

# Introdução ao FreeRTOS

Disciplina: Projeto de Circuitos Reconfiguráveis

Regina Marcela Ivo

Baseado no material de Sergio Pertuz Mendez

Prof. Daniel Muñoz

# O que é tempo real?

- “Um sistema de tempo-real é aquele em que a corretude das computações não depende somente da corretude lógica mas também no tempo em que o resultado é produzido. Se as restrições de tempo não são atendidas, é considerada uma falha do sistema.”

# Tempo Real

- **Sistema computacional em tempo real:**

Deve reagir a estímulos do ambiente controlado com um intervalo ditado pelo ambiente;

- **Deadline**

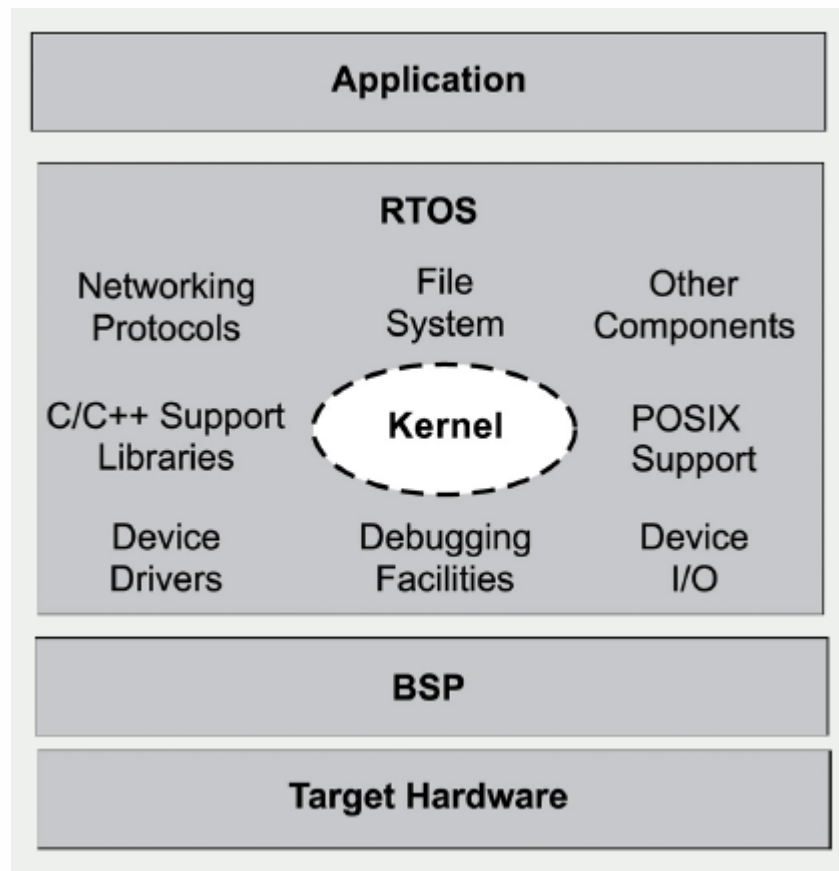
Instante de tempo em que um resultado deve ser produzido;

- ***Soft real-time:*** tolerância no deadline;
- ***Hard real-time:*** não há tolerância para o atraso.

# Preciso de um Sistema em Tempo Real?

1. Meus requisitos incluem *deadlines*? Minha aplicação seria válida caso complete as tarefas de maneira atrasada?
2. Estes atrasos causariam algum risco de segurança a alguém? Haveria alguma degradação no serviço?
3. Algum dispositivo externo que interage com minha aplicação exige algum requisito relacionado ao tempo de resposta?
4. Há alguma parte da minha aplicação que deveria ter prioridade de tarefas acima dos serviços padrão do próprio SO (rede, sistema de arquivos, etc)?
5. Preciso de alta precisão para controlar os *delays* e os *timeouts*?

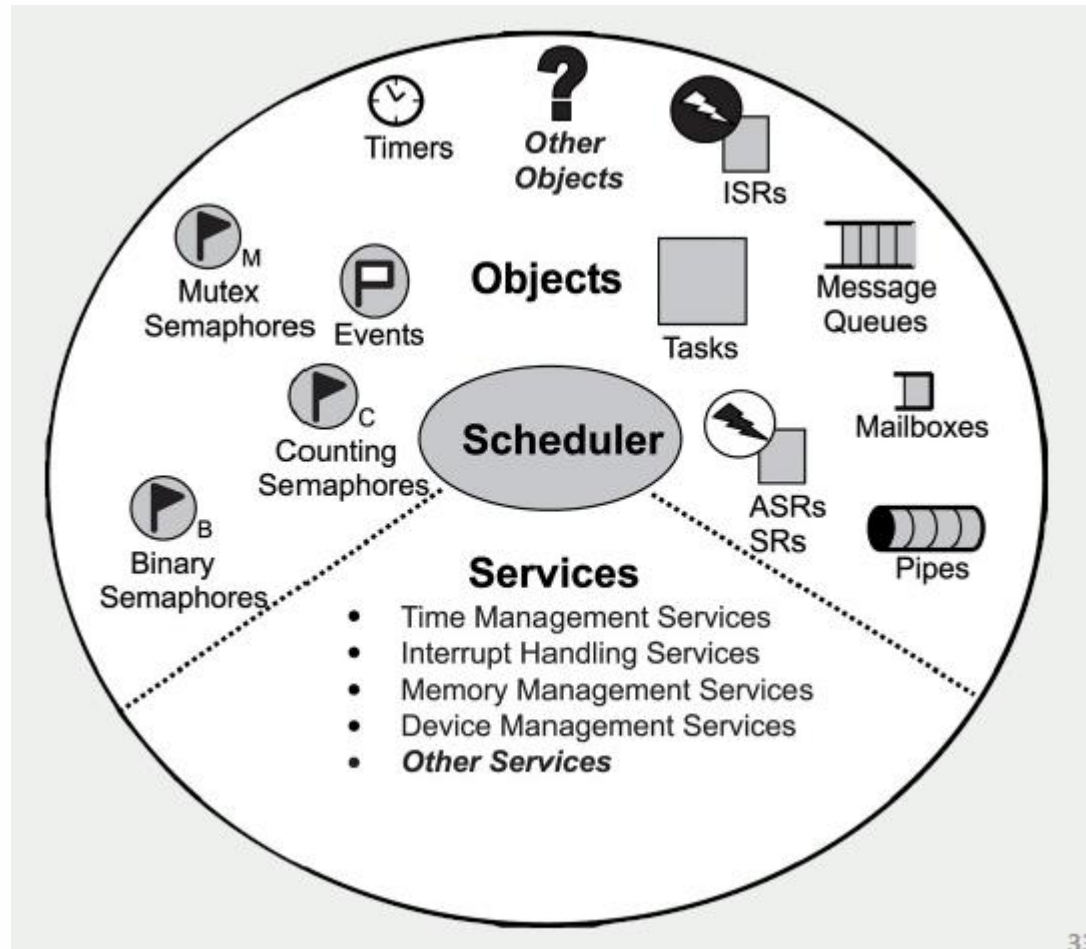
# Sistemas Operacionais em Tempo Real



Descrição  
de uma  
aplicação  
embarcada

# Sistemas Operacionais em Tempo Real

Principais  
Componentes  
de um Kernel  
RTOS



# Sistemas Operacionais de Tempo Real - RTOS

- Sistemas operacionais dedicados à aplicações que exigem máxima confiabilidade no tempo de execução de determinada tarefa a partir da ocorrência de um evento específico.



# O que é o FreeRTOS?

- É um kernel de tempo real, sobre o qual é possível construir aplicações que satisfaçam à requisitos críticos de tempo real.
- Idealmente adequado para aplicações embarcadas que exigem execução em tempo real e que utilizam microcontroladores ou pequenos microprocessadores.

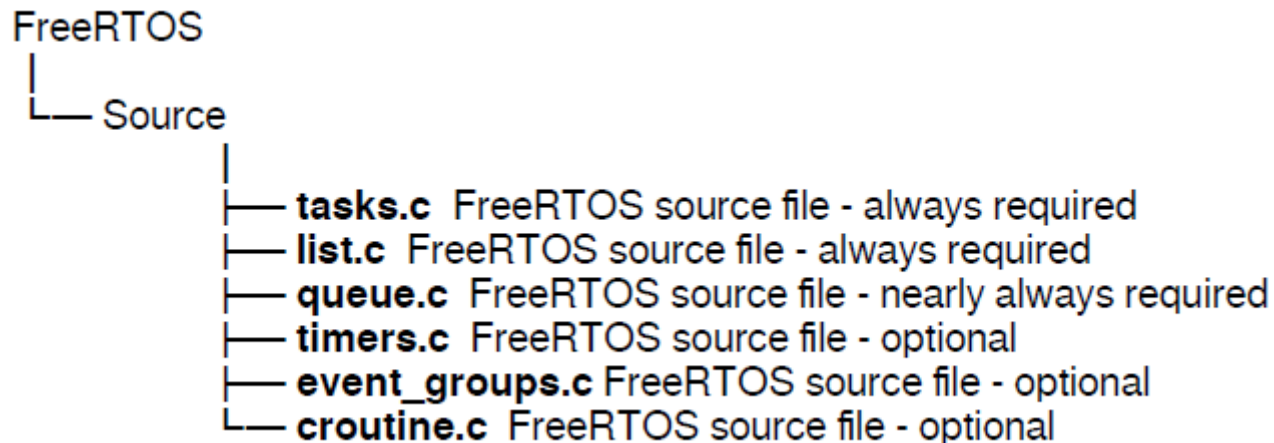


# Caraterísticas do FreeRTOS

- FreeRTOS é um “Sistema Operacional Embarcado” para micro controladores embarcados
- Capacidade multitarefa baseado em prioridades
- Tem um agendador Simples.
- Semáforos para gerenciar o compartilhamento de recursos entre várias tarefas.

