Migração de um kernel de tempo real para um processador digital de sinais

José Augusto M. Nacif[1], Luiz Fernando E. Moreira[1], Andréa Iabrudi Tavares[1] , José Monteiro da Mata[1], Antônio Otávio Fernandes[1], Claudionor Nunes Coelho Jr.[1], Luiz Filipe Vieira [1], Marcos Augusto Vieira[1]

[1]Laboratório de Engenharia de Computadores – LECOM
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais
Av. Antônio Carlos, 6627 - ICEx - Campus Pampulha – Belo Horizonte, MG

{jnacif,luizf,iabrudi,mata,otavio,coelho,lfvieira,mmvieira}@dcc.ufmg.br

Abstract. *Considering the complexity growth on signal processing applications and the performance of newest generation digital signal processors the use of small kernels for the signal processing domain becomes common. The use of an operating system facilitates the organization and management of application tasks. This work describes the porting process of a real time kernel to a DSP. The ported kernel is μC/OS, a preemptive kernel targeted to embedded systems. Texas Instruments is used on motion control applications.*

Resumo. *Com o crescimento da complexidade das aplicações de processamento de sinais, os DSPs (Digital Signal Processors) vivem hoje o que aconteceu há uma década atrás com os microcontroladores: a necessidade de se utilizar um sistema operacional. Essa necessidade visa tornar mais simples o gerenciamento das diversas tarefas executadas concorrentemente em um DSP. Este trabalho descreve a migração de um kernel de tempo real para um DSP. O kernel de tempo real migrado foi o μC/OS, um kernel preemptivo, destinado a sistemas embutidos. O processador utilizado no trabalho é um DSP da Texas Instruments empregado na área de controle de motores e inversores.*

1. Introdução

Este trabalho descreve a migração de um *kernel* de tempo real para um DSP (*Digital Signal Processor*). O *kernel* migrado foi o μC/OS, que é gratuito para aplicações não comerciais. A estrutura do μC/OS foi projetada para ser modular, de forma a facilitar sua migração para outros processadores. O processador utilizado é um DSP da *Texas Instruments*, usado na área de controle de motores. A maior parte das aplicações para DSPs são codificadas em *assembly*, organizadas por um laço. Com o crescimento da complexidade de tais aplicações este tipo de estrutura vem se tornando ineficaz em termos de portabilidade, facilidade de manutenção e principalmente aproveitamento de código previamente desenvolvido. A introdução de um *kernel* com recursos de escalonamento e sincronização de tarefas deverá resolver boa parte dessas limitações. As aplicações serão desenvolvidas em forma de tarefas, que serão executadas sobre o *kernel*, trazendo modularidade ao projeto. O processo de migração de um *kernel* para um DSP não é uma tarefa trivial, devido à sua arquitetura extremamente desenvolvida e otimizada para processamento de sinais. No decorrer do trabalho, serão apresentados conceitos de sistemas de tempo real, arquitetura do μC/OS e o detalhamento de todo o processo de migração, enfatizando as dificuldades encontradas

2. Sistemas Operacionais de Tempo Real

Sistemas de tempo real são aqueles onde respostas do sistema a um estímulo externo, em um tempo determinado, são vitais para o seu funcionamento [Krishna and Shin, 1997, Laplante, 1997]. Estes sistemas operacionais podem ser classificados em duas categorias distintas:

- Sistemas de tempo real *hard*: São sistemas que lidam com requisitos de tempo muito rígidos nos quais um possível atraso pode gerar conseqüências catastróficas como, por exemplo, em sistemas de freio de um automóvel.
- Sistemas de tempo real *soft*: São sistemas de tempo real mais flexíveis. Caso algum tempo de resposta pré-determinado não seja respeitado, nada catastrófico ocorre, como por exemplo sistemas multimídia. Porém, nestes sistemas, se os tempos de respostas não são respeitados, o desempenho das aplicações pode cair a valores inaceitáveis.

