↩
           architectures where the stack grows up from low memory. */
12 #endif
13
14 #if ( portCRITICAL_NESTING_IN_TCB == 1 )
15     unsigned portBASE_TYPE uxCriticalNesting; /*< Holds the critical section ↩
           nesting depth for ports that do not maintain their own count in the ↩
           port layer. */
16 #endif
17
18 } tskTCB;
```

Código 8 – Descrição de um Task Control Block

```
1 PRIVILEGED_DATA tskTCB * volatile pxCurrentTCB = NULL;
2 PRIVILEGED_DATA static xList pxReadyTasksLists[ configMAX_PRIORITIES ]; /*< ↩
      Prioritised ready tasks. */
3 PRIVILEGED_DATA static xList xDelayedTaskList1; /*< Delayed tasks. */
4 #if ( INCLUDE_vTaskSuspend == 1 )
5
6 PRIVILEGED_DATA static xList xSuspendedTaskList; /*< Tasks that are currently ↩
      suspended. */
7
8 #endif
```

taskSELECT_HIGHEST_PRIORITY_TASK()

A macro do Código 9 define a busca da task de maior prioridade disponível. Basicamente atua como uma interface para rapidamente buscá-la, pois faz uso de outras funções e macros: busca na lista de tasks disponíveis a serem executadas a que possui maior prioridade, então verifica se é uma resposta válida através da macro *configASSERT*, e por fim, a variável *pxCurrentTCB*, responsável por apontar a task em execução, recebe a próxima que deve ser executada.

xTaskCreate()

Para que tasks sejam criadas na memória, uma série de parâmetros são necessários, sendo alguns vistos na estrutura *Task Control Block*, como observado no Código 10: o ponteiro

Código 9 – Buscando a task de maior prioridade para ser executada

```
1 #define taskSELECT_HIGHEST_PRIORITY_TASK()                                    \
2 {                                                                             \
3     unsigned portBASE_TYPE uxTopPriority;                                     \
4     /* Find the highest priority queue that contains ready tasks. */          ↩
          \
5     portGET_HIGHEST_PRIORITY( uxTopPriority, uxTopReadyPriority );            ↩
          \
6     configASSERT( listCURRENT_LIST_LENGTH( &( pxReadyTasksLists[ uxTopPriority ↩
          ] ) ) > 0 );                                                          \
7     listGET_OWNER_OF_NEXT_ENTRY( pxCurrentTCB, &( pxReadyTasksLists[ ↩
          uxTopPriority ] ) );                                                  \
8 } /* taskSELECT_HIGHEST_PRIORITY_TASK() */
```

para a função da execução da task (*pxTaskCode*), o nome (*pcName*), o quanto de memória local vai precisar para seu *stack frame* (*usStackDepth*), os parâmetros exigidos para a execução da task(*pvParameters*), a prioridade (*uxPriority*) e possivelmente um manipulador (*handler*) para a task, utilizado em algumas funções do sistema. Outros parâmetros são utilizados somente a nível de kernel, como *puxStackBuffer*, cujo valor define exatamente aonde na memória o *stack frame* da task será alocado, mas caso seja *NULL* significa que será alocado na próxima área livre da *Heap*. Outro parâmetro de kernel é *xRegions*, utilizado por alguns microprocessadores, com propósito similar a *PRIVILEGED_DATA*. Caso seja *NULL* significa que a task será reservada na área cujo Sistema Operacional previamente reservou.

A implementação, devido aos padrões de programação e indentação do FreeRTOS (FREERTOS, 2013), se tornam bem legíveis e já se sabe o comportamento esperado das funções chamadas, mesmo que não se conheça o detalhamento do código. Sendo assim, primeiramente é verificado se a prioridade escolhida está no intervalo definido (por padrão, uma task pode ter prioridade de 0 a 3, mas este valor pode ser modificado, e quanto maior o valor, maior a prioridade. É importante destacar este detalhe, pois sistemas Linux possuem a filosofia oposta: quanto menor o valor, maior a prioridade (BOVET; CESATI, 2005)), então é alocado espaço na *Heap* para a task a ser criada.

Caso não haja memória suficiente para tal, é retornado um código de erro à aplicação. Do contrário, primeiramente é verificada posição do ponteiro de início do *stack frame* da task, dependendo da arquitetura do processador/controlador utilizado (em outras palavras, se é *big endian* ou *little endian*), então valores iniciais são inseridos na janela de pilha, e a partir deste ponto, o sistema entra em seção crítica (interrupções desabilitadas). Isso garante que o sistema não haja de forma inesperada, caso uma task esteja em criação, mas uma task entre em execução durante este processo.

Primeiramente, é incrementado o contador de tasks no sistema, e na sequência, é checado primeiramente se não havia outra task em execução, para que esta passe a ser processada. Caso o motivo seja porque o sistema está em inicialização, significa que a lista de tasks disponíveis deve ser criada. Uma última situação seria da task criada ser de

Código 10 – Criando uma task na memória

```
1 #define xTaskCreate xTaskGenericCreate
2
3 signed portBASE_TYPE xTaskGenericCreate(pdTASK_CODE pxTaskCode, const signed ↩
      char * const pcName, unsigned short usStackDepth, void *pvParameters, ↩
      unsigned portBASE_TYPE uxPriority, xTaskHandle *pxCreatedTask, ↩
      portSTACK_TYPE *puxStackBuffer, const xMemoryRegion * const xRegions) {
4     signed portBASE_TYPE xReturn;
5     tskTCB * pxNewTCB;
6
7     pxNewTCB = prvAllocateTCBAndStack(usStackDepth, puxStackBuffer);
8
9     if (pxNewTCB != NULL) {
10        portSTACK_TYPE *pxTopOfStack;
11
12        // Calculate the top of stack address.
13        pxTopOfStack = pxNewTCB->pxStack + (usStackDepth - (unsigned short) 1);
14        pxTopOfStack = (portSTACK_TYPE *) (((portPOINTER_SIZE_TYPE) ↩
              pxTopOfStack) & ((portPOINTER_SIZE_TYPE) ~portBYTE_ALIGNMENT_MASK));
15        prvInitialiseTCBVariables(pxNewTCB, pcName, uxPriority, xRegions, ↩
              usStackDepth);
16
17        pxNewTCB->pxTopOfStack = pxPortInitialiseStack(pxTopOfStack, ↩
              pxTaskCode, pvParameters);
18
19        if ((void *) pxCreatedTask != NULL) {
20            *pxCreatedTask = (xTaskHandle) pxNewTCB;
21        }
22
23 // Ensure interrupts don't access the task lists while they are being updated.
24        taskENTER_CRITICAL();
25        {
26            uxCurrentNumberOfTasks++;
27            if (pxCurrentTCB == NULL) {
28                pxCurrentTCB = pxNewTCB;
29                if (uxCurrentNumberOfTasks == (unsigned portBASE_TYPE) 1) {
30                    prvInitialiseTaskLists();
31                }
32            } else {
33                if (xSchedulerRunning == pdFALSE) {
34                    if (pxCurrentTCB->uxPriority <= uxPriority) {
35                        pxCurrentTCB = pxNewTCB;
36                    }
37                }
38            }
39            uxTaskNumber++;
40            prvAddTaskToReadyList(pxNewTCB);
41            xReturn = pdPASS;
42            portSETUP_TCB(pxNewTCB);
43        }
44        taskEXIT_CRITICAL();
45    } else {
46        xReturn = errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY;
47        traceTASK_CREATE_FAILED();
48    }
49    if (xReturn == pdPASS) {
50        if (xSchedulerRunning != pdFALSE) {
51            if (pxCurrentTCB->uxPriority < uxPriority) {
52                portYIELD_WITHIN_API();
53            }
54        }
55    }
56    return xReturn;
57 }
```

maior prioridade que a corrente, então o processamento deve ser passado à recém inserida. Então, já fora da região crítica, a task em execução deve ser atualizada.

vTaskDelayUntil()

O Código 11 é uma das funções que está na API de manipulação de tasks, sendo opcional seu uso. Simplificadamente, coloca uma task em estado Bloqueado até que um certo número de ticks do sistema passe. As trocas de contexto, como será vista mais adiante, pode ocorrer após um tempo determinado, ou forçosamente, caso a task corrente entre em estado Suspenso ou Bloqueado. Esta função se enquadra no segundo caso.

Vamos supor que a frequência de tick do sistema seja de 1 [KHz], ou seja, a cada milissegundo uma troca de contexto será chamada. Porém, como explicado, apesar de não haver passado esse tempo, uma próxima task pode ser chamada para que o processador não fique ocioso, mas para o sistema seria como se houvesse passado um tick. Isso significa que uma task chamar esta função para si com o parâmetro de 1 tick não fará efeito, pois assim que entrar em estado Bloqueado um tick será passado. Por outro lado, vamos supor que é a task de mais alta prioridade na memória, e que chama esta função para bloquear a si por 2 ticks. Significa que será re-executada daqui a 2 milissegundos.

Para que esta função funcione, deve estar em seção crítica, do contrário, os ticks continuarão a serem contados e o resultado desta função não corresponderá ao estado corrente do sistema.

Esta função checa quando foi a última vez que a task estava "acordada"(em modo Execução, de acordo com o parâmetro passado. Normalmente é passado o valor atual do tick) e checa se este é menor do que o tempo corrente. Se sim, passa para o estado Bloqueado. Na sequência, deve ser movida da lista de tasks disponíveis para as das que estão em *delay*, e então, é forçada uma troca de contexto.

vTaskStartScheduler()

Com as primeiras tasks já criadas na memória, resta criar uma task especial, chamada *Idle*, que é chamada somente quando todas as demais não estão disponíveis para serem executadas. Se houver sucesso, o sistema é informado de que o escalonador entrou em operação e o tick do sistema foi resetado, em outras palavras, o FreeRTOS inicia efetivamente sua execução aqui, com a chamada da função *xPortStartScheduler()*, que jamais retorna. A implementação pode ser vista no Código 12.

xTaskGetTickCount()

Esta função da API padrão simplesmente coleta, em seção crítica, o valor atual do tick do sistema e o retorna (Código 13).

Código 11 – Colocando uma task em estado Bloqueado por alguns ticks

```
#if ( INCLUDE_vTaskDelayUntil == 1 )

void vTaskDelayUntil(portTickType * const pxPreviousWakeTime, portTickType xTimeIncrement) {
    portTickType xTimeToWake;
    portBASE_TYPE xAlreadyYielded, xShouldDelay = pdFALSE;

    configASSERT(pxPreviousWakeTime);
    configASSERT((xTimeIncrement > 0U));

    vTaskSuspendAll();
    {
// Minor optimisation.  The tick count cannot change in this  block.
        const portTickType xConstTickCount = xTickCount;

// Generate the tick time at which the task wants to wake.
        xTimeToWake = *pxPreviousWakeTime + xTimeIncrement;

        if (xConstTickCount < *pxPreviousWakeTime) {
// The tick count has overflowed since this function was lasted called.
            if ((xTimeToWake < *pxPreviousWakeTime) && (xTimeToWake > xConstTickCount)) {
                xShouldDelay = pdTRUE;
            }
        } else {
// The tick time has not overflowed.  In this case we will delay if either
// the wake time has overflowed, and/or the tick time is less than the wake time.
            if ((xTimeToWake < *pxPreviousWakeTime) || (xTimeToWake > xConstTickCount)) {
                xShouldDelay = pdTRUE;
            }
        }

// Update the wake time ready for the next call.
        *pxPreviousWakeTime = xTimeToWake;

        if (xShouldDelay != pdFALSE) {
            traceTASK_DELAY_UNTIL();

// We must remove ourselves from the ready list before adding ourselves to the
// blocked list as the same list item is used for  both lists.
            if (uxListRemove(&(pxCurrentTCB->xGenericListItem)) == (unsigned portBASE_TYPE) 0) {
                portRESET_READY_PRIORITY(pxCurrentTCB->uxPriority, uxTopReadyPriority);
            }

            prvAddCurrentTaskToDelayedList(xTimeToWake);
        }
    }
    xAlreadyYielded = xTaskResumeAll();

    if (xAlreadyYielded == pdFALSE) {
        portYIELD_WITHIN_API();
    }
}

#endif /* INCLUDE_vTaskDelayUntil */
```

Código 12 – Iniciando o escalonador de processos do sistema

