EGM0017 (60H)

FLUXO E METODOLOGIAS DE PROJETO DE SISTEMAS EMBARCADOS

PROF. **JOSENALDE BARBOSA DE OLIVEIRA – UFRN**

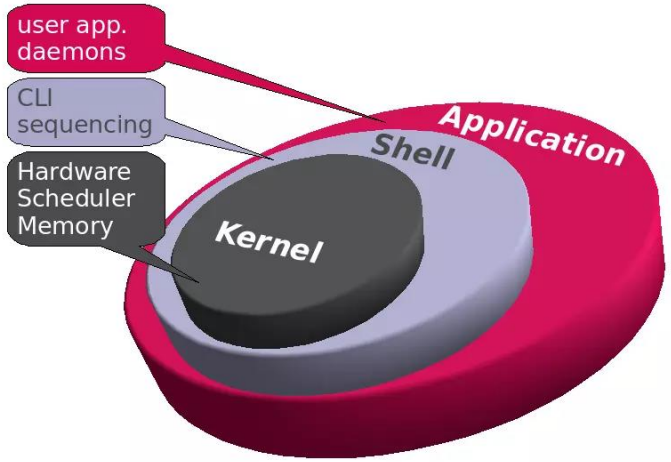
JOSENALDE.OLIVEIRA@UFRN.BR

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA MECATRÔNICA

conceitos base

- RTOS: ambiente multitarefas para sistemas embarcados
  - Um **Sistema Operacional** é um conjunto de ferramentas (softwares) que criam um ambiente multitarefas e também é uma abstração entre software e hardware, incluindo gerenciamento de recursos internos

https://www.freertos.org/index.html



- **Kernel**: camada de abstração entre sw e hw para gerência de recursos, como memória, processos, I/O etc.

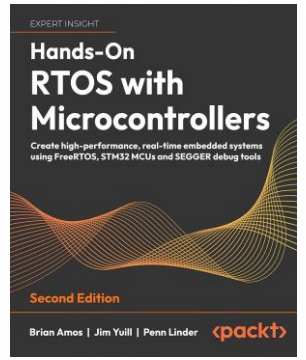
- **Scheduler**: software que "decide" quais tarefas executar, seja por maior prioridade ou, se iguais, por tentativa de divisão igualitária de tempo. No caso pthread_t as threads **bounded** (padrão) são gerenciadas pelo scheduler.

- **Task** (tarefa): como se fossem 'miniprogramas', podem ser totalmente independentes ou compartilhar recursos. Assim um código pode ser dividido entre tarefas gerenciadas pelo *scheduler*. *No FreeRTOS uma thread em execução é chamada TASK*

