Fundamentos de Sistemas Operacionais de Tempo Real

Criando o seu próprio escalonador de tarefas

Marcelo Barros de Almeida
marcelobarrosalmeida@gmail.com

Sumário

1 Introdução....1
2 O relógio do escalonador....1
3 Troca de contexto....2
3.1 Modo interrompido....2
3.2 Bloco de controle de tarefas....4
3.3 Adicionando tarefas ao RTOS....5
3.4 O escalonador....6
4 Controle de tarefas....9
5 Inicializando o sistema....11
6 Conclusão....11
7 Agradecimentos....12
8 Referência....12
9 Licenciamento....12

1 Introdução

Neste artigo são explorados os princípios básicos de criação de um sistema operacional de tempo real (RTOS). Usando um microcontrolador MSP430 como referência e um compilador GNU GCC, as tarefas mais importantes no processo de criação de um RTOS são detalhadas através de um projeto conhecido como “Basic RTOS” [1], criado especificamente para este fim e requerendo apenas 128 bytes de RAM e 1150 bytes de flash.

Mesmo sistemas com grandes restrições de memória podem se beneficiar de RTOSs como o descrito aqui, evitando estratégias tradicionais de funcionamento apenas baseado em interrupções e facilitando o desenvolvimento. Os conceitos abordados são gerais e não estão restritos ao MSP430 muito menos ao compilador empregado, provendo um conhecimento indispensável para desenvolvedores interessados em aplicar técnicas similares aos seus projetos.

2 O relógio do escalonador

Quando se coloca o requisito de executar mais de uma tarefa no sistema, o problema a ser resolvido imediatamente é o de como compartilhar o processador nesta situação. A estratégia comumente adotada em sistemas com um processador só é dividir o seu uso entre todas as tarefas, dando uma fatia de tempo para cada uma delas (chamado de *time slicing*, ou fatia de tempo) e criando uma forma de ir alternando as tarefas, isto é, um jeito de ir “escalonando” tarefas no processador.

Isto exige que seja criado um mecanismo de tempo que permita, de tempos em tempos, avaliar a lista de tarefas e decidir quem irá entrar em execução (função conhecida como escalonador ou

scheduler, em inglês). Em geral, usa-se uma interrupção de temporizador (*timer*) de alta prioridade do sistema e define-se um tempo de avaliação periódico do sistema. Este tempo será a resolução mínima de execução a ser usada por uma tarefa e é comumente chamado de "*tick* do sistema" (batimento). Por exemplo, é comum ter sistemas com *tick* de 10ms ou 1ms para PCs. Isto depende do que se pretende em termos de responsividade das tarefas e de quanto se quer perder de processamento executando o escalonador. Um número muito baixo pode acabar desperdiçando muito processamento no escalonador e um muito alto pode atrasar a resposta de uma tarefa.

A estratégia adotada é dependente dos recursos disponíveis na sua plataforma e, neste caso, foi usado um *watchdog timer* configurado com tick de 0,5ms, para um sistema que opera com relógio de 1MHz (o *clock* do sistema). Inicialmente, coloca-se o escalonador de tarefas na interrupção de *watchdog*, com mostrado no fragmento de código a seguir. A forma de fazer isto vai depender do seu compilador, hardware e software de suporte à placa. No exemplo está sendo usado o compilador GCC para MSP430.

```
/* RTOS scheduler function is allocated at watchdog interrupt */
static interrupt (WDT_VECTOR) BRTOS_Scheduler(void);
```

Definindo a interrupção de watchdog no GCC para MSP430

Depois, na configuração inicial do RTOS, o tick pode ser ajustado, através de uma chamada para *BRTOS_ConfigureClock()*, dentro da função de inicialização do sistema (chamada de *BRTOS_Initialize()* e discutida mais ao final):

```
static void BRTOS_ConfigureClock(void)
{
      WDTCTL = WDT_MDLY_0_5;        /* configuring interval timer */
      usTicksPerSecond = 1000/0.5; /* 2k ticks per second        */
}
```

Inicializando o clock do sistema

3 Troca de contexto

Antes de tratar a troca de contexto propriamente dita, é preciso entender o que acontece quando o processamento normal do microcontrolador é alterado por uma interrupção. Obviamente isto é uma questão dependente do hardware mas existem algumas ações esperadas.