O FreeRTOS é fornecido como um conjunto de arquivos em linguagem C. Alguns dos arquivos de origem são comuns a todas as portas, enquanto outros são específicos de uma porta.

**FreeRTOSConfig.h:** configura FreeRTOS.



# Compilando FreeRTOS

- **tasks.c** e **list.c**: são o código-fonte do FreeRTOS comum a todas as portas do FreeRTOS e eles estão localizados diretamente no diretório FreeRTOS / Source

Além desses dois arquivos, os seguintes arquivos fontes estão localizados no mesmo diretório:

**queue.c** e **timer.c**

- **queue.c** fornece serviços de fila e semáforo. Este arquivo é quase sempre necessário.
- **timers.c** fornece a funcionalidade do temporizador de software. Ele só precisa ser incluído no build se os timers do software forem realmente usados.

# Repositório do FreeRTOS

- Um Board Support Package (BSP) é uma biblioteca gerada pelo Xilinx SDK específica de um projeto de hardware.
- Ele contém código de inicialização para exibir os processadores ARM no ZYNQ e também contém drivers de software para todos os periféricos ZYNQ disponíveis.
- A porta FreeRTOS estende o BSP autônomo para incluir também arquivos fonte do FreeRTOS.
- Depois de usar essa porta no ambiente do Xilinx SDK, o usuário obtém todos os arquivos fonte do FreeRTOS em uma biblioteca do FreeRTOS BSP.
- Essa biblioteca usa a biblioteca BSP autônoma gerada pelo Xilinx SDK.

# Arquivos de Cabeçalho

Um projeto que usa o API do FreeRTOS deve incluir "FreeRTOS.h", seguido pelo arquivo de cabeçalho que contém o protótipo da função da API que está sendo usada.

- ❖ "Task.h",
- ❖ "queue.h"
- ❖ "semphr.h"
- ❖ "timers.h"
- ❖ "event\_groups.h".

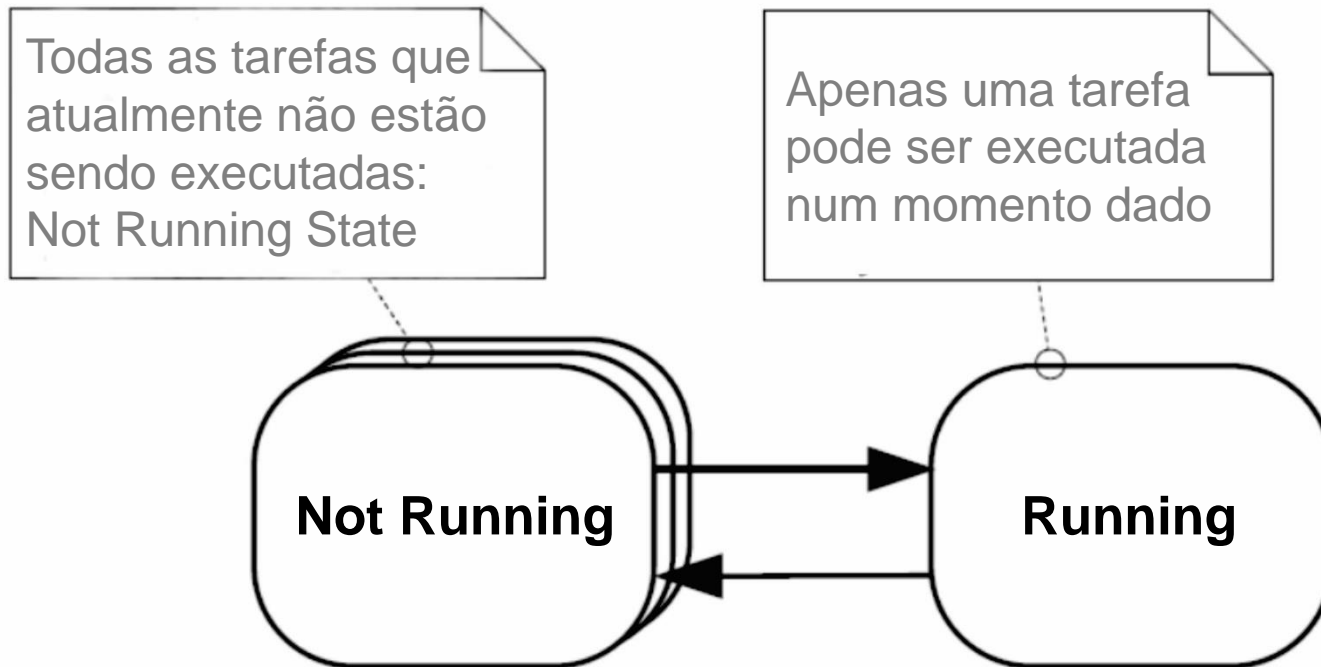
# Task

- É uma função C simples
- Um ponteiro para parâmetros (void \*) como entrada
- Cria um loop infinito ( while (1) )
- As tarefas são controladas pelo Agendador (função interna do freeRTOS)

Cada tarefa tem seu próprio stack:

- Cada variável que você declara ou aloca em memória usa a memória no stack.
- O tamanho do stack de uma tarefa depende da memória consumida pelas variáveis locais e da profundidade da chamada de função.

# Estados de Tarefas de Top Level



# Criando uma Task

- A função de task:

```
void ATaskFunction( void *pvParameters)
{
    // do initilisation
    while (1)
    {
        // Task execution code
    }
}
```

- Criação de uma task (no arquivo main.c)

```
portBASE_TYPE xTaskCreate(
    pdTASK_CODE    pvTaskCode,    // pointer to the Task
    Char*          pcName,        // String: name of Task for debug
    unsigned short usStackSize,   // Stacksize
    Void*          pvParameters,  // pointer to Parameters
    unsigned short uxPriority,     // Priority
    XtaskHandle*   pxCreatedTask); // Pointer to receive Task handle
```

Return **pdPASS** or **pdFAIL** (when insufficient heap memory)



# Exemplo

```
void hello_world_task (void* p)
{
    while(1)
    {
        Printf(" Hello World!");
        vTaskDelay(1000);
    }
}

void main(void )
{
    XtaskCreate (hello_world_task, "TestTask", 512, NULL, 1, NULL);
    vTaskStartScheduler();
    //  never comes here
}
```

- A função main em FreeRTOS é quem cria as tarefas.
- FreeRTOS permite multitarefas baseado nas suas tarefas e nível de prioridade.

# Tipos de Dados

Dois tipos de dados específicos para portas:

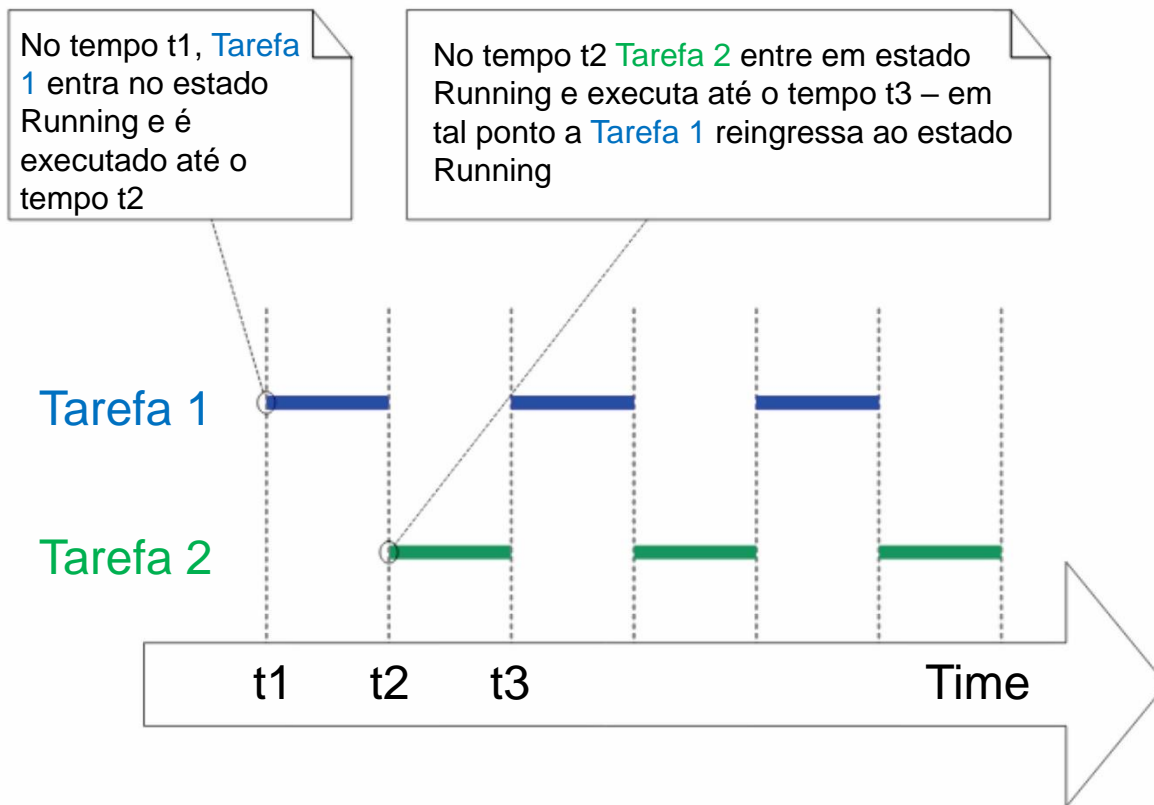
- **TickType\_t**
  - O FreeRTOS configura uma interrupção periódica chamada de interrupção de tick. O tempo entre duas interrupções tick é chamado de período do tick. Os horários são especificados como múltiplos de períodos de tick.
- **BaseType\_t**
  - Geralmente é usado para funções de retorno que podem levar apenas um intervalo muito limitado de valores, e booleans do tipo **pdTRUE** / **pdFALSE**.

# Execução de Tarefas com prioridade similar

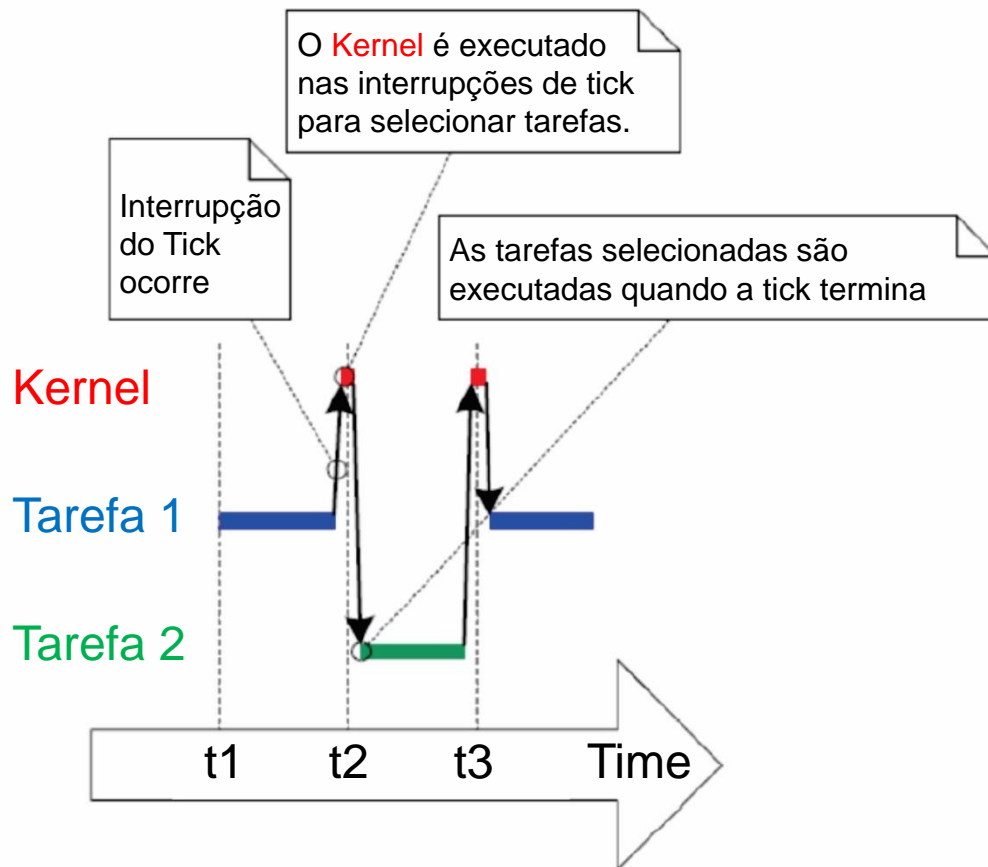
```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile uint32_t ul;
    /*The string to print out is passed in via the parameter.*/
    pcTaskName = ( char * ) pvParameters;
    /* As per most tasks, this task is implemented in an infiniteloop. */
    For( ;; )
    {
        vPrintString( pcTaskName );/* Print out the name of this task. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )/*Delay for a period. */
        {
        }
    }
}

/* main function */
Static const char *pcTextForTask1 ="Task 1 is running\r\n";
static const char *pcTextForTask2 ="Task 2 is running\r\n";
int main(void)
{
    /* Create one of the two tasks. */
    xTaskCreate(vTaskFunction,"Task 1",1000,(void*)pcTextForTask1,1,NULL);
    /* Create the second task from the SAME task implementation (vTaskFunction). Only the value
    passed in the parameter is different. */
    xTaskCreate(vTaskFunction,"Task 2",1000,(void*)pcTextForTask2,1,NULL);
    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
    for( ;; );
}
```

# Execução de Tarefas com prioridade similar



# Execução de Tarefas com prioridade similar



Para selecionar a próxima tarefa a ser executada, o próprio programador deve executar a cada interrupção periódica, chamada 'tick interrupt'.

A frequência de interrupção Tick, é configurada pelo aplicativo definido `configTICK_RATE_HZ` constante (tempo de compilação) dentro de `FreeRTOSConfig.h`.

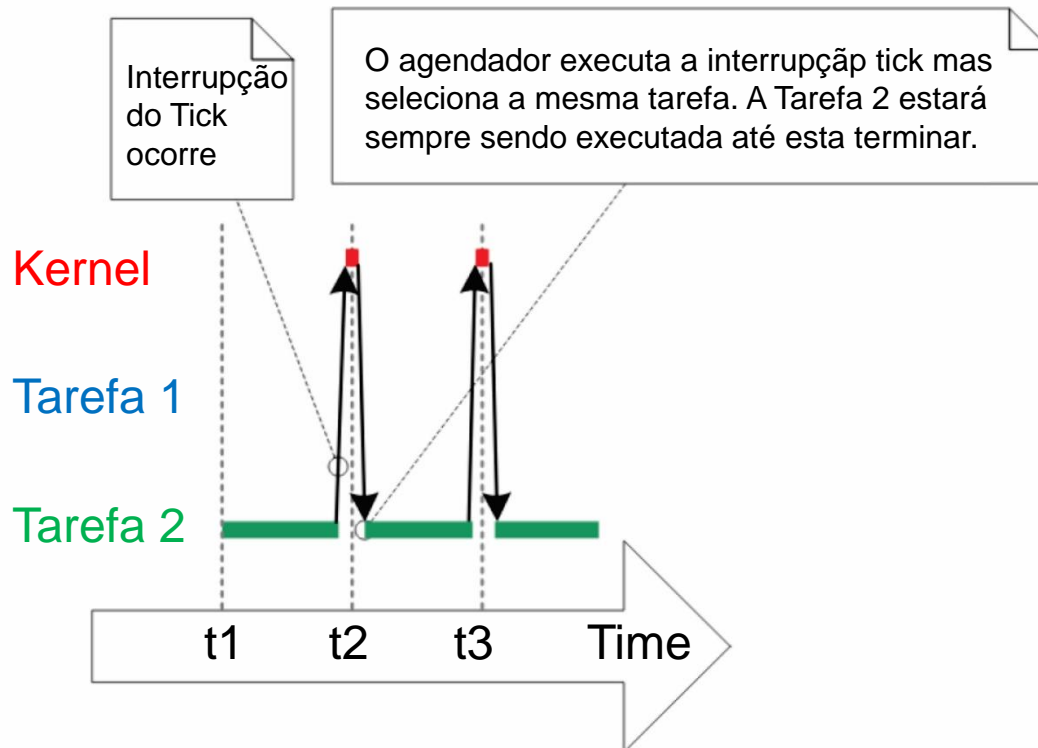
Valor típico de 100Hz  
Tempo = 10 ms

# Execução de Tarefas com prioridade diferente

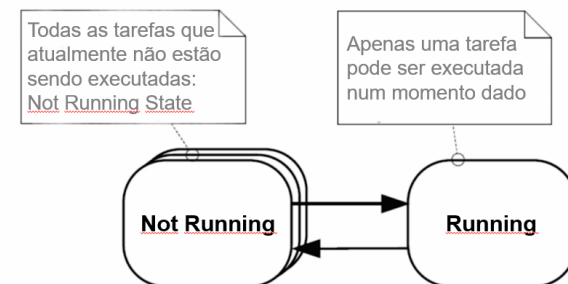
```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile uint32_t ul;
    /*The string to print out is passed in via the parameter.*/
    pcTaskName = ( char * ) pvParameters;
    /* As per most tasks, this task is implemented in an infiniteloop. */
    For( ;; )
    {
        vPrintString( pcTaskName );/* Print out the name of this task. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )/*Delay for a period. */
        {
        }
    }
}

/* main function */
Static const char *pcTextForTask1 ="Task 1 is running\r\n";
static const char *pcTextForTask2 ="Task 2 is running\r\n";
int main(void)
{
    /* Create one of the two tasks. */
    xTaskCreate(vTaskFunction,"Task 1",1000,(void*)pcTextForTask1,1,NULL);
    /* Create the second task with higher priority*/
    xTaskCreate(vTaskFunction,"Task 2",1000,(void*)pcTextForTask2,2,NULL);
    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
    for( ;; );
}
```