https://www.zephyrproject.org/

Exemplo de código de sincronização de tasks com semáforos no Zephyr

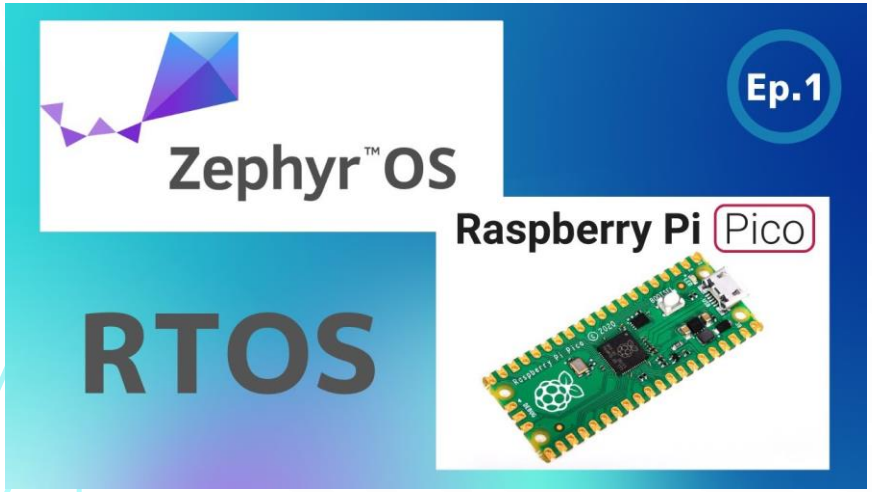
Referências

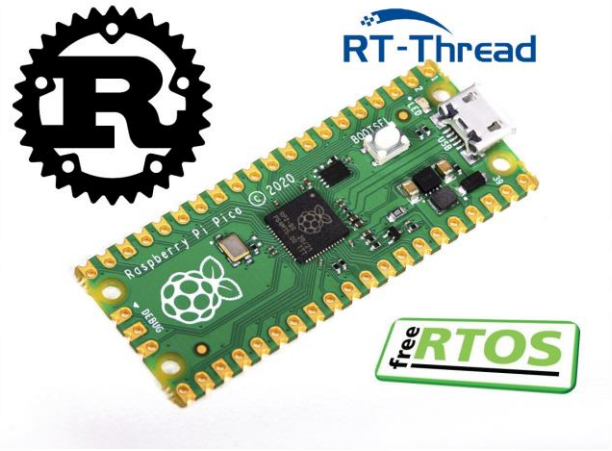


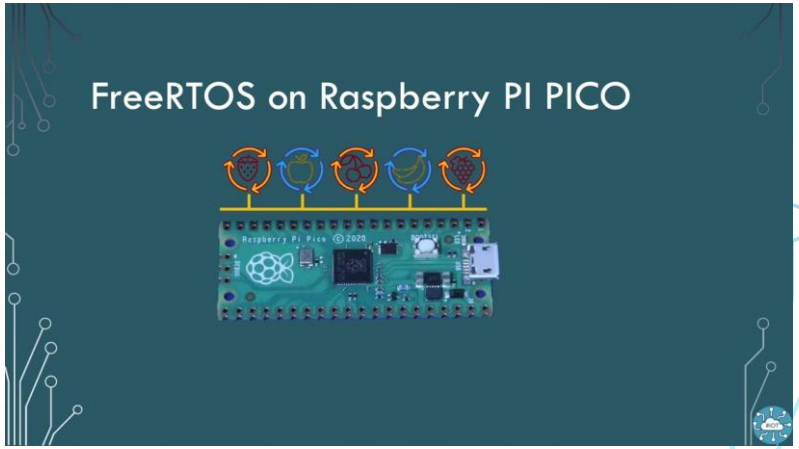
https://www.freertos.org/media/2018/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf

https://github.com/FreeRTOS/FreeRTOS-Kernel-Book/blob/main/toc.md

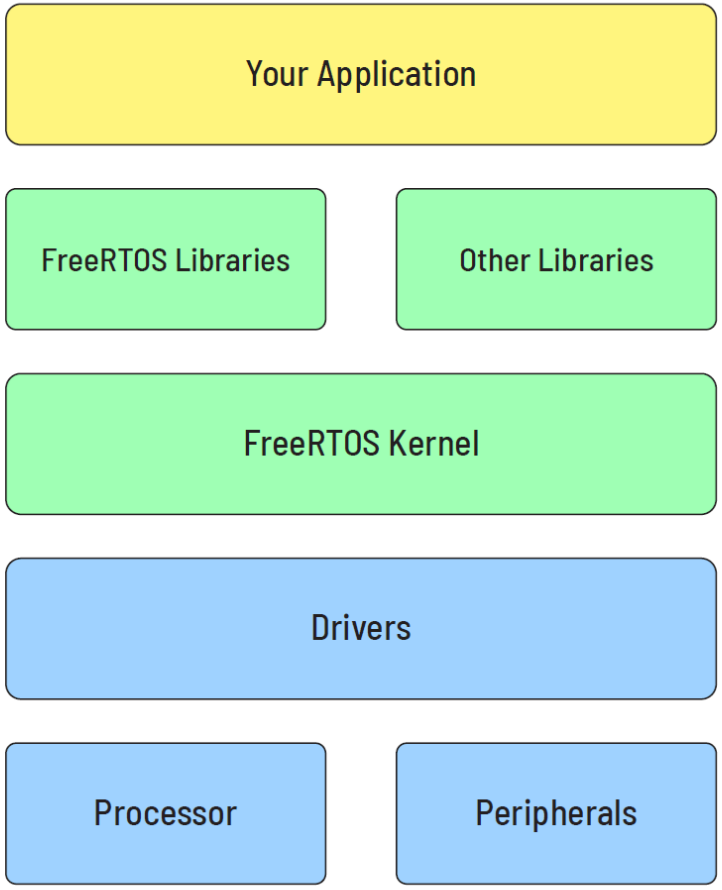
Outros “RTOSes”...