3.1 Modo interrompido

Quando uma interrupção está sendo tratada pelo microcontrolador é comum se referir à expressão "modo interrompido" para caracterizar esta situação. Em modo interrompido, o ponto do processamento irá mudar para que a interrupção seja atendida adequadamente. Neste caso, o contexto do sistema precisa ser salvo, isto é, uma espécie de foto dos registros e modo de operação do processador naquele instante é efetuada, para que o estado possa ser posteriormente recuperado quando a interrupção terminar de ser processada. Em geral, é comum realizar o salvamento de determinados elementos do processador, descritos a seguir:

- **O contador de programa**, mais conhecido como *program counter* (PC), que indica onde estava a execução no momento da interrupção. Em geral, o PC é um registro que armazena o valor do endereço de memória da próxima instrução a ser executada.

- **O ponteiro da pilha** ou *stack pointer* (SP). O SP indica a posição corrente do stack, podendo apontar para o endereço de memória da próxima posição livre na pilha. Algumas plataformas usam o SP de forma diferente, fazendo que ele aponte para a última posição ocupada. A direção em que a pilha é usada também varia. Algumas plataformas vão decrementando o SP à medida que a pilha é usada, com o SP inicialmente apontando para o fim do stack, enquanto em outras o SP é inicializado apontando para o início do stack, sendo o SP incrementado durante o seu uso.
- **O registro de status**, ou *status register* (SR), no momento da interrupção. Por exemplo, tinha um *flag* de *overflow* ligado no momento da interrupção? Se sim, ele precisa estar ligado quando o contexto for restaurado. Isto também pode variar entre plataforma mas é comum existir alguma forma de salvamento do status em todos os microcontroladores.
- **Registros de finalidade geral**, totalmente dependentes da arquitetura utilizada.

Certas controladores, como o ARM7TDMI, possuem registros espelhados. Por exemplo, ao entrar no modo interrompido, você tem um registro específico para o SP em modo interrompido, trocado pelo processador no momento da interrupção, automaticamente. Enfim, o que salvar é altamente dependente do processador e irá requerer um bom conhecimento do controlador que está sendo usado.

O seu compilador também irá influenciar neste processo, já que ele é que irá montar o salvamento de contexto antes da entrada da interrupção. Por exemplo, se no atendimento de uma interrupção ele só usa um subconjunto dos registros, ele pode gerar um código assembly que só salve este subconjunto, gerando um chaveamento de contexto mais rápido do que salvar todos os registros. Não é a toa que plataformas como ARM permitem o salvamento simultâneo de vários registros, numa única instrução, diminuindo o tempo.

No caso do MSP430, existem 16 registros, como pode ser visto na Figura 1.

R0 (PC) Program counter
R1 (SP) Stack pointer
R2 (SR) Status register
R3 (CG) Constant generator
R4 a R15 Registros gerais

Figura 1: Registros do MSP430

Ao acontecer uma interrupção, o processador coloca automaticamente na pilha corrente os valores do SR e PC. Qualquer salvamento adicional é feito pelo compilador, na geração do código de entrada e saída da interrupção. Ou, caso escrevendo código em assembly, será preciso salvar os registros que pretende usar dentro da função de interrupção na pilha e restaurá-los antes do retorno. O retorno de uma interrupção é feito com a instrução **reti**, que devolve automaticamente os valores do PC e SR, recuperando o contexto anterior. Em resumo, apenas o SR e PC são salvos automaticamente pelo microcontrolador (Figura 2), o compilador (ou programador) é responsável pelos outros registros.

