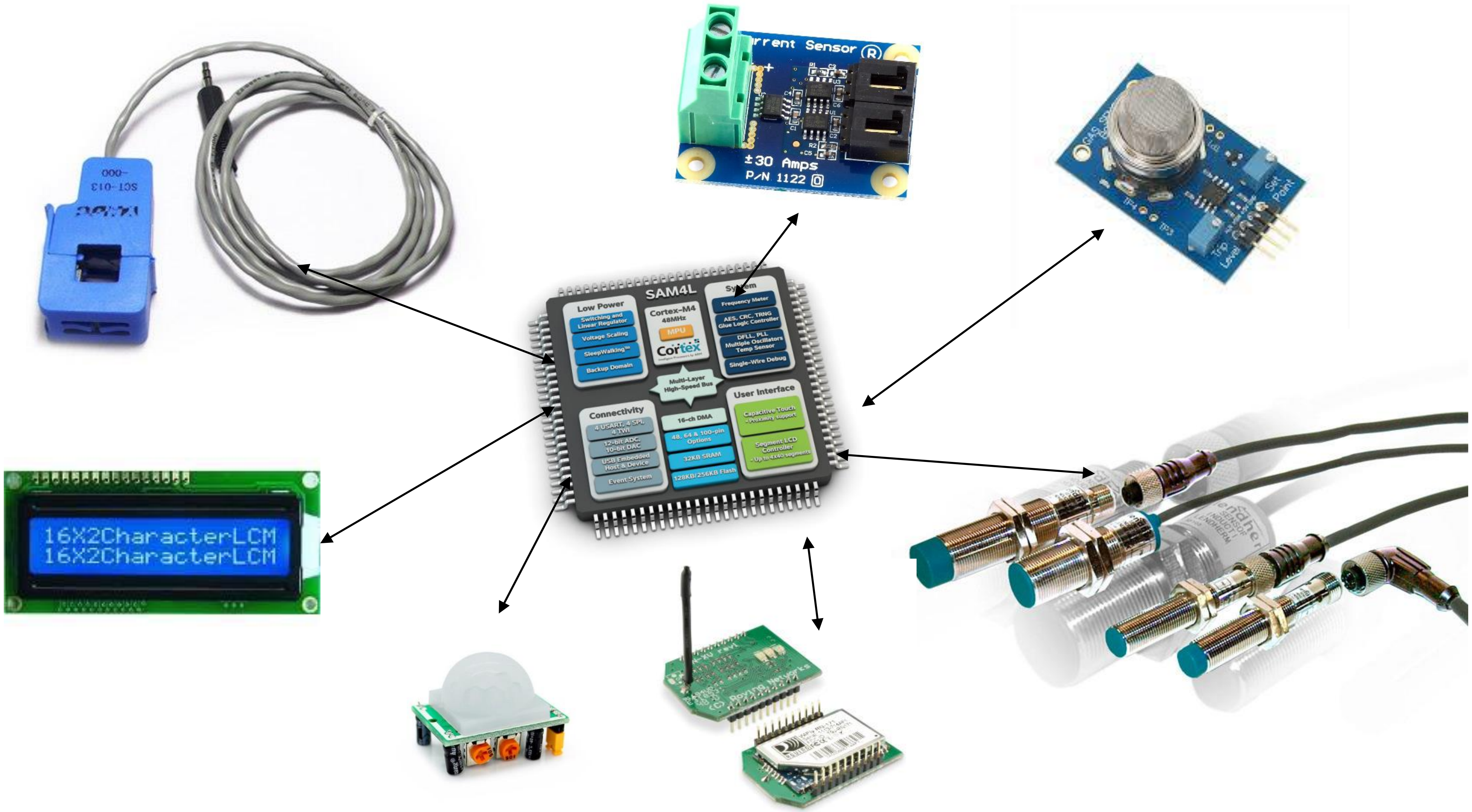
Sistema Operacional de Tempo Real

Prof° Fernando Simplicio

PROJETO com MCU



PROGRAMA em C

```
void main()
{
        InitSys();
        while(true) {
          SensorCorrente();
          Termopar()
          Umidade();
          SensorPresenca();
          SendToUart();
          UpdateLcd();
    }
}
```

PROGRAMA em C

```
void main()
{
        InitSys();
        while(true) {
          SensorCorrente();      //2ms
          Termopar()             //10ms
          Umidade();             //15ms
          SensorPresenca();      //100ms
          SendToUart();          //1ms
          UpdateLcd();           //10ms
    }
}    //Tempo Gasto por loop: 138ms
```

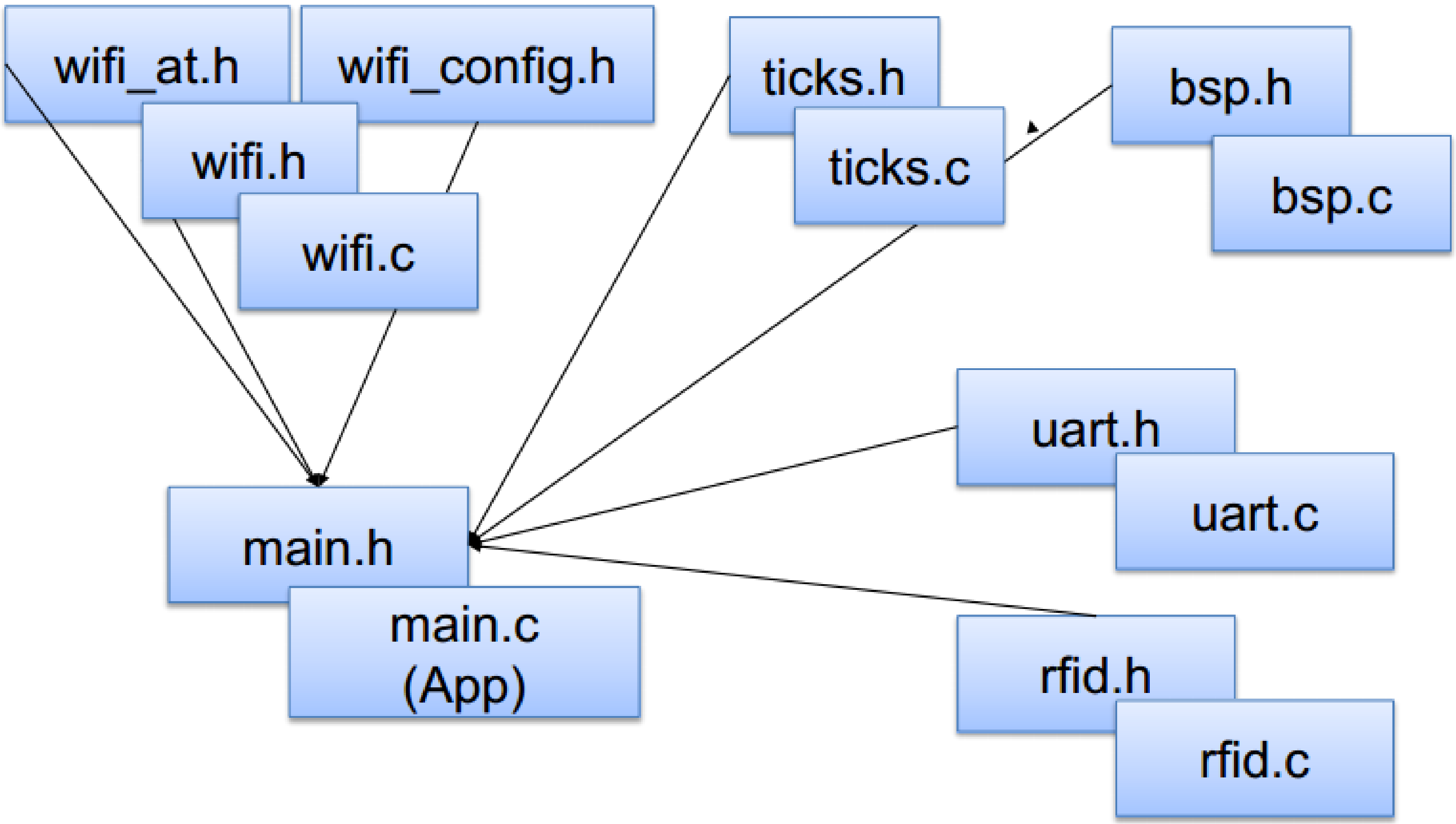
Máquina de Estado

- **Break long tasks into a state machine**

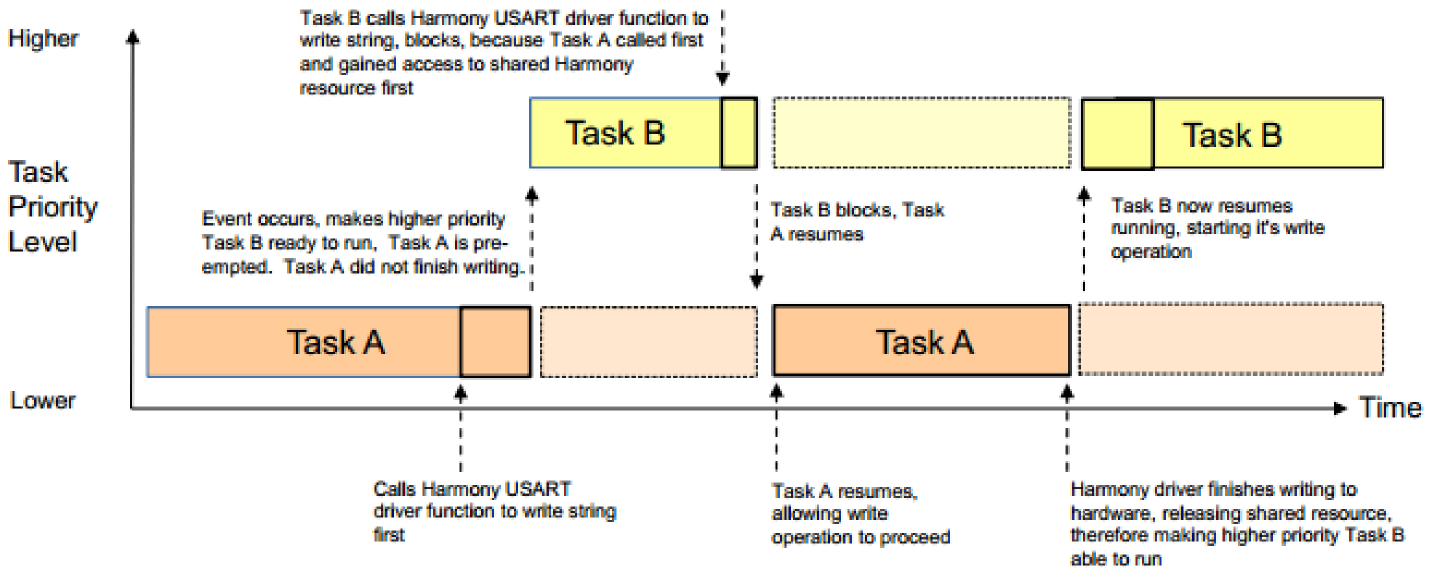
```
while (1)
{
  Task_1(); //60 us
  Task_2(); //2 us
}
//max loop time = 62 us
```

```
while (1)
{
  switch(Task_1_state)
  {
   case a:
     Task_1_state_a(); //20 us
     break;
   case b:
     Task_1_state_b(); //20 us
     break;
   case c:
     Task_1_state_c(); //20 us
     break;
  }
  Task_2();  //2 us
} //max loop time = 22 us
```

Compartilhamento dos Recursos



Drivers dos Recursos



Task A

```
signed char* pTaskA_str = "Hello World";

void Task_A_Func(void* pvParameter)
{
   while(true)
   {
      /* Perform a specific task job */
      DRV_USART_Write(UART_1_Handle,
         (void*)pTaskA_str,
      (size_t)(strlen(pTaskA_str)) );
   }
}
```

Task B

```
signed char* pTaskB_str = "Go Rattlers!";

void Task_B_Func(void* pvParameter)
{
   while(true)
   {
      /*perform Harmony or application specific
      task job*/
      DRV_USART_Write(UART_1_Handle,
         (void*)pTaskB_str,
      (size_t)(strlen(pTaskB_str)));
   }
}
```