Existem inúmeros sistemas operacionais comerciais destinados a operação em tempo real como por exemplo o QNX [QNX, 2004] e RTXC [Quadros, 2004] que disponibilizam versões para vários processadores. A utilização de um sistema operacional comercial geralmente envolve custos proibitivos, além de vincular-se o desenvolvimento com módulos tipo caixa preta. Assim, escolheu-se um kernel de tempo real de fonte aberta com vasta documentação para a adaptação ao DSP, o μC/OS [Labrosse, 1992, Labrosse, 2002].

3. Processamento Digital de Sinais

O processamento digital de sinais trata da forma como sinais analógicos são digitalizados e processados. O número de aplicações que envolvem este tipo de processamento cresce a cada dia. As principais áreas onde o processamento de sinais é empregado são: telecomunicações, médicas, industriais e de consumo (celulares, equipamentos de som e imagem). As funções utilizadas para se realizar tais processamentos podem ser implementadas através de qualquer processador convencional ou circuitos integrados específicos. Uma classe especial de processadores programáveis, otimizados para o processamento de sinais, os DSPs, foi criada na década de 80 e hoje são incorporados nos mais diversos tipos de aplicações dentre as citadas acima. As grandes vantagens do uso de DSP em relação aos circuitos dedicados neste tipo de aplicações são:

- Reprogramação em campo permitindo a melhoria de produtos;
- Modificação e correção de erros em projetos através do software;
- Alto desempenho para o processamento de sinais;
- Grande eficiência em consumo de energia;
- Baixo custo.

Os algoritmos utilizados em processamento de sinais modelaram a arquitetura destes processadores. Grande parte das operações realizadas em processamento digital de sinais como filtros, convoluções e transformadas de Fourier, utilizam-se basicamente de operações de multiplicação seguidas de somas. Geralmente os processadores de propósito geral implementam multiplicações através de deslocamentos e operações de adição, consumindo vários ciclos. Assim uma das primeiras características que diferenciam um DSP de um processador de uso geral é a presença de uma unidade de multiplicação que pode realizar uma operação em apenas um ciclo. Explorando as características comuns dos algoritmos de processamento de sinais, inúmeras outras funcionalidades foram criadas em *hardware* para a otimização destes processadores. As principais são:

- Múltiplas unidades de processamento;
- Acesso eficiente à memória;

- Modos de endereçamento especializados para os algoritmos;
- Mecanismos eficientes de I/O;
- Conjunto de instrução especializado;
- Barramentos de dados e endereços separados (arquitetura *Harvard*).

Todas estas características foram derivadas do comportamento dos algoritmos de processamento de sinais e incorporadas nos DSPs trazendo o alto desempenho numérico. As aplicações em processamento de sinais consistem em processos sincronizados de conversão analógico-digital, processamento, armazenamento e/ou conversão digital analógica. Este processo, de acordo com as aplicações que podem exigir taxas de amostragem altíssimas, apresenta características de processamento em tempo real.

4. Características Básicas do sistema operacional μC/OS

O μC/OS [Labrosse, 1992, Labrosse, 2002] foi desenvolvido com o objetivo de ser um *kernel* de tempo real, preemptivo, portável, multitarefa e destinado à utilização em qualquer tipo de microcontroladores, DSPs ou GPPs (*General Purpose Processors*). A motivação para a criação do μC/OS foi a falta de sistemas operacionais embutidos de tempo real que apresentassem boa confiabilidade e custo acessível. Grande parte do sistema é escrito em C, permitindo, assim, que o mesmo seja facilmente portado. A parte do sistema escrita em *assembly* é mínima e compreende características específicas do processador no qual o sistema será utilizado. A parte codificada em C pode ser compilada em qualquer compilador ANSI C. O *kernel* pode gerenciar até 63 tarefas e tem desempenho comparável a *kernels* comerciais, provendo vários serviços que incluem caixas de mensagem, filas, semáforos e funções de temporização.

