Dispositivos Conectados
Familiarização com o FreeRTOS

2025/2026

Authors: Paulo C. Bartolomeu

1 Objetivos

- Familiarização com o FreeRTOS
- Criação de tarefas e manipulação de parâmetros
- Estabelecimento de comunicações entre tarefas através de notificações e Mutexes

2 Introdução ao FreeRTOS

Um sistema operativo de tempo-real (*Real-Time Operating System* - RTOS) fornece um *scheduler* de tarefas determinístico. Embora as regras de agendamento de tarefas mudem dependendo do algoritmo selecionado, sabemos que a tarefa que criamos será concluída num determinado período de tempo dentro dessas regras. As principais vantagens de usar um RTOS são a redução da complexidade e a melhoria da arquitetura de software que facilita a respetiva manutenção.

O principal sistema operativo de tempo real suportado pelo ESP-IDF é o FreeRTOS. O ESP-IDF usa a sua própria versão do FreeRTOS portada para RISC-V (e Xtensa). A diferença fundamental em comparação com o FreeRTOS básico é o suporte dual-core. No ESP-IDF FreeRTOS, caso o processador possua dois núcleos (Xtensa), pode escolher a qual deles atribuir uma tarefa ou pode deixar o FreeRTOS escolhê-lo. Outras diferenças em comparação com o FreeRTOS original decorrem principalmente do suporte dual-core. O FreeRTOS é distribuído sob a licença MIT. Pode encontrar a documentação do ESP-IDF FreeRTOS neste URL.

2.1 O Kernel

O FreeRTOS corresponde a um conjunto de bibliotecas C composta por um *kernel* de tempo-real e um conjunto de bibliotecas modulares que implementam funcionalidades complementares.

O *kernel* FreeRTOS é ideal para aplicações de tempo-real *embedded* que operam em microcontroladores ou pequenos microprocessadores. Este tipo de aplicação inclui normalmente uma combinação de requisitos de tempo-real rígidos e flexíveis. Os requisitos de tempo-real flexíveis estabelecem um prazo, mas o incumprimento do prazo não tornaria o sistema inútil. Por exemplo, responder demasiado lentamente às teclas pressionadas pode fazer com que um sistema pareça irritantemente pouco responsivo, sem o tornar realmente inutilizável.

Os requisitos de tempo-real rígidos estabelecem um prazo, e o não cumprimento desse prazo resultaria na falha absoluta do sistema. Por exemplo, o *airbag* de um condutor tem o potencial de causar mais

danos do que benefícios se responder muito lentamente aos sinais do sensor de colisão.

O *kernel* FreeRTOS é um kernel de tempo-real (ou scheduler de tempo-real) que permite que as aplicações criadas no FreeRTOS atendam aos seus requisitos rígidos em tempo-real. O *kernel* permite que as aplicações sejam organizadas como uma coleção de *threads* de execução independentes. Por exemplo, num processador que tem apenas um núcleo, apenas um único *thread* de execução pode ser executado por vez. O *kernel* decide qual *thread* executar, examinando a prioridade atribuída a cada *thread* pelo *developer* da aplicação. No caso mais simples, o designer poderia atribuir prioridades mais altas às *threads* que implementam requisitos rígidos de tempo-real e prioridades mais baixas às *threads* que implementam requisitos flexíveis em tempo-real. A alocação de prioridades dessa forma garantiria que as *threads* rígidas em tempo-real fossem sempre executadas antes das *threads* flexíveis.

2.2 Porquê usar um sistema operativo de tempo-real?

Existem muitas técnicas bem estabelecidas para escrever bom *embedded software* sem usar um *kernel multithreading*. Se o sistema a desenvolver for simples, essas técnicas podem fornecer a solução mais adequada. Usar um *kernel* provavelmente será preferível em casos mais complexos, mas onde ocorre o ponto de cruzamento será sempre subjetivo.

A priorização de tarefas pode ajudar a garantir que uma aplicação cumpre os seus prazos de processamento, mas um *kernel* pode trazer outros benefícios menos óbvios. Alguns deles estão listados resumidamente abaixo.