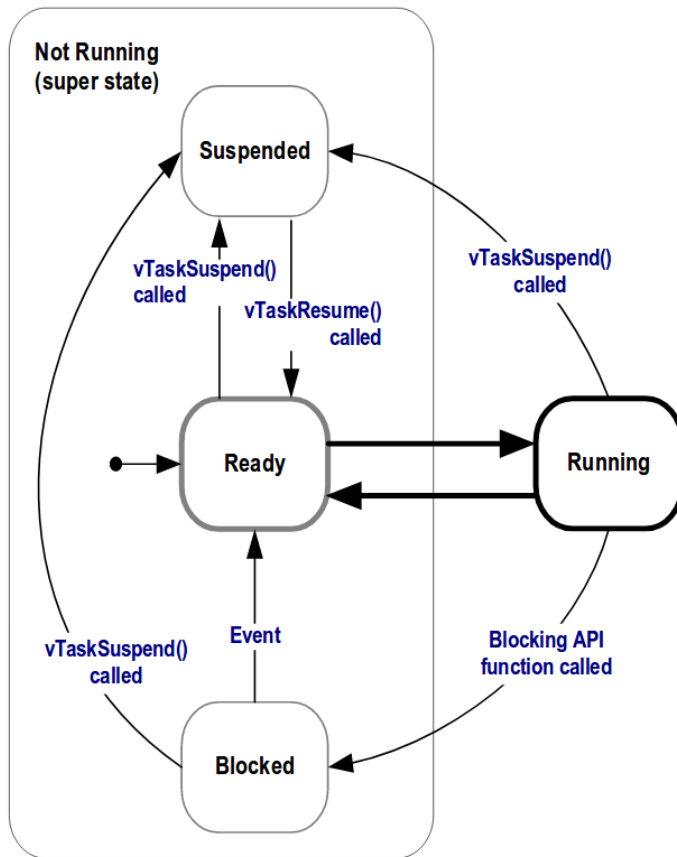
# Execução de Tarefas com prioridade diferente



A Tarefa 2 tem maior prioridade que a Tarefa 1



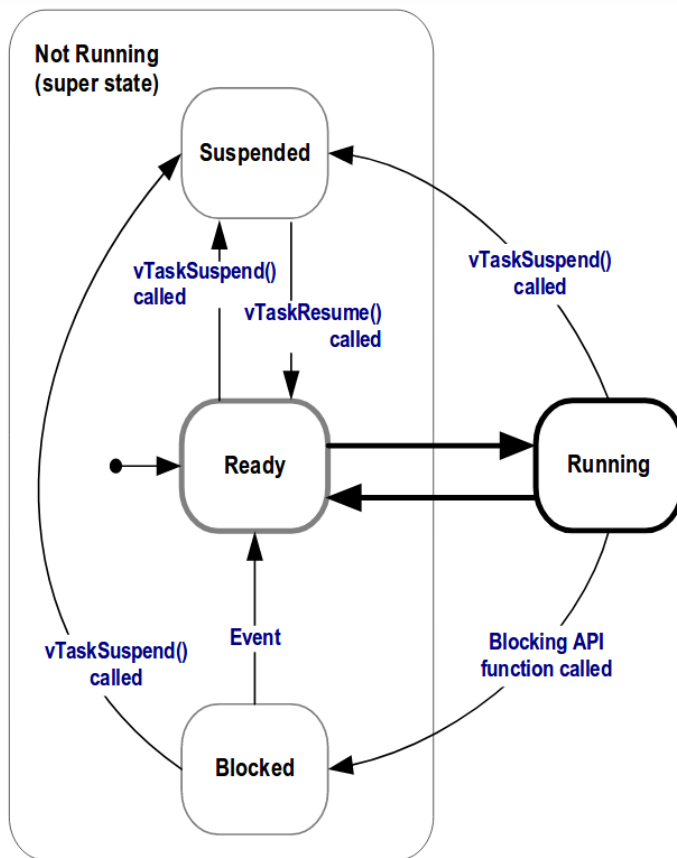
# Analizando o Estado 'Not Running'



- Para tornar as tarefas úteis, elas devem ser reescritas para serem **orientadas por eventos**.
- Uma tarefa é acionada quando ocorre um evento e não é possível entrar no estado '**Running**' antes que o evento tenha ocorrido.



# Analizando o Estado 'Not Running'



Quando uma tarefa está esperando por um evento, está bloqueada

- Para tipos de eventos
- Temporal - atrasos
- Sincronização - Aguardando dados em uma fila

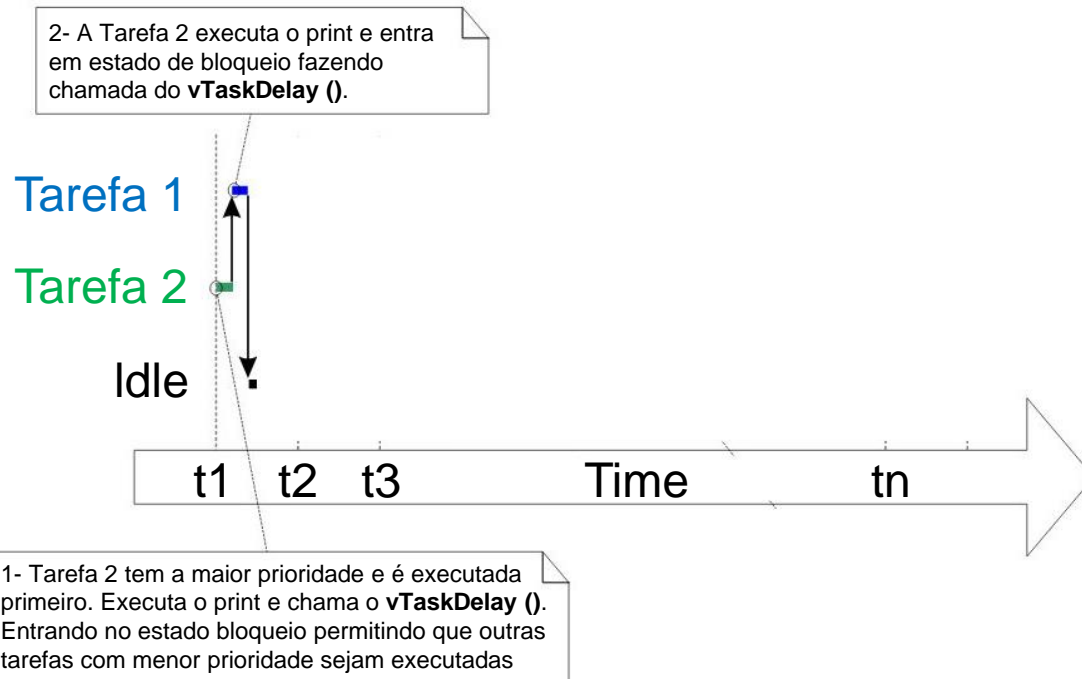
# Analizando o Estado 'Not Running'

