

# LABORATÓRIO RTOS



---

## Prática 02: Gerenciamento de Tasks

---

Prof. Francisco Helder

30 de abril de 2025

# 1 Aplicação baseada em Tasks

## 1.1 Função Task

As Tasks são implementadas como funções em C, a única coisa especial sobre eles é seu protótipo, que deve retornar void e receber um parâmetro de ponteiro void, como é demonstrado na Figura 1.

```
void ATaskFunction( void *pvParameters );
```

Figura 1: Protótipo da função que será uma Task

Cada Task é um pequeno programa por si só. Ele tem um ponto de entrada, normalmente rodará para sempre dentro de um loop infinito e não sairá. A estrutura de uma Task típica é mostrada na Figura 2.

As Tasks do FreeRTOS não devem ter permissão para retornar de sua função de forma alguma – elas não devem conter uma instrução ‘return’ e não devem ter permissão para serem executadas após o final da função. Se uma Task não for mais necessária, ela deverá ser explicitamente excluída. Isto também é demonstrado na Figura 2.

Uma única definição de função Task pode ser usada para criar qualquer número de Tasks – cada Task criada sendo uma instância de execução separada com sua própria pilha e sua própria cópia de quaisquer variáveis automáticas (pilha). definido dentro da própria Task.

```
1 void ATaskFunction( void *pvParameters ){
2     /* Variables can be declared just as per a normal function. Each instance
3     of a task created using this function will have its own copy of the
4     iVariableExample variable. This would not be true if the variable was
5     declared static - in which case only one copy of the variable would exist
6     and this copy would be shared by each created instance of the task. */
7     int iVariableExample = 0;
8
9     /* A task will normally be implemented as an infinite loop. */
10    for( ;; ){
11        /* The code to implement the task functionality will go here. */
12    }
13
14    /* Should the task implementation ever break out of the above loop
15    then the task must be deleted before reaching the end of this function.
16    The NULL parameter passed to the vTaskDelete() function indicates that
17    the task to be deleted is the calling (this) task. */
18    vTaskDelete( NULL );
19 }
```

Figura 2: A estrutura típica da função Task

## 1.2 Estados da Task de Nível Superior

Um aplicativo pode consistir em muitas Tasks. Se o microcontrolador que executa o aplicativo contém apenas um único núcleo, apenas uma Task pode realmente estar em execução a qualquer momento. Isso implica que uma Task pode existir em um dos dois estados, **Running** e **Not Running**. Consideraremos este modelo simplista primeiro – mas tenha em mente que esta é uma simplificação excessiva, pois mais tarde veremos que o estado Not Running contém, na verdade, vários subestados.

Quando uma Task está no estado Running, o processador está na verdade executando seu código. Quando uma Task está no estado Not Running, a Task fica inativa, seu status foi salvo,

pronto para retomar a execução na próxima vez que o escalonador decidir que deve entrar no estado Running. Quando uma Task retoma a Running, ela o faz exatamente a partir da instrução que estava prestes a executar antes de sair do estado Running pela última vez (como visto na Figura 3).

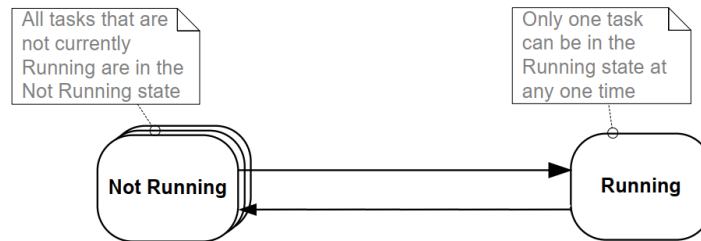


Figura 3: Estados e transições das Tasks de nível superior.

### 1.3 Criando Tasks

As Tasks são criadas usando a API **xTaskCreate()** do FreeRTOS. Esta é provavelmente a mais complexa de todas as funções da API, por isso é lamentável que seja a primeira encontrada, mas as Tasks devem ser dominadas primeiro, pois é o componente mais fundamental de um sistema multiTask.

```
1 portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,  
2     const signed portCHAR * const pcName,  
3     unsigned portSHORT usStackDepth,  
4     void *pvParameters,  
5     unsigned portBASE_TYPE uxPriority,  
6     xTaskHandle *pxCreatedTask  
7     );
```

Figura 4: Protótipo da função xTaskCreate()

### 1.4 Exemplo de criação de Task

Nessa atividade iremos desenvolver nossa primeira aplicação baseada em Tasks. Vamos escrever duas Tasks, cada uma responsável por enviar mensagens de textos via USB. Aproveite o projeto que você criou para gerar uma imagem do FreeRTOS, altere o conteúdo do arquivo main.c e insira os includes dos arquivos de cabeçalho do kernel FreeRTOS e Raspberry Pico, além da rotina de delay que utilizaremos nas Tasks:

```
1 #include "FreeRTOS.h"  
2 #include "task.h"  
3 #include <stdio.h>  
4 #include "pico/stdlib.h"  
5  
6 #define TICK_PERIOD_MS 2000000  
7 #define mainRUN_ON_CORE 0  
8  
9 void delay() {  
10     unsigned int i;  
11     for(i = 0; i < TICK_PERIOD_MS; i++)  
12         asm("nop");  
13 }
```