O μC/OS é um *kernel* baseado em prioridades e sempre executa a tarefa pronta de maior prioridade. Este *kernel* é totalmente preemptivo de modo que, se uma interrupção suspender a execução de uma determinada tarefa e se outra tarefa de maior prioridade ficar ”pronta”como efeito da interrupção, a mesma irá executar assim que o tratamento da interrupção terminar. Este mecanismo de interrupção pode ser aninhado em até 255 níveis. O gerenciamento de seções críticas é tratado no μC/OS através da desabilitação de interrupções. Assim, quando o μC/OS vai entrar em uma seção crítica, para garantir acesso exclusivo e a integridade dos dados que estão sendo acessados, ele desabilita as interrupções, voltando a habilitá-las quando termina. São providas duas macros responsáveis por esta operação: OS_ENTER_CRITICAL e OS_EXIT_CRITICAL. Estas macros são específicas do processador utilizado, devendo ser criadas durante o processo de migração. Uma das limitações do μC/OS é a prioridade única, assim duas tarefas nunca podem ter a mesma prioridade não sendo possível o escalonamento em *round-robin*. As tarefas no μC/OS são laços infinitos. Mesmo o laço infinito pode ser interrompido na presença de uma tarefa de prioridade mais alta. As chamadas de sistema que estão acessíveis às tarefas são:

- OSTask_DEL: Apaga uma tarefa através de sua prioridade;
- OSTimeDly: Atrasa a tarefa em um determinado número de *ticks* de relógio;
- OSSEMPend: Decrementa o semáforo e retorna. Caso o valor do semáforo seja menor ou igual a zero, a função decrementa o semáforo e coloca a tarefa em uma lista de espera. Esta chamada é usada quando uma tarefa necessita de acesso exclusivo a algum recurso, sincronizar com alguma ISR (*Interrupt Service Routine*) ou esperar a ocorrência de um evento;
- OSMboxPend: Usada quando uma tarefa precisa receber uma mensagem;
- OSQPend: Usada quando uma tarefa precisa receber uma mensagem de uma fila.

As tarefas recebem, em sua criação, um único nível de prioridade de 0 a 62, totalizando as 63 tarefas possíveis. No μC/OS uma tarefa pode estar em um dos seis estados possíveis: dormindo, pronta, executando, atrasada, pendente e interrompida.

Na Figura 1 tem-se o diagrama de estados das tarefas e as chamadas responsáveis pelas trocas de estado.

