RTOS
Sistemas Operacionais de Tempo Real

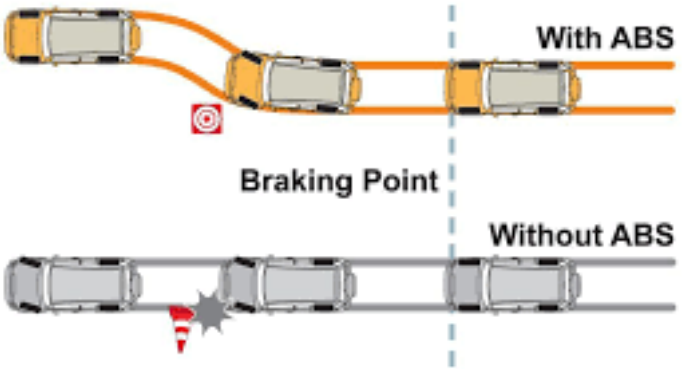
FREERTOS API para kernel RTOS

Sistemas Embarcados

Prof. Marcos Chaves

Sistemas Operacionais de Tempo Real

- No contexto do desenvolvimento de software, um sistema é projetado para receber um estímulo (ou evento), que pode ser interno ou externo, realizar o processamento e produzir uma saída.
- Alguns sistemas trabalham com eventos que possuem restrição de tempo, ou seja, possuem prazo ou limite de tempo para que o estímulo seja processado e gere a saída correspondente.
- Esses tipos de sistemas são chamados de "Sistemas de tempo real".



RTOS

A principal tarefa de um RTOS é gerenciar os recursos do microcontrolador de modo que uma determinada operação execute exatamente em intervalos iguais.

Sistemas Operacionais de Tempo Real

- Portanto, um sistema em tempo real precisa garantir que todos os eventos sejam atendidos dentro de suas respectivas restrições de tempo.
- É por isso que um sistema em tempo real está relacionado ao determinismo, não ao tempo de execução!
- Existem basicamente **dois tipos de sistemas de tempo real**, classificados de acordo com a tolerância às restrições de tempo e as consequências em não respeitar essas restrições.

Sistemas de Tempo Real Suave (Soft Real-Time Systems)

- Nestes sistemas, embora seja importante que as tarefas e eventos sejam concluídos dentro do prazo, não é catastrófico se alguns prazos forem perdidos ocasionalmente. A qualidade do serviço pode diminuir, mas o sistema ainda pode funcionar de forma aceitável.

Exemplo: Streaming de vídeo, sistemas de telefonia, jogos online.

Sistemas de Tempo Real Duro (Hard Real-Time Systems)

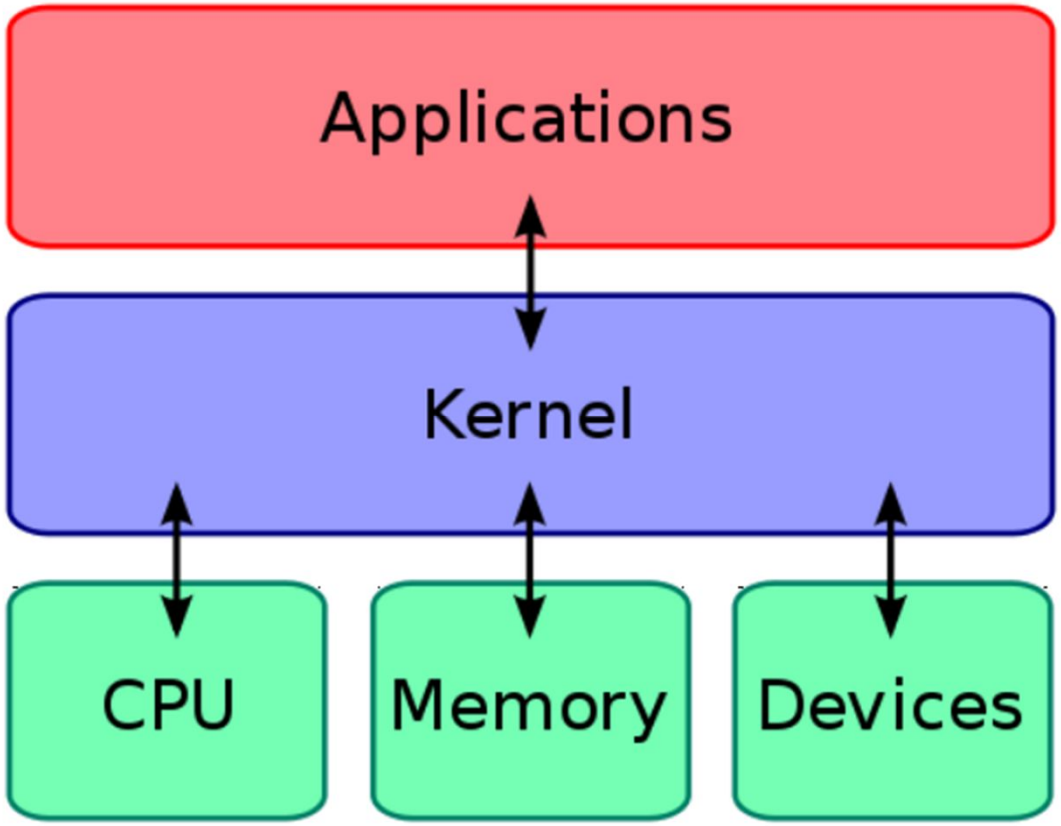
- Nestes sistemas, é absolutamente crítico que todas as tarefas e eventos sejam concluídos dentro do prazo estabelecido. Qualquer atraso ou falha em atender aos prazos pode resultar em falhas catastróficas do sistema.

Exemplo: Controle de airbags em automóveis, sistemas de controle de voo, equipamentos médicos críticos.

O kernel em tempo real

- Um kernel em tempo real é um software que gerencia o tempo e os recursos da CPU e é baseado no conceito de tarefas e prioridades.
- Todos os recursos do sistema são divididos em tarefas (tarefas ou threads).
- O kernel decide quando uma tarefa deve ser executada com base na prioridade da tarefa.
- É responsabilidade do desenvolvedor dividir o sistema em tarefas e definir as prioridades de acordo com as características em tempo real de cada uma.

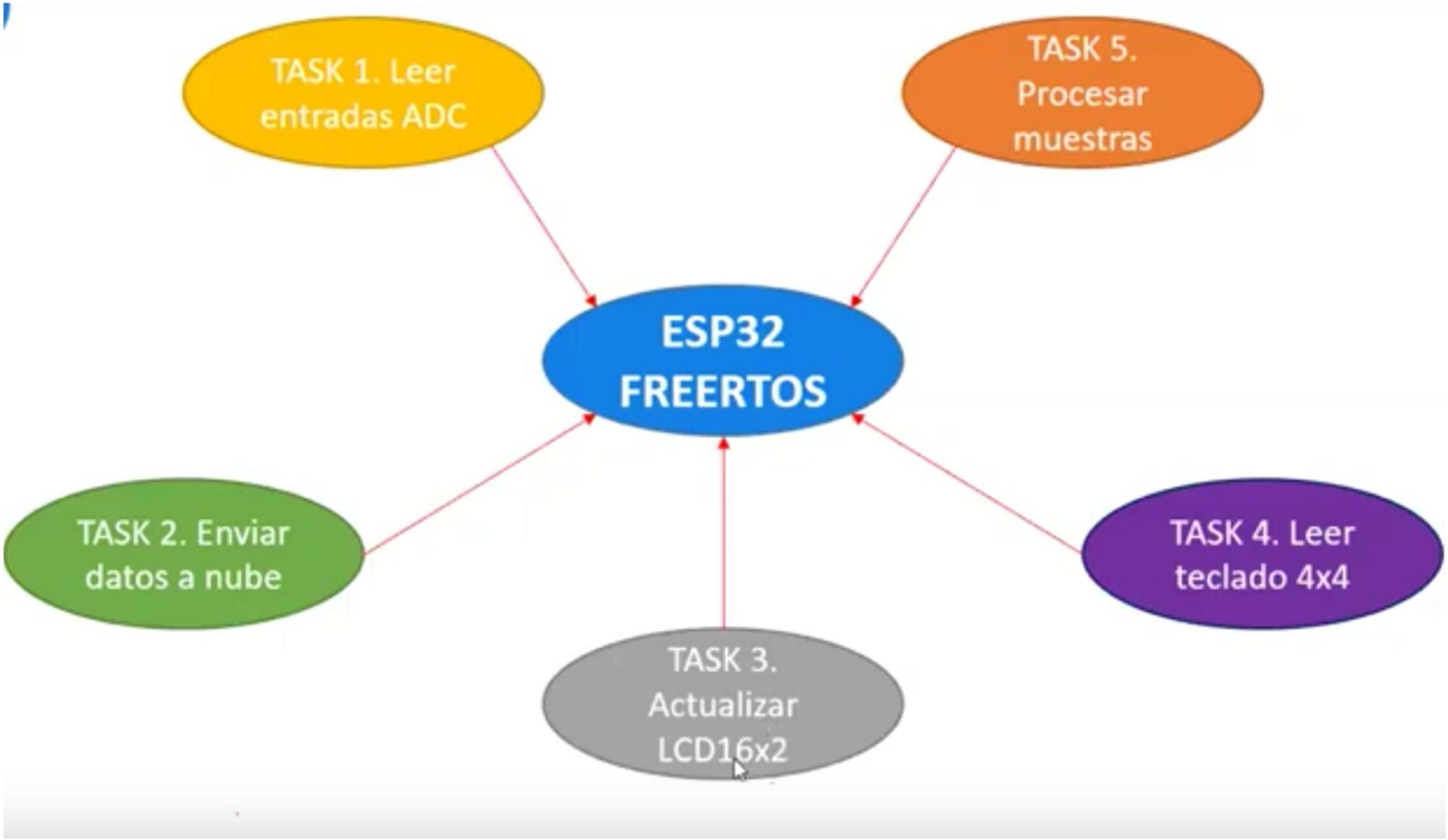
Sistemas Operacionais e Kernel



Sistemas Operacionais e Kernel

Kernel é o núcleo do sistema operacional, sendo uma camada de abstração entre o software e hardware para gerenciar diversos recursos:

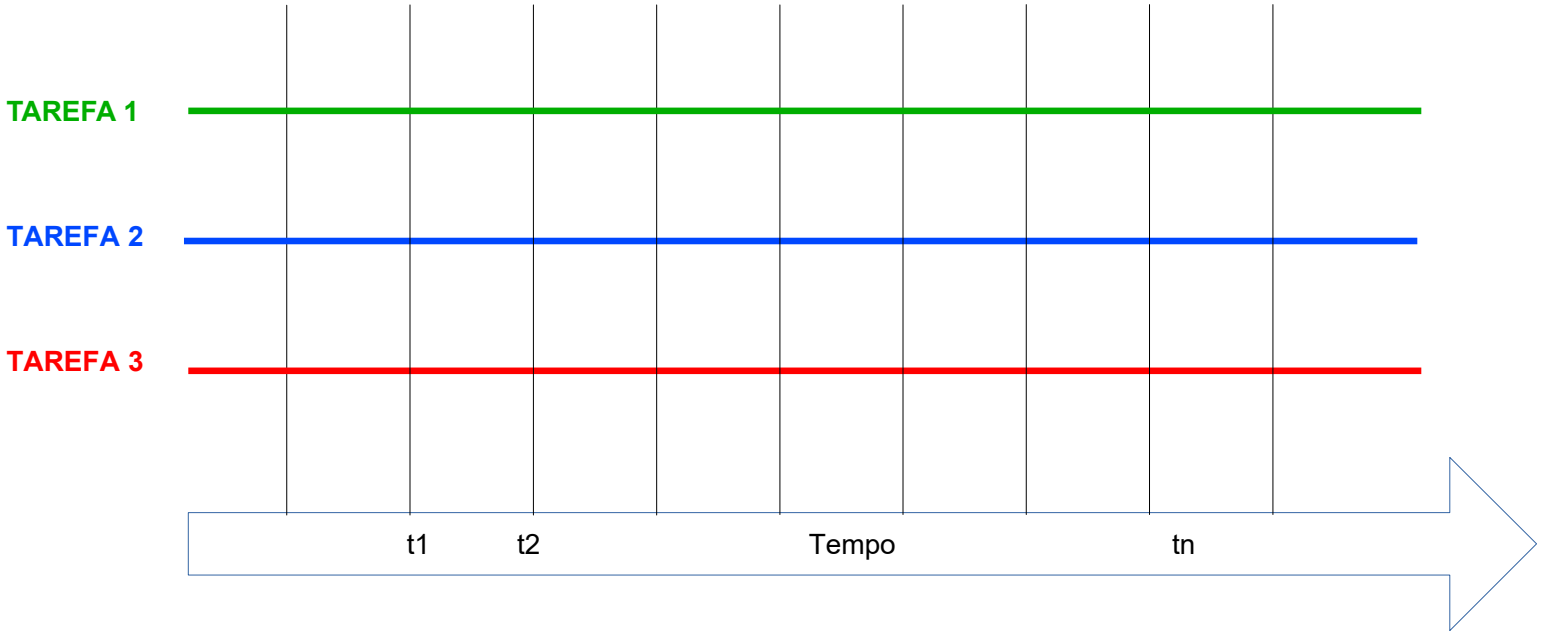
1. Gerenciar e coordenar a execução dos processos através de algum critério
2. Manusear a memória disponível e coordenar o acesso dos processos a ela
3. Intermediar a comunicação entre os drivers de hardware e os processos