- **Abstração das informações de tempo**: O RTOS é responsável pelo tempo de execução e fornece uma API relacionada ao tempo para a aplicação. Isso permite que a estrutura do código da aplicação seja mais direta e que o tamanho geral do código seja menor.
- **Manutenção/Extensibilidade**: abstração dos detalhes de temporização resulta em menos interdependências entre os módulos e permite que o software evolua de forma controlada e previsível. Além disso, o *kernel* é responsável pela temporização, portanto, o desempenho da aplicação é menos suscetível a alterações no *hardware* subjacente.
- **Modularidade**: As tarefas são módulos independentes, cada um dos quais deve ter um objetivo bem definido.
- **Desenvolvimento em equipa**: As tarefas também devem ter interfaces bem definidas, permitindo um desenvolvimento em equipa mais fácil.
- **Testes mais fáceis**:Tarefas que são módulos independentes bem definidos com interfaces limpas são mais fáceis de testar isoladamente.
- **Reutilização de código**: O código projetado com maior modularidade e menos interdependências é mais fácil de reutilizar.
- **Maior eficiência**: O código da aplicação que usa um RTOS pode ser totalmente orientado a eventos. Não é necessário desperdiçar tempo de processamento a fazer *polling* a eventos que não ocorreram. Opondo-se à eficiência obtida pela orientação a eventos, há a necessidade de processar a interrupção do RTOS e alternar a execução entre tarefas. No entanto, as aplicações que não utilizam um RTOS normalmente incluem alguma forma de interrupção de qualquer forma.
- **Tempo *Idle***: A tarefa *idle* criada automaticamente é executada quando não há tarefas da aplicação que exijam processamento. A tarefa *idle* pode medir a capacidade de processamento

disponível, realizar verificações em segundo plano ou colocar o processador em modo de baixo consumo de energia.

- **Gestão de energia**: Os ganhos de eficiência resultantes da utilização de um RTOS permitem que o processador passe mais tempo em modo de baixo consumo de energia. O consumo de energia pode ser reduzido significativamente colocando o processador num estado de baixo consumo de energia cada vez que a tarefa *idle* é executada. O FreeRTOS também possui um modo especial sem *tick*. A utilização do modo sem *tick* permite que o processador entre num modo de menor consumo de energia do que seria possível de outra forma e permaneça nesse modo por mais tempo.
- **Tratamento flexível de interrupções**: Os manipuladores (*handlers*) de interrupções podem ser mantidos muito curtos, adiando o processamento para uma tarefa criada pelo programador da aplicação ou para a tarefa *daemon* RTOS criada automaticamente (também conhecida como *timer task*).
- **Requisitos de processamento mistos**: Padrões de design simples podem alcançar uma mistura de processamento periódico, contínuo e orientado a eventos dentro de uma aplicação. Além disso, os requisitos de tempo real rígidos e flexíveis podem ser atendidos selecionando as prioridades apropriadas de tarefas e interrupções.

3 Conceitos Fundamentais

O FreeRTOS pode ser compilado com aproximadamente vinte compiladores diferentes e pode ser executado em mais de quarenta arquiteturas de processador diferentes. Cada combinação suportada de compilador e processador corresponde a um porto do FreeRTOS.

3.1 Tipos de dados

Cada porto do FreeRTOS possui um ficheiro de cabeçalho `portmacro.h` exclusivo que contém (entre outras coisas) definições para dois tipos de dados específicos do porto: `TickType_t` e `BaseType_t`.

3.1.1 `TickType_t`

O FreeRTOS configura uma interrupção periódica chamada interrupção de *tick*. O número de interrupções de *tick* que ocorreram desde o arranque da aplicação FreeRTOS é chamado de contagem de *ticks*. Esta contagem é usada como uma medida de tempo.

O tempo entre duas interrupções de *tick* é designado por período de *tick* e normalmente os tempos são especificados como múltiplos de períodos de *tick*.

`TickType_t` é o tipo de dados utilizado para armazenar o valor da contagem de *ticks* e para especificar tempos. `TickType_t` pode ser um tipo sem sinal de 16 bits, um tipo sem sinal de 32 bits ou um tipo sem sinal de 64 bits, dependendo da configuração de `configTICK_TYPE_WIDTH_IN_BITS` no ficheiro `FreeRTOSConfig.h`.

3.1.2 `BaseType_t`

É sempre definido como o tipo de dados mais eficiente para a arquitetura. Normalmente, é um tipo de 64 bits numa arquitetura de 64 bits, um tipo de 32 bits numa arquitetura de 32 bits, um tipo de 16 bits numa arquitetura de 16 bits e um tipo de 8 bits numa arquitetura de 8 bits.

O `BaseType_t` é geralmente usado para tipos de retorno que aceitam apenas uma gama muito limitada de valores e para booleanos do tipo `pdTRUE`/`pdFALSE`.