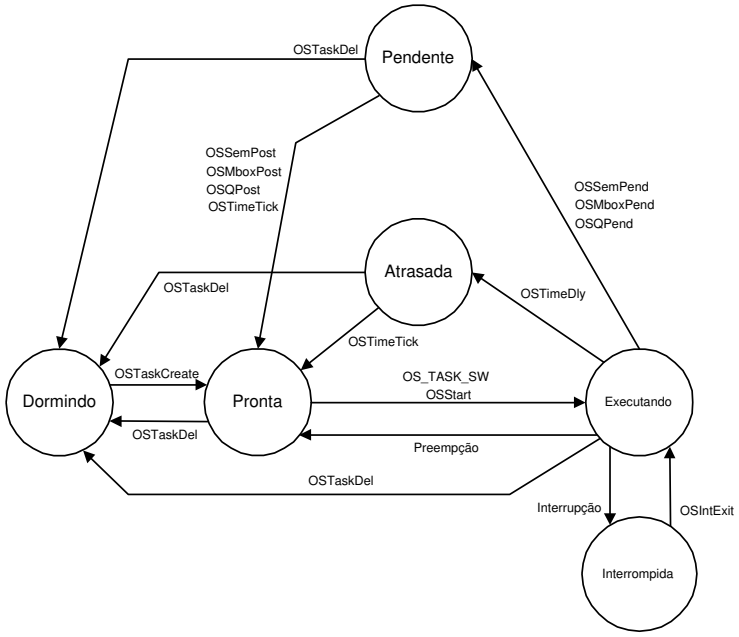

Figura 1: Diagrama de transição de estados do μC/OS

O controle de estado das tarefas é realizado através de uma estrutura de dados. Cada tarefa possui uma estrutura de controle associada, chamada TCB (*Task Control Block*). O bloco de controle de tarefas é responsável por manter o estado da tarefa quando ela é interrompida. Quando a tarefa volta a ser executada pela CPU, ela carrega os dados do TCB, continuando exatamente do ponto em que havia parado. A estrutura do TCB é:

- OSTCBStkPtr: Contém o apontador para o topo da pilha da tarefa. No μC/OS, cada tarefa pode ter pilhas de tamanhos diferentes;
- OSTCBStat: Nesta variável temos o estado da tarefa, por exemplo se o valor for 0, a tarefa está pronta;
- OSTCBPrio: Contém a prioridade da tarefa;
- OSTCBDly: É usado quando uma tarefa precisa ser atrasada por um certo número de *ticks*. Quando o valor é 0 a tarefa não precisa ser atrasada;
- OSTCBX, OSTCBY, OSTCBBitX e OSTCBBitY: São usadas para acelerar o processo de tornar uma tarefa pronta ou para fazer uma tarefa esperar por um evento;
- OSTCBNext e OSTCBPrev: São usados para criar uma lista duplamente encadeada de TCBs. Esta estrutura é usada para que um elemento na cadeia seja rapidamente removido.
- OSTCBEventPtr: É um apontador para um Bloco de Controle de Eventos.

O μC/OS executa sempre a tarefa pronta de maior prioridade, por esse motivo, cada tarefa tem um número de prioridade único. Todas as tarefas prontas para executar são colocadas em uma lista de prontos composta por duas variáveis, OSRdyGrp e OSRdyTbl[]. Cada bit na variável OSRdyGrp indica que pelo menos uma tarefa em cada grupo da OSRdyTbl[] está pronta para executar, como mostra a Figura 2. A relação entre OSRdyGrp e OSRdyTbl[] é mostrada a seguir:

O Bit i na OSRdyGrp é 1 quando qualquer Bit em OSRdyTbl[i] é 1.

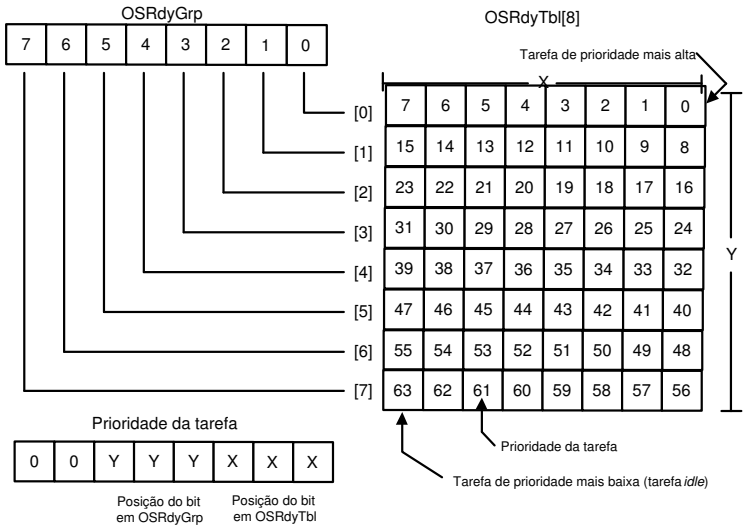

Figura 2: Lista de tarefas prontas no μC/OS

Além das variáveis OSRdyGrp e OSRdyTbl[], dois vetores de constantes são usadas para o cálculo da tarefa a ser executada. OSMapTbl é usado quando uma tarefa é colocada na lista de prontos. OSUnMapTbl é a tabela de resolução de prioridades. O cálculo da tarefa de mais alta prioridade pronta é mostrado na Figura 3.

```
y = OSUnMapTbl[OSRdyGrp];
x = OSUnMapTbl[OSRdyTbl[y]];
prio = (y<< 3) + x;
```