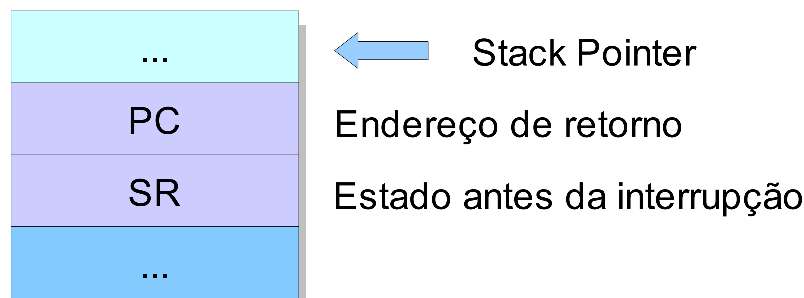

Figura 2: Salvamento de contexto realizado pelo microcontrolador

3.2 Bloco de controle de tarefas

Agora que já descrito como lidar com o salvamento de contexto no caso de uma interrupção, é hora de tratar as tarefas. Um sistema com o objetivo de executar várias tarefas precisa, de alguma forma, controlá-las. É comum se usar uma estrutura conhecida como TCB (*Task Control Block*) para manter os dados mais importantes das tarefas do sistema.

No BRTOS, foi usada a seguinte estrutura, cujos elementos estão descritos mais abaixo. A quantidade de elementos na estrutura irá variar dependendo da sua implementação de RTOS.

```
typedef struct {
      pfTaskEntry     pfEntryPoint;      /* task entry point    */
      unsigned char   ucPriority;        /* task priority       */
      unsigned char   ucTaskState;       /* current task state  */
      unsigned short  usTimeSlice;       /* desired time slice  */
      unsigned short *pusStackBeg;       /* stack beginning     */
      unsigned short *pusStackPtr;       /* stack pointer       */
      unsigned short  usSleepTicks;      /* count sleep ticks   */
      unsigned short  usTicks;           /* count slice ticks   */
} BRTOS_TCB;
```

TCB utilizado para o controle de tarefas

- **pfEntryPoint**: ponto de entrada da tarefa, isto é, o endereço da função que representa a tarefa.
- **ucPriority**: prioridade da tarefa, não usada nesta implementação, mas importante em sistemas que escalonam tarefas levando em consideração a prioridade.
- **ucTaskState**: estado atual da tarefa (em execução, dormindo, etc), importante para o escalonador.
- **usTimeSlice**: *time slice* da tarefa, ou seja, o tempo que a tarefa ficará em execução antes de perder o controle do processador para a próxima tarefa. Medido em ticks, nesta implementação.
- **pusStackBeg**: ponteiro para o início da pilha da tarefa. Cada tarefa irá necessitar de uma pilha própria para que possa executar de forma independente. Um pouco de memória precisará ser reservada para cada pilha sendo que a quantidade final está relacionada com o código da tarefa. Um erro neste dimensionamento em sistemas onde a memória de processos é compartilhada (sem MMU, *Memory Managment Unit*) pode gerar um estouro da pilha. Neste caso, pode-se ter um problema grave ao invadir áreas de memória de outras tarefas ou do sistema operacional.

- **pusStackPtr**: ponteiro para o valor corrente da pilha da tarefa. Este ponteiro será usado para salvar a posição da pilha enquanto a tarefa espera por espaço no processador. Ao voltar a ser executada, o valor desta variável é copiada novamente no SP na operação de restauração de contexto.
- **usSleepTicks**: este campo é usado como contador do tempo decrescente (em ticks) quando a tarefa é colocada em estado de suspensão por um determinado período.
- **usTicks**: contador do número de ticks durante a execução da tarefa, usado para controlar o tempo de execução da tarefa dentro do time slice planejado.

