# FreeRTOS

Embedded **Labworks**

Por Sergio Prado. São Paulo, Outubro de 2016

# SOBRE O INSTRUTOR

- Sergio Prado tem mais de 20 anos de experiência em desenvolvimento de software para sistemas embarcados, em diversas arquiteturas de CPU (ARM, PPC, MIPS, x86, 68K), atuando em projetos com Linux embarcado e sistemas operacionais de tempo real.

- É sócio da Embedded Labworks, onde atua com consultoria, treinamento e desenvolvimento de software para sistemas embarcados:
  http://e-labworks.com

- Mantém um blog pessoal sobre sistemas embarcados em:
  http://sergioprado.org

# AGENDA DO TREINAMENTO

- **DIA 1**: Introdução, hardware e ambiente de desenvolvimento, estudando e integrando os fontes do FreeRTOS, trabalhando com tarefas, prioridades e escalonamento, rotinas de delay, idle task, comunicação entre tarefas com queues.

- **DIA 2**: Interrupção e mecanismos de sincronização, semáforos binários e contadores, task notifications, deferindo trabalho com queues, protegendo e controlando acesso a recursos com mutex, tarefas gatekeeper, gerenciamento de memória, trabalhando com o stack e com o heap.

- **DIA 3**: Software timers, co-routines, queue sets, event groups, integração com bibliotecas, ferramentas de depuração, estatísticas de execução, projetando com um RTOS, projeto final.

Embedded Labworks

# FreeRTOS

## Introdução a sistemas de tempo real

# SISTEMAS DE TEMPO REAL

- No contexto de 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 restrições de tempo, ou seja, possuem um prazo ou tempo-limite para o estímulo ser processado e gerar a saída correspondente.
- Estes tipos de sistemas são chamados “Sistemas de Tempo Real”.

# SISTEMAS DE TEMPO REAL (cont.)

- Portanto, um sistema de tempo real precisa garantir com que todos os eventos sejam atendidos dentro das suas respectivas restrições de tempo.

- É por isso que um sistema de tempo real está relacionado ao **determinismo**, e 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 às consequências em não respeitar estas restrições.

# CLASSIFICAÇÃO

- **Soft real-time**: Uma restrição de tempo não atingida tem como consequência a percepção de baixa qualidade do sistema. Exemplo: um display com touch que demora para responder ao tocar na tela.

- **Hard real-time**: Uma restrição de tempo não atingida pode inutilizar o sistema ou provocar consequências catastróficas. Exemplo: um sistema de airbag que não responde no tempo correto no momento da colisão de um veículo.

Embedded **Labworks**

# DESENVOLVIMENTO

## Como desenvolver um sistema com características de tempo real?

# FOREGROUND/BACKGROUND

- Projetos pequenos e de baixa complexidade são desenvolvidos como sistemas foreground/background (também chamados de **super-loop**).
  - A aplicação consiste em um loop infinito que chama algumas funções para realizar as operações desejadas (**background**).
  - E rotinas de tratamento de interrupção tratam eventos assíncronos (**foreground**).

# O SUPER-LOOP

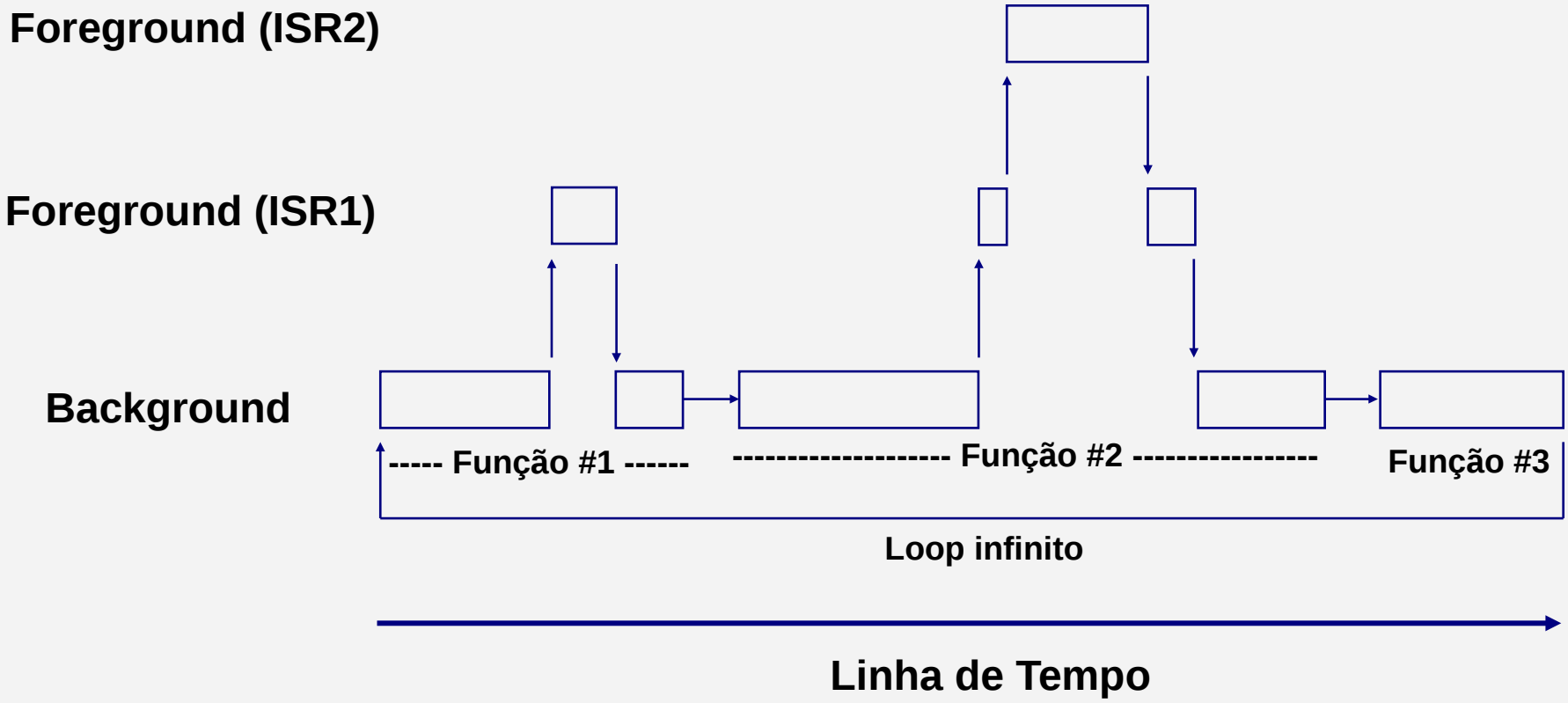
```
int main(void)
{
    init_hardware();

    while (1) {
        I2C_ReadPkg();
        UART_Receive();
        USB_GetPacket();
        Audio_Play();
        ADC_Convert();
        LCD_Show();
        ...
    }
}
```

**BACKGROUND**

```
void USB_ISR(void)
{
    Limpa interrupção;
    Lê pacote;
}


void UART_ISR(void)
{
    Limpa interrupção;
    Trata byte recebido;
}
```

**FOREGROUND**

# O SUPER-LOOP (cont.)



# VANTAGENS DO SUPER-LOOP

- Fácil e rápido de desenvolver.
- Não requer treinamento ou conhecimento de API's específicas de um sistema operacional.
- Não consome recursos adicionais comparado à solução com um sistema operacional.
- Solução ótima em projetos pequenos e com requisitos modestos de restrições de tempo.

# DEFICIÊNCIAS DO SUPER-LOOP

- Difícil garantir que uma operação irá ser executada dentro das restrições de tempo.
- Todo o código em background tem a mesma prioridade.
- Se uma das funções demorar mais do que o esperado, todo o sistema será impactado.

# DEFICIÊNCIAS DO SUPER-LOOP (cont.)

```
while (1) {
    ADC_Read();
    UART_Receive();
    USB_GetPacket();
    ...
}
```

```
void ADC_Read (void) {

    configure_ADC();

    while (conv_rdy == 0) {
        ;
    }

    ...
}
```

Delays e outras rotinas podem impactar todas as funções rodando em background.

# DEFICIÊNCIAS DO SUPER-LOOP (cont.)

- Tarefas de alta prioridade precisam ser colocadas em foreground (ISR).
- ISRs muito longas podem impactar o tempo de resposta do sistema.
- É difícil coordenar a execução de rotinas em background e em foreground.

# DEFICIÊNCIAS DO SUPER-LOOP (cont.)

```
while (1) {
    I2C_ReadPkg();
    UART_Receive();
    USB_GetPacket();
    Audio_Play();
    ADC_Convert();
    LCD_Show();
    ...
}
```

```
void  USB_ISR (void) {
    clear_interrupt();
    read_packet();
}
```

Se um pacote USB é recebido logo após a chamada desta função, o tempo de resposta será maior.

# DEFICIÊNCIAS DO SUPER-LOOP (cont.)

- Qualquer alteração em determinada parte do código pode impactar o tempo de resposta de todo o sistema.
- Difícil de garantir as restrições de tempo da aplicação.
- Sentimento de "medo" para alterar o código.
- Problemas podem aparecer quando o código é mantido por múltiplos desenvolvedores. Como controlar?

# A SOLUÇÃO

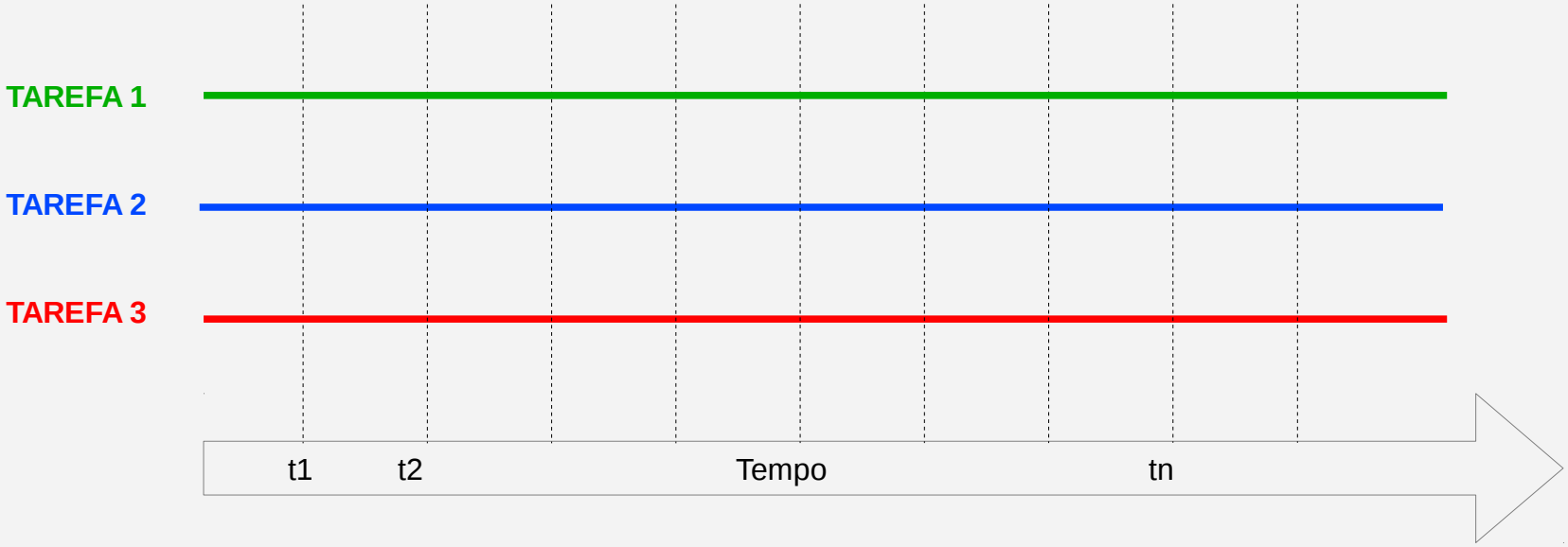
Precisamos então de uma solução que gerencie corretamente os requisitos de tempo real do sistema. É aí que entra o kernel de tempo real!

# O KERNEL DE TEMPO REAL

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

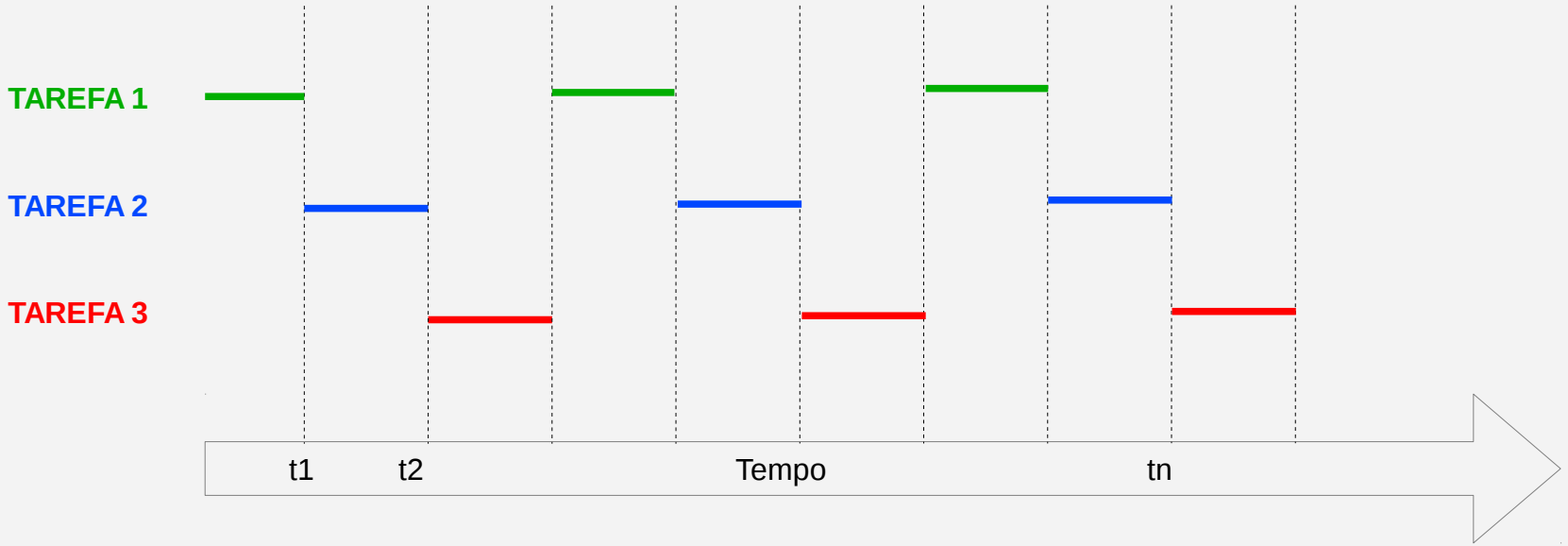
# MULTITAREFA

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



# MULTITAREFA (cont.)

- Mas como o processador só pode executar uma tarefa de cada vez (considerando-se uma CPU com apenas um núcleo), é realizado um chaveamento entre as tarefas:



# MULTITAREFA (cont.)

- Este chaveamento ou troca de tarefas pode acontecer em diferentes situações:
  - Uma tarefa pode bloquear esperando um recurso (Ex: porta serial) estar disponível ou um evento acontecer (Ex: receber um pacote da interface USB).
  - Uma tarefa pode dormir por um tempo (Ex: tarefas periódicas).
  - Uma tarefa pode ser suspensa involuntariamente pelo kernel. Neste caso, chamamos o kernel de preemptivo.
- Esta troca de tarefas também é chamada de mudança de contexto (**context switching**).

# MUDANÇA DE CONTEXTO

- Enquanto uma tarefa está em execução, ela possui determinado contexto (stack, registradores da CPU, etc).
- Ao mudar a tarefa 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 tarefas é realizado através de uma estrutura chamada TCB (**Task Control Block**).

# O ESCALONADOR

- O escalonador de tarefas entra em ação durante as mudanças de contexto.
- Ele é a parte do kernel responsável por decidir qual é a próxima tarefa a ser executada em determinado momento.
- O algoritmo responsável por decidir qual é a próxima tarefa a ser executada é chamado de política de escalonamento.

# ALGUMAS POLÍTICAS DE ESCALONAMENTO

- First-in, First-out ou FirstCome, FirstServed.
- Shortest remaining time.
- Round-robin scheduling.
- Fixed priority preemptive scheduling.
- Dynamic priority preemptive scheduling.

# KERNEL DE TEMPO REAL

- Além de gerenciar o uso da CPU, um kernel de tempo real possui normalmente outras responsabilidades, dentre elas:
  - Gerenciar a comunicação entre as tarefas.
  - Gerenciar a comunicação entre interrupções e tarefas.
  - Gerenciar o acesso aos recursos da aplicação (hardware, estruturas de dados, etc).
  - Gerenciar o uso de memória.
  - Prover outras funcionalidades como timers, tracing, etc.

# RTOS E KERNEL

- Para algumas literaturas, RTOS e kernel de tempo real são a mesma coisa.
- Para outras literaturas, além de um kernel de tempo real, um RTOS pode incluir outros serviços de alto nível como gerenciamento de arquivos, pilhas de protocolo, interface gráfica, etc.
- Para nós, trataremos RTOS e kernel de tempo real da mesma maneira!

# AS VANTAGENS DE UM RTOS

- A vantagem mais clara é a possibilidade de priorizar tarefas para garantir que as restrições de tempo da aplicação sejam atendidas.
- O uso de um RTOS permite que você trabalhe orientado a eventos, evitando por exemplo rotinas de polling, e melhorando a eficiência do sistema. Polling é igual a desperdício de CPU!
- Como você precisa dividir a aplicação em tarefas, incentiva uma melhor arquitetura e modularidade do sistema.

# AS VANTAGENS DE UM RTOS (cont.)

- Com um sistema modular, fica mais fácil dar manutenção.
- Facilita os testes, já que cada tarefa é uma entidade independente.
- Como cada tarefa tem uma interface bem definida, facilita o desenvolvimento em equipes com múltiplos desenvolvedores.
- Incentiva o reuso de código.
- Facilita o gerenciamento de energia.

# CONSUMO DE RECURSOS

- Um RTOS facilita bastante o desenvolvimento de sistemas com características de tempo real, mas ele pode não ser a solução ideal para determinado projeto:
  - Os serviços providos pelo kernel adicionam um overhead de execução, que pode variar entre 2% e 5% do uso da CPU, dependendo do RTOS.
  - Um RTOS precisa de um espaço extra de ROM (flash) para armazenar o código, que pode variar de algumas centenas de bytes até algumas centenas de kilobytes.
  - Um RTOS consome RAM para armazenar o contexto e o stack de cada tarefa, que pode variar de algumas centenas de bytes até algumas dezenas de kilobytes.

Embedded Labworks

# FreeRTOS

## Hardware e ambiente de desenvolvimento

# NXP FREEDOM PLATFORM

- Plataforma de desenvolvimento e prototipação de baixo custo da NXP ($12 a $50).

  http://www.nxp.com/freedom

- Ideal para prototipar sistemas com microcontroladores da linha Kinetis e sensores da NXP.

- Compatível com shields Arduino (revisão 3).

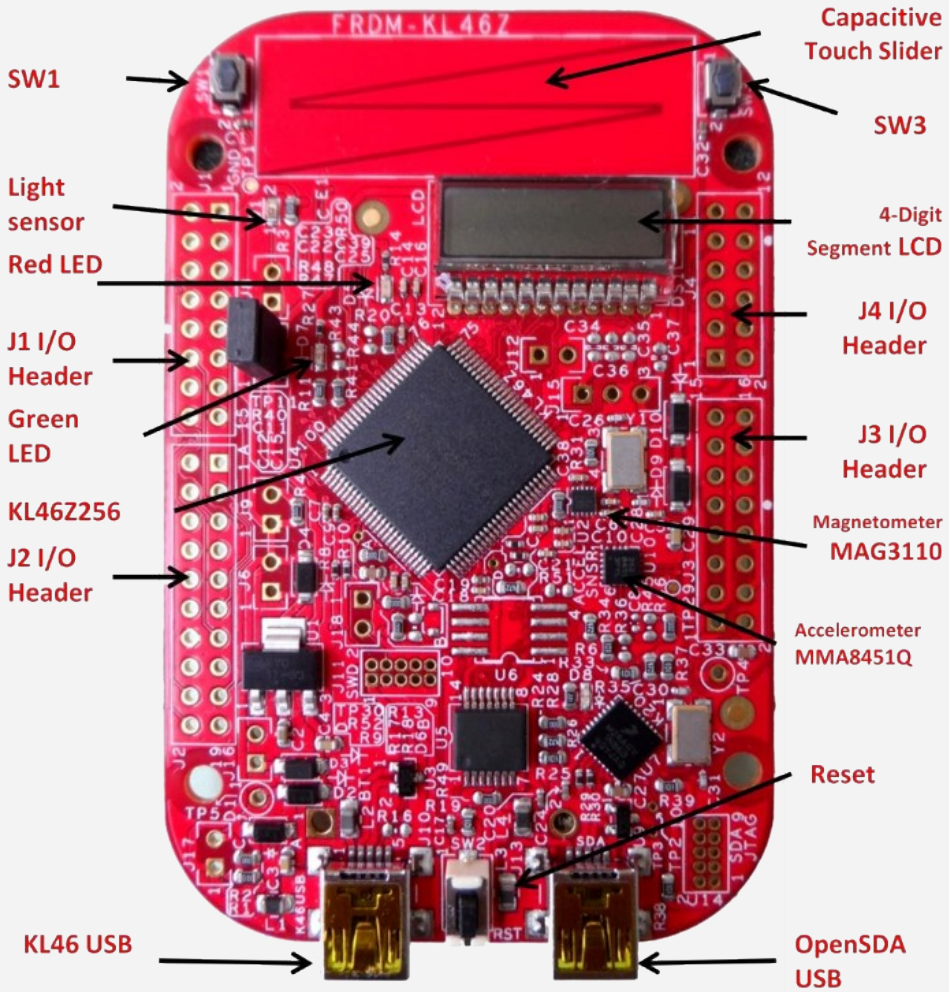
  http://arduino.cc/en/Main/ArduinoShields

# FRDM-KL46Z

# FRDM-KL46Z (cont.)

- MCU MKL46Z256VLL4 (ARM Cortex-M0+) rodando a até 48MHz, 256KB de flash e 32KB de RAM.
- Acelerômetro MMA8451Q e magnetômetro MAG3110.
- Sensor de luz ambiente e touch capacitivo deslizante.
- Display LCD de 4 dígitos, 2 leds e 2 botões.
- Interface USB device.
- Depuração e gravação integrada (OpenSDA).

# FRDM-KL46Z (cont.)



# DOCUMENTAÇÃO

- A documentação do hardware está disponível no ambiente de laboratório em `docs/hardware`:
  - FRDM-KL46Z - User's Manual.pdf (manual da placa).
  - FRDM-KL46Z Schematic.pdf (esquemático da placa).
  - KL46Z Data Sheet.pdf (datasheet do microcontrolador)
  - KL46Z Reference Manual.pdf (manual de referência do microcontrolador).

# REFERÊNCIAS

- Para mais informações sobre esta placa, consulte a página oficial da NXP:

  http://nxp.com/frdm-kl46z

- O site da comunidade de microcontroladores Kinetis da NXP também pode ajudar no caso de dúvidas ou problemas:

  http://community.nxp.com/community/kinetis

# AMBIENTE DE DESENVOLVIMENTO

- O Kinetis Design Studio é o ambiente de desenvolvimento para placas com microcontroladores da linha Kinetis da NXP.

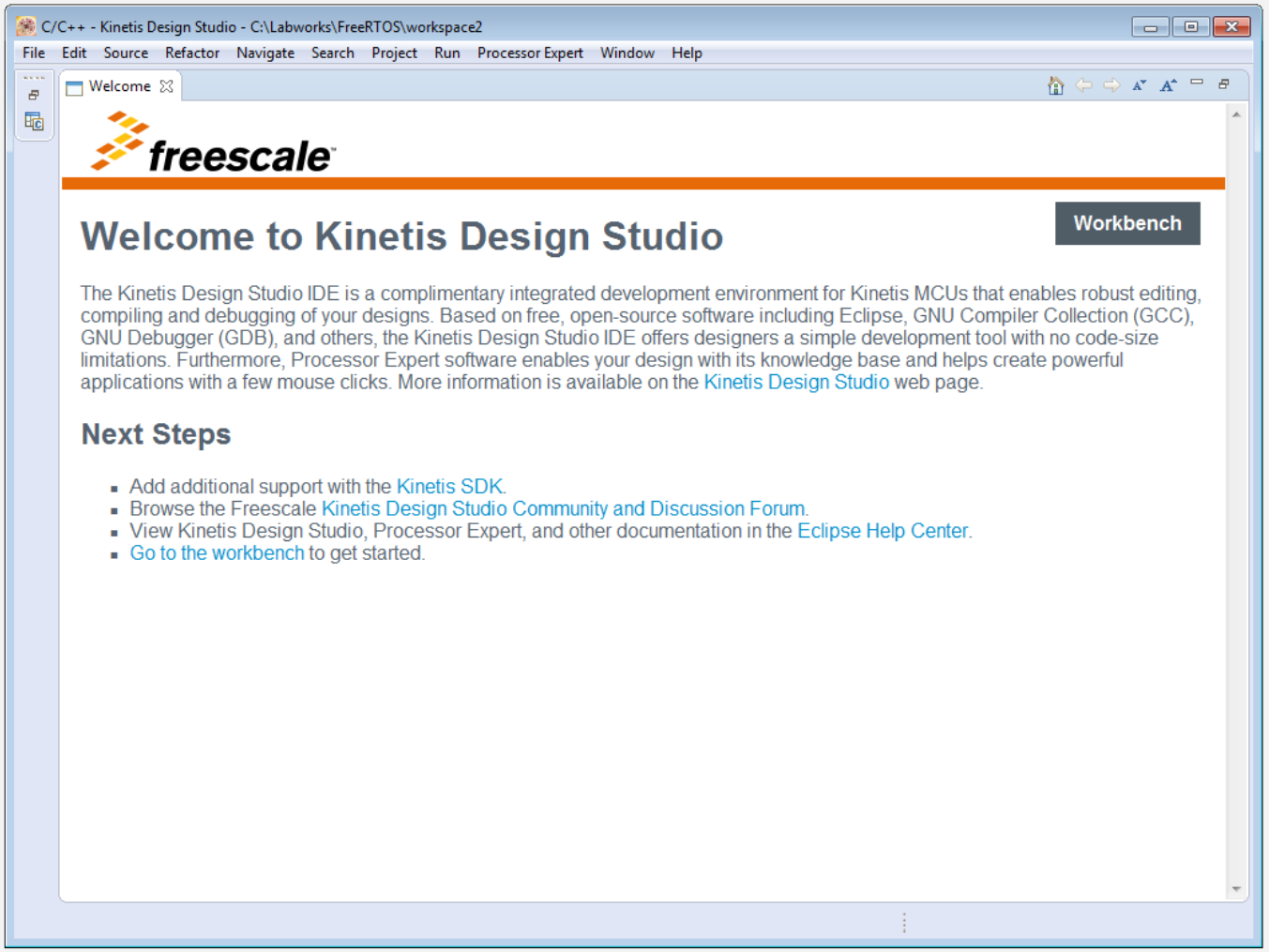
  http://nxp.com/kds

- Ferramenta gratuíta e sem limitação de código.

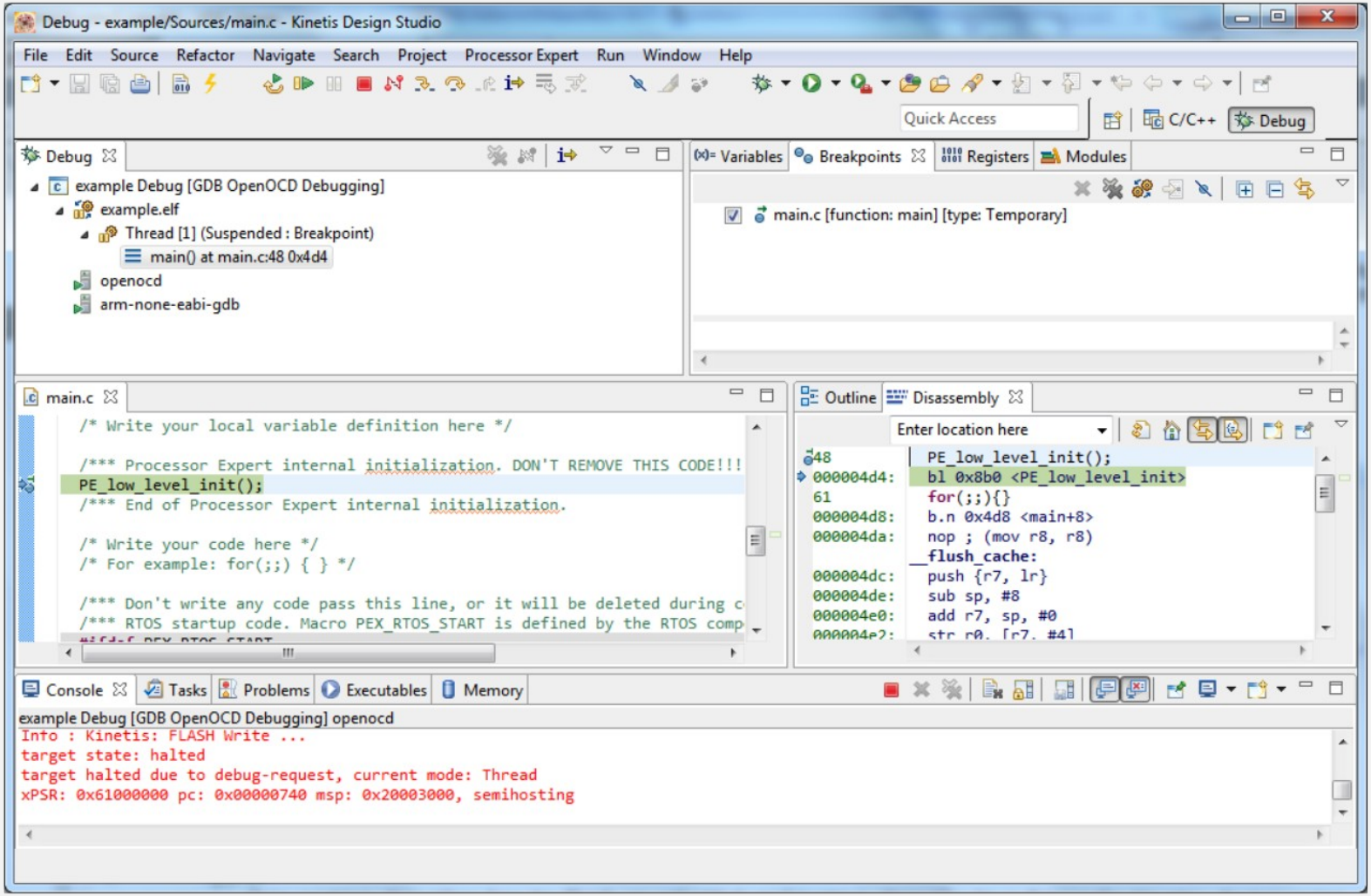
- É baseada no Eclipse e utiliza o GCC como toolchain.

- Suporte oficial para Windows 7/8/10, Mac OS X e Linux (Ubuntu, Redhat, Centos).

