LABORATÓRIO RTOS

Prática 03: Estados e Configurações de Tasks

Prof. Francisco Helder

16 de maio de 2025

1 Prioridades da Tasks

O parâmetro uxPriority da função API **xTaskCreate()** atribui uma prioridade inicial à Task que está sendo criada. A prioridade pode ser alterada após o scheduler ter sido iniciado usando a função da API **vTaskPrioritySet()**.

O número máximo de prioridades disponíveis é definido pela constante de configuração em tempo de compilação configMAX_PRIORITIES definida pelo aplicativo em FreeRTOSConfig.h. O próprio FreeRTOS não limita o valor máximo que esta constante pode assumir, mas quanto maior o valor de configMAX_PRIORITIES mais RAM o kernel consumirá, por isso é sempre aconselhável manter o valor definido no mínimo necessário.

O FreeRTOS não impõe nenhuma restrição sobre como as prioridades podem ser atribuídas às Tasks. Qualquer número de Tasks pode compartilhar a mesma prioridade – garantindo máxima flexibilidade de projeto. Você pode atribuir uma prioridade exclusiva a cada Task se isso for desejável (conforme exigido por alguns algoritmos de capacidade de schedule), mas essa restrição não é imposta de forma alguma.

Valores de prioridade numérica baixa denotam Tasks de baixa prioridade, sendo a prioridade 0 a prioridade mais baixa possível. A faixa de prioridades disponíveis é, portanto, de 0 a (configMAX_PRIORITIES – 1). O scheduler sempre garantirá que a Task de prioridade mais alta capaz de ser executada seja a Task selecionada para entrar no estado RUNNING. Onde mais de uma Task da mesma prioridade for capaz de ser executada, o scheduler fará a transição de cada Task para dentro e para fora do estado RUNNING, por sua vez. Este é o comportamento observado nos exemplos até agora, onde ambas as Tasks de teste foram criadas com a mesma prioridade e ambas sempre puderam ser executadas. Cada Task é executada por um "período de tempo", ela entra no estado RUNNING no início do intervalo de tempo e sai do estado RUNNING no final do intervalo de tempo. Na Figura 1, o tempo entre t1 e t2 é igual a um único intervalo de tempo.

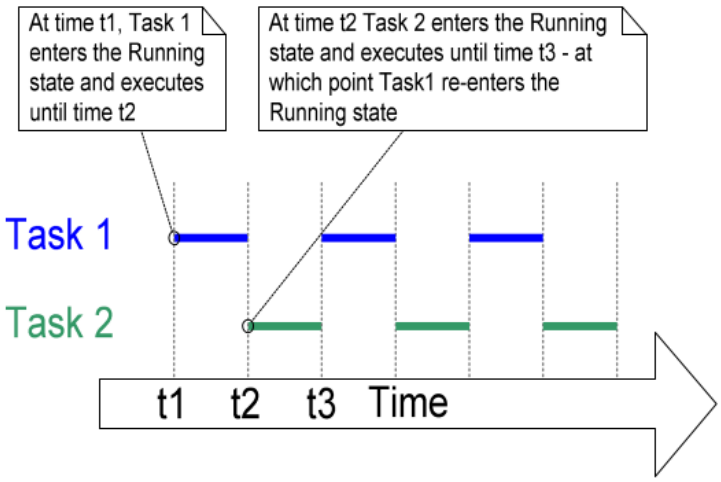

Figura 1: Sequência RUNNING de duas Tasks.

Para poder selecionar a próxima Task a ser executada, o próprio scheduler deve ser executado no final de cada intervalo de tempo. Uma interrupção periódica chamada interrupção de tick é usada para esse propósito. A duração do intervalo de tempo é efetivamente definida pela frequência de interrupção do tick que é configurada pela constante de configuração em tempo de compilação configTICK_RATE_HZ em FreeRTOSConfig.h. Por exemplo, se configTICK_RATE_HZ estiver definido como $100\,\mathrm{Hz}$, o intervalo de tempo será de $10\,\mathrm{ms}$. A Figura 1

pode ser expandida para mostrar a execução do próprio scheduler na sequência RUNNING. Isso é mostrado na Figura 2.

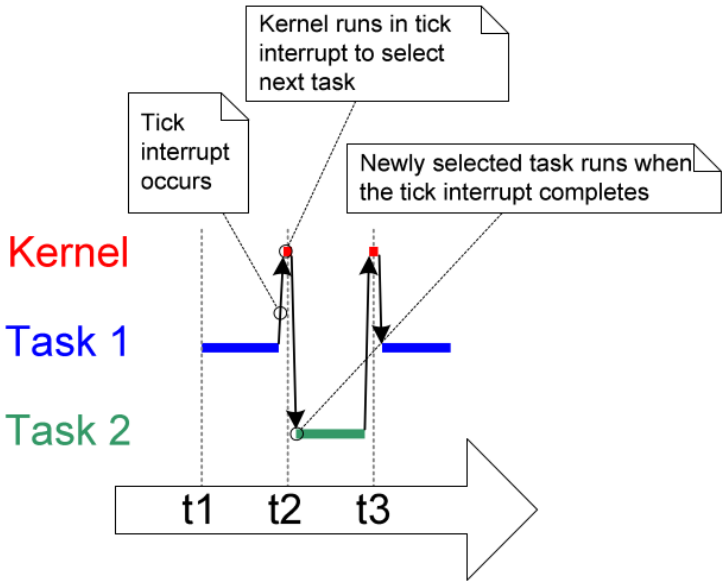

Figura 2: A sequência RUNNING foi expandida para mostrar a interrupção do tick em execução.