```
1   void vTaskStartScheduler(void) {
2     portBASE_TYPE xReturn;
3
4     /* Add the idle task at the lowest priority. */
5     /* Create the idle task without storing its handle. */
6     xReturn = xTaskCreate(prvIdleTask, (signed char *) "IDLE", ↩
          tskIDLE_STACK_SIZE, (void *) NULL, (tskIDLE_PRIORITY | ↩
          portPRIVILEGE_BIT), NULL); /*lint !e961 MISRA exception, justified as ↩
          it is not a redundant explicit cast to all supported compilers. */
7
8     if (xReturn == pdPASS) {
9         /* Interrupts are turned off here, to ensure a tick does not occur
10        before or during the call to xPortStartScheduler(). The stacks of
11        the created tasks contain a status word with interrupts switched on
12        so interrupts will automatically get re-enabled when the first task
13        starts to run.
14
15        STEPPING THROUGH HERE USING A DEBUGGER CAN CAUSE BIG PROBLEMS IF THE
16        DEBUGGER ALLOWS INTERRUPTS TO BE PROCESSED. */
17        portDISABLE_INTERRUPTS();
18
19        xSchedulerRunning = pdTRUE;
20        xTickCount = (portTickType) 0U;
21
22        /* Setting up the timer tick is hardware specific and thus in the
23        portable interface. */
24        if (xPortStartScheduler() != pdFALSE) {
25            /* Should not reach here as if the scheduler is running the
26            function will not return. */
27        } else {
28            /* Should only reach here if a task calls xTaskEndScheduler(). */
29        }
30    } else {
31        /* This line will only be reached if the kernel could not be started,
32        because there was not enough FreeRTOS heap to create the idle task
33        or the timer task. */
34        configASSERT(xReturn);
35    }
36 }
```

Código 13 – Recebendo o valor atual do tick do sistema

```
1 portTickType xTaskGetTickCount(void) {
2     portTickType xTicks;
3
4     /* Critical section required if running on a 16 bit processor. */
5     taskENTER_CRITICAL();
6     {
7         xTicks = xTickCount;
8     }
9     taskEXIT_CRITICAL();
10
11    return xTicks;
12 }
```

vTaskSuspendAll() e xTaskResumeAll()

Para que o escalonador seja colocado em modo suspenso é imediato: a instrução atômica de incremento da variável *uxSchedulerSuspended* é chamada. O escalonador de processos só entra em execução quando este valor for nulo, então quando o mesmo realizar esta auto-verificação, nenhuma task será posta em execução.

Por outro lado, para retirá-lo do modo suspenso é necessário se atentar a mais alguns detalhes: primeiramente, *uxSchedulerSuspended* deve ser decrementado. Então, adentra-se à região crítica, para que as operações na sequência sejam executadas sem interrupções. Enquanto o escalonador estava suspenso, pode ser que algumas tasks foram criadas ou se tornaram disponíveis para execução (mudaram do estado Suspenso ou Bloqueado para Disponível), então devem ser colocadas na lista de disponíveis, assim como as prioridades devem ser alinhadas. Depois disso, pode ser que alguns ticks tenham se passado, então a contagem deve ser atualizada e as tasks que não foram processadas devem ser executadas. Por fim, o escalonador é chamado e o sistema continua em execução.

Ambas as implementações podem ser vistas no Código 14.

vTaskSwitchContext()

Como visto no Código 3, a função de troca de contexto é chamada, e seu objetivo é primeiramente checar se o escalonador está suspenso. Caso contrário, verifica se não houve algum estouro de faixa dos *stack frames* (*Heap overflow*). Então, a task de maior prioridade disponível é buscada na lista, e o escalonador a coloca em execução.

Implementando tasks

As tasks utilizadas em uma aplicação FreeRTOS geralmente seguem o padrão descrito no Código 16, de modo que podem possuir qualquer nome, e podem chamar quaisquer funções que desejar. Neste caso, uma task possui uma implementação de nome *vTaskCode*, possui um nome *"NAME"*, um tamanho de *stack frame* igual a *STACK_SIZE*, pede os parâmetros *ucParametersToPass* (os quais serão interpretados pelo ponteiro para void *pvParameters*), possui uma prioridade *tskPriority* e contém o manipulador *xHandle*.

3.3 Instrumentação e Acomodação do Hardware

Para o hardware utilizado foram necessárias algumas adaptações para que os testes pudessem ser realizados. Com isso, dois pinos do microcontrolador foram soldados a dois fios, e os mesmos representariam a chamada de troca de contexto e a quantidade de vezes que o sistema entraria e sairia do modo *Idle*. O circuito pode ser visto na Figura 10.

Para que o hardware respondesse aos processamentos do software, a função *vPowerUpLed()* mostrada no Código 17 foi criada e colocada no arquivo *port.c*, de modo que um

Código 14 – Colocando e retirando o escalonador do modo Suspenso

```
1  void vTaskSuspendAll(void) {
2      ++uxSchedulerSuspended;
3  }
4
5  signed portBASE_TYPE xTaskResumeAll(void) {
6      tskTCB *pxTCB;
7      portBASE_TYPE xAlreadyYielded = pdFALSE;
8      portBASE_TYPE xYieldRequired = pdFALSE;
9
10     taskENTER_CRITICAL();
11     {
12         --uxSchedulerSuspended;
13         if (uxSchedulerSuspended == (unsigned portBASE_TYPE) pdFALSE) {
14             if (uxCurrentNumberOfTasks > (unsigned portBASE_TYPE) 0U) {
15                 /* Move any readied tasks from the pending list into the
16                 appropriate ready list. */
17                 while (listLIST_IS_EMPTY(&xPendingReadyList) == pdFALSE) {
18                     pxTCB = (tskTCB *) ↩
                           listGET_OWNER_OF_HEAD_ENTRY((&xPendingReadyList));
19                     (void) uxListRemove(&(pxTCB->xEventListItem));
20                     (void) uxListRemove(&(pxTCB->xGenericListItem));
21                     prvAddTaskToReadyList(pxTCB);
22
23                     /* If we have moved a task that has a priority higher than
24                     the current task then we should yield. */
25                     if (pxTCB->uxPriority >= pxCurrentTCB->uxPriority) {
26                         xYieldRequired = pdTRUE;
27                     }
28                 }
29                 /* If any ticks occurred while the scheduler was suspended then
30                 they should be processed.  This ensures the tick count doesn't
31                 slip, and any delayed tasks are resumed at the correct time. */
32                 if (uxPendedTicks > (unsigned portBASE_TYPE) 0U) {
33                     while (uxPendedTicks > (unsigned portBASE_TYPE) 0U) {
34                         if (xTaskIncrementTick() != pdFALSE) {
35                             xYieldRequired = pdTRUE;
36                         }
37                         --uxPendedTicks;
38                     }
39                 }
40                 if ((xYieldRequired == pdTRUE) || (xYieldPending == pdTRUE)) {
41                     xAlreadyYielded = pdTRUE;
42                     xYieldPending = pdFALSE;
43                     portYIELD_WITHIN_API();
44                 }
45             }
46         }
47     }
48     taskEXIT_CRITICAL();
49
50     return xAlreadyYielded;
51 }
```

Código 15 – Colocando e retirando o escalonador do modo Suspenso

```
1  void vTaskSwitchContext(void) {
2      if (uxSchedulerSuspended != (unsigned portBASE_TYPE) pdFALSE) {
3          /* The scheduler is currently suspended - do not allow a context
4          switch. */
5          xYieldPending = pdTRUE;
6      } else {
7          traceTASK_SWITCHED_OUT();
8
9          taskFIRST_CHECK_FOR_STACK_OVERFLOW();
10         taskSECOND_CHECK_FOR_STACK_OVERFLOW();
11
12         taskSELECT_HIGHEST_PRIORITY_TASK();
13
14         traceTASK_SWITCHED_IN();
15     }
16 }
```

Código 16 – Modelo padrão de implementação de tasks

```
1  /* Task to be created. */
2  void vTaskCode(void * pvParameters) {
3      for (;;) {
4          /* Task code goes here. */
5      }
6  }
7
8  /* Function that creates a task. */
9  void vOtherFunction(void) {
10     static unsigned char ucParameterToPass;
11     xTaskHandle xHandle;
12
13     /* Create the task, storing the handle.  Note that the passed parameter
14     ucParameterToPass must exist for the lifetime of the task, so in this
15     case is declared static.  If it was just an an automatic stack variable
16     it might no longer exist, or at least have been corrupted, by the time
17     the new task attempts to access it. */
18     xTaskCreate(vTaskCode, "NAME", STACK_SIZE, &ucParameterToPass, tskPriority,
19             &xHandle);
20
21 }
```

nível lógico alto seria inserido na porta configurada. Para os testes feitos, pinos 2 e 3 do microcontrolador MSP430F5172 foram utilizados. A simulação pode ser feita colocando-se LED's conectados a estes pinos, porém, para realizar as leituras sem prejudicar nos gastos energéticos do hardware no momento dos ensaios, foram conectados à pontas de prova de um osciloscópio.

3.4 CSB: Context Switch Bitmap

Uma solução para realizar a contagem de quantas vezes o FreeRTOS entra e sai do modo *Idle* pode ser feito utilizando um vetor binário (definido como *Bitmap* (SILBERSCHATZ; GALVIN; GAGNE, 2013)) no qual a cada chamada à interrupção do tick - a qual, indicará uma troca de contexto, seja ela por tempo expirado ou forçado porque a task atual saiu

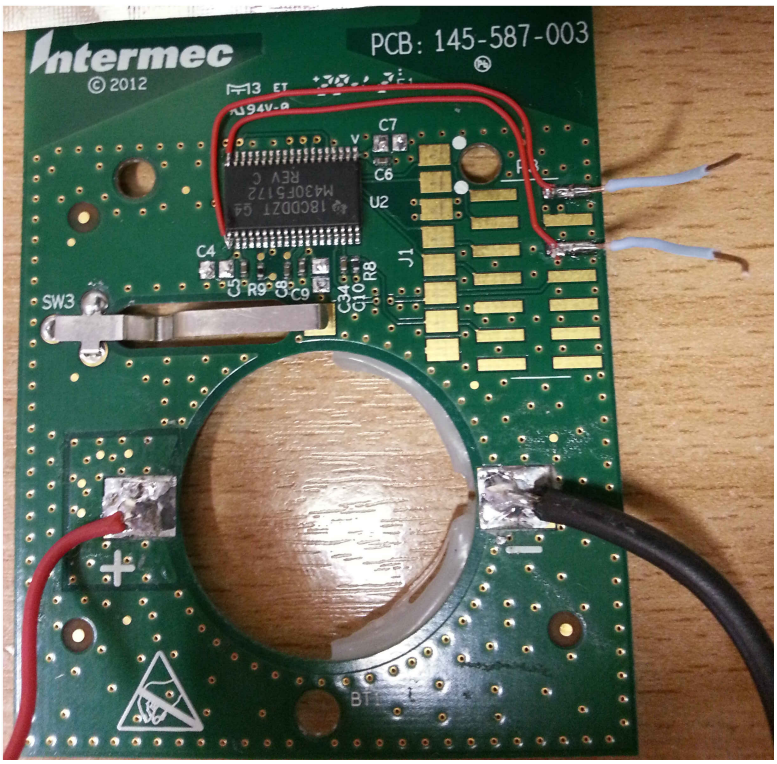

Figura 10 – Circuito proprietário utilizado para os testes e implementação do sistema

Código 17 – Código utilizado para depuração do sistema