# KINETIS DESIGN STUDIO



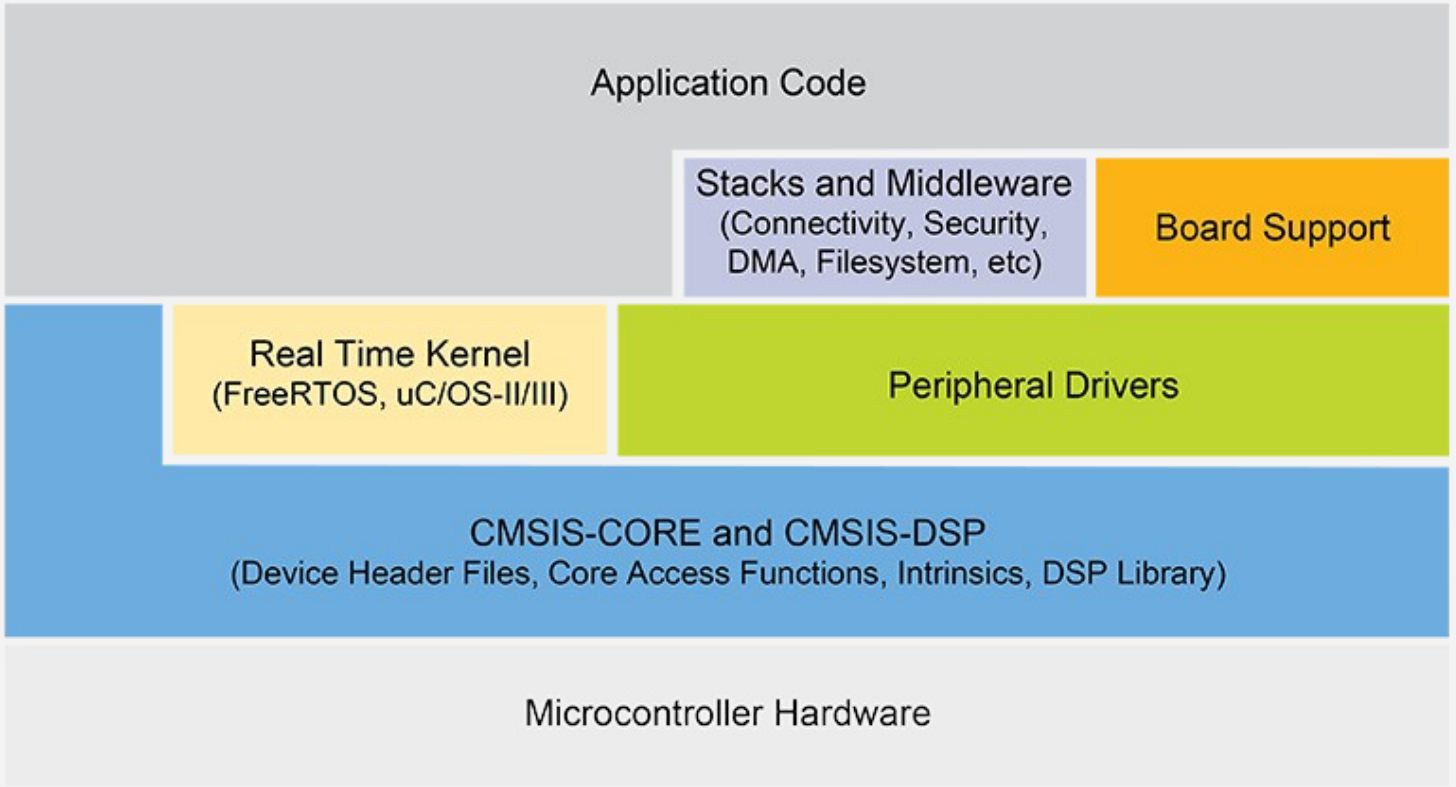
# KINETIS DESIGN STUDIO (cont.)



# KSDK

- O Kinetis Software Development Kit (KSDK) provê um conjunto de componentes de software e bibliotecas prontas para o desenvolvimento de aplicações com microcontroladores da linha Kinetis da NXP, incluindo:
  - Drivers de dispositivo.
  - Pilhas de protocolo.
  - CMSIS Core e DSP.
- Gratuito e código-fonte completo disponível.

# KSDK (cont.)


Application Code
Stacks and Middleware
(Connectivity, Security,
DMA, Filesystem, etc)
Board Support
Real Time Kernel
(FreeRTOS, uC/OS-II/III)
Peripheral Drivers
CMSIS-CORE and CMSIS-DSP
(Device Header Files, Core Access Functions, Intrinsics, DSP Library)
Microcontroller Hardware

# DOCUMENTAÇÃO

- A documentação do KDS está disponível no ambiente de laboratório em `docs/kds`:
  - Kinetis Design Studio v3.0.0 User Guide.pdf
- A documentação do KSDK também está disponível no ambiente de laboratório em `docs/ksdk`:
  - Getting Started with Kinetis SDK.pdf

Embedded **Labworks**

# LABORATÓRIO

## A primeira aplicação

Embedded Labworks

# FreeRTOS

## Introdução ao FreeRTOS

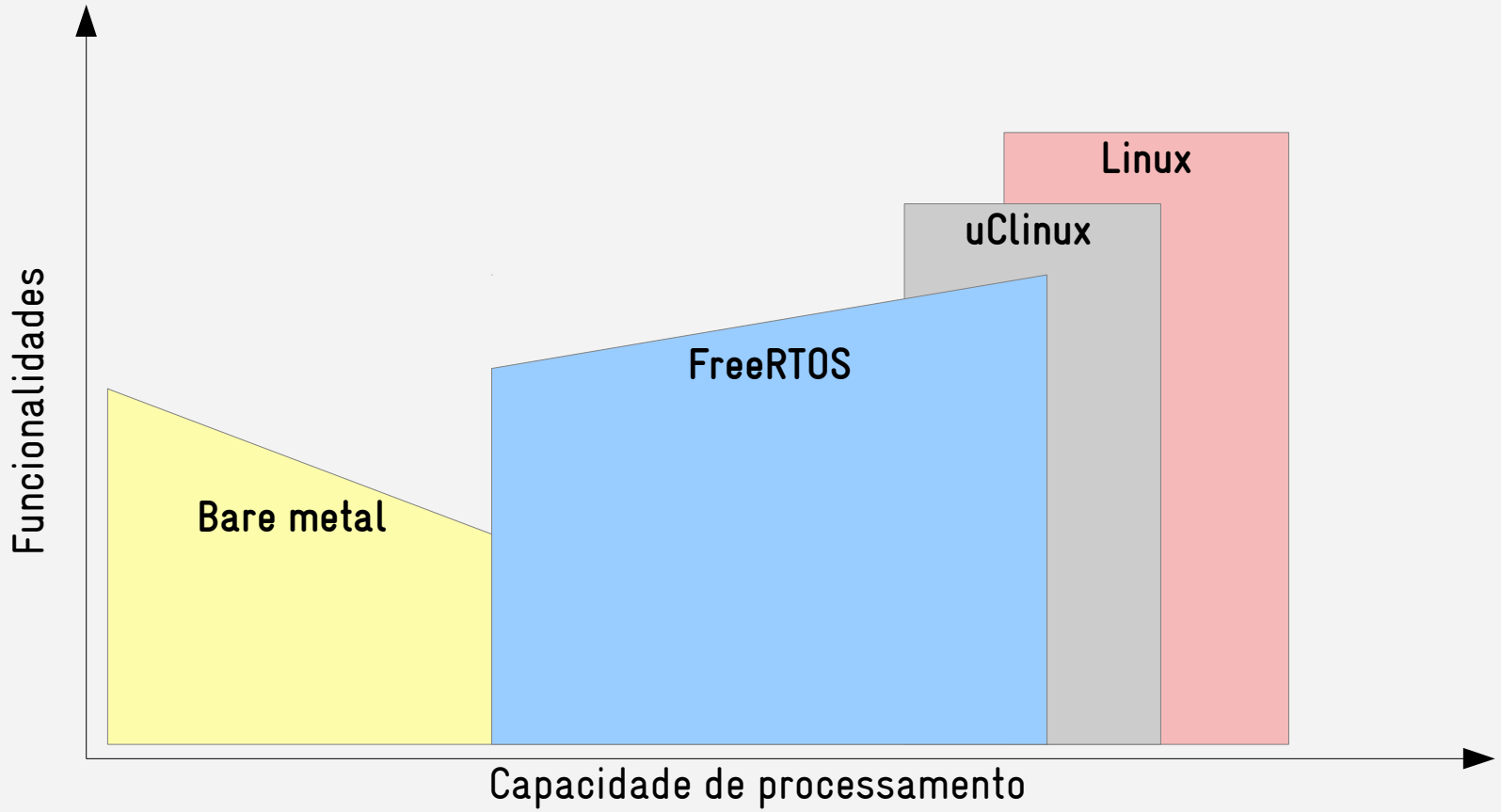
# O FreeRTOS

- Criado por volta do ano 2000 por Richard Barry, e hoje mantindo pela empresa Real Time Engineers Ltd.
  http://www.freertos.org/
- RTOS de código aberto mais utilizado no mundo.
- É simples, pequeno e extremamente portável.
- Documentação da API disponível no site do projeto.
  http://www.freertos.org/a00106.html

# O FreeRTOS (cont.)

- Código-fonte disponível no SourceForge.
  http://sourceforge.net/projects/freertos/files/
- Histórico de alterações disponível no site do projeto.
  http://www.freertos.org/History.txt
- Usaremos no treinamento o FreeRTOS V9.0.0 (maio/2016).

# POSIÇÃO NO MERCADO



# CARACTERÍSTICAS

- Projetado para ser pequeno, simples e fácil de usar.
- Escrito em C, extremamente portável.
- Suporta mais de 30 arquiteturas diferentes.
- Em uma configuração típica, o kernel do FreeRTOS pode ocupar de 4KB a 9KB de código (ROM/flash) e em torno de 200 bytes de dados (RAM).
  - Porte do RL78 com 13 tarefas, 2 queues and 4 software timers consome menos que 4K de RAM!

# CARACTERÍSTICAS (cont.)

- Kernel pode trabalhar de forma preemptiva ou colaborativa.
- Mutex com suporte à herança de prioridade.
- Capacidades de trace e detecção de stack overflow.
- Sem restrição de quantidade de tarefas que podem ser criadas, ou da quantidade de prioridades que podem ser usadas.
- Suporte a aplicações de baixo consumo (tickless mode).

# CARACTERÍSTICAS (cont.)

- Diversos projetos e aplicações de demonstração para facilitar o aprendizado.
- Código aberto, sem royalty e com fórum gratuito disponível.
  http://sourceforge.net/p/freertos/discussion/
- Ferramentas de desenvolvimento abertas e gratuitas.
- Comunidade grande de usuários.
- Suporte e licença comercial se necessário.

# CÓDIGO-FONTE

```
FreeRTOS
├── Source
│   ├── croutine.c
│   ├── event_groups.c
│   ├── include
│   ├── list.c
│   ├── portable
│   ├── queue.c
│   ├── readme.txt
│   ├── tasks.c
│   └── timers.c
├── Demo
│   ├── Common
│   ├── CORTEX_A2F200_IAR_and_Keil
│   ├── CORTEX_A2F200_SoftConsole
│   ├── CORTEX_A5_SAMA5D3x_Xplained_IAR
├── License
│   └── license.txt
└── readme.txt
```

# CÓDIGO-FONTE (PORTE)

```
portable/
├── CCS
│   ├── ARM_Cortex-R4
│   └── MSP430X
├── GCC
│   ├── ARM7_AT91FR40008
│   ├── ARM7_AT91SAM7S
│   └── ARM_CM0
│       ├── port.c
│       └── portmacro.h
├── IAR
│   ├── 78K0R
│   ├── ARM_CM0
│   └── V850ES
├── MemMang
│   ├── heap_1.c
│   ├── heap_2.c
│   ├── heap_3.c
│   ├── heap_4.c
│   └── heap_5.c
```

# FreeRTOSConfig.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

# PADRÃO DE CODIFICAÇÃO

- MISRA (Motor Industry Software Reliability Association) é uma associação de empresas (principalmente da área automotiva) com o objetivo de promover as melhores práticas para o desenvolvimento de produtos eletrônicos automotivos.

- Todo o código comum do FreeRTOS é baseado no padrão MISRA C, com algumas exceções, por exemplo:
  - Algumas funções da API tem mais de um ponto de saída.
  - Uso de aritmética de ponteiros.

# CONVENÇÕES

- No FreeRTOS usa-se um prefixo no nome das variáveis para indicar seu tipo. Exemplos:
  - Variáveis do tipo char começam com "c".
  - Variáveis do tipo short começam com "s".
  - Outros tipos começam com "x", como por exemplo estruturas.
  - Variáveis sem sinal começam com "u". Ex: "us".
  - Ponteiros começam com "p". Ex: "pul".
- Funções privadas em um arquivo começam com "prv".
- Funções da API são prefixadas com o tipo de retorno da função (mesma convenção usada no nome de variáveis).

# LICENÇA

- O FreeRTOS é licenciado pela Real Time Engineers Ltd., sob uma versão modificada da GPL.
- Tem código fonte aberto, não precisa pagar royalties e pode ser usado livremente em aplicações comerciais.
- Não precisa liberar os fontes da sua aplicação, desde que você não crie nenhuma funcionalidade já fornecida pelo FreeRTOS.
- Precisa indicar que usa o FreeRTOS (um link para o site é suficiente).
- Qualquer alteração no kernel precisa ser liberada de forma open-source.

# OPENRTOS

- Versão comercial do FreeRTOS provida pela WITTENSTEIN High Integrity Systems.
  https://www.highintegritysystems.com/openrtos/
- Praticamente o mesmo código-fonte, mas sem nenhuma referência à GPL, permitindo alterar o kernel sem precisar liberar os fontes.
- Integração com bibliotecas e pilhas de protocolo.
- Acesso a suporte técnico e desenvolvimento.

# SAFERTOS

- Versão comercial e modificada do FreeRTOS, projetada para aplicações críticas.
  https://www.highintegritysystems.com/safertos/
- Todas as vantagens da versão comercial (suporte, garantia, etc).
- Certificado para aplicações industriais e automotivas (IEC 61508-3 SIL e ISO 26262 ASIL D).

Embedded **Labworks**

# LABORATÓRIO

## Estudando e integrando o FreeRTOS

Embedded Labworks

# FreeRTOS

## Gerenciamento de tarefas

# GERENCIAMENTO DE TAREFAS

- Cada tarefa se comporta como um programa isolado:
  - Tem um ponto de entrada.
  - É implementada normalmente com um loop infinito.
  - Normalmente não retorna. Se uma tarefa finalizar, é responsabilidade do desenvolvedor removê-la da lista de tarefas do kernel.
- Protótipo de uma tarefa:

```
void vTaskCode(void *pvParameters);
```

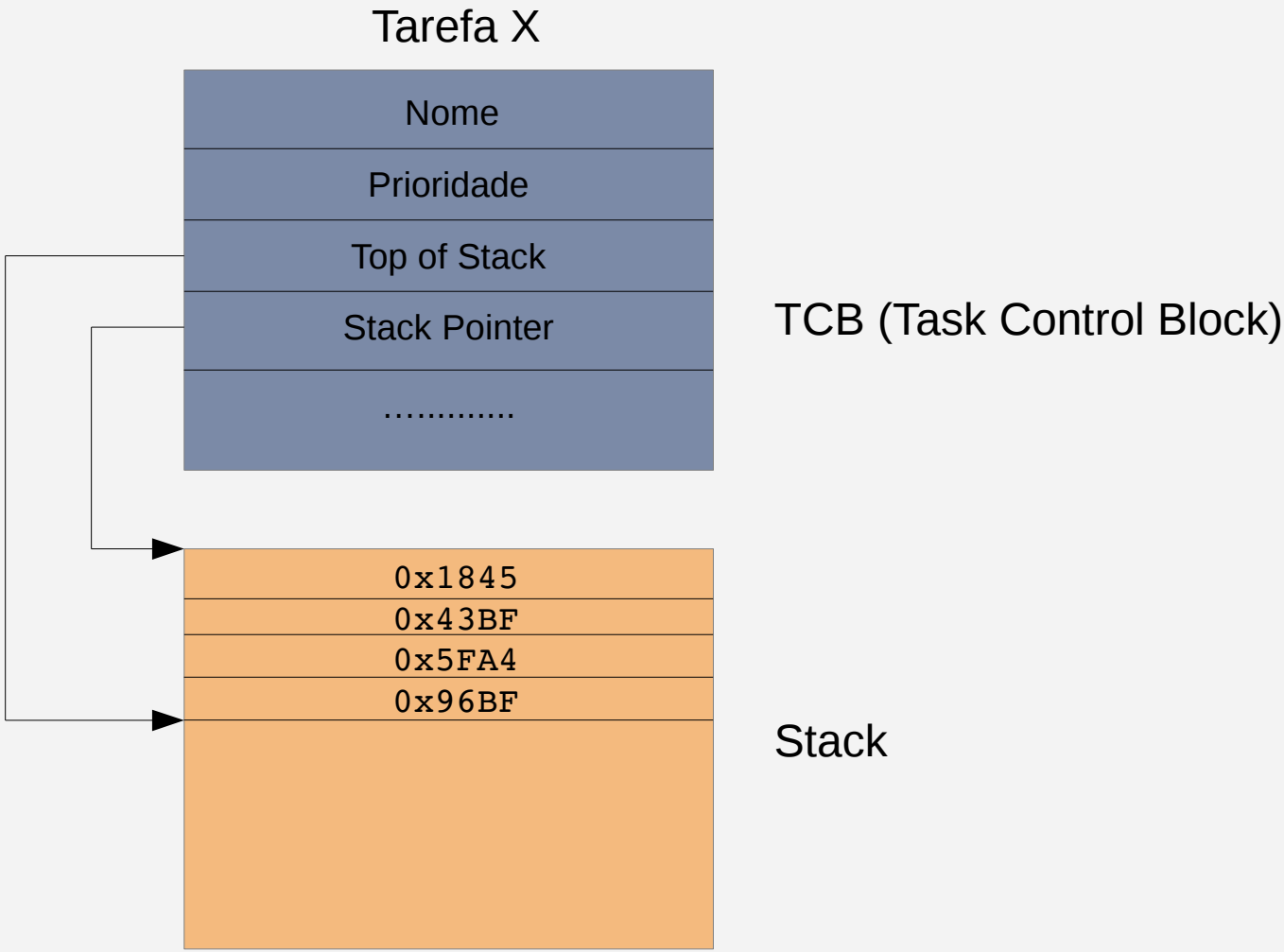
# ESQUELETO DE UMA TAREFA

```
void vTaskCode(void *pvParameters)
{
    for(;;)
    {
        /* task code */
    }
}
```

# CRIANDO UMA TAREFA

```
#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 *pxCreatedTask
);
```

# TCB E STACK


Tarefa X
Nome
Prioridade
Top of Stack
Stack Pointer
.....
0x1845
0x43BF
0x5FA4
0x96BF
TCB (Task Control Block)
Stack

# TRABALHANDO COM TAREFAS

- Uma aplicação pode conter diversas tarefas.
- Em um microcontrolador com apenas um núcleo de CPU, apenas uma tarefa estará em execução em um determinado momento.
- Portanto, isso significa que uma tarefa pode estar em dois estados: "**Running**" (executando) ou "**Not running**" (não executando, parada). Veremos mais adiante que o estado "**Not running**" possui alguns sub-estados.
- Esta troca de estados é realizada pelo scheduler ou **escalonador de tarefas**.

# INICIANDO O ESCALONADOR

```
#include "FreeRTOS.h"
#include "task.h"

int main(void)
{

    [...]

    /* create task */
    xTaskCreate(vTaskCode, (signed char *)"TaskName",
                configMINIMAL_STACK_SIZE, (void *)NULL,
                1, NULL);

    /* start the scheduler */
    vTaskStartScheduler();

    /* should never reach here! */
    for(;;);
}
```

# REMOVENDO UMA TAREFA

- Antes de retornar, uma tarefa deve remover ela mesma da lista de tarefas do kernel com a função `vTaskDelete()`.
- Para usar esta função, habilite a opção `INCLUDE_vTaskDelete` no arquivo `FreeRTOSConfig.h`.

# REMOVENDO UMA TAREFA (cont.)

```
#include "task.h"

/* remove a task from the RTOS kernel management */
void vTaskDelete(TaskHandle_t xTask);
```

# REMOVENDO UMA TAREFA (cont.)

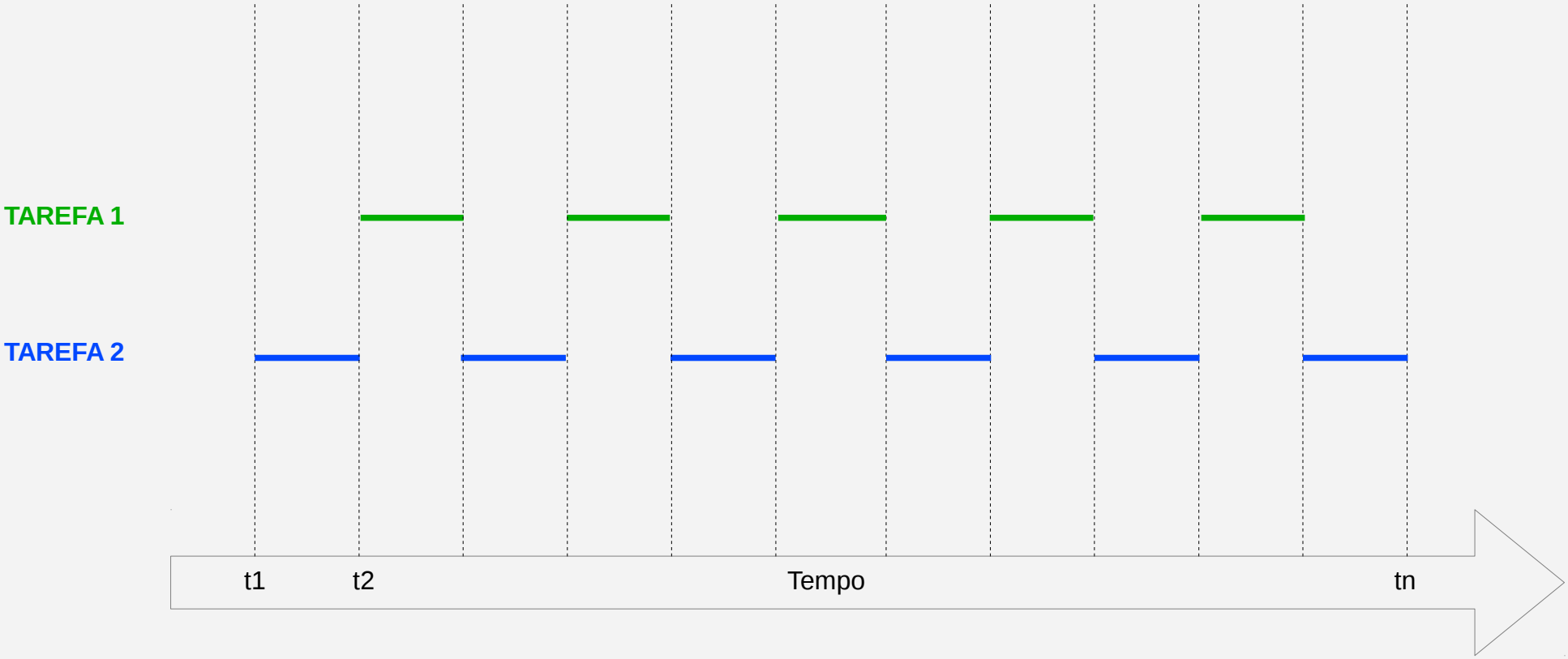
```
#include "task.h"

void vTaskCode(void *pvParameters)
{
    for(;;)
    {
        /* task code */
    }

    vTaskDelete(NULL);
}
```

Embedded **Labworks**

# LABORATÓRIO

## Aplicação baseada em tarefas

# TAREFAS EM EXECUÇÃO



# PRIORIDADES

- A prioridade de uma tarefa é definida no parâmetro `uxPriority` da função `xTaskCreate()`.
- As prioridades vão de 0 (menor prioridade) até (`configMAX_PRIORITIES – 1`), definida no arquivo `FreeRTOSConfig.h`.
- Quanto maior o valor máximo, maior o consumo de RAM!
- Em tempo de execução, também é possível alterar a prioridade de uma tarefa com a função `vTaskPrioritySet()`.

# PRIORIDADES E O ESCALONADOR

- As tarefas terão diferentes prioridades, dependendo das suas características de tempo real.
- Ao selecionar uma tarefa para execução, o escalonador irá executar sempre a tarefa de maior prioridade.
- O escalonador pode funcionar de forma **colaborativa** ou **preemptiva**, dependendo do valor da opção `configUSE_PREEMPTION` no arquivo `FreeRTOSConfig.h`.

# MODO COLABORATIVO

- No modo colaborativo, as tarefas não são interrompidas pelo kernel durante sua execução.
- Neste caso, uma tarefa precisa liberar a CPU de forma voluntária para o escalonador selecionar uma outra tarefa para execução.
- Para isso, a tarefa pode:
  - Bloquear esperando um evento.
  - Liberar a CPU através da macro `taskYIELD()`.
- Mesmo no modo colaborativo, as tarefas ainda podem ser interrompidas para o tratamento de uma interrupção.

# USANDO taskYIELD

```
#include "task.h"

void vTaskCode(void *pvParameters)
{
    for (;;) {

        process_something();

        taskYIELD();
    }
}
```

# MODO PREEMPTIVO

- No modo preemptivo, sempre que uma tarefa de maior prioridade ficar pronta para execução, o kernel irá interromper a tarefa de menor prioridade para executar a tarefa de maior prioridade.

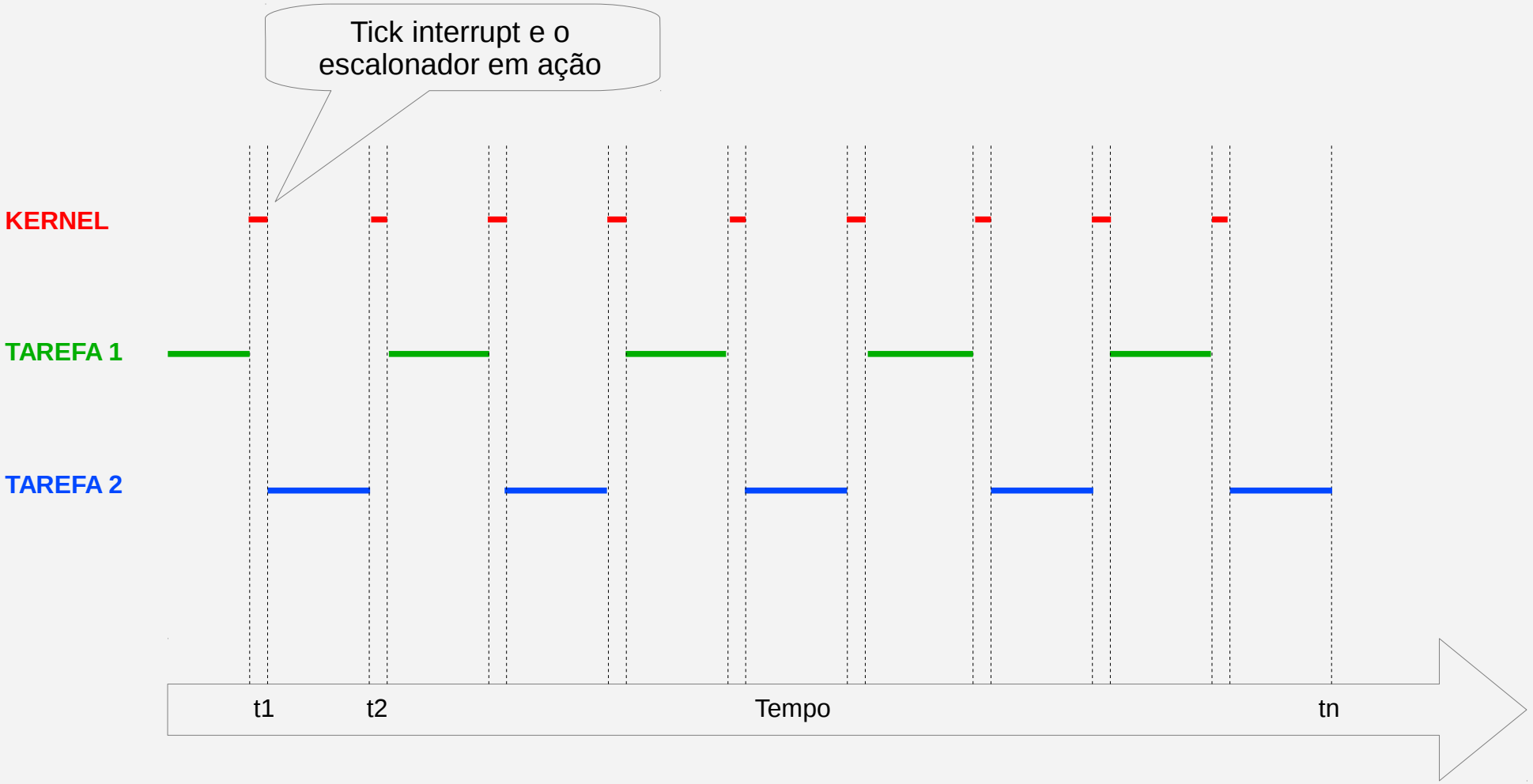
- Para tarefas de mesma prioridade, o kernel irá definir uma fatia de tempo (**time slice**) da CPU para cada tarefa, e irá chavear entre elas usando o algoritmo de **round robin**.

- Se necessário, a preempção por time slice pode ser desabilitada através da opção `configUSE_TIME_SLICING` no arquivo `FreeRTOSConfig.h`.

# TICK INTERRUPT

- Para interromper a tarefa em execução e trocar de contexto para uma nova tarefa, o kernel usa uma interrupção do sistema.
- Esta interrupção é chamada de **tick interrupt** (system tick ou clock tick).
- É uma interrupção periódica cuja frequência é definida em `configTICK_RATE_HZ` no arquivo `FreeRTOSConfig.h`.
- Por exemplo, se estiver definida como 100 (Hz), significa que a fatia de tempo será de 10ms.

# TICK INTERRUPT E PREEMPÇÃO



# TICK INTERRUPT HOOK

- É possível configurar uma função de callback para ser chamada em cada tick interrupt:
  - Esta função pode ser usada para executar uma rotina periódica.
  - Como esta rotina roda em contexto de interrupção, mantenha seu processamento o mais breve possível.
- Habilite a opção `configUSE_TICK_HOOK` no arquivo `FreeRTOSConfig.h` e implemente a função:

```
void vApplicationTickHook(void);
```

Embedded **Labworks**

# LABORATÓRIO

## Prioridades e escalonamento

# EVENTOS

- Nas atividades que desenvolvemos até agora, as tarefas não bloqueiam ou esperam por algo, elas simplesmente monopolizam a CPU.

- Em um RTOS, se criarmos tarefas que monopolizam a CPU, elas deverão ter sempre a menor prioridade, porque se elas tiverem uma prioridade mais alta, tarefas de menor prioridade nunca serão executadas.

- Por isso, sempre que possível, as tarefas devem ser orientadas à eventos, ou seja, devem aguardar um evento para realizar o processamento.

# TAREFA ORIENTADA À EVENTOS

```
void vTaskCode(void *pvParameters)
{
    for(;;)
    {
        wait_event();
        process_event();
    }
}
```

# ESTADO BLOCKED

- Uma tarefa esperando um evento está no estado Blocked ou bloqueada.
- Um tarefa pode estar bloqueada aguardando dois tipos de eventos:
  - **Eventos temporais**: evento gerado pelo próprio kernel. Ex: rotinas de delay.
  - **Eventos de sincronização**: evento originado por uma outra tarefa ou interrupção. Ex: mecanismos de comunicação como queues e semáforos.

