Prof Otávio Gomes (otavio.gomes@unifei.edu.br) - Prof Rodrigo Almeida (rodrigomax@unifei.edu.br)

Roteiro 05

Gerenciamento de prioridades e múltiplas tarefas

Leia com atenção - Informações iniciais:

1. No início de cada tópico/assunto é apresentado um **exercício de revisão** em que basta copiar o código na ferramenta, realizar a compilação e a execução e, então, interpretar o resultado. Este tipo de exercício tem como objetivo auxiliar o aluno a relembrar alguns conceitos e a validar as ferramentas que estão sendo utilizadas. Este código sempre estará correto e funcionando.
2. Os exercícios estão apresentados em **ordem crescente de dificuldade**.
3. Os exercícios abordam todos os conceitos relacionados ao conteúdo da aula em questão. Deste modo, caso o aluno não consiga resolver alguns dos exercícios, recomenda-se que o aluno participe dos **plantões de dúvidas/monitorias** e que busque aprender os conceitos envolvidos na atividade.
4. A **próxima atividade** de laboratório admitirá que os conceitos aqui apresentados já foram plenamente compreendidos.

Este tutorial foi traduzido e adaptado a partir do trabalho disponível em "The Designers Guide to the Cortex-M Processor Family" de Trevor Martin.

1) Introdução

Quando uma *thread* é criada, é atribuída a ela um nível de prioridade. O *scheduler* RTOS usa a prioridade de uma *thread* para decidir qual delas deve ser agendada para execução. Se várias threads estiverem prontas para execução, aquela com a prioridade mais alta será colocada no estado de execução.

Se uma *thread* de alta prioridade estiver pronta para ser executada, ela interromperá uma thread em execução de prioridade mais baixa. É importante ressaltar que uma *thread* de alta prioridade em execução na CPU não deixará de ser executada, a menos que seja bloqueada em uma chamada de API RTOS ou seja interrompida por uma *thread* de prioridade mais alta. A prioridade padrão é *osPriorityNormal*.

A prioridade de uma *thread* é definida na estrutura do encadeamento e as seguintes definições de prioridade CMSIS-RTOS estão disponíveis:

- *osPriorityIdle*
- *osPriorityLow*
- *osPriorityBelowNormal*
- *osPriorityNormal*
- *osPriorityAboveNormal*
- *osPriorityHigh*
- *osPriorityRealTime*
- *osPriorityError*

Depois que as *threads* estão em execução, há um pequeno número de chamadas de sistema do SO que são usadas para gerenciá-las. Também é possível elevar ou diminuir a prioridade de uma *thread* de outra função ou de seu próprio código:

```
osStatus osThreadSetPriority(threadID, prioridade);
osPriority osThreadGetPriority(threadID);
```

Além de criar threads, também é possível que uma *thread* exclua a si mesmo ou a outra *thread* ativa do RTOS. Novamente, usamos o ID do encadeamento (*thread*) em vez do nome da função:

```
osStatus = osThreadTerminate (threadID1);
```

Por fim, há um caso especial de troca em que a *thread* em execução passa o controle para a próxima *thread* pronta com a mesma prioridade. Isso é usado para implementar uma terceira forma de escalonamento chamada *troca de thread cooperativa*.

```
osStatus osThreadYield(); //muda para o próximo pronto para executar a thread
```

2) Gerenciando e criando *Threads*

Lembrando do roteiro anterior e analisando-o, é possível mudar a prioridade do LED

```
osThreadDef(led_thread2, osPriorityAboveNormal, 1, 0);
osThreadDef(led_thread1, osPriorityNormal, 1, 0);
```

Neste exemplo, led_thread1 está sendo executado com prioridade normal e led_thread2 está sendo executado com prioridade mais alta, portanto, irá realizar a preempção de led_thread1.

Deste modo, led_thread2 nunca é bloqueado, então ele será executado para sempre, evitando que o thread de prioridade mais baixa seja executado. Esse tipo de erro é muito comum quando os projetistas e programadores começam a usar um RTOS.