Figura 3: Trecho de código onde é calculada a tarefa de mais alta prioridade

Por exemplo, no caso da prioridade mais alta ser 59: OSRdyGrp = 128 y = OSUnMapTbl[OSRdyGrp] = 7 OSRdyTbl[y] = 248 x = OSUnMapTbl[OSRdyTbl[y]] = 3

A localização da prioridade está em x=3 e y=7 (observar OSRdyTbl, Figura 2). Uma tarefa é criada através da função OSTaskCreate(). Trata-se de uma função específica de processador. O protótipo de OSTaskCreate() é mostrado na Figura 4:

```
UBYTE OSTaskCreate(void (OS\_FAR * task)(void *pd), void *pdata,
void *pstk, UBYTE p)
```

Figura 4: Protótipo de OSTaskCreate()

Onde *task* é o apontador para o código da tarefa, *data* é uma área definida pelo usuário para passar argumentos para tarefa, *pstk* é o apontador para a pilha da tarefa (usada para salvar o conteúdo dos registradores quando a tarefa é interrompida) e *p* é a prioridade da tarefa.

5. Migração do sistema operacional μC/OS para o DSP

O μC/OS foi projetado para ser um *kernel* de fácil portabilidade. Algumas características do processador de destino e do compilador utilizado são necessárias:

- O compilador deve ser capaz de gerar código reentrante, isto é, código que possa ser compartilhado por mais de uma tarefa sem o risco de corrupção dos dados envolvidos;
- O processador deve suportar interrupções e deve prover uma interrupção em intervalo periódico na faixa de 10 a 100ms;
- Deve ser possível desabilitar todas a interrupções do processador;
- O processador deve suportar endereçamento de pilha.

O processador deve possuir instruções específicas para carregar e salvar registradores e o ponteiro da pilha na memória ou na própria pilha. O código do *kernel* pode ser dividido da seguinte forma:

- Código Fonte Independente de Processador: este código está escrito em ANSI C e não precisa ser modificado para nenhum processador;
- Código Fonte Específico do Processador: esta parte do código tem que ser codificada de acordo com o processador.

O DSP alvo da migração é o TMS320F243 [TexasInstruments, 1999] da família C2000. A família C2000, de 16 Bits, é muito utilizada em controle de motores e inversores. Verificou-se a adequação de todas as características necessárias ao processador utilizado, bem como ao seu compilador, além do ambiente de desenvolvimento que fornece algumas bibliotecas úteis, como, por exemplo, as funções de salvamento de contexto. Funções de salvamento e carregamento de registradores são fornecidas pela *Texas Instruments*. A otimização de tais funções pode interferir diretamente no *throughput* do sistema uma vez que, dependendo da organização e quantidade de registradores, o tempo para realizá-las pode ser longo. As primeiras macros escritas foram relativas à habilitação e desabilitação das interrupções, OS_ENTER_CRITICAL() e OS_EXIT_CRITICAL(). Através do estudo da arquitetura do DSP observou-se que através de apenas um registrador poderia-se habilitar e desabilitar globalmente todas as interrupções mascaráveis. A Figura 5 apresenta as macros.

```
#define OS\_ENTER\_CRITICAL() asm("     SETC INTM");
#define OS\_EXIT\_CRITICAL()  asm("     CLRC INTM");
```