Multitasking

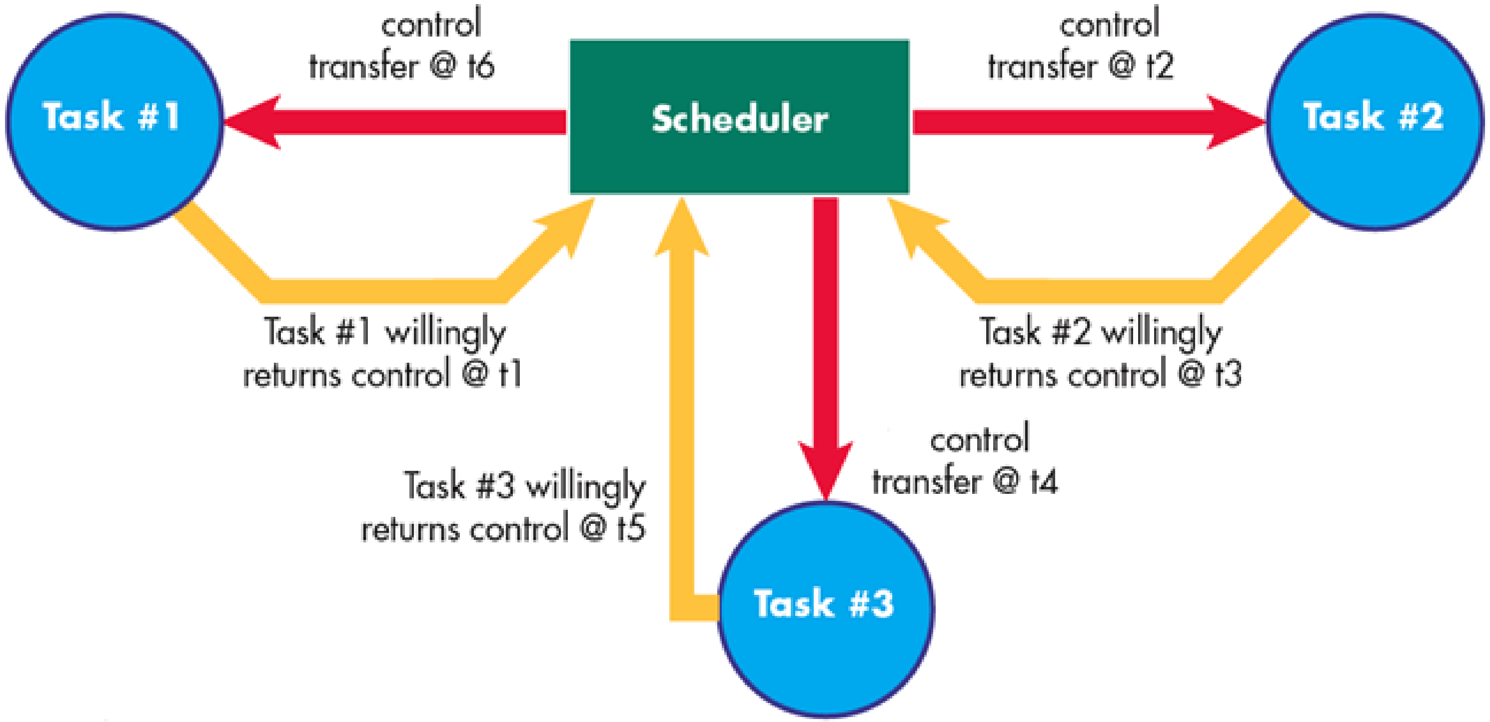
- **Quais os problemas de um programa Multitasking?**

Em *multitask* devemos considerar (...)

- **O tempo para troca de contexto.**
- **O consumo de memória para armazenamento do contexto.**
- ***As Tasks* podem ser tolerantes a *loops* infinitos.**
- **O compartilhamento de recursos (registradores internos, memória, unidades lógicas aritméticas da CPU, periféricos (LCD, Sensores e atuadores)) entre as outras *tasks* concorrentes.**

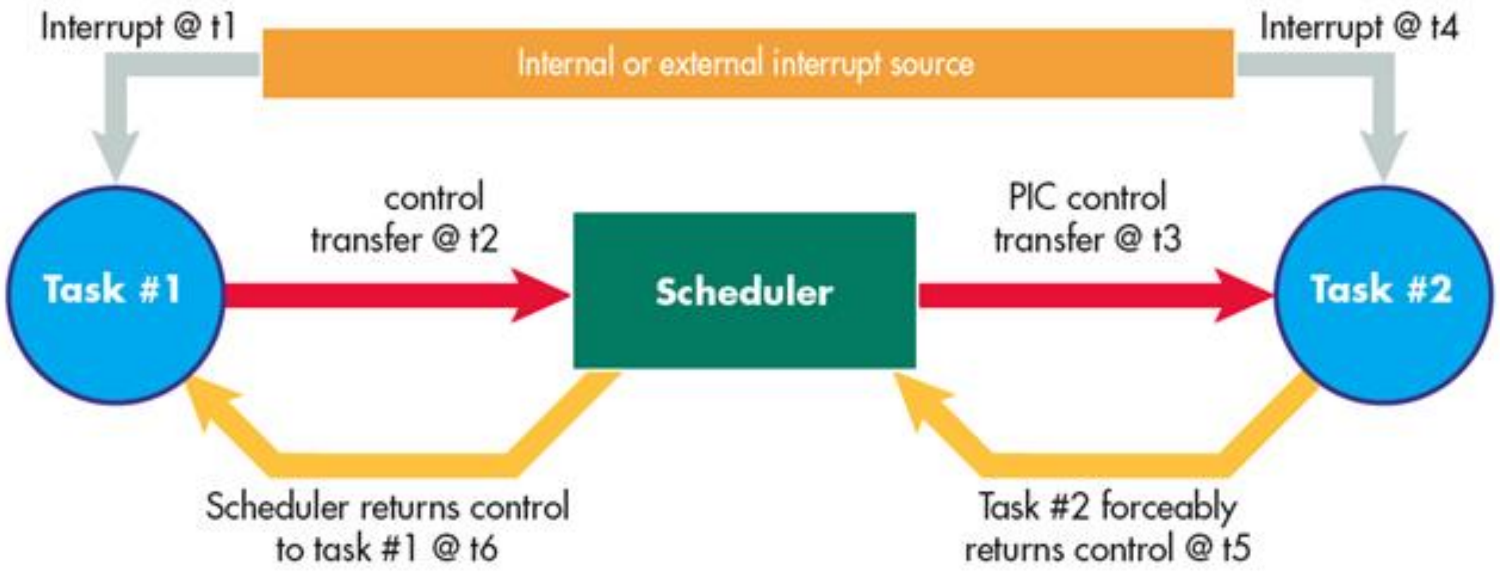
multitasking policy

- As regras de chaveamento das tasks é determinada pelo *Scheduler*.

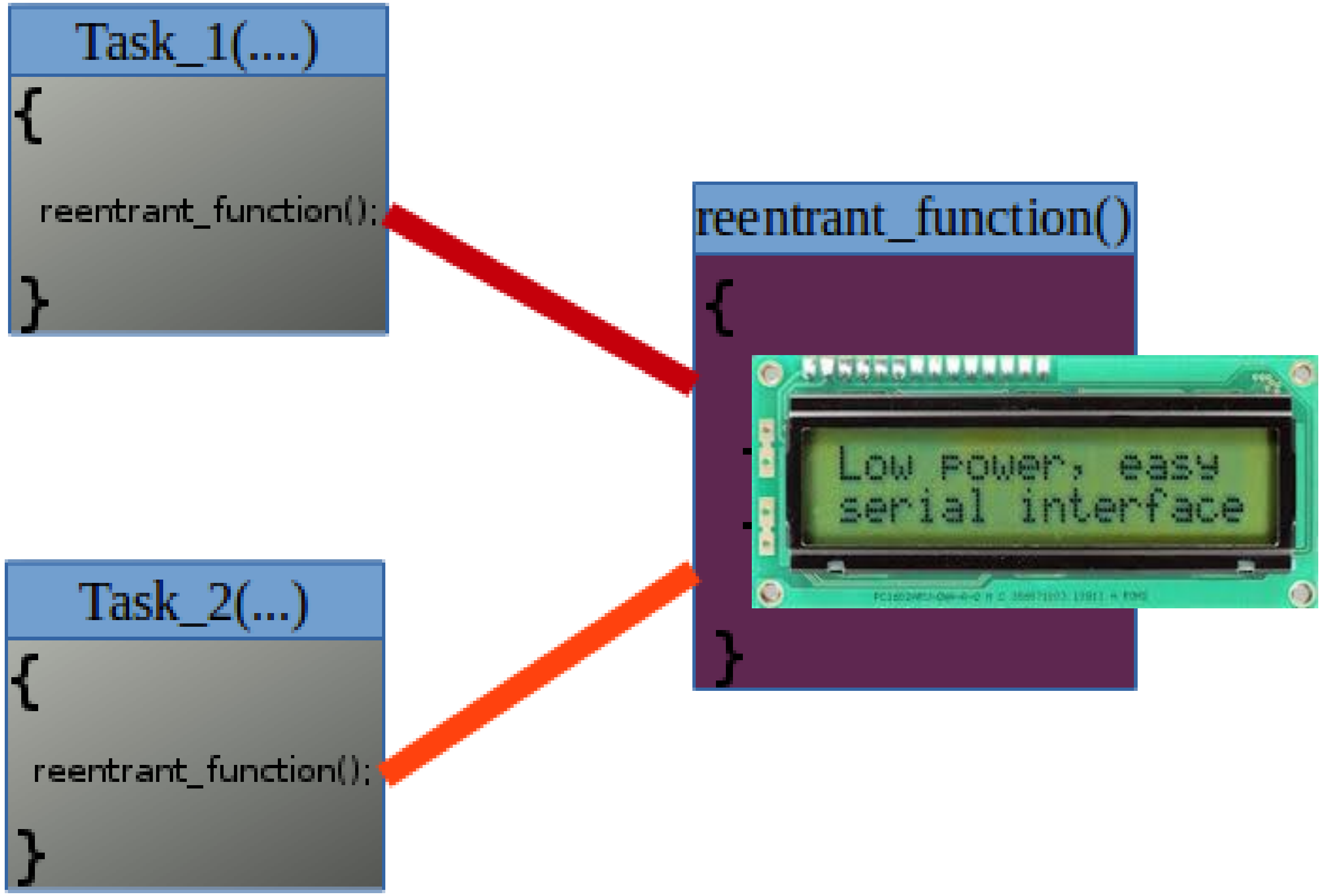


multitasking preemptive

- É quando uma *task* em execução é interrompida em algum ponto por um evento interno ou externo do sistema.



Setores Críticas



Reentrância ("re-enter”)

- **Denota os problemas que podem ocorrer quando o mesmo código é executado simultaneamente por várias tarefas ou quando os dados globais são acessados simultaneamente por várias tarefas.**

- **Em um ambiente *multitasking*, as funções preferencialmente devem ser reentrantes.**

Seções Críticas

- **Instruções de Microcontroladores**
- **(Non-atomic Access)**

```
/* The C code being compiled. */
PORTA |= 0x01;

/* The assembly code produced when the C code is compiled. */
LOAD   R1,[#PORTA] ; Read a value from PORTA into R1
MOVE   R2,#0x01    ; Move the absolute constant 1 into R2
OR     R1,R2       ; Bitwise OR R1 (PORTA) with R2 (constant 1)
STORE  R1,[#PORTA] ; Store the new value back to PORTA
```

Seções Críticas

- Isso SIM é uma função REENTRANTE!

```
/* A parameter is passed into the function.  This will either be passed on the stack,
or in a processor register.  Either way is safe as each task or interrupt that calls
the function maintains its own stack and its own set of register values, so each task
or interrupt that calls the function will have its own copy of lVar1. */
long lAddOneHundred( long lVar1 )
{
/* This function scope variable will also be allocated to the stack or a register,
depending on the compiler and optimization level.  Each task or interrupt that calls
this function will have its own copy of lVar2. */
long lVar2;

    lVar2 = lVar1 + 100;
    return lVar2;
}
```

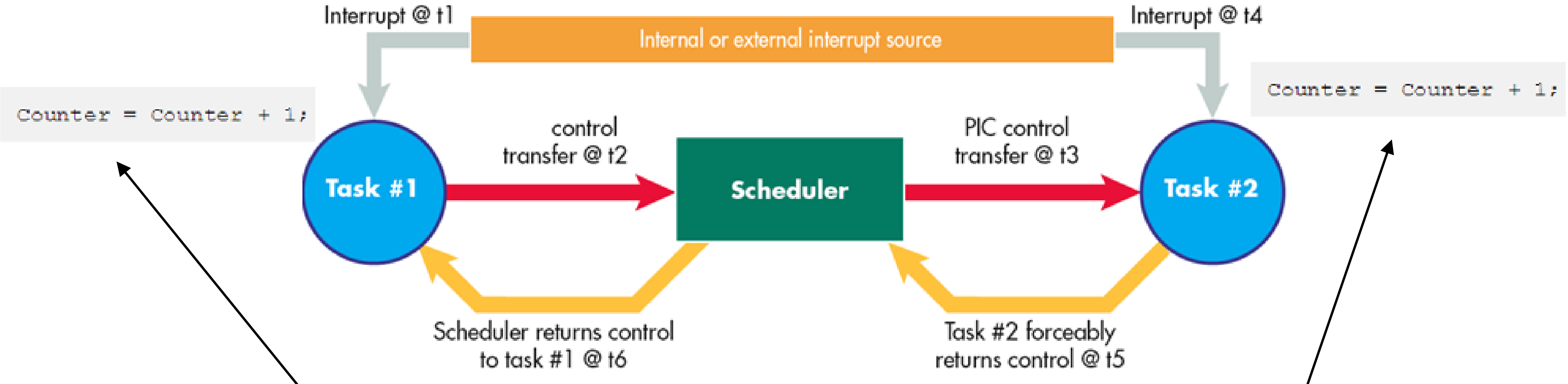
Seções Críticas

- Isso NÃO é uma função REENTRANTE!

```
/* In this case lVar1 is a global variable, so every task that calls
lNonsenseFunction will access the same single copy of the variable. */
long lVar1;

long lNonsenseFunction( void )
{
/* lState is static, so is not allocated on the stack.  Each task that calls this
function will access the same single copy of the variable. */
static long lState = 0;
long lReturn;

    switch( lState )
    {
        case 0 : lReturn = lVar1 + 10;
                 lState = 1;
                 break;

        case 1 : lReturn = lVar1 + 20;
                 lState = 0;
                 break;
    }
}
```