3) Múltiplas instâncias

É possível criar várias instâncias em execução do mesmo código de *thread* base. Por exemplo, você pode escrever um *thread* para controlar um UART e, em seguida, criar duas instâncias em execução do mesmo código de *thread*. Aqui cada instância do código pode gerenciar um UART diferente.

Primeiro, criamos a estrutura da *thread* e definimos o número de instâncias do encadeamento para 2:

```
osThreadDef(thread1, osPriorityNormal, 2, 0);
```

Em seguida, podemos criar duas instâncias da *thread* atribuídas a diferentes manipuladores de encadeamento. Também é passado um parâmetro para permitir que cada instância identifique por qual UART ela é responsável.

```
ThreadID_1_0 = osThreadCreate(osThread(thread1), UART1);
ThreadID_1_1 = osThreadCreate(osThread(thread1), UART2);
```

Neste projeto, veremos como criar uma *thread* e, em seguida, criar várias instâncias de tempo de execução do mesmo thread: No *Pack Installer*, selecione **"Ex 4 Multiple Instances"** e copie-o para o diretório do seu tutorial.

Este projeto executa a mesma função do programa pisca-LED anterior. No entanto, agora temos uma função de chaveamento de led que usa um argumento passado como parâmetro para decidir qual LED piscar:

```
void ledSwitcher (void const *argument) {
    for (;;) {
        LED_On((uint32_t)argument);
        delay(500);
        LED_Off((uint32_t)argument);
        delay(500);
    }
}
```

Quando definimos a thread, ajustamos o parâmetro *instances* para 2:

```
osThreadDef(ledSwitcher, osPriorityNormal, 2, 0);
```

Então, na *thread* principal, criamos duas *threads* que são instâncias diferentes do mesmo código base. Passamos um parâmetro diferente que corresponde ao led que será alternado pela instância da *thread*.

```
led_ID1 = osThreadCreate(osThread(ledSwitcher),(void *) 1UL);
led_ID2 = osThreadCreate(osThread(ledSwitcher),(void *) 2UL);
```

Crie o código e inicie o *debugger*

Inicie a execução do código, abra as tarefas RTX e a janela do sistema (Figura 01):

ID	Name	Priority	State
255	os_idle_demon	0	Ready
3	ledSwitcher	4	Ready
2	ledSwitcher	4	Running

Figura 01 – Controle de diferentes instâncias

Aqui podemos ver ambas as instâncias da tarefa *ledSwitcher*, cada uma com um ID diferente.

Examine a janela **Call stack** + **locals:**

Figura 02 – Parâmetros das instâncias da tarefa *ledSwitcher*

Na Figura 02 é possível ver ambas as instâncias das *threads ledSwitcher* e o estado de suas variáveis. Um argumento diferente foi passado para cada instância da *thread*.

4) Esperando por um evento

Além de um atraso de tempo puro, é possível interromper uma *thread* e entrar no estado de espera até que a *thread* seja acionada por outro evento RTOS. Os eventos RTOS podem ser um sinal, uma mensagem ou um evento de e-mail. A chamada de API osWait() também possui um

período de tempo limite definido em milissegundos que permite que a *thread* seja ativada e continue a execução se nenhum evento ocorrer.

```
osStatus osWait (uint32_t milissegundos)
```

Quando o intervalo expirar, a *thread* passará do estado de espera para o estado PRONTO e será colocado no estado de execução pelo escalonador.

* *osWait* é uma chamada de API opcional dentro da especificação CMSIS RTOS. Se você pretende usar esta função, primeiro verifique se ela é suportada pelo RTOS que você está usando. A chamada API *osWait* não é suportada pelo Keil RTX RTOS.

