Laboratório 1 – Implementação de um Sistema Operacional de Tempo Real Elementar (ERTOS)

A implementação do ERTOS é inspirada na implementação do BRTOS Marcelo B. de Almeida em seu artigo "Implementando Sistemas Operacionais de Tempo Real em Microcontroladores: Edição MSP430". Neste laboratório, iremos criar um kernel elementar com um escalonador de tarefas sequencial cíclico (Round-Robin). Isso nos permitirá entender os diferentes mecanismos envolvidos na troca de tarefas num sistema operacional de tempo real.

Exercício 1 – Aprendendo a misturar C com Assembly.

Crie um projeto no CCS usando o template em C e configure-o para usar o compilador GCC ao invés do compilador da TI. Implemente o código abaixo

```
int main() {
    volatile uint32_t i;

    WDTCTL = WDTPW | WDTHOLD;           // stop watchdog timer
    P4DIR |= BIT7;                      // Configura o LED verde como saída

    while(1) {                          // Entra em loop infinito
        i = 25000;                      // Fica alternando o estado do LED
        while(i--);                     // em intervalos periódicos
        P4OUT ^= BIT7;
    }
}
```

Após a linha `i = 25000;` modifique o valor de i para 50000 em assembly. Você deverá descobrir onde o compilador GCC alocou a variável i na memória usando a vista "Disassembly" em modo debug. Para inserir código em assembly, use a função especial asm(), como abaixo:

```
asm ("mov.w #12,R10");
```

Se você precisa referenciar variáveis de C no código assembly, pode fazer isso usando a sintaxe completa da função asm(), da seguinte forma:

```
asm ("add.w %0,%1" : "=m" (outputVar) : "m" (inputVar) );
```

A instrução e os operandos são escritos na primeira entrada da função. Em seguida vem os operadores de saída e entrada separados por dois-pontos ":". Para operadores de saída, podemos instruir "+" ou forçar "=" o GCC a usar a memória "m" ou os registros "r" para a variável referida em seguida. No exemplo, estamos forçando o GCC a somar (instrução add) a variável inputVar com outputVar e salvar o resultado na memória RAM.

Se não houver entrada, não há a necessidade de escrever a parcela da sintaxe referente à entrada. Se houver apenas a entrada, ou seja, o operador da esquerda (SRC), a parcela relativa à saída deve ser escrita, entretanto deve ficar vazia.

Remova a modificação anterior e substitua a linha `i = 25000;` pela sua equivalente em assembly. Mais informações em:

- http://www.ti.com/lit/an/slaa140a/slaa140a.pdf
- http://mspgcc.sourceforge.net/manual/c1225.html#register-usage
- https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html#Using-Assembly-Language-with-C

Exercício 2 – Uso do Watchdog timer

Configure o WDT para o modo de intervalo e use a sua interrupção para fazer o LED vermelho piscar.

Exercício 3 – Estrutura de dados das tarefas

Escalonadores de processos usam diversas informações para auxiliar na escolha de qual será a próxima tarefa a ser executada pelo processador. Essas informações ficam numa estrutura de dados conhecida como TCB (Task Control Block). Exemplos de informações dessa estrutura são: Ponto de entrada da tarefa, ponteiro para a pilha da tarefa, prioridade, quantum, estado da tarefa, tempo de execução, tempo ocupado no processador, etc.

Iremos modelar apenas o ponto de entrada para a tarefa e o ponteiro atual da pilha da tarefa nessa estrutura de dados. Crie uma struct "task" com apenas esses dois tipos de dados. Em seguida, crie um vetor de tarefas "tasks[]" com um máximo de 10 tarefas.

Registro de novas tarefas.

Crie uma rotina registerTask(...) que recebe um ponteiro de função e inicializa apenas uma entrada do vetor de tarefas, a cada nova tarefa registrada, incremente o índice das tarefas registradas.

Você deve inicializar a pilha da tarefa para prepará-la para um retorno de uma interrupção (RETI). Ela irá executar após o *despachante* realizar a troca de contexto dentro da rotina de tratamento de interrupção do Watchdog. A pilha deve ser inicializada com os seguintes dados:

Ponteiro da pilha após salvar o contexto →	0x000 | R15 [19:16]
	R15 [15:0]
	...
	0x000 | R4 [19:16]
	R4 [15:0]
Ponto de retorno da interrupção →	PC[15:0]
	PC[19:16] | SR [11:0]
Início da pilha →	

A pilha para cada tarefa deve ser reservada num espaço livre da memória. Minha recomendação é iniciar no endereço 0x2800 e reservar 0x80 bytes para cada tarefa (só o contexto já ocupa 0x30 bytes de registros).

Exercício 4 – Escalonador Round-Robin

O escalonador deve ser escrito na rotina de tratamento da interrupção do Watchdog timer. O escalonador (etapa 4) deve ser rodeado pelas etapas de troca de contexto do despachante.

1. Entrada na ISR (SR e PC já estão salvos na pilha da tarefa)
2. Salva o contexto, colocando os registros R[4-15] na pilha
3. Mover o ponteiro da pilha para a pilha do escalonador
4. **Executar o escalonador e obter a nova tarefa a ser executada**
5. Salvar o ponteiro da pilha do escalonador
6. Restaurar o ponteiro da pilha da nova tarefa
7. Restaura o contexto da nova tarefa
8. Retorna da interrupção (RETI)

A ISR do WDT deve ter os atributos abaixo para não alterar a pilha ao entrar/sair da função (naked) e ser mapeada como rotina de interrupção (interrupt(N)), ou seja, seu endereço será colocado na tabela das ISRs.

```
__attribute__ ((naked))
__attribute__ ((interrupt(WDT_VECTOR)))
```

Exercício 5 – Início de execução

O início de execução de um RTOS é especial e será tratado numa rotina a parte. Chamaremos essa rotina de startERTOS(). Ela irá habilitar as interrupções permitindo que os processos sejam interrompidos no meio. Além disso, essa rotina deve desfazer a inicialização da pilha da tarefa (feita no momento do registro) já que a tarefa ainda não foi executada. Finalmente deve saltar para o endereço de entrada da tarefa em questão, ou por meio de um branch (só é válido para endereços de 16-bits), de um salto (relativo ao PC), ou pelo próprio retorno da função (para isso a pilha deve conter o valor de retorno para a primeira tarefa).