Sistemas com restrições grandes de memória podem tentar construir TCBs com menos elementos para diminuir o uso de RAM. Além disso, é comum usar uma lista duplamente ligada de elementos BRTOS_TCB para manter o controle das tarefas. Mas, para simplificar, reduzir o tamanho do código e uso de RAM, foi criado um vetor com BRTOS_MAX_TASKS posições já previamente alocadas.

3.3 Adicionando tarefas ao RTOS

Para adicionar uma nova tarefa no sistema, basta definir a função que a represente, reservar memória para a pilha e adicioná-la no TCB através de uma chamada para a função BRTOS_CreateTask().

Num microcontrolador com 1Kb de RAM e 8Kb de Flash, chamadas como *malloc()* provavelmente não estão disponíveis para uso e a reserva de espaço para pilha pode ser feita através do emprego de um vetor global, do tamanho desejado. No exemplo a seguir, foram reservados 50 bytes para a pilha da tarefa. Note que foi usado um vetor com elementos de 16 bits (*short*) e não de 8 bits (*char*), com tamanho igual à metade do espaço desejado para fazer a reserva adequadamente. Pode parecer estranho não usar diretamente um vetor do tipo *char*, mas esta ação irá garantir que o vetor esteja alinhado na memória em valores múltiplos de 16 bits, um requisito para a pilha em plataformas onde o alinhamento é importante. Desta forma, use um tipo de dado no seu vetor que seja do mesmo tamanho da palavra da plataforma (o tipo int, em geral, resolve isso).

A tarefa, por sua vez, é apenas um laço infinito que executa a sua operação, como pode ser visto no código a seguir.

```
#define TASK_STACK_SIZE 50

unsigned short usStack[TASK_STACK_SIZE/2];

void task(unsigned long ulArgc)
{
    while(1)
    {
        int i = 0;
        while(i < 50)
        {
            i = i + 1;
        }
        BRTOS_Sleep(5);
    }
}
```

Exemplo de tarefa e alocação de stack

No Basic RTOS, todas as tarefas devem ser criadas dentro da chamada BRTOS_Application_Initialize(). Não foi previsto a criação de tarefas dinamicamente, uma restrição relativamente sensata quando se pensa num sistema microprocessado pequeno.

```
void BRTOS_Application_Initialize(void)
{
  BRTOS_CreateTask(task,                           /* ponto de entrada     */
    (unsigned short)&usStack[TASK_STACK_SIZE/2-1],/* pilha (fim do vetor) */
    10,                                            /* time slice (ticks)   */
    BRTOS_TASK_PRIORITY_1);                        /* prioridade (ñ usada) */
}
```

Adicionando uma tarefa ao sistema

Em geral, não se espera que a tarefa retorne mas, se isto acontecer, o sistema operacional precisa lidar com esta condição. Neste caso, no momento da criação da tarefa, é gerado um contexto especial na pilha da tarefa que, caso aconteça um retorno, a coloque em um estado seguro.

3.4 O escalonador

Nesta seção é descrito como implementar um escalonador para o sistema, através da rotina BRTOS_Scheduler(). Esta rotina associada com a rotina de interrupção do watchdog e é o ponto de entrada do batimento periódico do sistema (tick), como descrito anteriormente. Por ser executada em modo interrompido o retorno dela será feito por uma instrução "**reti**", que restaura automaticamente os registros SR e PC da pilha.

Para esta rotina é necessário um tratamento especial por parte do compilador. A palavra reservada "NAKED", presente na definição da função, irá instruir o MSPGCC a criar uma função que não faça nenhuma mudança na pilha ao ser executada. Neste caso, nenhum contexto adicional ou argumento de chamada de função é salvo na pilha, algo bem apropriado para uma função com papel de escalonador do sistema. Caso o escalonador precise usar a pilha, o SP precisa ser salvo e a pilha do escalonador definida.