5) Gestão do tempo de exercício

No Pack Installer, selecione ***"Ex 5 Time Management"*** e copie-o para o diretório do seu tutorial. Este é o programa de pisca-LED original, mas a função de atraso simples foi substituída pela chamada de API ***osDelay***. LED2 é alternado a cada 100mS e LED1 é alternado a cada 500mS.

```
void ledOn (void const *argument) {
    for (;;) {
        LED_On(1);
        osDelay(500);
        LED_Off(1);
        osDelay(500);
    }
}
```

Compile o projeto e inicie o *debugger*

Inicie a execução do código e abra a janela do visualizador de eventos

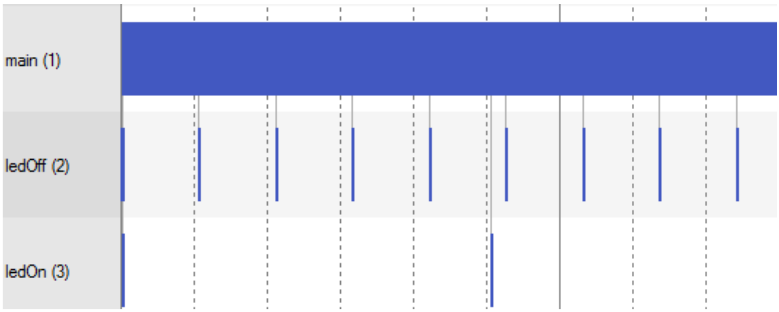

Figura 03 – Gestão de tempo de execução

Com a execução (Figura 03), é possível perceber que a atividade do código é muito diferente. Quando cada uma das tarefas de LED atinge a API ***osDelay***, a própria API bloqueia a execução e move-se para um estado de espera. A tarefa principal estará em um estado pronto para que o *scheduler* comece a executá-la. Quando o período de atraso expirar, as tarefas passarão para o estado pronto e serão colocadas no estado de execução pelo *scheduler*. Este é um programa *multi-thread* onde o tempo de execução da CPU é compartilhado de forma eficiente entre as tarefas.

6) Temporizadores virtuais

A API CMSIS-RTOS pode ser usada para definir qualquer número de cronômetros virtuais que atuam como cronômetros de contagem regressiva. Quando eles expiram, eles executam uma função de retorno de chamada do usuário para executar uma ação específica. Cada temporizador pode ser configurado como um disparo único ou um temporizador de repetição. Um cronômetro virtual é criado definindo primeiro uma estrutura de cronômetro.

```
osTimerDef(timer0,led_function);
```

Isso define um nome para o cronômetro e o nome da função de retorno de chamada. O timer deve então ser instanciado em um thread RTOS.

```
osTimerId timer0_handle = osTimerCreate (timer(timer0), osTimerPeriodic, (void *)0);
```

Isso cria o cronômetro e o define como um cronômetro periódico ou um cronômetro de disparo único (osTimerOnce). O parâmetro final passa um argumento para a função de retorno de chamada quando o cronômetro expira.

```
osTimerStart (timer0_handle,0x100);
```

O cronômetro pode então ser iniciado em qualquer ponto em uma *thread*. A função de início do cronômetro invoca o cronômetro por seu identificador e define um período de contagem em milissegundos.

7) Entendendo o código-fonte

Após realizar o estudo, a análise e a correta execução dos itens anteriores, faça:

a) Compile e execute o arquivo disponível neste link:
   1. Altere a prioridade das *threads* conforme configurações apresentadas no início deste roteiro e verifique seu funcionamento na placa.
   2. Analise as possíveis aplicações para cada tipo de prioridade.

b) Verifique as duas novas funções fornecidas no código-fonte utilizado: *ledsOn* e *ledsOff* e analise seu funcionamento. A partir disso, faça:
   1. Desative as *threads* e altere o código de modo que as duas funções sejam executadas sequencialmente e de modo contínuo na placa.
   2. Ative as *threads* e verifique o funcionamento em conjunto.

8) Exercícios complementares

8.1) Desenvolva uma função que, após um número arbitrário X de interrupções realizadas, bloqueia ou impede a execução de somente uma das funções *ledsOn* ou *ledsOff*. Execute-a na placa.

8.2) Ao invés de utilizar uma variável contador (conforme proposto no item anterior), de que modo uma interrupção de botão (*switch*) poderia bloquear ou impedir a execução de somente uma das funções *ledsOn* ou *ledsOff*?

O código-fonte com o teste de todas as funcionalidades da placa está disponível neste link.