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

Roteiro 06

Sinais de controle, rotinas de interrupção (ISR) e comunicação entre 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(pág 35 a 43).

Até agora, vimos como o código de aplicação pode ser definido como threads independentes e como podemos acessar os serviços de temporização fornecidos pelo RTOS. Em uma aplicação real, precisamos ser capazes de comunicar entre threads para tornar a aplicação útil. Para isso, um RTOS típico suporta vários objetos de comunicação diferentes que podem ser usados para ligar as threads e formar um programa significativo. A API CMSIS-RTOS suporta a comunicação entre threads com sinais, semáforos, mutexes, caixas de correio e filas de mensagens. Na primeira seção, o conceito chave foi concorrência.

1) Sinais

O CMSIS-RTOS Keil RTX suporta até dezesseis sinais para cada thread (Fig. 01). Esses sinais são armazenados no bloco de controle da thread. É possível interromper a execução de uma thread até que um sinal específico ou um grupo de sinais seja definido por outra thread no sistema.

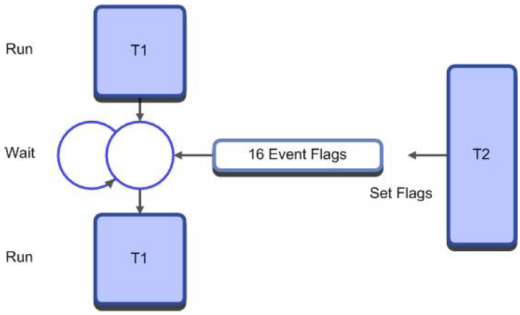


Figura 01 – Adição de arquivos

As chamadas de sistema **osSignalWait** suspendem a execução da thread e a colocam no estado **wait_evnt**. A execução da thread não começará até que todos os sinais definidos na chamada **osSignalWait** tenham sido definidos. Também é possível definir um tempo limite periódico após o qual a thread em espera voltará ao estado pronto para que possa retomar a execução quando selecionada pelo escalonador. Um valor de 0xFFFF define um período de tempo limite infinito.

```
osEvent osSignalWait (int32_t signals, uint32_t millisec);
```

Se a variável signals for definida como zero quando **osSignalWait** for chamada, qualquer sinal definido fará com que a thread retome a execução. Você pode ver qual sinal foi definido lendo o valor de retorno **osEvent.value.signals**.

Qualquer thread pode definir ou limpar um sinal em qualquer outra thread.

```
int32_t osSignalSet (osThreadId thread_id, int32_t signals);
int32_t osSignalClear (osThreadId thread_id, int32_t signals);
```

2) Exercício: Sinais

Neste exercício, vamos usar sinais para acionar atividades entre duas threads. Embora seja um programa simples, ele introduz o conceito de sincronizar a atividade das threads.

- Selecione **"Ex 8 Signals"** no *Pack Installer* e copie para o diretório do tutorial.

Código da Thread LED:

```
void led_Thread2(void const *argument) {
  for (;;) {
    LED_On(2);
    osSignalSet(T_led_ID1, 0x01);
    osDelay(500);
    LED_Off(2);
    osSignalSet(T_led_ID1, 0x01);
    osDelay(500);
  }
}

void led_Thread1(void const *argument) {
  for (;;) {
    osSignalWait(0x01, osWaitForever);
    LED_On(1);
    osSignalWait(0x01, osWaitForever);
    LED_Off(1);
  }
}
```

- Compile o projeto e inicie o debugger.
- Abra a janela do periférico GPIOB e inicie a execução do código.

Agora, os pinos da porta parecerão estar ligando e desligando juntos. Sincronizar as threads dá a ilusão de que ambas estão rodando em paralelo. Este é um exercício simples, mas ilustra o conceito chave de sincronizar a atividade entre threads em uma aplicação baseada em RTOS.

3) Tratamento de Interrupções no RTOS

O uso de sinais é um método simples e eficiente de acionar ações entre threads rodando dentro do RTOS. Os sinais são também um método importante de acionar threads do RTOS a partir de fontes de interrupção dentro do microcontrolador Cortex-M.

Embora seja possível rodar código C em uma rotina de serviço de interrupção (ISR), isso não é desejável dentro de um RTOS se o código da interrupção for rodar por mais de um curto período de tempo. Isso atrasa o *tick* do temporizador e interrompe o kernel do RTOS. O temporizador *SysTick* roda com a menor prioridade dentro do NVIC (*Nested Vectored Interrupt Controller*), então não há sobrecarga em atingir a rotina de interrupção (Figura 02).