A rotina do escalonador está a seguir, com comentários sobre cada parte dela logo depois.

```
NAKED( BRTOS_Scheduler )
{
save_context_entry:

    SaveContext();
    SaveStackPointer();
    RestoreSchedStackPointer();

dont_save_context_entry:

    /* Update timers */
    BRTOS_ProcessTimers();
    /* check sleeping tasks */
    BRTOS_SleepTasks();
    /* Get the next task to run */
    ucCurrentTask = BRTOS_RoundRobin(ucCurrentPriLevel);

    if(ucCurrentTask == BRTOS_NO_TASK_TO_RUN)
    {
        /* nothing to do: sleep */
        GoToLowPowerMode3();
        goto dont_save_context_entry;
    }
```

```
    /* save scheduler context */
    SaveSchedStackPointer();
    /* restore stack pointer */
    RestoreStackPointer();
    /* restore other registers */
    RestoreContext();
    /* reti will pop PC and SR from stack (naked function) */
    EnableInterrupts();
    ReturnFromInterrupt();
}
```

Escalonador de tarefas do sistema

A primeira ação é salvar o contexto com SaveContext(). Perceba que deve existir uma tarefa em execução e o ponteiro da pilha atual (SP) tem relação com esta tarefa. SaveContext() irá colocar na pilha desta tarefa os registros relacionados com a chamada de SaveContext(), deixando a pilha da tarefa com a seguinte estrutura:

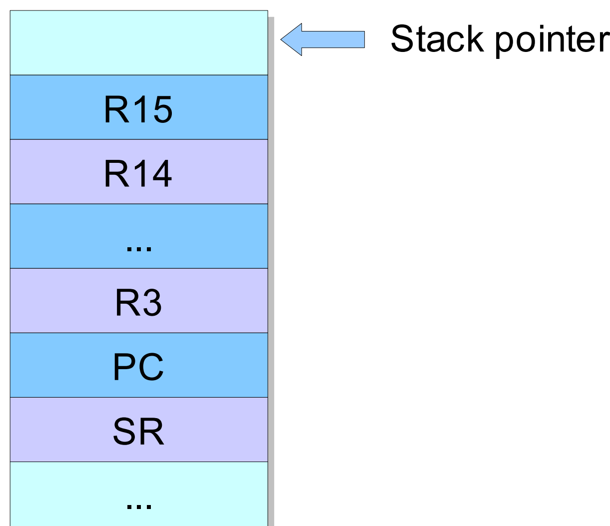

Figura 3: Pilha com o contexto da tarefa salvo

SaveContext() é escrita em assembly já que é preciso controle total neste momento:

```
#define SaveContext() __asm__ __volatile__ ("push R3\n"  \
                                             "push R4\n"  \
                                             "push R5\n"  \
                                             "push R6\n"  \
                                             "push R7\n"  \
                                             "push R8\n"  \
                                             "push R9\n"  \
                                             "push R10\n" \
                                             "push R11\n" \
                                             "push R12\n" \
                                             "push R13\n" \
                                             "push R14\n" \
                                             "push R15" )
```

Salvamento dos registros da tarefa corrente na pilha

Com o contexto salvo, falta agora salvar a posição atual do SP da tarefa no TCB dela, com a chamada SaveStackPointer(). A notação é do conjunto de ferramentas do GCC:

```
#define SaveStackPointer() \
   asm("mov.w R1,%0"  : "=m" (asBrtosTasks[ucCurrentTask].pusStackPtr))
```

Salvando o stack pointer da tarefa corrente

Neste ponto está criada a condição ideal para trocar o SP da tarefa pelo SP do escalonador, devidamente guardado na variável usSchStackPtr através da função RestoreSchedStackPointer(). Definir o SP do escalonador vai permitir que chamadas em C sejam feitas e variáveis locais sejam criadas, reduzindo ao máximo a quantidade de código assembly no sistema. A inicialização do SP do escalonador é feita na partida do sistema, descrito mais adiante.

```
#define RestoreSchedStackPointer() \
   asm("mov.w %0,R1" :: "m" (usSchStackPtr))
```