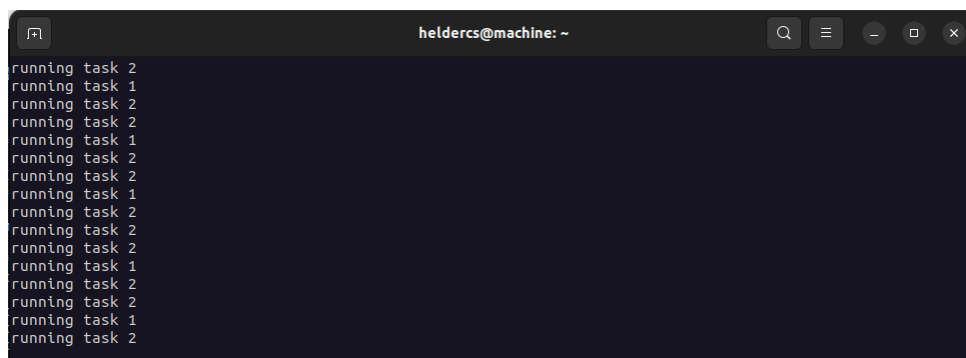
Agora insira a implementação das Tasks:

```
1 void task1(void *pvParameters){
2     /* Remove compiler warning about unused parameter. */
3     (void) pvParameters;
4     for (;;) {
5         delay();
6         printf("running_task_1\n");
7         delay();
8     }
9 }
10 void task2(void *pvParameters){
11     /* Remove compiler warning about unused parameter. */
12     (void) pvParameters;
13     for (;;) {
14         printf("running_task_2\n");
15         delay();
16     }
17 }
```

E por último, implemente a função main():

```
1 int main(void){
2     printf("_Creating_task!!!\n");
3
4     /* create task 1 */
5     xTaskCreate(task1, (signed char *)"Task1",
6                 configMINIMAL_STACK_SIZE,
7                 (void *) NULL, 1, NULL);
8
9     /* create task 2 */
10    xTaskCreate(task2, (signed char *)"Task2",
11                configMINIMAL_STACK_SIZE,
12                (void *) NULL, 1, NULL);
13
14    /* start the scheduler */
15    vTaskStartScheduler();
16
17    for(;;); /* should never reach here! */
18 }
```

Compile e teste o funcionamento do FreeRTOS. A execução do exemplo deve gerar um resultado como mostrado na Figura 5.



```
heldercs@machine: ~
running task 2
running task 1
running task 2
running task 2
running task 1
running task 2
running task 2
running task 1
running task 2
running task 2
running task 1
running task 2
running task 2
running task 1
running task 2
running task 2
running task 1
running task 2
running task 1
running task 2
```

Figura 5: Executando o exemplo de com duas Tasks.

## 1.5 Atividades práticas

### pratica 1:

Nesta prática você deve criar etapas necessárias para criar duas Tasks simples e piscar um LED cada Task.

Nas Tasks apenas envie sinal pelo GPIO, usando um loop de delay para criar o atraso do período. Ambas as Tasks são criadas com a mesma prioridade e são idênticas, exceto pela saída do GPIO, pois usam pinos de GPIO diferentes.

Como as duas Tasks tem o mesmo objetivo (piscar um led), altere-as para compartilharem o mesmo código da Task. Para isso, a aplicação deverá passar como parâmetro para a função **xTaskCreate()** o led a ser piscado.

Crie uma outra Task para piscar um dos leds apenas três vezes e finalizar sua execução. Neste caso, a Task deverá chamar no fim a função **vTaskDelete()**.

Não esqueça de verificar se a opção **INCLUDE\_vTaskDelete** esta habilitada no arquivo de configuração **FreeRTOSConfig.h**.

Ambas as Tasks estão sendo executadas com a mesma prioridade, portanto, compartilhe o tempo no processador, você deve apresentar o padrão de execução real como mostrado na Figura 6.

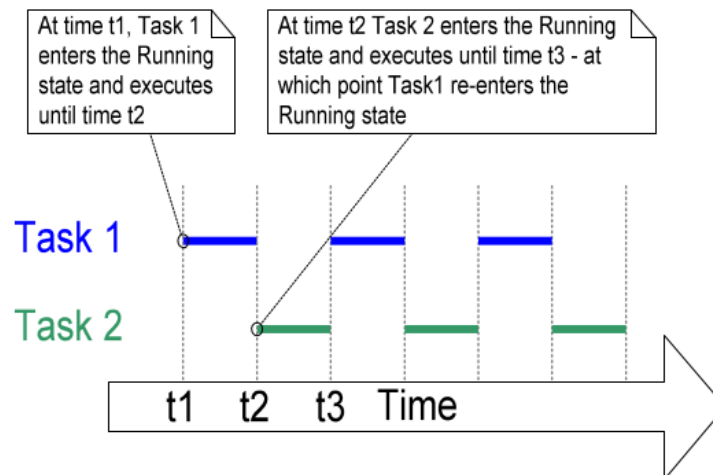


Figura 6: O padrão de execução real das duas Tasks que você deve apresentar

### pratica 2:

Agora altere a função **Delay** para uma API do FreeRTOS, que não use o processador quando não for fazer nada.