Observe que as chamadas da API FreeRTOS sempre especificam o tempo nas interrupções de tick (comumente chamadas apenas de 'ticks'). A constante portTICK_RATE_MS é fornecida para permitir que os atrasos sejam convertidos do número de interrupções de tick em milissegundos. A resolução disponível depende da frequência do tick. O valor de 'contagem de ticks' é o número total de interrupções de ticks que ocorreram desde que o agendador foi iniciado; assumindo que a contagem de ticks não excedeu. Os aplicativos do usuário não precisam considerar overflows ao especificar períodos de atraso, pois a consistência do tempo é gerenciada internamente pelo kernel.

1.1 Trabalhando com Prioridades

O escalonador sempre garantirá que a Task de prioridade mais alta capaz de ser executada seja a Task selecionada para entrar no estado RUNNING. Em nossas práticas até agora, duas Tasks foram criadas com a mesma prioridade, portanto, ambas entraram e saíram do estado RUNNING. Agora vamos analisar o que acontece quando alteramos a prioridade de uma das duas Tasks criadas. Desta vez a primeira Task será criada na prioridade 1 e a segunda na prioridade 2.

pratica 1:

Implemente e avalie a criação e Tasks com prioridades diferentes

2 Expandindo o Estado Not Running

Até o momento, as Tasks criadas sempre tiveram processamento a ser executado e nunca precisaram esperar por nada – e, como nunca precisaram esperar por nada, sempre puderam

entrar no estado RUNNING. Essas Tasks de "processamento contínuo" têm utilidade limitada, pois só podem ser criadas com a prioridade mais baixa. Se forem executadas em qualquer outra prioridade, impedirão que Tasks de menor prioridade sejam executadas.

Para tornar essas Tasks úteis, elas devem ser reescritas para serem orientadas a eventos. Uma Task orientada a eventos só tem trabalho (processamento) a ser executado após ser acionada por um evento e não pode entrar no estado RUNNING antes desse momento. O escalonador sempre seleciona a Task de maior prioridade que pode ser executada. Se uma Task de alta prioridade não puder ser selecionada porque está aguardando um evento, o escalonador deve, em vez disso, selecionar uma Task de menor prioridade que possa ser executada. Portanto, escrever Tasks orientadas a eventos significa que Tasks podem ser criadas com prioridades diferentes sem que as Tasks de maior prioridade privem todas as Tasks de menor prioridade de tempo de processamento.

2.1 Estado Blocked

Diz-se que uma Task aguardando um evento está no estado BLOCKED, um subestado do estado NOT RUNNING. As Tasks podem entrar no estado BLOCKED para aguardar dois tipos diferentes de eventos:

- Eventos temporais (relacionados ao tempo) – esses eventos ocorrem quando um período de atraso expira ou um tempo absoluto é atingido. Por exemplo, uma Task pode entrar no estado BLOCKED para aguardar 10 milissegundos.

- Eventos de sincronização – esses eventos se originam de outra Task ou interrupção. Por exemplo, uma Task pode entrar no estado BLOCKED para aguardar a chegada de dados em uma fila. Eventos de sincronização abrangem uma ampla gama de tipos de eventos.

Filas do FreeRTOS, semáforos binários, semáforos de contagem, mutexes, mutexes recursivos, grupos de eventos, buffers de fluxo, buffers de mensagem e notificações diretas para Tasks podem criar eventos de sincronização.

Uma Task pode bloquear um evento de sincronização com um tempo limite, bloqueando efetivamente ambos os tipos de eventos simultaneamente. Por exemplo, uma Task pode optar por aguardar no máximo 10 milissegundos para que os dados cheguem a uma fila. A Task sairá do estado BLOCKED se os dados chegarem em até 10 milissegundos ou se passarem 10 milissegundos sem que os dados cheguem.

2.2 Estado Supended

Suspend também é um subestado de NOT RUNNING. Tasks no estado Suspend não estão disponíveis para o escalonador. A única maneira de entrar no estado SUSPEND é por meio de uma chamada à função da API **vTaskSuspend()**, e a única maneira de sair é por meio de uma chamada às funções da API **vTaskResume()** ou **xTaskResumeFromISR()**. A maioria dos aplicativos não utiliza o estado Suspenso.

2.3 Estado Ready

Tasks que estão no estado NOT RUNNING e não estão Bloqueadas ou Suspensas são consideradas no estado READY. Elas podem ser executadas e, portanto, estão prontas para execução, mas não estão atualmente no estado RUNNING.