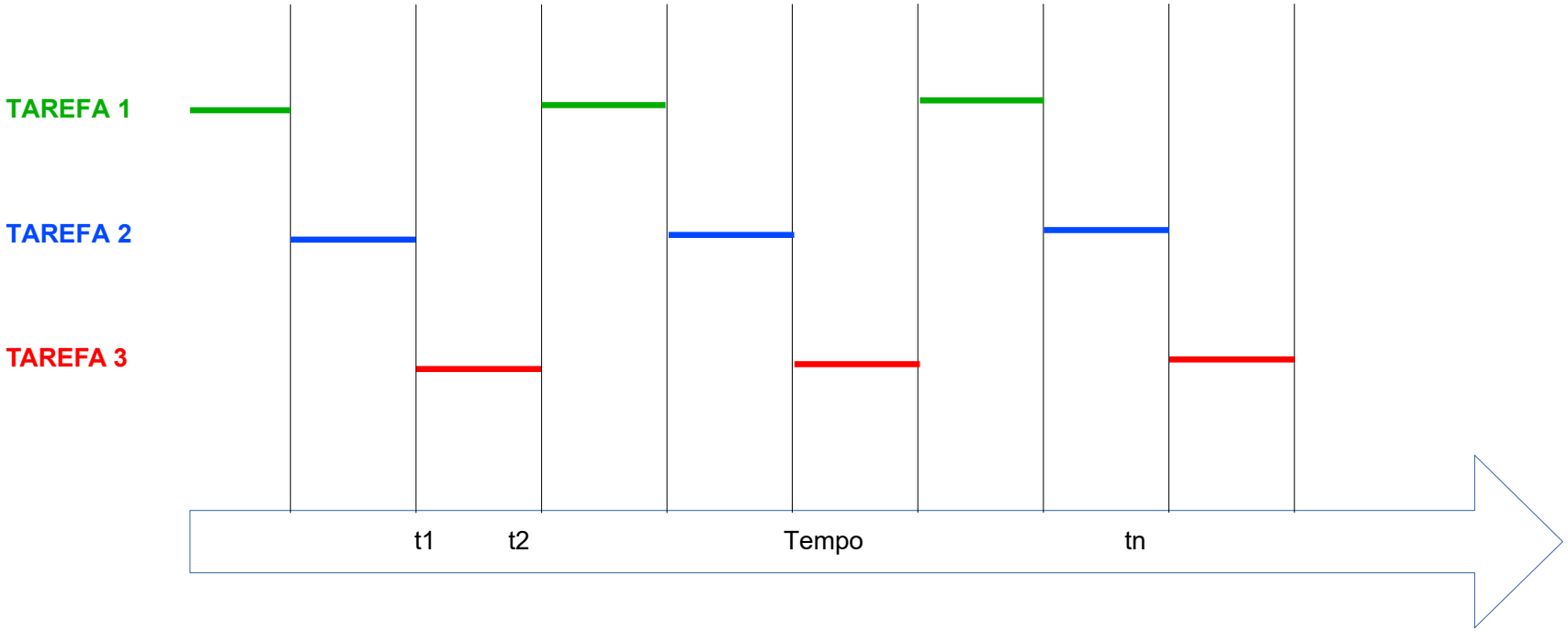
Programação típica com divisão de processos (tarefas ou tasks)

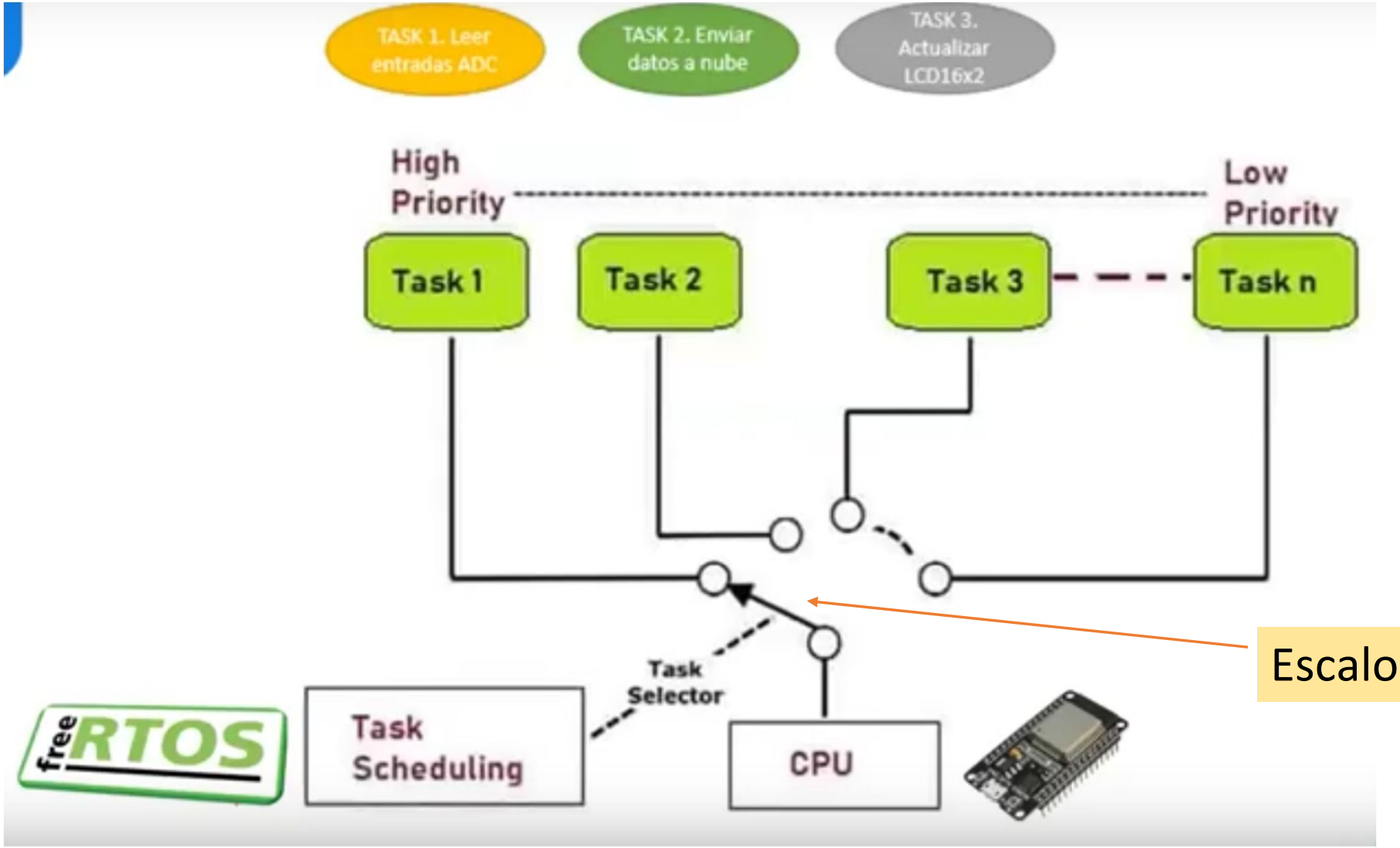
Multitask

- Em um sistema multitarefa, temos a impressão de que todas as tarefas estão sendo executadas ao mesmo tempo.



Multitask





Programação típica com divisão de processos (tarefas ou tasks)

Scheduler

Scheduler (agendador ou escalonador) é o grande responsável por administrar as tarefas que irão obter o uso da CPU. Há diversos algoritmos para que o scheduler decida a tarefa e você também pode escolher o mais apropriado ao seu embarcado, como por exemplo:
RR (Round Robin), SJF (Shortest Job First) e SRT (Shortest Remaining Time).

Formas de trabalho:

- Preempetivo
- Cooperativo
- Troca de contexto
- Time Slicing

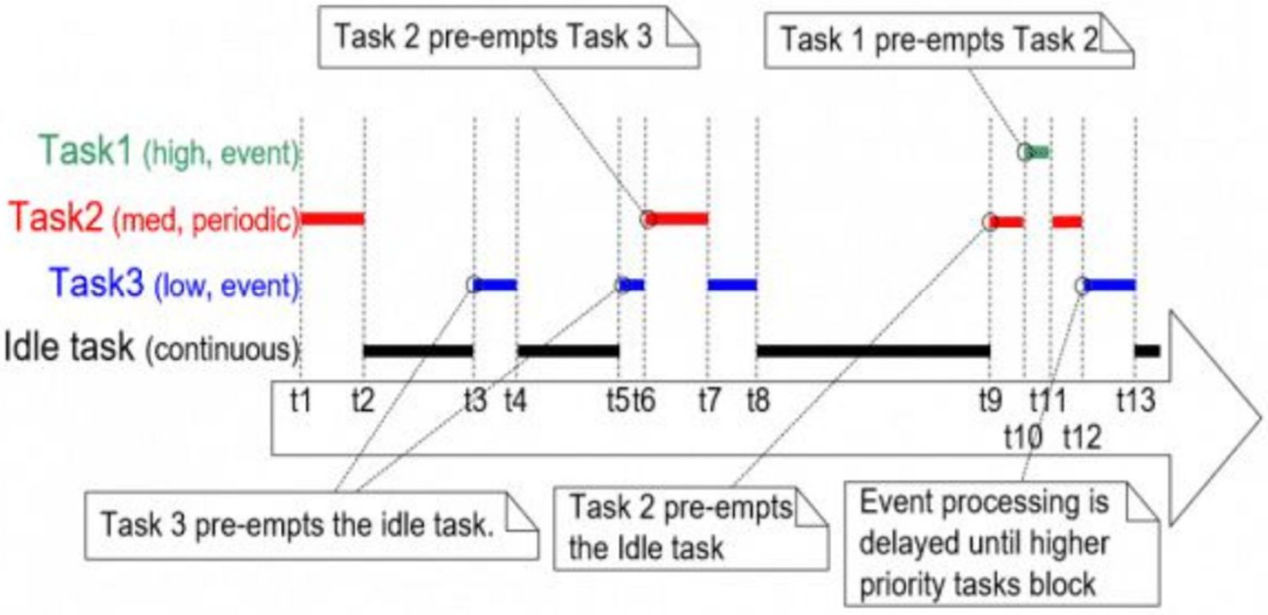
Scheduler (Escalonador)

- O agendador de tarefas age durante as mudanças de contexto.
- É a parte do *kernel* responsável por decidir a próxima *tasks* a ser executada a qualquer momento.
- O algoritmo responsável por decidir qual a próxima *tasks* a ser executada é chamado de política de agendamento.