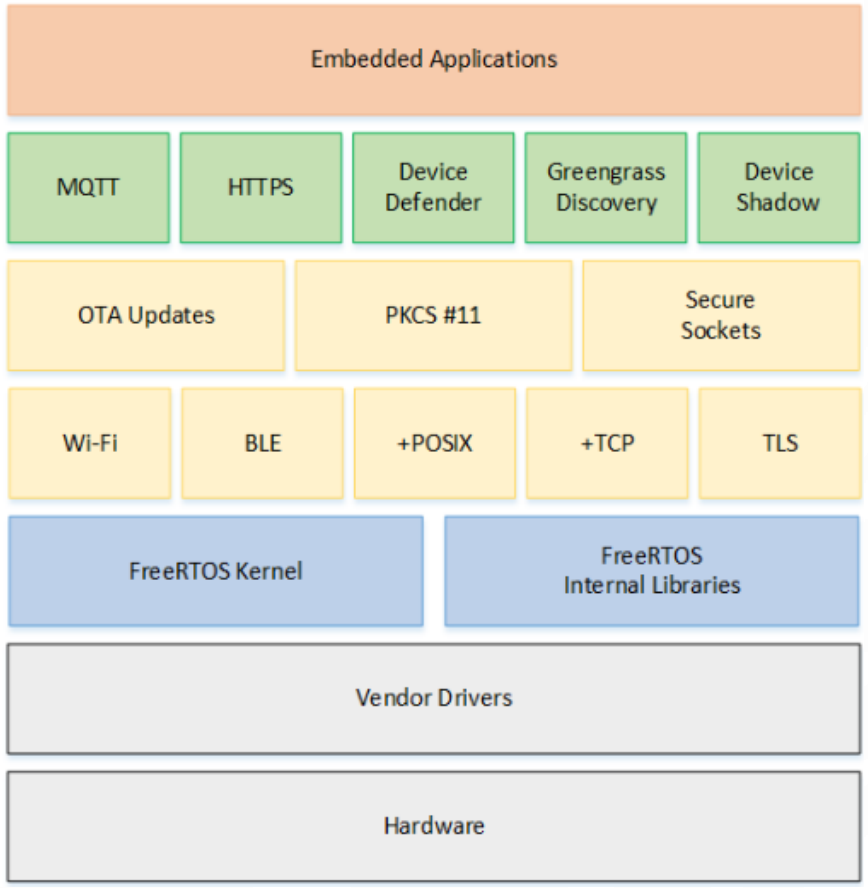






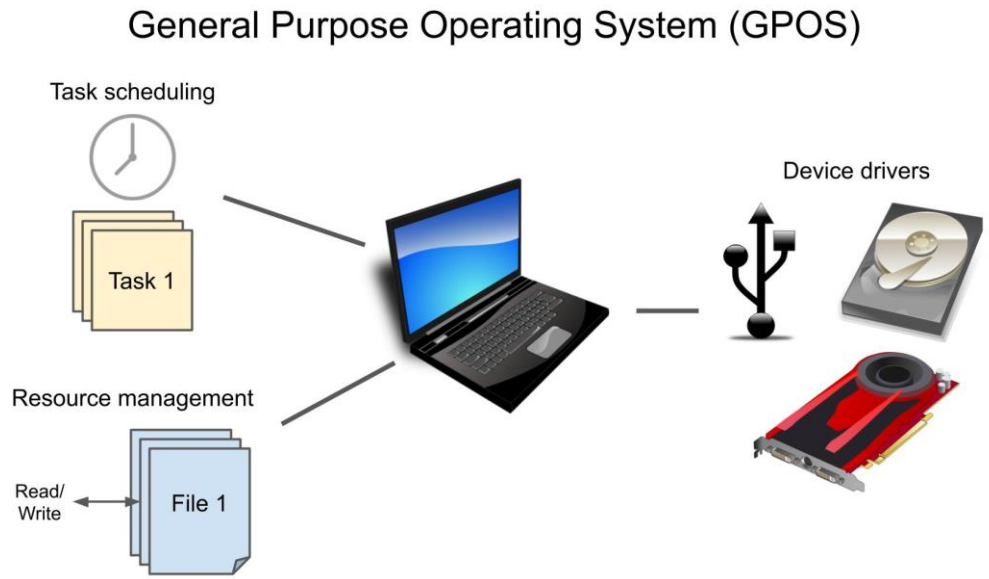
conceitos base

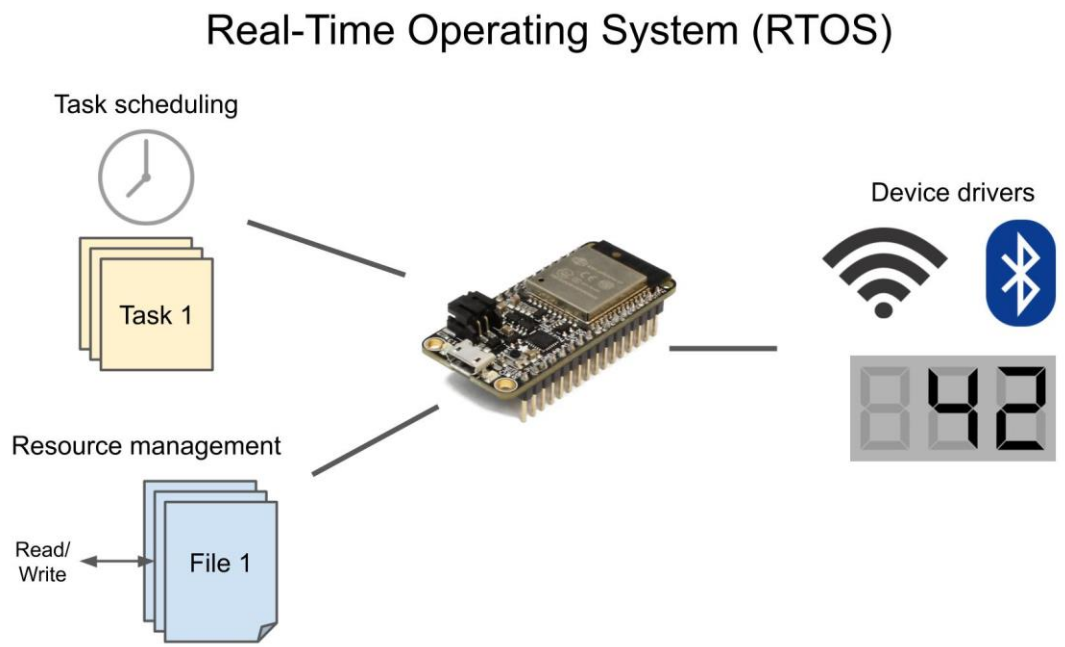




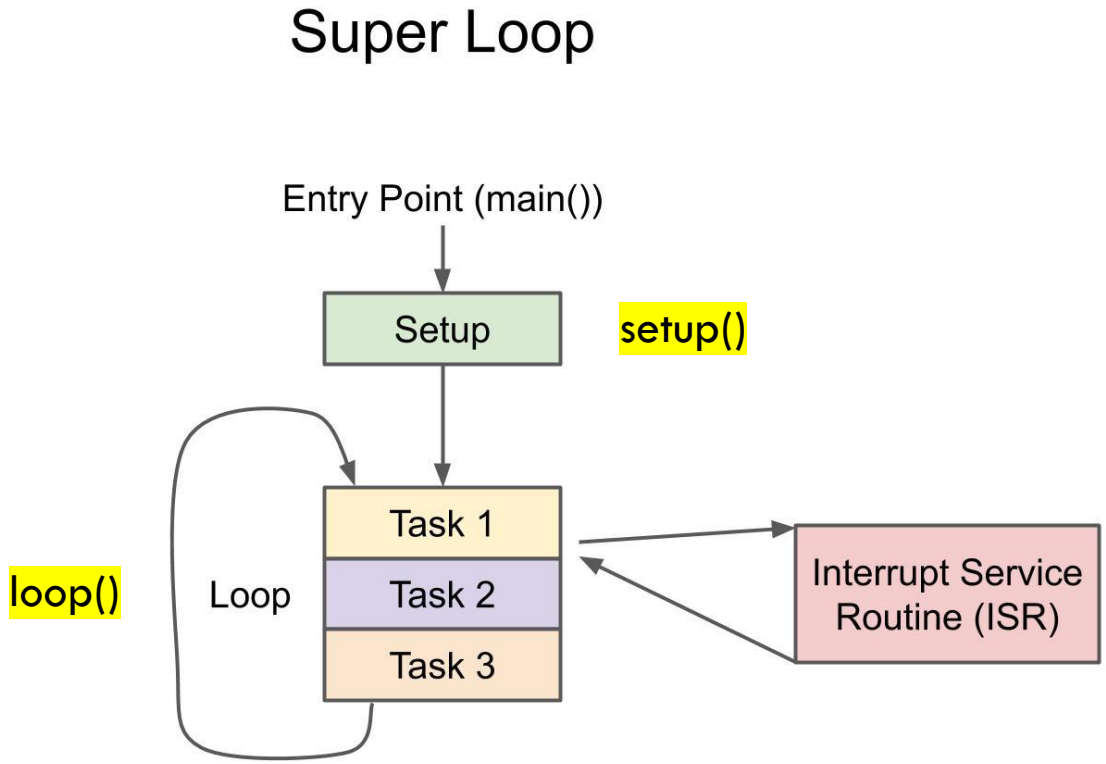
FreeRTOS Organization

GPOS x RTOS


General Purpose Operating System (GPOS)
Task scheduling
Task 1
Resource management
Read/
Write
File 1
Device drivers


Real-Time Operating System (RTOS)
Task scheduling
Task 1
Resource management
Read/
Write
File 1
Device drivers
42