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );
    volatile uint32_t ul;
    /*The string to print out is passed in via the parameter.*/
    pcTaskName = ( char * ) pvParameters;
    /* As per most tasks, this task is implemented in an infiniteloop. */
    For( ;; )
    {
        vPrintString( pcTaskName );/* Print out the name of this task. */
        vTaskDelay(xDelay250ms);
    }
}
```

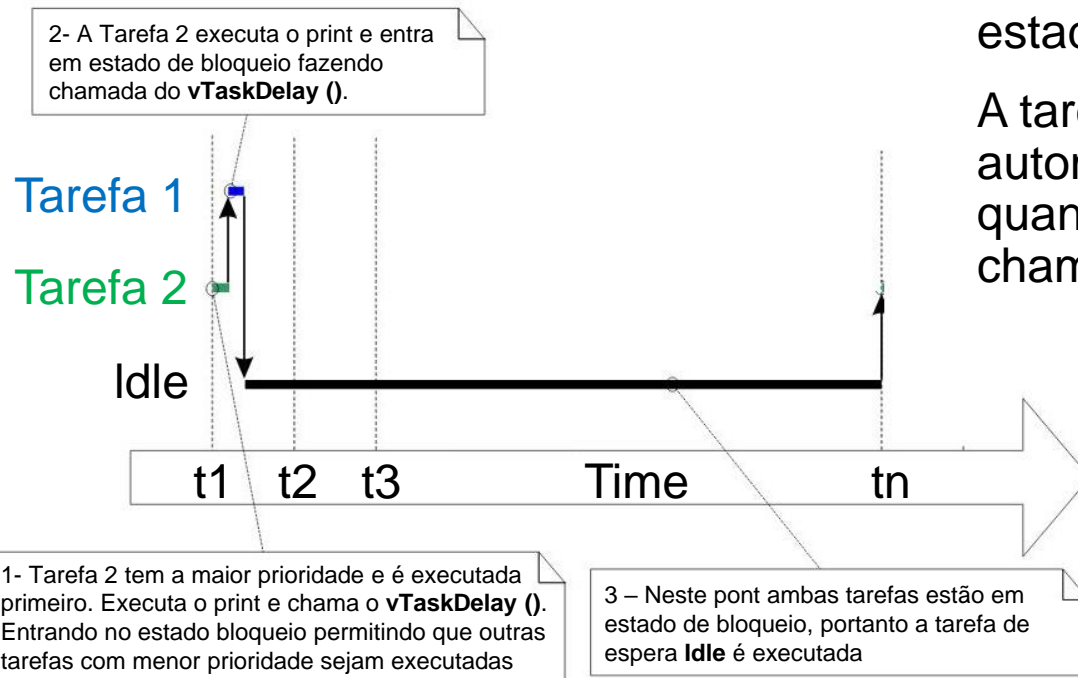
**vTaskDelay ()** coloca a tarefa no estado Bloqueado até que o período de atraso expirar.

**void vTaskDelay (portTickType xTicksToDelay);**

# Analizando o Estado 'Not Running'



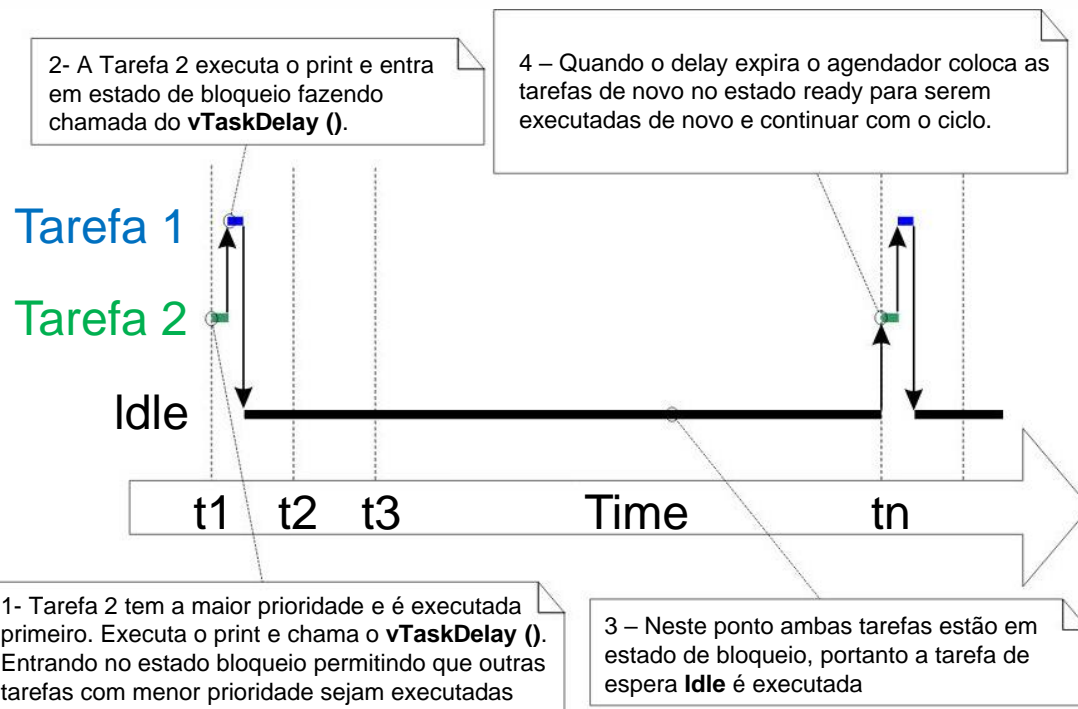
# Analizando o Estado 'Not Running'



Deve sempre haver pelo menos uma tarefa que possa entrar no estado Running.

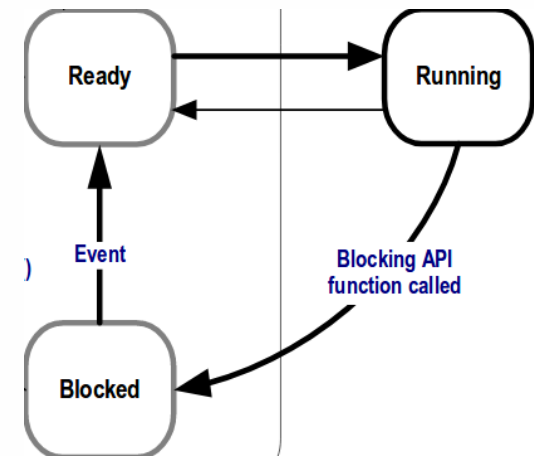
A tarefa Idle é criada automaticamente pelo agendador quando **vTaskStartScheduler ()** é chamado.

# Analizando o Estado 'Not Running'

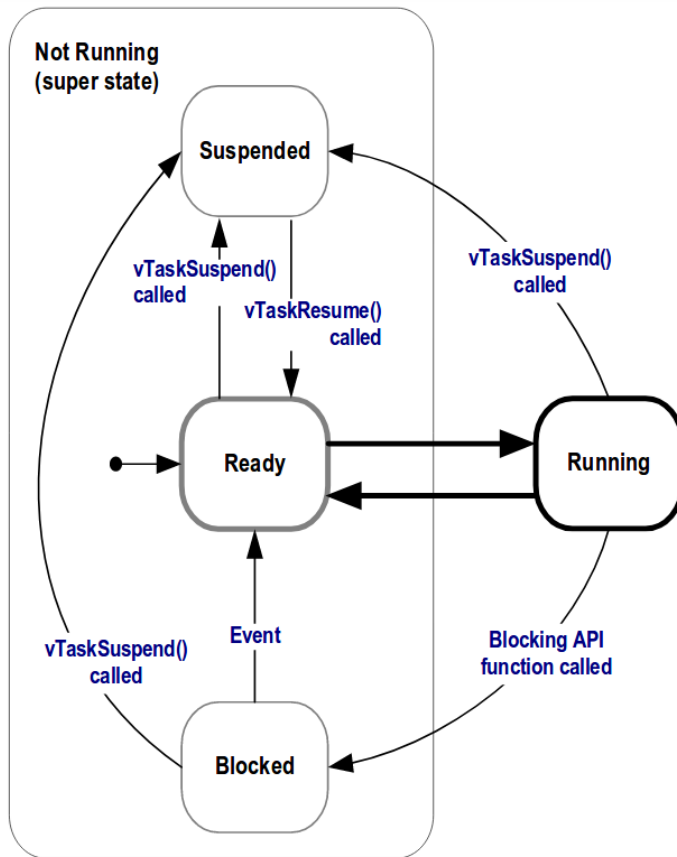


Implementação mais eficiente de tarefas.

Menos Uso do processador.



# Analizando o Estado 'Not Running'



O Estado Suspenso também é um subestado de 'Not Running'.

Tarefas no estado suspenso não estão disponíveis para o agendador.

API `vTaskSuspend()`

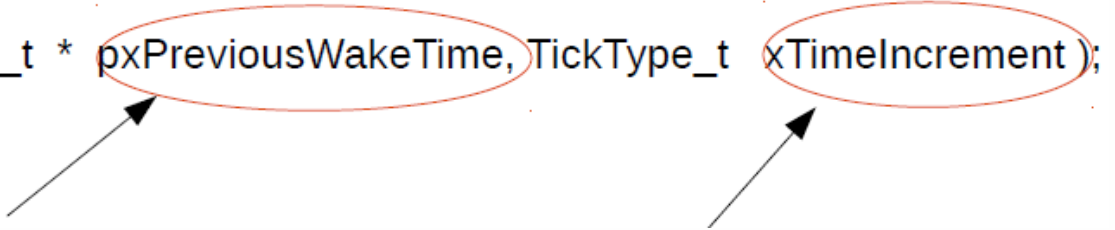
API `vTaskResume()` ou `xTaskResumeFromISR()`.

A maioria das aplicações não usa o estado Suspenso.

# Executando Tarefas Periódicas

- Usar **vTaskDelay()** não garante o período no qual uma tarefa será executada. Esta apenas indica o numero de interrupções Tick que está se manterá em estado de bloqueio.