3.2 Estilo de programação

A adopção de um estilo de programação conpotencia um código mais legível e consistente, facilitando a portabilidade entre variantes da família e reduzindo erros em tarefas concorrentes, tornando o desenvolvimento de aplicações embebidas mais robusto e escalável.

3.2.1 Nomes de variáveis

As variáveis são prefixadas com o seu tipo: 'c' para `char`, 's' para `int16_t` (*short*), 'l' para `int32_t` (*long*) e 'x' para `BaseType_t` e quaisquer outros tipos não padrão (estruturas, identificadores de tarefas, identificadores de filas, etc.).

Se uma variável não tiver sinal, terá também o prefixo 'u'. Se uma variável for um ponteiro, terá também terá o prefixo 'p'. Por exemplo, uma variável do tipo `uint8_t` terá o prefixo 'uc' e uma variável do tipo ponteiro para `char` (`char` *) terá o prefixo 'pc'.

3.2.2 Nomes de funções

As funções são prefixadas com o tipo que retornam e o ficheiro em que estão definidas. Por exemplo:

- `vTaskPrioritySet()` retorna um `void` e está definida em `tasks.c`.
- `xQueueReceive()` retorna uma variável do tipo `BaseType_t` e é definida em `queue.c`.
- `pvTimerGetTimerID()` retorna um ponteiro para `void` e é definida em `timers.c`.

Funções de âmbito de ficheiro (privadas) são prefixadas com 'prv'.

3.2.3 Macros

A maioria das macros é escrita em maiúsculas e precedida por letras minúsculas que indicam onde a macro está definida.

Prefix	Location of macro definition
port (for example, portMAX_DELAY)	portable.h or portmacro.h
task (for example, taskENTER_CRITICAL())	task.h
pd (for example, pdTRUE)	projdefs.h
config (for example, configUSE_PREEMPTION)	FreeRTOSConfig.h
err (for example, errQUEUE_FULL)	projdefs.h

Para aceder a informação mais pormenorizada, pode consultar o livro *Mastering the FreeRTOS Real Time Kernel* ou Manual de Referência FreeRTOS disponíveis online.

4 Trabalho Prático

O trabalho prático desta aula desenvolve-se com base no livro *Mastering the FreeRTOS Real Time Kernel*, concretamente dos capítulos 4 - "Task Management", 8 - "Resource Management" e 10 - "Task Notifications". Recomenda-se ter a versão PDF do livro para consulta ao longo do guião.

4.1 Gestão de tarefas

As tarefas do FreeRTOS são a espinha dorsal do FreeRTOS. Pode-se pensar em cada tarefa como uma aplicação individual em execução dentro da sua aplicação como um todo. Isso torna muito fácil controlar o que está a acontecer na sua aplicação e segregar a lógica, de modo que tarefas individuais sejam responsáveis por determinadas secções da sua aplicação.

O FreeRTOS é responsável por agendar as tarefas a serem executadas de forma que ambas ou mais de uma tarefa possam ser executadas ao mesmo tempo.

Nesta secção, vamos examinar as tarefas do FreeRTOS e como elas nos ajudam a estruturar o nosso código para que seja fácil escrever programas modulares.

Em seguida é facultado o "esqueleto" de um programa que contém duas funções. A tarefa 1 faz algo como simular a leitura de temperaturas e a tarefa 2 simula a leitura de humidades, estando ambas num *loop while* infinito. Neste exemplo, pretende-se ler a temperatura a cada segundo e, ao mesmo tempo, ler a humidade a cada dois segundos.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void task1()
{
  while (true)
  {
    printf("reading temperature \n");
    vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}

void task2()
{
  while (true)
  {
    printf("reading humidity \n");
    vTaskDelay(2000 / portTICK_PERIOD_MS);
  }
}

void app_main(void)
{
   task1();
   task2();
}
```

Prosseguindo com a compilação e execução deste código verificará que a leitura da humidade nunca é chamada.

> **QUESTÃO 1:** Porque motivo nunca é invocada a leitura da humidade?

O FreeRTOS foi projetado de forma a que, quando o `VTaskDelay` é chamado e enquanto essa tarefa específica está em espera, o FreeRTOS procura quaisquer outras tarefas que estejam ativas ou precisem ser acionadas e executa-as também.

> **QUESTÃO 2:** Estude o capítulo 4, concretamente as funções de criação de tarefas, e altere o programa fornecido de modo a que as tarefas se executem de acordo com o esperado.

Explore as diferentes opções de criação de tarefas que estão disponíveis no FreeRTOS.

QUESTÃO 3: Considere que se quer identificar tarefa que está a ser executada passando essa informação como argumento de entrada na criação da *thread*. Altere o programa de modo a que se obtenha um resultado semelhante ao seguinte:

```
~$ reading temperature from task1
~$ reading temperature from task1
~$ reading humidity from task2
~$ reading temperature from task1
~$ reading temperature from task1
```

QUESTÃO 4: Considerando o exemplo anterior, crie tarefas adicionais e experimente com diferentes períodos e prioridades. Consegue encontrar um cenário em que uma das tarefa é sempre bloqueada?

QUESTÃO 5: A função `xTaskCreatePinnedToCore` permite seleccionar o núcleo em que uma tarefa pode ser executada. Considerando um *target* ESP32-C6, qual a vantagem relativamente à função `xTaskCreate`?

4.2 Comunicação entre tarefas

4.2.1 Notificações

Nesta seção, vamos analisar as notificações de tarefas. As notificações de tarefas são uma das formas mais simples de fazer com que uma tarefa, por exemplo, um remetente, notifique algo a um destinatário.

As notificações de tarefas exigem que tenhamos um *handler* nas tarefas criadas. Neste caso, vamos criar um *handler* no destinatário, que o remetente poderá utilizar para notificar.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

static TaskHandle_t receiverHandler = NULL;

void sender(void * params)
{
    while (true)
    {
      xTaskNotifyGive(receiverHandler);
      vTaskDelay(5000 / portTICK_PERIOD_MS);
    }

}

void receiver(void * params)
{
  while (true)
  {
     ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
     printf("received notification\n", count);
  }

}

void app_main(void)
{
```

```
    xTaskCreate(&receiver, "sender", 2048, NULL, 2, &receiverHandler);
    xTaskCreate(&sender, "receiver", 2048, NULL, 2, NULL);
}
```

Deverá observar que a cada 5 segundos o destinatário recebe uma notificação.

QUESTÃO 6: Estude a função `ulTaskNotifyTake` e indique porque motivo não é necessário colocar um delay na execução da tarefa *receiver*?

QUESTÃO 7: Qual é o tempo máximo que a tarefa de chamada deve permanecer no estado Bloqueado para aguardar o seu valor de notificação?

QUESTÃO 8: Considere que se pretende obter o *output* apresentado no seguinte exemplo. Realize as alterações que considerar necessárias ao código fornecido de modo uma linha a cada 5 segundos.

```
~$ received notification 4 times
~$ received notification 4 times
~$ received notification 4 times
~$ received notification 4 times
```

QUESTÃO 9: Considerando o código anterior e analisando em pormenor as funções `xTaskNotify` e `xTaskNotifyWait`, evolua o seu código para que este apresente o *output* do seguinte bloco. Note que para alcançar este resultado terá de invocar 4 vezes a função `xTaskNotify`, com parâmetros diferentes, de modo a que o resultado impresso na função `receiver` reflita esses valores de entrada.

```
~$ received notification 1
~$ received notification 2
~$ received notification 4
~$ received notification 8
```

4.2.2 Mutex

O bloco de código seguinte ilustra um processo de escrita de valores de temperatura e humidade para um recurso partilhado, neste caso um barramento.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"

void writeToBus(char *message)
{
  printf(message);
}

void task1(void *params)
{
  while (true)
  {
       printf("reading temperature \n");
       writeToBus("temperature is 25c\n");
       vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}
```

```
void task2(void *params)
{
  while (true)
  {
       printf("reading humidity\n");
       writeToBus("humidity is 50 \n");
       vTaskDelay(2000 / portTICK_PERIOD_MS);
  }
}

void app_main(void)
{
  xTaskCreate(&task1, "temperature reading", 2048, NULL, 2, NULL);
  xTaskCreate(&task2, "humidity reading", 2048, NULL, 2, NULL);
}
```

QUESTÃO 10: Analise o código e explique porque motivo este código apresenta um problema.

QUESTÃO 11: Partindo do código fornecido e considerando as funções `xSemaphoreCreateMutex`, `xSemaphoreTake`, e `xSemaphoreGive`, construa uma programa que solucione o problema identificado na **questão 10** e, no caso de haver um erro na escrita da humidade ou temperatura para o barramento, apresente a mensagem "writing humidity timed out"/"writing temperature timed out", respectivamente.

5 Acknowledgements

Originally authored by Paulo C. Bartolomeu