```
1 void vPowerUpLed(portBASE_TYPE arg, portBASE_TYPE bit) {
2     if (arg == pdTRUE) {
3         P2OUT |= 1 << bit;
4     } else // arg == pdFALSE
5     {
6         P2OUT &= ~(1 << bit);
7     }
8 }
```

do estado Execução - é inserido um valor '1' se entrou em *Idle*, ou '0' caso contrário.

De acordo com a contagem de bits, um determinado processamento será feito. Para este propósito, o Algoritmo de Contagem Paralela de Bits (BIT..., 2005) visto no Código 18 foi utilizado, para que houvesse o mínimo de *overhead* na tarefa desejada. Com o valor contado, pode-se decidir qual modo de economia de energia é o mais viável para aquele estado do sistema.

3.5 Troca de Contexto

A troca de contexto no sistema FreeRTOS, como vimos, é uma sequência de eventos: ocorre uma interrupção pelo timer utilizado para o tick do sistema, a função *vTaskSwitchContext()* é chamada, e por fim, os registros são modificados para que a task de maior prioridade disponível possa ser processada. A primeira modificação é na rotina de trata-

Código 18 – Algoritmo de Contagem Paralela de Bits

```
1     /*   Counting bits set, in parallel:
2      *   http://graphics.stanford.edu/~seander/bithacks.html
3      *
4      *   Adapted for 16-bit counting.
5      */
6     unsigned portSHORT v = usContextSwitchBitmap; // count bits set in this ↩
          (16-bit value)
7     unsigned portSHORT c; // store the total here
8
9     c = v - ( ( v >> 1 ) & 0x5555 );
10    c = ( ( c >> 2 ) & 0x3333 ) + ( c & 0x3333 );
11    c = ( ( c >> 4 ) + c ) & 0x0F0F;
12    c = ( ( c >> 8 ) + c ) & 0x00FF;
```

Código 19 – Rotina de Tratamento de Interrupção modificada do tick do sistema

```
1  vPortTickISR:
2  /* The sr is not saved in portSAVE_CONTEXT() because vPortYield() needs
3  to save it manually before it gets modified (interrupts get disabled).
4  Entering through this interrupt means the SR is already on the stack, but
5  this keeps the stack frames identical. */
6    push.w sr
7    portSAVE_CONTEXT
8    calla #xTaskIncrementTick
9
10 /*********************************************************************
11  ***************** MODIFIED FOR POWER SAVING PURPOSES *****************
12  *********************************************************************/
13 //  cmp.w   #0x0, R12
14 //        jeq      SkipContextSwitch
15
16   calla #vTaskSwitchContext
17 SkipContextSwitch:
18         portRESTORE_CONTEXT
```

mento de interrupção do tick, a qual uma otimização foi removida com o intuito de realizar a contagem correta de quantas vezes *Idle* é chamada: se a task corrente não terminou de ser processada e a task corrente pode continuar em execução, então seu contexto deve ser restaurado para que continue com seu processamento, sem chamar a função de troca de contexto. A modificação pode ser vista no Código 19.

As seguintes modificações foram feitas no arquivo *tasks.c*, o qual trata da criação e manipulação de tasks no sistema. Para este projeto, a função *vTaskDelayUntil()* foi utilizada para testar trocas de contexto forçadas (quando o tempo de tick não expirou), de modo que foi necessário o auxílio da variável *flagNotTick*, um booleano responsável por informar se uma task retornou a ela mesma após um tick. Um trecho de implementação pode ser visto em Código 20, somente com os trechos inseridos sendo exibidos.

Com isso, a função propriamente dita de troca de contexto pode ser corretamente modificada, de modo a tirar máximo proveito do *Context Switch Bitmap*. Para facilitar as notações, o índice '2' se refere ao pino que é acionado/desativado quando o processo Idle é chamado, enquanto que o índice '3' se refere ao pino que é ligado/desligado quando

Código 20 – Modificação da função vTaskDelayUntil()

```
1 void vTaskDelayUntil(portTickType * const pxPreviousWakeTime, portTickType ↩
      xTimeIncrement) {
2     /* The whole code wasn't modified, just the very end, which is shown below. ↩
          */
3
4     /* Force a reschedule if xTaskResumeAll has not already done so, we may
5     have put ourselves to sleep. */
6     if (xAlreadyYielded == pdFALSE) {
7         /*****************************************************
8          ******** MODIFIED FOR POWER SAVING PURPOSES ********
9          *****************************************************/
10        flagNotTick = pdTRUE;
11
12        portYIELD_WITHIN_API();
13    }
14 }
```

Código 21 – Modificação da função vTaskSwitchContext()

```
1  void vTaskSwitchContext(void) {
2      /*********************************************************************
3       ***************** MODIFIED FOR POWER SAVING PURPOSES *****************
4       *********************************************************************/
5      // debug here
6      vPowerUpLed(pdFALSE, 2);
7      vPowerUpLed(pdTRUE, 3);
8
9      /* The rest wasn't modified, so is the exact same code. Them some more
10     * more stuff is added to the end. */
11
12     /*********************************************************************
13      ***************** MODIFIED FOR POWER SAVING PURPOSES *****************
14      *********************************************************************/
15     if (flagNotTick) {
16         flagNotTick = pdFALSE;
17     } else {
18         usContextSwitchBitmap <<= 1;
19         if (pxCurrentTCB->uxPriority == tskIDLE_PRIORITY) {
20             usContextSwitchBitmap++;
21         }
22     }
23     vPowerUpLed(pdFALSE, 3);
24
25 }
```

uma troca de contexto está em andamento.

Ao entrar na função do Código 21, o pino 2 deve ser desabilitado e o pino 3 ligado. O restante da função é executada como deveria, sem quaisquer modificações. Assim que a troca de contexto for definida, é checado se um novo processo foi chamado ou se será retornada para a task em execução. Caso sim, um novo bit é inserido no *Context Switch Bitmap*, que pode ser 0 (uma nova task) ou 1 (Idle). Por fim, o pino 3 é desligado.

3.6 Idle Task

O Sistema Operacional FreeRTOS define uma task especial de prioridade mínima, que é chamada quando não há mais tasks possíveis de serem executadas. Com isso, tarefas como desalocação de memória, requisições que não necessitam de atenção especial em determinado sistema ou então economia de energia podem ser implementadas.

Sendo o último caso para este projeto, é necessário realizar a contagem de quantas vezes o sistema entrou em modo Idle, sendo uma medida indireta da carga do sistema. Em outras palavras, se poucas vezes o sistema entrou em Idle, significa que passa a maior parte em processamento, então atrasos para economizar energia podem prejudicar a resposta esperada no tempo predeterminado. Caso tenha uma grande quantidade de acessos a Idle, significa que a maior parte do tempo está em "descanso", então atrasos para que se possa economizar energia não devem prejudicar o andamento do processamento.

Com essas diretrizes, cria-se a implementação da task Idle, como visto no Código 22. Note que este é somente um exemplo, e as decisões baseadas na quantidade de Idle's e qual o melhor modo de economia de energia é uma decisão puramente tomada pelo projetista.

3.7 Low Power Mode - LPM

O fabricante (T.I., 2011) define diversos modos de economia de energia, os quais desabilitam alguns módulos internos e tornam a exigência da fonte de energia do sistema menor. Os módulos desativados são mostrados na Figura 11.

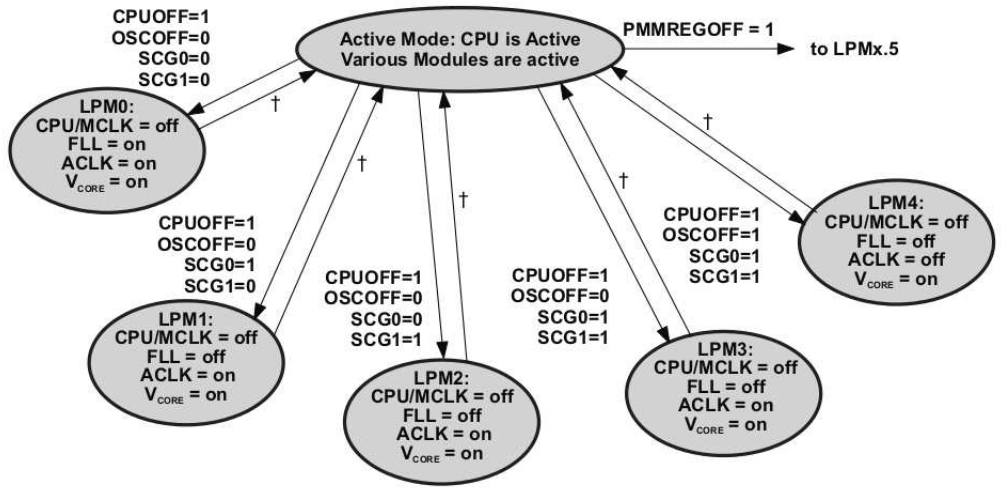

Figura 11 – Os diferentes Low Power Modes(Modos de Economia de Energia) do microcontrolador MSP430F5172

Para que seja economizada energia, como não são esperadas execuções, o núcleo de processamento(CPU) é desabilitado. Para os demais modos, são clocks internos, chamados de MCLK (*Master Clock*), SMCLK (*Sub-Master Clock*) e ACLK (*Asynchronous Clock*),

Código 22 – Implementação da task Idle