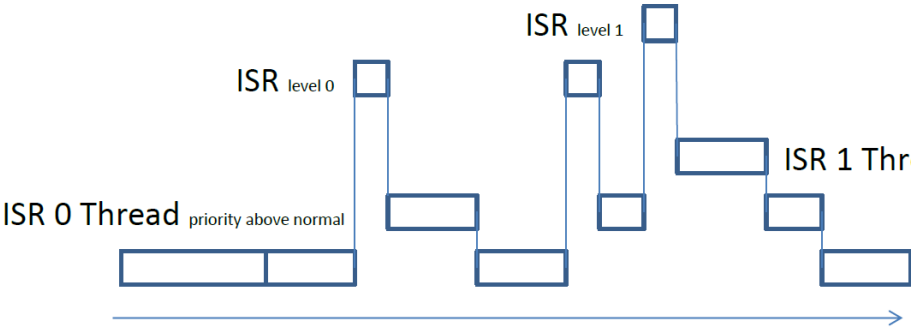


Figura 02 – Rotinas de interrupção como Threads

4) Sinais de Interrupção

Neste exercício, vamos demonstrar uma técnica de sinalizar uma thread a partir de uma interrupção e atender à interrupção do periférico com uma thread em vez de uma rotina de serviço de interrupção padrão.
No entanto, há um problema quando entramos na função `main`: o RTOS pode estar configurado para executar as threads em modo não privilegiado, de forma que não podemos acessar os registradores do NVIC sem causar uma exceção de falha. Existem várias maneiras de contornar isso. A mais simples é dar às threads acesso privilegiado alterando a configuração no arquivo `RTX_Conf_CM.c` (Fig 03).

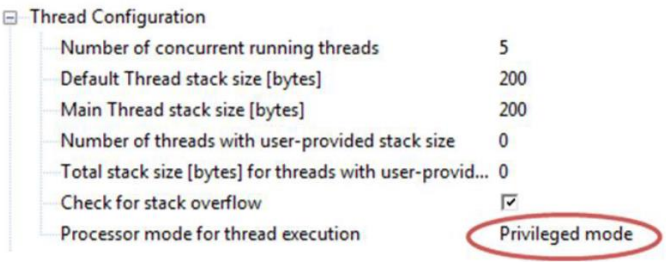


Figura 03 – RTX_Conf_CM.c

- Selecione "**Ex 9 Interrupt Signals**" no *Pack Installer* e copie para o diretório do tutorial.

Código da Thread ADC:

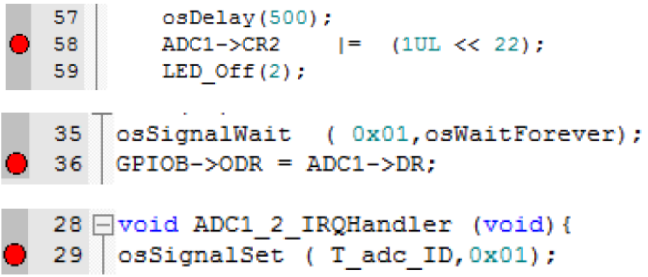
```
osThreadDef(adc_Thread, osPriorityAboveNormal, 1, 0);


int main(void) {
  LED_Init();
  init_ADC();
  T_led_ID1 = osThreadCreate(osThread(led_Thread1), NULL);
  T_led_ID2 = osThreadCreate(osThread(led_Thread2), NULL);
  T_adc_ID = osThreadCreate(osThread(adc_Thread), NULL);
}


void adc_Thread(void const *argument) {
  for (;;) {
    osSignalWait(isrSignal, osWaitForever);
    // Handle the interrupt
  }
}


void ADC1_2_IRQHandler(void) {
  osSignalSet(T_adc_ID, isrSignal);
}
```

- Compile o código e inicie o debugger.
- Defina pontos de interrupção em led_Thread2, adc_Thread e ADC1_2_IRQHandler.



- Execute o código.

  Você deve atingir o primeiro ponto de interrupção que inicia a conversão ADC, depois execute o código novamente e você deve entrar no manipulador de interrupção ADC.

  O manipulador define o sinal da thread ADC e sai.

  Definir o sinal fará com que a thread ADC preempte qualquer outra tarefa em execução, execute o código de serviço ADC e depois bloqueie aguardando o próximo sinal.