Restaurando o stack pointer do escalonador

Depois deste início com um bom grau de dependência da plataforma e do compilador, duas tarefas de faxina são feitas. Uma delas é processar os temporizadores (*timers*), ainda não implementada na versão atual, e a outra verificar se as tarefas em modo "SLEEP" ainda precisam continuar dormindo ou não.

```
BRTOS_ProcessTimers();
BRTOS_SleepTasks();
```

Processamento de timers e tarefas em estado de suspensão

O próximo passo é decidir quem ficará com o controle do processador, feito através da chamada BRTOS_RoundRobin() que irá retornar qual tarefa deve ter seu contexto restaurado e ser colocada em execução. O funcionamento desta rotina é descrito mais adiante.

```
ucCurrentTask = BRTOS_RoundRobin(ucCurrentPriLevel);
```

Determinando a próxima tarefa a ser colocada em execução.

Se não existe nenhuma tarefa pronta para entrar em execução, o escalonar coloca o processador em modo de espera (GoToLowPowerMode3()) até que a próxima interrupção do watchdog acorde o processador, voltando algumas linhas através de um goto e refazendo as tarefas de rotina. Este ponto pode ser melhorado já que se o processador sair do modo de espera por outro motivo, a contagem de ticks para tarefas em SLEEP não estará correta. Uma saída bastante empregada é ter uma tarefa para cuidar desta situação, ao invés de fazer isto no escalonador, evitando maiores problemas.

```
f(ucCurrentTask == BRTOS_NO_TASK_TO_RUN)
{
    /* nothing to do: sleep */
    GoToLowPowerMode3();
    goto dont_save_context_entry;
}
```

Modo de economia de energia caso não exista tarefa para ser executada.

Uma vez que se tenha a tarefa que será executada definida, é hora de restaurar o contexto e devolver o processador para ela. As tarefas são relativamente simples: salvar o SP do escalonador, restaurar o SP da tarefa, restaurar os registros da tarefa e retornar da interrupção. A pilha esperada para a tarefa deve ser como a descrita anteriormente para que a tarefa assuma corretamente o contexto.

```
SaveSchedStackPointer();
```

```
RestoreStackPointer();
RestoreContext();
EnableInterrupts();
ReturnFromInterrupt();
```

Tarefas necessárias para restauração do contexto

A rotina apresentada pode ser melhorada e é bem dependente do processador e compilador utilizados. A dependência ficou oculta através de macros que invocam chamadas em assembly, mas são partes do código que precisam ser repensadas caso a plataforma seja diferente.

4 Controle de tarefas

Apesar de a maior parte do BRTOS ter sido coberta, existem alguns detalhes importantes que precisam ser explicados para o adequado funcionamento do sistema. Neste seção serão explorados os detalhes relacionados ao gerenciamento de tarefas através da rotina BRTOS_CreateTask().

O controle de tarefas do sistema é extremamente simples, composto por um vetor de TCBs chamado asBrtosTasks. Ao pedir a criação de uma tarefa, uma nova posição é usada neste vetor. Os parâmetros necessários são o ponto de entrada da tarefa (entry_point), o endereço da pilha (stack_addr), fatia de tempo máxima (time_slice) e prioridade (pri). Estes parâmetros são guardados no vetor de TCBs e a tarefa é colocada no estado de "READY", ou seja, pronta para executar. Vale lembrar que a maioria dos RTOSs permite iniciar uma tarefa em estado de espera, ou seja, que não saia rodando imediatamente. Isto não está implementado neste sistema.

```
asBrtosTasks[usNumTasks].pfEntryPoint = entry_point;
asBrtosTasks[usNumTasks].pusStackBeg  = (unsigned short *) stack_addr;
/* ponteiro corrente da pilha */
asBrtosTasks[usNumTasks].pusStackPtr  = (unsigned short *) stack_addr;
asBrtosTasks[usNumTasks].usTimeSlice  = MSEC_TO_TICKS(time_slice);
asBrtosTasks[usNumTasks].ucPriority   = pri;
asBrtosTasks[usNumTasks].ucTaskState  = BRTOS_TASK_STATE_READY;
```