```
1  void vApplicationIdleHook( void )
2  {
3      unsigned portSHORT v = usContextSwitchBitmap;
4      unsigned portSHORT c;
5
6      // Parallel bit count: c = sum_ones(v)
7      c = v - ( ( v >> 1 ) & 0x5555 );
8      c = ( ( c >> 2 ) & 0x3333 ) + ( c & 0x3333 );
9      c = ( ( c >> 4 ) + c ) & 0x0F0F;
10     c = ( ( c >> 8 ) + c ) & 0x00FF;
11
12     // Turns On LPM LED debug
13     vPowerUpLed( pdTRUE, 2 );
14     switch( c )
15     {
16         // For a full-load system, not a good idea to go to any LPM mode
17         case 0:
18           break;
19         // For a high-load system, wake-up time can be a problem for RT
20         /* Fall through case. */
21         case 1:
22         case 2:
23         case 3:
24           __bis_SR_register( LPM1_bits + GIE );
25           break;
26
27         // Otherwise, it is worth saving energy
28         /* Fall through case. */
29         case 4:
30         case 5:
31         case 6:
32         case 7:
33         case 8:
34         case 9:
35         case 10:
36         case 11:
37         case 12:
38         case 13:
39         case 14:
40         case 15:
41         case 16:
42           __bis_SR_register( LPM3_bits + GIE );
43           break;
44
45         default:
46           /* Unexpected value. Keep active */
47           break;
48     }
49 }
```

Modo de Low-Power	**Tempo de Wake-up**
LPM0	≈ 0
LPM1	≈ 0
LPM2	$6.5\ \mu s$
LPM3	$6.5\ \mu s$
LPM4	$6.5\ \mu s$

Tabela 1 – Tempos de *wake-up* para os diferentes modos de baixo consumo, de acordo com o fabricante

sendo que este último é o de mais alta velocidade. Devido a isso, o Timer A neste projeto foi utilizado baseado em ACLK e é a interrupção que dispara o tick interno do FreeRTOS. Se for desabilitado, torna o sistema totalmente inoperante, tornando o LPM4 inviável.

LPM4.5 também não pode ser utilizado pois, além de desabilitar os mesmos módulos do LPM4, não mantém o estado da memória RAM interna e dos pinos de entrada/saída do circuito. Isso significa que, após uma interrupção externa acordar o sistema, nenhuma task estará disponível para execução, pois a memória pode ter sido apagada.

Agora restam os modos *Active* (Ativo, em execução), LPM0, LPM1, LPM2 e LPM3, os quais foram melhor estudados durante este projeto. Para que algumas decisões referentes a temporização sejam tomadas, o fenômeno de "acordar"um processador deve ser aprofundado.

O fato de desabilitar opções internas do MSP430 torna-o mais econômico, entretanto, quando uma requisição de interrupção for feita e atendida, significa que o microcontrolador deve mudar rapidamente para o modo Ativo, mas reconfigurar registros internos pode levar um certo tempo, o qual é chamado de *wake-up time*. O fabricante disponibiliza informações a respeito dos modos LPM2, LPM3 e LPM4, alegando que LPM0 e LPM1 são instantâneos, tal qual pode ser visto na Tabela 1.

4 Resultados

4.1 Previsão temporal da troca de contexto

Primeiramente, para que as medições não levem a conclusões errôneas, deve-se haver uma expectativa do que será visualizado. Quatro análises serão feitas: a troca de contextos na teoria, as transições do microcontrolador entre os modos de energia, a entrada para a task Idle (nos moldes da implementação deste projeto) e a chamada de troca de contexto propriamente dita (nos moldes de implementação deste projeto).

Troca de contexto real

De acordo com a teoria de Sistemas Operacionais, a troca de contexto é chamada após ou durante a execução de uma task (SILBERSCHATZ; GALVIN; GAGNE, 2013). Caso não hajam mais tarefas a serem feitas, o sistema se prepara para entrar em modo Idle (visto em vermelho na Figura 12 como *post-processing*), quando finalmente vai para um modo ocioso (Idle), ou seja, não é esperada a execução de tarefa alguma que produza resultados para o qual o sistema foi projetado.

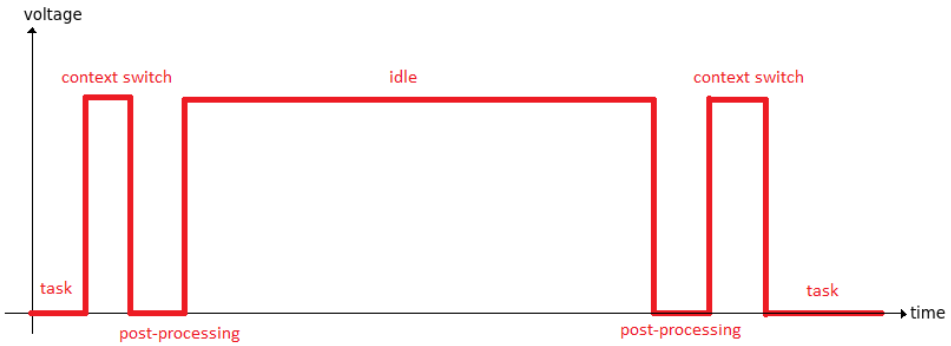

Figura 12 – As trocas de contexto segundo o ponto de vista do Sistema Operacional

Entrada e saída do modo de baixo consumo de energia

Implementando um Sistema Operacional de Tempo Real dentro de um microcontrolador, é esperado que uma economia de energia ocorra quando operar em modo ocioso (Idle), e há um certo atraso entre o pedido e a reconfiguração interna. Após certo tempo em modo de economia de energia, um evento (normalmente uma interrupção) o "acorda", e passado o

wake-up time, retorna ao modo ativo, antes do Sistema Operacional efetivamente transitar de seu modo ocioso ao modo de execução, como visto em verde na Figura 13.

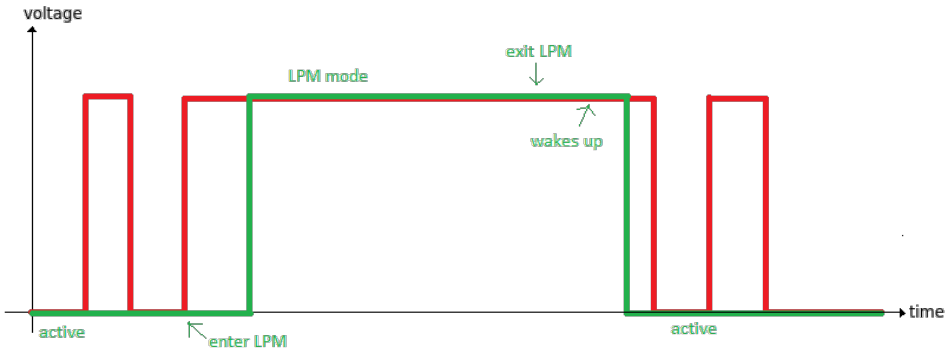

Figura 13 – Transições do microcontrolador entre os modos de energia

Utilização de saída para monitoramento do processo Idle

Utilizando as implementações mostradas no capítulo anterior, o pino de Idle é ativado durante a task Idle um pouco antes do microcontrolador entrar em modo de economia de energia, sendo desativado somente na função de troca de contexto propriamente dita, como é visto em amarelo na Figura 14. Isso dificulta a análise de *wake-up time*, pois o estado deste pino apresenta um instante que não representa com fidelidade nem o estado do Sistema Operacional, nem o do microcontrolador. Entretanto, ainda é possível realizar estimativas acerca dos gráficos gerados, pois os tempos devem estar na faixa determinada pelo fabricante.

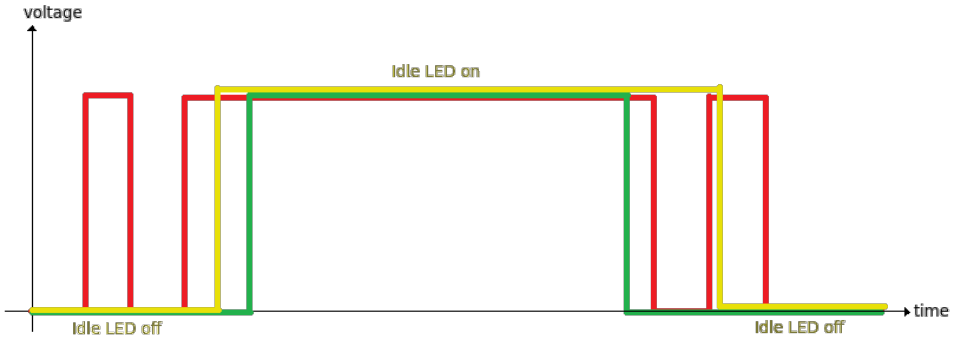

Figura 14 – Transições do microcontrolador entre os modos de energia

Utilização de saída para monitoramento da troca de contexto