Exemplo: Reentrância ("re-enter")

- **Por uma questão de simplicidade, suponha que ambas as tarefas tenham a mesma prioridade, o agendamento preventivo e o** *time slicing* **está ativo.**

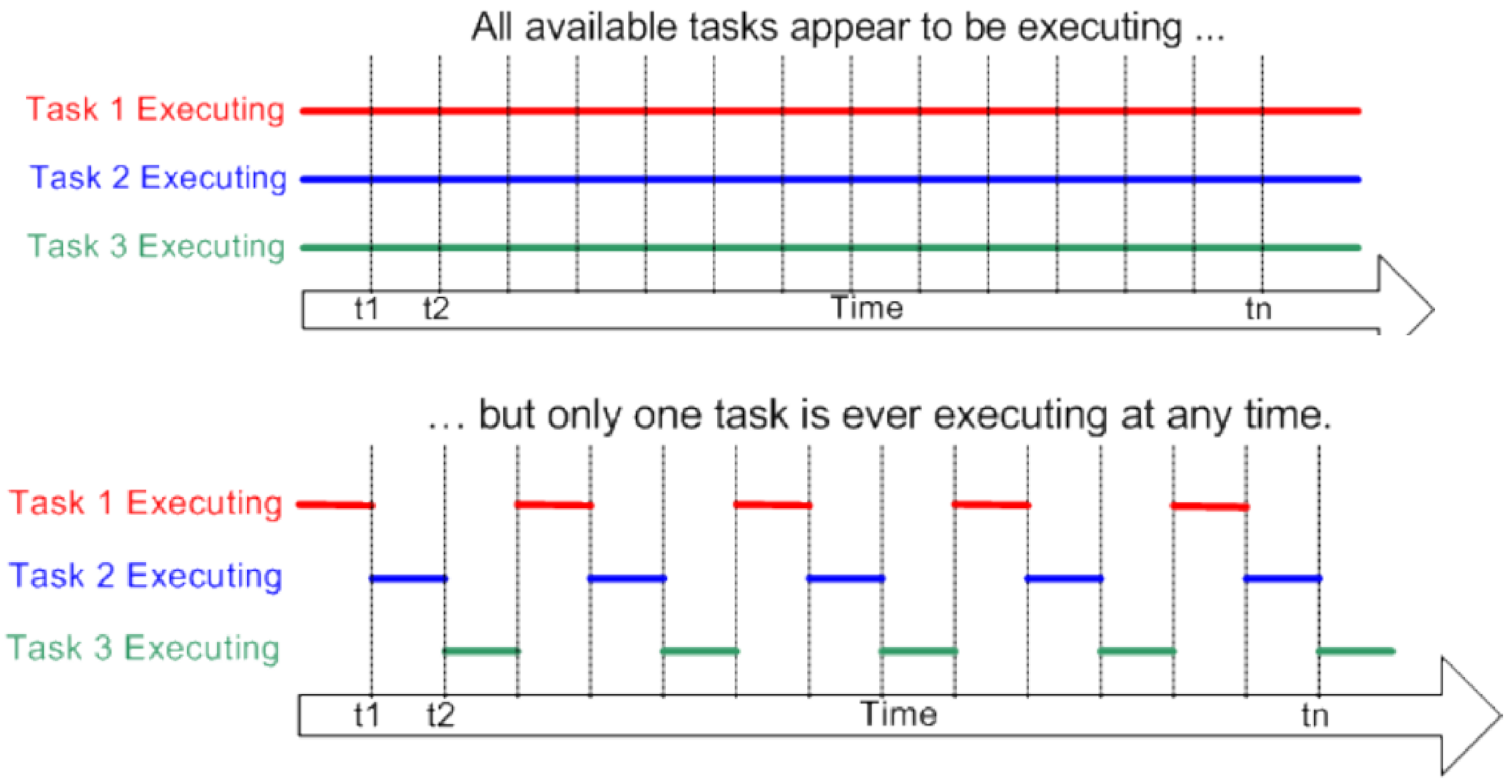



```
Counter = Counter + 1;
```

```
MOV   EAX, [Counter]     ; line 1
ADD   EAX, 1             ; line 2
MOV   [Counter], EAX     ; line 3
```

Benefícios de um prog. *Multitasking*

- Um processador convencional só pode executar uma única tarefa por vez. **Porém um sistema operacional *multitasking* pode fazer aparecer como se cada tarefa estivesse sendo executada simultaneamente.**

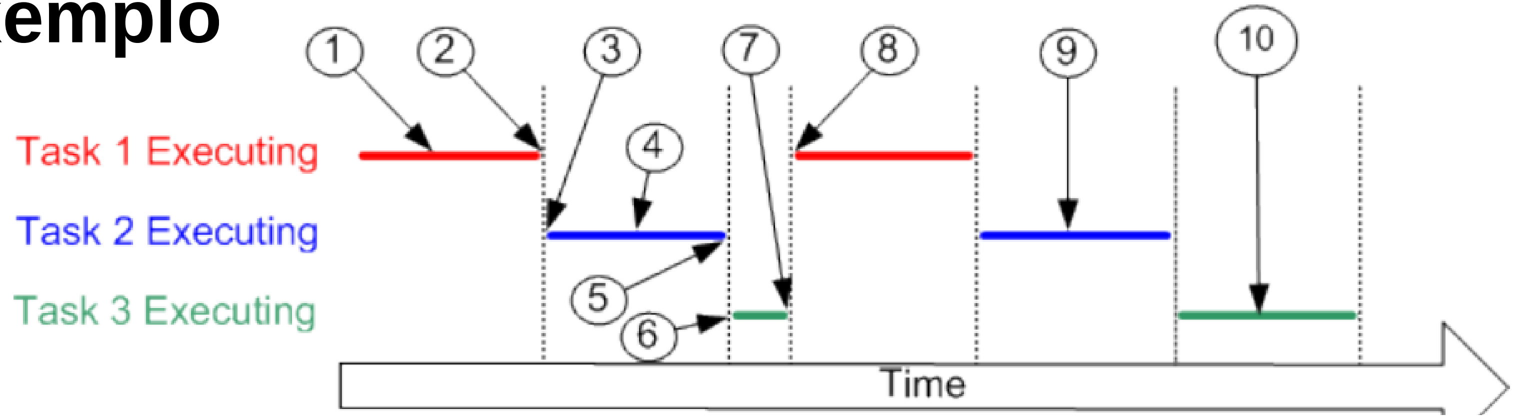


Scheduler

- **O *Scheduler* é a parte do kernel responsável por decidir qual tarefa deve ser executada na unidade de tempo.**
- **O kernel pode suspender e depois retomar uma tarefa muitas vezes durante a vida útil da tarefa.**

Scheduler

- Exemplo



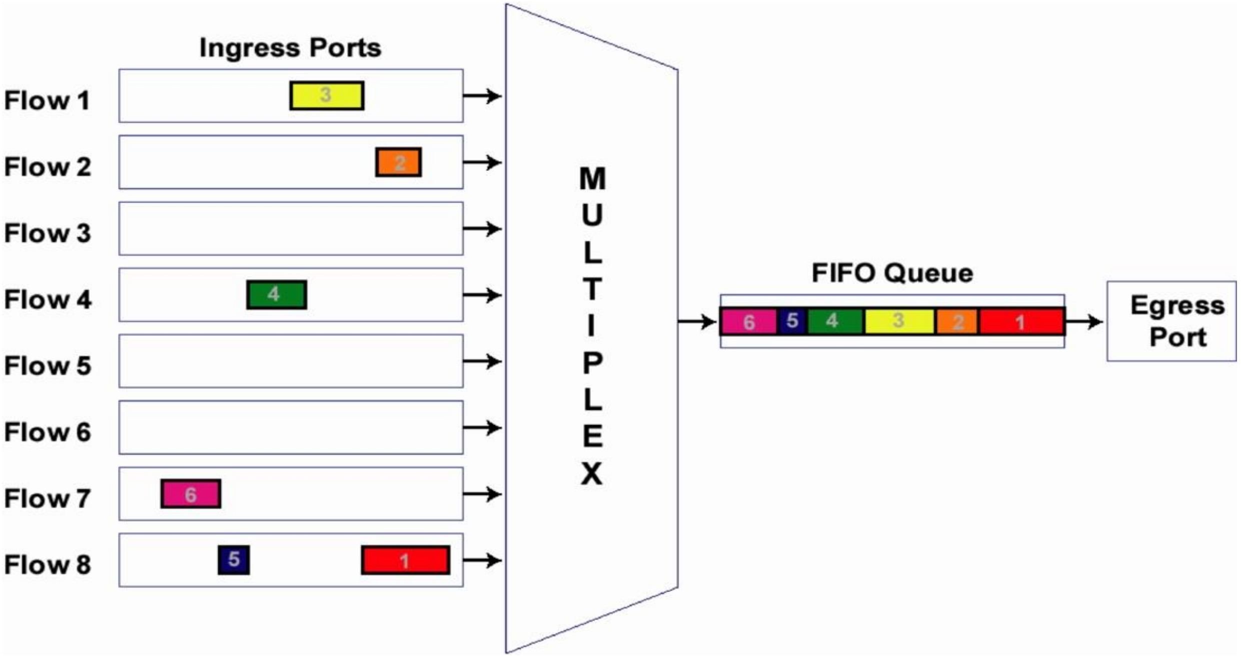
Referring to the numbers in the diagram above:

- At (1) task 1 is executing.
- At (2) the kernel suspends task 1 ...
- ... and at (3) resumes task 2.
- While task 2 is executing (4), it locks a processor peripheral for its own exclusive access.
- At (5) the kernel suspends task 2 ...
- ... and at (6) resumes task 3.
- Task 3 tries to access the same processor peripheral, finding it locked task 3 cannot continue so suspends itself at (7).
- At (8) the kernel resumes task 1.
- Etc.
- The next time task 2 is executing (9) it finishes with the processor peripheral and unlocks it.
- The next time task 3 is executing (10) it finds it can now access the processor peripheral and this time executes until suspended by the kernel.

http://www.freertos.org/implementation/a00005.html

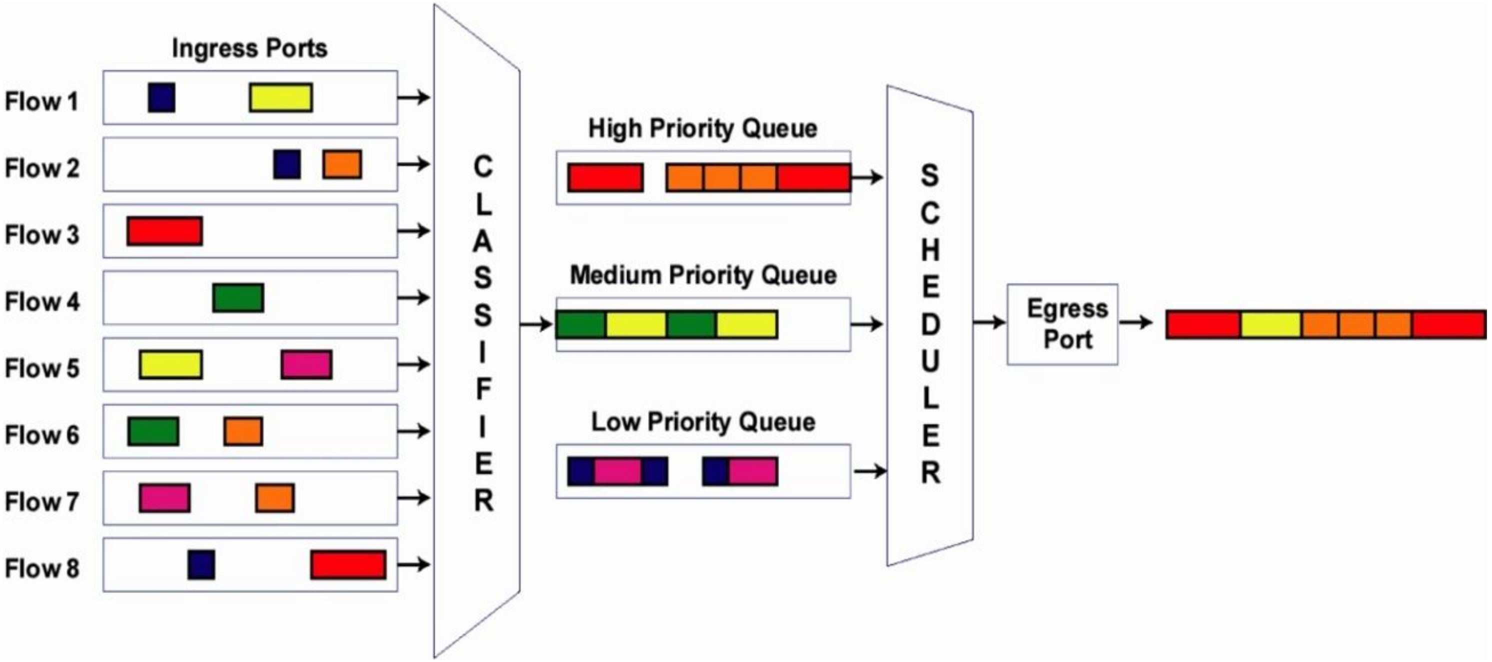
Filas e Scheduler

- **Em um sistema de fila única, todos os pacotes são colocados no *link* de saída obedecendo a uma ordem de distribuição definida no *Schedule* (buffer FIFO de saída).**