Inicializando o TCB

O que não é óbvio no processo de adição de uma nova tarefa no sistema é que a pilha inicial da tarefa precisa ser preparada. Como visto anteriormente, quando a interrupção do escalonador acontece, é esperado que a tarefa em execução possua um contexto a ser salvo formado pelos registros gerais (R3 a R15) e também os registros SP e o PC, colocados na pilha por causa da interrupção do escalonador. Logo, é necessário imitar esta pilha inicial no momento da criação da tarefa.

E tem mais um detalhe que não pode ser ignorado. Geralmente, uma tarefa é implementada como um laço infinito mas que alguns cuidados precisam ser tomados caso ela não siga este padrão ou simplesmente retorne. Neste caso, seria um retorno de modo não interrompido, implementado no MSP430 pela instrução “ret”, que apenas muda o PC corrente para um possível PC que esteja na pilha (sem SR, neste caso). A recomendação aqui é colocar um local válido e seguro para este novo PC, no caso uma função chamada BRTOS_TaskEnd(), descrita mais abaixo.

Com todos estes fatores considerados, o diagrama da pilha inicial da tarefa ficará então como na Figura 4. Lembre-se que a lógica de pilha do MSP430 é apontar para um valor livre e decrescer quando valores são adicionados (isto também é dependente de plataforma).

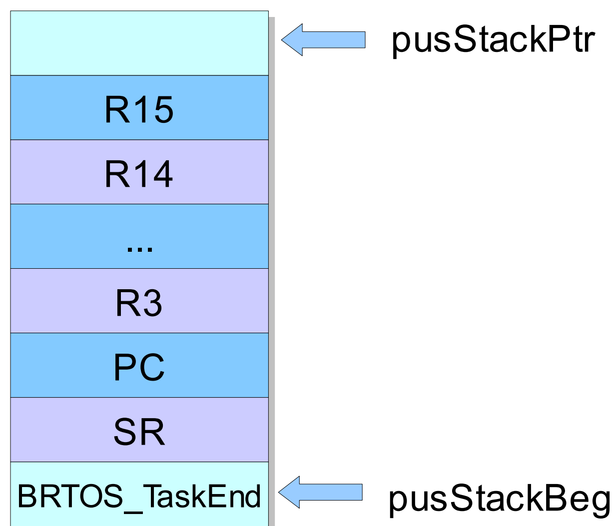

Figura 4: Pilha inicial da tarefa

O código necessário para criar esta pilha inicial pode ser visto abaixo:

```
/* if task returns some day, prepare stack */
*(asBrtosTasks[usNumTasks].pusStackPtr) = (unsigned short) BRTOS_TaskEnd;
asBrtosTasks[usNumTasks].pusStackPtr -=1;

/* status register and program counter */
*(asBrtosTasks[usNumTasks].pusStackPtr) = (unsigned short) 0;
asBrtosTasks[usNumTasks].pusStackPtr -=1;
*(asBrtosTasks[usNumTasks].pusStackPtr) = (unsigned short) entry_point;
asBrtosTasks[usNumTasks].pusStackPtr -=1;

/* prepare for first RestoreContext(): dummy NUM_REGS_IN_CONTEXT regs */
for(i = 0 ; i < NUM_REGS_IN_CONTEXT ; i++)
{
    *(asBrtosTasks[usNumTasks].pusStackPtr) = (unsigned short) 0;
    asBrtosTasks[usNumTasks].pusStackPtr -= 1;
}
```

Criação completa do stack inicial da tarefa.