Como já estudado, a função Troca de Contexto só ocorre após uma interrupção do timer de tick. Utilizando uma análise simplificada, este seria o momento no qual a troca de contexto se inicia. Entretanto, como há a execução da rotina de tratamento de interrupção

e também as chamadas das instruções de inicialização da sub-rotina *vTaskSwitchContext()*, o pino referente a esta análise é ativado com certo atraso, em relação ao ponto de vista do Sistema Operacional, como visto em azul na Figura 15.

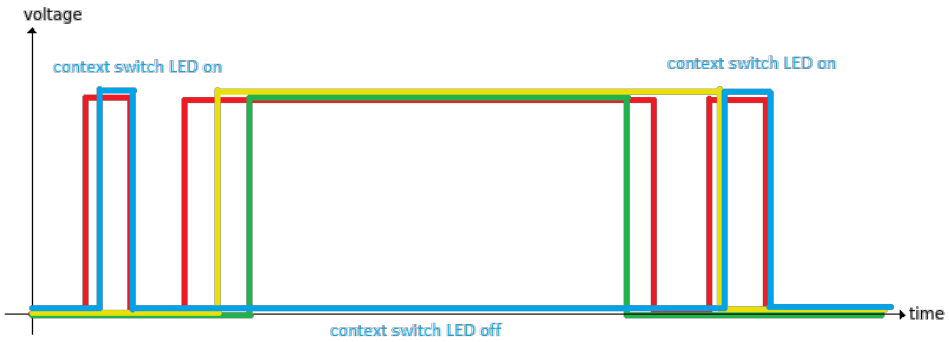

Figura 15 – Função Troca de Contexto

4.2 Atrasos referentes ao *Wake-up Time*

Reforçando as ilustrações anteriores, uma aproximação do que será visto nas medições dos pinos no osciloscópio pode ser visto na Figura 16.

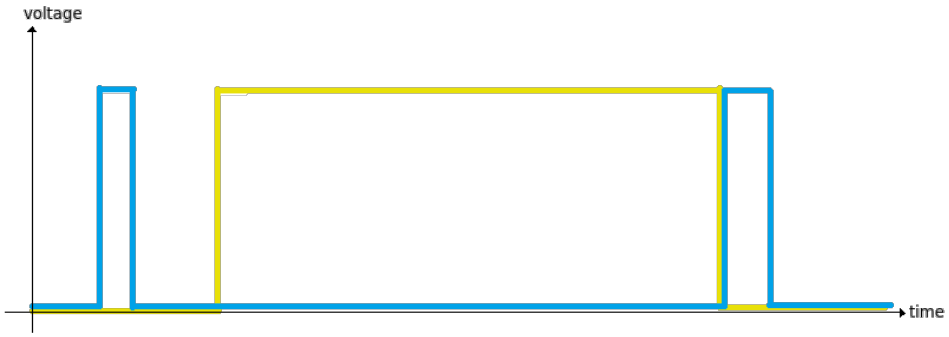

Figura 16 – Forma de onda esperada nas medições com osciloscópio

Já contextualizado do que pode ser analizado nas saídas dos pinos, foram testados diferentes *Low Power Modes*, e também o modo *Active*. Os mesmos devem estar em conformidade com a tabela apresentada ao final do capítulo anterior. As figuras a seguir mostram os resultados. Mantendo conformidade com as estimativas anteriores, as curvas em amarelo representam a entrada e saída do modo Idle, enquanto que as curvas em azul estão relacionadas com a função Troca de Contexto.

Estes testes foram realizados com uma troca de contexto na frequência de $32.767[Hz]$. Cada medição possui um valor absoluto determinado pelo osciloscópio, uma média aritmética, uma média dos valores mínimos, uma média dos valores máximos e o desvio padrão das

Modo de Low-Power	Tempo em modo Idle	Tempo de Wake-up
Active	17.76 μs	-
LPM0	17.79 μs	0.03 μs
LPM1	17.61 μs	-0.15 μs
LPM2	24.64 μs	6.88 μs
LPM3	24.44 μs	6.68 μs

Tabela 2 – Tempos de *wake-up* estimados através das medições no osciloscópio

medidas. A medição *+Largura* mede o eixo das abcissas (tempo) da curva em amarelo, enquanto que as medidas *Alta*, *Baixa* e *Média* são referentes à curva em azul, checando os níveis de tensão quando se está em nível lógico alto e em nível lógico baixo, além de checar a média aritmética dos valores coletados.

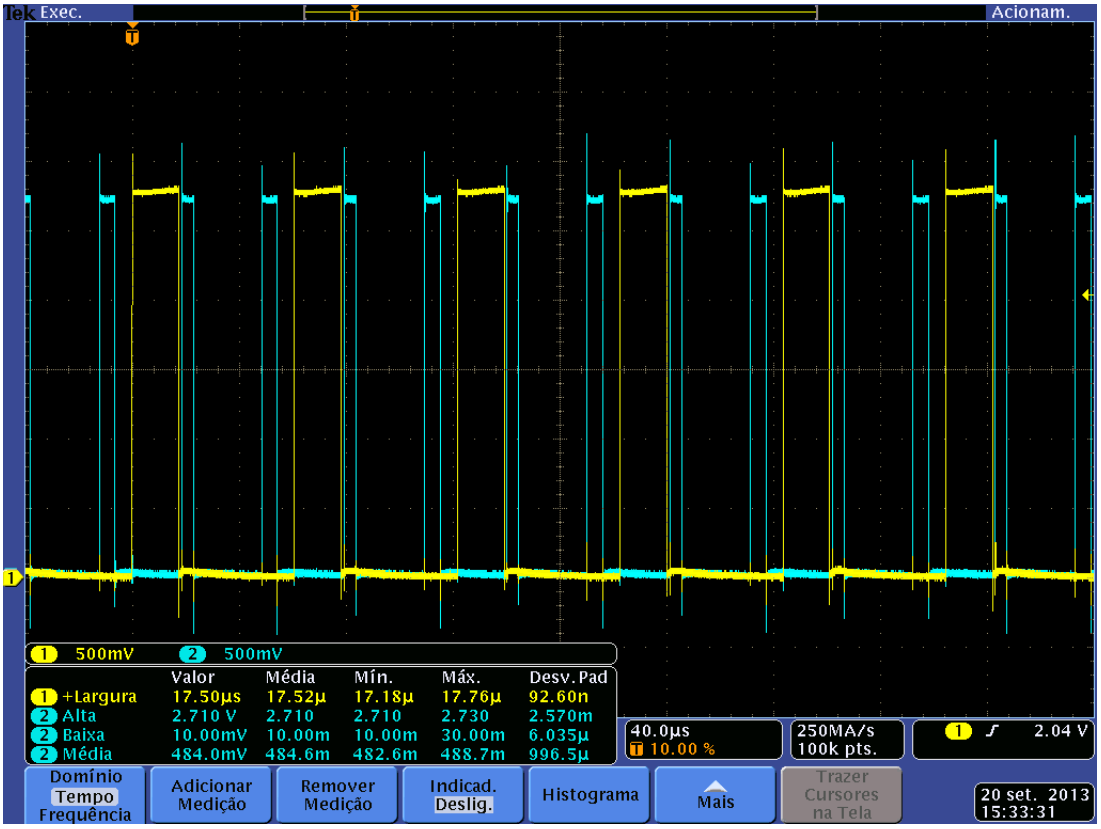

Figura 17 – Processo idle mantido em modo ativo

Sintetizando os resultados na Tabela 2:

O modo Ativo foi colocado na tabela somente como referência de tempo de execução. Note que os valores estão muito próximos dos estabelecidos pelo fabricante, e a precisão do osciloscópio utilizado foi suficiente para que os truncamentos dos valores não prejudicassem as análises. Isso indica que em casos de alta demanda de processamento, os modos LPM0 e LPM1 são ideais, e caso não sejam utilizados muitos dos módulos disponíveis pelo microcontrolador, LPM1 será a escolha ideal.

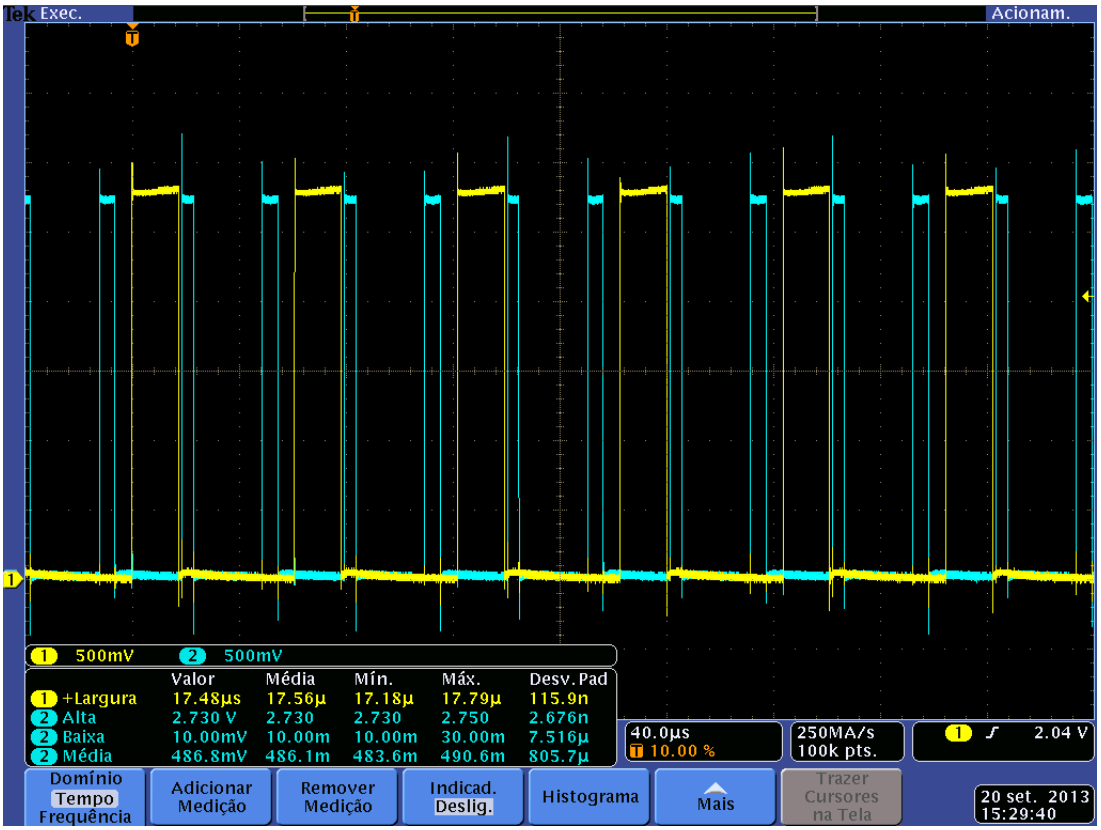

Figura 18 – Processo idle mantido em LPM0

Se em determinado momento a demanda da CPU não for alta, então a economia de energia se dará pelos modos LPM2 e LPM3, pois os atrasos devido ao *wake-up* não devem prejudicar o bom funcionamento do sistema. Analogamente à conclusão anterior, se não forem utilizados muitos dos módulos disponíveis pelo microcontrolador, LPM3 será a escolha ideal.

4.3 Análise do *Context Switch Bitmap*

De acordo com a implementação utilizada, a variável *usContextSwitchBitmap* norteará os resultados neste momento. Através de seus valores é que será determinado qual o melhor modo de economia de energia. Entretanto, os valores adquiridos podem se tornar confusos caso não esteja atento à **janela de tempo de execução** (*time window*).

4.3.1 Janela de tempo de execução

Supondo que uma determinada task A demora 1 tick e meio para ser executada, enquanto que uma task B demora aproximadamente 1 tick. Entretanto, a janela de tempo máxima reservada que uma task possui para ser executada é de 1 tick, isso significa que uma task que necessite de mais tempo deve ser executada quando uma outra poderia ser processada

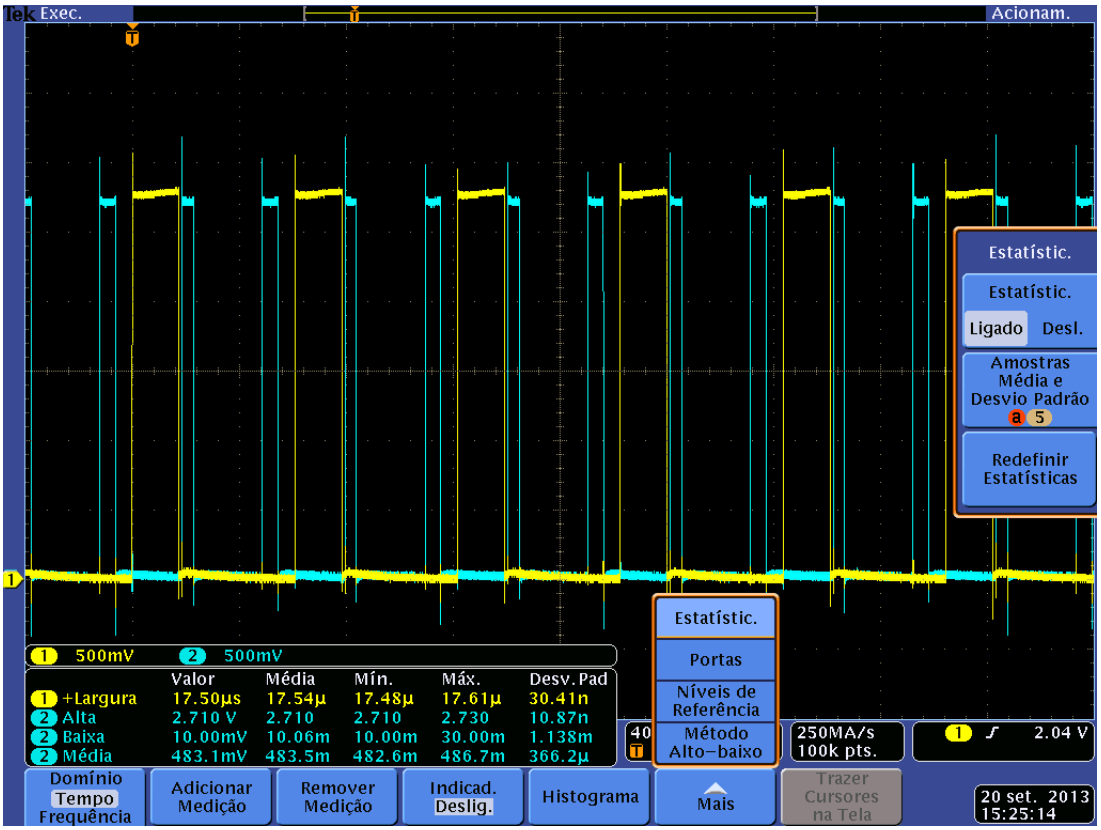

Figura 19 – Processo idle mantido em LPM1

completamente, ou seja, a janela de tempo da task A é maior do que o que foi reservado para si.

Isso significa que não há uma periodicidade simples na entrada e saída do modo Idle, mas sim composta (isso significa que não será algo como 001111 em *Context Switch Bitmap*, ou seja, uma sequência de zeros e depois uma de '1', sempre com os mesmos tamanhos, mas sim 00111100010111). Há meios de tornar a sequência de '0' e '1', o que pode influenciar em uma economia de energia mais efetiva, mas tal técnica será comentada mais adiante.

Um exemplo de situação pode ser visualizado pelas Figuras 22, 23 e 24.

Sendo que cada janela de tempo é seguida de uma chamada de troca de contexto, cada janela de tempo com uma task dentro simboliza um '0' à variável *Context Switch Bitmap*. Em outras palavras, em termos de implementação do FreeRTOS, cada janela de tempo é um tick. Para o caso da Figura 24, duas tasks consomem 3 ticks.

Supondo um caso no qual uma task C foi colocada em estado Bloqueado por 4 ticks, imediatamente antes da task A ser colocada para execução. Isso significa que, após o término da task B, C "acordará", e o sistema não entrará em Idle, como poderia ser esperado. Baseado nessa hipótese, três tasks foram colocadas para serem executadas, todas possuindo a mesma implementação, definida pelo Código 23.

A implementação é simples: capta o valor corrente do tick e recebe como parâmetro

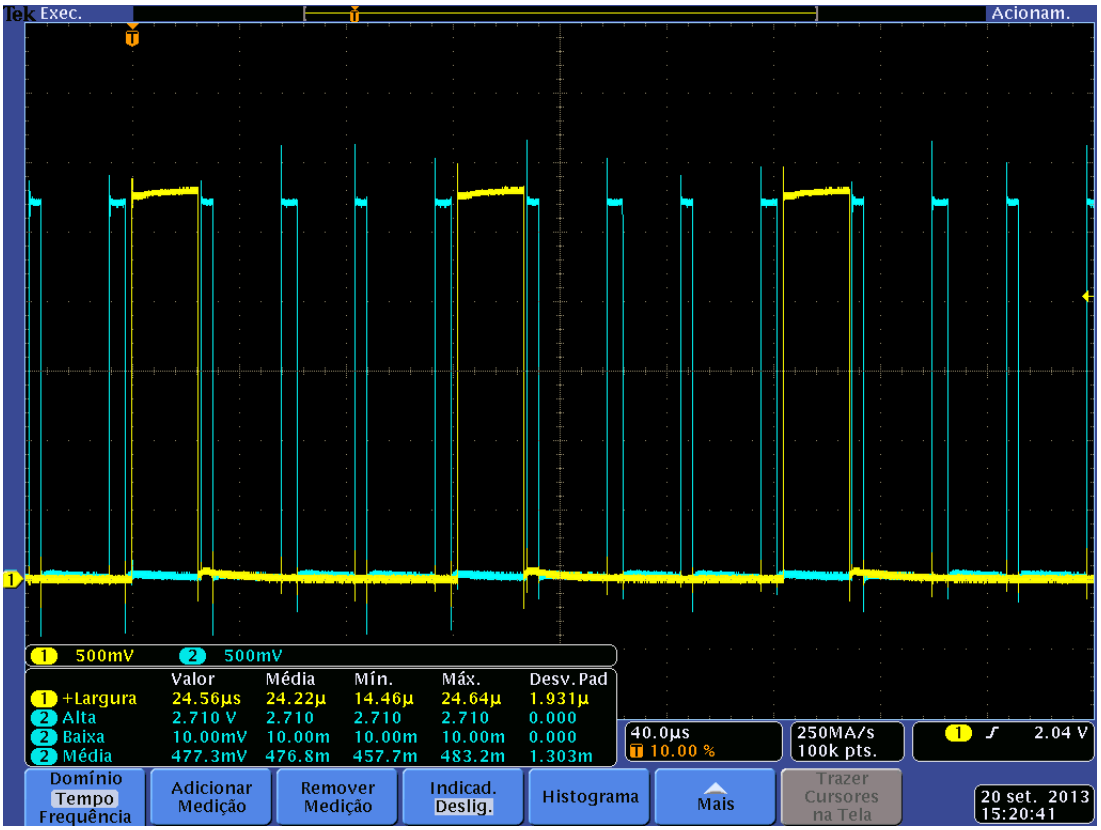

Figura 20 – Processo idle mantido em LPM2

Código 23 – Implementação-teste de tasks para análise da janela de tempo