Preempção

Preemptivos: permitem que uma tarefa seja temporariamente suspensa para a execução de outra com maior prioridade

Não-preemptivos: impedem que a execução de uma tarefa seja interrompida até sua conclusão



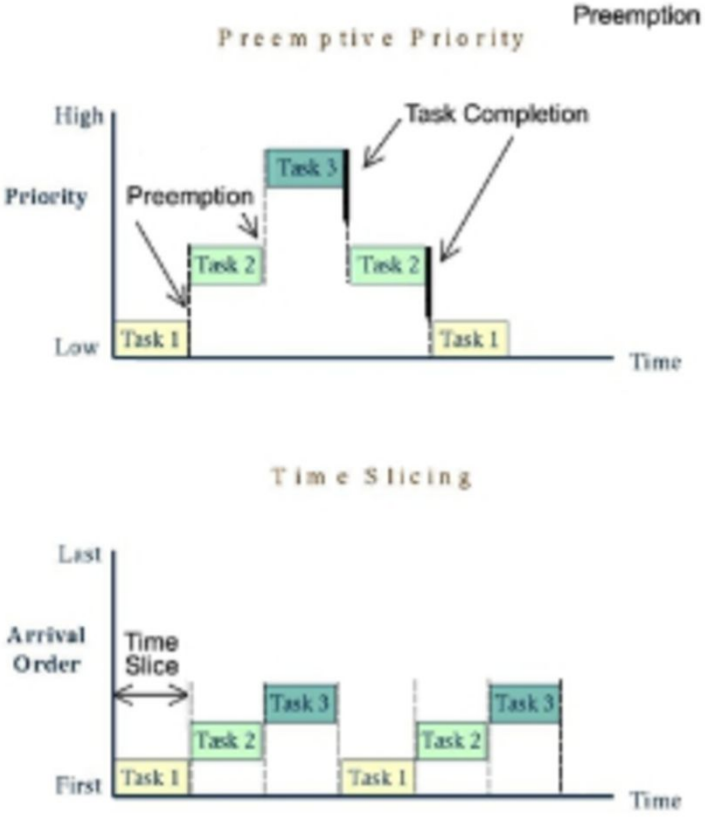
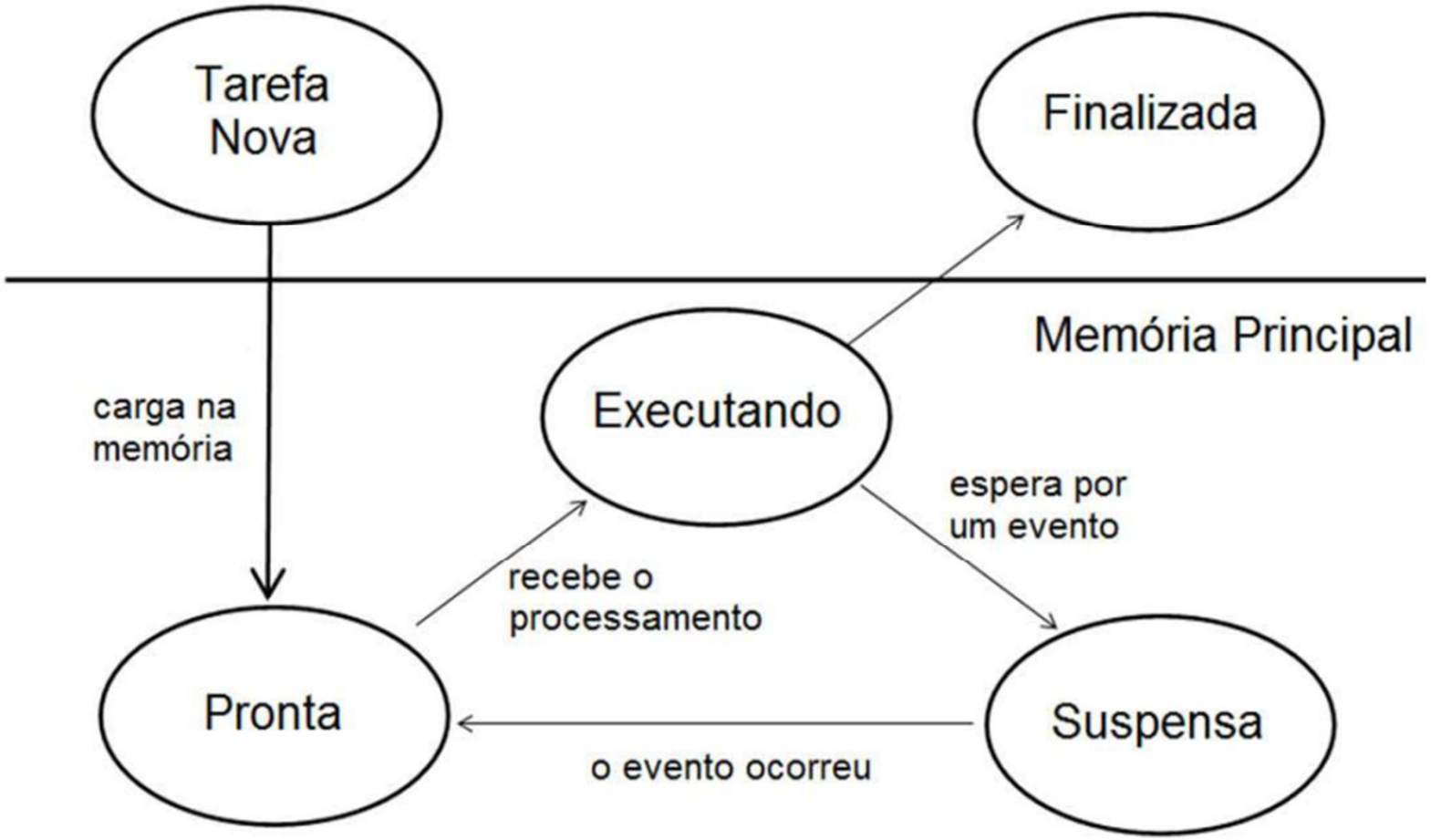


Diagrama de estados das tarefas em um sistema multitarefa



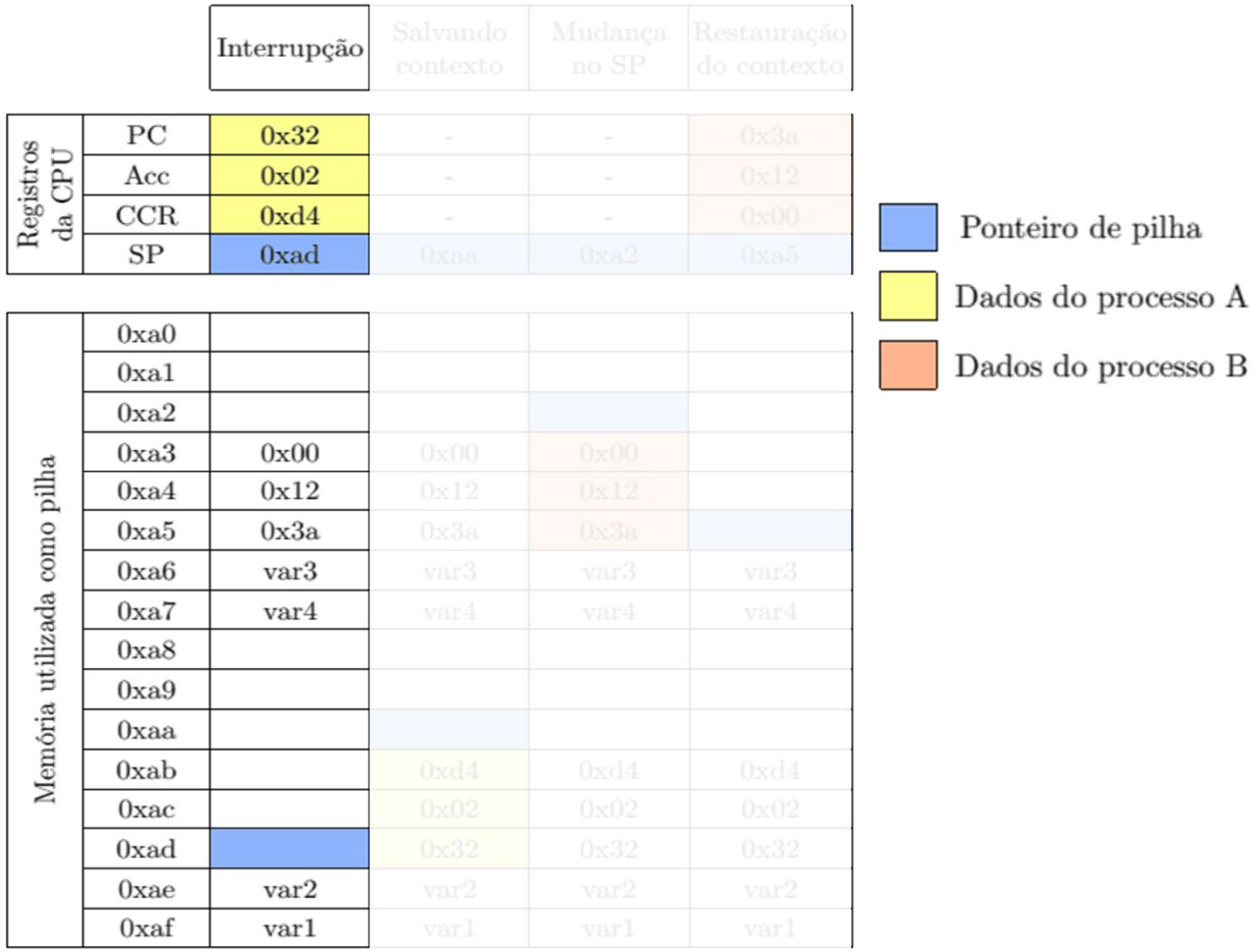
Multitask (cont.)

- Essa troca ou mudança de tarefa pode acontecer em diferentes situações: :
  - Uma *tasks* ficar bloqueada a espera de um recurso (por exemplo, porta serial) ficar disponível ou um evento ocorrer (por exemplo, receber um pacote da interface USB).
  - Uma *tasks* pode adormecer (bloquear) por um tempo.
  - Uma *tasks* pode ser suspensa involuntariamente pelo kernel. Nesse caso, chamamos o **kernel preemptivo**.
  - Esta mudança de *tasks* também é chamada de **mudança de contexto**

Mudança de contexto

- Enquanto uma *tasks* está em execução, ela possui um determinado contexto (pilha, registros da CPU, etc).
- Ao alterar a *tasks* em execução, o kernel salva o contexto da tarefa a ser suspensa e recupera o contexto da próxima tarefa a ser executada.
- O controle do contexto de cada uma das *tasks* é realizado através de uma estrutura denominada TCB (Bloco de Controle de *tasks*).