# ROTINAS DE DELAY

```
#include "task.h"

/* delay a task for a given number of ticks */
void vTaskDelay(
    const TickType_t xTicksToDelay
);

/* delay a task until a specified time */
void vTaskDelayUntil(
    TickType_t *pxPreviousWakeTime,
    const TickType_t xTimeIncrement
);
```

# EXEMPLO ROTINA DELAY

```
/* blink led every 500ms */
void taskBlinkLed(void *pvParameters)
{
    for (;;) {
        vTaskDelay(500/portTICK_PERIOD_MS);
        blink_led();
    }
}
```

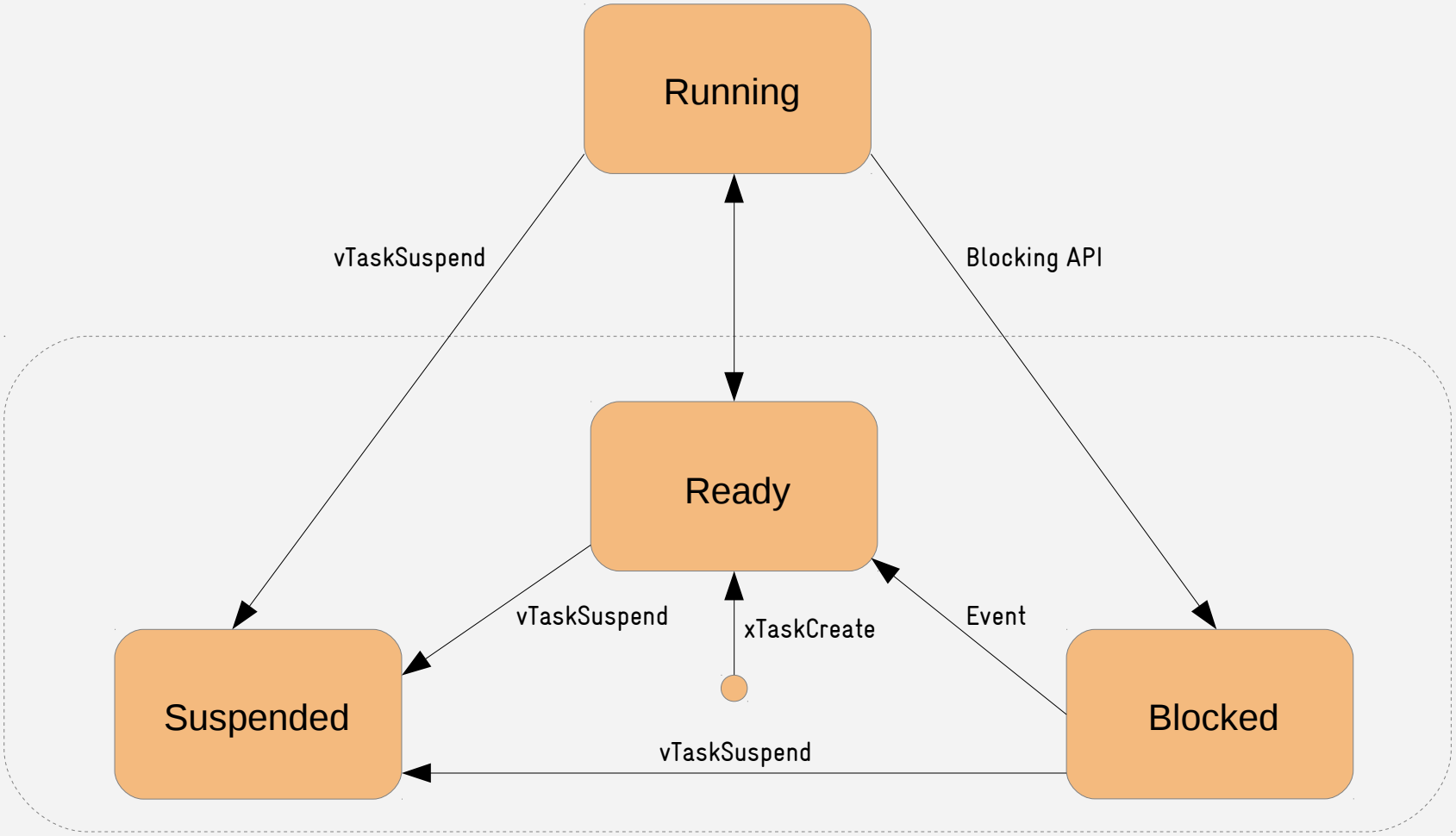
# ESTADO SUSPENDED

- A função `vTaskSuspend()` pode ser usada para colocar uma tarefa no estado Suspended.
- Tarefas no estado Suspended não são escalonadas (executadas) pelo kernel.
- A função `vTaskResume()` pode ser usada para tirar uma tarefa do estado Suspended.

# ESTADO READY

- Tarefas que não estão nos estados Blocked ou Suspended estão no estado Ready.
- Estas tarefas estão aguardando na fila, prontas para serem selecionadas e executadas pelo escalonador.

# DIAGRAMA DE ESTADOS DAS TAREFAS



Embedded **Labworks**

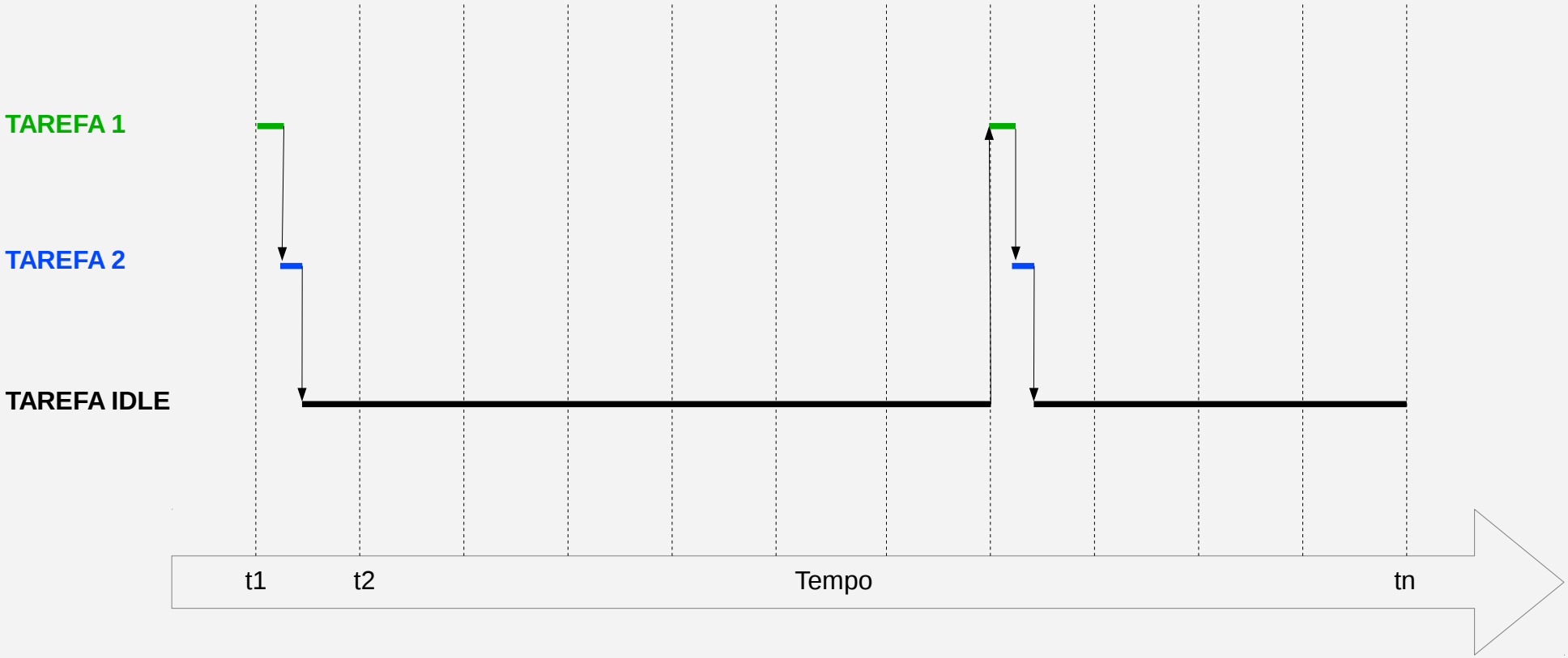
# LABORATÓRIO

## Rotinas de delay

# TAREFA IDLE

- Quando trabalhamos com eventos, as tarefas estão a maioria do tempo no estado Blocked.
- Quando as tarefas estão neste estado, não podem ser escolhidas e executadas pelo escalonador.
- Mas a CPU precisa estar sempre executando alguma coisa. É por isso que existe a tarefa **Idle**!
- Esta tarefa é criada automaticamente quando iniciamos o escalonador ao chamar a função `vTaskStartScheduler()`.
- Esta tarefa tem prioridade 0 (menor prioridade possível).

# TAREFAS EM EXECUÇÃO



# IMPLEMENTAÇÃO DA TAREFA IDLE

- À princípio, a tarefa Idle é uma tarefa que não executa nada!

```
void idleTask(void *pvParameters)
{
    for( ;; )
    {
        /* do nothing! */
    }
}
```

# IMPLEMENTAÇÃO DA TAREFA IDLE

- Porém, no FreeRTOS ela tem algumas responsabilidades!

```
void prvIdleTask(void *pvParameters)
{
    for(;;)
    {
        check_deleted_tasks();

        check_if_another_task_is_ready();

        execute_idle_task_hook();

        check_if_should_sleep();
    }
}
```

# IDLE TASK HOOK

- É possível configurar uma função de callback para ser chamada quando a tarefa Idle for executada para:
  - Executar um processamento contínuo em background.
  - Medir a quantidade de processamento livre disponível.
  - Colocar o processador em modo de baixo consumo.
- Para isso, habilite a opção `configUSE_IDLE_HOOK` no arquivo `FreeRTOSConfig.h` e implemente a função:

```
void vApplicationIdleHook(void);
```

# REGRAS PARA IMPLEMENTAR

- A implementação da Idle Task Hook nunca deve bloquear ou suspender, já que neste caso pode acontecer um cenário onde nenhuma tarefa está pronta para execução.

- É dentro da função Idle que é feita a limpeza das tarefas deletadas pela aplicação. Portanto, se a aplicação usa a função `vTaskDelete()`, então a função Idle Task Hook sempre deve retornar, de forma que a limpeza possa ser realizada dentro da tarefa Idle.

# ECONOMIA DE ENERGIA

- É comum o uso da Idle Task Hook para colocar a CPU em um modo de baixo consumo.
- O problema é que a interrupção do tick precisa continuar ligada para que o kernel possa gerenciar a passagem do tempo e acordar tarefas periódicas quando necessário.
- E se a frequência do tick for muito alta, o tempo e a energia consumida para entrar e sair do modo de baixo consumo a cada interrupção do tick pode invalidar qualquer energia economizada no modo de baixo consumo.
- Para resolver este problema, o FreeRTOS possui uma funcionalidade chamada de **Tickless Idle Mode**.

# TICKLESS IDLE MODE

- Esta funcionalidade é capaz de parar a interrupção do tick em períodos ociosos antes de colocar o sistema em modo de baixo consumo.
- Isso é feito ajustando a frequência da interrupção do tick para a próxima tarefa dependente do tick do sistema (Ex: tarefa periódica usando uma função de delay).
- Na próxima interrupção do tick, o contador do tick é ajustado para o valor correto.
- Tudo isso é feito automaticamente pelo FreeRTOS!

# TICKLESS IDLE MODE (cont.)

- Esta funcionalidade é suportada nativamente nos portes de algumas arquiteturas, incluindo RX100 e ARM Cortex-M3/M4/M7.
- Para usá-la, basta definir a opção `configUSE_TICKLESS_IDLE` com 1 no arquivo `FreeRTOSConfig.h`.
- Para os portes de arquiteturas que não possuem esta funcionalidade implementada nativamente, é possível definir a opção `configUSE_TICKLESS_IDLE` com 2 e implementar a macro abaixo:

```
portSUPPRESS_TICKS_AND_SLEEP(xExpectedIdleTime);
```

Embedded **Labworks**

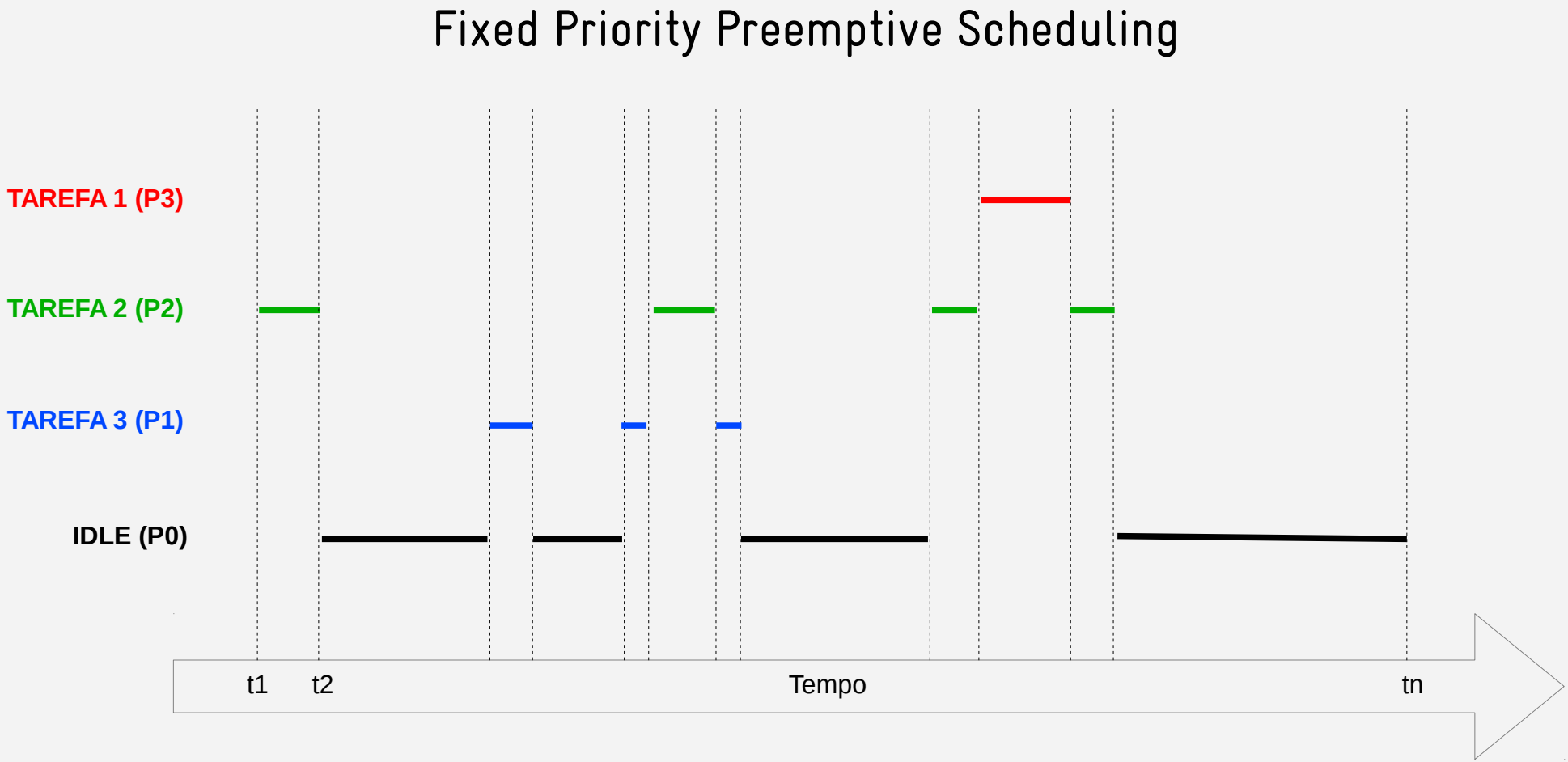
# LABORATÓRIO

## Idle Task Hook

# UM RESUMO DO ESCALONADOR

- Uma aplicação é composta por uma ou mais tarefas.
- Cada tarefa tem uma prioridade (se necessário, as tarefas podem compartilhar a mesma prioridade).
- Cada tarefa pode estar em um determinado estado (Running, Ready, Blocked, Suspended).
- Apenas uma tarefa pode estar no estado Running em determinado momento.
- O escalonador sempre seleciona a tarefa de **maior prioridade** e no **estado Ready** para ser executada.

# O KERNEL DO FREERTOS



Embedded Labworks

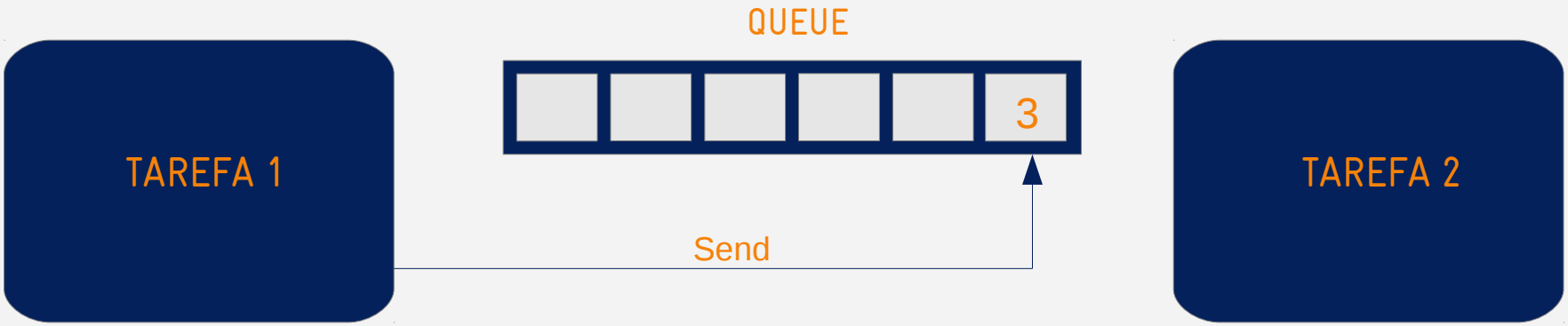
# FreeRTOS

## Gerenciamento de queues

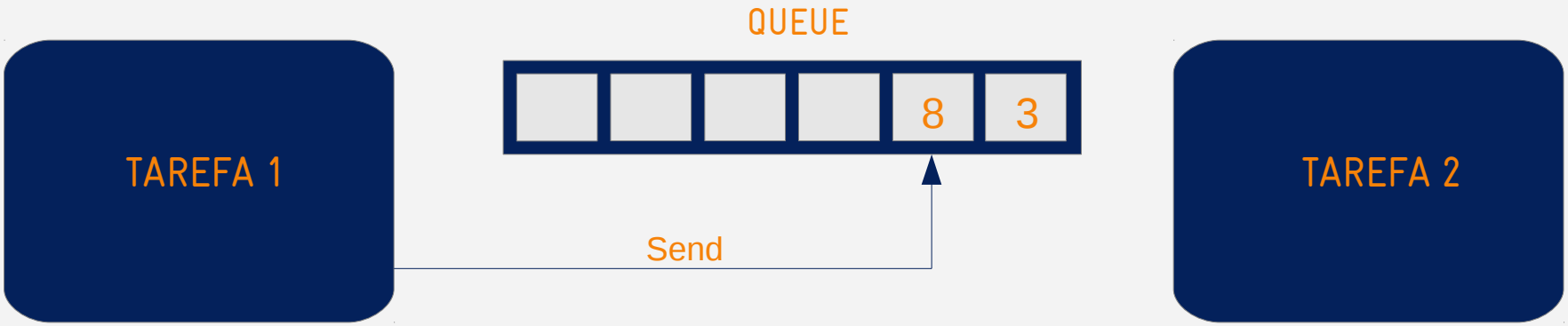
# GERENCIAMENTO DE QUEUE

- Vimos até aqui que uma aplicação usando o FreeRTOS é estruturada através de um conjunto de tarefas independentes.
- É bem provável que estas tarefas precisem se comunicar entre si.
- Queue é um mecanismo de comunicação (troca de mensagens) entre tarefas ou entre uma tarefa e uma interrupção.

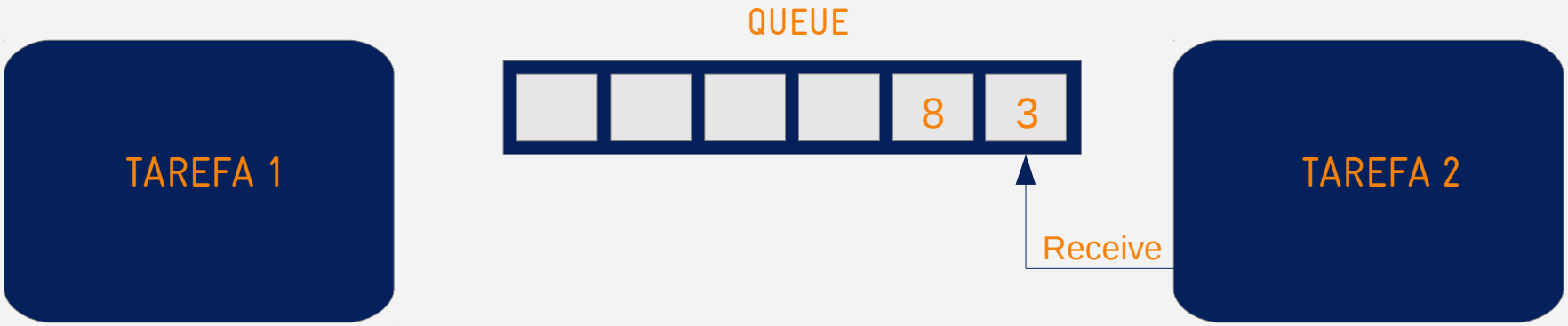
# EXEMPLO



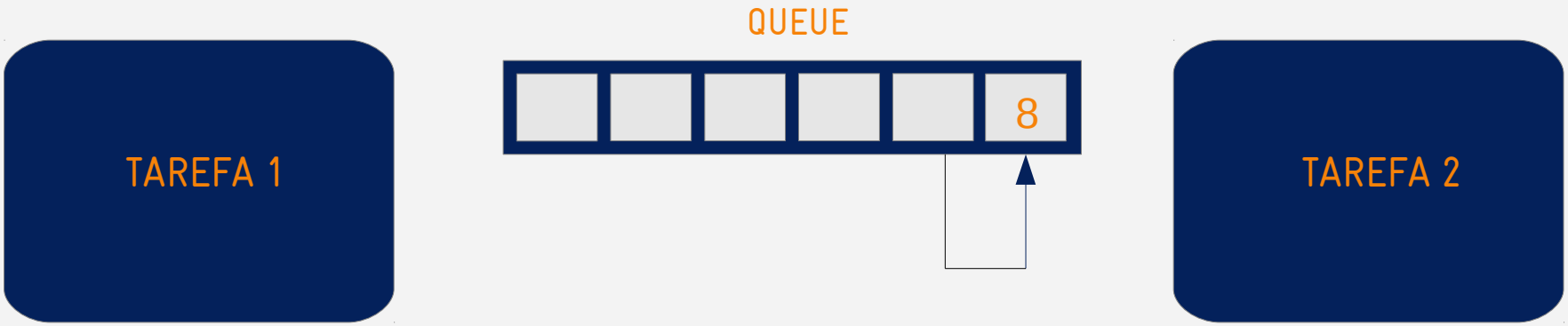
# EXEMPLO (cont.)



# EXEMPLO (cont.)



# EXEMPLO (cont.)



# O QUEUE

- Um queue não pertence à nenhuma tarefa em específico.
- Diversas tarefas e interrupções podem compartilhar o mesmo queue, tanto para ler, quanto para escrever.
- Cada queue armazena um conjunto finito de itens (queue lenght).
- Cada item do queue pode ter um tamanho fixo de bytes (item size).
- Ambos "queue lenght" e "item size" são definidos no momento da criação do queue (o FreeRTOS aloca espaço no heap para armazenar o queue).

# CRIANDO E DELETANDO UM QUEUE

```
#include "queue.h"

/* create a new queue instance */
QueueHandle_t xQueueCreate(
    UBaseType_t uxQueueLength,
    UBaseType_t uxItemSize
);

/* delete a queue */
void vQueueDelete(QueueHandle_t xQueue);

/* reset a queue to its original empty state */
BaseType_t xQueueReset(QueueHandle_t xQueue);
```

# USANDO UM QUEUE

- Normalmente, os queues são usados como FIFO (First In First Out), onde os dados são escritos no fim do queue e lidos no início.
- Mas também é possível escrever no início do queue.
- Escrever no queue significa copiar os dados byte a byte! Por este motivo, se o elemento do queue for muito grande, o ideal é trabalhar com ponteiros.

# LENDO DO QUEUE

- Ao ler do queue, uma tarefa entra no estado **Blocked** aguardando um item do queue.
- Uma tarefa pode definir um timeout de leitura (tempo que ficará no estado **Blocked** esperando um item no queue).
- Uma tarefa esperando um item no queue é automaticamente colocada no estado **Ready** quando:
  - Um item é escrito no queue.
  - O timeout de leitura expira.

# LENDO DO QUEUE (cont.)

- Queues podem ter mais de uma tarefa esperando a leitura de um item.
- Neste caso, se um item é escrito no queue, apenas a tarefa de maior prioridade é passada para o estado **Ready**.
- Se existir mais de uma tarefa com a mesma prioridade aguardando um item no queue, a tarefa que esta esperando a mais tempo será passada para o estado **Ready**.

# LENDO DO QUEUE (cont.)

```
#include "queue.h"

/* receive an item from a queue */
BaseType_t xQueueReceive(
                QueueHandle_t xQueue,
                void *pvBuffer,
                TickType_t xTicksToWait);

/* receive an item from a queue without removing
   the item from the queue */
BaseType_t xQueuePeek(
                QueueHandle_t xQueue,
                void *pvBuffer,
                TickType_t xTicksToWait);
```

# LENDO DO QUEUE (cont.)

```
#include "queue.h"

/* return the number of free spaces in a queue */
UBaseType_t uxQueueSpacesAvailable(QueueHandle_t xQueue);

/* return the number of messages stored in a queue */
UBaseType_t uxQueueMessagesWaiting(QueueHandle_t xQueue);
```

# ESCREVENDO NO QUEUE

- Ao escrever no queue, uma tarefa pode entrar no estado **Blocked** se o queue estiver cheio, aguardando um espaço no queue para salvar o novo item.
- Uma tarefa pode definir um timeout de escrita (tempo que ficará no estado **Blocked** esperando um espaço no queue para salvar o novo item).
- Uma tarefa escrevendo um item no queue é automaticamente colocada no estado **Ready** quando:
  - O elemento é escrito no queue com sucesso.
  - O timeout de escrita expira.

# ESCREVENDO NO QUEUE (cont.)

- É possível ter mais de uma tarefa escrevendo itens no queue.
- Se o queue esta cheio, todas as tarefas que escrevem no queue são colocadas no estado **Blocked**, esperando um espaço no queue.
- Quando um espaço ficar livre no queue, apenas a tarefa de maior prioridade é colocada no estado **Ready**, de forma que ela possa escrever no queue.
- Se existir mais de uma tarefa com a mesma prioridade aguardando um espaço livre para escrever no queue, a tarefa que esta esperando a mais tempo será passada para o estado **Ready**.

# ESCREVENDO NO QUEUE (cont.)

```
#include "queue.h"

/* post an item to the front of a queue */
BaseType_t xQueueSendToFront(
              QueueHandle_t xQueue,
              const void *pvItemToQueue,
              TickType_t xTicksToWait);

/* post an item to the back of a queue */
BaseType_t xQueueSendToBack(
              QueueHandle_t xQueue,
              const void *pvItemToQueue,
              TickType_t xTicksToWait);
```

# ESCREVENDO NO QUEUE (cont.)

```
#include "queue.h"

/* post an item on a queue - same as xQueueSendToBack() */
BaseType_t xQueueSend(
              QueueHandle_t xQueue,
              const void *pvItemToQueue,
              TickType_t xTicksToWait);

/* A version of xQueueSendToBack() that will write to the
   queue even if the queue is full, overwriting data that
   is already held in the queue. */
BaseType_t xQueueOverwrite(
              QueueHandle_t xQueue,
              const void *pvItemToQueue);
```

# EM INTERRUPÇÕES

- Nunca use estas funções em rotinas de tratamento de interrupção!
- Em rotinas de tratamento de interrupção, use sempre as funções que terminam com `FromISR()`:
  - `xQueueSendToFrontFromISR()`.
  - `xQueueSendToBackFromISR()`.
  - `xQueueReceiveFromISR()`.
- Falaremos mais sobre este assunto no decorrer do treinamento.

Embedded **Labworks**

# LABORATÓRIO

## Comunicação entre tarefas com queues

Embedded **Labworks**

# FreeRTOS

## Interrupção e mecanismos de sincronização

# INTERRUPÇÃO

- Sistemas embarcados precisam tomar ações baseados em eventos externos. Ex: um pacote chegando em uma interface Ethernet precisa ser recebido e tratado imediatamente.

- Normalmente, os eventos são tratados através de interrupções, dentro de uma rotina de tratamento de interrupção (ISR).

- Mas como uma interrupção pode interromper uma tarefa importante ou ser executada com outras interrupções desabilitadas, devemos manter o seu processamento o mais breve possível para não impactar o tempo de resposta da aplicação.

# DESENVOLVENDO UMA ISR

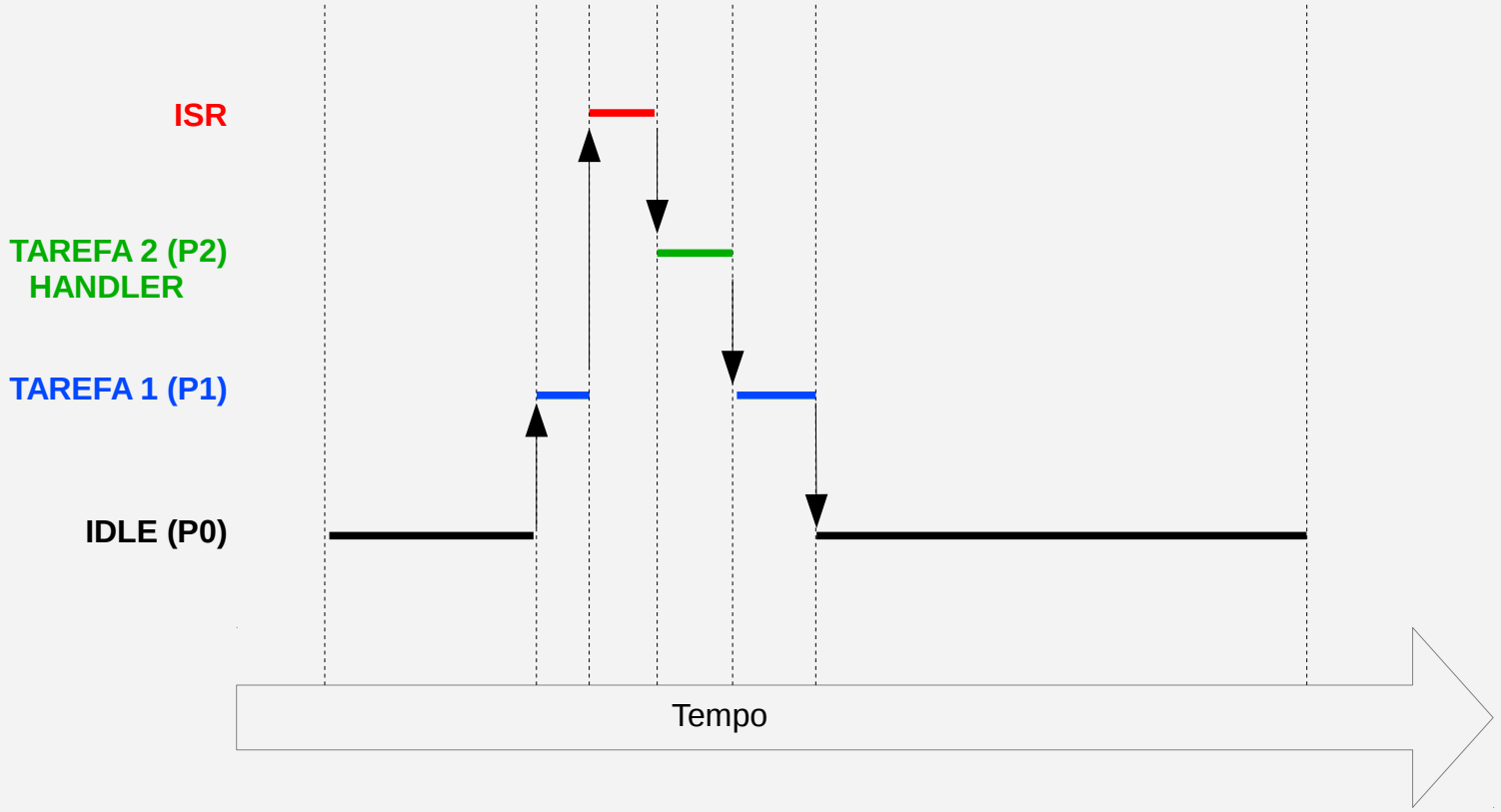
- Resolvemos este problema dividindo o processamento da interrupção entre a ISR e uma tarefa do RTOS.