Filas e Scheduler (c/ Prioridade)

- **Frequentemente o SO diferenciam as tasks de acordo com suas prioridades, e para isso implementam um sistema de prioridades nas regras do *scheduler*.**



Scheduler (Prioridade)

Um sistema de filas c/ prioridades é composto por:

- **FIFO(s) separados em classes/prioridades.**
- **Pacotes de prioridade inferior iniciam a transmissão somente se nenhum pacote de prioridade mais alta estiver aguardando.**

Scheduler (tipos básicos)

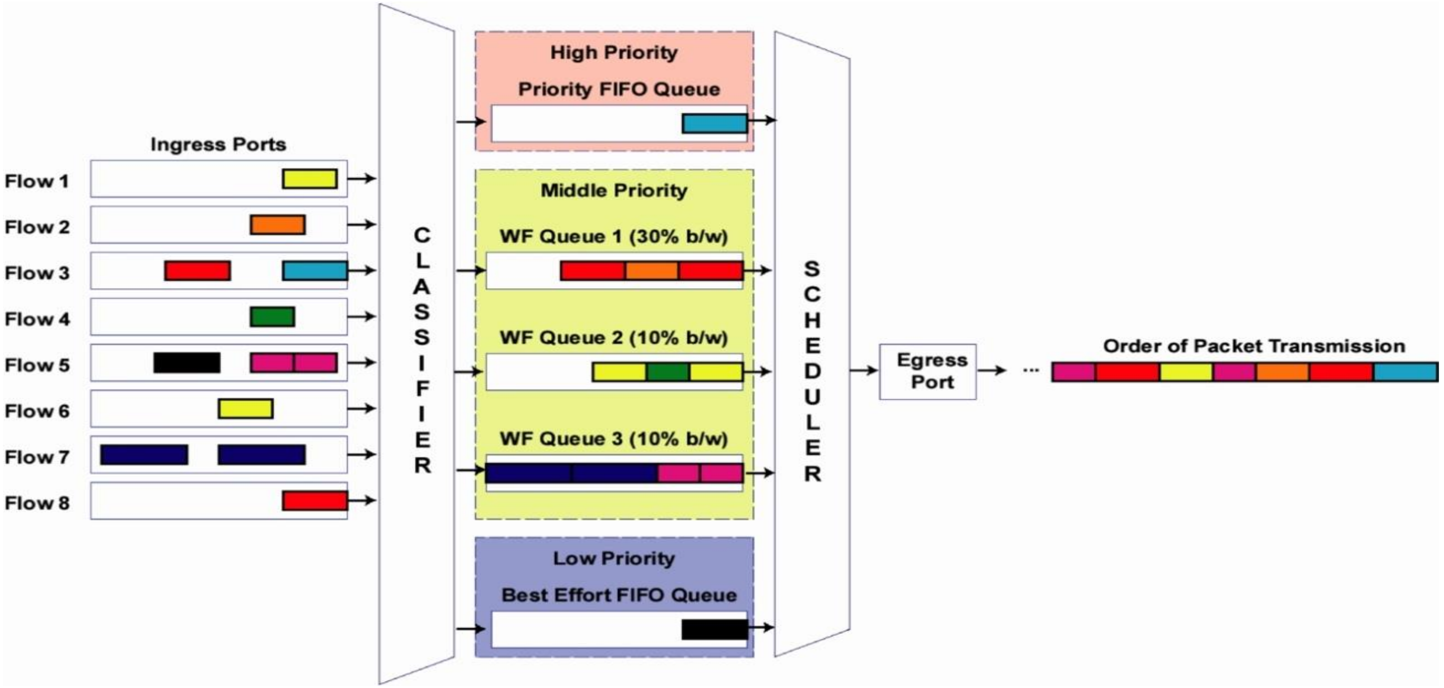
Um sistema de filas de prioridades é composto por:

- **Não preemptivo.** Quando um pacote de alta prioridade aguardar o término da transmissão de um pacote de baixa prioridade.

- **Preemptivo.** Quando um pacote de alta prioridade não precisar esperar, ou seja, mesmo que um pacote de baixa prioridade estiver em execução, este será interrompido e colocado na condição de espera, enquanto o pacote de alta prioridade é tratado.

Filas + Scheduler (tipos básicos)

- **Alguns algorítmos para *scheduler* mais complexos utilizam a combinação de vários esquemas de filas (Exemplo: Sistema LLQ (*Low Latency Queueing with Priority Percentage Support*).**



SENAI

FreeRTOS

Prof° Fernando Simplicio

RTOS

- **O *software* em tempo real deve fornecer resultados de computação sob restrições de tempo específicas da aplicação. Quando um resultado é disponibilizado muito tarde (ou muito cedo em alguns sistemas), o *software* falhou, mesmo que o resultado seja correto.**

- **Um *software* de tempo real NÃO precisa ser Multitarefa!**

RTOS Comercial

- Suportado por muitas plataformas de pequeno e médio desempenho.
- Os RTOS comercial amplamente testado por seus desenvolvedores e comunidade.

FreeRTOS

- **Sistema Operacional de Tempo Real.**
- **Gratuito e de Código Aberto.**
- **Desenvolvido pela Real Time Engineers Ltda.**
- **Possui serviços de gerenciamento de *tasks* e de memória.**
- **Desenvolvido para ser aplicado em MCUs de pequena capacidade de memória. (ARM7 ~4.3Kbytes).**
- **Desenvolvido em Linguagem C (podendo ser compilado no GCC, Borland C++, etc).**
- **Suporta ilimitado número de tasks e prioridades (depende do hardware utilizado).**

FreeRTOS

- **Infelizmente NÃO acompanha drivers/interfaces de comunicação de Rede, drivers ou arquivos (FileSystem) - (para algumas famílias de MCUs é *free*), .**
- **Implementa serviços de Filas (queues), semáforos binários, contadores, mutexes e *software timers* .**
- **Suporta atualmente mais de 38 diferentes arquiteturas.**

FreeRTOS

- **Distribuição do FreeRTOS**

```
FreeRTOS
 │    │
 │    ├─Source   Directory containing the FreeRTOS source files
 │    │
 │    └─Demo     Directory containing pre-configured and port specific FreeRTOS demo projects
 │
FreeRTOS-Plus
      │
      ├─Source   Directory containing source code for some FreeRTOS+ ecosystem components
      │
      └─Demo     Directory containing demo projects for FreeRTOS+ ecosystem components
```

FreeRTOS

- **Arquivos comuns para todos os *ports* com FreeRTOS.**

```
FreeRTOS
    │
    └─Source
        │
        ├─tasks.c         FreeRTOS source file - always required
        ├─list.c          FreeRTOS source file - always required
        ├─queue.c         FreeRTOS source file - nearly always required
        ├─timers.c        FreeRTOS source file - optional
        ├─event_groups.c  FreeRTOS source file - optional
        └─croutine.c      FreeRTOS source file - optional
```

FreeRTOS

- **Arquivos de Inclusão referente ao *port* no FreeRTOS.**

```
FreeRTOS
  │
  └─Source
      │
      └─portable Directory containing all port specific source files
          │
          ├─MemMang Directory containing the 5 alternative heap allocation source files
          │
          ├─[compiler 1] Directory containing port files specific to compiler 1
          │   │
          │   ├─[architecture 1] Contains files for the compiler 1 architecture 1 port
          │   ├─[architecture 2] Contains files for the compiler 1 architecture 2 port
          │   └─[architecture 3] Contains files for the compiler 1 architecture 3 port
          │
          └─[compiler 2] Directory containing port files specific to compiler 2
              │
              ├─[architecture 1] Contains files for the compiler 2 architecture 1 port
              ├─[architecture 2] Contains files for the compiler 2 architecture 2 port
              └─[etc.]
```

FreeRTOS

- **Vai iniciar com FreeRTOS? Então comece por um projeto demo!**

```
FreeRTOS
    │
    └─Demo          Directory containing all the demo projects
        │
        ├─[Demo x]  Contains the project file that builds demo 'x'
        │
        ├─[Demo y]  Contains the project file that builds demo 'y'
        │
        ├─[Demo z]  Contains the project file that builds demo 'z'
        │
        └─Common    Contains files that are built by all the demo applications
```

FreeRTOS

A. Variables

   i. Variables of type *char* are prefixed *c*
   ii. Variables of type *short* are prefixed *s*
   iii. Variables of type *long* are prefixed *l*
   iv. Enumerated variables are prefixed *e*
   v. Other types (e.g. structs) are prefixed *x*
   vi. Pointers have an additional prefixed *p*, for example a pointer to a short will have prefix *ps*
   vii. Unsigned variables have an additional prefixed *u*, for example an unsigned short will have prefix *us*, and a pointer to an unsigned short will have prefix *pus*.

B. Functions

   i. File private functions are prefixed with *prv* File private functions are prefixed with *prv*
   ii. API functions are prefixed with their return type, as per the convention defined for variables
   iii. Function names start with the file in which they are defined. For example *vTaskDelete* is defined in the tasks.c file, and has a void return type. A function prefix of *pv* is a pointer to a function that returns a *void*.

C. Macros

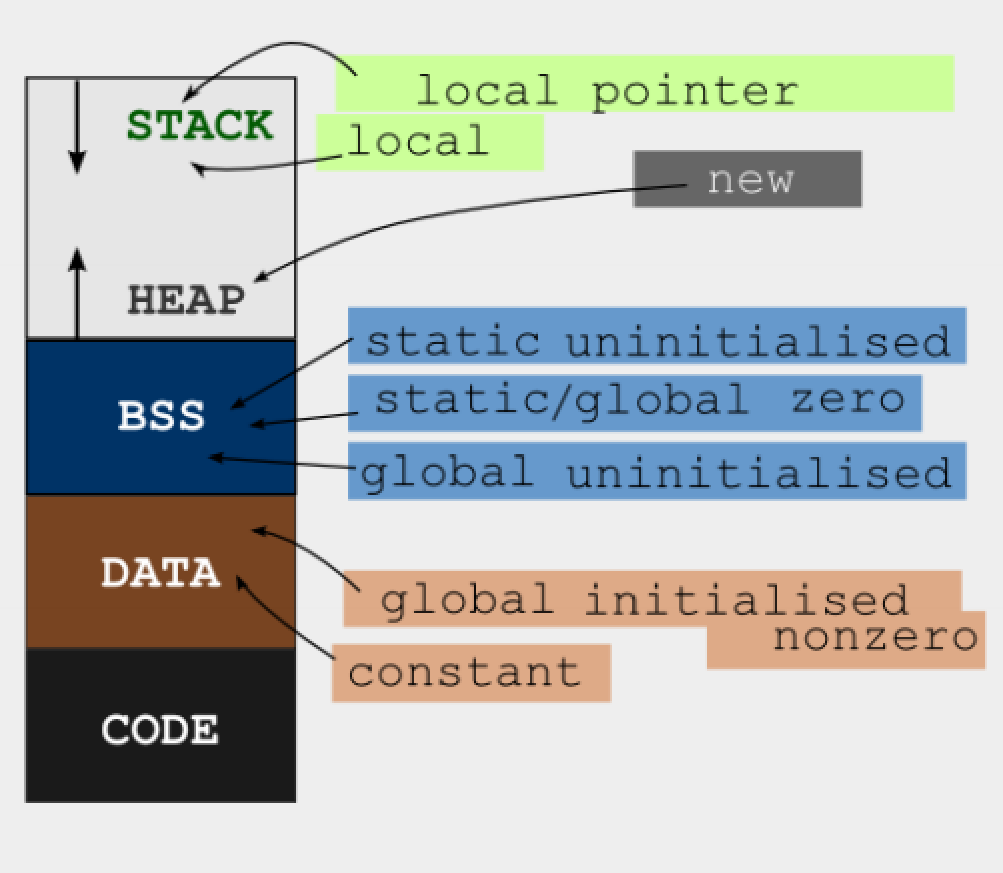
   i. Macros are pre-fixed with the file in which they are defined. The pre-fix is lower case. For example, configUSE_PREEMPTION is defined in FreeRTOSConifg.h.
   ii. Other than the pre-fix, macros are written in all upper case, and use an underscore to separate words.