Mudança de contexto



Mudança de contexto

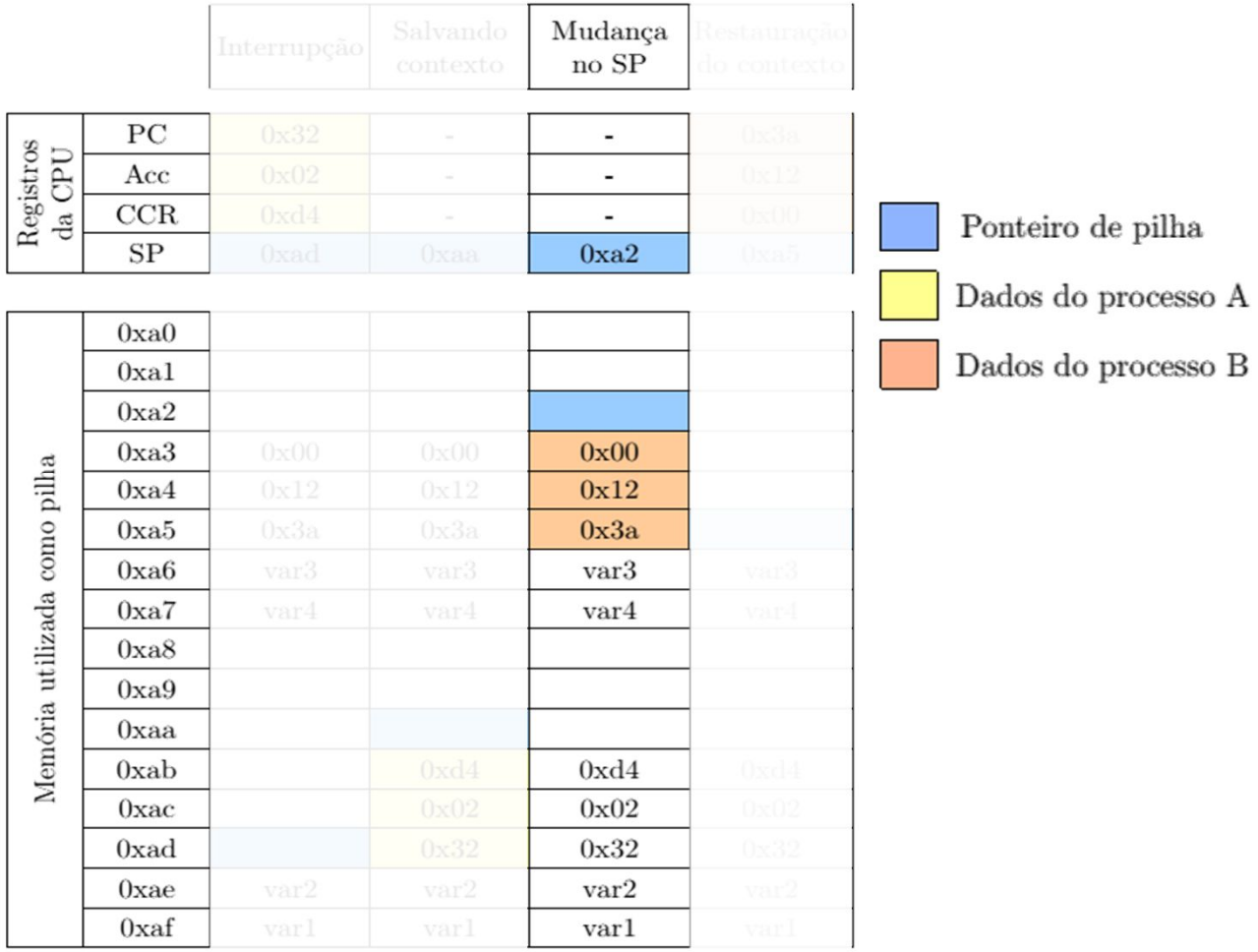
		Interrupção	Salvando contexto	Mudança no SP	Restauração do contexto
Registros da CPU	PC	0x32	-	-	0x3a
	Acc	0x02	-	-	0x12
	CCR	0xd4	-	-	0x00
	SP	0xad	0xaa	0xa2	0xa5
Memória utilizada como pilha	0xa0				
	0xa1				
	0xa2				
	0xa3	0x00	0x00	0x00	
	0xa4	0x12	0x12	0x12	
	0xa5	0x3a	0x3a	0x3a	
	0xa6	var3	var3	var3	var3
	0xa7	var4	var4	var4	var4
	0xa8				
	0xa9				
	0xaa				
	0xab		0xd4	0xd4	0xd4
	0xac		0x02	0x02	0x02
	0xad		0x32	0x32	0x32
	0xae	var2	var2	var2	var2
	0xaf	var1	var1	var1	var1

Ponteiro de pilha

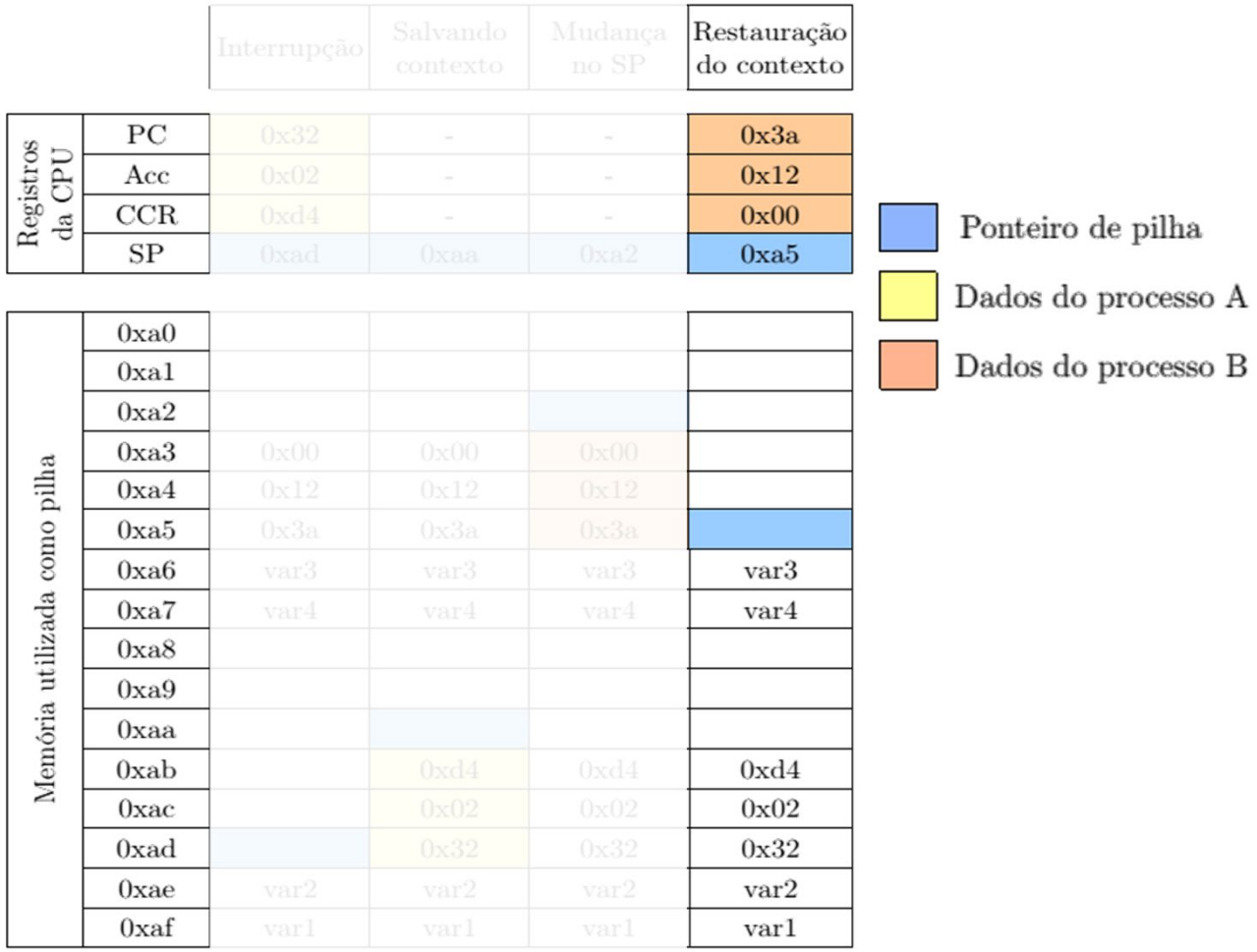
Dados do processo A

Dados do processo B

Mudança de contexto



Mudança de contexto



Mudança de contexto

		Interrupção	Salvando contexto	Mudança no SP	Restauração do contexto
Registros da CPU	PC	0x32	-	-	0x3a
	Acc	0x02	-	-	0x12
	CCR	0xd4	-	-	0x00
	SP	0xad	0xaa	0xa2	0xa5
Memória utilizada como pilha	0xa0				
	0xa1				
	0xa2				
	0xa3	0x00	0x00	0x00	
	0xa4	0x12	0x12	0x12	
	0xa5	0x3a	0x3a	0x3a	
	0xa6	var3	var3	var3	var3
	0xa7	var4	var4	var4	var4
	0xa8				
	0xa9				
	0xaa				
	0xab		0xd4	0xd4	0xd4
	0xac		0x02	0x02	0x02
	0xad		0x32	0x32	0x32
	0xae	var2	var2	var2	var2
	0xaf	var1	var1	var1	var1

Ponteiro de pilha

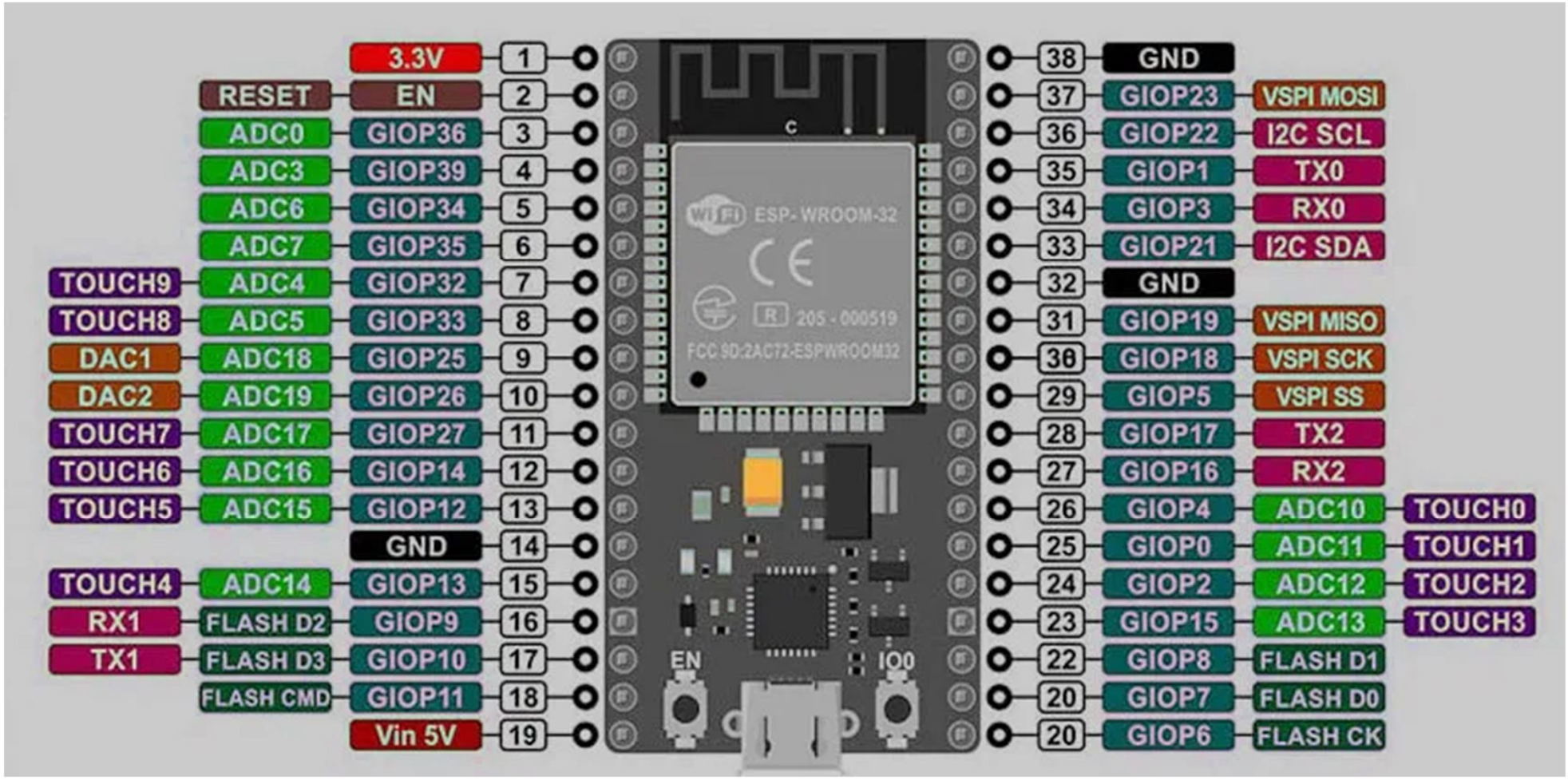
Dados do processo A

Dados do processo B

Outras responsabilidades do kernel

- Além de gerenciar o uso da CPU, um kernel em tempo real normalmente tem outras responsabilidades, incluindo:
  - Gerenciar a comunicação entre as tasks.
  - Gerenciar a comunicação entre interrupções e tasks.
  - Gerenciar o acesso aos recursos do aplicativo (hardware, estruturas de dados, etc).
  - Gerenciar o uso de memória.
  - Fornece outros recursos, como temporizadores, rastreamento, etc.