- Portanto, em uma ISR devemos:
  - Reconhecer a interrupção (ACK).
  - Receber os dados do evento.
  - Deferir o trabalho para uma tarefa (handler) da aplicação.
  - Forçar a troca de contexto (se necessário).

# ESQUELETO DE UMA ISR

```
void isr_handler(void)
{
    ack_int();

    recebe_dados();

    defere_trabalho();

    troca_contexto();
}
```

# TRATANDO UMA ISR



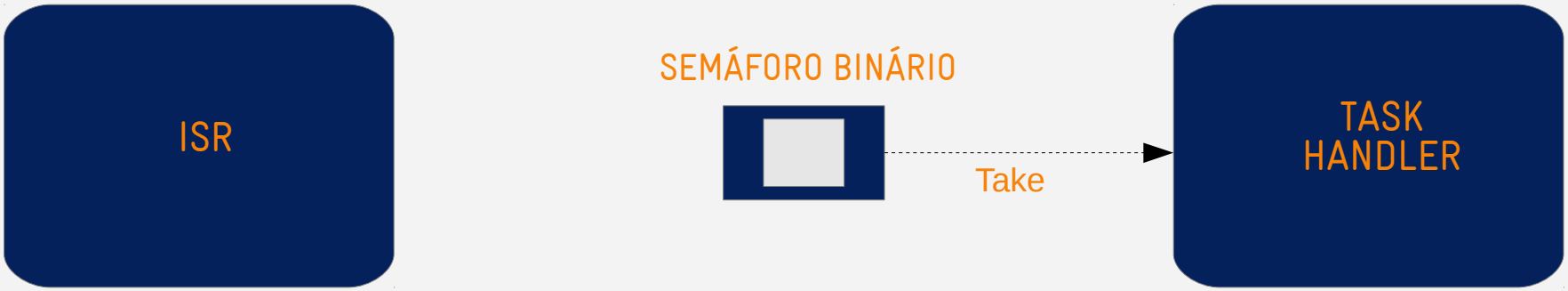
# MECANISMOS DE SINCRONIZAÇÃO

- Uma interrupção é capaz de deferir trabalho para uma tarefa através de mecanismos de sincronização.
- Principais mecanismos de sincronização do FreeRTOS:
  - Semáforos (binários e contadores).
  - Task Notifications.
  - Queues.
  - Event Groups.
  - Queue Sets.
- Estes mecanismos de sincronização podem ser usados tanto para comunicação entre tarefas quanto para comunicação entre interrupções e tarefas.

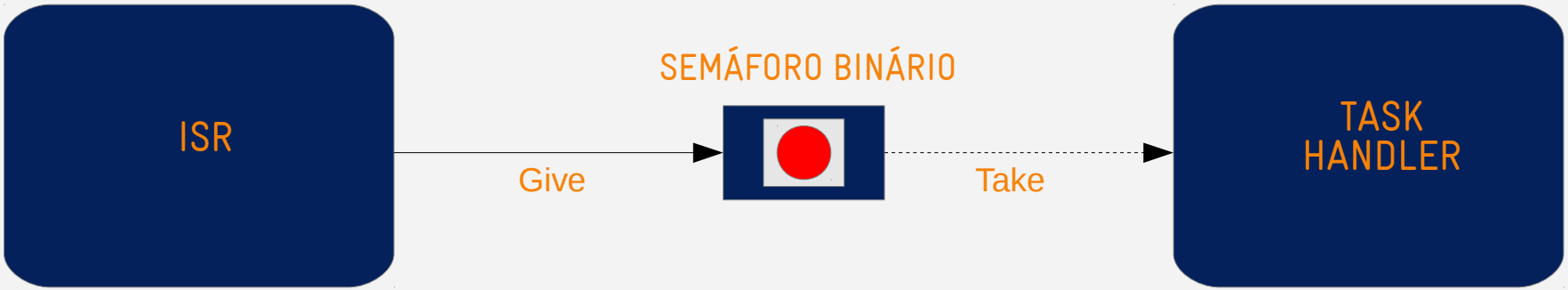
# SEMÁFOROS BINÁRIOS

- Um Semáforo Binário (**Binary Semaphore**) é um mecanismo de sincronização disponibilizado pelo FreeRTOS.
- Ele pode ser usado para acordar (desbloquear) uma tarefa quando determinada interrupção acontecer, sincronizando a interrupção com a tarefa.
- Desta forma, apenas o essencial é executado na interrupção, o restante do trabalho é deferido para a tarefa correspondente ao tratamento da interrupção.

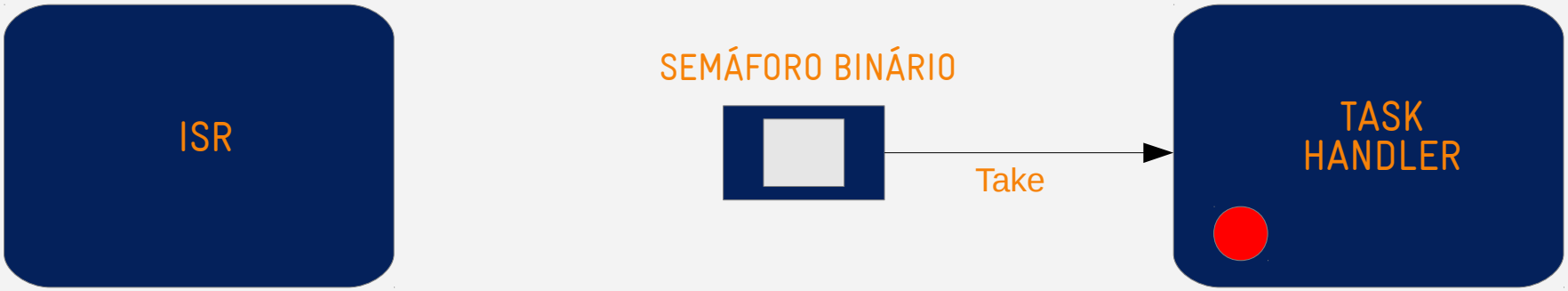
# EXEMPLO



# EXEMPLO (cont.)



# EXEMPLO (cont.)



# CRIANDO UM SEMÁFORO BINÁRIO

```
#include "semphr.h"

/* create a binary semaphore */
SemaphoreHandle_t xSemaphoreCreateBinary(void);

/* delete a semaphore */
void vSemaphoreDelete(SemaphoreHandle_t xSemaphore);
```

# PEGANDO UM SEMÁFORO BINÁRIO

```
#include "semphr.h"

/* obtain a semaphore */
BaseType_t xSemaphoreTake(
            SemaphoreHandle_t xSemaphore,
            TickType_t xTicksToWait);

/* obtain a semaphore from an ISR */
BaseType_t xSemaphoreTakeFromISR(
            SemaphoreHandle_t xSemaphore,
            signed BaseType_t *pxHigherPriorityTaskWoken);
```

# LIBERANDO UM SEMÁFORO BINÁRIO

```
#include "semphr.h"

/* release a semaphore */
BaseType_t xSemaphoreGive(
               SemaphoreHandle_t xSemaphore);

/* release a semaphore from an ISR */
BaseType_t xSemaphoreGiveFromISR(
               SemaphoreHandle_t xSemaphore,
               signed BaseType_t *pxHigherPriorityTaskWoken);
```

# TROCA DE CONTEXTO NA ISR

- Para forçar a troca de contexto e o chaveamento para uma tarefa de maior prioridade, após o processamento da ISR devemos chamar uma função disponibilizada pelo porte do FreeRTOS.

```
void portYIELD_FROM_ISR(BaseType_t flag);
```

- O nome desta função (que normalmente é uma macro) pode variar dependendo do porte que estiver utilizando.

- No parâmetro `flag` devemos utilizar o valor retornado na variável `pxHigherPriorityTaskWoken` das funções que terminam com `FromISR`.

# EXEMPLO ISR

```
void dummyISR(void)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* ACK interrupt, receive data, etc */
    handle_dummy_int();

    /* give semaphore to unblock the task */
    xSemaphoreGiveFromISR(dummy_semaphore,
                          &xHigherPriorityTaskWoken);

    /* switch to task if it's the highest priority task
       ready to run */
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

# SEMÁFOROS CONTADORES

- Semáforos binários são úteis quando a frequência de interrupções é baixa. Mas quando a frequência de interrupções é alta, existe a possibilidade de perdermos interrupções.

- O problema acontece quando mais de uma interrupção acontece no momento em que a tarefa ainda esta tratando o trabalho deferido pela interrupção anterior.

- Para estes casos, podemos usar os semáforos contadores (counting semaphores) no lugar dos semáforos binários.

# EXEMPLO

# EXEMPLO (cont.)

# EXEMPLO (cont.)

# EXEMPLO (cont.)

# EXEMPLO (cont.)

# CRIANDO UM SEMÁFORO CONTADOR

```
#include "semphr.h"

/* create a counting semaphore */
SemaphoreHandle_t xSemaphoreCreateCounting(
       UBaseType_t uxMaxCount,
       UBaseType_t uxInitialCount);
```

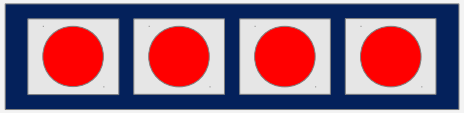
- As funções para pegar e liberar um semáforo contador são as mesmas usadas em um semáforo binário.

# GERENCIANDO RECURSOS

- Um semáforo contador também pode ser usado para gerenciar o acesso a uma quantidade limitada de recursos do sistema.
- Para obter o controle de um recurso, uma tarefa precisa primeiro obter um semáforo (**take**), decrementando o contador de semáforos. Quando o contador atingir o valor de zero, significa que não existem mais recursos disponíveis.
- Quando uma tarefa terminar o uso do recurso, deve devolver o semáforo (**give**).
- Semáforos contadores que são usados para gerenciar recursos são inicializados com a quantidade de recursos disponíveis.

# EXEMPLO

SEMÁFORO CONTADOR

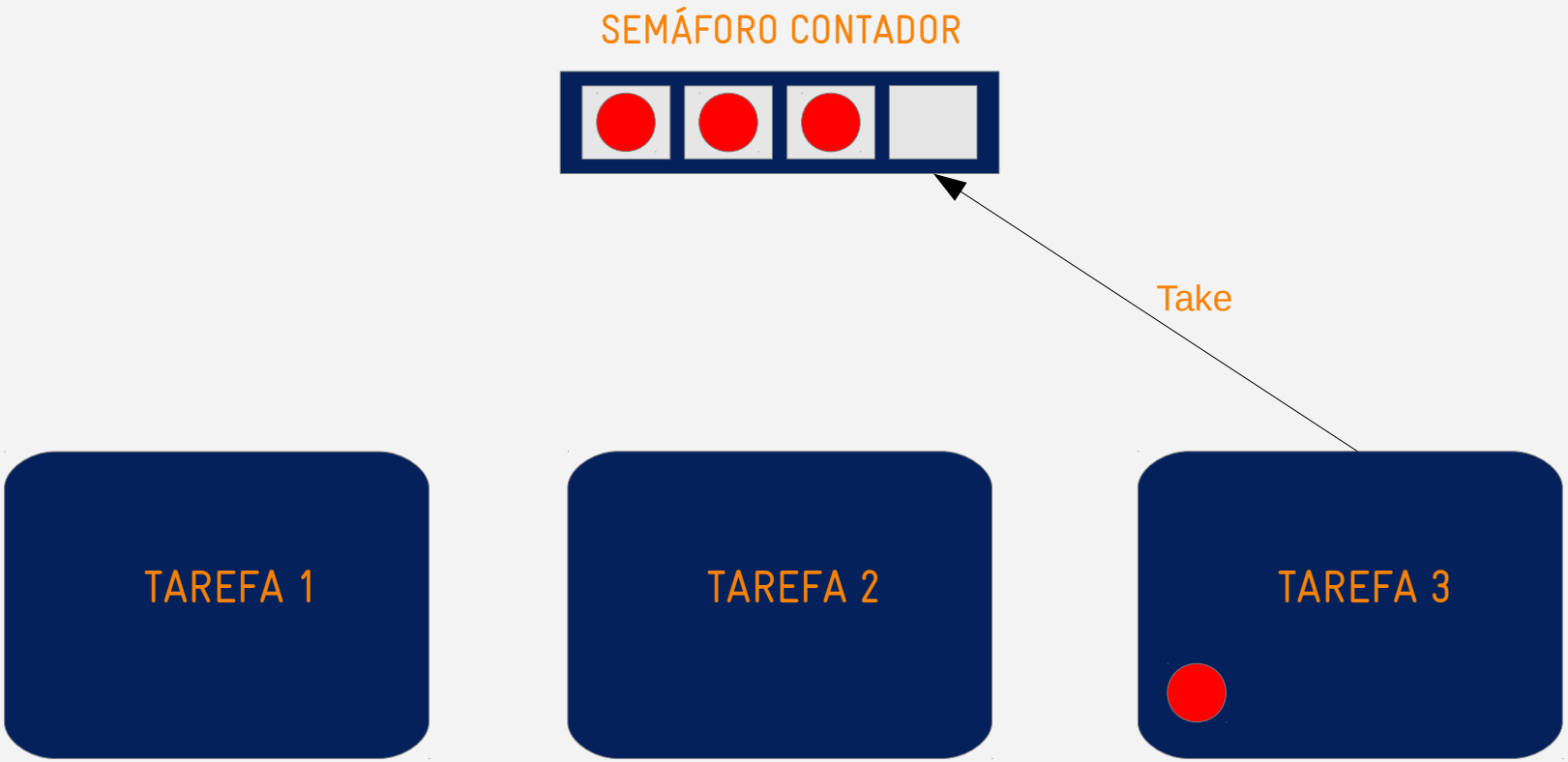


TAREFA 1

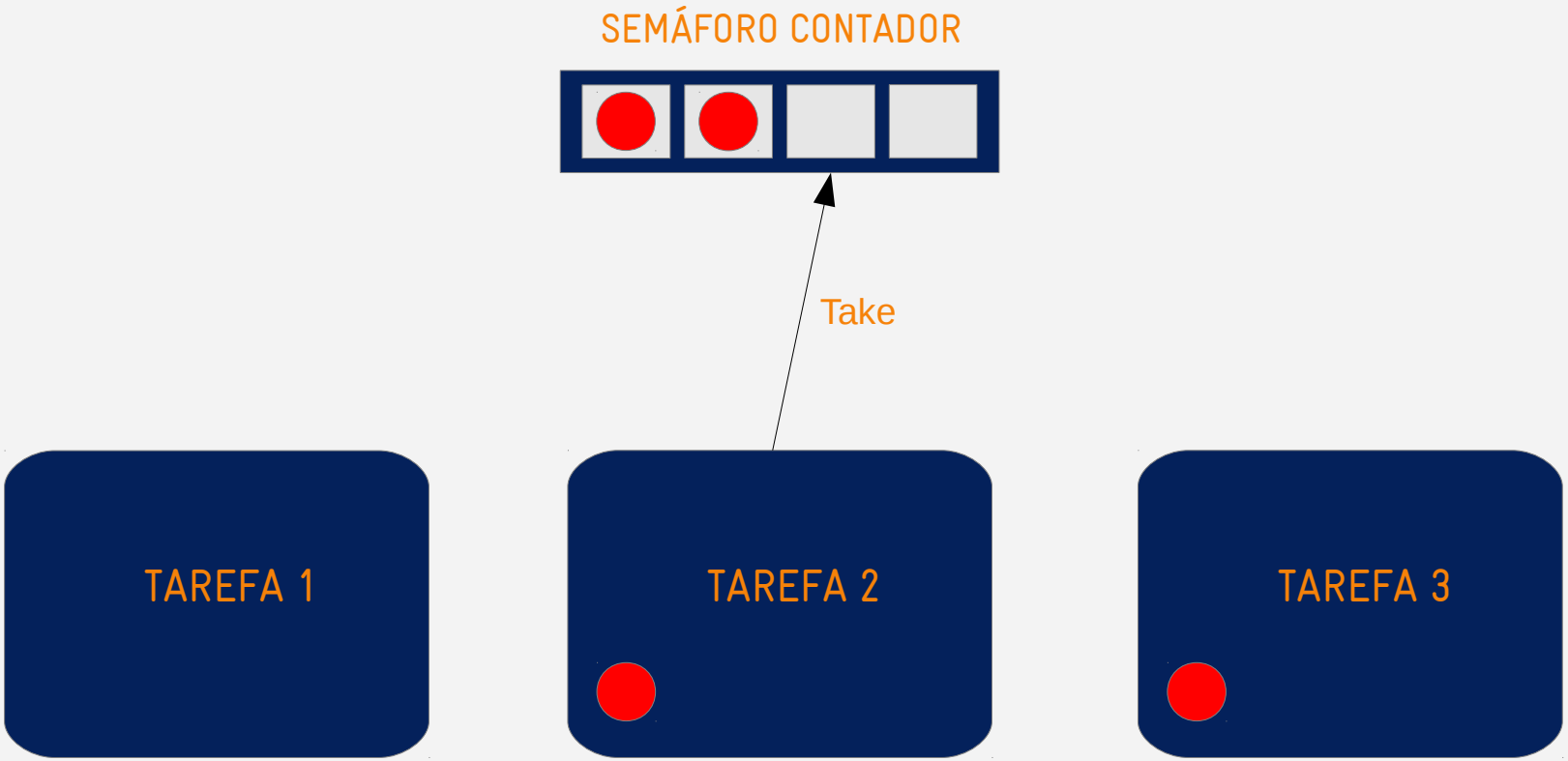
TAREFA 2

TAREFA 3

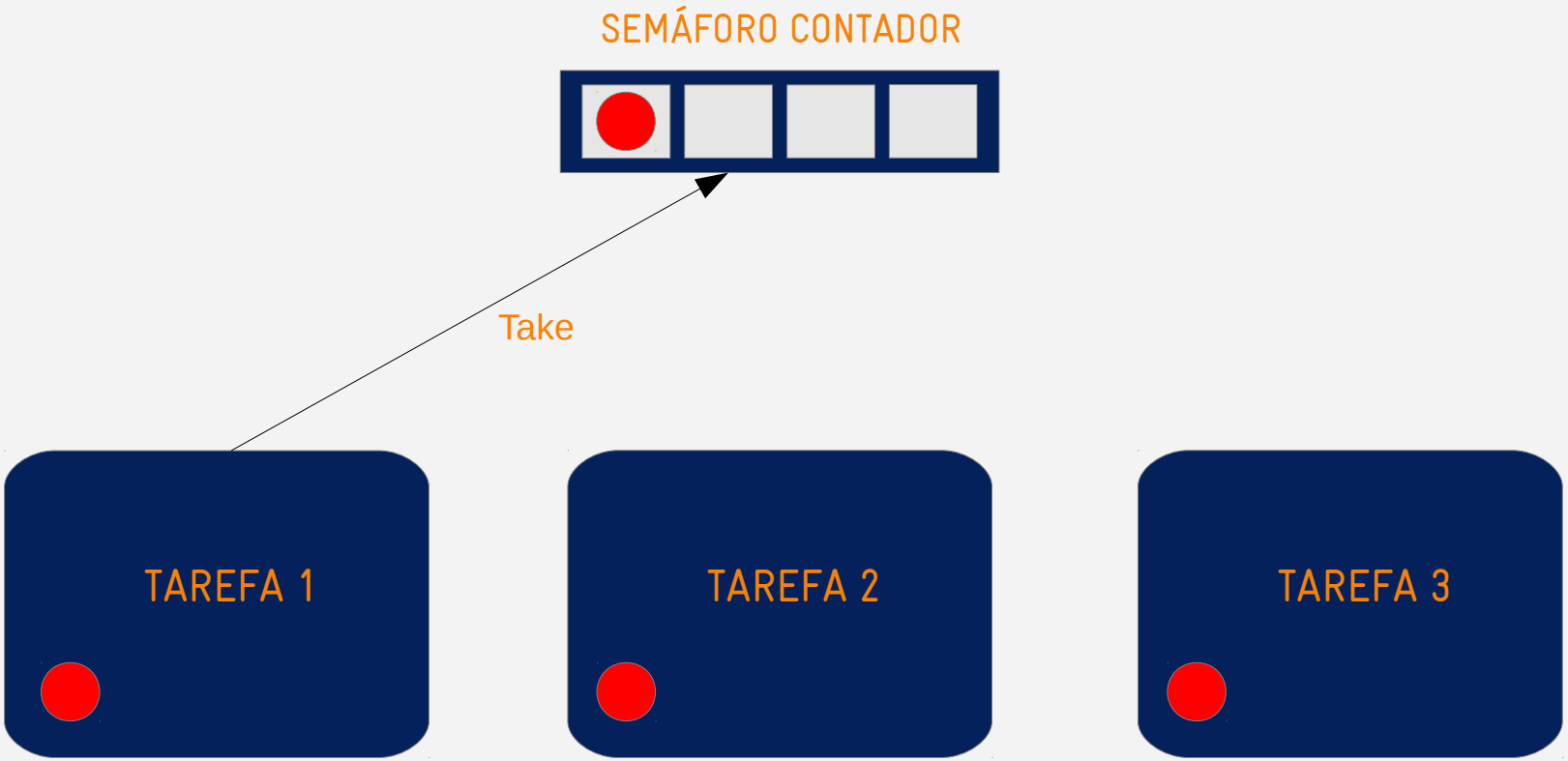
# EXEMPLO (cont.)



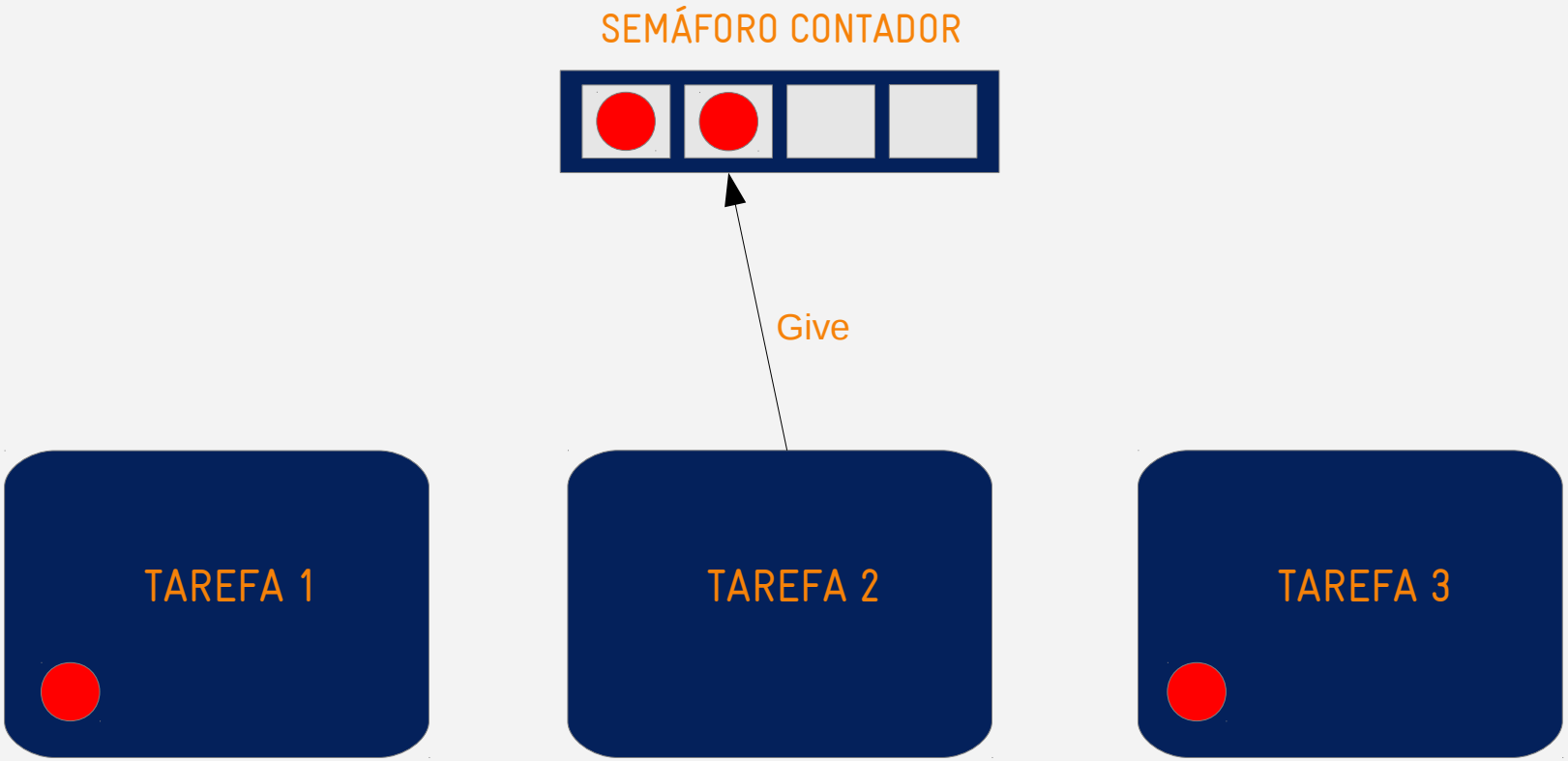
# EXEMPLO (cont.)



# EXEMPLO (cont.)



# EXEMPLO (cont.)



Embedded **Labworks**

# LABORATÓRIO

## Trabalhando com semáforos

# TASK NOTIFICATIONS

- Cada tarefa no FreeRTOS possui uma variável de 32 bits chamada "**notification value**" que pode ser utilizada para sincronização.
- Uma notificação pode ser enviada diretamente para uma tarefa através desta variável.
- Este recurso é chamado de **Task Notification** e pode substituir o uso de semáforos quando apenas uma tarefa é responsável por tratar determinado evento do sistema.

# TASK NOTIFICATIONS API

- Para substituir o semáforo por uma Task Notification, a tarefa deve esperar por notificações através da função `ulTaskNotifyTake()`.
- Da mesma forma, as notificações podem ser enviadas com a função `xTaskNotifyGive()`.
- Lembre-se que, dentro de rotinas de tratamento de interrupção, é necessário utilizar as funções que terminam com `FromISR()`.

# TASK NOTIFICATIONS API (cont.)

```
#include "task.h"

/* wait for a the task's notification value to be set */
uint32_t ulTaskNotifyTake(BaseType_t xClearCountOnExit,
                          TickType_t xTicksToWait);

/* set a task's notification value */
BaseType_t xTaskNotifyGive(TaskHandle_t xTaskToNotify);
```

# TASK NOTIFICATIONS API (cont.)

- Como a variável de notificação tem 32 bits, uma tarefa pode receber até 32 notificações de eventos diferentes.
- Neste caso, a tarefa pode esperar por múltiplos eventos utilizando a função `xTaskNotifyWait()`.
- Da mesma forma, as notificações podem ser enviadas com a função `xTaskNotify()`.

# TASK NOTIFICATIONS API (cont.)

```
#include "task.h"

/* waits for the calling task to receive a notification */
BaseType_t xTaskNotifyWait(uint32_t ulBitsToClearOnEntry,
                           uint32_t ulBitsToClearOnExit,
                           uint32_t *pulNotificationValue,
                           TickType_t xTicksToWait);

/* sends a notification to a specific task */
BaseType_t xTaskNotify(TaskHandle_t xTaskToNotify,
                       uint32_t ulValue,
                       eNotifyAction eAction);
```

# USANDO TASK NOTIFICATIONS

- Segundo a documentação do FreeRTOS, acordar uma tarefa utilizando Task Notifications é 45% mais rápido e consome menos RAM comparado com o uso de um semáforo binário.
- Esta funcionalidade é habilitada por padrão no FreeRTOS.
- Para desabilitá-la (e economizar 8 bytes por tarefa) é necessário configurar a opção `configUSE_TASK_NOTIFICATIONS` com `0` no `FreeRTOSConfig.h`.

Embedded **Labworks**

# LABORATÓRIO

## Utilizando task notifications

# USANDO QUEUES EM INTERRUPÇÕES

- Task notifications e semáforos são utilizados para comunicar eventos entre tarefas, ou entre uma tarefa e uma interrupção.
- Já os queues podem ser utilizados não só para comunicar eventos, mas também para transferir dados entre tarefas, ou entre uma tarefa e uma interrupção.
- Para trabalhar com queues em rotinas de tratamento de interrupção, utilize as funções que terminam com `FromISR`.

# ESCREVENDO NO QUEUE

```
#include "queue.h"

/* post an item to the front of a queue (from an ISR) */
BaseType_t xQueueSendToFrontFromISR(
                    QueueHandle_t xQueue,
                    const void *pvItemToQueue,
                    BaseType_t *pxHigherPriorityTaskWoken);

/* post an item to the back of a queue (from an ISR) */
BaseType_t xQueueSendToBackFromISR(
                    QueueHandle_t xQueue,
                    const void *pvItemToQueue,
                    BaseType_t *pxHigherPriorityTaskWoken);
```

# ESCREVENDO NO QUEUE (cont.)