A implementação da função BRTOS_TaskEnd() é trival, apenas coloca a tarefa no estado de "TERMINATED" e fica num laço infinito de forma que ela não estrague nada até que o escalonador rode novamente e escolha uma tarefa do grupo de tarefas prontas para serem executadas (tarefas em READY). Seria possível fazer algo melhor aqui sem grandes modificações, como uma implementação que permita um reinício (RESTART) da tarefa mesmo depois de ela ter ido para TERMINATED, algo geralmente encontrado em sistemas de tempo real.

```
static void BRTOS_TaskEnd(void)
{
      EnterCriticalSection();
      asBrtosTasks[ucCurrentTask].ucTaskState = BRTOS_TASK_STATE_TERMINATED;
      LeaveCriticalSection();
      while(1);
}
```

Terminado uma tarefa de forma segura.

5 Inicializando o sistema

A partida do sistema merece também atenção, devendo ser feita de forma organizada. Nesta seção é feita uma breve discussão sobre a inicialização do sistema, através da função main(), listada a seguir:

```
int main(void)
{
    /* Saving the current stack pointer. It will be used by the scheduler. */
    SaveSchedStackPointer();
    EnterCriticalSection();

    /* initialize control structures and create all user tasks */
    BRTOS_Initialize();

    /* call user initialization: you must create at least one thread */
    BRTOS_Application_Initialize();

    /* the stack pointer should point to the first available task stack,
      and the context should be subtracted. This way, the calling of
      GoToScheduler will not fail
    */
    asBrtosTasks[ucCurrentTask].pusStackPtr += NUM_REGS_IN_CONTEXT;
    RestoreStackPointer();
    LeaveCriticalSection();

    /* go to to scheduler routine*/
    GoToScheduler();
}
```

Rotina de inicialização do sistema

Várias tarefas são esperadas e não requerem explicações. Perceba que o escalonador vai compartilhar a pilha desta função, já que o SP dele é salvo à partir dela. O que tem de estranho aqui é a manipulação feita no ponteiro da pilha da tarefa corrente e no SP. No fundo, está sendo recriada a situação de pilha que acontece no caso de uma interrupção, por isso remove-se os registros de R3 a R15, restauramos o SP da tarefa e realizamos um salto para o escalonador, como se esta fosse realmente a tarefa corrente. Depois deste ponto, o processamento segue normalmente.

Os pontos de entrada dentro do escalonador (**save_context_entry** e **dont_save_context_entry**) foram inicialmente criados para permitir entrada no escalonador com ou sem salvamento de contexto, o que gera uma ação mais otimizada na partida, sem perder tempo fazendo e desfazendo esta pilha inicial.

6 Conclusão

Através deste artigo foram evidenciados as principais questões envolvidas no processo de construção de um RTOs, como escalonares, troca de contexto, etc. Estas questões foram detalhadas e explicadas, com o objetivo de aumentando o entendimento e fomentar o uso e desenvolvimento de sistemas de tempo real.

7 Agradecimentos

Apesar de este trabalho ter sido totalmente escrito fora da empresa onde o autor trabalha, não há dúvida de que só foi possível devido a experiência adquirida nos vários projetos que participou. Assim sendo, o autor gostaria de agradecer à Smar Equipamentos Industriais LTDA pelas oportunidades e desafios enfrentados.

8 Referência

[1] Basic RTOS project page. http://code.google.com/p/basicrtos/

9 Licenciamento

A obra ”Fundamentos de Sistemas Operacionais de Tempo Real - Criando seu próprio escalonador de tarefas” de Marcelo Barros de Almeida foi licenciada com uma Licença Creative Commons - Atribuição - Uso Não-Comercial - Partilha nos Mesmos Termos 3.0 Não Adaptada.

Com base na obra disponível em http://code.google.com/p/basicrtos/

Podem estar disponíveis permissões adicionais ao âmbito desta licença através do contato direto ao autor via e-mail marcelobarrosalmeida@gmail.com.