Figura 5: Macros de habilitação/desabilitação de interrupções

Com as instruções SETC e CLRC o valor do bit INTM está sendo modificado no registrador ST0, registrador de *status* e controle, responsável pelo controle global das interrupções. A diretiva asm indica ao compilador C a ocorrência de uma instrução *assembly* que é apenas copiada junto ao código gerado pelo mesmo, sem modificações. O passo seguinte no processo de migração foi o processo de geração do *tick* de interrupção. Para geração de temporização, o DSP da *Texas Instruments* possui dois *Timers* que podem ser usados para gerar tal *tick*. Optou-se pelo *Timer* 1, porém a programação é genérica podendo ser utilizado com o *Timer 2*, necessitando-se apenas a mudança nos vetores de interrupção. As interrupções referentes ao *Timer* 1 estão concatenadas na INT2. Poderiam ser usados *Underflow*, *Overflow*, período ou comparação para gerar o *tick*. Optou-se por gerar interrupções de período. Cada interrupção é gerada quando o contador do timer atinge o valor carregado no registrador de período. Em seguida o contador do timer é zerado e inicia-se novamente a contagem. Para determinar o período, deve-se levar em conta a freqüência em que o DSP está funcionando e o *prescaler* do *timer*. O prescaler divide a freqüência do DSP por um valor x, antes da entrada do *timer*. Isto permite que o timer trabalhe em diferentes faixas de freqüência. Na placa

de desenvolvimento, o DSP opera a 20 MHz, ou seja com um período de 50 ns. O *prescaler* foi setado para 128 ou seja 20MHz/128 que é igual a 156,25 KHz. Com a freqüência de 156,25 KHz, tem-se um período de 6,65 ms. O valor escolhido para o *tick* do *kernel* foi 50 ms, mas qualquer valor entre 10 ms e 200 ms seria igualmente satisfatório [Labrosse, 1992, Labrosse, 2002]. O valor que deve ser carregado no registrador de período é de 1E84h, pois: $1E84h * 6,65E - 06 = 50ms$ Para programar o *timer 1* para gerar uma interrupção periódica, deve-se manipular os seguintes registradores [TexasInstruments, 1999]:

- GPTCON: *Status* do *timer 1*, polaridade da saída do *timer 1*;
- T1CNT: Contador do *timer 1*, deve ser zerado;
- T1CON: Comportamento da emulação, Modo de contagem do *timer* 1, valor do *prescaler* e fonte do *clock* de entrada;
- EVIMRA: Habilita interrupção de período;
- T1PR: Possui o valor que o contador deve atingir para ser zerado;
- EVIFRA: Limpa as interrupções pendentes.

A seqüência correta de inicialização dos registradores acima é mostrada na Figura 6

```
SPLK #0000000000001111b,GPTCON
SPLK #0000000000000000b,T1CNT
SPLK #1101011101000000b,T1CON
SPLK #1E84h,T1PR
SPLK #0000000010000000b,EVIMRA
SPLK #0000000010000000b,EVIFRA
```

Figura 6: Inicialização dos registradores do DSP para geração do *tick*

Na seqüência da migração 4 funções em *assembly* e uma em C devem ser codificadas. As funções escritas para a migração em assembly são:

- _OSTickISR: Gerencia o *tick* de relógio do sistema;
- _OSStartHighRdy: Executa a tarefa de mais alta prioridade;
- _OSCtxSw: Realiza a troca de contexto;
- _OSIntCtxSw: Realiza a troca de contexto chamada por uma interrupção.

A função em C é a função OSTaskCreate(). Em seguida serão detalhadas as funções escritas através de seus algoritmos. A função OSTickISR é responsável pelo gerenciamento do *tick* de relógio do sistema. O algoritmo de OSTickISR pode ser visto na Figura 7.

A próxima função é _OSStartHighRdy que é responsável por inicializar a tarefa de mais alta prioridade que esteja pronta, inicializando a multitarefa. O algoritmo desta função pode ser visto na Figura 8.

Em seguida foi escrita a função _OSCtxSw responsável pela realização da troca de contexto. Pode-se ver na Figura 9 o algoritmo desta função.