```
void vTaskDelayUntil(TickType_t * pxPreviousWakeTime, TickType_t xTimeIncrement );
```

A diagram illustrating the parameters of the vTaskDelayUntil function. The parameter 'pxPreviousWakeTime' is circled in red, with an arrow pointing to it from below. The parameter 'xTimeIncrement' is also circled in red, with an arrow pointing to it from below.

- Os parâmetros de **vTaskDelayUntil()** especifica a quantidade exata de Ticks na qual uma tarefa será chamada depois de um estado de bloqueio.

# Combinando Tarefas Periódicas e Contínuas

```
void vContinuousFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile uint32_t ul;
    pcTaskName = ( char * ) pvParameters;
    For( ;; )
    {
        vPrintString( pcTaskName );/* Print out the name of this task. */
    }
}

void vPeriodicFunction( void *pvParameters ){
    char *pcTaskName;
    TickType_t xLastWakeTime;
    pcTaskName = ( char * ) pvParameters;
    xLastWakeTime = xTaskGetTickCount();/* current tickcount.*/
    for( ;; ){/* Print out the name of this task. */
        vPrintString( pcTaskName );
        vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ));
    }
}

/* main function */
Static const char *pcTextForTask1 ="Continuous task 1 running\r\n";
static const char *pcTextForTask2 ="Continuous task 2 running\r\n";
static const char *pcTextforperiodic ="Periodic task is running\r\n";
int main(void)
{
    xTaskCreate(vContinuousFunction,"Task 1",1000,(void*)pcTextForTask1,1,NULL);
    xTaskCreate(vContinuousFunction,"Task 2",1000,(void*)pcTextForTask2,1,NULL);
    xTaskCreate(vPeriodicFunction,"Task periodic",1000,(void*)pcTextforperiodic,2,NULL);
    vTaskStartScheduler();
    for( ;; );
}
```



# Outras funções relacionadas a Tarefas

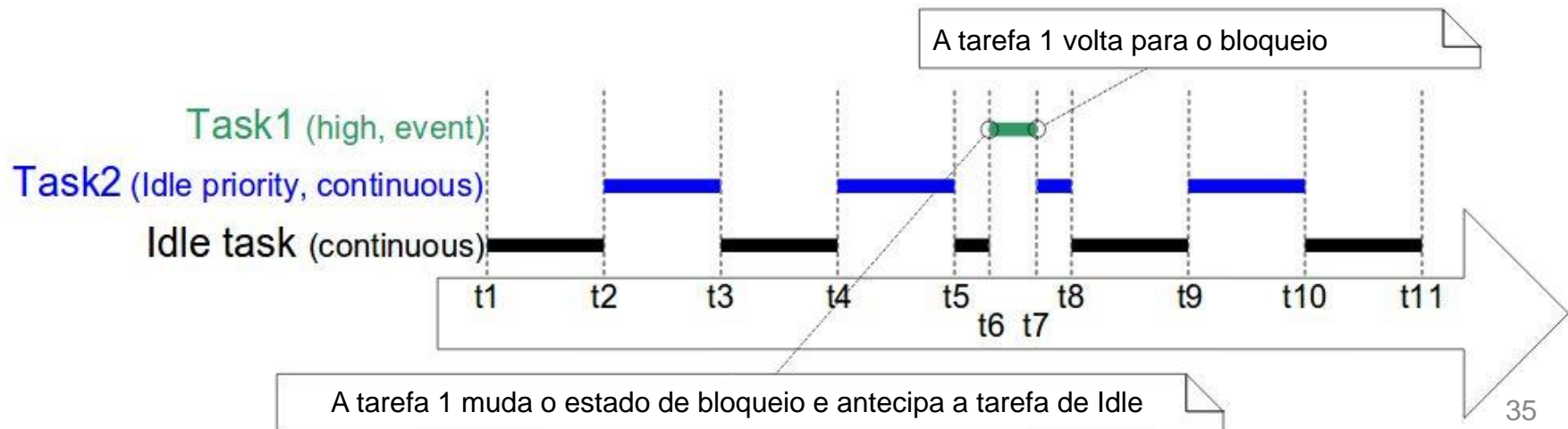
- void **vTaskPrioritySet** (TaskHandle\_t pxTask, UbaseType\_t uxNewPriority);
  - pxTask: O handle da tarefa (último parâmetro da função **taskCreate**)
  - uxNewPriority: Nova prioridade a ser definida
- UbaseType\_t **uxTaskPriorityGet** (TaskHandle\_t pxTask);
- void **vTaskDelete** (TaskHandle\_t pxTaskToDelete);
  - pxTaskToDelete: O handle da tarefa

# Algoritmos de Agendamento

- Programação prioritária de prioridade fixa com fatiamento de tempo
  - Prioridade fixa: não altere os níveis atribuídos a tarefas
  - Preemptivo: antecipa-se imediatamente à tarefa em execução se uma tarefa de prioridade mais alta entrar no estado Ready
  - Time slicing: é usado para compartilhar tempo de processamento entre tarefas de igual prioridade - Tempo entre duas interrupções de ticks RTOS

Configurado no **FreeRTOSConfig.h**

- configUSE\_PREEMPTION 1
- configUSE\_TIME\_SLICING 1

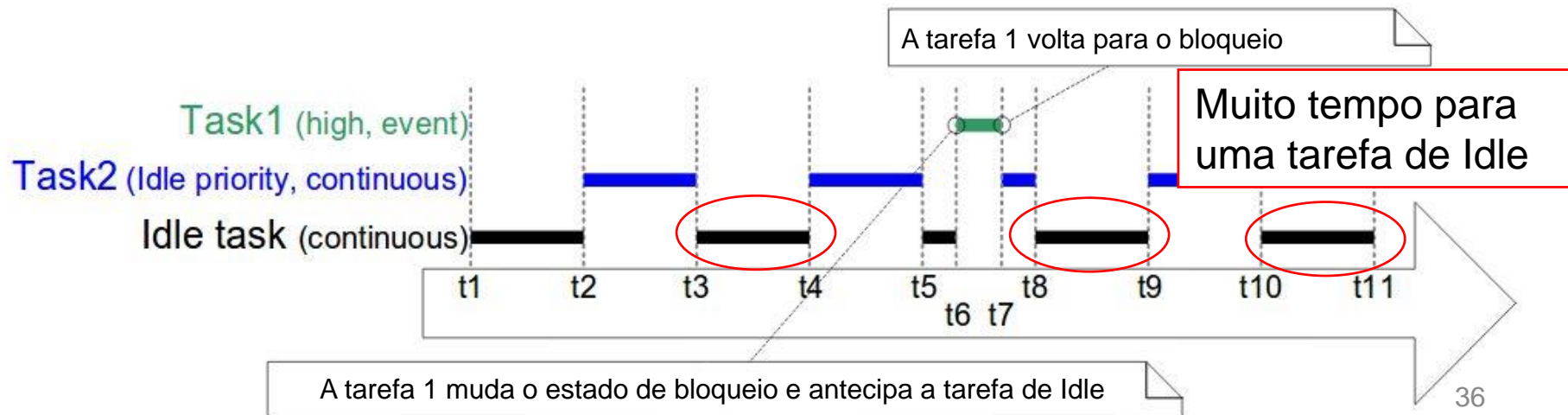


# Algoritmos de Agendamento

- Programação prioritária de prioridade fixa com fatiamento de tempo