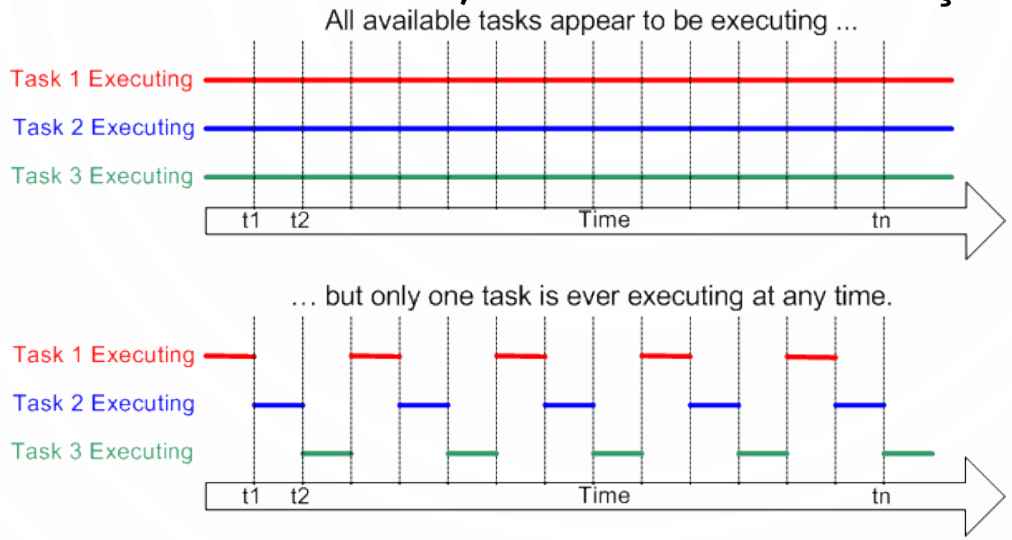
Super loop



Programação convencional de embarcados (bare metal) sem SO – tarefas sequenciais no Loop
Escalonador baseado em round-robin, com distribuição de tempo igualitária. Para muitas tarefas pode comprometer

Escalonador

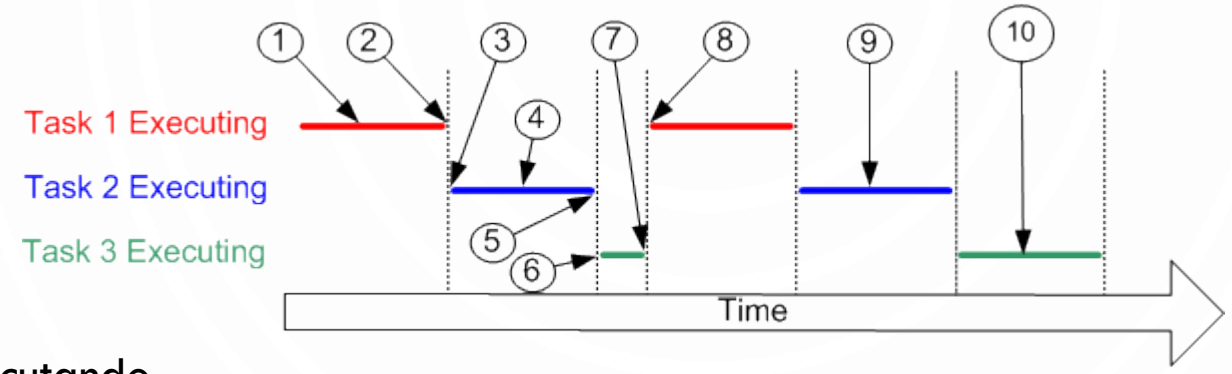
- Na prática, em GPOS (General Purpose OS), cada núcleo só executa uma thread por vez, mas o scheduler chaveia rapidamente entre várias threads, dando 'ilusão' de execução simultânea



- O scheduler no RTOS é projetado para prover um padrão de execução previsível (determinístico). Isto é particularmente útil em sistemas embarcados, que geralmente possuem restrições de tempo real, ou seja, que uma tarefa deva responder estritamente dentro de um período de tempo. Isto só pode ocorrer se o comportamento do scheduler do SO puder ser predito. No freeRTOS isto é controlado pelo programador por meio da definição de PRIORIDADES para as TASKS.
- O freeRTOS é uma classe de RTOS. Disponibiliza primitivas para scheduler, comunicação entre processos (IPC), temporização e sincronismo