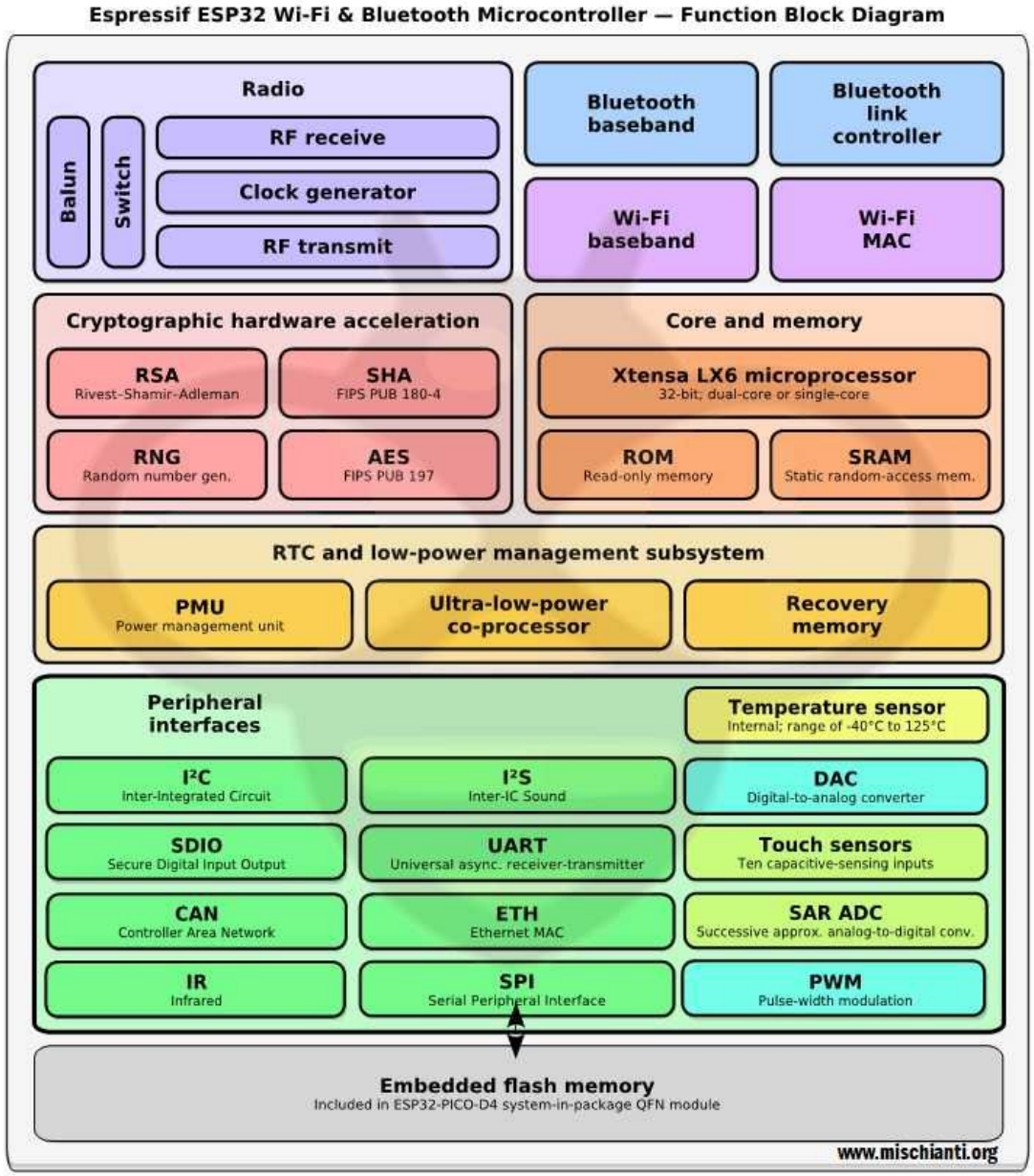
O Hardware: ESP32 Expressif



https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

	ESP8266	ESP32
MCU	Xtensa Single-core 32-bit L106	Xtensa Dual-Core 32-bit LX6 with 600 DMIPS
802.11 b/g/n Wi-Fi	HT20	HT40
Bluetooth	**No**	Bluetooth 4.2 and BLE
Typical Frequency	80 MHz	160 MHz
SRAM	**No**	**Yes**
Flash	**No**	**Yes**
GPIO	17	36
Hardware /Software PWM	None / 8 channels	None / 16 channels
SPI/I2C/I2S/UART	2/1/2/2	4/2/2/2
ADC	10-bit	12-bit
CAN	**No**	**Yes**
Ethernet MAC Interface	**No**	**Yes**
Touch Sensor	**No**	**Yes**
Temperature Sensor	**No**	**Yes**
Hall effect sensor	**No**	**Yes**
Working Temperature	-40°C to 125°C	-40°C to 125°C

O Hardware: ESP32 Expressif

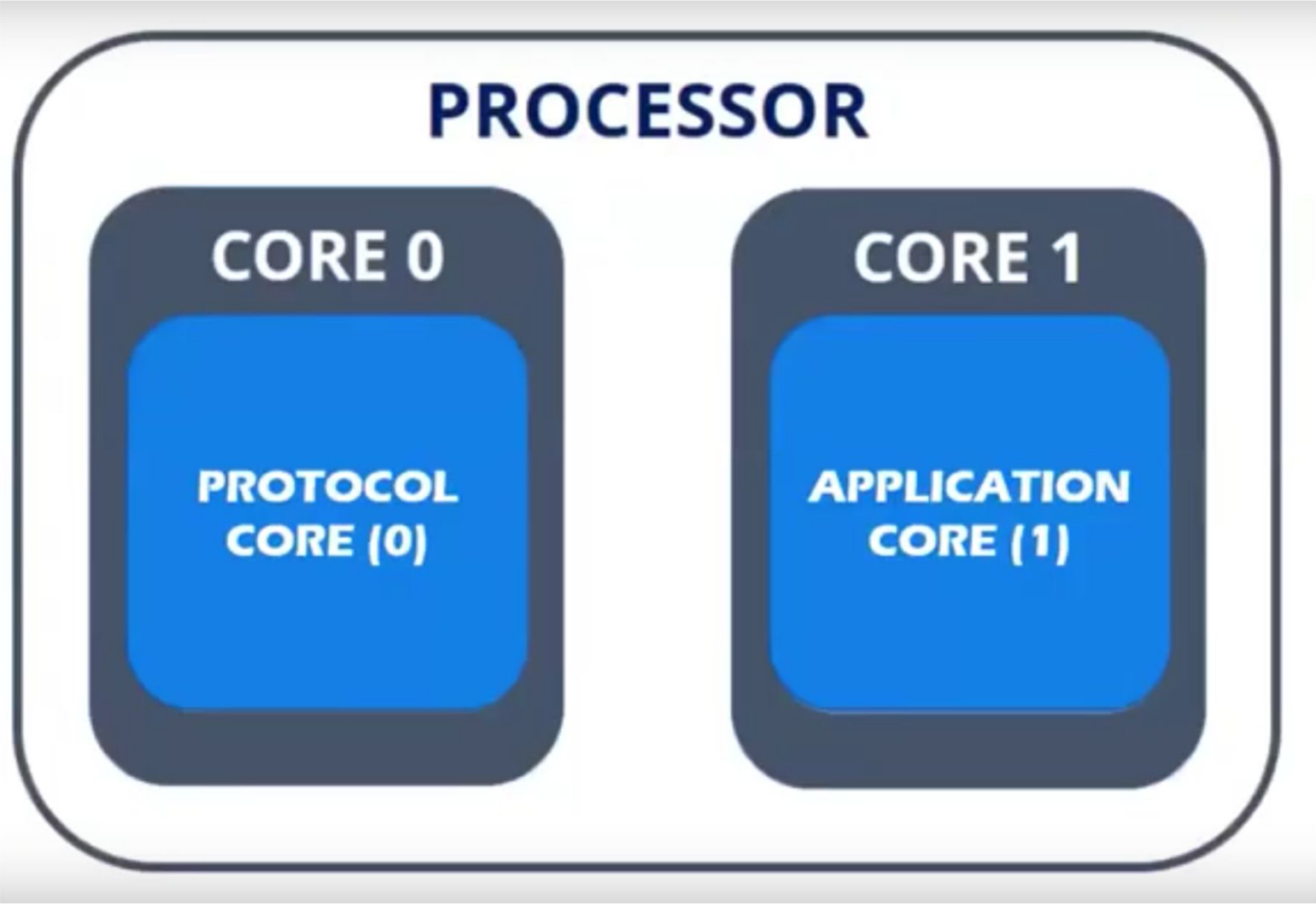


http://esp32.net/

https://learnesp32.com/

http://www.esp32learning.com/

O Hardware: ESP32 Expressif



FreeRTOS

O FreeRTOS (Amazon RTOS)

- Criado por volta do ano 2000 por Richard Barry, mantindo pela empresa Real Time Engineers Ltd até sua compra pela Amazon em 2017.

- Código aberto RTOS mais amplamente utilizado no mundo.
- É simples, pequeno e extremamente portátil. Site do projeto, com muita documentação disponível http://www.freertos.org/
- O código-fonte pode ser baixado em: http://sourceforge.net/projects/freertos/files/
  https://github.com/FreeRTOS/FreeRTOS

https://aws.amazon.com/pt/freertos/

O FreeRTOS (Amazon RTOS)

- Projetado para ser pequeno, simples e fácil de usar.
- Escrito em C, extremamente portátil.
- Suporta mais de 30 arquiteturas diferentes.
- Em uma configuração típica, o kernel do FreeRTOS pode ocupar de 4 KB a 9 KB de código (ROM / flash) e cerca de 200 bytes de dados (RAM).

The FreeRTOS

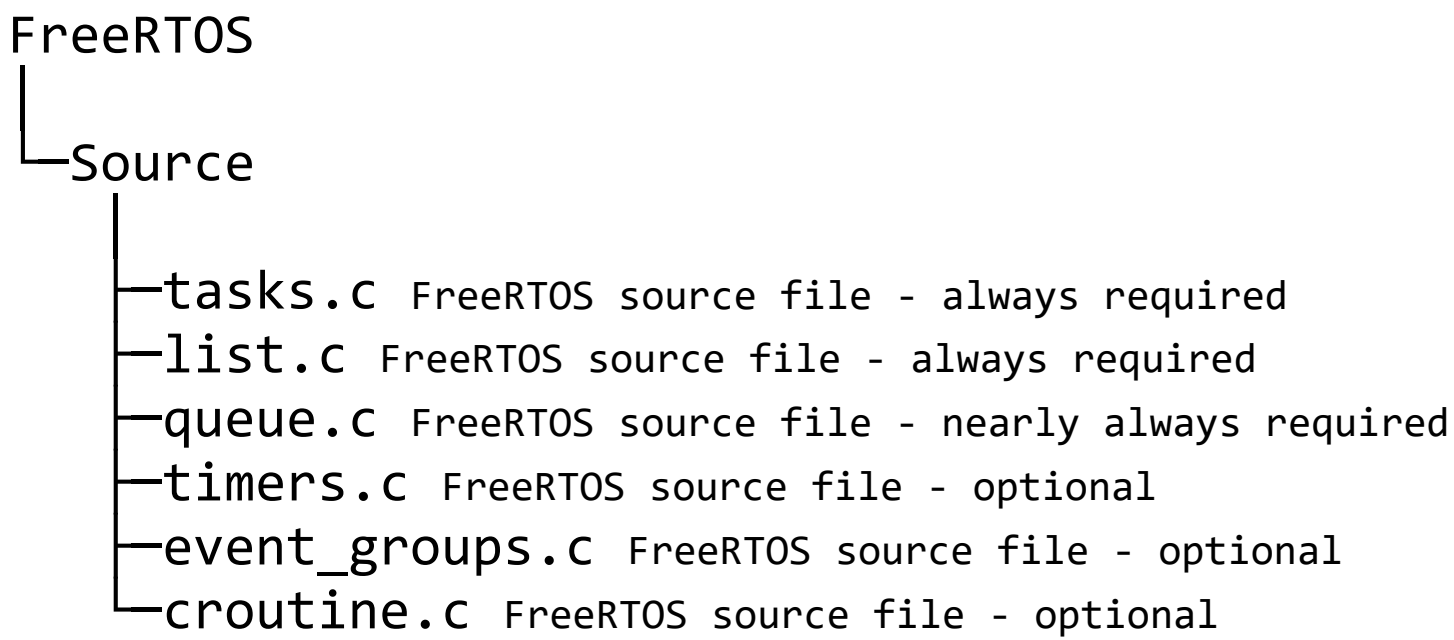
- O kernel pode trabalhar preemtivamente ou colaborativamente.
- Mutex com suporte de herança de prioridade.
- Recursos de rastreamento e detecção de estouro de pilha.
- Nenhuma restrição no número de *tasks* que podem ser criadas ou no número de prioridades que podem ser usadas.