```
#include "queue.h"

/* post an item on a queue (from an ISR) - same as
   xQueueSendToBackFromISR() */
BaseType_t xQueueSendFromISR(
                QueueHandle_t xQueue,
                const void *pvItemToQueue,
                BaseType_t *pxHigherPriorityTaskWoken);

/* A version of xQueueSendToBack() that will write to the
queue even if the queue is full, overwriting data that is
already held in the queue (to be used from an ISR) */
BaseType_t xQueueOverwriteFromISR(
                QueueHandle_t xQueue,
                const void *pvItemToQueue,
                BaseType_t *pxHigherPriorityTaskWoken);
```

# LENDO DO QUEUE

```
#include "queue.h"

/* receive an item from a queue (from an ISR) */
BaseType_t xQueueReceiveFromISR(
                QueueHandle_t xQueue,
                void *pvBuffer,
                BaseType_t *pxHigherPriorityTaskWoken);

/* receive an item from a queue without removing
   the item from the queue (from an ISR) */
BaseType_t xQueuePeekFromISR(
                QueueHandle_t xQueue,
                void *pvBuffer);
```

# TÉCNICAS PARA TRANSFERIR DADOS

- Você pode usar algumas técnicas para transferir dados de uma interrupção para uma tarefa:
  - Se a taxa de transferência for baixa, você pode simplesmente transferir byte a byte usando um queue.
  - Se a taxa de transferência for alta, você pode salvar os dados transferidos em um buffer, e quando receber uma mensagem completa, notificar a tarefa com um semáforo ou enviar a mensagem com um queue.
  - Você pode também decodificar mensagens direto da ISR, e passar os dados já interpretados via queue para a tarefa. Este técnica só é válida se a decodificação dos dados for rápida o suficiente para ser executada dentro da ISR.

Embedded **Labworks**

# LABORATÓRIO

## Deferindo trabalho com queues

# INTERRUPÇÃO NO ARM CORTEX-M

- O controle das interrupções em um ARM Cortex-M é realizado pelo NVIC (Nested Vectored Interrupt Controller).
- Esta arquitetura suporta até 240 interrupções e até 256 níveis de prioridade (estes valores são definidos pelo fabricante do chip).
- Para um microcontrolador com 16 níveis de prioridade de interrupção, 0 é o nível de maior prioridade e 15 é o nível de menor prioridade.
- Se durante a execução de uma interrupção, uma outra interrupção de maior prioridade acontecer, a interrupção atual será interrompida para a execução da interrupção de maior prioridade.

# INTERRUPÇÃO NO FREERTOS

- O FreeRTOS usa 3 fontes de interrupção no porte para o ARM Cortex-M:
  - **SysTick** (System Tick Timer): É uma interrupção periódica usada pelo kernel controlar o tick do sistema e forçar a troca de contexto no modo preemptivo, setando o registrador PENDSV do NVIC, e consequentemente habilitando a exceção PendSV.
  - **PendSV** (Pended System Call): Esta exceção fica pendente e é executada assim que outras exceções com maior prioridade forem tratadas. É ela que faz a troca de contexto.
  - **SVCall** (System Service Call): É uma interrupção de software que pode ser usada para gerar chamadas de sistema. É usada pelo kernel basicamente para executar a primeira tarefa da aplicação.

# CONFIGURANDO AS PRIORIDADES

- Existem duas opções no arquivo de configuração do FreeRTOS para configurar as prioridades das interrupções usadas pelo kernel:
  - `configKERNEL_INTERRUPT_PRIORITY`: configura a prioridade das interrupções usadas pelo kernel (SysTick e PendSV). É normalmente configurada com a menor prioridade possível.
  - `configMAX_SYSCALL_INTERRUPT_PRIORITY`: define a interrupção de maior prioridade que pode usar a API do FreeRTOS. Isso porque, ao executar uma seção crítica, o kernel desabilita todas as interrupções de prioridade menor ou igual à definida por esta constante. Isso significa que o FreeRTOS nunca desabilita todas as interrupções por completo, mesmo dentro de seções críticas!

# EXEMPLO

```
configKERNEL_INTERRUPT_PRIORITY = 15
configMAX_SYSCALL_INTERRUPT_PRIORITY = 5
```


Interrupções que NÃO usam a API do FreeRTOS podem ter qualquer prioridade.

Estas interrupções tem maior prioridade que o kernel, mas não podem usar a API do FreeRTOS.

Interrupções que podem usar a API do FreeRTOS, mas serão desabilitadas em regiões críticas

# NA PRÁTICA

- A constante `configKERNEL_INTERRUPT_PRIORITY` deve sempre estar configurada com a menor prioridade possível.
- Como todas as interrupções tem prioridade máxima (valor 0) por padrão no boot, as rotinas de tratamento de interrupção que usam serviços do FreeRTOS precisam ser inicializadas com um valor maior ou igual que `configMAX_SYSCALL_INTERRUPT_PRIORITY`.
- Rotinas de interrupção extremamente críticas podem ter uma prioridade maior, implicando um valor menor que `configMAX_SYSCALL_INTERRUPT_PRIORITY`, mas não podem usar nenhuma função da API do FreeRTOS.

Embedded Labworks

# FreeRTOS

## Gerenciamento de recursos

# GERENCIAMENTO DE RECURSOS

- Em um sistema multitarefa, existe a possibilidade de uma tarefa ser interrompida durante o acesso a um recurso, deixando este recurso em um estado inconsistente, e podendo causar problemas se uma outra tarefa tentar acessar este mesmo recurso.

- Exemplos:
  - Acesso à variáveis globais.
  - Acesso a periféricos.
  - Compartilhamento de código.

# ACESSO À VARIÁVEIS GLOBAIS

- Imagine o seguinte código em C:

```
contador += 10;
```

- Agora imagine seu correspondente em um assembly “genérico”:

```
LOAD    R, [#1234]
SUM     R, 10
STORE   R, [#1234]
```

- A operação em C acima não é atômica, porque precisa de mais de uma instrução da CPU para completar. Qual a consequência disso?

# ACESSO À VARIÁVEIS GLOBAIS (cont.)

- Imagine a seguinte situação:
  - Tarefa A carrega o valor da variável para o registrador (primeira instrução assembly).
  - Tarefa A é interrompida pela tarefa B.
  - Tarefa B altera o valor da variável global, e depois dorme.
  - Tarefa A retorna do ponto onde parou, e executa as outras duas instruções em assembly, mas usando um valor antigo da variável global, já que esta foi alterada pela tarefa B.
- Neste caso, como a variável global pode ser acessada por mais de uma tarefa ao mesmo tempo, é necessário um controle no acesso a esta variável para garantir sua consistência.

# ACESSO A PERIFÉRICOS

- Considere um cenário onde duas tarefas escrevem no display LCD:
  - Tarefa A começa a escrever "`Digite sua senha no display:`".
  - Enquanto escrevia no display, a tarefa A é interrompida pela tarefa B, e consegue escrever apenas "*`Digite sua sen`*".
  - Tarefa B escreve "*`Erro na comunicação`*" no display.
  - Tarefa A continua de onde parou e escreve "*`ha no display`*".
  - No fim, o display exibe a seguinte mensagem: "*`Digite sua senErro na comunicaçãoha no display`*`!`".
- O acesso concorrente a qualquer recurso de hardware por mais de uma tarefa da aplicação precisa ser gerenciado corretamente.

# FUNÇÕES THREAD-SAFE

- Em um ambiente multitarefa, se uma determinada função pode ser chamada por mais de uma thread (tarefa), esta função deve ser **thread-safe**.
- Isso é comum quando compartilhamos código entre tarefas, por exemplo através do uso de uma biblioteca.
- Uma função thread-safe proteje o acesso aos recursos compartilhados pelas tarefas, garantindo o acesso concorrente com segurança.

# EXEMPLO

```
// Esta implementação NÃO
// é thread-safe

struct Date d;

void getDate()
{
    d.day   = getDay();
    d.month = getMon();
    d.year  = getYear();
}
```

```
// Esta implementação
// é thread-safe

void getDate(struct Date *d)
{
    d->day   = getDay();
    d->month = getMon();
    d->year  = getYear();
}
```

# CRITICAL SESSION E MUTUAL EXCLUSION

- **Critical Sessions** são regiões de código que acessam um recurso compartilhado, e não devem ser acessados concorrentemente por mais de uma tarefa.
- O acesso a um recurso que é compartilhado entre tarefas deve ser gerenciado com a técnica de **mutual exclusion**, para garantir a consistência no acesso.
- A idéia é garantir que apenas uma tarefa tenha acesso exclusivo ao recurso em determinado momento.
- O FreeRTOS provê diversos mecanismos para implementar mutual exclusion.

# MUTEX

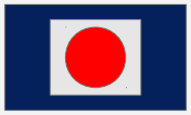
- O mutex é um tipo especial de semáforo binário usado para proteger o acesso a um recurso compartilhado por mais de uma tarefa.
- Um mutex é basicamente um token associado a determinado recurso.
- Para uma tarefa acessar o recurso, ela precisa antes pegar o token (**take**).

# MUTEX (cont.)

- Se o recurso estiver ocupado, ela deverá esperar a liberação do recurso para poder usá-lo.
- Ao terminar de usar o recurso, ela deverá liberá-lo (**give**).
- Este mecanismo é totalmente dependente da disciplina do desenvolvedor!

# EXEMPLO

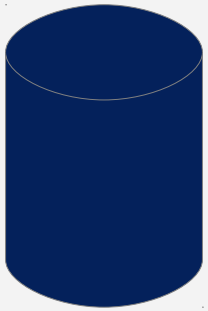
MUTEX



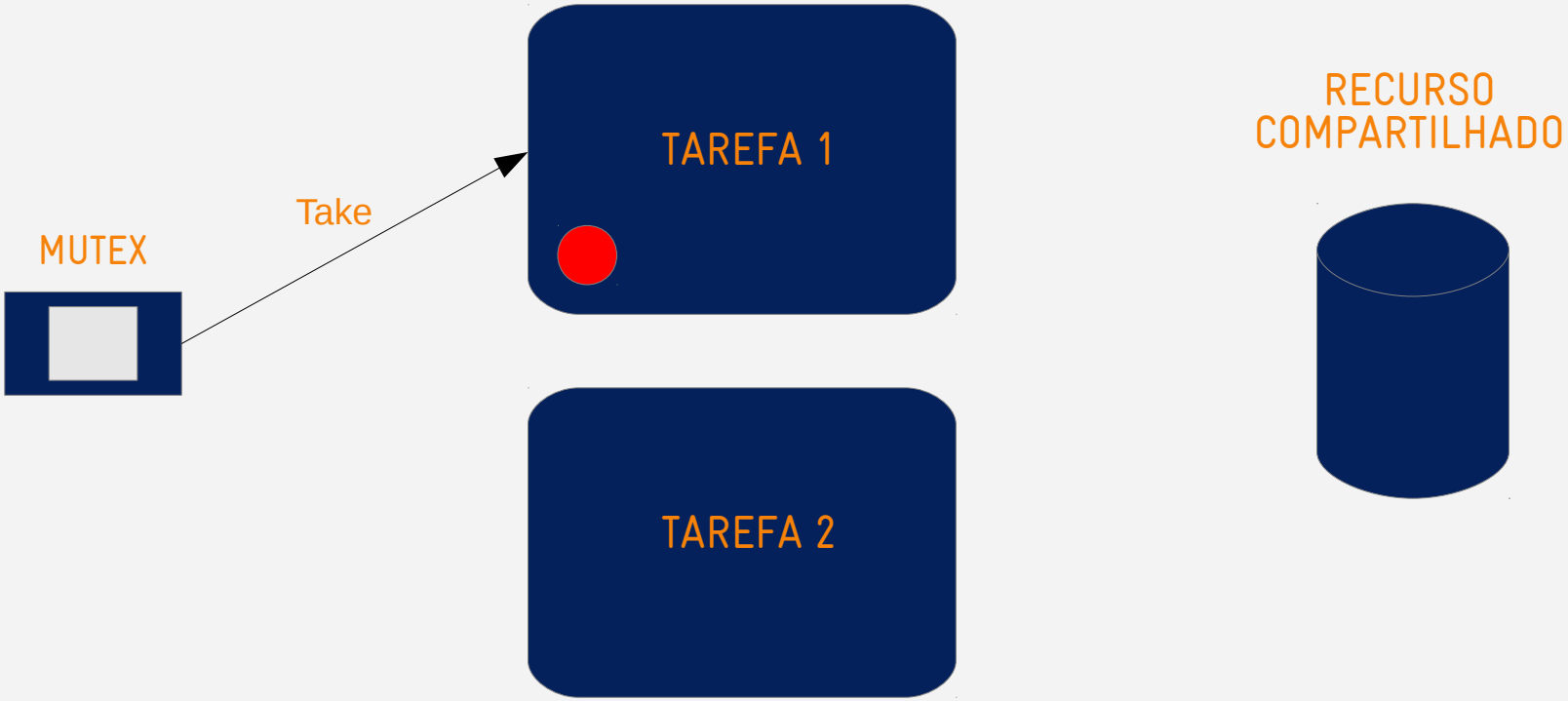
TAREFA 1

TAREFA 2

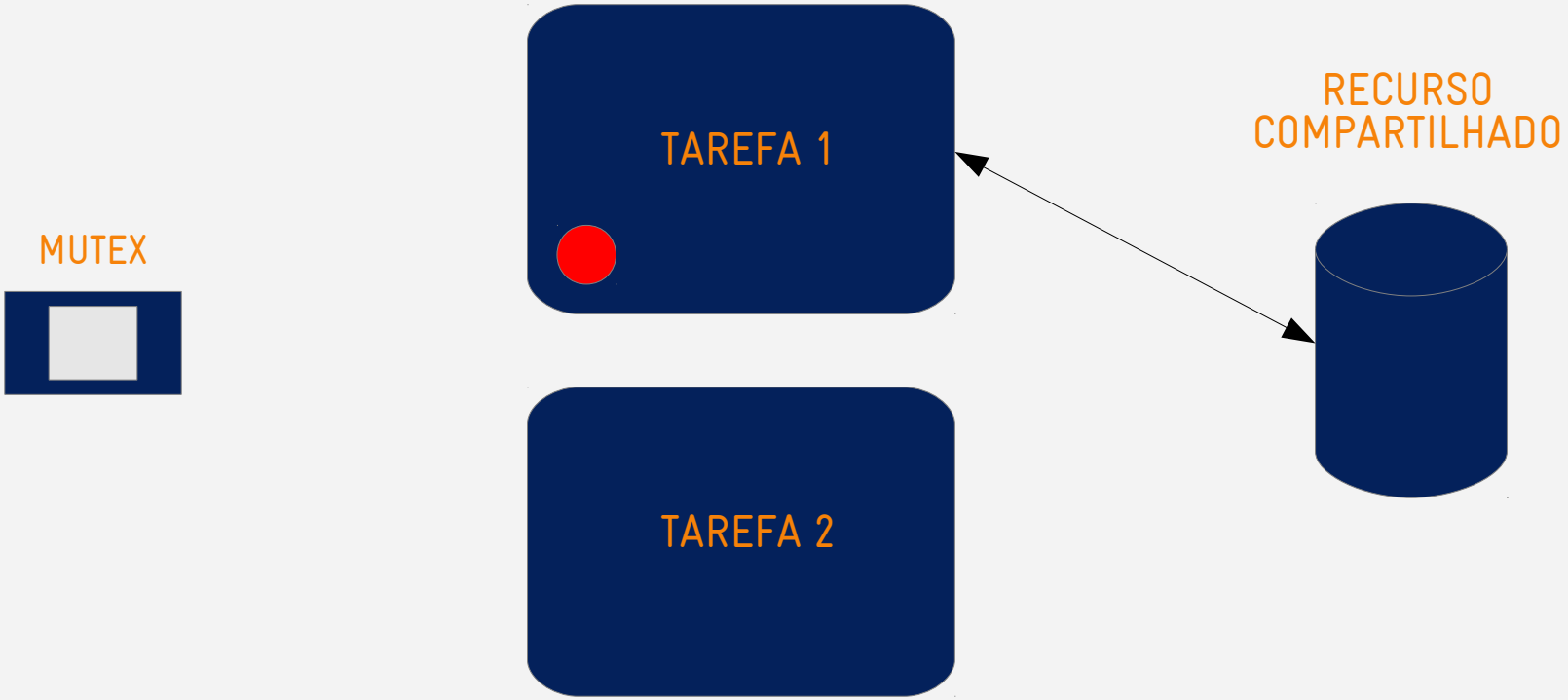
RECURSO COMPARTILHADO



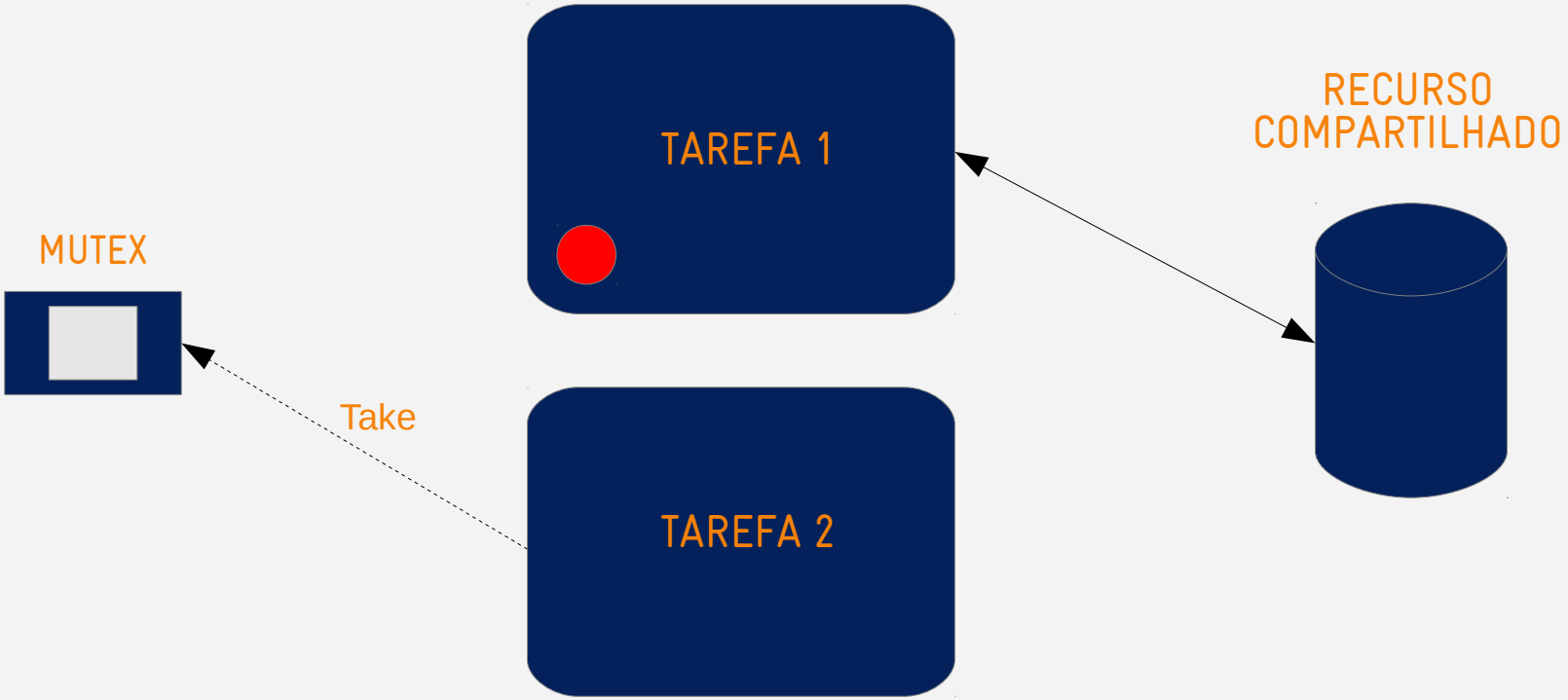
# EXEMPLO (cont.)



# EXEMPLO (cont.)

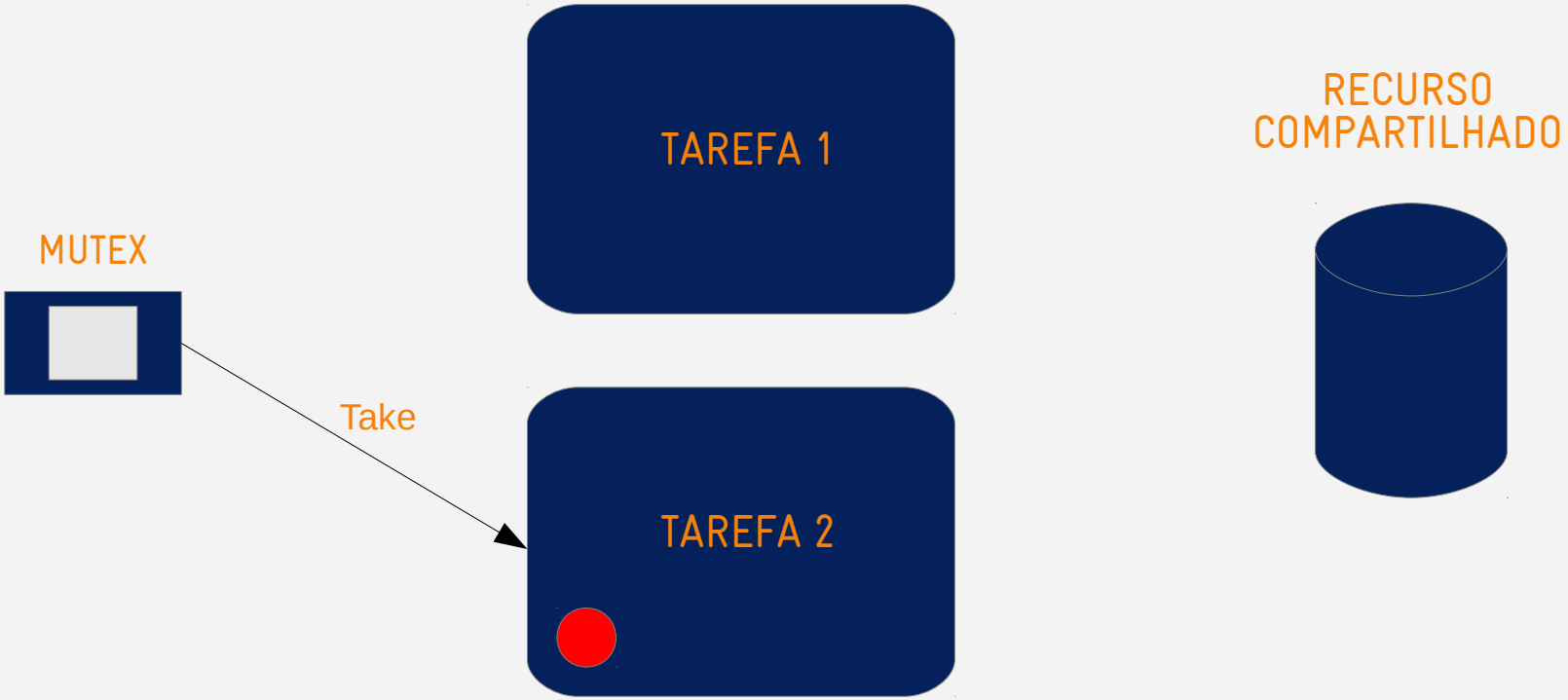


# EXEMPLO (cont.)

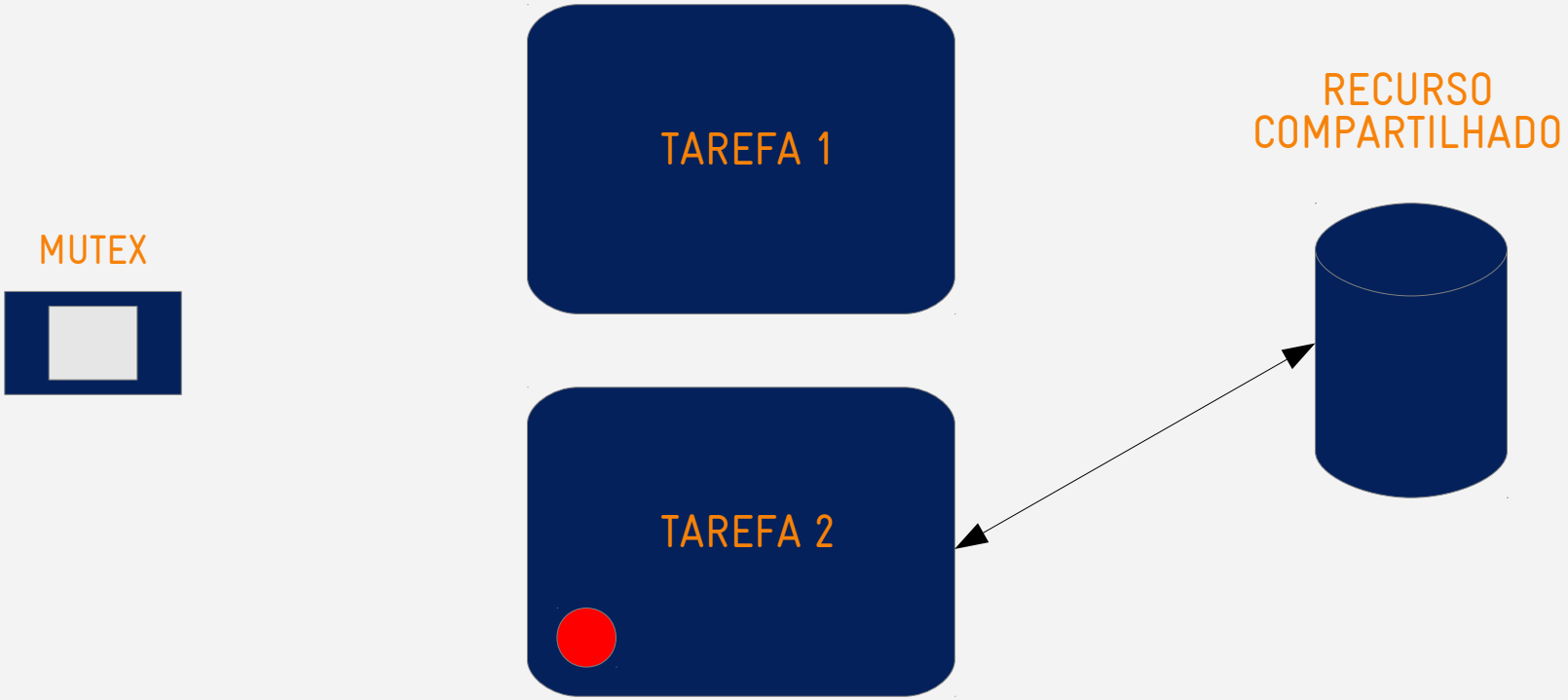


# EXEMPLO (cont.)

# EXEMPLO (cont.)



# EXEMPLO (cont.)



# USANDO UM MUTEX

```
#include "semphr.h"

/* create a mutex */
SemaphoreHandle_t xSemaphoreCreateMutex(void);

/* obtain a mutex */
BaseType_t xSemaphoreTake(
           SemaphoreHandle_t xSemaphore,
           TickType_t xTicksToWait);

/* release a mutex */
BaseType_t xSemaphoreGive(
           SemaphoreHandle_t xSemaphore);
```

# USANDO UM MUTEX RECURSIVO

```
#include "semphr.h"

/* create a recursive mutex */
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(void);

/* obtain a recursive mutex */
BaseType_t xSemaphoreTakeRecursive(
             SemaphoreHandle_t xSemaphore,
             TickType_t xTicksToWait);

/* release a recursive mutex */
BaseType_t xSemaphoreGiveRecursive(
                SemaphoreHandle_t xSemaphore);
```

# PAUSANDO O ESCALONADOR

- Em alguns casos, não é possível utilizar um mutex. Exemplos:
  - Código do kernel.
  - Bibliotecas utilizadas pelo kernel (Ex: rotinas de alocação de memória).
- Para estes casos, pode-se pausar o escalonador com a função `vTaskSuspendAll()`.

# PAUSANDO O ESCALONADOR (cont.)

- Ao pausar o escalonador, as interrupções continuam habilitadas, mas não haverá mudança de contexto, e outras tarefas não serão executadas.

- Para reiniciar o escalonador, pode-se usar a função `xTaskResumeAll()`.

# PAUSANDO O ESCALONADOR (cont.)