FreeRTOS

- **Exemplo:**

- **vTaskPrioritySet()**
- **xQueueReceive()**
- **pvTimerGetTimerID()**

Alocação Estática e Dinâmica



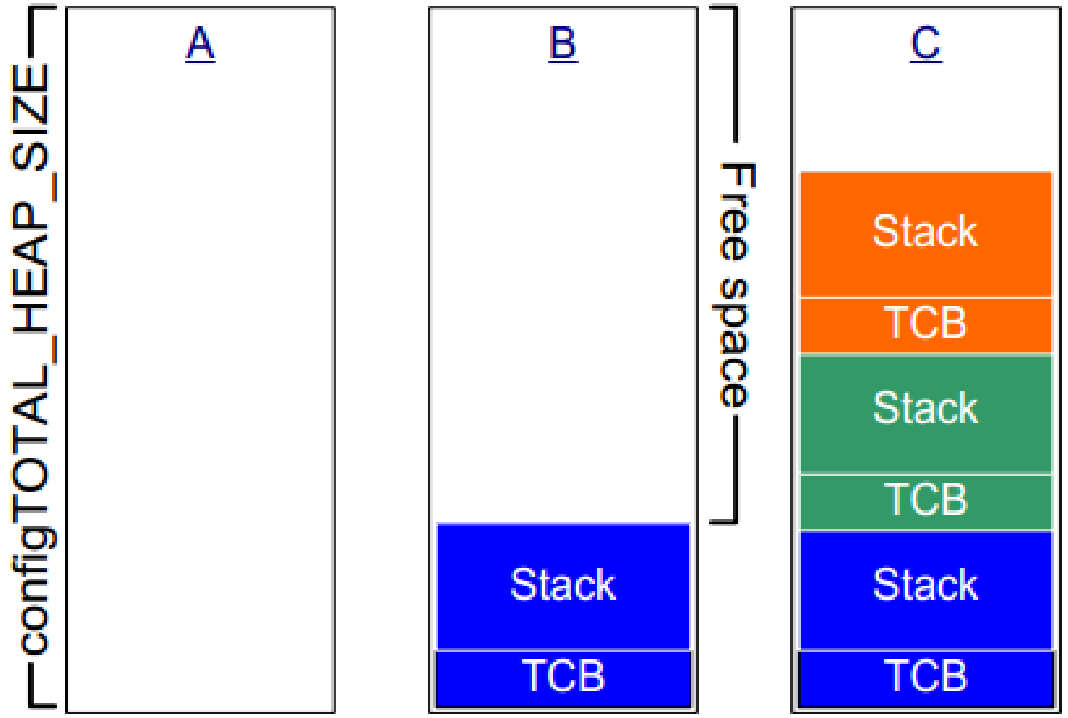
Fonte: http://wiki.darshansonde.com/index.php?title=CppMemoryLayout

Malloc() e Free()

- **Consumo considerável de memória para sua implementação.**
- **Raramente são do tipo *Thread-Safe*.**
- **Não são determinísticos (O tempo de execução das funções será diferente a cada chamada).**
- **Pode fragmentar a memória.**

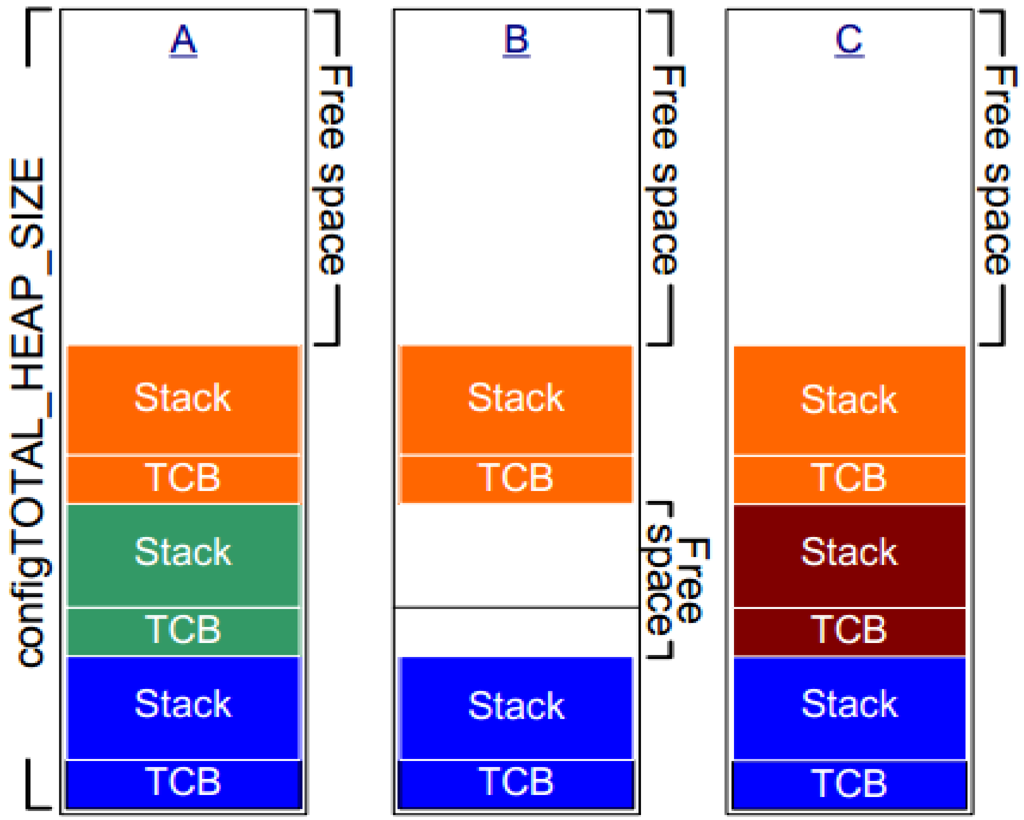
pvPortMalloc() e vPortFree()

- Heap_1.c



pvPortMalloc() e vPortFree()

- **Heap_2.c**



pvPortMalloc() e vPortFree()

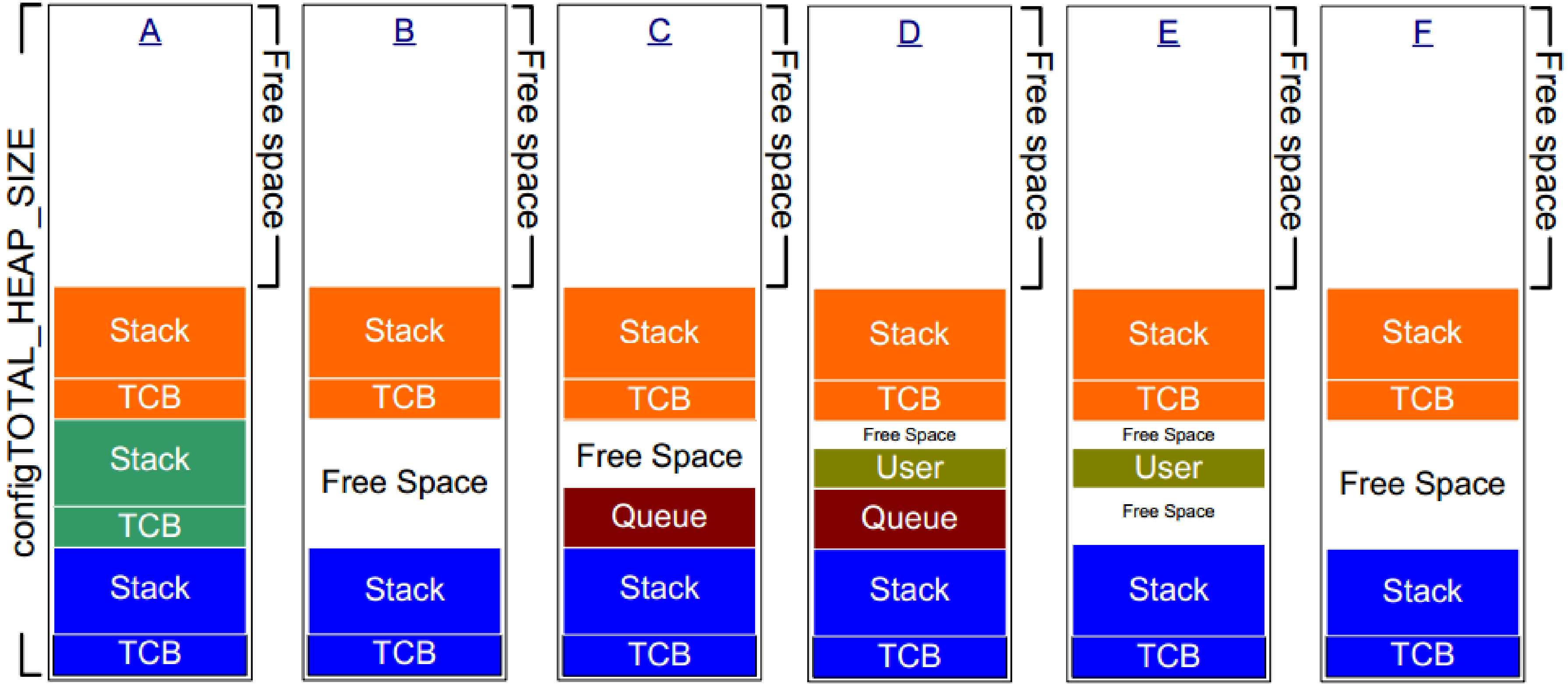
- **Heap_3.c**

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

	vTaskSuspendAll();
	{
		pvReturn = malloc( xWantedSize );
	}
	xTaskResumeAll();

	#if( configUSE_MALLOC_FAILED_HOOK == 1 )
	{
		if( pvReturn == NULL )
		{
			extern void vApplicationMallocFailedHook( void );
			vApplicationMallocFailedHook();
		}
	}
	#endif

	return pvReturn;
}
```

```
void vPortFree( void *pv )
{
	if( pv )
	{
		vTaskSuspendAll();
		{
			free( pv );
		}
		xTaskResumeAll();
	}
}
```

pvPortMalloc() e vPortFree()

- **Heap_4.c**



Tasks no FreeRTOS

Prof° Fernando Simplicio

Tasks

```c
int main( void )
{
    /* Create one of the two tasks. */
    xTaskCreate(    vTask1,     /* Pointer to the function that implements the task. */
                    "Task 1",   /* Text name for the task.  This is to facilitate debugging only. */
                    1000,       /* Stack depth - most small microcontrollers will use much less stack than this. */
                    NULL,       /* We are not using the task parameter. */
                    1,          /* This task will run at priority 1. */
                    NULL );     /* We are not using the task handle. */

    /* Create the other task in exactly the same way. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler to start the tasks executing. */
    vTaskStartScheduler();

    /* The following line should never be reached because vTaskStartScheduler()
    will only return if there was not enough FreeRTOS heap memory available to
    create the Idle and (if configured) Timer tasks.  Heap management, and
    techniques for trapping heap exhaustion, are described in the book text. */
    for( ;; );
    return 0;
}
```

Tasks

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile uint32_t ul;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

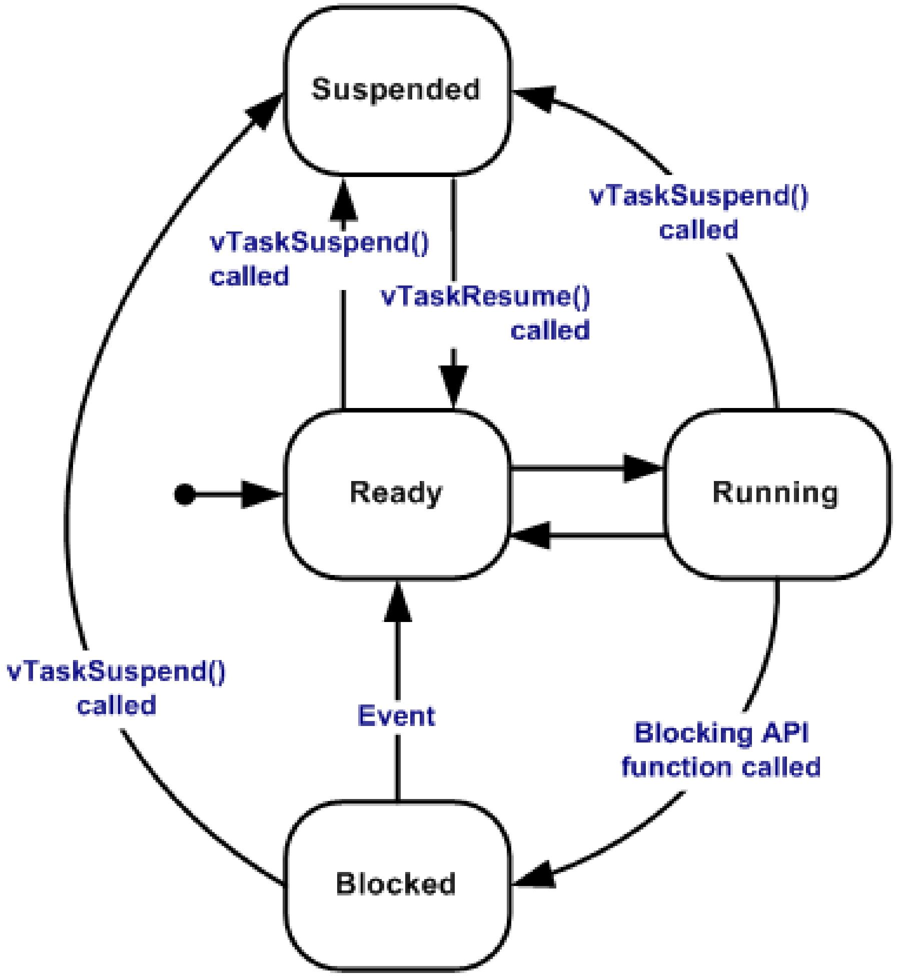
Tasks