The FreeRTOS

- Vários projetos de demonstração e aplicações para facilitar o aprendizado.
- Código aberto, sem *royalties* e com fórum gratuito disponível.
- Ferramentas de desenvolvimento abertas e gratuitas.
- Grande comunidade de usuários.
- Suporte e licença comercial se necessário

https://www.freertos.org/a00106.html

The FreeRTOS files



The FreeRTOS files

- tasks.c
  - *Task* básicas de manipulação
- list.c
  - Implementa uma função de lista para armazenar outros recursos
- queue.c
  - fornece serviços de fila e semáforo. *queue.c* quase sempre é necessário.
- timers.c
  - fornece funcionalidade de temporizador de software.
- event_groups.c
  - fornece funcionalidade de grupo de eventos.
- croutine.c
  - implementa a funcionalidade de co-rotina do FreeRTOS. As co-rotinas foram projetadas para uso em microcontroladores muito pequenos, raramente são usadas agora e, portanto, não são mantidas no mesmo nível que outros recursos do FreeRTOS.

Slide 36

M1 Marcos; 08/06/2021

Fabricantes

- Altera
- Atmel
- Cadence
- Cortus
- Cypress
- Energy Micro (see Silicon Labs)
- Freescale
- Imagination/MIPS
- Infineon
- Luminary Micro
- Microchip
- NEC
- Microsemi (formally Actel)
- NXP
- Renesas
- RISC-V
- Silicon Labs
- Spansion (ex Fujitsu)
- ST Microelectronics
- Synopsys ARC
- Texas Instruments
- Xilinx
- x86 (real mode)
- x86 / Windows Simulator
- Unsupported and contributed ports

FreeRTOS config.h

```
#define configUSE_PREEMPTION 1
#define configCPU_CLOCK_HZ 58982400
#define configTICK_RATE_HZ 250
#define configMAX_PRIORITIES 5
#define configMINIMAL_STACK_SIZE 128
#define configTOTAL_HEAP_SIZE 10240
#define configMAX_TASK_NAME_LEN 16
#define configUSE_MUTEXES 0
...
#define INCLUDE_vTaskDelete 1
#define INCLUDE_vTaskDelay 1
#define INCLUDE_xTaskGetCurrentTaskHandle 1
#define INCLUDE_uxTaskGetStackHighWaterMark 0
#define INCLUDE_xTaskGetIdleTaskHandle 0
...

//http://www.freertos.org/a00110.html
```

FreeRTOS - convenções

- Variáveis unsigned começam com “u”
- Variáveis do tipo char (8 bits) começam com “c”
- Variáveis do tipo short (16 bits) começam com “s”
- Variáveis do tipo do tipo long (32 bits) começam com “l”
- Ponteiros começam com “p”
- Funções privadas em um arquivo começam com “prv”
- As funções da API são prefixadas com seu tipo de retorno, conforme a convenção definida para variáveis, com a adição do prefixo v for void.
- Os nomes das funções da API começam com o nome do arquivo no qual estão definidos. Por exemplo vTaskDelete é definido em tasks.c e possui um tipo de retorno void.

Fonte: Curso udemy

Diagrama de estados de Tasks

Fig. 21.3 – Diagrama de estados de um sistema monotarefa.

As tarefas podem estar em estados diferentes:

•Bloqueado – a tarefa está aguardando um evento (por exemplo, tempo limite de atraso, disponibilidade de dados/recursos)

•Pronto – a tarefa está pronta para executar na CPU, mas não está executando porque a CPU está em uso por outra tarefa

•Execução – a tarefa está atribuída para ser executada na CPU

Diagrama de estados de Tasks

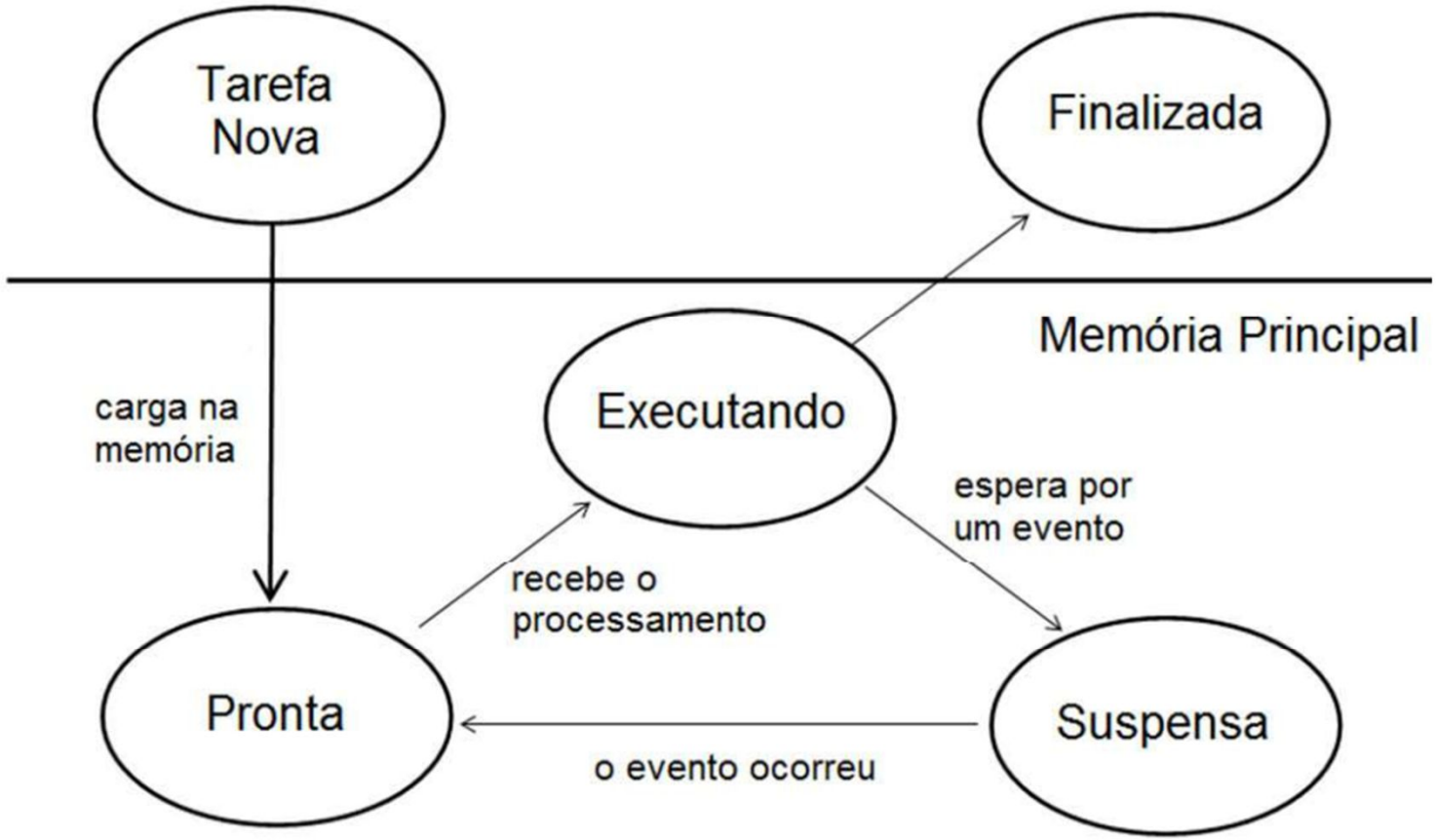
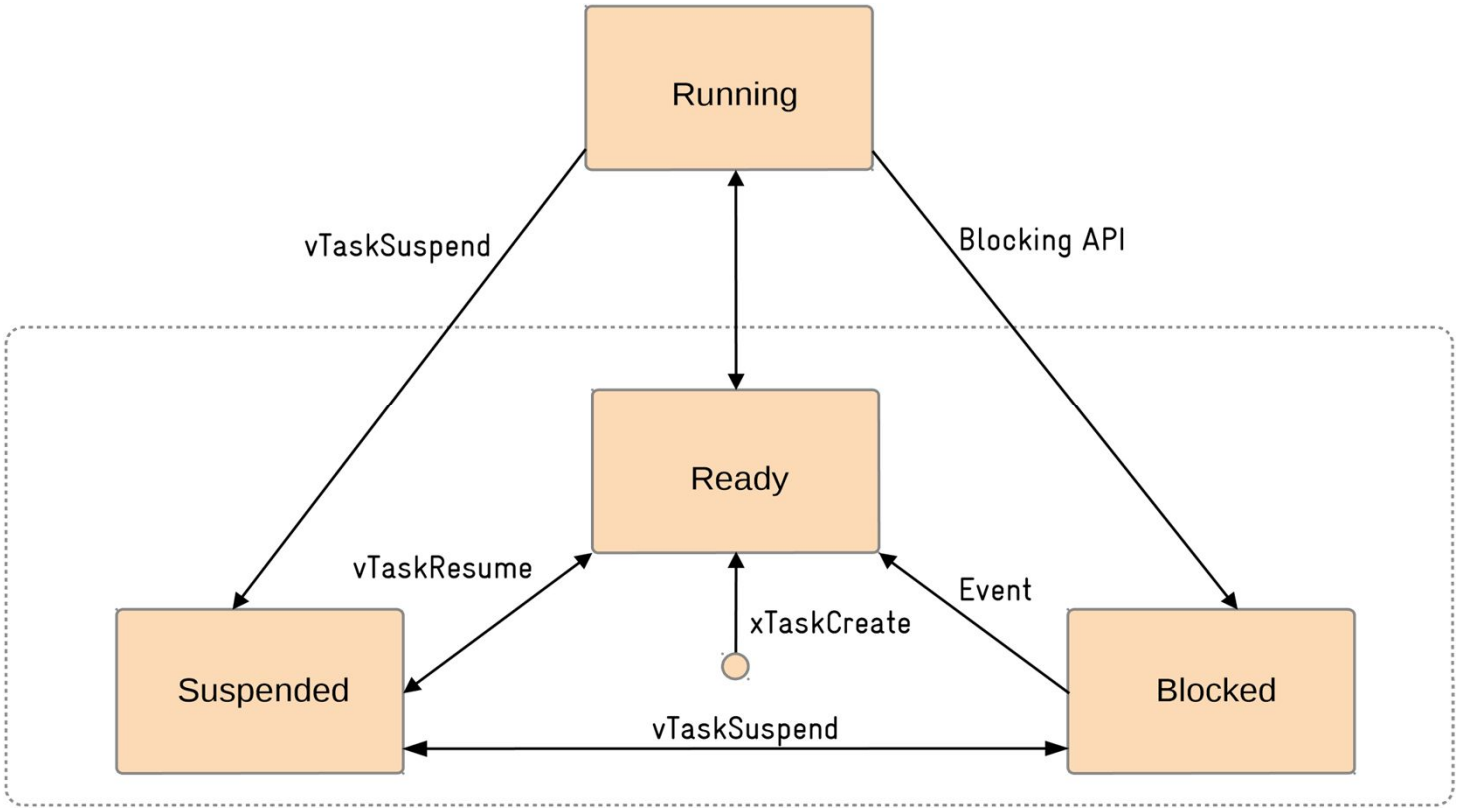


Fig. 21.4 – Diagrama de estados das tarefas em um sistema multitarefa.