Em seguida foi escrita a função _OSIntCtxSw que também é responsável pela troca de contexto porém, quando esta troca é inicializada através de uma interrupção. A Figura 10 apresenta o algoritmo desta função.

A última função codificada foi OSTaskCreate(), responsável pela criação de uma nova tarefa no μC/OS. Trata-se da única função específica de processador codificada em C. O algoritmo desta função pode ser visto na Figura 11:

```
void OSTickISR(void) {
salva os registradores da tarefa corrente;
habilita interrupções aninhadas através de OSIntNesting;
chama OSIntEnter() informa ao sistema sobre a entrada em
rotina de interrupção;
chama OSTimeTick() incrementa o tick de relógio do sistema;
chama OSIntExit () informa ao kernel o término da
rotina de interrupção;
restaura os registradores; }
```

Figura 7: Algoritmo de OSTickISR()

```
void OSStartHighRdy (void) {
carrega o apontador de pilha da tarefa de mais alta prioridade;
aponta para a tarefa de mais alta prioridade;
restaura os registradores da pilha da nova tarefa;}
```

Figura 8: Algoritmo de OSStartHighRdy()

```
void OSCtxSw (void) {
salva os registradores da tarefa corrente;
salva o ponteiro de pilha da tarefa interrompida no OS_TCB;
aponta para o TCB da tarefa de mais alta prioridade;
carrega o novo ponteiro de pilha; restaura os registradores;
retorna para a nova tarefa; }
```

Figura 9: Algoritmo de OSCtxSw()

```
void OSIntCtxSw (void) {
salva os registradores da tarefa corrente;
habilita interrupções aninhadas através de OSIntNesting;
chama OSIntEnter() que informa ao sistema sobre a entrada em
rotina de interrupção;
chama OSTimeTick() que incrementa o tick de relógio do sistema;
chama OSIntExit() informando ao kernel sobre a saída da rotina;
restaura os registradores; }
```

Figura 10: Algoritmo de OSIntCtxSw()

O programa principal (Figura 12)é responsável pela inicialização do sistema (inclusive a programação do timer que irá gerar o *tick* de relógio), criação das tarefas, e inicialização do mecanismo de multitarefa.

A função InitTimer() carrega o valores nos registradores do DSP de forma a gerar um *tick* de 50 ms. O sistema é inicializado pela função OSInit(). Primeiramente, variáveis

```
void OSTaskCreate(void) {
salva os registradores da tarefa corrente;
habilita interrupções aninhadas através de OSIntNesting;
chama OSIntEnter() que informa ao sistema sobre a entrada em
rotina de interrupção;
chama OSTimeTick() que incrementa o tick de relógio do sistema;
chama OSIntExit() informando ao kernel sobre a saída da
rotina;
restaura os registradores; }
```

Figura 11: Algoritmo de OSTaskCreate()

```
void main(void) {
    int i;
    extern void InitTimer(void);
     InitTimer();
    OSInit();
    OSTaskCreate(pisca,(void *)&i,(void *)
    &piscastk[OS\_TASK\_IDLE\_STK\_TOP], 59);
    OSTaskCreate(memo1,(void *)0,(void *)
    &memo1stk[OS\_TASK\_IDLE\_STK\_TOP], 60);
    OSTaskCreate(memo2,(void *)0,(void *)
    &memo2stk[OS\_TASK\_IDLE\_STK\_TOP], 61);
    OSTaskCreate(memo3,(void *)0,(void *)
    &memo3stk[OS\_TASK\_IDLE\_STK\_TOP], 62);
    OSStart();}
```