  - Prioridade fixa: não altere os níveis atribuídos a tarefas
  - Preemptivo: antecipa-se imediatamente à tarefa em execução se uma tarefa de prioridade mais alta entrar no estado Ready
  - Time slicing: é usado para compartilhar tempo de processamento entre tarefas de igual prioridade - Tempo entre duas interrupções de ticks RTOS

Configurado no **FreeRTOSConfig.h**

- configUSE\_PREEMPTION 1
- configUSE\_TIME\_SLICING 1



# Algoritmos de Agendamento

- Programação prioritária de prioridade fixa com fatiamento de tempo
- Programação prioritária de prioridade fixa **sem fatiamento de tempo**

Configurado no **FreeRTOSConfig.h**

- `configUSE_PREEMPTION 1`
- `configUSE_TIME_SLICING 0`

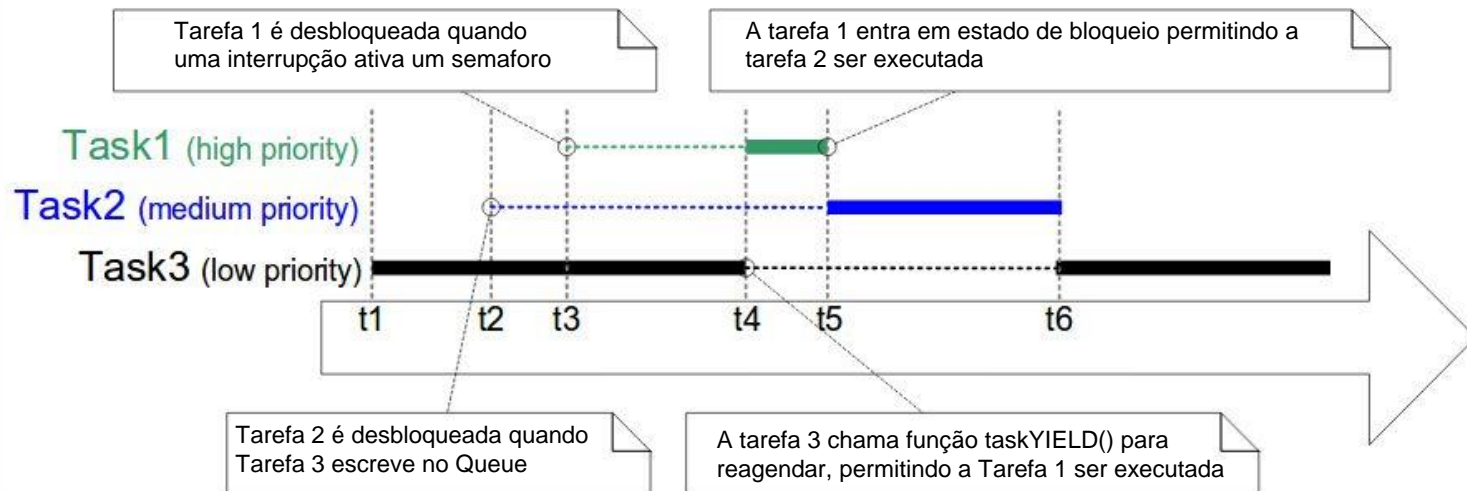
# Algoritmos de Agendamento

- Programação prioritária de prioridade fixa com fatiamento de tempo
- Programação prioritária de prioridade fixa sem fatiamento de tempo
- **Agendamento Cooperativo**

Configurado no **FreeRTOSConfig.h**

- configUSE\_PREEMPTION 0
- configUSE\_TIME\_SLICING any

Chamar função **taskYIELD()** para reagendar



# Queue

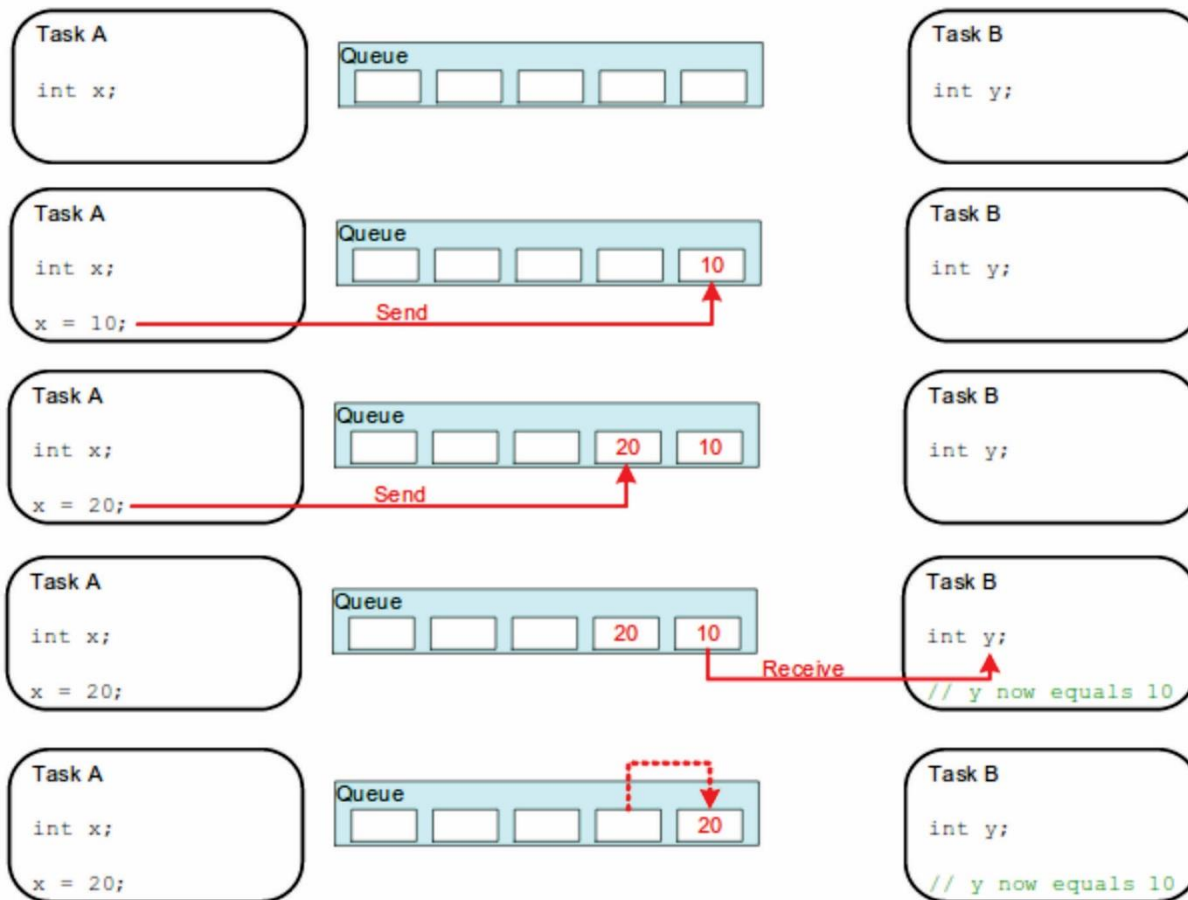
Queues são filas que fornecem um mecanismo de comunicação tarefa-tarefa, tarefa-interrupção e interrupção-tarefas.

- Queue contêm um número finito de dados de tamanho fixo
- As filas são normalmente usadas como buffers FIFO

O FreeRTOS usa fila pelo **método de cópia**.

- A tarefa de envio e a tarefa de recebimento são completamente desacopladas.
- O RTOS assume total responsabilidade pela alocação da memória usada para armazenar dados.

# Gerenciamento de Queue



# Criando uma Queue

## Criando uma Queue

QueueHandle\_t **xQueueCreate**(UBaseType\_t uxQueueLength, UbaseType\_t uxItemSize);

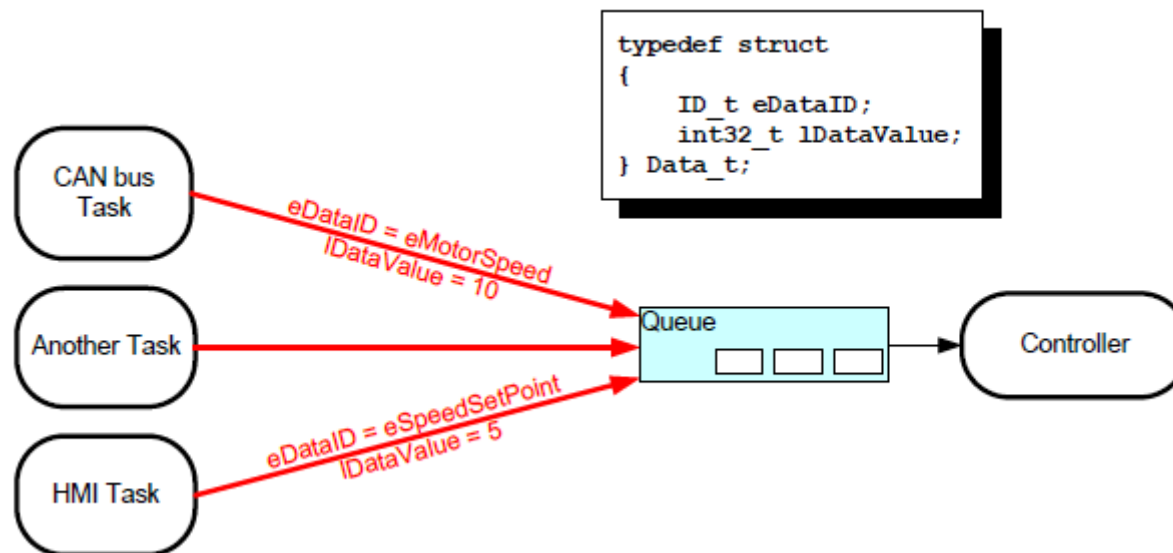
- UxQueueLength: O numero máximo de itens que podem ser guardados.
- UxItemSize: O tamanho de bytes de cada item na fila.

## Types

- Queue (conventional)
- Mailbox : Queue com apenas um elemento
- Queue Set : Permite que uma tarefa receba dados de mais de uma queue sem que a tarefa sonde (polling) cada queue para verificar quais contém dados disponíveis.



# Gerenciamento de Queue



Recebendo dados de múltiplas fontes

# Gerenciamento de Queue

## Escrever em uma Queue

```
BaseType_t  
xQueueSendToFront(QueueHandle_t xQueue,  
    const void * pvltemToQueue,  
    TickType_t xTicksToWait );
```

```
BaseType_t  
xQueueSendToBack(QueueHandle_t xQueue,  
    const void * pvltemToQueue,  
    TickType_t xTicksToWait );
```

- xQueue: O handle da fila.
- pvltemToQueue: O pontero do dado que vai ser copiado.
- xTicksToWait: O tempo maximo que a tarefa deve ficar em estado de bloqueio para liberar o espaço na queue

## Ler de uma Queue