```
1  void vTaskCode( void * pvParameters )
2  {
3      portTickType xLastWakeTime;
4      const portTickType xFrequency = ( portTickType ) pvParameters;
5
6      // Initialise the xLastWakeTime variable with the current time.
7      xLastWakeTime = xTaskGetTickCount();
8      int i = 0;
9
10     for( ;; )
11     {
12         for( i = 0; i < 100; i++);
13         if( xFrequency > 0 )
14         {
15             vTaskDelayUntil( &xLastWakeTime, xFrequency );
16         }
17     }
18 }
```

por quantos ticks deseja ser bloqueada, após finalizar sua execução. O processamento útil desta task se dá por conta de um loop que conta de 0 a 100.

Na execução do sistema, todas as tasks receberam o parâmetro '5', ou seja, cada task dorme por 5 ticks.

Para examinar a variável *Context Switch Bitmap*, em outras palavras, para facilitar a depuração dos testes obteve-se auxílio da IDE IAR Embedded Workbench (IAR...,

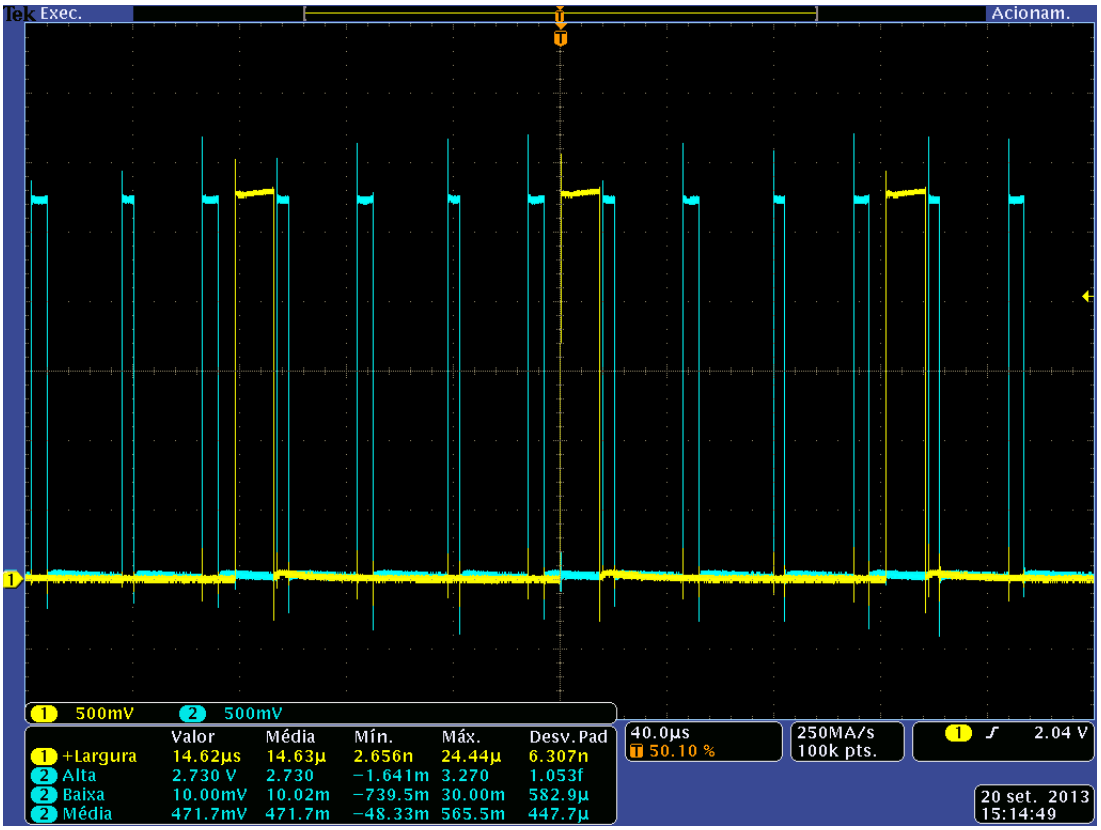

Figura 21 – Processo idle mantido em LPM3

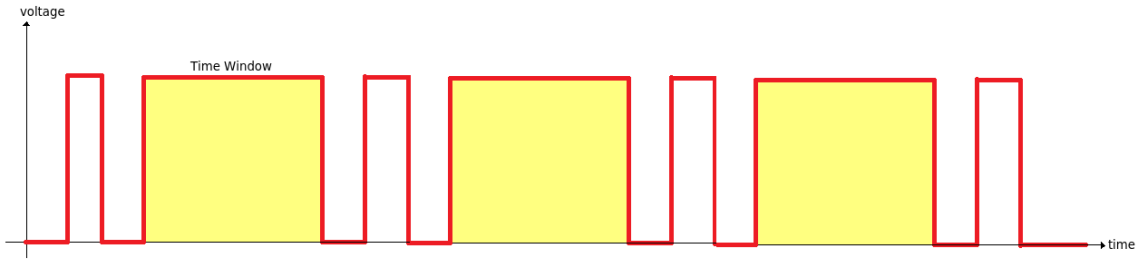

Figura 22 – Conceito de janela de tempo

2013), a qual exibe o estado dos registradores internos do controlador, a pilha da aplicação em execução, valor corrente das variáveis alocadas e também das tasks criadas no sistema FreeRTOS.

Checando *Context Switch Bitmap*, após certo tempo, seus valores foram coletados e colocados na Tabela 3:

Note que a razão Idle-tasks está em torno de 63%, o que significa que seu período deve conter este valor. Por outro lado, a periodicidade dele deve ser maior que 16 trocas de contexto, sendo provada pela oscilação da quantidade de zeros e uns. Então prova-se que a janela de tempo dada pelo tick deste sistema não é suficiente para que as tasks terminem seu processamento com certa regularidade.

A seguir, a Tabela 4 mostrando os menores e maiores valores de entradas no modo

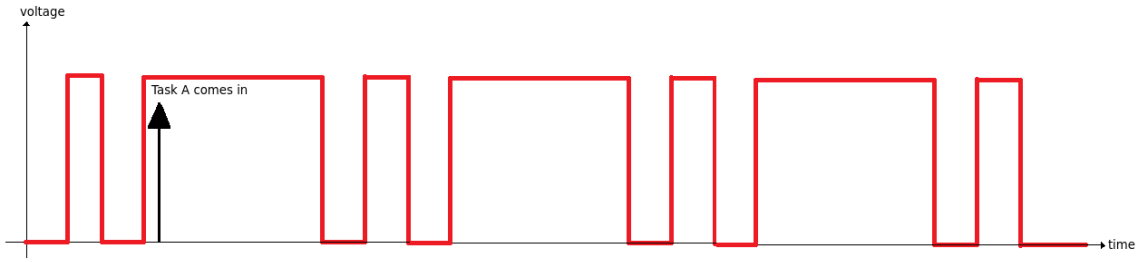

Figura 23 – Inicio da execução de uma task (A)

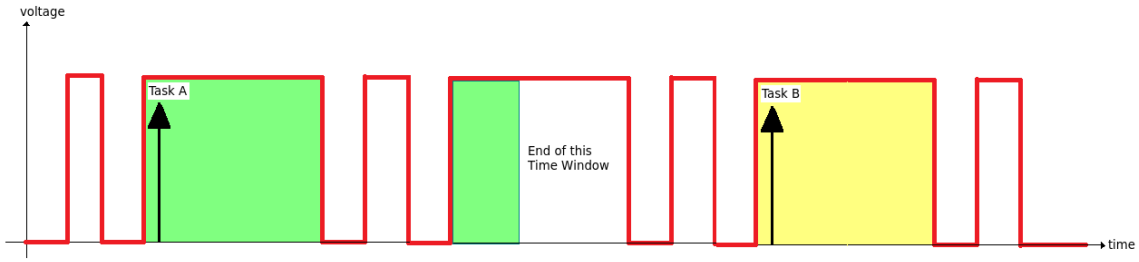

Figura 24 – Uma task (A) necessitando de mais de um tick para finalizar sua execução

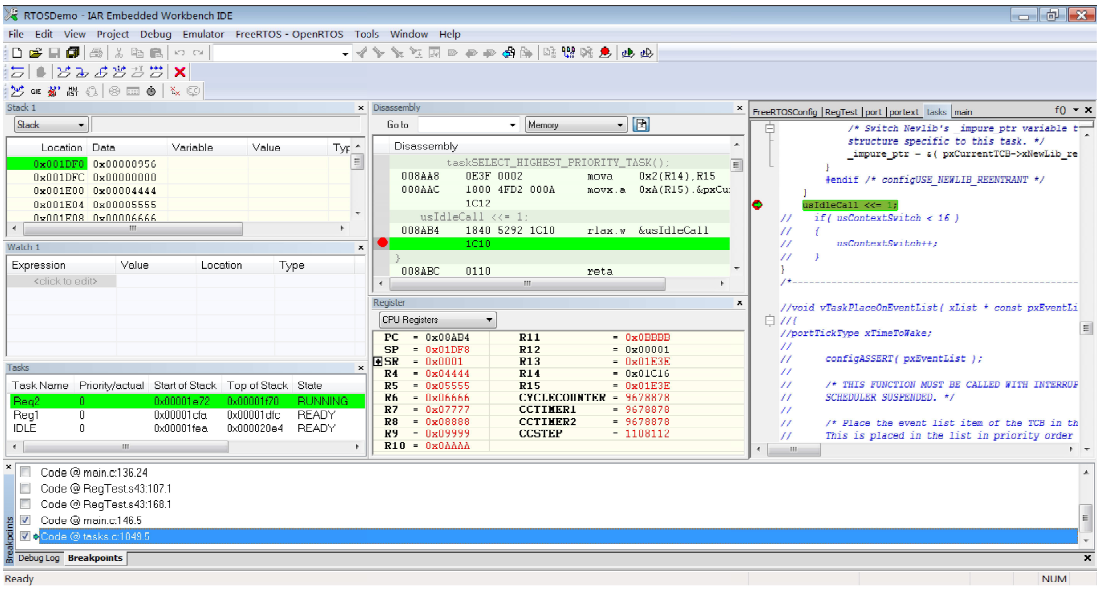

Figura 25 – Ambiente de desenvolvimento IAR Embedded Workbench

Idle:

Como existem oscilações na contagem de entradas no modo Idle, isso indica que o modo de economia de energia pode oscilar, o que é ideal, pois deve se ajustar dinamicamente de acordo com as necessidades de processamento do sistema.

Número do Teste	Bitmap	Razão Idle-Task	Porcentagem Idle-Task
1	0111111100000011	9/16	56%
2	1111000011111100	10/16	63%
3	0011111100001111	10/16	63%
4	0000111111000001	7/16	44%
5	1111000000111111	10/16	63%

Tabela 3 – Diferentes testes para a variável Context Switch Bitmap, analisando a razão Idle-Tasks

Teste	Bitmap	Razão Idle-Task	Porcentagem Idle-Task
Pior caso	0000001111110000	6/16	38%
Melhor caso	1111111000011111	12/16	75%

Tabela 4 – O melhor e o pior caso medidos através do Context Switch Bitmap

4.4 Economia de Energia

Já foram analisadas as questões temporais e detalhes relativos à decisão de qual o melhor modo de economia de energia. Entretanto, resta saber se há uma economia propriamente dita. Os gráficos das Figuras 26, 27 e 28 mostram um sistema sendo ligado e executado, considerando diferentes modos de energia: Ativo, LPM1 e LPM3. Estes testes foram feitos com uma fonte de 3[V] e medição da corrente requisitada pela fonte, e o tick usado no sistema foi 1[kHz].

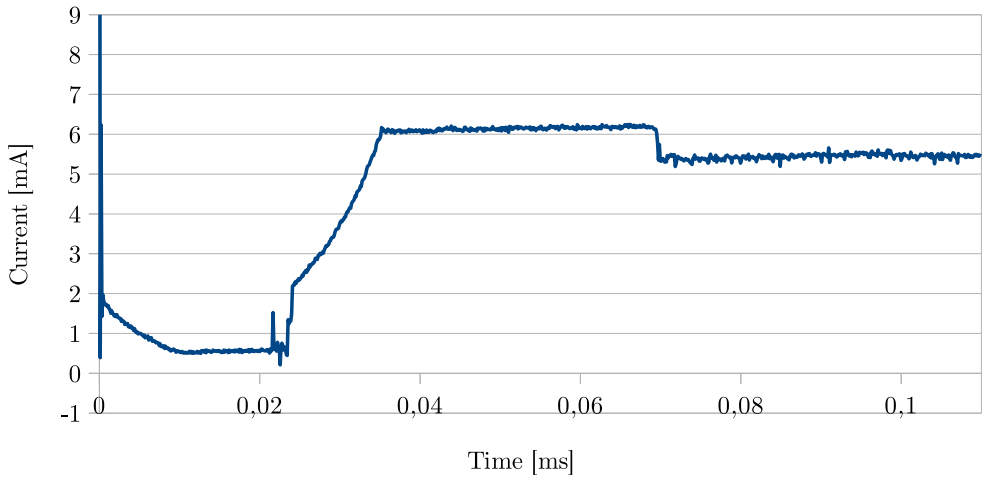

Figura 26 – Consumo de Corrente do sistema em modo Ativo

Não é possível realizar uma completa visualização do consumo de corrente devido à escala utilizada. Entretanto, é possível notar um consumo não esperado no início das medições: se trata do *Power On Reset - POR*. De acordo com a documentação do fabricante, "no momento em que o microcontrolador é ligado, uma rampa ocorre, gerando um

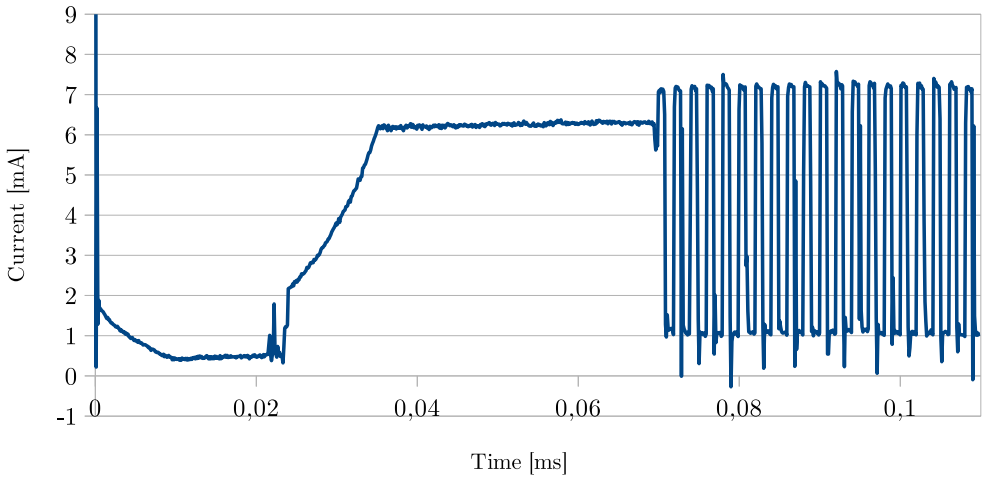

Figura 27 – Consumo de Corrente do sistema em LPM1 quando em Idle

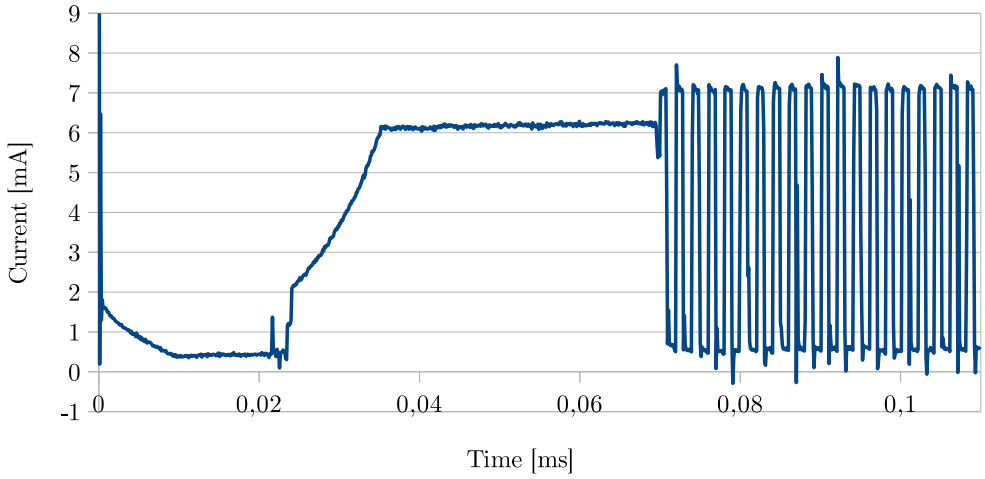

Figura 28 – Consumo de Corrente do sistema em LPM3 quando em Idle

processo de *POR* que o inicializa" (T.I., 2011). Logo após, o processo de POR continua, até que o código assemblado no microcontrolador começa a ser executado, podendo ser visualizado como uma leve queda no nível de corrente (ou oscilações, que representam as transições de estado Ativo e algum LPMx).

Para esclarecer as diferenças de consumo de corrente, o processo de POR foi removido e destacado somente a execução do sistema, sendo visualizados nas Figuras 29, 30 e 31.

Desta vez é possível visualizar que há, efetivamente, economia de energia ao chamar um dos modos LPM (representados pelos degraus nos gráficos). Observando cautelosamente, nota-se que há uma diferença nos níveis de corrente entre os modos LPM1 e LPM3, os quais podem ser melhor visualizados através da Tabela 5. O valor médio foi determinado utilizando-se os valores das Figuras 29, 30 e 31 contidos no intervalo entre a média absoluta

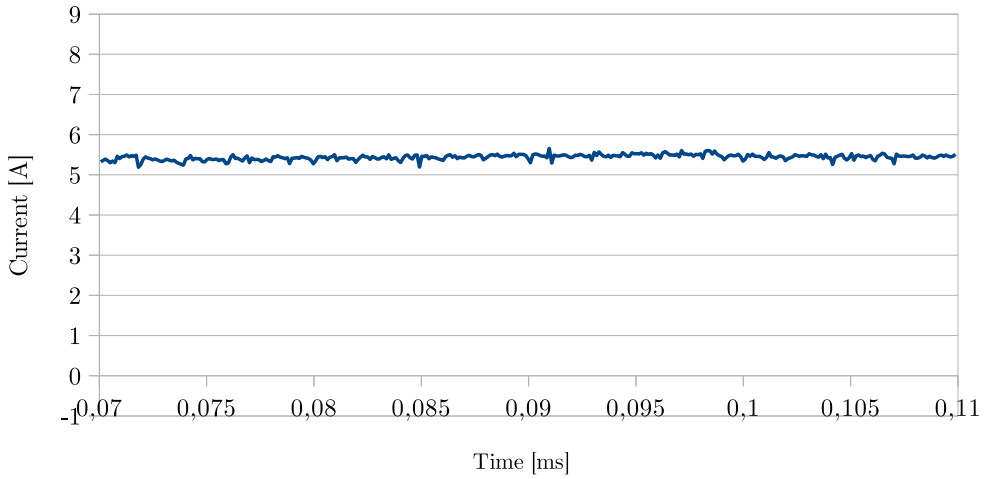

Figura 29 – Consumo de Corrente (durante execução) do sistema em modo Ativo

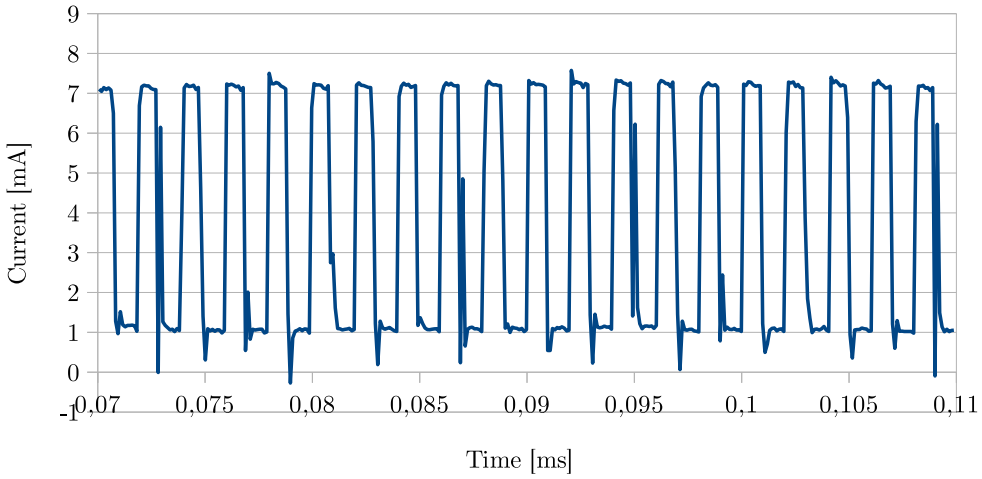

Figura 30 – Consumo de Corrente (durante execução) do sistema em LPM1 quando em Idle

e o valor mínimo (então uma nova média é calculada e inserida na tabela), enquanto que o campo "Economia"é a subtração entre o consumo médio do modo da linha anterior e do modo em análise:

Visualmente, percebe-se que LPM3 está abaixo de $1000\mu A$, enquanto que LPM1 está no entorno deste valor. Analisando os valores, há a influência da corrente consumida no modo Ativo, entretanto, a economia causada pelo LPM3 se torna óbvia ao observar os consumos médios de corrente.

Checando os valores dados pelo fabricante, uma grande dificuldade existente é o quão próximos da realidade um valor de corrente, energia ou potência fornecidos por um fabricante estão. De acordo com um artigo feita pela Renesas (INC., 2013), os dados fornecidos pelo fabricante foram determinados executando uma intrução "NOP", um loop infinito

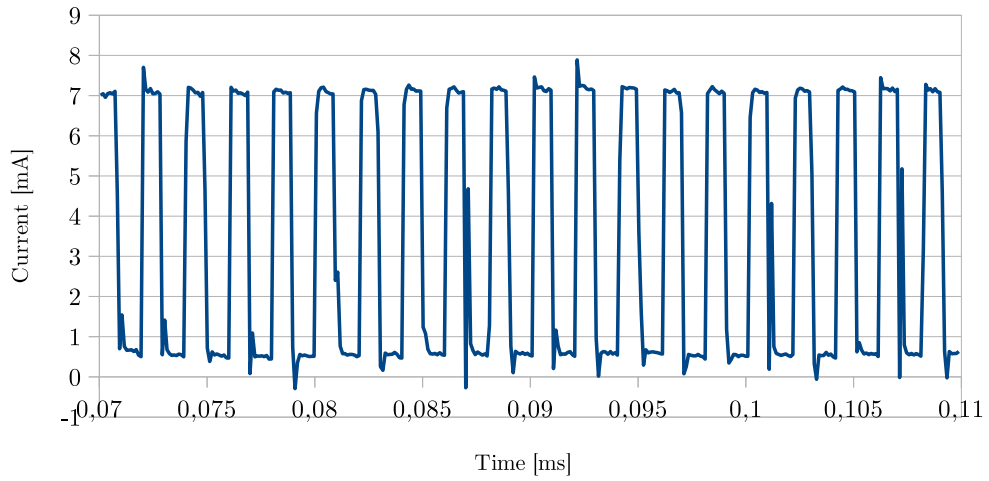

Figura 31 – Consumo de Corrente (durante execução) do sistema em LPM3 quando em Idle

Modo de Energia	**Fabricante**	**Consumo Médio Medido**	**Economia**
Active	3.64 *a* 6.15 mA	5382 μA	-
LPM1	85 *a* 104 μA	1076 μA	4305 μA
LPM3	1.2 *a* 3.0 μA	597.8 μA	478.8 μA

Tabela 5 – Comparação entre os diferentes valores de corrente nos modos do microcontrolador em contraste com o fornecido pelo datasheet do fabricante

Modo de Energia	**Processo de POR**	**Retorno ao Modo Ativo**
Active	6.133 mA	5.487 mA
LPM1		7.035 mA
LPM3		6.957 mA

Tabela 6 – Comparação entre as correntes consumidas no modo ativo e no processo de Power On Reset

ou um algoritmo específico? Além disso, quais periféricos estão ligados e quais estão desligados? Deste modo, o que é fornecido em um datasheet representa uma referência, mas não uma realidade absoluta para todo e qualquer projeto. Com tal explanação, como há dois pinos habilitados para realizar a leitura no osciloscópio, o importante é que os valores lidos estejam na mesma ordem de grandeza do que é fornecido pelo fabricante, sendo então medidas eficazes e suficientes para que as conclusões anteriormente tiradas sejam validadas.

Por outro lado, nota-se que há um degrau entre a saída de um modo de economia de energia e o modo ativo. Comparando-se estes valores com o utilizado no processo de POR:

Colocando-se na Tabela 6 as médias determinadas para o Modo Ativo em cada ensaio e o adquirido com a Figura 32:

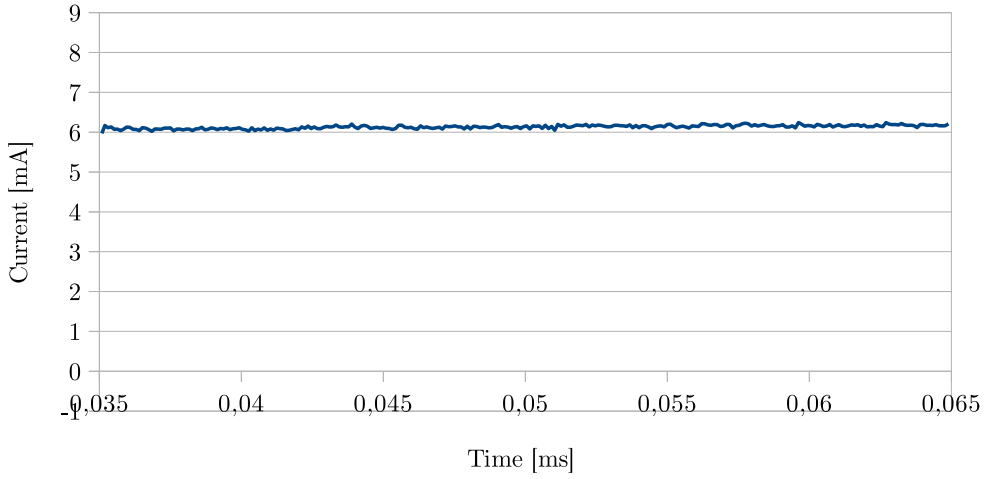

Figura 32 – Consumo de Corrente durante o processo de Power On Reset do microcontrolador

Um microcontrolador sendo executado continuamente no modo ativo necessita de menos corrente quando comparado em seu processo de inicialização. Por outro lado, a partir do momento em que "acorda"de um modo de baixo consumo, é necessário reativar seus periféricos e configurar registros internos, elevando suas necessidades de corrente.

O fabricante não fornece os valores de corrente durante o processo de POR.

5 Conclusão

Um Sistema Operacional robusto como o FreeRTOS está há mais de 6 anos disponível à comunidade, o que implica em uma completa e extensa documentação, tornando-o modular e flexível para diferentes projetos. Uma das características que facilita a fácil compreensão é a padronização na documentação, especificando normas aconselhadas para nomear variáveis e definir tipos para certas situações.

Além do mais, diversos exemplos e demonstrações também se mostram disponíveis, facilitando a abstração do Sistema Operacional em relação à codificação do driver de um hardware, ou mesmo apontando quais grupos de código devem ser modificados para se buscar determinada otimização. Uma destas sugestões está na tarefa ociosa do sistema (definido pelo nome *Idle Task*), de modo que foi o passo inicial para este projeto.

Realizando modificações no kernel do sistema, focado na implementação do escalonador, uma técnica de dinâmica de economia de energia foi criada e testada, de modo a definir um escalonador de processos dinâmico voltado à economia de energia. Seu princípio se baseia na contagem de quantas vezes o sistema se torna ocioso (sendo um histórico dos últimos processos executados), e este valor determina o quanto é possível economizar energia no sistema.

Utilizando-se do microcontrolador MSP430F5172 como base, viu-se que não é válido analisar os modos de economia de energia de um hardware somente pelos valores de corrente exigidos, mas um outro fator importante denominado *wake-up time*, que é o tempo levado para sair de um modo de baixo consumo para o modo de execução (para este microcontrolador, chamados respectivamente de *Low Power Mode* e *Active Mode*).

Através dos testes realizados, uma estrutura de vetor binário (*bitmap*) foi definida (*Context Switch Bitmap*) para realizar a contagem de modo simples e eficiente de quantas vezes o sistema se tornou ocioso. A partir deste valor e com base nos *wake-up times*, o melhor modo de economia de energia pode ser selecionado dinamicamente.

Porém, não é uma técnica idealmente eficiente para todo e qualquer sistema, sendo necessário levantar os dados de consumo de corrente do microprocessador/microcontrolador utilizado, visto que gastos durante o modo ativo, quando chaveados de um modo de baixo consumo, são comparáveis com o necessário para inicializar o sistema (processo conhecido como *Power-On Reset*), que geralmente é mais elevado por conta de estar configurando periféricos e registros internos da CPU.

Importante salientar que esta técnica desenvolvida pode ser extendida para quaisquer Sistemas Operacionais que sejam portados em hardwares que possuam modos de economia de energia, seja ela uma arquitetura voltada à sistemas *Desktop*, *Mobile* ou outras