Figura 12: Programa principal do μC/OS

do μC/OS como TCBs (*Task Control Blocks*), *Event Control Blocks* e OSRdyTbl[] são criadas e inicializadas. Em seguida, a tarefa *idle* é criada com a menor prioridade possível (63). Após a inicialização do sistema, as tarefas que irão ser executadas no μC/OS são criadas através da função OSTaskCreate(). Com as tarefas já criadas, agora podemos inicializar o mecanismo de multitarefa através da função OSStart(). Esta função faz o cálculo da tarefa de maior prioridade e a executa. Primeiramente observou-se que quando a função OS_TaskCreate() cria a tarefa *idle*, o parâmetro referente à pilha estava aponta inicialmente para o fim. Isto porque o μC/OS foi escrito inicialmente para uma arquitetura intel x86, em que a pilha deve se decrementada para salvar contexto e incrementada para restaurar. As rotinas que foram usadas para salvar e restaurar contexto, provenientes da *Texas Instruments*, fazem exatamente o oposto, incrementam o apontador da pilha para salvar contexto e decrementam para restaurar. Para solucionar o problema, o parâmetro referente à pilha foi mudado para 0, ficando assim o apontador referenciando inicialmente o topo da pilha. Continuando a depuração do código, observou-se que o cálculo da prioridade não estava sendo feito corretamente. Examinando o código independente de processador de escalonamento contido no arquivo UCOS.C (onde são feitos os cálculos das prioridades) notou-se que os vetores usados para o cálculo da prioridade são definidas como constantes. Como o DSP usa uma arquitetura *harvard* (memória de programa e de dados separaradas), constantes são por *default* mapeadas na memória de programa. Desta

maneira o linker ao invés de pegar o conteúdo dos vetores de constantes OSMapTbl e OSUnMapTbl o μC/OS fazia o cálculo de prioridades substituindo o valor pelo endereço em que estavam alocados OSMapTbl e OSUnMapTbl. Este problema foi contornado alocando as constantes OSMapTbl e OSUnMapTbl na memória de dados. Outro erro encontrado durante a depuração foi a habilitação do *tick* de relógio antes da inicialização da multitarefa. Após várias consultas ao material bibliográfico, descobriu-se no livro μC/OS II referência a este tipo de erro. Quando o *tick* é habilitado antes da inicialização da multitarefa, o μC/OS se encontra em estado desconhecido, causando a parada do sistema.

A quantidade de memória necessária para armazenar o *kernel* é de aproximadamente 30 kb. Como o DSP utilizado possui apenas 8 kb de memória flash interna, tornou-se necessária a utilização de memória externa (já disponível na placa de desenvolvimento) para o armazenamento do *kernel*.

6. Conclusões e trabalhos futuros

O trabalho realizado constituiu na migração de um *kernel* de tempo real para um DSP. Apesar da utilização de sistemas operacionais não ser comum em DSPs, devido a particularidades da arquitetura e das aplicações, a migração foi realizada com sucesso. Contudo, houve necessidade de modificação em algumas partes do código do μC/OS, a princípio independente de processador. A seqüência deste trabalho será a análise de desempenho de todo o sistema, através de aplicações de processamento de sinais, como filtros, e também aplicações de controle em tempo real. Aplicações deste tipo utilizam grande parte dos periféricos do DSP como ADC e *Timers*, além das particularidades da arquitetura, como os multiplicadores e os modos de endereçamento. A análise de desempenho será feita em relação a parâmetros ajustáveis do *kernel* e a diferentes cargas de aplicações, para definir os limites de desempenho e o impacto do *kernel* no ambiente.

Referências

Krishna, C. M. and Shin, K. G. (1997). *Real-Time Systems*. McGraw-Hill.

Labrosse, J. J. (1992). *μC/OS The Real-Time Kernel*. R & D Publications.

Labrosse, J. J. (2002). *μC/OS-II The Real-Time Kernel*. CMP Books.

Laplante, P. A. (1997). *Real Time Systems Design and Analysis*. IEEE Press.

QNX (2004). Qnx - real time operating systems. http://www.qnx.com.

Quadros (2004). Rtxc - real time operating systems. http://www.rtxc.com.

TexasInstruments (1999). Tms320f243, tms320f241 dsp controllers datasheet.