```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile uint32_t ul;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT/2; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Estados das *Task no* FreeRTOS

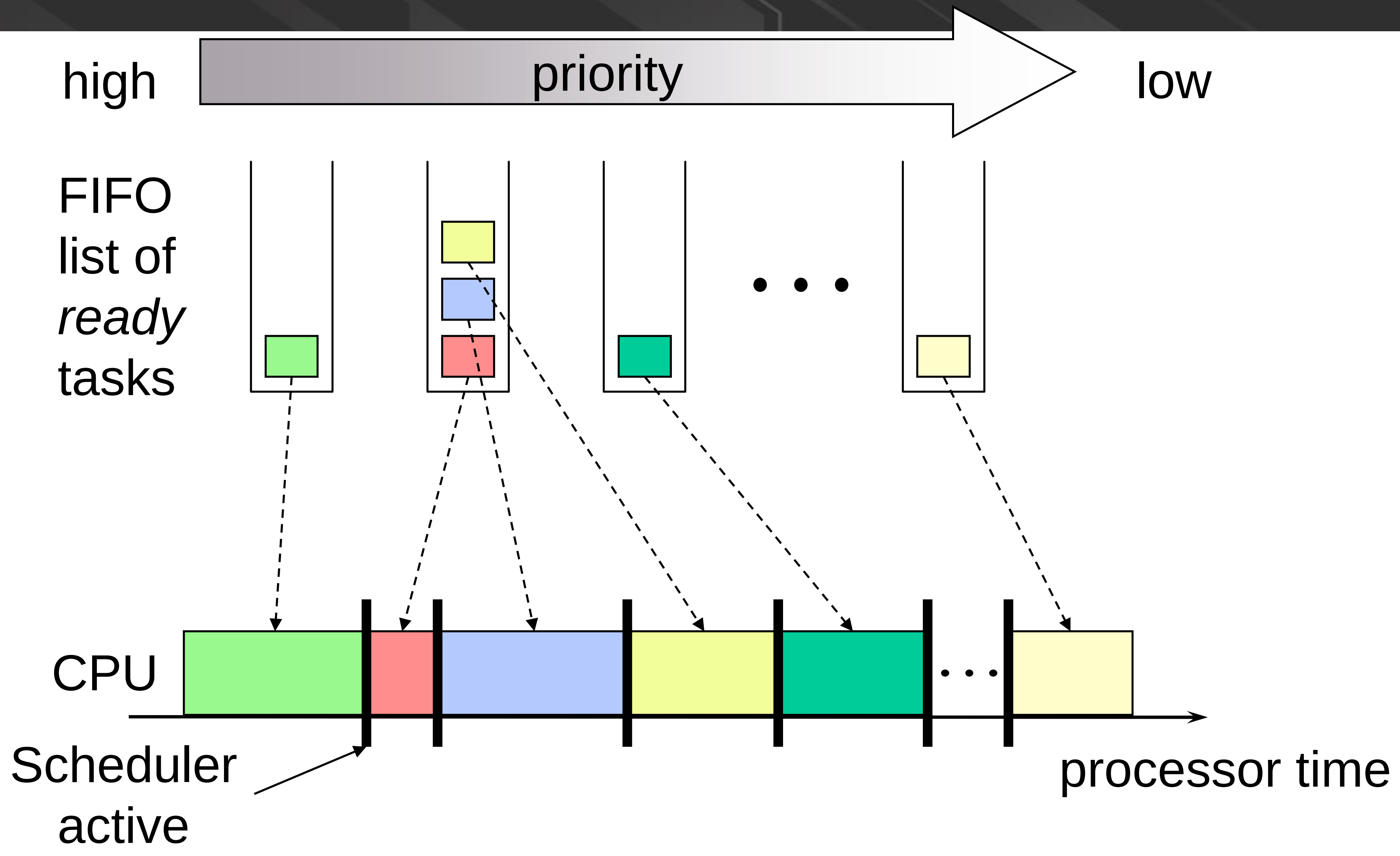
Prof° Fernando Simplicio

Task States

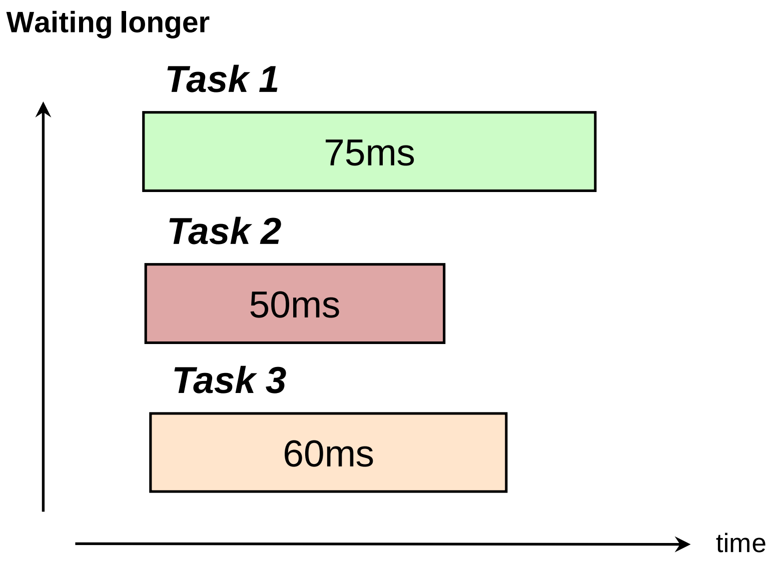


Valid task state transitions

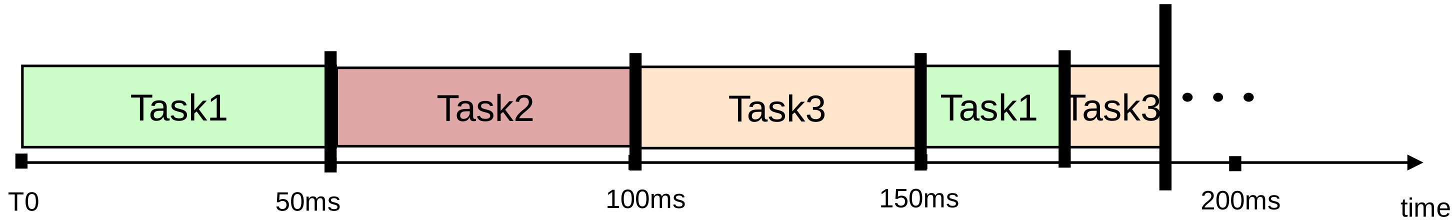
Priority Based FIFO Scheduling



Round-Robin Scheduling



Time Slice = 50ms



Laboratório com FreeRTOS

Prof° Fernando Simplicio

Prioridades das Tasks no FreeRTOS

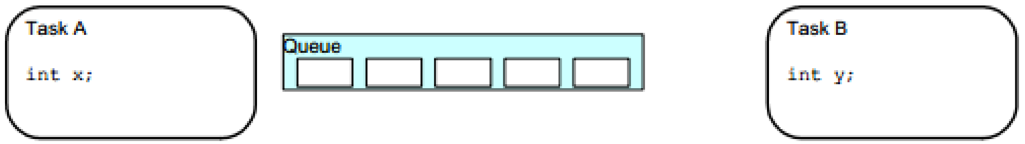
Prof° Fernando Simplicio

Fila (Queue) no FreeRTOS

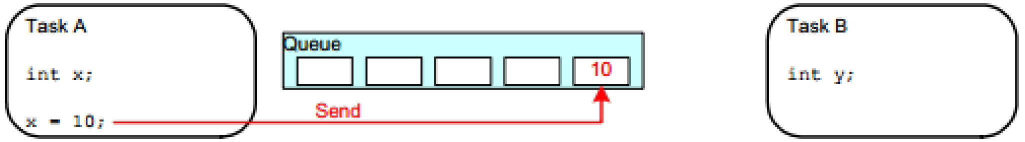
Prof° Fernando Simplicio

Filas no FreeRTOS

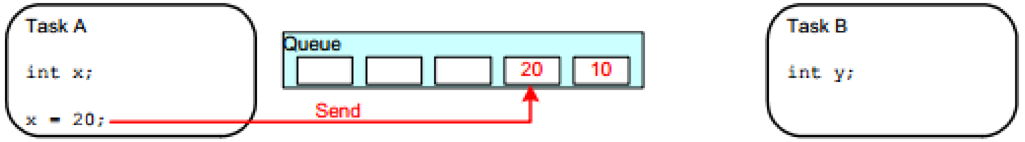
- **Modelo FIFO (*Fist In Fist Out*)**
- **O tipo e o tamanho da fila é definido quando em criação.**
- **Fila – passagem por Cópia: ao enviar um byte para a fila, este byte será copiado, e ao ler um byte da fila, este byte também será copiado.**
- **Fila – passagem por Referência: é passado o endereço de uma estrutura para a fila (cria-se uma fila de ponteiros).**
- **Qualquer task pode ler e escrever numa mesma fila.**



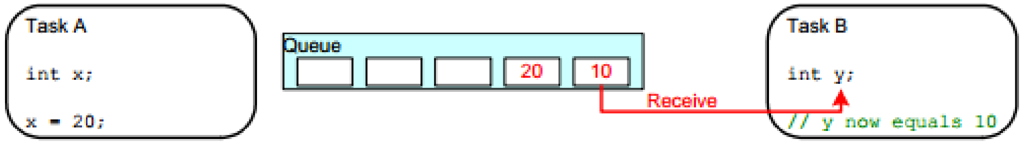
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



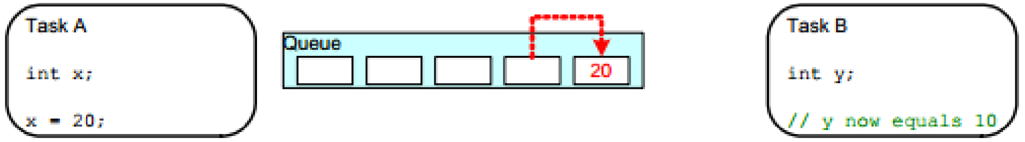
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Criar uma Fila no FreeRTOS

```
xQueueHandle xQueueCreate( unsigned portBASE_TYPE uxQueueLength,
                           unsigned portBASE_TYPE uxItemSize
                         );
```

```
/* Declare a variable of type xQueueHandle.  This is used to store the handle
to the queue that is accessed by all three tasks. */
xQueueHandle xQueue;


int main( void )
{
    /* The queue is created to hold a maximum of 5 values, each of which is
    large enough to hold a variable of type long. */
    xQueue = xQueueCreate( 5, sizeof( long ) );

    if( xQueue != NULL )
    {
```

Gravar em uma Fila no FreeRTOS

```
portBASE_TYPE xQueueSendToFront(   xQueueHandle xQueue,
                                   const void * pvItemToQueue,
                                   portTickType xTicksToWait
                                );
```

```
portBASE_TYPE xQueueSendToBack(    xQueueHandle xQueue,
                                   const void * pvItemToQueue,
                                   portTickType xTicksToWait
                                );
```

```
static void vSenderTask( void *pvParameters )
{
long lValueToSend;
portBASE_TYPE xStatus;

    /* Two instances of this task are created so the value that is sent to the
    queue is passed in via the task parameter - this way each instance can use
    a different value.  The queue was created to hold values of type long,
    so cast the parameter to the required type. */
    lValueToSend = ( long ) pvParameters;

    xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );
```

Ler uma Fila do FreeRTOS

```
portBASE_TYPE xQueueReceive(
                                xQueueHandle xQueue,
                                const void * pvBuffer,
                                portTickType xTicksToWait
                            );
```

```
portBASE_TYPE xQueuePeek(
                                xQueueHandle xQueue,
                                const void * pvBuffer,
                                portTickType xTicksToWait
                            );
```

```
xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );

if( xStatus == pdPASS )
{
    /* Data was successfully received from the queue, print out the received
    value. */
    vPrintStringAndNumber( "Received = ", lReceivedValue );
}
else
{
    /* Data was not received from the queue even after waiting for 100ms.
    This must be an error as the sending tasks are free running and will be
    continuously writing to the queue. */
    vPrintString( "Could not receive from the queue.\n" );
}
```