Diagrama de estados de Tasks



Criando uma Task

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

```
xTaskCreate(
  task_sensor    /* Funcao a qual esta implementado o que a tarefa deve fazer */
  , (const portCHAR *)"sensor"   /* Nome (para fins de debug, se necessário) */
  , 128          /* Tamanho da stack (em words) reservada para essa tarefa */
  , NULL                  /* Parâmetro passado durante a criação */
  , 3                   /* Prioridade */
  , NULL );               /* Handle da tarefa, endereçamento desta task */
```

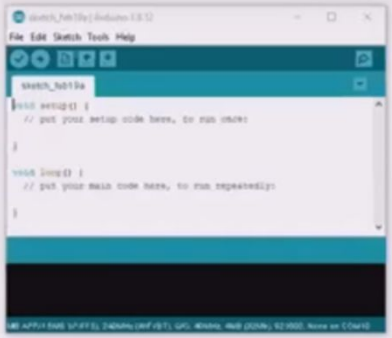
https://www.freertos.org/a00125.html

Creating a task

```
#include "task.h"
/* create a new task and add it to the list of tasks that are
ready to run */
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,
    const char *const pcName,
    unsigned short usStackDepth,
    void *pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t *pvCreatedTask );
```

```
void loopTask(void *pvParameters) {

  setup();
  for (;;) {
    loop();
  }
}
```



```
extern "C" void app_main() {

  initArduino();
  xTaskCreatePinnedToCore(
    loopTask,          // function to run
    "loopTask",        // Name of the task
    8192,              // Stack size (bytes!)
    NULL,              // No parameters
    1,                 // Priority
    &loopTaskHandle,   // Task Handle
    1);                // ARDUINO_RUNNING_CORE
}
```

Deletando uma Task

```
void vOtherFunction( void )
{
TaskHandle_t xHandle = NULL;

    // Create the task, storing the handle.
    xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL, tskIDLE_PRIORITY, &xHandle );

    // Use the handle to delete the task.
    if( xHandle != NULL )
    {
        vTaskDelete( xHandle );
    }
}
```

https://www.freertos.org/a00126.html

Deletando uma Task

```c
#include "task.h"
void ATaskFunction(void *pvParameters)
{
    for(;;)
    {
        /* task code */
    }
    vTaskDelete(NULL);
}
```

Parando e reiniciando uma Task

```
void vTaskSuspend( TaskHandle_t xTaskToSuspend );

void vTaskResume( TaskHandle_t xTaskToResume );

xYieldRequired = xTaskResumeFromISR( xHandle );
```

https://www.freertos.org/a00130.html

https://www.freertos.org/a00131.html

https://www.freertos.org/taskresumefromisr.htm

Passando parâmetros para uma tarefa

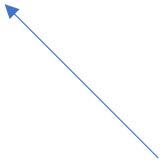
void vTaskCode(void *pvParameters);

pvParameters: Um valor que será passado para a tarefa criada como parâmetro da tarefa.

Se **pvParameters** estiver definido como o endereço de uma variável, a variável ainda deverá existir quando a tarefa criada for executada – portanto, não é válido transmitir o endereço de uma variável de pilha.

Escolha de núcleo

```
xTaskCreatePinnedToCore
  (vTask2,
   "TASK2",
   configMINIMAL_STACK_SIZE+1024,
   NULL,
   2,
   &task2Handle,
   PRO_CPU_NUM);
```



núcleo

Utilitários de tarefas:

uxTaskGetStackHighWaterMark: Retorna a quantidade de espaço restante na pilha de uma tarefa

xTaskGetCurrentTaskHandle : retorna uma referência para a atual tarefa em execução.

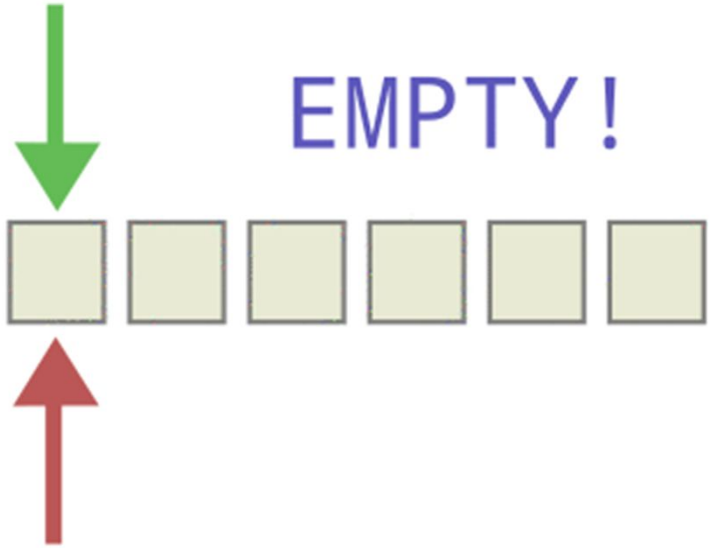
xTaskGetTickCount: retorna o tempo decorrido desde a inicialização do escalonador;

xTaskGetSchedulerState: retorna o estado do escalonador

uxTaskGetNumberOfTasks : retorna o número de tarefas do sistema

Filas

Conceitos de filas e seu uso em comunicação entre tarefas



https://www.filipeflop.com/blog/fila-com-freertos/

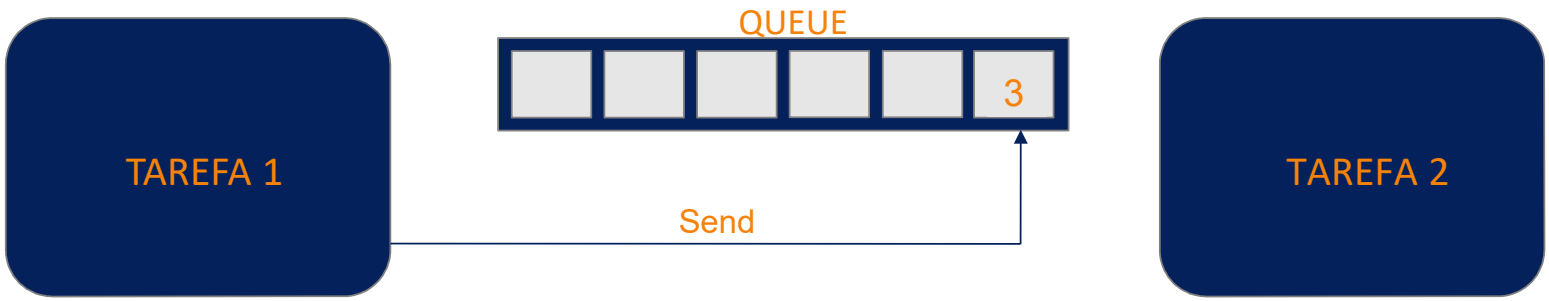
Filas

O que são filas?

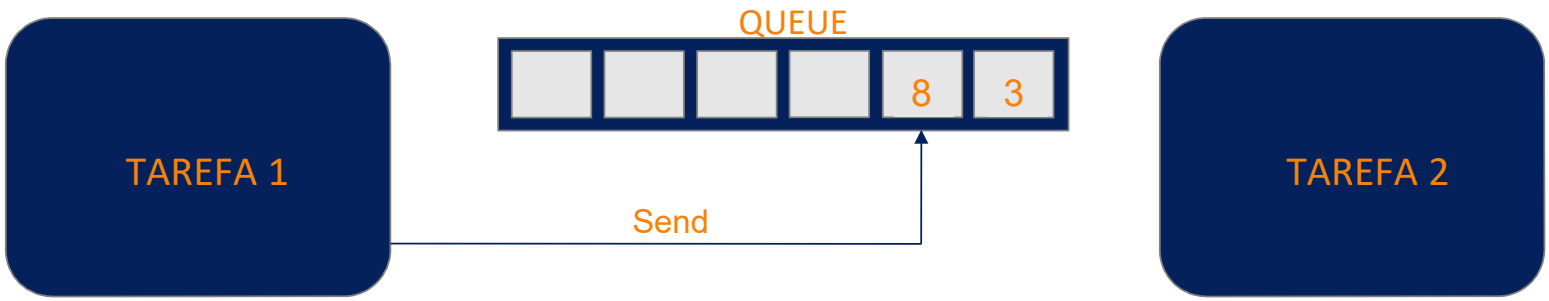
IN

OUT

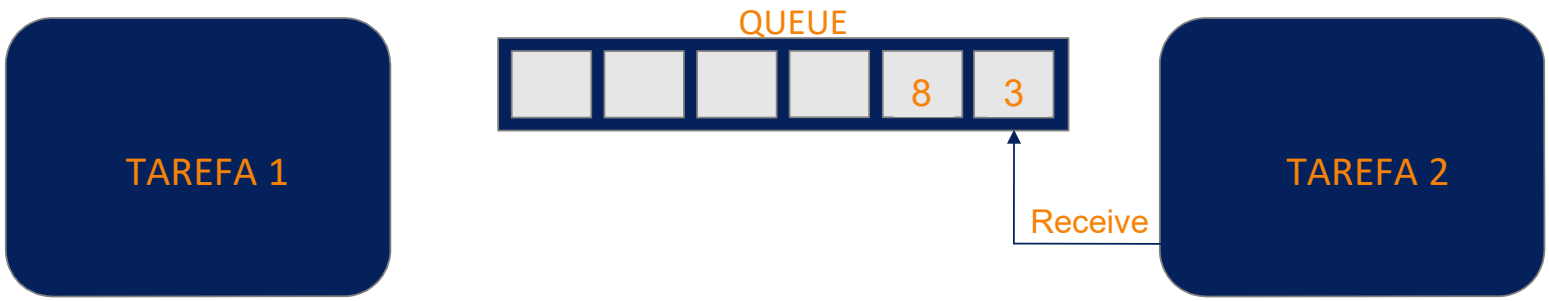
Exemplo Filas



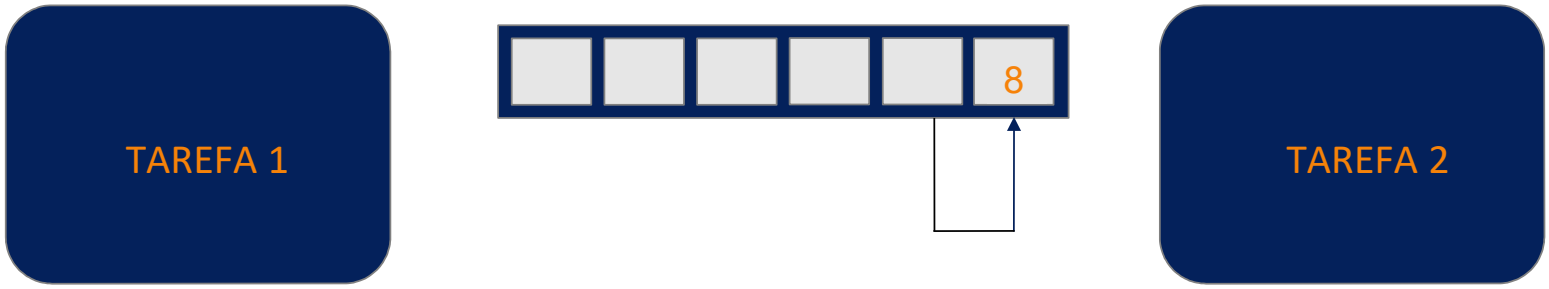
Examplo Filas



Exemplo Filas



Exemplo Filas



Filas

- São usadas para troca de mensagens entre tarefas ou entre uma interrupção e uma tarefa.
- Não pertence a nenhuma tarefa e podem ser acessadas por diversas tarefas e interrupções
- Possuem quantidade de elementos finitos, definido na criação da fila
- Os tamanhos dos elementos são fixos, definido na criação
- Passagem de valores por cópia ou referência

Filas

O que são filas?

Uma fila é uma estrutura de dados dinâmica que admite remoção de elementos e inserção de novos objetos.
Mais especificamente, uma fila (*queue*) é uma estrutura sujeita “a seguinte regra de operação: sempre que houver uma remoção, o elemento removido é o que está na estrutura há mais tempo.

Em outras palavras, o primeiro objeto inserido na fila é também o primeiro a ser removido. Esta política é também conhecida pela siga FIFO (= First-In-First-Out).

Comunicação entre tarefas (ITC)

- Fila de mensagens
  - Fila do “primeiro a entrar, primeiro a sair” (FIFO) para passar dados
  - Os dados podem ser enviados por cópia ou por referência (ponteiro)
  - Usado para enviar dados entre tarefas ou entre interrupção e tarefa
- Semáforo
  - Pode ser tratado como um contador de referência para registrar disponibilidade de um recurso particular
  - Pode ser um semáforo binário ou de contagem
  - Usado para proteger o uso de recursos ou sincronizar a execução de tarefas
- Mutex
  - Semelhante ao semáforo binário, geralmente usado para proteger o uso de um único recurso (exclusão mútua)
  - O mutex FreeRTOS vem com um mecanismo de herança de prioridade, para evitar o problema de inversão de prioridade (condição quando a tarefa de alta prioridade termina esperando pela tarefa de baixa prioridade).
- Caixa de correio
  - Local de armazenamento simples para compartilhar uma variável única
  - Pode ser considerada com uma fila de elemento único
- Grupo de evento
  - Grupo de condições (disponibilidade de semáforo, fila, sinalizador de evento, etc.)
  - A tarefa pode ser bloqueada e pode esperar que uma condição de combinação específica seja atendida
  - Disponível no Zephyr como API de sondagem, no FreeRTOS como QueueSets

https://www.digikey.com.br/pt/articles/real-time-operating-systems-and-their-applications

Para inicializar uma fila, faça conforme a seguir:

```
xQueue_Teste = xQueueCreate( NUMERO_ITENS_FILA,
TAMANHO_DE_CADA_ITEM );
```

Onde:

- **NUMERO_ITENS_FILA:** quantidade total de itens que você deseja que sua fila possua. Lembre-se que mais itens significa que mais memória RAM é ocupada.
- **TAMANHO_DE_CADA_ITEM:** tamanho **(em bytes)** de cada item da fila. Por exemplo, se cada item de sua fila for um número inteiro, este campo deverá ser igual a sizeof(int).