A Figura 3 expande o diagrama de estados simplificado para incluir todos os subestados NOT RUNNING, READY, BLOCKED e SUSPEND.

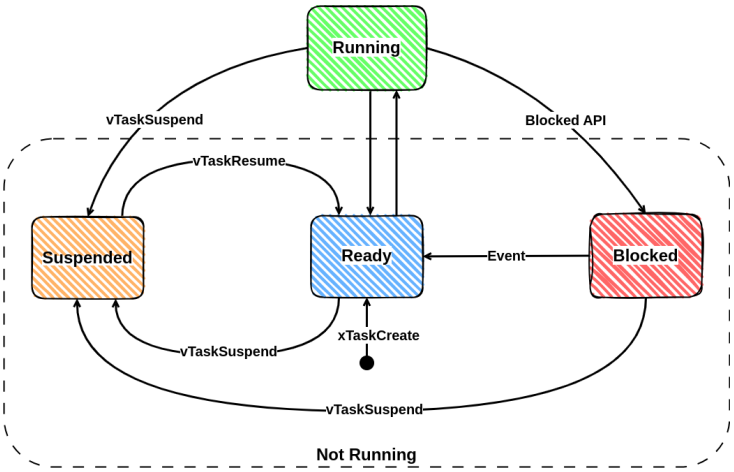

Figura 3: Máquina de estado completa de um RTOS.

2.4 Trabalhabndo com API Delay

Nesta atividade aprenderemos a trabalhar com as rotinas de delay do FreeRTOS. Remova a rotina de delay e implemente a Task que pisca o led usando a API de delay do FreeRTOS:

```
1 void taskLed(void *pvParameters){
2   unsigned int led = (unsigned int) pvParameters;
3   for (;;) {
4     drvLedsSet(led, DRV_LEDS_TOGGLE);
5     vTaskDelay(500/portTICK_RATE_MS);
6   }
7 }
```

pratica 2:

Compile e teste. Perceba que agora as rotinas de delay são muito mais precisas que as rotinas baseadas em polling usadas anteriormente. Confirme via oscilosópio a frequência do pisca LED.

2.5 Task IDLE e Iniciando a Task IDLE HOOK

O processador sempre precisa de algo para executar – sempre deve haver pelo menos uma Task que possa entrar no estado RUNNING. Para garantir que este seja o caso, uma Task IDLE é criada automaticamente pelo scheduler quando **vTaskStartScheduler()** é chamado. A Task IDLE faz muito pouco mais do que ficar em loop – assim como as Tasks nos exemplos originais, ela sempre pode ser executada.

A Task IDLE tem a prioridade mais baixa possível (prioridade 0) para garantir que nunca impeça que uma Task de aplicativo de prioridade mais alta entre no estado RUNNING.

A execução na prioridade mais baixa garante que a Task IDLE sairá imediatamente do estado RUNNING assim que uma Task de prioridade mais alta entrar no estado READY.

2.5.1 Implementando a Função IDLE HOOK

Nesta atividade iremos medir o uso da CPU através da função IDLE HOOK. Mediremos a quantidade de ticks que a aplicação fica na função IDLE, para então comparar com a quantidade total de ticks da aplicação, calculando a porcentagem de uso da CPU.

Implemente a função IDLE HOOK para medir o uso da CPU, primeiro habilite a opção configUSE_IDLE_HOOK no arquivo de configuração FreeRTOSConfig.h. Crie uma variável global para armazenar a quantidade total de ticks e implemente a função IDLE HOOK no seu código.

```
static unsigned long int idle_tick_counter = 0;

...

void vApplicationIdleHook(void){
  unsigned long int tick = xTaskGetTickCount();
  while (xTaskGetTickCount() == tick);
  idle_tick_counter++;
}
```

A função IDLE HOOK irá contar a quantidade de ticks durante a execução da Task IDLE e armazenar na variável idle_tick_counter, então implemente a Task que irá calcular o uso da CPU, e com isso os valores serão impressos no terminal.

```
void taskCPUUsage(void *pvParameters){

  unsigned long int idle_tick_last, ticks;
  idle_tick_last = idle_tick_counter = 0;

  for(;;){
    /* wait for 3 seconds */
    vTaskDelay(3000/portTICK_RATE_MS);

    /* calculate quantity of idle ticks per second */
    if (idle_tick_counter > idle_tick_last)
      ticks = idle_tick_counter - idle_tick_last;
    else
      ticks = 0xFFFFFFFF - idle_tick_last + idle_tick_counter;

    ticks /= 3;

    /* print idle ticks per second */
    printf("%ld idle ticks per second (out of %ld)\n", ticks, configTICK_RATE_HZ);

    /* calc and print CPU usage */
    ticks = (configTICK_RATE_HZ - ticks)/10;
    printf("CPU usage: %d%%\n", ticks);

    /* update idle ticks */
    idle_tick_last = idle_tick_counter;
  }
}
```

pratica 3:

Execute seu código anterior com dois LEDs (adicionado do medidor de CPU) e mostre o uso da CPU para sues testes.