```
void *pvPortMalloc(size_t xWantedSize)
{
    void *pvReturn;

    vTaskSuspendAll();
    {
        pvReturn = malloc(xWantedSize);
        traceMALLOC(pvReturn, xWantedSize);
    }
    (void) xTaskResumeAll();

[...]
```

# PAUSANDO O ESCALONADOR (cont.)

- A tarefa que pausou o escalonador nunca deverá bloquear esperando um evento.
- Normalmente apenas o kernel utiliza esta funcionalidade.
- Para a aplicação, um mutex deverá ser suficiente, a não ser que seja necessário executar um trecho de código sem ser interrompido por outra tarefa.

# DESABILITANDO INTERRUPÇÕES

- Caso o recurso seja compartilhado entre uma tarefa e uma interrupção, a única forma de gerenciar o acesso compartilhado à este recurso é desabilitando as interrupções.

- Para isso, existem duas famílias de funções:
  - `taskDISABLE_INTERRUPTS()` e `taskENABLE_INTERRUPTS()`: desabilita e habilita as interrupções, respectivamente.
  - `taskENTER_CRITICAL()` e `taskEXIT_CRITICAL()`: mesmo comportamento, porém permite chamadas aninhadas.

# DESABILITANDO INTERRUPÇÕES (cont.)

- Estas funções desabilitam todas as interrupções até a interrupção com prioridade definida na constante `configMAX_SYSCALL_INTERRUPT_ PRIORITY`.
- Desabilitam também a preempção, já que a preempção é realizada pela interrupção de tick.
- Para evitar problemas em chamadas aninhadas, é aconselhável o uso das funções `taskENTER_CRITICAL()` e `taskEXIT_CRITICAL()`.

# DESABILITANDO INTERRUPÇÕES (cont.)

- Por questões de performance, pode-se avaliar o uso destas funções ao invés de um mutex para gerenciar o acesso a recursos compartilhados entre tarefas.

- Mas lembre-se sempre que desabilitar as interrupções aumenta o tempo de latência do sistema!

# RESUMO

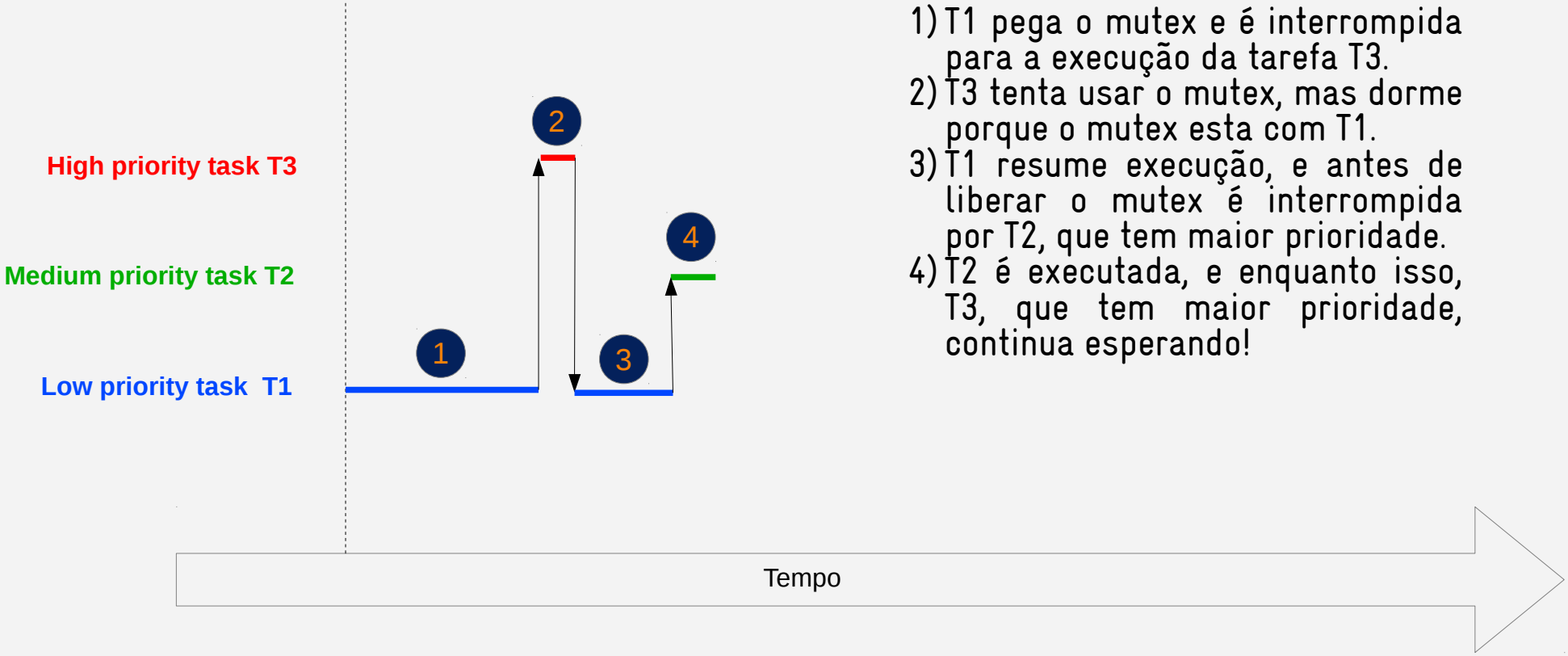
- Para compartilhar um recurso entre tarefas e interrupções, use as funções `taskENTER_CRITICAL()` e `taskEXIT_CRITICAL()`.
- Para compartilhar recursos entre tarefas:
  - Por padrão, use um mutex.
  - Caso a performance seja importante, e a seção crítica seja pequena, avalie o uso das funções `taskENTER_CRITICAL()` e `taskEXIT_CRITICAL()`.
- Regra geral: quanto menos recursos compartilhados, melhor!

Embedded **Labworks**

# LABORATÓRIO

## Controlando acesso com mutex

# INVERSÃO DE PRIORIDADE




1) T1 pega o mutex e é interrompida para a execução da tarefa T3.
2) T3 tenta usar o mutex, mas dorme porque o mutex esta com T1.
3) T1 resume execução, e antes de liberar o mutex é interrompida por T2, que tem maior prioridade.
4) T2 é executada, e enquanto isso, T3, que tem maior prioridade, continua esperando!

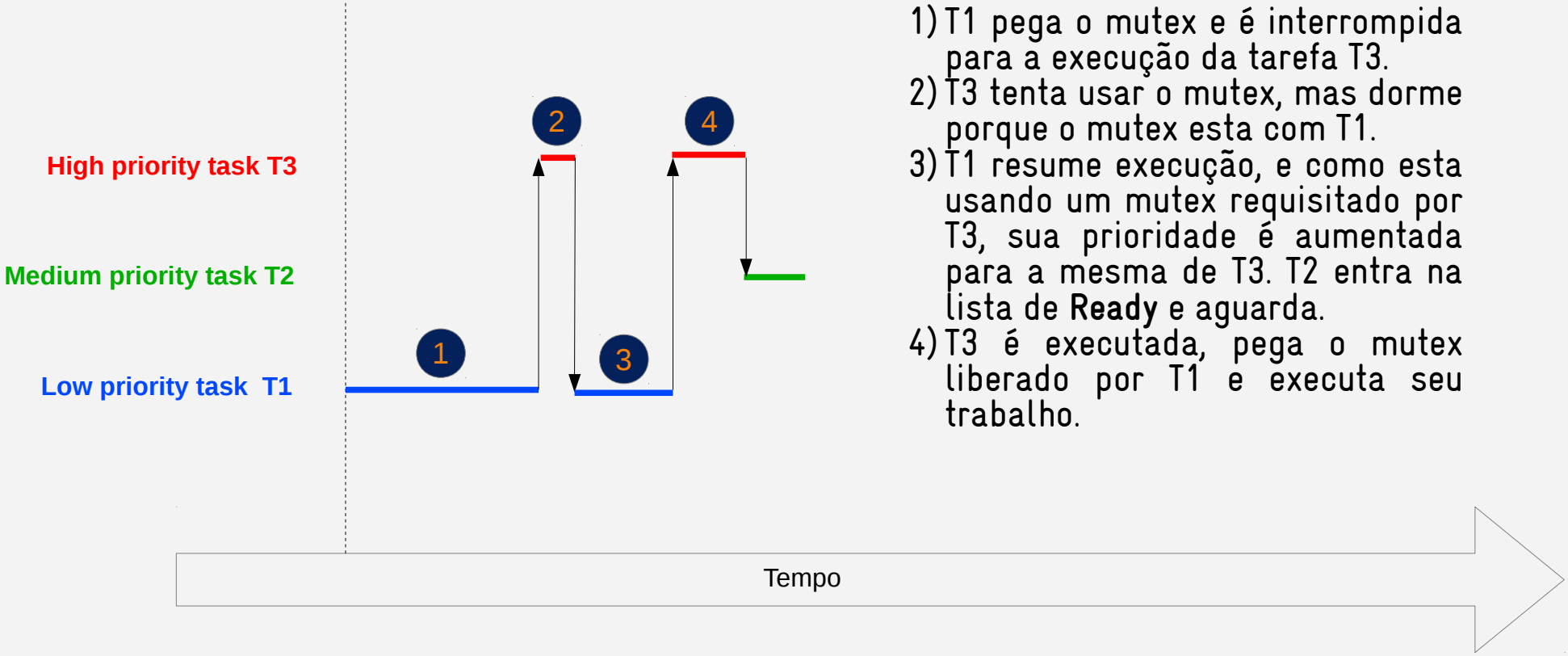
# HERANÇA DE PRIORIDADE

- A inversão de prioridade acontece quando uma tarefa de maior prioridade precisa esperar uma tarefa de menor prioridade ser executada.

- Para diminuir o impacto da inversão de prioridade, o FreeRTOS usa a técnica de **herança de prioridade**.

- No nosso exemplo, a prioridade da tarefa T1, que contém o mutex, é aumentada momentaneamente para a mesma prioridade de T3 para finalizar o processamento, liberar o mutex, e possibilitar a execução da tarefa T3.

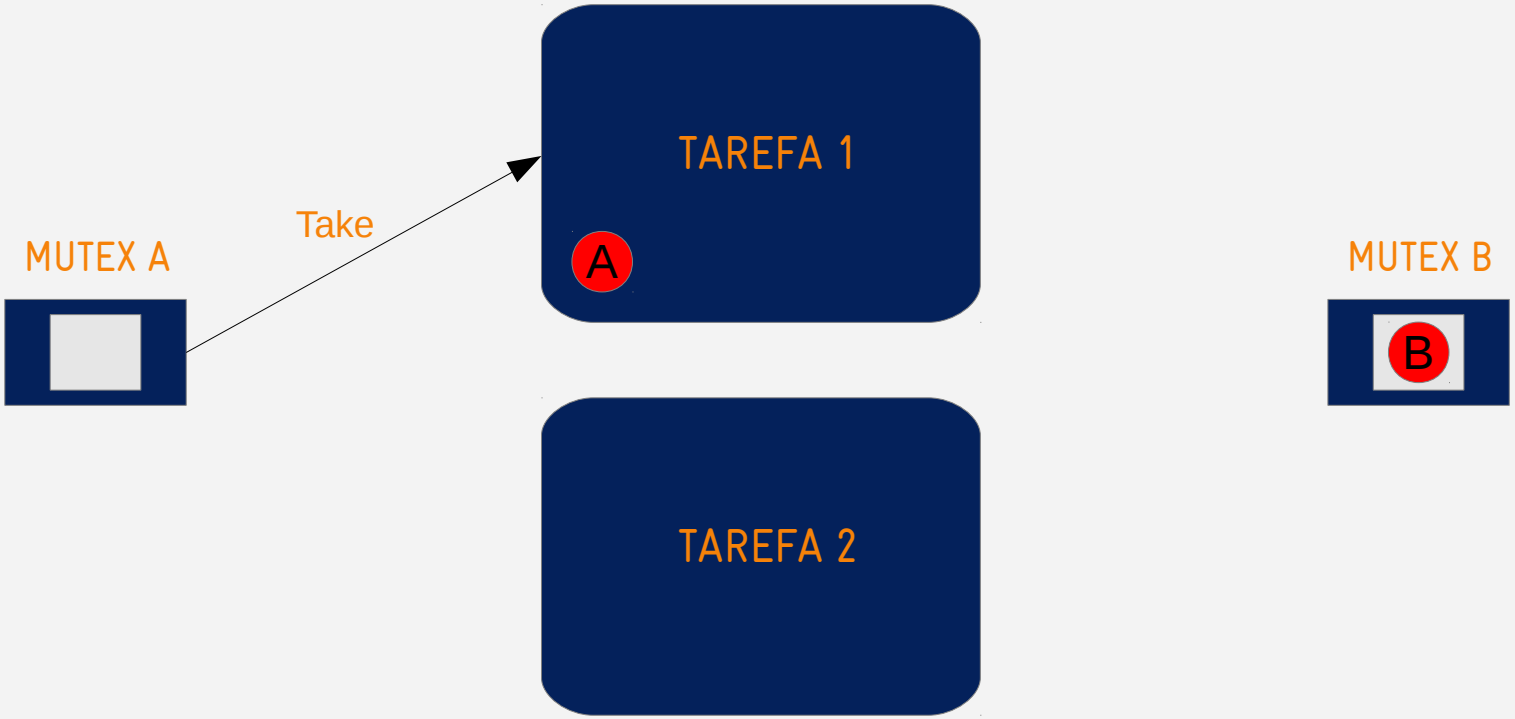
# HERANÇA DE PRIORIDADE (cont.)



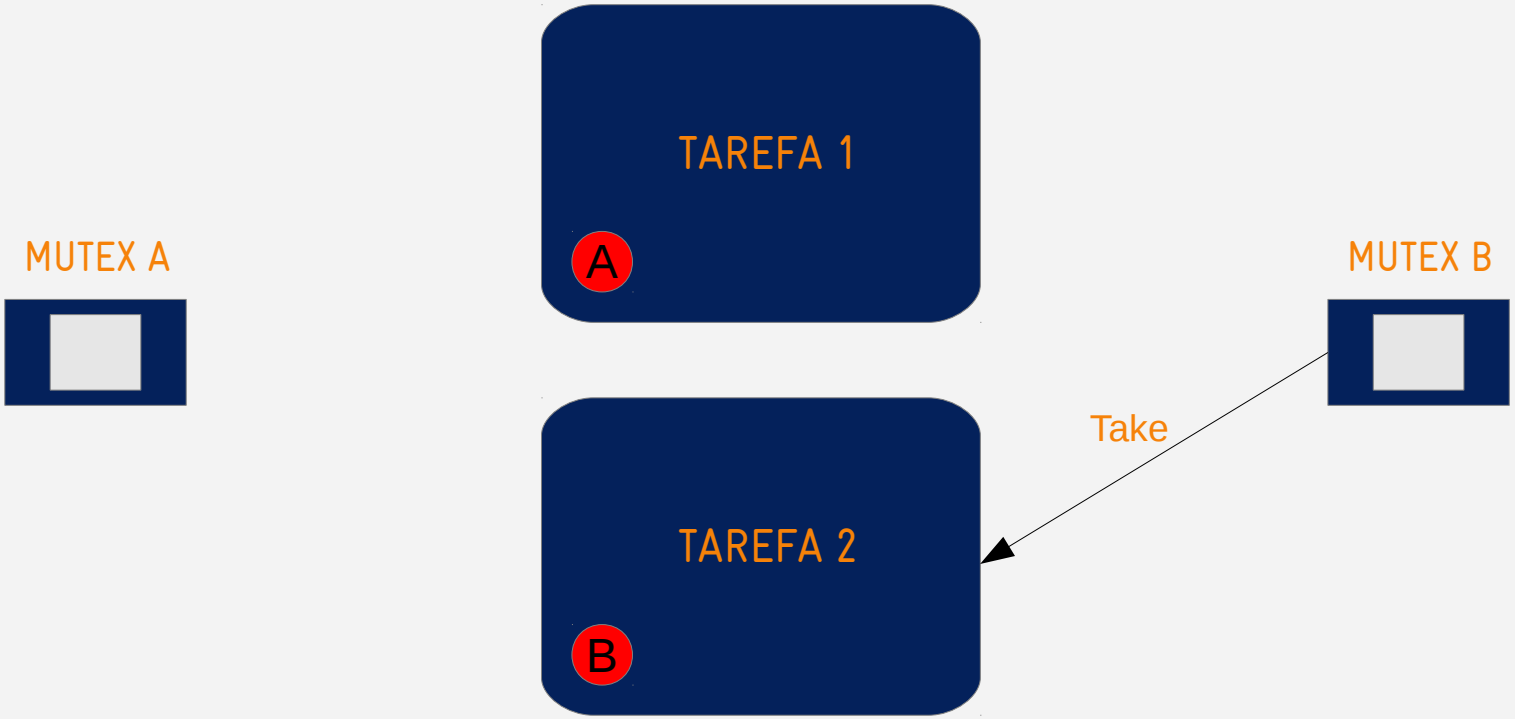

1) T1 pega o mutex e é interrompida para a execução da tarefa T3.
2) T3 tenta usar o mutex, mas dorme porque o mutex esta com T1.
3) T1 resume execução, e como esta usando um mutex requisitado por T3, sua prioridade é aumentada para a mesma de T3. T2 entra na lista de **Ready** e aguarda.
4) T3 é executada, pega o mutex liberado por T1 e executa seu trabalho.

# DEADLOCK

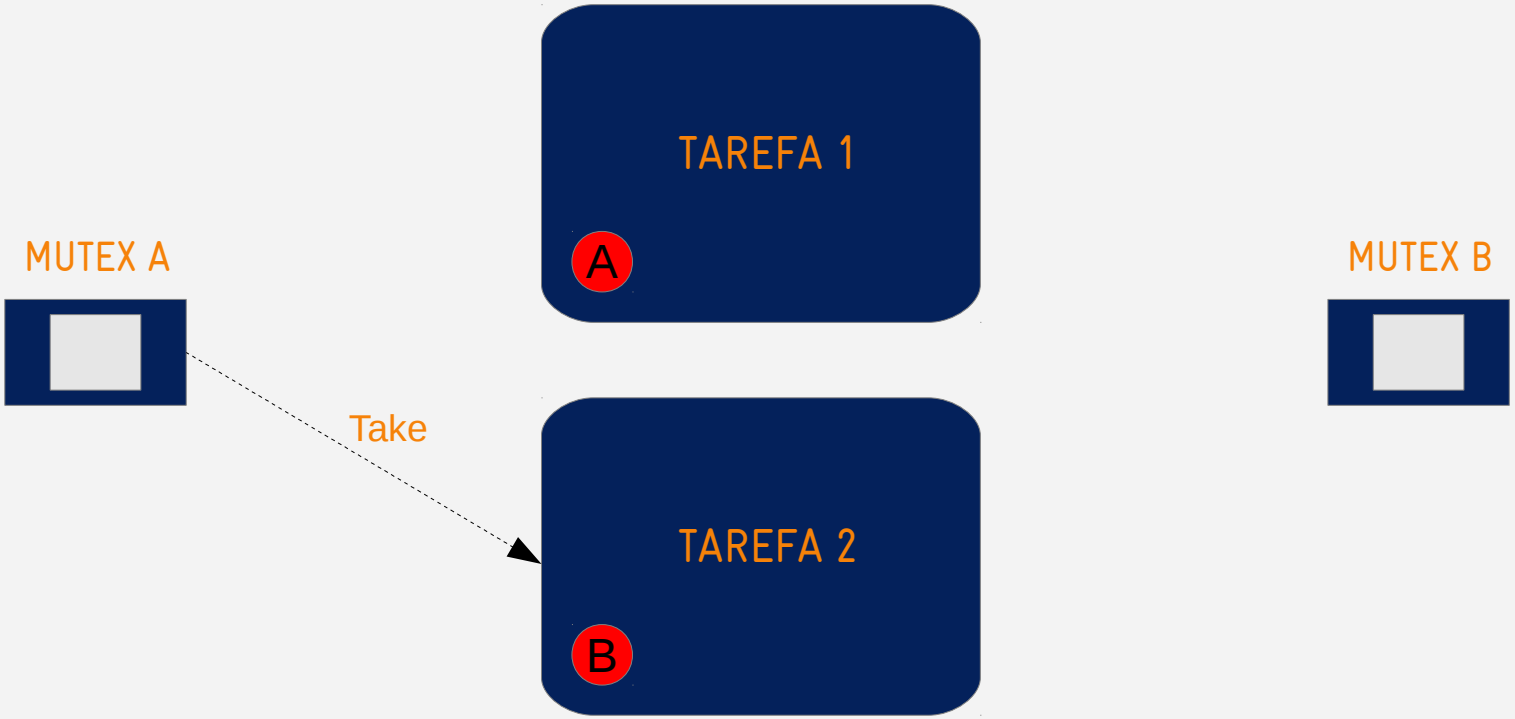
# DEADLOCK (cont.)



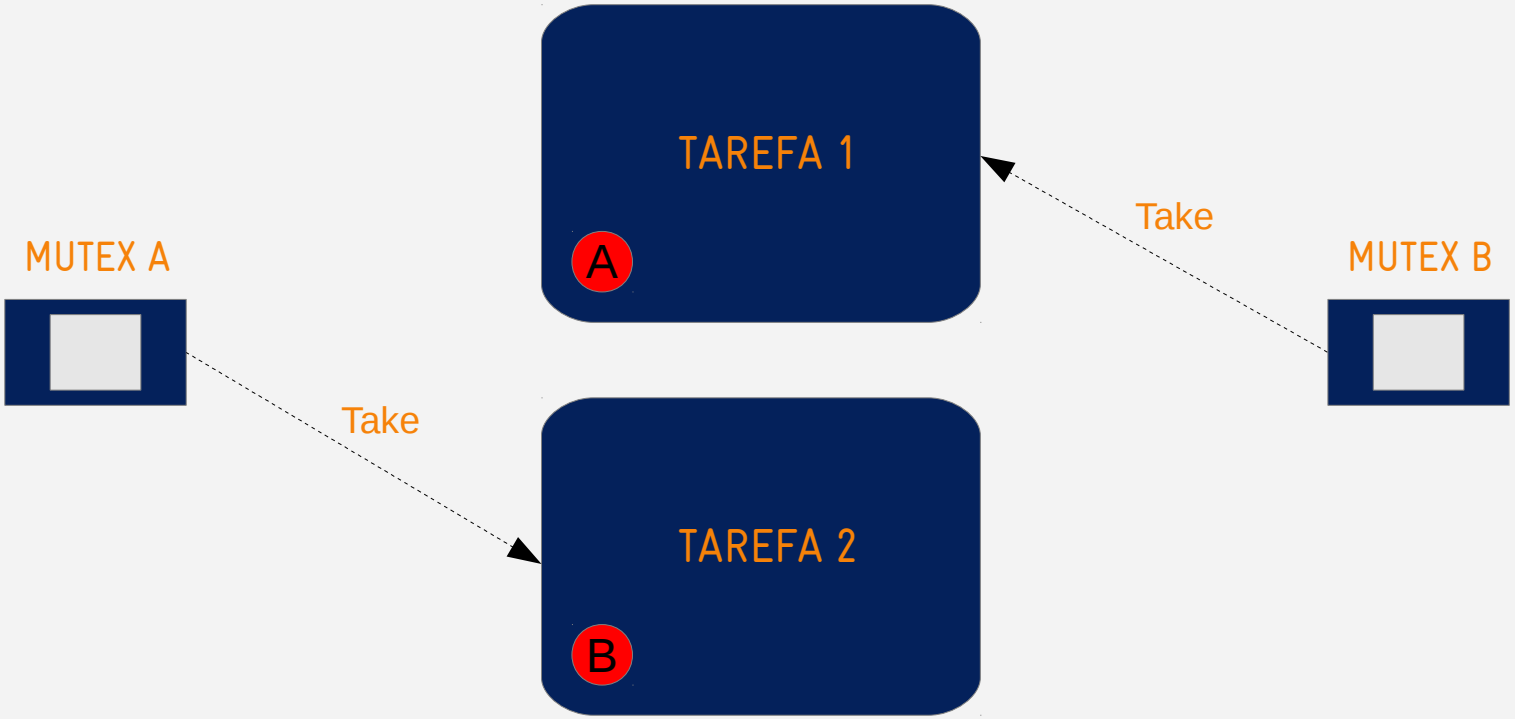
# DEADLOCK (cont.)



# DEADLOCK (cont.)



# DEADLOCK (cont.)



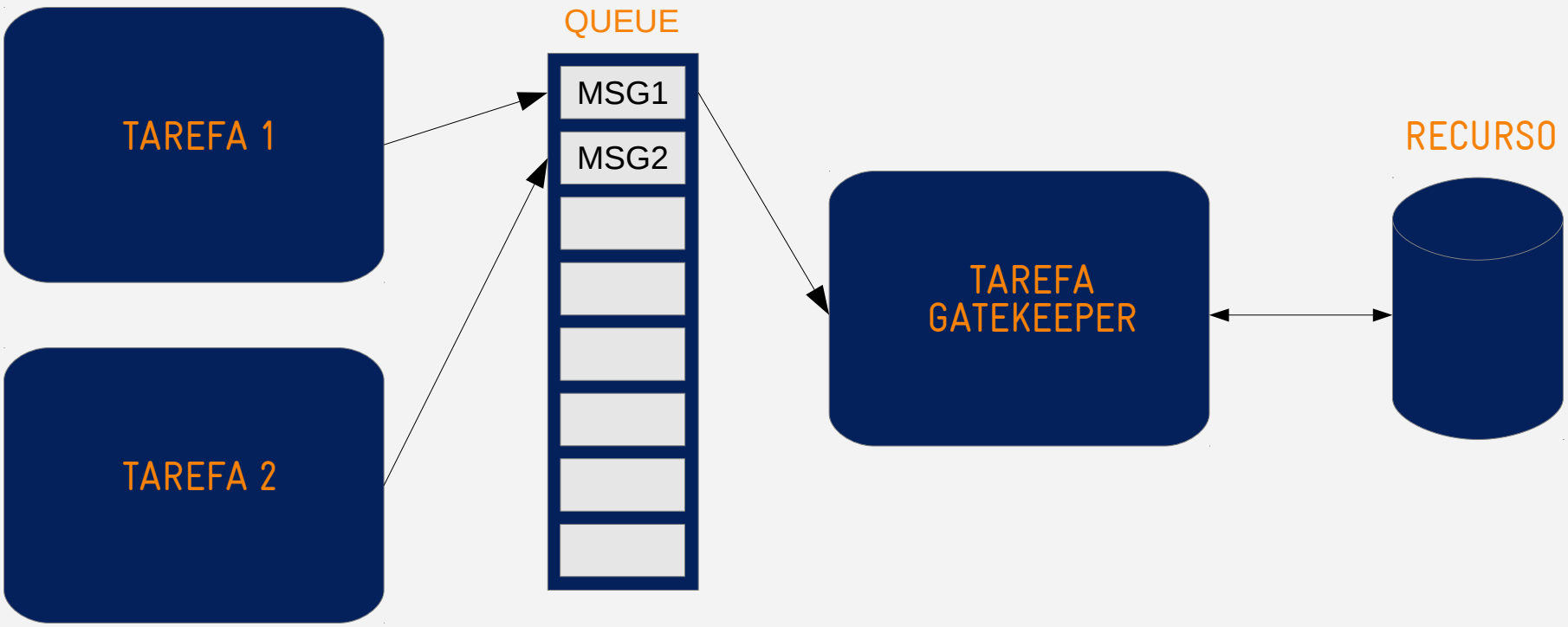
# DEADLOCK (cont.)

- Deadlocks acontecem quando duas ou mais tarefas estão simultaneamente aguardando por recursos que estão sendo ocupados por outra(s) tarefa(s).
- Neste caso, nenhuma das tarefas será executada novamente!
- Deadlocks são erros de projeto da aplicação.

# GATEKEEPER

- Gatekeeper é uma técnica para implementar um mecanismo de **mutual exclusion** sem os riscos de inversão de prioridade ou deadlocks.
- Uma tarefa do tipo gatekeeper possui acesso exclusivo a determinado recurso, fornecendo serviços para outras tarefas acessarem este recurso.
- Todas as tarefas que querem acessar o recurso protegido devem utilizar os serviços fornecidos pela tarefa gatekeeper.
- Essa técnica permite também que uma interrupção acesse um recurso mais facilmente!

# GATEKEEPER (cont.)



Embedded Labworks

# LABORATÓRIO

## Trabalhando com tarefas Gatekeeper

Embedded Labworks

# FreeRTOS

## Gerenciamento de memória

# GERENCIAMENTO DE MEMÓRIA

- O FreeRTOS irá alocar memória dinamicamente toda vez que precisar criar um objeto do sistema (tarefa, queue, semáforo, etc).
- Por este motivo, o FreeRTOS precisa de rotinas de alocação de memória.
- É muito comum a biblioteca do sistema disponibilizar estas rotinas, normalmente chamadas de `malloc()` e `free()`.
- Podemos usá-las? Nem sempre!

# GERENCIAMENTO DE MEMÓRIA

- Cada aplicação pode ter necessidades diferentes com relação à alocação de memória, como por exemplo:
  - Footprint (espaço ocupado em RAM e flash).
  - Algoritmo de desfragmentação.
  - Determinismo.
- Nem sempre as rotinas de alocação de memória do sistema são capazes de atender as necessidades de uma aplicação de tempo real.
- Por este motivo, no FreeRTOS a alocação de memória é tratada na camada portável, e o usuário deve prover ao sistema a implementação das rotinas de alocação e desalocação de memória.

# GERENCIAMENTO DE MEMÓRIA

- Para alocar memória, ao invés de chamar `malloc()` diretamente, o FreeRTOS chama `pvPortMalloc()`.
- Da mesma forma, para liberar memória, ao invés de chamar a função `free()`, o FreeRTOS chama a função `vPortFree()`.
- Cabe ao desenvolvedor da aplicação fornecer uma implementação para as rotinas de alocação de memória `pvPortMalloc()` e `vPortFree()`.

# ALOCAÇÃO NO FREERTOS

- O FreeRTOS fornece 5 diferentes implementações para estas rotinas em `Source/portable/MemMang`.
- Cada uma destas implementações possuem diferentes características, e podem ser selecionadas conforme as necessidades da aplicação.

# ALOCAÇÃO NO FREERTOS (cont.)

- `heap_1.c`: apenas aloca memória.
- `heap_2.c`: aloca e desaloca memória, mas não trata fragmentação.
- `heap_3.c`: usa a implementação padrão de `malloc()` e `free()` da biblioteca C.
- `heap_4.c`: aloca e desaloca memória, trata fragmentação e é mais eficiente que a maioria das implementações da biblioteca C padrão.
- `heap_5.c`: utiliza o mesmo algoritmo que a `heap_4.c`, porém permite utilizar como heap regiões não contínuas de memória.

# HEAP_1

- Implementação básica de `pvPortMalloc()`.
- Não implementa `vPortFree()`.
- Este algoritmo implementa o heap através de um array de bytes definido em tempo de compilação, que será dividido em pequenos blocos a cada chamada a `pvPortMalloc()`.
- O tamanho deste heap pode ser configurado na constante `configTOTAL_HEAP_SIZE` definida no `FreeRTOSConfig.h`.

# HEAP_1 (cont.)

- Como a alocação do heap é estática (em tempo de compilação), pode ser que o compilador reclame que o programa está usando muita memória RAM.
- Esta implementação é sempre determinística, ou seja, o tempo de execução é sempre constante em qualquer chamada à `pvPortMalloc()`.
- **Quando usar**: em aplicações que apenas alocam, mas não desalocam memória.

# HEAP_2

- Implementa tanto a função de alocação `pvPortMalloc()` quanto a função de desalocação `vPortFree()`.
- Também implementa o heap através de um array de bytes definido em tempo de compilação, que será dividido em pequenos blocos a cada chamada a `pvPortMalloc()`.
- O tamanho deste heap pode ser configurado na constante `configTOTAL_HEAP_SIZE` definida no `FreeRTOSConfig.h`.
- Da mesma forma que `heap_1`, como a alocação do heap é estática, pode ser que o compilador reclame que o programa está usando muita memória RAM!

# HEAP_2 (cont.)

- Ao alocar, o algoritmo procura pelo menor bloco de memória possível para realizar a alocação requisitada.
- Mas como não combina blocos de memória livres e adjacentes, pode sofrer de fragmentação.
- **Quando usar**: em aplicações que sempre alocam e desalocam uma quantidade fixa de bytes. Por exemplo, quando uma aplicação cria e remove tarefas com o mesmo tamanho de stack frequentemente.

# HEAP_3

- Usa a implementação de `malloc()` e `free()` da biblioteca do sistema.
- Suspende o escalonador para tornar estas funções thread-safe.
- Nesta implementação, **NÃO** usa um buffer alocado estaticamente em tempo de compilação.

# HEAP_3 (cont.)

- Utiliza as configurações do linker para definir a localização e o tamanho do heap.
- **Quando usar**: em aplicações que alocam e desalocam constantemente buffers de tamanhos diferentes, e quando você confia na sua biblioteca de sistema!

# HEAP_4

- Disponível a partir da versão 7.2.0 do FreeRTOS.
- Aloca e desaloca memória, igual ao `heap_2`.
- Mas tem um algoritmo capaz de combinar regiões adjacentes de memória, diminuindo bastante os riscos de fragmentação de memória.
- **Quando usar**: em aplicações que alocam e desalocam constantemente buffers de tamanhos diferentes.

# HEAP_5

- Disponível a partir da versão 8.1.0 do FreeRTOS.
- Utiliza o mesmo algoritmo da `heap_4.c`.
- Capaz de utilizar como heap regiões não contínuas de memória.
- **Quando usar**: em sistemas onde a memória RAM é endereçada de forma não contínua.

# HEAP_5 (cont.)