Segue abaixo as principais formas de inserção de itens numa fila no FreeRTOS:

•**xQueueSend:** adiciona elemento a uma fila. Esta função **não** deve ser utilizada dentro do tratamento de uma interrupção (ou dentro de callbacks).
•**xQueueSendFromISR:** adiciona elemento a uma fila. Esta função **deve ser somente usada** dentro do tratamento de uma interrupção (ou dentro de callbacks).
•**xQueueOverwrite:** sobrescreve o primeiro elemento de uma fila. Essa função é especialmente útil quando se utiliza uma fila de um único elemento, onde somente o valor mais recente (última leitura de um sensor, por exemplo) é que importa ser mantido. Tipicamente, as filas que usam esse tipo de inserção são filas unitárias (de um só item).
Esta função **não** deve ser utilizada dentro do tratamento de uma interrupção (ou dentro de callbacks).
•**xQueueOverwriteFromISR:** análogo ao anterior, ou seja, sobrescreve o primeiro elemento de uma fila, porém deve ser usada **somente dentro de um tratamento de interrupção**(ou dentro de callbacks). Exatamente como o caso acima, essa função é especialmente útil quando se utiliza uma fila de um único elemento, onde somente o valor mais recente (última leitura de um sensor, por exemplo) é que importa ser mantido. Tipicamente, as filas que usam esse tipo de inserção são filas unitárias (de um só item).

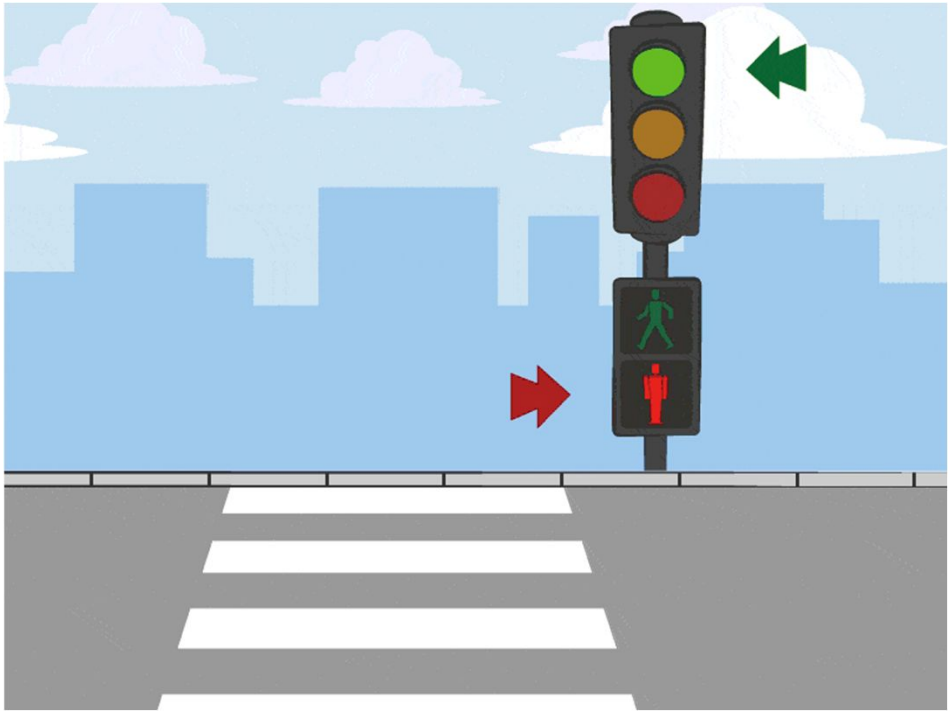
https://www.filipeflop.com/blog/fila-com-freertos/

Formas de ler / remover itens na fila com FreeRTOS

•**xQueueReceive:** lê um elemento da fila. Esta função **não** deve ser utilizada dentro do tratamento de uma interrupção (ou dentro de callbacks).
•**xQueueReceiveFromISR:** lê um elemento da fila. Esta função **somente deve ser utilizada** dentro do tratamento de uma interrupção (ou dentro de callbacks).
•**xQueuePeek:** faz a leitura do elemento da fila, porém, sem retirá-lo dela. Isso é útil quando a tarefa deseja verificar se a informação na fila deve ou não ser tratada por ela, sem alterar nada da fila para isso. Em analogia livre, é como "dar uma espiadinha" no item a ser lido / removido da fila.
Esta função **não** deve ser utilizada dentro do tratamento de uma interrupção (ou dentro de callbacks).
•**xQueuePeekFromISR:** análogo ao anterior, ou seja, faz a leitura o elemento da fila, porém sem retirá-lo dela, porém **somente deve ser utilizada** dentro do tratamento de uma interrupção (ou dentro de callbacks).
Isso é especialmente útil quando a tarefa deseja verificar se a informação na fila deve ou não ser tratada por ela, sem alterar nada da fila para isso.

Semáforo no FreeRTOS.

Um **semáforo** é um recurso disponibilizado pelo FreeRTOS para permitir que **recursos únicos** (uma porta serial, um GPIO, etc.) tenham um **controle de acesso**, de modo a poderem ser **compartilhados entre várias tarefas distintas**. Ou seja, os semáforos servem para controlar o acesso a recursos quando pode haver a chance, remota ou não, de um recurso (como uma interface de comunicação ou um GPIO, por exemplo) ser usado por duas ou mais tarefas simultaneamente.



https://www.filipeflop.com/blog/semaforo-no-freertos/

Semáforo

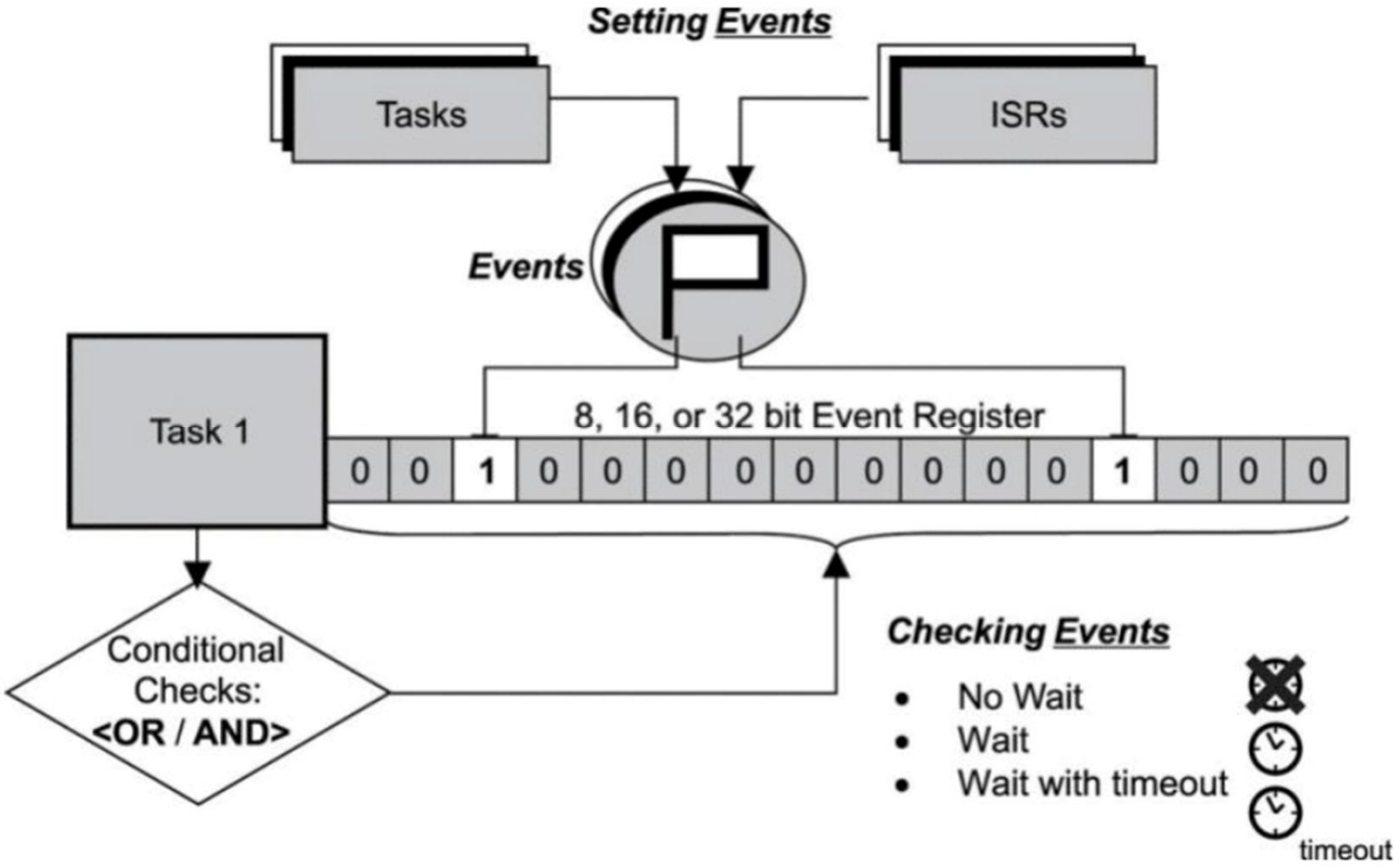
O semáforo é um mecanismo de sincronização entre tarefas. Funciona como uma guarda para a tarefa executar uma operação sincronizada ou acessar um recurso compartilhado. Desse modo, antes de executar tal ação, a tarefa deve solicitar o semáforo responsável pela guarda da ação. Caso o semáforo esteja disponível, a tarefa realiza a ação, caso contrário, a tarefa é bloqueada até que o semáforo seja liberado.

Binário: É o mais simples dos semáforos, como uma única variável valendo “1“ ou “0“. Quando em “1”, permite que tarefas o obtenham e quando “0”, a tarefa não conseguirá obtê-lo.

Mutex é uma estrutura parecida com o semáforo binário. A única diferença entre os dois é que o mutex implementa o mecanismo de herança de prioridade, o qual impede que uma tarefa de maior prioridade fique bloqueada à espera de um mutex (inversão de prioridade).

Counting: Similar ao binário mas conta com uma fila de valores, similar a um vetor (array). É muito utilizado para minimizar problemas entre ISR e os outros semáforos, já que se ocorrer mais de uma ISR antes que a tarefa o obtenha, perderemos essa ISR visto que os outros semáforos só têm um ”espaço”. Utilizando o semáforo counting, não perdemos a ISR já que temos vários “espaços”

Grupo de eventos



FreeRTOS oferece, além dos 2 métodos anteriores para comunicação e sincronização de tarefas, os grupos de eventos (abreviando para G.E.). Um G.E. pode armazenar diversos eventos (flags) e, com isso, podemos deixar uma tarefa em espera (estado bloqueado) até que um ou vários eventos específicos ocorram.

Referências:

Galvão, S.S.L.: Especificação do micronúcleo FreeRTOS utilizando Método B. Master Thesis, DIMAp/UFRN (2010)

https://www.embarcados.com.br/rtos-para-iniciantes-com-arduino-e-freertos/

https://www.digikey.com.br/pt/articles/real-time-operating-systems-and-their-applications

https://blog.eletrogate.com/freertos-semaforo-binario-compartilhamento/

https://blog.eletrogate.com/freertos-sinalizando-eventos-com-event-bit-e-group/

The work "Embedded System Design: From Electronics to Microkernel Development" of Rodrigo Maximiano Antunes de.

- Adapted from original work by *Sergio Prado – Embedded Labworks*
- Added info from *Mastering the FreeRTOS™ Real Time Kernel – Richard Barry*

ATIY LIVE 3 || CURSO DE FREERTOS PARTE 1. https://www.youtube.com/watch?v=uoYSw7MpzDY

ebook: **Coleção ESP32 do Embarcados** - Parte 2.

https://github.com/FBSeletronica

https://github.com/FBSeletronica/Curso-primeiros-passos-com-freeRTOS-Codigos