Qtd de Dados na Fila do FreeRTOS

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

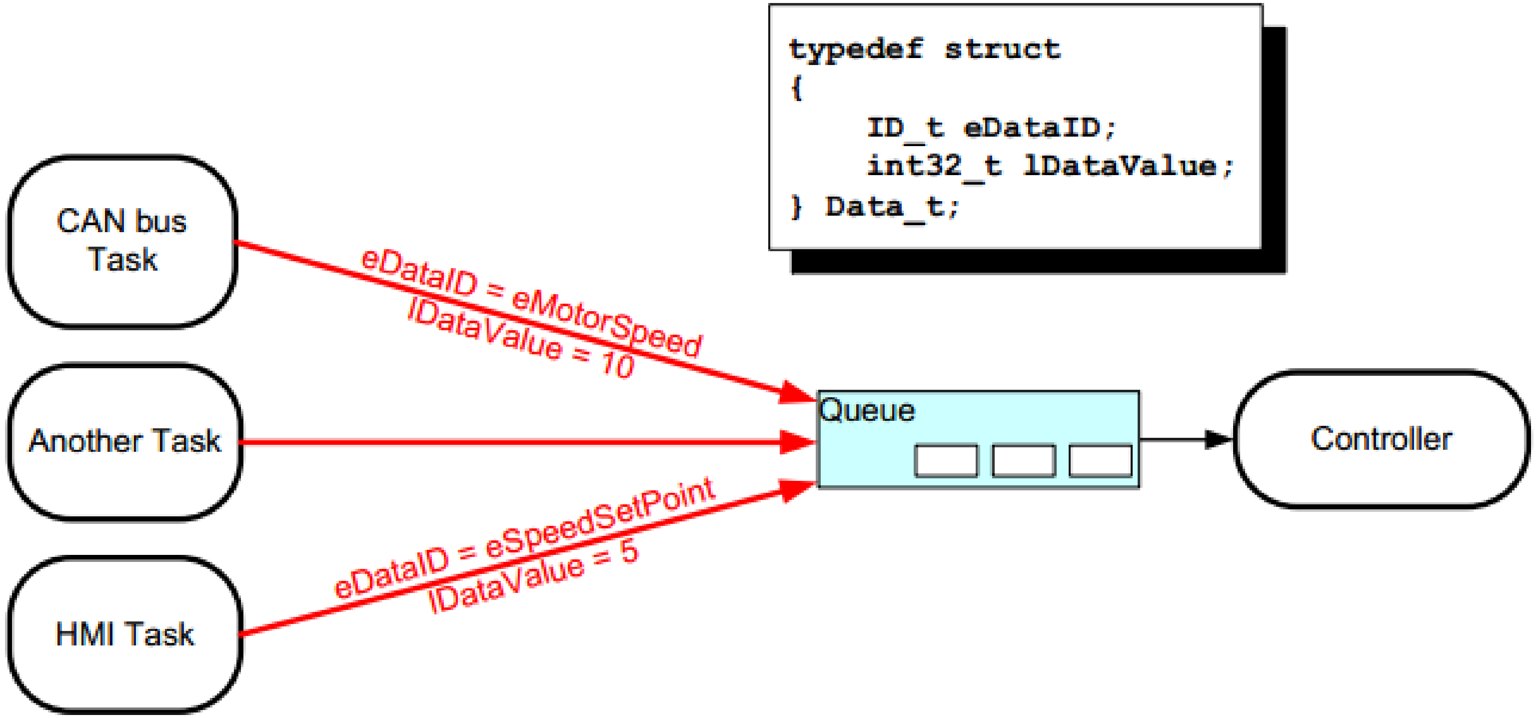
```
/* This call should always find the queue empty because this task will
immediately remove any data that is written to the queue. */
if( uxQueueMessagesWaiting( xQueue ) != 0 )
{
    vPrintString( "Queue should have been empty!\n" );
}
```

Fila (Queue) no FreeRTOS

via ponteiros

Prof° Fernando Simplicio

Filas no FreeRTOS



Passagem por Referência (ponteiros)

```
/* Declare a variable of type QueueHandle_t to hold the handle of the queue being created. */
QueueHandle_t xPointerQueue;

/* Create a queue that can hold a maximum of 5 pointers, in this case character pointers. */
xPointerQueue = xQueueCreate( 5, sizeof( char * ) );


/* A task that obtains a buffer, writes a string to the buffer, then sends the address of the
buffer to the queue created in Listing 52. */
void vStringSendingTask( void *pvParameters )
{
char *pcStringToSend;
const size_t xMaxStringLength = 50;
BaseType_t xStringNumber = 0;

    for( ;; )
    {
        /* Obtain a buffer that is at least xMaxStringLength characters big.  The implementation
        of prvGetBuffer() is not shown – it might obtain the buffer from a pool of pre-allocated
        buffers, or just allocate the buffer dynamically. */
        pcStringToSend = ( char * ) prvGetBuffer( xMaxStringLength );

        /* Write a string into the buffer. */
        snprintf( pcStringToSend, xMaxStringLength, "String number %d\r\n", xStringNumber );

        /* Increment the counter so the string is different on each iteration of this task. */
        xStringNumber++;

        /* Send the address of the buffer to the queue that was created in Listing 52. The
        address of the buffer is stored in the pcStringToSend variable.*/
        xQueueSend( xPointerQueue,   /* The handle of the queue. */
                    &pcStringToSend, /* The address of the pointer that points to the buffer. */
                    portMAX_DELAY );
    }
}
```

Passagem por Referência (ponteiros)

```
/* A task that receives the address of a buffer from the queue created in Listing 52, and
written to in Listing 53.  The buffer contains a string, which is printed out. */
void vStringReceivingTask( void *pvParameters )
{
char *pcReceivedString;

    for( ;; )
    {
        /* Receive the address of a buffer. */
        xQueueReceive( xPointerQueue,     /* The handle of the queue. */
                       &pcReceivedString, /* Store the buffer's address in pcReceivedString. */
                       portMAX_DELAY );

        /* The buffer holds a string, print it out. */
        vPrintString( pcReceivedString );

        /* The buffer is not required any more - release it so it can be freed, or re-used. */
        prvReleaseBuffer( pcReceivedString );
    }
}
```

Mutex no FreeRTOS

Prof° Fernando Simplicio

Mutex no FreeRTOS

- **Mutex = *MUTual Exclusion***
- **Habilitado através de configUSE_MUTEXES (1) em FreeRTOSConfig.h**

Mutex no FreeRTOS

- **xSemaphoreCreateMutex()**
- **xSemaphoreTake() e xSemaphoreGive()**

```
if( xSemaphoreTake( xMutex, xDelay100ms ) == pdTRUE )
{
    UART2_WriteStr((char*)pcString);
    xSemaphoreGive( xMutex );
}
```

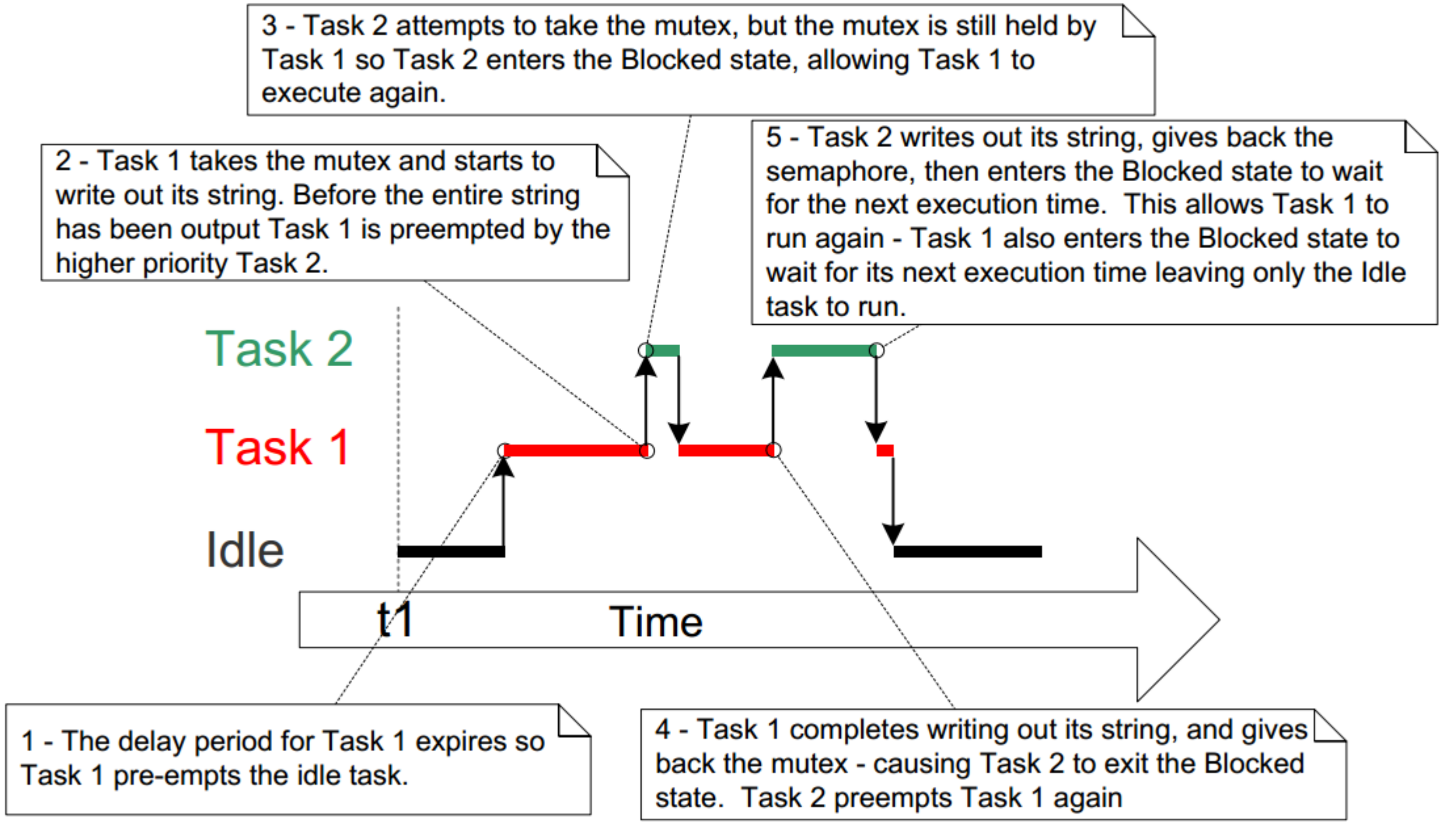
Mutex no FreeRTOS

- xSemaphoreCreateMutex()
- xSemaphoreTake() e xSemaphoreGive()

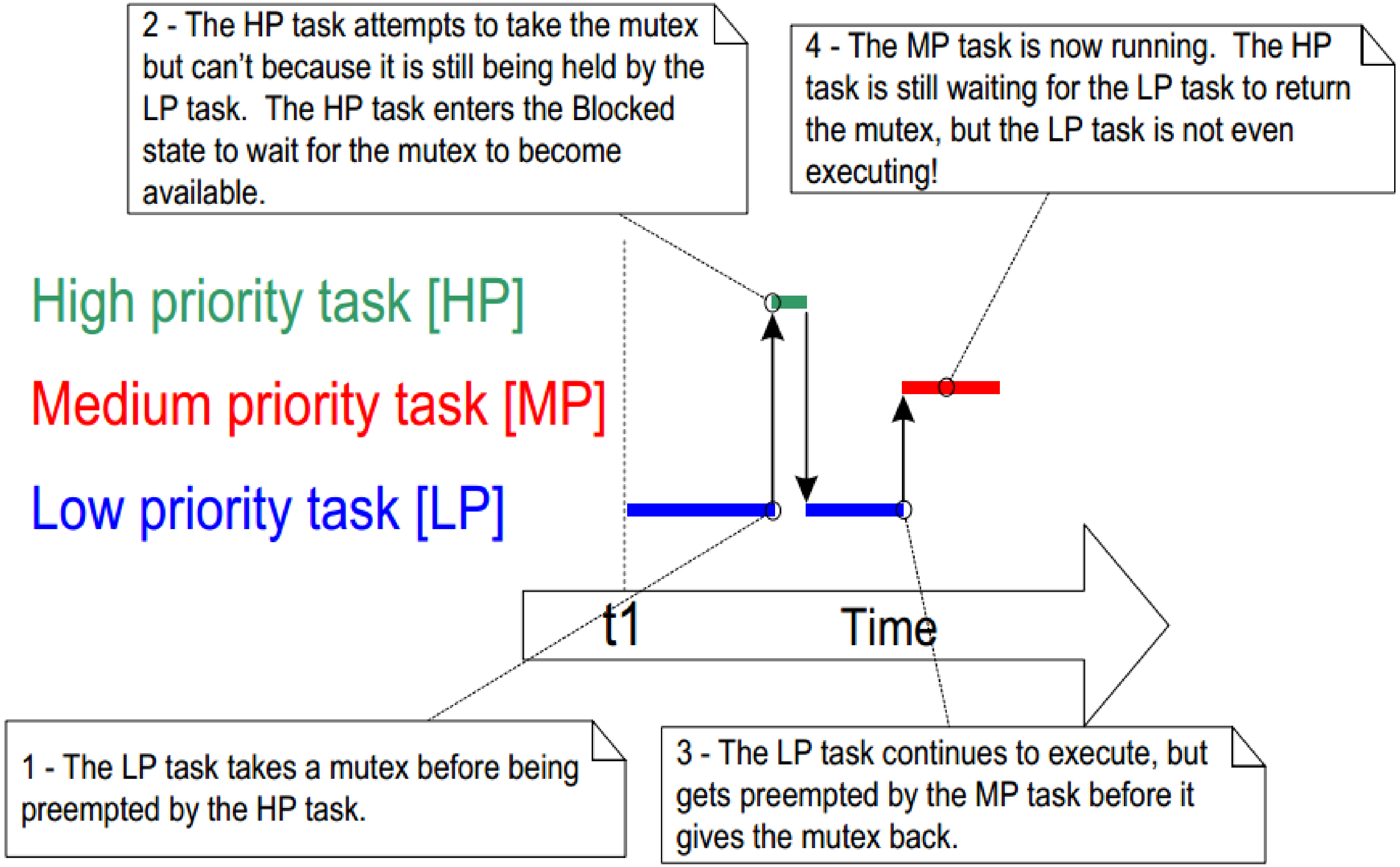
```
static void prvNewPrintString( const char *pcString )
{
    /* The mutex is created before the scheduler is started, so already exists by the
    time this task executes.

    Attempt to take the mutex, blocking indefinitely to wait for the mutex if it is
    not available straight away.  The call to xSemaphoreTake() will only return when
    the mutex has been successfully obtained, so there is no need to check the
    function return value.  If any other delay period was used then the code must
    check that xSemaphoreTake() returns pdTRUE before accessing the shared resource
    (which in this case is standard out).  As noted earlier in this book, indefinite
    time outs are not recommended for production code. */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* The following line will only execute once the mutex has been successfully
        obtained.  Standard out can be accessed freely now as only one task can have
        the mutex at any one time. */
        printf( "%s", pcString );
        fflush( stdout );

        /* The mutex MUST be given back! */
    }
    xSemaphoreGive( xMutex );
}
```