```
BaseType_t  
xQueueReceive( QueueHandle_t xQueue,  
    void * const pvBuffer,  
    TickType_t xTicksToWait );
```

- pBuffer: Um puntero do destino do dado a ser copiado
- xTicksToWait: O maximo tempo de espero que a tarefa deve ficar em estado de bloqueio para que o espaço esteja disponivel

Retornam:

- pdPASS – OK
- errQUEUE\_EMPTY – Error, queue full

# Software Timer

- Timers de Software são usados para agendar a execução de uma função em um tempo específico ou periodicamente com uma frequência fixa.
- All software timer callback functions execute in the context of the same RTOS daemon (or 'timer service') task1.

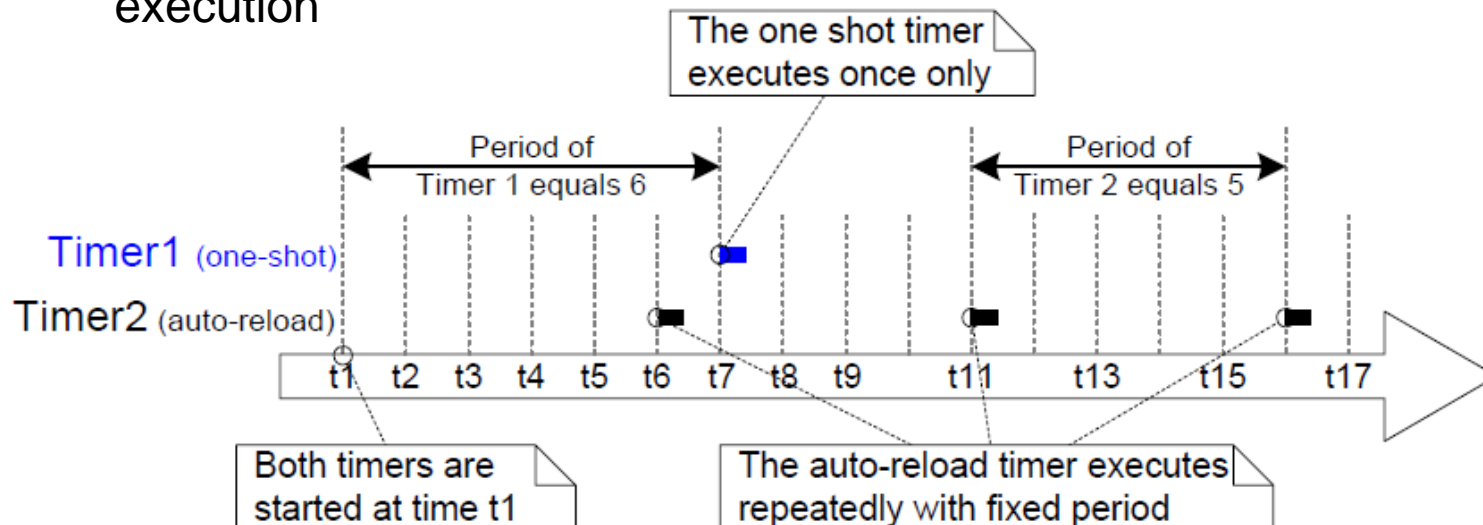
# Criando um Timer

## Type

- One-Shot Timer
- Auto Reload Timer

## Timer's Period

- Time between timer being started and the software timer's call-back function execution



# Interrupções

## Eventos

- Os sistemas embarcados de tempo real precisam executar ações em resposta a eventos originados do ambiente.
- Como eles devem ser detectados? Interrupções, polling
- Que tipo de processamento precisa ser feito? Dentro do ISR, fora do ISR

## Prioridade de interrupção versus prioridade da tarefa

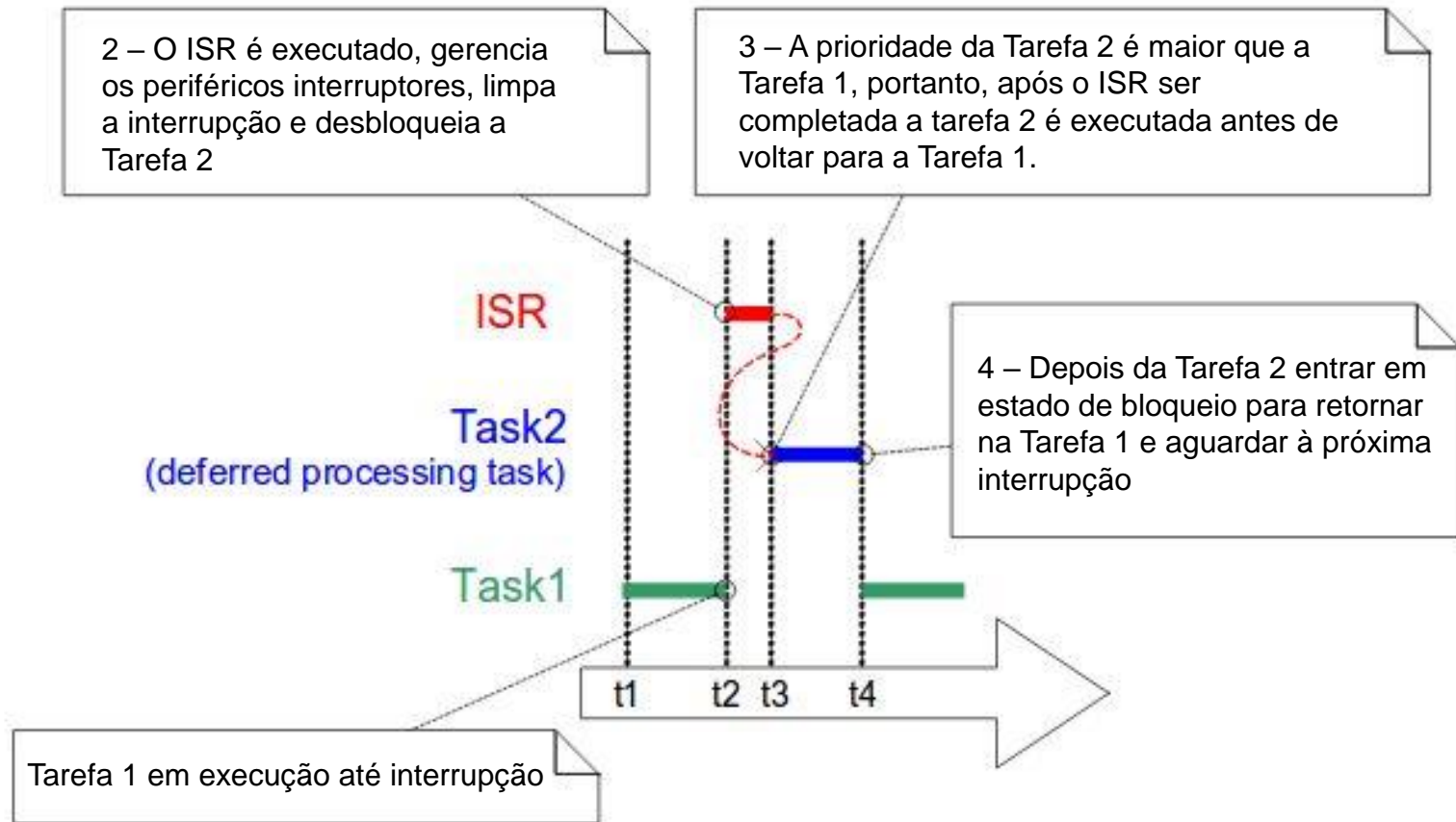
- Interrupção de prioridade mais baixa antecipa a tarefa de prioridade mais alta

## API Função Segura para Interrupção

O FreeRTOS fornece duas versões de algumas funções das API:

- um para uso de tarefas,
- e um para uso de ISRs ("FromISR" anexado ao seu nome).

# Gerenciamento de Interrupções



# Gerenciamento de Interrupções

## Semáforos Binários Usados para Sincronização

- A tarefa de processamento pode ser controlada usando um ISR
- O ISR "dá" um semáforo para desbloquear a tarefa adiada
- A tarefa adiada "leva" o semáforo para entrar no estado bloqueado

## API Funções para gerenciar semáforos

### Criando um Semáforo

- SemaphoreHandle\_t **xSemaphoreCreateBinary** (void);

### Take

- BaseType\_t **xSemaphoreTake** (SemaphoreHandle\_t xSemaphore,  
TickType\_t xTicksToWait);

### Give

- BaseType\_t **xSemaphoreGiveFromISR** (SemaphoreHandle\_t xSemaphore,  
BaseType\_t \*pxHigherPriorityTaskWoken);

# Vamos praticar?





# Referências

- BARRY, Richard. Mastering the FreeRTOS Real Time Kernel. **Real Time Engineers Ltd**, 2016.
- **FreeRTOS reference manual: API functions and configuration options.** Real Time Engineers Limited, 2009.

# Obrigada!