```
/* allocate two blocks of RAM for use by the heap */
const HeapRegion_t xHeapRegions[] =
{
    { ( uint8_t * ) 0x80000000UL, 0x10000 },
    { ( uint8_t * ) 0x90000000UL, 0xa0000 },
    { NULL, 0 }
};

/* pass the array into vPortDefineHeapRegions() */
vPortDefineHeapRegions(xHeapRegions);
```

# LENDO O TAMANHO LIVRE DO HEAP

- Se a sua aplicação estiver usando as implementações `heap_1`, `heap_2`, `heap_4` ou `heap_5`, é possível ler o tamanho livre do heap com a função abaixo:

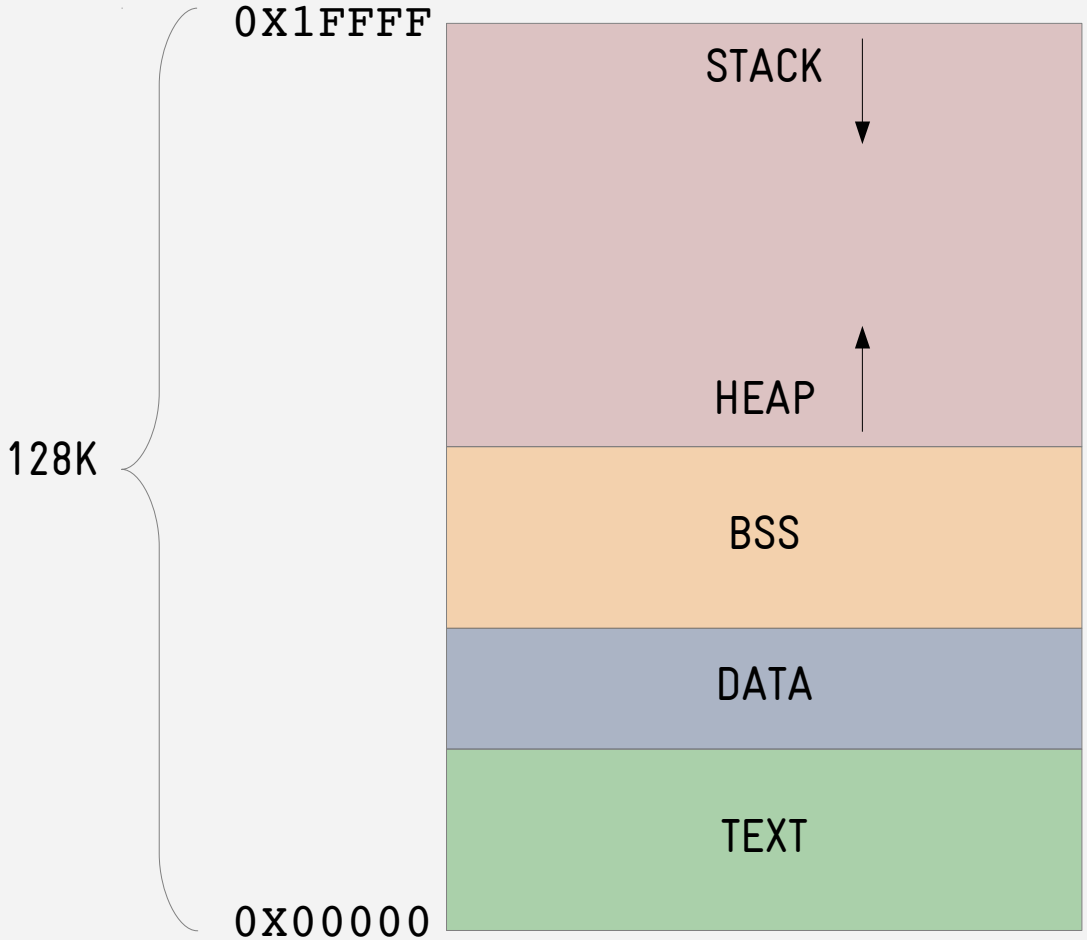
```
/* return free heap memory size (in bytes) */
size_t xPortGetFreeHeapSize(void);
```

# MALLOC FAILED HOOK

- Se der erro na alocação de memória (`pvPortMalloc()` retornar `NULL`), o FreeRTOS é capaz de chamar uma função de callback definida pelo desenvolvedor.
- Para isso, basta definir `configUSE_MALLOC_FAILED_HOOK` com `1` no `FreeRTOSConfig.h`.
- E então implementar a função de callback conforme protótipo abaixo:

```
void vApplicationMallocFailedHook(void);
```

# DE ONDE VEM A MEMÓRIA?



Embedded **Labworks**

# LABORATÓRIO

## Monitorando e ajustando o heap

# STACK

- O stack é uma região de memória usada para armazenar variáveis locais e salvar registradores e parâmetros durante as chamadas de função.
- Cada tarefa tem o seu stack, definido na criação da tarefa.
- Se o tamanho do stack for subdimensionado, existe a possibilidade da tarefa ultrapassar o espaço alocado para o stack na criação da tarefa.
- Chamamos este problema de estouro de pilha ou stack overflow.

# STACK OVERFLOW

- Stack overflow é uma das principais causas de problemas encontrados no FreeRTOS, principalmente por novos usuários.
- Existem algumas técnicas que podem ser usadas para monitorar o uso do stack e detectar stack overflow.

# HIGH WATER MARK

- O FreeRTOS provê uma função para a aplicação requisitar o quanto uma tarefa está perto de ultrapassar o espaço alocado para o stack.
- Quanto mais perto de zero, mais próximo da ocorrência de stack overflow.
- Este valor é chamado de "**high water mark**", e pode ser obtido através da função `uxTaskGetStackHighWaterMark()`.

# HIGH WATER MARK (cont.)

```
/* return the minimum amount of remaining stack space
   (in words) that was available to the task since the
   task started executing */
UBaseType_t uxTaskGetStackHighWaterMark(
                                TaskHandle_t xTask);
```

Embedded **Labworks**

# LABORATÓRIO

## Monitorando o stack de uma tarefa

# STACK OVERFLOW HOOK

- O FreeRTOS provê dois mecanismos para monitorar o stack das tarefas em tempo de execução.
- Estes mecanismos são habilitados através da constante `configCHECK_FOR_STACK_OVERFLOW` no arquivo `FreeRTOSConfig.h`.
- Se um destes mecanismos for habilitado, o kernel irá monitorar o stack das tarefas e executar uma função de callback (stack overflow hook) caso identifique stack overflow.
- Ambos os métodos aumentam o tempo necessário para realizar a troca de contexto.

# STACK OVERFLOW HOOK (cont.)

- Para usar este mecanismo de checagem do stack, basta configurar a constante `configCHECK_FOR_STACK_OVERFLOW` com `1` ou `2` e prover a implementação da função de callback de stack overflow, conforme protótipo abaixo:

```
void vApplicationStackOverflowHook(
      TaskHandle_t xTask, signed char *pcTaskName);
```

- Use esta função para identificar e corrigir problemas de stack durante o desenvolvimento da aplicação.

- O objetivo desta função é simplificar a depuração de problemas com o stack. De qualquer forma, não existe nenhum jeito fácil de se recuperar de um stack overflow.

# MÉTODO 1

- O método 1 de checagem do stack é selecionado quando configuramos `configCHECK_FOR_STACK_OVERFLOW` com 1.
- Neste método, após a troca de contexto de uma tarefa, o kernel verifica se o stack pointer está dentro dos limites do stack da tarefa. Se não estiver, chamará a função de stack overflow hook definida pela aplicação.
- Este método é rápido, mas pode perder alguns stack overflows, já que apenas verifica o stack pointer na troca de contexto, ou seja, este método não sabe se durante algum momento na execução da tarefa o stack pointer ultrapassou os limites do stack.

# MÉTODO 2

- O método 2 de checagem do stack é selecionado quando configuramos `configCHECK_FOR_STACK_OVERFLOW` com 2, e realiza checagens adicionais além das descritas no método 1.

- Quando uma tarefa é criada, o stack é preenchido com um padrão conhecido. Este método verifica se este padrão foi sobreescrito nos últimos 20 bytes do stack. Em caso afirmativo, a função de stack overflow hook será chamada.

- O método 2 não é tão rápido quanto o método 1, mas tem a vantagem de garantir quase que 100% de acerto.

Embedded **Labworks**

# LABORATÓRIO

## Monitorando o stack com callbacks

Embedded Labworks

# FreeRTOS

## Outras APIs do FreeRTOS

# TRATANDO MÚLTIPLOS EVENTOS

- Um problema comum em aplicações com um RTOS é quando temos uma tarefa que pode receber múltiplos eventos de outras tarefas do sistema.
- Se estes eventos tiverem características diferentes (uns apenas notificam, outros enviam um byte, outros enviam um buffer de dados, etc), você não pode usar apenas um queue ou semáforo para receber estes eventos.
- Um design pattern comum usado nestes casos é a criação de uma estrutura para representar estes eventos, contendo o ID do evento e um ponteiro para os dados do evento, e enviar elementos desta estrutura através de queues.

# TRATANDO MÚLTIPLOS EVENTOS (cont.)

```
/* application events structure */
typedef struct
{
    int id;
    void *data;
} AppEvent;
```

# TRATANDO MÚLTIPLOS EVENTOS (cont.)

```
void vTaskEvent(void * pvParameters)
{
    AppEvent event;

    for(;;)
    {
        /* wait event */
        xQueueReceive(eventQueue, &event, portMAX_DELAY);

        /* handle event */
        switch(event.id) {

            case EVENT_KEY:
                processEventKey(event.data);
                break;

            case EVENT_PRINTER:
                processEventPrinter();
                break;
        [...]
```

# QUEUE SETS

- A partir da versão 7.4.0, o FreeRTOS possibilita solucionar este problema com uma nova API chamada **Queue Sets**.
- Com os queue sets é possível fazer com que uma tarefa bloqueie esperando por múltiplos semáforos e/ou queues ao mesmo tempo.
- Para usar esta funcionalidade é necessário:
  - Criar um queue set com a função `xQueueCreateSet()`.
  - Agrupar semáforos e/ou queues com a função `xQueueAddToSet()`.
  - Bloquear esperando pela recepção de um destes elementos com a função `xQueueSelectFromSet()`.

# EXEMPLO: QUEUE SETS

```
void vTaskEvent(void * pvParameters)
{
    xQueueSetMemberHandle xActivatedMember;
    xQueueSetHandle xQueueSet;
    unsigned char key;

    /* create queue set */
    xQueueSet = xQueueCreateSet(EVENT_MAX);

    /* add queues to queue set */
    xQueueAddToSet(xQueueKey, xQueueSet);

    /* add semaphores to queue set */
    xQueueAddToSet(xSemaphoreDisplay, xQueueSet);
    xQueueAddToSet(xSemaphorePrinter, xQueueSet);

    [...]
```

# EXEMPLO: QUEUE SETS (cont.)

```
[...]

for( ;; )
{
    /* wait event */
    xActivatedMember = xQueueSelectFromSet(xQueueSet, portMAX_DELAY);

    /* handle event */
    if(xActivatedMember == xQueueKey) {
        xQueueReceive(xActivatedMember, &key, 0);
        processEventKey(key);
    }
    else if(xActivatedMember == xSemaphoreDisplay) {
        xSemaphoreTake(xActivatedMember, 0);
        processEventDisplay();
    }

    [...]
```

# DESVANTAGENS

- Comparado à solução inicial, o uso de queue sets possui algumas desvantagens:
  - A implementação usando queue sets utiliza mais memória RAM, já que precisamos criar um queue ou semáforo para cada tipo de evento.
  - O código fica maior, ocupando mais espaço em flash.
  - Consome mais ciclos de CPU, já que verificar um queue set leva mais tempo do que verificar um simples queue.

# QUANDO USAR?

- A principal motivação para o uso desta API é a migração de aplicações para o FreeRTOS.
- É muito comum os RTOSs de mercado terem uma função que bloqueia em múltiplos objetos do kernel. O `uC/OS-III` por exemplo tem a função `OSPendMulti()`.
- Então uma API deste tipo facilitaria a migração de aplicações que rodam em outros RTOSs para o FreeRTOS.

# EVENT GROUPS

- O FreeRTOS possui uma funcionalidade interessante chamada Event Groups, disponível a partir da versão 8.0.0.
- A idéia é agrupar bits (Event Bits) em grupos (Event Groups), que podem ser usados como mecanismo de sincronização (notificação) entre tarefas.
- Um Event Group pode ser criado com a função `xEventGroupCreate()`.

# EVENT GROUPS (cont.)

- Uma tarefa pode bloquear esperando por uma mudança em um ou mais Event Bits com a função `xEventGroupWaitBits()`.
- Outra tarefa pode setar um ou mais Event Bits com a função `xEventGroupSetBits()`.

# EVENT GROUPS API

```
/* create a new event group */
EventGroupHandle_t xEventGroupCreate(void);

/* wait for a bit or group of bits to become set */
EventBits_t xEventGroupWaitBits(
                    const EventGroupHandle_t xEventGroup,
                    const EventBits_t uxBitsToWaitFor,
                    const BaseType_t xClearOnExit,
                    const BaseType_t xWaitForAllBits,
                    TickType_t xTicksToWait);

/* set a bit or group of bits */
EventBits_t xEventGroupSetBits(
                    EventGroupHandle_t xEventGroup,
                    const EventBits_t uxBitsToSet);
```

# CASOS DE USO

- Avalie o uso desta API sempre que uma tarefa precisar esperar por mais de um evento do sistema.
- Usando semáforos e Queue Sets, podemos ter uma funcionalidade parecida com a oferecida por esta API.
- Mas além de ser mais leve, esta API possui algumas diferenças conceituais bem importantes:
  - Uma tarefa consegue ser notificada apenas quando um grupo de Event Bits forem setados.
  - Quando várias tarefas estiverem esperando no mesmo bit do Event Group, todas serão acordadas quando o evento acontecer (broadcast).

# SOFTWARE TIMER

- Um software timer é basicamente um timer que possibilita uma função ser executada em determinado tempo no futuro.
- A função executada pelo software timer é chamada de **função de callback**, e o tempo entre a inicialização do timer e a execução da função de callback é chamada de **período do timer**.
- Portanto, com o software timer você consegue configurar uma função de callback para ser executada quando um timer expirar.

# TIPOS DE SOFTWARE TIMERS

- **One-shot**: executa a função de callback apenas uma vez, mas pode ser reiniciado manualmente.

- **Auto-reload**: após a execução da função de callback, reinicia sua execução automaticamente. Ou seja, executa a função de callback periodicamente.

# SOFTWARE TIMER (cont.)

- A funcionalidade de software timer do FreeRTOS não faz parte do núcleo do kernel, e foi implementada de forma a não adicionar overhead de processamento à aplicação.
- Por este motivo, o FreeRTOS não usa o tick interrupt e não executa as funções de callback do timer em contexto de interrupção.
- Basicamente, a implementação de software timer do FreeRTOS atua como uma tarefa usando os recursos providos pelo FreeRTOS.
- Ela é composta por um conjunto de APIs que se comunicam com a tarefa de timer através de queues.

# HABILITANDO

- Para habilitar esta funcionalidade, adicione o arquivo `timers.c` ao seu projeto e configure as seguintes opções no arquivo de configuração `FreeRTOSConfig.h`:
  - `configUSE_TIMERS`: "1" para habilitar a funcionalidade de timer.
  - `configTIMER_TASK_PRIORITY`: prioridade da tarefa de timer.
  - `configTIMER_QUEUE_LENGTH`: tamanho do queue de comandos da tarefa de timer.
  - `configTIMER_TASK_STACK_DEPTH`: tamanho do stack da tarefa de timer.

# SOFTWARE TIMER API

```
/* creates a new software timer instance */
TimerHandle_t xTimerCreate(
                const char *const pcTimerName,
                const TickType_t xTimerPeriod,
                const UBaseType_t uxAutoReload,
                void *const pvTimerID,
                TimerCallbackFunction_t pxCallbackFunction);

/* deletes a software timer */
BaseType_t xTimerDelete(
                TimerHandle_t xTimer,
                TickType_t xBlockTime);
```

# SOFTWARE TIMER API (cont.)

```
/* starts a software timer */
BaseType_t xTimerStart(
                TimerHandle_t xTimer,
                TickType_t xBlockTime);

/* stops a software timer */
BaseType_t xTimerStop(
                TimerHandle_t xTimer,
                TickType_t xBlockTime);

/* restarts a software timer */
BaseType_t xTimerReset(
                TimerHandle_t xTimer,
                TickType_t xBlockTime);
```

# ALOCAÇÃO ESTÁTICA

- A partir da versão 9.0.0 do FreeRTOS é possível criar objetos (tarefas, semáforos, queues, etc) estaticamente, sem precisar utilizar alocação dinâmica de memória.
- Foram criadas variações das funções de criação de objetos, adicionando no final `Static()`, como por exemplo `xTaskCreateStatic()`.
- Estas funções recebem como parâmetro um ponteiro para um buffer que deverá ser alocado estaticamente pelo desenvolvedor e que será utilizado para armazenar o objeto.
- Para utilizar esta funcionalidade, é necessário configurar a opção `configSUPPORT_STATIC_ALLOCATION` com `1` no `FreeRTOSConfig.h`.

# API ALOCAÇÃO ESTÁTICA

```
#include "task.h"

/* create a new task statically and add it to the list
   of tasks that are ready to run */
TaskHandle_t xTaskCreateStatic
(
    TaskFunction_t pxTaskCode,
    const char *const pcName,
    unsigned uint32_t usStackDepth,
    void *const pvParameters,
    UBaseType_t uxPriority,
    StackType_t *const puxStackBuffer,
    StaticTask_t *const pxTaskBuffer
);
```

# API ALOCAÇÃO ESTÁTICA (cont.)

```
#include "queue.h"

/* create a new queue statically */
QueueHandle_t xQueueCreateStatic(
    UBaseType_t uxQueueLength,
    UBaseType_t uxItemSize,
    uint8_t *pucQueueStorageBuffer,
    StaticQueue_t *pxQueueBuffer
);
```

# API ALOCAÇÃO ESTÁTICA (cont.)

```
#include "semphr.h"

/* create a binary semaphore statically */
SemaphoreHandle_t xSemaphoreCreateBinaryStatic(
                        StaticSemaphore_t *pxSemaphoreBuffer);

/* create a mutex statically */
SemaphoreHandle_t xSemaphoreCreateMutexStatic(
                        StaticSemaphore_t *pxMutexBuffer);
```

# VANTAGENS

- A alocação estática permite um controle maior sobre o uso de memória e tem algumas vantagens, incluindo:
  - Controle sobre a região de memória que determinado objeto será alocado.
  - Consumo de RAM pode ser definido em tempo de compilação.
  - Não é necessário se preocupar com erros de alocação de memória.
  - Permite o uso do FreeRTOS em aplicações que não permitem alocação dinâmica de memória.

# DESVANTAGENS

- A alocação estática tem algumas desvantagens, incluindo:
  - É necessário alocar memória estaticamente para cada objeto que será criado.
  - É necessário passar mais parâmetros para as funções de criação de objetos do FreeRTOS.
  - Como não é possível desalocar uma região de memória alocada estaticamente, pode causar um consumo maior de RAM.

# CO-ROUTINES

- As co-routines são uma funcionalidade disponível a partir da versão 4.0.0 do FreeRTOS.
- Elas tem os mesmos conceitos de uma tarefa, mas com duas diferenças fundamentais:
  - Todas as co-routines de uma aplicação compartilham o mesmo stack, reduzindo drasticamente o uso de memória RAM. Por este motivo, existem algumas restrições no uso de RAM e da API do kernel em uma co-routine.
  - O escalonamento de co-routines é controlado pelo usuário, e realizado de forma cooperativa. É possível ter em uma mesma aplicação tarefas preemptivas e co-routines cooperativas.

# CO-ROUTINES (cont.)

- Tem o objetivo de ser usada em sistemas com pouquíssimos recursos de RAM, normalmente em microcontroladores de 8 bits, e por isso não é comum seu uso em arquiteturas de 32 bits.
- Co-routines só podem estar nos estados Running, Ready e Blocked.
- As prioridades de uma co-routine vão de 0 até `configMAX_CO_ROUTINE_PRIORITIES - 1`, definida no arquivo `FreeRTOSConfig.h`.

# CRIANDO UMA CO-ROUTINE

```
/* create a new co-routine and add it to the list of
co-routines that are ready to run */
BaseType_t xCoRoutineCreate(
                    crCOROUTINE_CODE pxCoRoutineCode,
                    UBaseType_t uxPriority,
                    UBaseType_t uxIndex);
```

# APLICAÇÃO COM CO-ROUTINES

```
#include "task.h"
#include "croutine.h"

void main(void)
{
    /* create a co-routine */
    xCoRoutineCreate(vFlashCoRoutine, PRIORITY_0, 0);

    // NOTE:  Tasks can also be created here!

    /* start the scheduler */
    vTaskStartScheduler();
}
```

# EXEMPLO DE UMA CO-ROUTINE

```
void vFlashCoRoutine(xCoRoutineHandle xHandle,
                     unsigned portBASE_TYPE uxIndex)
{
    /* co-routines must start with a call to crSTART() */
    crSTART(xHandle);

    for( ;; )
    {
        /* delay for a fixed period */
        crDELAY( xHandle, 10 );

        /* toggle a LED */
        vParTestToggleLED(0);
    }

    /* co-routines must end with a call to crEND() */
    crEND();
}
```

# ESCALONANDO CO-ROUTINES

```
/* schedule the co-routines in the idle task hook */
void vApplicationIdleHook(void)
{
    vCoRoutineSchedule(void);
}
```

Embedded **Labworks**

# FreeRTOS

## Integração com bibliotecas

# INTEGRAÇÃO COM BIBLIOTECAS

- Durante o desenvolvimento de um projeto com o FreeRTOS, diversas bibliotecas podem ser integradas à aplicação:
  - Bibliotecas disponibilizadas pelo fabricante do hardware.
  - Bibliotecas de código-aberto disponibilizadas pela comunidade.
  - Bibliotecas disponibilizadas pela Real Time Engineers e por seus parceiros (FreeRTOS Labs e FreeRTOS+).

# SELECIONANDO UMA BIBLIOTECA

- Alguns fatores devem ser levados em consideração ao selecionar uma biblioteca para integração com o FreeRTOS:
  - Thread-safe.
  - Consumo de recursos.
  - Determinismo.
  - Licença.
  - Acesso ao código-fonte.
  - Facilidade de adaptação e impacto na licença.

# FreeRTOS+

- Conjunto de bibliotecas e ferramentas disponibilizadas por parceiros para facilitar o desenvolvimento e a depuração de aplicações com o FreeRTOS.

  http://www.freertos.org/FreeRTOS-Plus/index.shtml

- A maioria dos projetos com código-fonte disponível em `FreeRTOS-Plus/Source/`.

- A maioria possui licença comercial (alguns são gratuitos para determinados chips/fabricantes).

# FreeRTOS+ (cont.)

- **Nabto**: solução de IoT para conexão remota pela Internet.
- **FAT SL**: sistema de arquivos FAT12, FAT16 e FAT32.
- **Embedded TCP/IP**: stack de rede TCP/IP (IPv4/IPv6).
- **Trace**: ferramenta de diagnóstico e depuração em tempo de execução.

# FreeRTOS+ (cont.)

- **UDP**: stack de rede UDP/IP (IPv4).
- **CLI**: interpretador de linha de comando.
- **CyaSSL**: biblioteca TLS/SSL.
- **IO**: implementação do padrão POSIX (`open()`, `read()`, `write()`, `ioctl()`, etc) para interfacear com drivers de dispositivo.

# FreeRTOS LABS

- Espaço reservado para bibliotecas e ferramentas ainda em fase de desenvolvimento, testes e consolidação.

  http://www.freertos.org/FreeRTOS-Labs/index.shtml

- No momento possui os projetos TCP (pilha de protocolos TCP/IP) e FAT (sistema de arquivos FAT).

- Assim que as bibliotecas deste projeto estiverem consolidadas e testadas, farão parte do FreeRTOS+.

# BIBLIOTECAS DA COMUNIDADE

- **uIP**: pilha de protocolos TCP/IP para microcontroladores de 8/16 bits (agora parte do projeto **Contiki**).
  https://github.com/adamdunkels/uip

- **lwIP**: pilha de protocolos TCP/IP para sistemas embarcados.
  http://savannah.nongnu.org/projects/lwip/

- **FatFs**: sistema de arquivos FAT para sistemas embarcados.
  http://elm-chan.org/fsw/ff/00index_e.html

# BIBLIOTECAS DE FABRICANTES

- A maioria dos fabricantes de hardware liberam junto com as ferramentas de desenvolvimento um conjunto de bibliotecas, incluindo drivers e pilhas de protocolo, para agregar valor à solução com o seu chip.
  - **NXP**: Kinetis SDK, LPCWare.
  - **Texas Instruments**: TivaWare Software.
  - **Atmel**: Atmel Software Framework (ASF).
  - **Microchip**: MPLAB Harmony Integrated Software Framework.
  - **ST**: STM32Cube.

# BIBLIOTECAS DE FABRICANTES (cont.)

- A grande vantagem do uso destas bibliotecas é a facilidade de integração com a plataforma de hardware do fabricante.
- A principal desvantagem é o desenvolvimento de um produto cujo software será dependente de uma plataforma ou fabricante de hardware.

# DICAS PARA A INTEGRAÇÃO

- Se a biblioteca não for thread-safe, crie funções para abstrair o acesso à biblioteca e proteja este acesso com um mutex.
- Se necessário, altere a biblioteca para utilizar as rotinas de alocação de memória e de delay do FreeRTOS.
- Implemente uma ou mais tarefas do FreeRTOS para paralelizar a execução das rotinas da biblioteca.
- Analise o código e mensure o consumo de CPU das tarefas que utilizam a biblioteca para identificar possíveis problemas de implementação (ex: polling).

Embedded Labworks

# FreeRTOS

## Ferramentas

# LISTA DE TAREFAS EM EXECUÇÃO

- O FreeRTOS provê a função `vTaskList()`, que retorna em tempo real uma string formatada com todas as tarefas em execução, o status das tarefas e o uso do stack.

```
void vTaskList(char *pcWriteBuffer);
```

- Para usá-la, basta habilitar as opções `configUSE_TRACE_FACILITY` e `configUSE_STATS_FORMATTING_FUNCTIONS` no arquivo `FreeRTOSConfig.h`.

- Esta rotina só deve ser usada para fins de teste e depuração, porque ela é lenta e desabilita todas as interrupções durante sua execução.

# LISTA DE TAREFAS EM EXECUÇÃO (cont.)

```
Name          State   Priority  Stack   Num
*********************************************
Print           R       4       331     29
Math7           R       0       417     7
Math8           R       0       407     8
QConsB2         R       0       53      14
QProdB5         R       0       52      17
QConsB4         R       0       53      16
SEM1            R       0       50      27
SEM1            R       0       50      28
IDLE            R       0       64      0
Math1           R       0       436     1
Math2           R       0       436     2
Math3           R       0       417     3
Math4           R       0       407     4
Math5           R       0       436     5
Math6           R       0       436     6
QProdNB         B       2       52      12
LEDx            B       1       63      19
LEDx            B       1       74      22
LEDx            B       1       73      20
LEDx            B       1       63      25
LEDx            B       1       73      21
LEDx            B       1       63      24
COMTx           B       2       44      9
LEDx            B       1       63      26
LEDx            B       1       67      23
SUICIDE1        B       3       215     55
SUICIDE2        B       3       215     56
SUICIDE1        B       3       215     57
SUICIDE2        B       3       215     58
CREATOR         B       3       170     30
QProdB1         B       3       53      13
QConsB6         B       0       52      18
QProdB3         B       3       54      15
COMRx           B       3       51      10
QConsNB         B       2       57      11
```

# ESTATÍSTICAS DE EXECUÇÃO

- O FreeRTOS tem a capacidade de calcular e armazenar o tempo de processamento alocado para cada tarefa.
- A função `vTaskGetRunTimeStats()` pode ser usada para formatar estas informações para serem melhor visualizadas.

```
void vTaskGetRunTimeStats(char *pcWriteBuffer);
```

- Para cada tarefa, dois valores são apresentados:
  - **Abs Time**: Tempo total de execução da tarefa.
  - **% Time**: Porcentagem de execução da tarefa comparada ao tempo total disponível.

# RUNTIME STATISTICS

## Run-time statistics

Page will refresh every 2 seconds.

| Task | Abs Time | % Time |
|---|---|---|
| uIP | 12050 | <1% |
| IDLE | 587724 | 24% |
| QProdB2 | 2172 | <1% |
| QProdB3 | 10002 | <1% |
| QProdB5 | 11504 | <1% |
| QConsB6 | 11671 | <1% |
| PolSEM1 | 60033 | 2% |
| PolSEM2 | 59957 | 2% |
| IntMath | 349246 | 14% |
| MuLow | 36619 | 1% |
| GenQ | 579715 | 24% |
| PeekL | 146 | <1% |
| Rec3 | 582497 | 24% |

# HABILITANDO

- Para usar esta funcionalidade é necessário:
  - Habilitar as opções `configGENERATE_RUN_TIME_STATS` e `configUSE_STATS_FORMATTING_FUNCTIONS`.
  - Definir a macro `portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()`, que deverá configurar o timer que será usado para gerar as estatísticas. Para que as estatísticas sejam mais precisas, este timer deve ser de 10 a 100 vezes mais rápido que a interrupção do tick.
  - Definir a macro `portGET_RUN_TIME_COUNTER_VALUE()`, que deverá retornar a leitura atual do timer.
- Esta rotina só deve ser usada para fins de teste e depuração, porque ela é lenta e desabilita todas as interrupções durante sua execução.

# uxTaskGetSystemState

- As funções `vTaskList()` e `vTaskGetRunTimeStats()` usam internamente a função `uxTaskGetSystemState()` para coletar informações sobre as tarefas do sistema e formatar uma string para o usuário.
- Para usar a função `uxTaskGetSystemState()`, basta habilitar a opção `configUSE_TRACE_FACILITY` no arquivo `FreeRTOSConfig.h`.
- Informações sobre cada tarefa serão retornadas em uma estrutura do tipo `TaskStatus_t`.
- Esta rotina só deve ser usada para fins de teste e depuração, porque ela desabilita o escalonador durante sua execução.

# uxTaskGetSystemState (cont)

```
#include "task.h"

/* returns the state of the system, populating a
   TaskStatus_t structure for each task */
UBaseType_t uxTaskGetSystemState(
    TaskStatus_t *const pxTaskStatusArray,
    const UBaseType_t uxArraySize,
    unsigned long *const pulTotalRunTime);
);
```

# uxTaskGetSystemState (cont)

```
typedef struct xTASK_STATUS
{
    TaskHandle_t xHandle;
    const signed char *pcTaskName;
    UBaseType_t xTaskNumber;
    eTaskState eCurrentState;
    UBaseType_t uxCurrentPriority;
    UBaseType_t uxBasePriority;
    unsigned long ulRunTimeCounter;
    unsigned short usStackHighWaterMark;
} TaskStatus_t;
```

Embedded **Labworks**

# LABORATÓRIO

## Exibindo estatísticas da aplicação

# TRACING

- Em engenharia de software, tracing é a técnica usada para armazenar informações sobre a execução de um programa.
- Estas informações são normalmente usadas para entender o fluxo de execução de uma aplicação, e para diagnosticar e corrigir problemas.
- O FreeRTOS possui alguns mecanismos de tracing, de forma que possamos analisar o fluxo de execução da aplicação.

# LEGACY TRACE UTILITY

- As versões do FreeRTOS anteriores à V7.1.0 possuíam uma implementação de tracing, que armazenava a sequência e o tempo de execução de cada uma das tarefas.
- Para usar esta funcionalidade, bastava habilitar a opção `configUSE_TRACE_FACILITY` no arquivo de configuração `FreeRTOSConfig.h`.
- E depois usar as funções `vTaskStartTrace()` e `ulTaskEndTrace()` para iniciar e finalizar o tracing.

# LEGACY TRACE UTILITY (cont.)