Até este momento, abordamos a comunicação entre threads utilizando sinais, a sincronização de threads LED e o tratamento de interrupções no RTOS. Os exercícios práticos fornecem uma base sólida para entender como sincronizar atividades e lidar com interrupções de maneira eficiente em um sistema baseado em RTOS.

5) Sincronização de Threads LED

A sincronização de threads é um aspecto crucial no desenvolvimento de aplicações multithread, garantindo que as threads operem de forma ordenada e livre de conflitos. Neste tópico, exploraremos a sincronização de threads utilizando sinais para controlar o piscar de LEDs.

Conceitos Básicos:

- Sinais: São mecanismos leves e eficientes para comunicação entre threads no FreeRTOS. Uma thread pode definir ou limpar sinais em outras threads, permitindo que elas se comuniquem e coordenem suas ações.
- Sincronização: Envolve a coordenação da execução de threads para garantir que elas operem de forma ordenada e consistente, evitando conflitos de acesso a recursos compartilhados.

Aplicação: Sincronizar duas threads para controlar o piscar alternado de LEDs

a) Implemente as funções led_Thread1 e led_Thread2:

```
void led_Thread1(void const *argument) {
  for (;;) {
    LED_On(1); // Liga o LED 1
    osSignalWait(0x01, osWaitForever); // Aguarda sinal
    LED_Off(1); // Desliga o LED 1
    osSignalWait(0x01, osWaitForever); // Aguarda sinal
  }
}
void led_Thread2(void const *argument) {
  for (;;) {
    LED_On(2); // Liga o LED 2
    osSignalSet(T_led_ID1, 0x01); // Define sinal para a thread 1
    osDelay(500); // Aguarda 500 ms
    LED_Off(2); // Desliga o LED 2
    osSignalSet(T_led_ID1, 0x01); // Define sinal para a thread 1
    osDelay(500); // Aguarda 500 ms
  }
}
```

b) Crie a função main:

```
int main(void) {
  LED_Init(); // Inicializa os LEDs
  T_led_ID1 = osThreadCreate(osThread(led_Thread1), NULL); // Cria a thread 1
  T_led_ID2 = osThreadCreate(osThread(led_Thread2), NULL); // Cria a thread 2
  osKernelStart(); // Inicia o RTOS
  while (1) {
    // Tarefas principais da aplicação
  }
}
```

c) Compile e execute o código.

d) Explicação do funcionamento: A thread led_Thread1 controla o LED 1 e a thread led_Thread2 controla o LED 2. Cada thread liga e desliga seu LED respectivo, mas o tempo de espera entre ligar e desligar é controlado por sinais. A thread led_Thread2 define um sinal para a thread led_Thread1 após ligar o LED 2 e após desligá-lo. A thread led_Thread1 aguarda o sinal antes de ligar ou desligar o LED 1. Essa sincronização garante que os LEDs pisquem alternadamente, sem sobreposição.

6) Exercício de aplicação

6.1) Controle de Fila de LEDs

Crie um sistema que controle uma fila de LEDs, acendendo-os sequencialmente e apagando-os após um intervalo de tempo definido. Utilize sinais para sincronizar a ação das threads responsáveis pelo acendimento e apagamento dos LEDs.

Desafios:

- Implementar uma fila de LEDs eficiente.
- Gerenciar o tempo de espera entre acender e apagar os LEDs.
- Sincronizar as threads para evitar conflitos de acesso à fila.

7) Exercícios complementares

7.1) Sincronização de Sensores e Atuadores

Desenvolva um sistema que leia dados de um sensor e os utilize para controlar um atuador. Utilize sinais para sincronizar a leitura dos dados do sensor com a acionamento do atuador.

Desafios:

- Implementar a interface com o sensor e o atuador.
- Processar os dados do sensor de forma eficiente.
- Sincronizar a leitura e o acionamento para evitar inconsistências.

7.2) Controle de Semáforo com Sinais e Temporizador

Simule o funcionamento de um semáforo (de trânsito) utilizando sinais e um temporizador. As threads responsáveis pelo controle dos LEDs verde, amarelo e vermelho devem ser sincronizadas através de sinais e do temporizador para garantir o ciclo semafórico correto.

Desafios:

- Implementar o ciclo semafórico com tempos precisos.
- Sincronizar as threads de forma eficiente para evitar conflitos.
- Gerenciar o tempo de cada fase do semáforo (de trânsito).