aplicações computacionais ou embarcadas.

5.1 Trabalhos Futuros

Um dos problemas detectados foi se os valores examinados em *Context Switch Bitmap* seriam periódicos ou não. Uma das maneiras de facilitar a análise seria promovê-lo a um tipo que armazenasse mais bits, como o tipo nativo em C *long*. Entretanto, variáveis de maior capacidade geram *overhead*, mesmo que pequeno, porque o acesso à variável se dá por modificação dos valores de mais de uma posição de memória (em uma arquitetura 16 bits, são feitas no mínimo dois acessos à memória a uma variável de 32 bits).

A não determinação dessa periodicidade pode fazer o sistema permutar entre diferentes modos de economia de energia, de modo a não tirar o melhor proveito desta seleção dinâmica. Uma das soluções propostas é modificar o valor do tick do sistema, pois se a janela de tempo dada a cada task for maior, pode fazer o sistema se definir por mais tempo em modo Idle. Uma possível melhora seria tornar a seleção do tick dinâmica, modificando-se de acordo com a carga de execução corrente do sistema.

Mesmo com um tick ideal, o sistema ainda está pouco otimizado para tasks de alto processamento. Por exemplo, suponha haver uma câmera em constante movimento conectada a um microprocessador/microcontrolador, sendo abstraída sob a forma de uma task. Se a câmera necessita captar movimentos sob demanda, o atraso gerado pela rotina de tratamento de interrupção do timer de tick pode prejudicar e não permitir uma devida leitura dos dados enviados. O sistema poderia ser otimizado para lidar com tais situações, pois tarefas deste tipo requereriam alta disponibilidade do núcleo de processamento, porém, quando desativadas, espera-se alta economia de energia.

6 Agradecimentos

Primeiramente, gostaria de agradecer ao meu orientador deste Trabalho Final de Graduação, Prof. MSc. Rodrigo Maximiano Antunes de Almeida, sem o qual não haveria pensado em tal otimização para Sistemas Operacionais. Também agradeço à empresa Intermec, na qual estive em período de estágio enquanto elaborava esta tese, que me apoiou na implementação deste projeto, cedendo os recursos necessários. Sem dúvida, outra pessoa que me ajudou a chegar neste nível de maturidade na área de Sistemas Embarcados foi meu colega e grande amigo Lucas Carvalho de Sousa, o qual sempre apaixonado por Eletrônica, me auxiliou muito a aprender e navegar neste assunto quando éramos bolsistas de Iniciação Científica.

Além disso, agradeço à minha família, que até hoje me apoiou e me deu apoio incondicional para que eu terminasse minha Graduação, e graças a eles, pude suportar e superar todas as adversidades financeiras e emocionais sob as quais um universitário está sujeito ao longo dos anos de curso.

Não posso deixar de citar a UNIFEI, e todos os projetos e professores que passaram por minha vida acadêmica, pois foram estes que me direcionaram e deram toda a formação para que eu me consagrasse um Engenheiro da Computação.

Referências

ARDUINO: Site. 2006. Disponível em: <http://arduino.cc>. Acesso em: 03 set. 2013.

BAER, J. *Microprocessor Architecture From Simple Pipelines to Chip Multiprocessors*. [S.l.]: Cambridge University Press, 2010.

BARRETT, S. F.; PACK, D. J. *Embedded Systems. Design and Applications with the 68HC12 and HCS12*. [S.l.]: Editora Prentice Hall, 2004.

BARROS, E.; CAVALCANTE, S. Introdução aos sistemas embarcados. 2002. 2002. Tese apresentada à Universidade Federal de Pernambuco.

BIT Twiddling Hacks - Counting bits set, in parallel: Site. 2005. Disponível em: <http://graphics.stanford.edu/~seander/bithacks.html>. Acesso em: 03 set. 2013.

BOVET, D. P.; CESATI, M. *Understanding the Linux Kernel 3rd Ed*. [S.l.]: O'Reilly Media, 2005.

CHRISTOFFERSON, M. *4 Ways to Improve Linux Performance*. [S.l.], 2013.

DICK, R. P. et al. Power analysis of embedded operating systems. 2000. 2000.

FREERTOS: Site. 2013. Disponível em: <http://www.freertos.org>. Acesso em: 27 jun. 2013.

IAR Embedded Workbench: Site. 2013. Disponível em: <http://www.iar.com>. Acesso em: 04 jun. 2013.

INC., R. E. A. *The True Low Power Concept - Implementing Powerful Embedded Controls with Minimum Energy Requirements*. [S.l.], 2013.

INTERMEC South America: Site. 2013. Disponível em: <http://www.intermec.com>. Acesso em: 10 ago. 2013.

JUANG, P. et al. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. *ACM*, 2002. 2002.

KING, C.-T. *Low-Power Optimization*. [S.l.], 2008. Microcontroller Class Notes.

PINTO, J. V. L. *Microprocessadores II*. [S.l.], 2006. Notas de Aula.

SANTOS, C. A. M. d.; ALMEIDA, R. M. A. Estudo e implementação de sistemas de tempo real para microcontroladores de 8 bits. 2012. 2012.

SANTOS, C. A. M. d.; DEVERICK, J. Balancing responsiveness and power consumption in real time operating systems. 2013. 2013.

SANTOS, C. A. M. d.; DEVERICK, J. Estimating the impacts of scheduling algorithms on energy consumption. 2013. 2013.

SHAW, K. *Advanced Computer Architecture*. [S.l.], 2013. Notas de Aula.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating System Concepts*. [S.l.]: WileyPlus, 2013.

T.I. *MSP430F51x1, MSP430F51x2 Digital Signal Microcontroller Datasheet*. [S.l.], 2010.

T.I. *MSP430x5xx/MSP430x6xx Family User's Guide*. [S.l.], 2011.

TORRES, G. *Hardware Versão Revisada e Atualizada*. [S.l.]: Novaterra Editora e Distribuidora Ltda, 2013.

César Augusto Marcelino dos Santos
Graduando

Prof. MSc. Rodrigo Maximiano A. Almeida
Orientador