```
#include "task.h"

/* starts a kernel activity trace */
void vTaskStartTrace(
          char *pcBuffer,
          unsigned long ulBufferSize);

/* stops a kernel activity trace */
unsigned long ulTaskEndTrace(void);
```

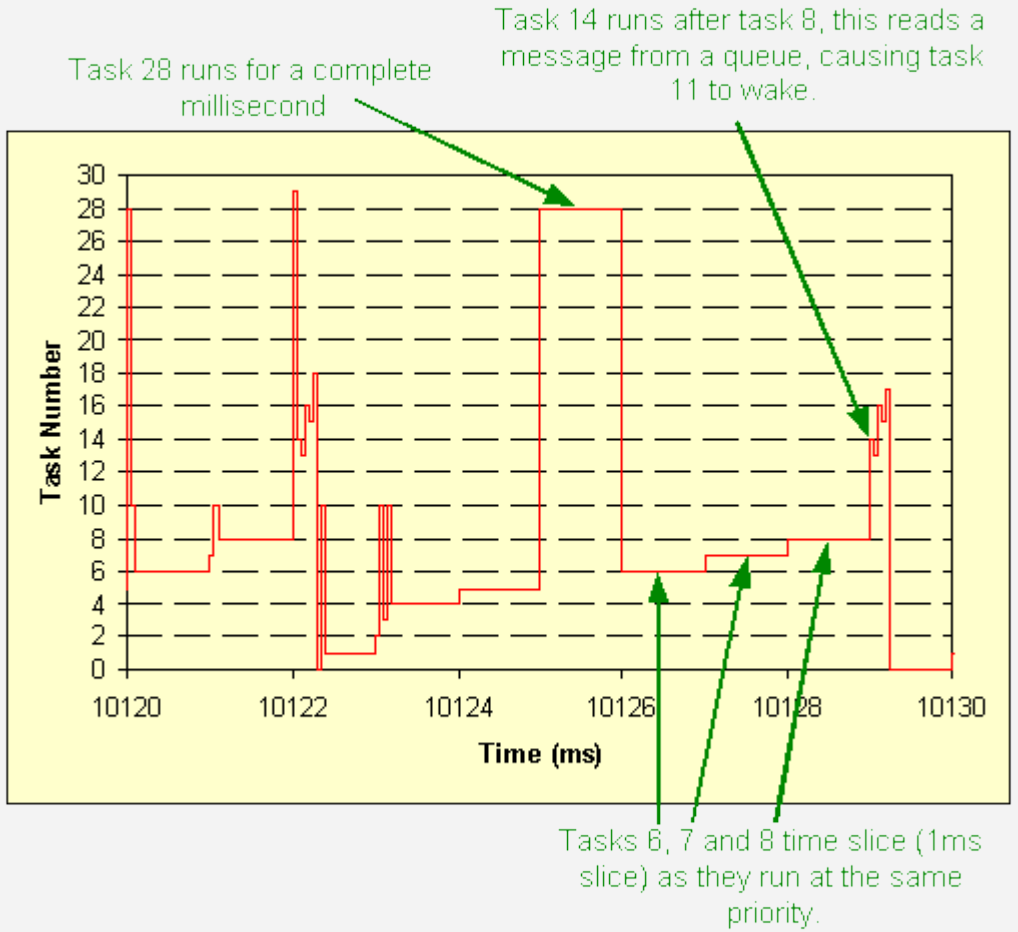
# LEGACY TRACE UTILITY (cont.)

- As informações do trace são armazenadas em um buffer, e o trace é finalizado se não houver espaço suficiente no buffer para armazenar mais informações.

- Depois é possível transferir este buffer através de um meio de comunicação (RS232, Ethernet, USB) ou de uma mídia removível (cartão SD, pendrive, etc).

- Existe uma ferramenta para DOS/Windows chamada `tracecon.exe` que converte este buffer em um arquivo com campos separados por TAB, e que pode ser aberto por qualquer programa de planilhas para gerar relatórios do fluxo de execução da aplicação.

# LEGACY TRACE UTILITY (cont.)




Fonte: Site *freertos.org*

# TRACE HOOK MACROS

- A partir da versão 7.1.0, as macros de tracing são a implementação padrão do FreeRTOS para realizar tracing em aplicações.
- Esta funcionalidade é composta por um conjunto de macros que permitem coletar dados sobre o funcionamento da aplicação.
- Alguns pontos-chave no código-fonte do FreeRTOS chamam macros (vazias por padrão), mas que podem ser definidas pela aplicação para prover funcionalidades de tracing para o desenvolvedor.
- Com estas macros é possível capturar as trocas de contexto, medir os tempos de execução de cada tarefa, fazer log de eventos do kernel, realizar integração com ferramentas de depuração, etc.

# ALGUMAS MACROS DE TRACING

- Macro chamada durante a interrupção de tick:

```
traceTASK_INCREMENT_TICK(xTickCount)
```

- Macro chamada antes de uma tarefa ser selecionada para execução:

```
traceTASK_SWITCHED_OUT()
```

- Macro chamada logo após uma tarefa ser selecionada para execução:

```
traceTASK_SWITCHED_IN()
```

# ALGUMAS MACROS DE TRACING (cont.)

- Macro chamada se der erro ao criar um queue:

```
traceQUEUE_CREATE_FAILED()
```

- Macro chamada ao enviar um item para o queue:

```
traceQUEUE_SEND(pxQueue)
```

# EXEMPLO 1

```
void vTask1(void *pvParameters)
{
    vTaskSetApplicationTaskTag(NULL, (void *) 1);

    for(;;)
    {
    }
}

void vTask1(void *pvParameters)
{
    vTaskSetApplicationTaskTag(NULL, (void *) 2);

    for(;;)
    {
    }
}

#define traceTASK_SWITCHED_IN() \
                    vSetAnalogueOutput(0, (int)pxCurrentTCB->pxTaskTag)
```

# EXEMPLO 2

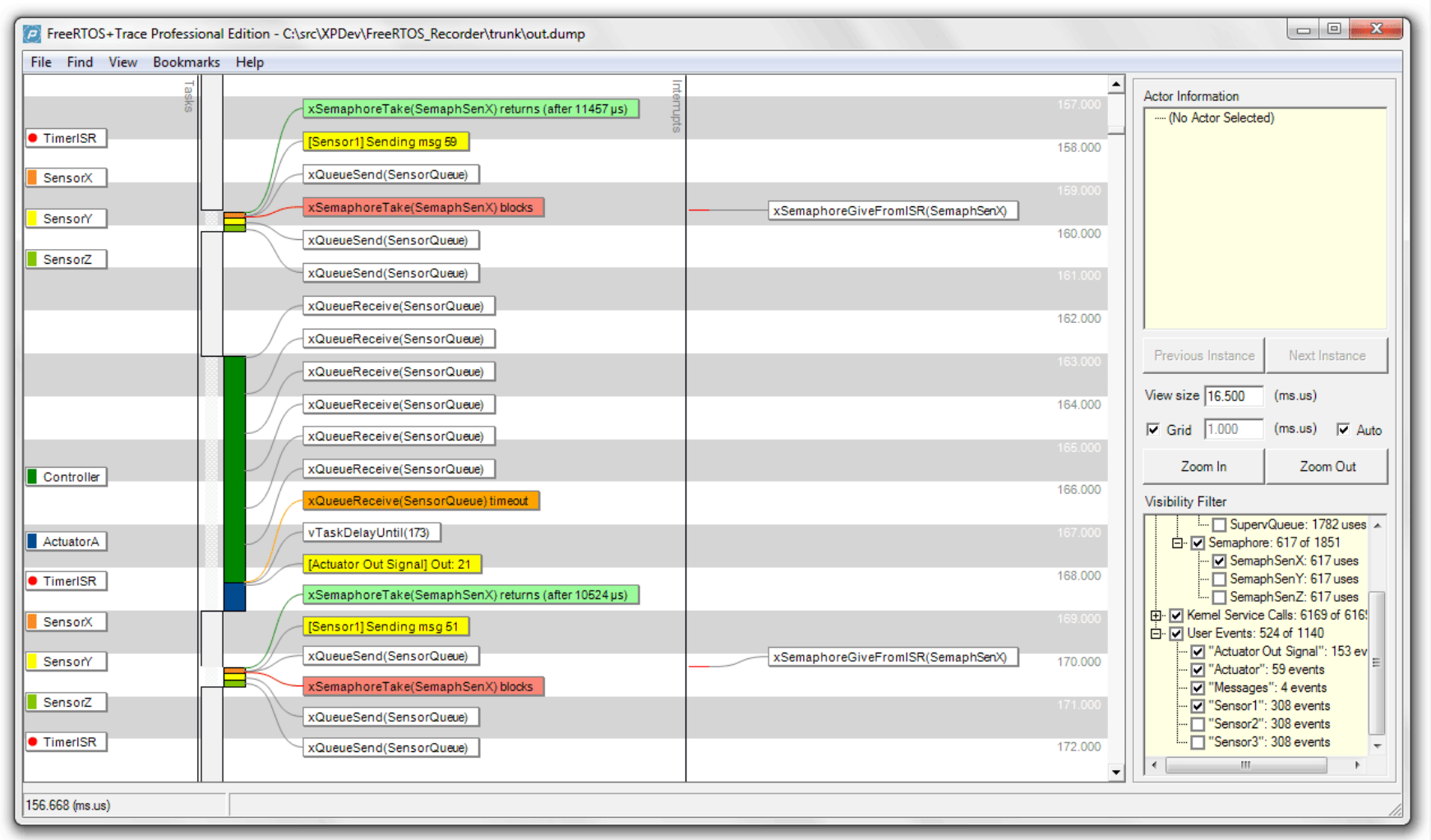
```
#define traceBLOCKING_ON_QUEUE_RECEIVE(pxQueue) \
    ulSwitchReason = reasonBLOCKING_ON_QUEUE_READ;

#define traceBLOCKING_ON_QUEUE_SEND(pxQueue) \
    ulSwitchReason = reasonBLOCKING_ON_QUEUE_SEND;

...

#define traceTASK_SWITCHED_OUT() \
    log_event(pxCurrentTCB, ulSwitchReason);
```

# MACROS DE TRACING

- Existem macros para quase todas as chamadas de função do FreeRTOS, possibilitando criar um sistema de tracing bem completo.
- A documentação de todas as macros de tracing está disponível no site do FreeRTOS.

  http://www.freertos.org/rtos-trace-macros.html
- O FreeRTOS+Trace da Percepio é uma ferramenta de tracing que implementa estas macros do FreeRTOS, e que se comunica com uma ferramenta no PC para visualização dos dados coletados.

# FreeRTOS+TRACE



# STATEVIEWER PLUGIN

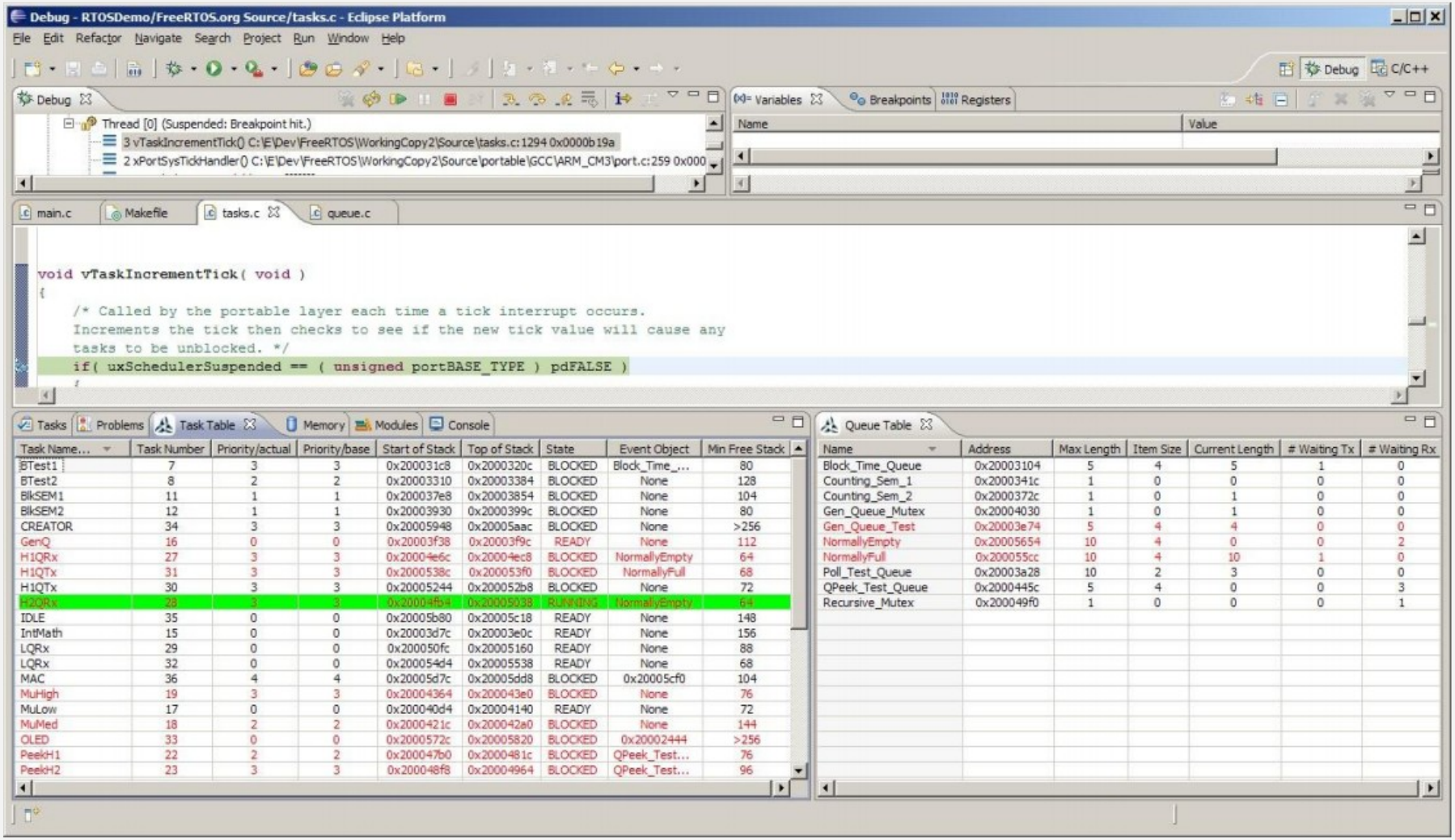
- O **STATEVIEWER** é um plugin para Eclipse e IAR que provê um ambiente de depuração para o FreeRTOS (kernel aware).

  https://www.highintegritysystems.com/tools/stateviewer/

- É desenvolvido e disponibilizado gratuitamente pela WITTENSTEIN High Integrity Systems, mesma empresa responsável pelo OpenRTOS e pelo SafeRTOS.

- Provê um status atual das tarefas em execução, recursos em uso (semáforos, queues, etc), consumo do stack por cada tarefa, etc.

# STATEVIEWER NO ECLIPSE



# SIMULADORES

- **FreeRTOS Windows Simulator** é um porte do FreeRTOS para rodar no Windows, funciona no Visual Studio 2010 e no Eclipse, e está disponível a partir do FreeRTOS V6.1.1.

- **Posix/Linux Simulator for FreeRTOS** é um porte do FreeRTOS para rodar em máquinas com sistemas operacionais GNU/Linux, funciona no Eclipse e é disponibilizado separadamente da versão oficial do FreeRTOS.

Embedded Labworks

# FreeRTOS

## Projetando com o FreeRTOS

# DEFININDO TAREFAS

- É melhor ter mais ou menos tarefas?
- Vantagens de ter mais tarefas:
  - Maior controle sobre os tempos de resposta das diferentes partes do sistema.
  - Aplicação mais modular, facilitando o desenvolvimento e testes.
  - Código mais limpo, facilitando a manutenção.

# DEFININDO TAREFAS (cont.)

- Desvantagens de ter mais tarefas:
  - Aumenta o compartilhamento de dados e o uso da API do kernel para sincronismo e troca de informações entre as tarefas.
  - Exige mais memória RAM, já que cada tarefa requer um stack.
  - Consome mais tempo de CPU para realizar a troca de contexto.

# DIVIDINDO EM TAREFAS

- Pense e divida as funções do seu sistema em tarefas.
- Atividades que podem ser executadas em paralelo devem ser implementadas através de uma tarefa.
- Funções com prioridades diferentes exigem tarefas diferentes.

# DIVIDINDO EM TAREFAS (cont.)

- Funções periódicas devem ser implementadas em uma tarefa.
- Implemente uma tarefa para cada dispositivo de hardware compartilhado entre as diversas funções do sistema.
- Interrupções devem sincronizar ou transferir dados para tarefas, portanto é bem provável que cada interrupção que gere um evento no sistema tenha uma ou mais tarefas associadas à ela.

# DEFININDO PRIORIDADES

- É relativamente fácil atribuir prioridades às tarefas em sistemas mais simples. Normalmente está bem evidente que, por exemplo, determinada tarefa de controle tem mais prioridade que uma tarefa que gerencia a interface com o usuário.
- Mas na maioria das vezes, não é tão simples assim, devido à complexidade de sistemas de tempo real.
- Como regra geral, tarefas hard real-time devem ter maior prioridade sobre tarefas soft real-time.
- Mas outras características como frequência de execução e uso da CPU devem ser levadas em consideração.

# RATE MONOTONIC SCHEDULING

- Uma técnica interessante chamada Rate Monotonic Scheduling (RMS) atribui prioridades às tarefas de acordo com sua frequência de execução.

- Com o RMS, quanto maior a frequência de execução de uma tarefa, maior sua prioridade.

# FÓRMULA RMS

- Dado um conjunto de tarefas que receberam prioridades de acordo com a técnica RMS, é garantido que o deadline de todas as tarefas serão atingidos se a seguinte equação for verdadeira:

$$\sum_i \frac{E_i}{T_i} \leq n\left(2^{1/n} - 1\right)$$

Ei é o tempo máximo de execução da tarefa i
Ti é a frequência de execução da tarefa i
Ei/Ti é a fração da CPU necessária para executar a tarefa i

# TABELA RMS

| Number of Tasks | $n(2^{1/n}-1)$ |
|---|---|
| 1 | 1.00 |
| 2 | 0.828 |
| 3 | 0.779 |
| 4 | 0.756 |
| 5 | 0.743 |
| : | : |
| : | : |
| : | : |
| Infinite | 0.693 |

# DEADLINES COM O RMS

- Portanto, segundo a técnica RMS, para atingir o deadline das tarefas, a soma do uso da CPU de todas as tarefas precisa ser menor que 69,3%.
- Esta técnica possui algumas restrições, como por exemplo as tarefas não devem se comunicar entre si.
- De qualquer forma, pode ser um ponto de partida para definir as prioridades das tarefas do sistema.

# OUTRAS TÉCNICAS

- Existem algumas outras técnicas comuns para atribuir prioridades às tarefas, dentre elas:
  - Tarefas com processamento intensivo devem ter menor prioridade, para evitar o uso excessivo da CPU (starving).
  - Tarefas periódicas que rodam em unidades de milisegundos devem ser executadas em uma tarefa com prioridade maior.
  - Tarefas periódicas que rodam em unidades de microsegundos para menos devem ser executadas em uma ISR.
  - Rotinas que manipulam interface com o usuário devem ser executadas na ordem de centenas de milisegundos em uma prioridade que deve refletir seu deadline.

# TAMANHO DO STACK

- O tamanho do stack de cada tarefa depende da aplicação.
- Ao definir o tamanho do stack, é necessário levar em consideração todas as chamadas de função, variáveis locais e contexto da CPU para as rotinas de interrupção.
- É possível calcular o stack de uma tarefa manualmente, mas pode ser algo trabalhoso e sujeito à erros.

# TAMANHO DO STACK (cont.)

- Na prática, defina um tamanho padrão para o stack de cada tarefa.
- E durante o desenvolvimento e testes da aplicação, monitore o stack e ajuste-o de acordo com sua utilização.
- Como regra geral, evite escrever funções recursivas.

# STACK OVERFLOW

- Stack overflow é uma das principais causas de problemas encontrados no FreeRTOS, principalmente por novos usuários.
- Existem algumas técnicas que podem ser usadas para monitorar o uso do stack e detectar stack overflow.
- É possível monitorar o stack de uma tarefa com a função `uxTaskGetStackHighWaterMark()`.
- O FreeRTOS provê dois mecanismos para monitorar o stack das tarefas em tempo de execução, habilitando a opção *configCHECK_FOR_STACK_OVERFLOW*.

# PRINTF E STACK OVERFLOW

- O uso do stack pode ficar muito maior quando funções da biblioteca C padrão são usadas, especialmente as que manipulam I/O e string como a família de funções `printf()`.

- O FreeRTOS possui uma versão simples e eficiente do `sprintf()` no arquivo `printf-stdarg.c`, que pode ser usada pela aplicação, e que consome muito menos stack que a implementação padrão da biblioteca C.

# COMUNICAÇÃO ENTRE TAREFAS

- Use semáforos para sincronizar (notificar) tarefas.
- Quando, além de sincronizar, é necessário trocar dados, use queues.
- Sempre proteja o acesso à regiões críticas.
- Crie tarefas gatekeeper para compartilhar acesso à recursos de hardware.

# INTERRUPÇÕES

- Uma interrupção nunca deve usar uma função do RTOS que bloqueia.
- Use sempre funções do FreeRTOS que terminam com `fromISR()`.
- Implemente rotinas de tratamento de interrupção curtas.
- Sempre que possível, transfira o trabalho para uma tarefa, usando semáforos ou queues.

# INTERRUPÇÕES (cont.)

- Não use funções da API do FreeRTOS em interrupções cuja prioridade estão acima de `configMAX_SYSCALL_INTERRUPT_PRIORITY`.

- Só habilite as interrupções que usam funções do kernel após iniciar o escalonador de tarefas com a função `vTaskStartScheduler()`.

# HEAP

- É no heap que o kernel aloca memória para armazenar os dados das tarefas, dos queues e dos semáforos usados na aplicação.
- Conforme evoluimos na implementação da aplicação, precisamos de um espaço maior no heap. E se faltar espaço no heap, o kernel não conseguirá alocar memória para criar os objetos requisitados.
- Por isso, verifique sempre o retorno das funções de criação de tarefas, queues e semáforos para identificar problemas de alocação de memória.
- Monitore também erros de alocação de memória através da função de callback `vApplicationMallocFailedHook()`.

# ESCALONADOR

- Os seguintes modos de escalonamento de processos estão disponíveis no FreeRTOS:
  - Colaborativo.
  - Preemptivo sem time slice.
  - Preemptivo.

# ESCALONADOR E TROCA DE CONTEXTO

| | Bloquear ou liberar CPU | Retorno de uma API do FreeRTOS | Fim do time slice | Retorno de uma ISR |
|---|---|---|---|---|
| Colaborativo | SIM | NÃO | NÃO | SIM |
| Preemptivo sem time slice | SIM | SIM | NÃO | SIM |
| Preemptivo | SIM | SIM | SIM | SIM |

# QUAL USAR?

- É mais fácil desenvolver e garantir tempos de resposta no modo preemptivo, com a desvantagem de aumentar o uso de CPU devido às trocas de contexto.
- Já no modo colaborativo o overhead de CPU é menor, porém impactamos negativamente os tempos de resposta da aplicação.
- Sistemas hard real-time tendem a usar o escalonador no modo preemptivo.
- Sistemas mais simples, com características de soft real-time ou com foco maior em processamento, podem avaliar o uso do escalonador no modo colaborativo ou preemptivo com time slice desabilitado.

Embedded **Labworks**

# LABORATÓRIO

## Projeto final

Embedded Labworks

# FreeRTOS

## E agora?

# RECURSOS ONLINE

- Site do projeto:
  http://freertos.org
- Forum do projeto:
  http://sourceforge.net/p/freertos/discussion/
- Blog do Sergio Prado:
  http://sergioprado.org

# OUTROS RTOS'S OPEN SOURCE

- BRTOS:
  http://code.google.com/p/brtos/
- BeRTOS:
  http://www.bertos.org/
- eCos:
  http://ecos.sourceware.org/

# OUTROS RTOS'S OPEN SOURCE (cont.)

- RTEMS Real Time Operating System:
  http://www.rtems.com/

- CooCox CoOS (ARM Cortex-M):
  http://www.coocox.org/CoOS.htm

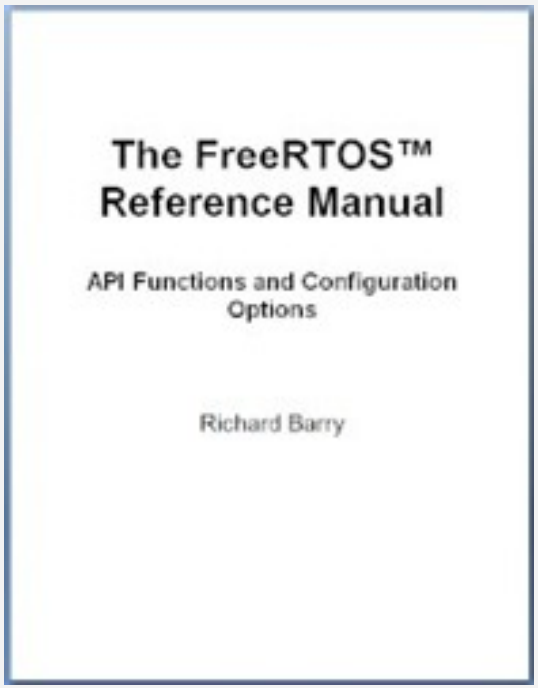
# ALGUNS RTOS'S COMERCIAIS

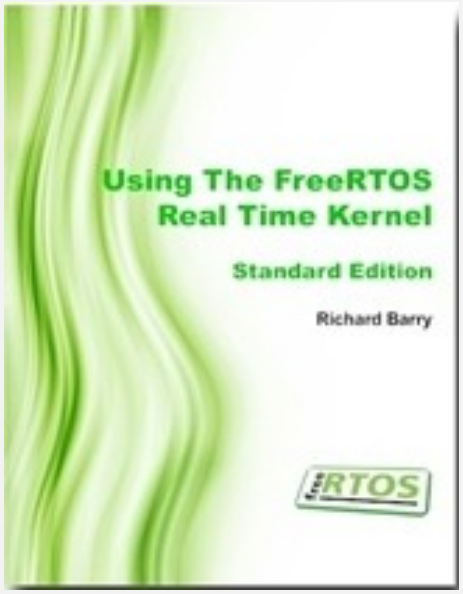
- UC/OS-III (Micrium):
  http://micrium.com/page/products/rtos/os-iii
- MQX (NXP):
  http://nxp.com/mqx
- TI-RTOS (Texas Instruments):
  http://www.ti.com/tool/ti-rtos
- RTX (Keil):
  http://www.keil.com/rtos/

# ALGUNS RTOS'S COMERCIAIS (cont.)

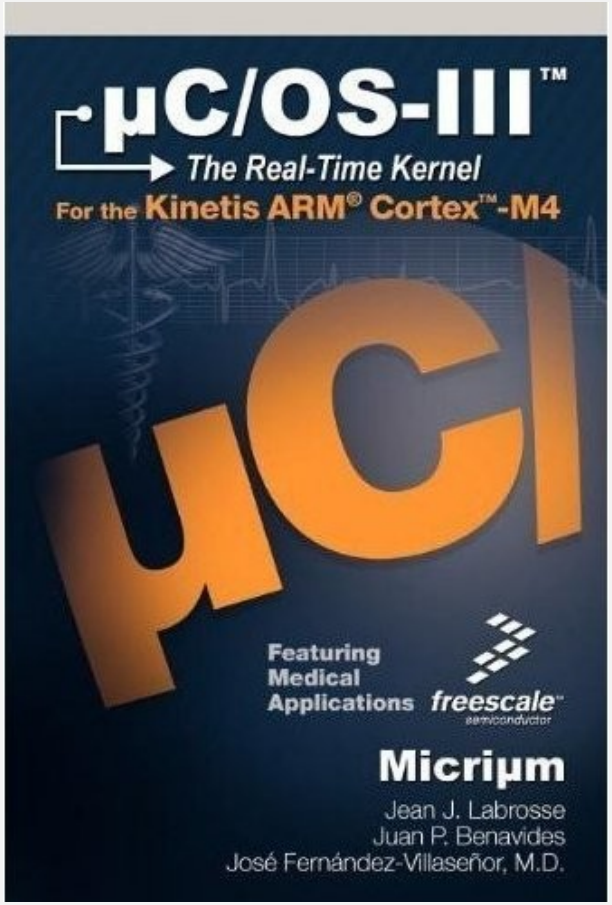
- ThreadX (Express Logic):
  http://rtos.com/products/threadx/
- Nucleus OS (Mentor Graphics):
  http://www.mentor.com/embedded-software/nucleus/
- VxWorks (Wind River):
  http://www.windriver.com/products/vxworks/

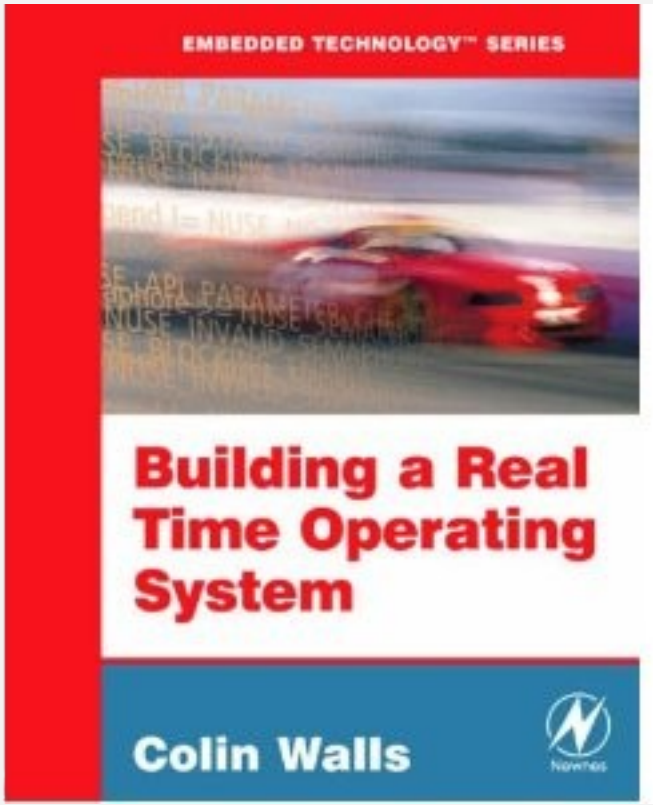
# LIVROS FREERTOS



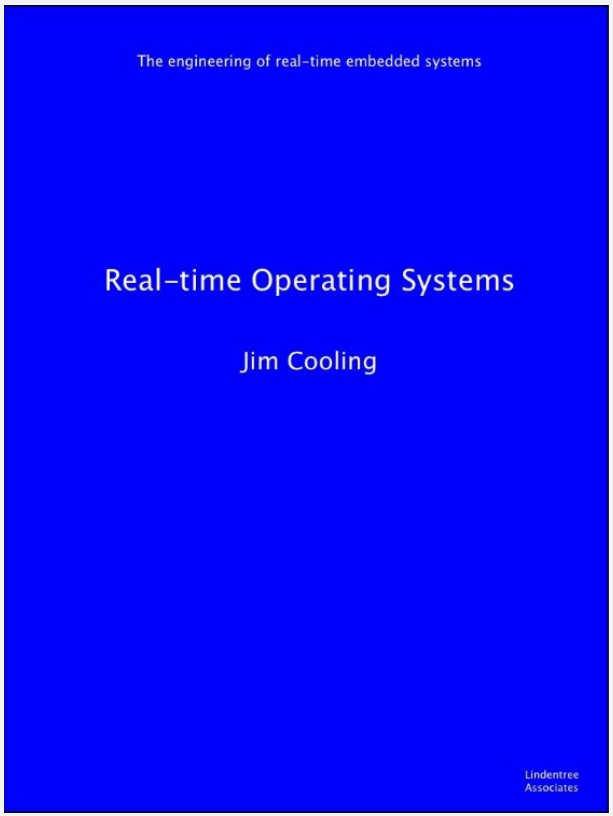



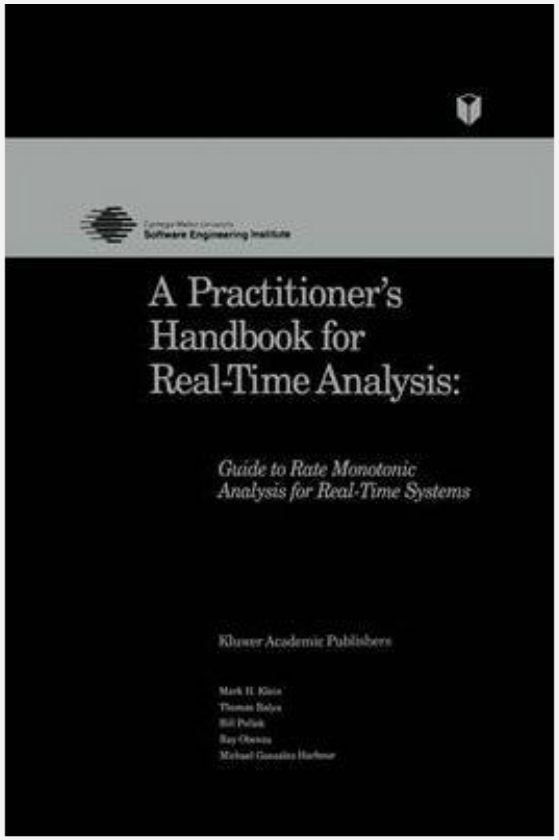
# OUTROS LIVROS



# OUTROS LIVROS (cont.)



# OBRIGADO!

E-mail sergio.prado@e-labworks.com
Website http://e-labworks.com

Embedded **Labworks**

Por Sergio Prado. São Paulo, Julho de 2012