Escalonador

- Ideia geral de escalonamento

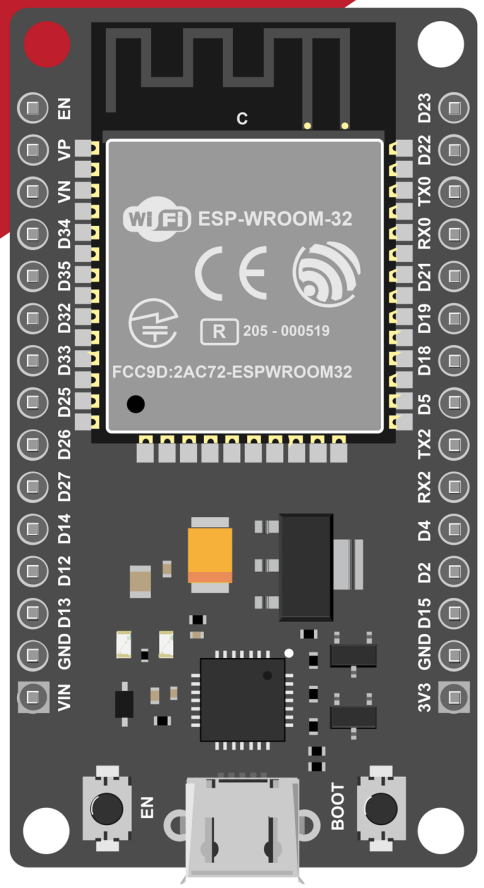


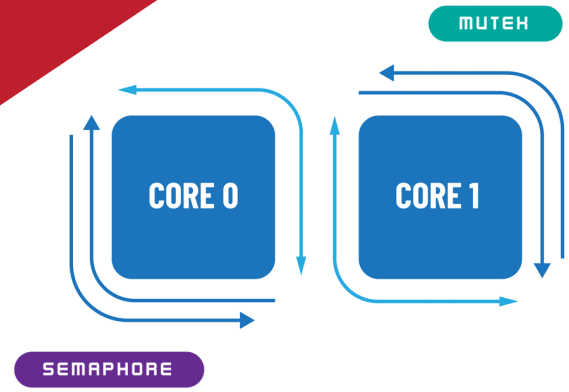
- (1) task1 executando
- Em (2) kernel suspende task1 e assume task2 em (3)
- Quando task2 está executando (4) bloqueia um periférico para seu próprio uso
- Em (5) kernel suspende task2 e em (6) assume task3
- task3 tenta usar o mesmo periférico que task2, não consegue, e se auto suspende em (7)
- Em (8) task1 assume etc.
- Da próxima vez que task2 executar em (9) concluiu o uso do periférico libera-o
- E agora task3 em (10) acessa o periférico e executa até ser suspensa pelo kernel

Write Parallel Multitasking Applications for ESP32 with FreeRTOS & Arduino

Learn how to take advantage of multitasking features of FreeRTOS for ESP32 dual-core SoC using your favorite Arduino IDE.

https://circuitstate.com/pmulesp32





https://www.circuitstate.com/tutorials/how-to-write-parallel-multitasking-applications-for-esp32-using-freertos-arduino/

Criando tasks

- No código executado em setup() é utilizada a função

  `xTaskCreatePinnedToCore()` que é uma especialização de `xTaskCreate()`

xTaskCreate

[Task Creation]

task. h

```
BaseType_t xTaskCreate(    TaskFunction_t pvTaskCode,
                           const char * const pcName,
                           configSTACK_DEPTH_TYPE usStackDepth,
                           void *pvParameters,
                           UBaseType_t uxPriority,
                           TaskHandle_t *pxCreatedTask
                       );
```

Cria nova task e a inclui na lista de tarefas READY_TO_RUN

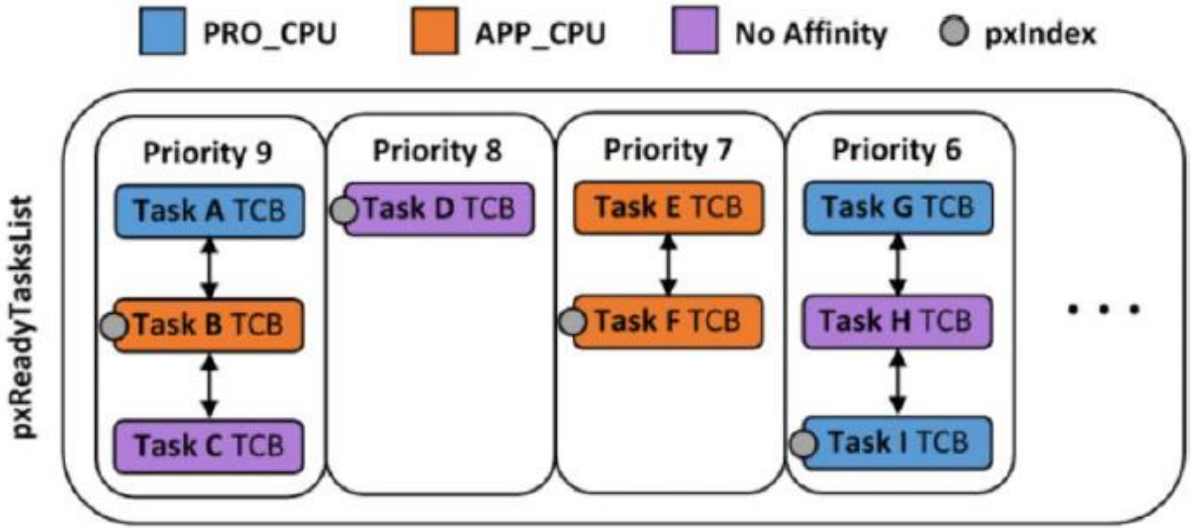
- RAM automaticamente alocada pelo heap do freeRTOS
- **pvTaskCode**: nome da função que implementa a task
- **pcName**: nome descritivo para a task. Pode ser usado, se necessário, para resgatar o handle da task, com xTaskGetHandle(*pcName*)
- **usStackDepth**: número de WORDS (grupos de 16bits/2bytes) para alocar à pilha da task (task stack). Contudo no ESP-IDF está em bytes mesmo.
- **pvParameters**: parâmetros para passar a task (tal como no pthread_create)
- **uxPriority**: prioridade da task
- pxCreatedTask: handle externo da task. Em geral NULL.

Sucesso: pdPASS
Falha: errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY

Criando tasks, núcleos, prioridades

- No código executado em setup() é utilizada a função

  `xTaskCreatePinnedToCore()`



Afinidade por núcleo: introduz parâmetro extra xCoreID, que pode ser:

- 0 (CPU0): Protocol CPU (PRO_CPU)
- 1 (CPU1): Application CPU (APP_CPU)

Obs: em geral comunicações (wireless etc.) no PRO_CPU e o resto da aplicação no APP_CPU (por exemplo setup, loop())

Criando tasks

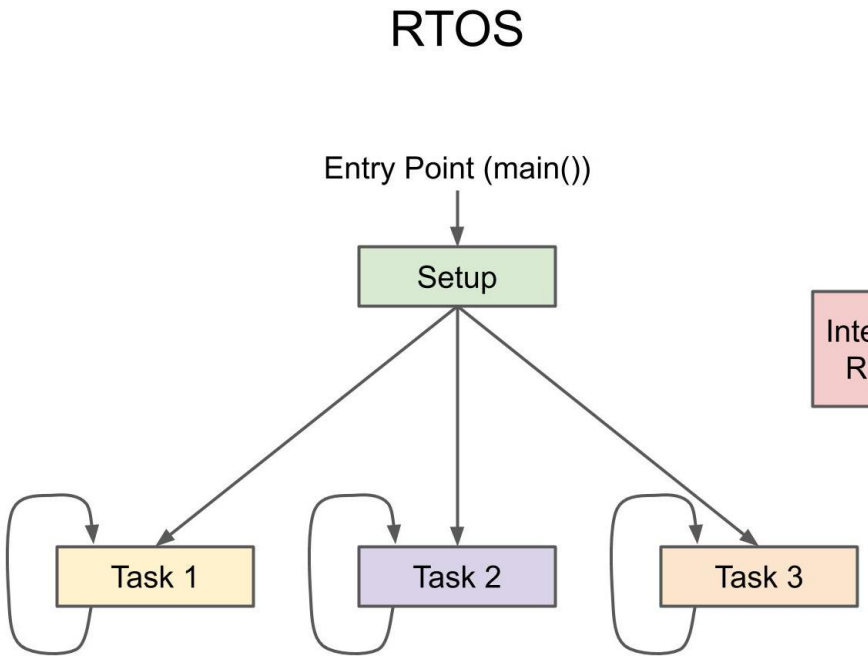
- Na função t1 é utilizado vTaskDelay(ticks) – exemplo de uso:

```
void vTaskFunction(void *pvParameters)
{
          /* Block for 500ms. */
          const TickType_t xDelay = 500 / portTICK_PERIOD_MS;
          for( ;; ) {
          /* Simply toggle the LED every 500ms, blocking between each toggle. */
             vToggleLED();
             vTaskDelay(xDelay);
          }
}
```

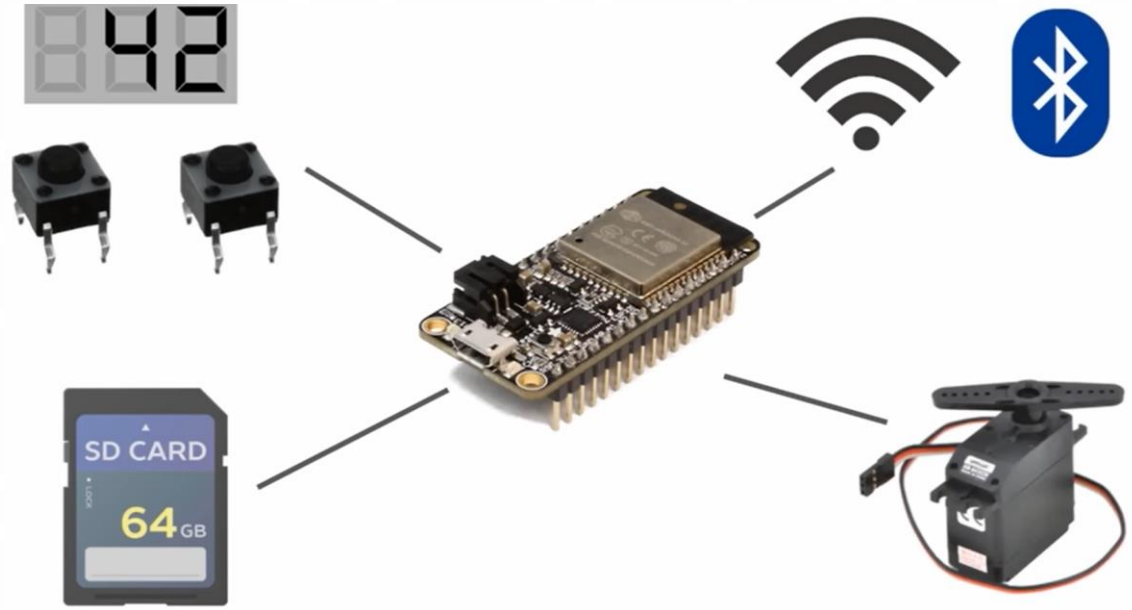
Mas para processos real time com necessidade precisão temporal, indicado vTaskDelayUntil() vTaskDelay() dependerá do momento em que foi chamada em relação ao tick Count. Pode ser usado usado pdMS_TO_TICKS para obter o número de ticks associado ao tempo em milissegundos.

```
vTaskDelay(pdMS_TO_TICKS(400)); //default 1 tick=1ms
```

Alguns exercícios sobre sincronismo e tarefas



Interrupt Service Routine (ISR)

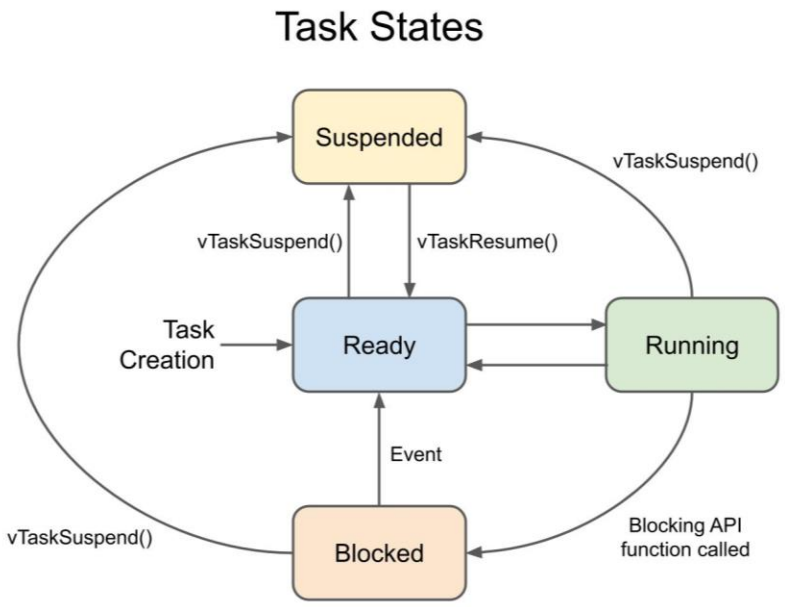


Programação com RTOS – tarefas em loop, podendo ser interrompidas por ISR, que devolve à task após a execução

Vamos olhar dois exemplos simples de multitask no ESP32
https://github.com/josenalde/control_outputs_rtos_semaphore

FreeRTOS – lidando com TASKS

- No FreeRTOS, a fatia de tempo (tick) é 1ms
- A cada 1ms, o temporizador de hardware cria uma interrupção. A ISR do temporizador roda o scheduler, escolhe a próxima tarefa a executar
- A cada interrupção (tick), a tarefa com maior prioridade é escolhida
- Se as tarefas de maior prioridade possuem a mesma prioridade, são executadas em round-robin (divisão de tempo)
- Se uma task de maior prioridade que a task e em execução entra em modo READY, é executada imediatamente, sem esperar próximo TICK
- Uma ISR tem prioridade maior que qualquer task em execução



Criação – Pronta (Ready) – Scheduler escolhe e coloca em Running
Pode retornar ao Ready, ou ser bloqueada (Blocked) por um vTaskDelay ou por que está aguardando algum recurso, como um semáforo. Se Bloqueada (Blocked) outras tasks podem rodar. Qualquer thread pode colocar a si e outras em suspenso (Suspended) (vTaskSuspend())
e retornar com vTaskResume()

FreeRTOS – lidando com TASKS

- Lendo serial e piscando LEDs

https://github.com/josenalde/flux-embedded-design/blob/main/src/led_serial_tasks.cpp

- alterar prioridade do Led para 2, mantendo a Serial em 1, o que ocorre?
- alterar prioridade do Serial para 2, mantendo Led em 1, o que ocorre?

- Lendo serial e piscando LEDs com Mutex para garantir leitura de variável

https://github.com/josenalde/flux-embedded-design/blob/main/src/led_serial_mutex_2.cpp