Mutex no FreeRTOS

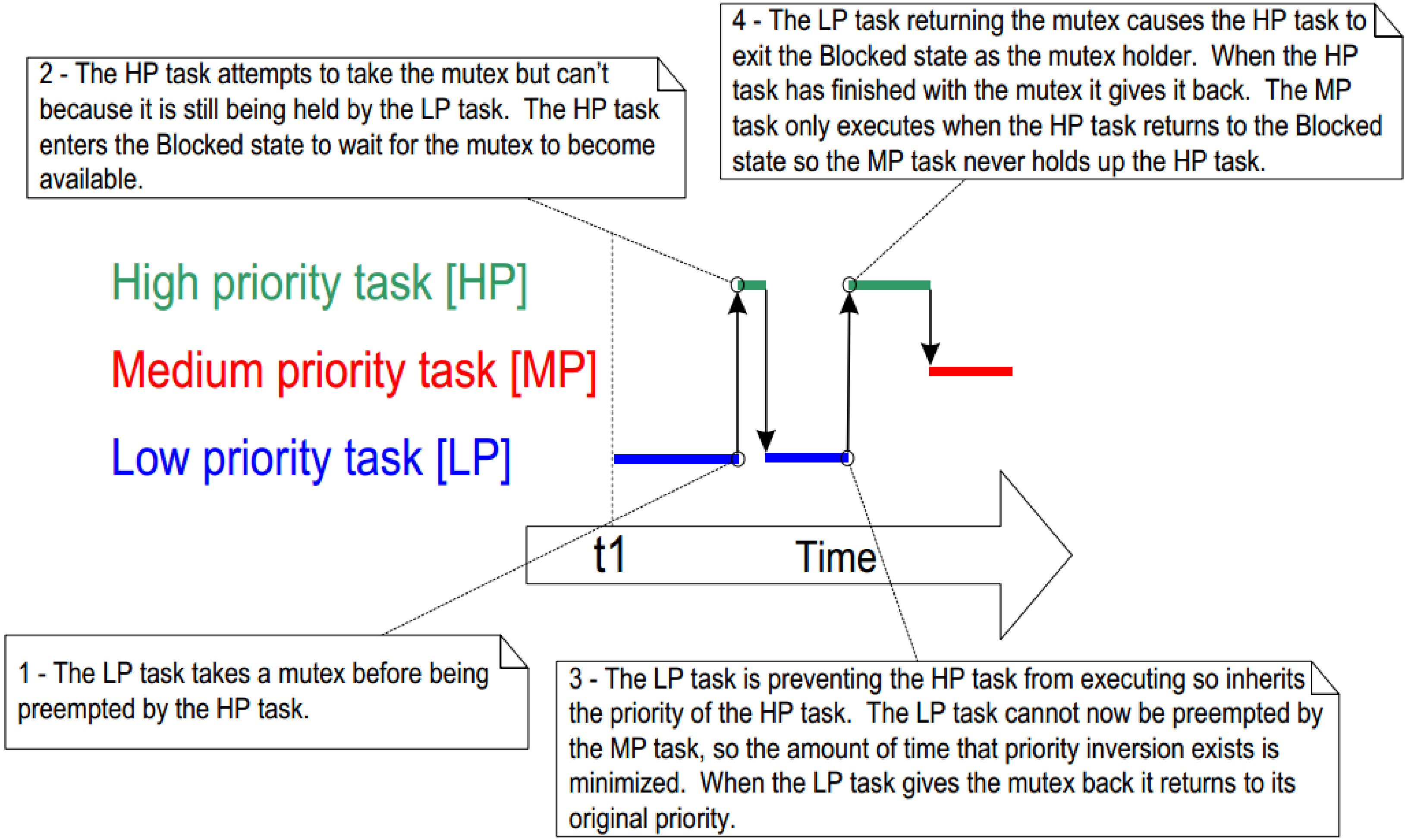
- **xSemaphoreCreateMutex()**
- **xSemaphoreTake() e xSemaphoreGive()**



Inversão de Prioridades



Mutex no FreeRTOS (Inversão de Polaridade)



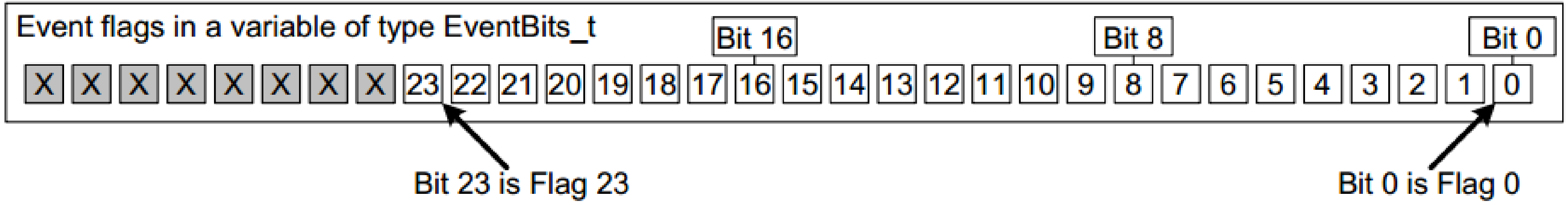
Event Groups no FreeRTOS

Prof° Fernando Simplicio

Event Group do FreeRTOS

- **Flags (bits) de Sinalização.**
- **Recurso utilizado para sincronização das Tasks.**
- **Caso configUSE_16_BIT_TICKS (1), event group conterá 8 bits para sinalização.**
- **Caso configUSE_16_BIT_TICKS (0), event group conterá 24 bits para sinalização.**

EventBits_t.



ISR xHigherPriorityTaskWoken

Prof° Fernando Simplicio

Semáforo Contador

```
#define LONG_TIME 0xffff
#define TICKS_TO_WAIT    10

SemaphoreHandle_t xSemaphore = NULL;

/* Repetitive task. */
void vATask( void * pvParameters )
{
    /* We are using the semaphore for synchronisation so we create a binary
    semaphore rather than a mutex.  We must make sure that the interrupt
    does not attempt to use the semaphore before it is created! */
    xSemaphore = xSemaphoreCreateBinary();

    for( ;; )
    {
        /* We want this task to run every 10 ticks of a timer.  The semaphore
        was created before this task was started.

        Block waiting for the semaphore to become available. */
        if( xSemaphoreTake( xSemaphore, LONG_TIME ) == pdTRUE )
        {
            /* It is time to execute. */

            ...

            /* We have finished our task.  Return to the top of the loop where
            we will block on the semaphore until it is time to execute
            again.  Note when using the semaphore for synchronisation with an
            ISR in this manner there is no need to 'give' the semaphore
            back. */
        }
    }
}
```

Semáforo Contador

```
/* Timer ISR */
void vTimerISR( void * pvParameters )
{
static unsigned char ucLocalTickCount = 0;
static signed BaseType_t xHigherPriorityTaskWoken;

    /* A timer tick has occurred. */

    ... Do other time functions.

    /* Is it time for vATask() to run? */
    xHigherPriorityTaskWoken = pdFALSE;
    ucLocalTickCount++;
    if( ucLocalTickCount >= TICKS_TO_WAIT )
    {
        /* Unblock the task by releasing the semaphore. */
        xSemaphoreGiveFromISR( xSemaphore, &xHigherPriorityTaskWoken );

        /* Reset the count so we release the semaphore again in 10 ticks
        time. */
        ucLocalTickCount = 0;
    }

    /* If xHigherPriorityTaskWoken was set to true you
    we should yield.  The actual macro used here is
    port specific. */
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

taskSuspend()
taskResume()

Prof° Fernando Simplicio

taskSuspend() | taskResume()

```
/* Task4 with priority 4 */
static void MyTask4(void* pvParameters)
{
    Serial.println(F("Task4 Running, Suspending all tasks"));
    vTaskSuspend(TaskHandle_2); //Suspend Task2/3
    vTaskSuspend(TaskHandle_3);
    vTaskSuspend(NULL); //Suspend Own Task

    Serial.println(F("Back in Task4, Deleting Itself"));
    vTaskDelete(TaskHandle_4);
}
```

```
/* Task1 with priority 1 */
static void MyTask1(void* pvParameters)
{
    Serial.println(F("Task1 Resuming Task2"));
    vTaskResume(TaskHandle_2);

    Serial.println(F("Task1 Resuming Task3"));
    vTaskResume(TaskHandle_3);

    Serial.println(F("Task1 Resuming Task4"));
    vTaskResume(TaskHandle_4);

    Serial.println(F("Task1 Deleting Itself"));
    vTaskDelete(TaskHandle_1);
}
```

FreeRTOSconfig.h

Config definition	Description
configUSE_PREEMPTION	Set to 1 to use the preemptive RTOS scheduler, or 0 to use the cooperative RTOS scheduler
configUSE_IDLE_HOOK	Enable/disable IDLE Hook (callback when system has no active task)
configUSE_TICK_HOOK	Enable/disable TICK Hook (callback on every tick)
configCPU_CLOCK_HZ	Defines CPU clock for tick generation
configTICK_RATE_HZ	Defines Tick Frequency in Hertz
configMAX_PRIORITIES	Defines the number priority level that kernel need to manage
configMINIMAL_STACK_SIZE	Defines the minimal stack size allocated to a task
configTOTAL_HEAP_SIZE	Defines the size of the system heap
configMAX_TASK_NAME_LEN	Defines the Maximum Task name length (used for debug)
configUSE_TRACE_FACILITY	Build/omit Trace facility (used for debug)

FreeRTOSconfig.h

Config definition	Description
configUSE_16_BIT_TICKS	1: portTickType = uint_16; 0: portTickType = uint_32 Improve performance of the system, but Impact the maximum time a task can be delayed
configIDLE_SHOULD_YIELD	The users application creates tasks that run at the idle priority
configUSE_MUTEXES	Build/omit Mutex support functions
configQUEUE_REGISTRY_SIZE	Defines the maximum number of queues and semaphores that can be registered
configCHECK_FOR_STACK_OVERFLOW	Enables stack over flow detection
configUSE_RECURSIVE_MUTEXES	Build/omit Recursive Mutex support functions
configUSE_MALLOC_FAILED_HOOK	Build/omit Malloc failed support functions
configUSE_APPLICATION_TASK_TAG	Build/omit Task tag functions
configUSE_COUNTING_SEMAPHORES	Build/omit counting semaphore support functions
configUSE_CO_ROUTINES	Build/omit co-routines support functions
configMAX_CO_ROUTINE_PRIORITIES	Defines the maximum level of priority for coroutines
configUSE_TIMERS	Build/omit timers support functions
configTIMER_TASK_PRIORITY	Defines timer task priority level
configTIMER_QUEUE_LENGTH	Sets the length of the software timer command queue
configTIMER_TASK_STACK_DEPTH	Sets the stack depth allocated to the software timer service/daemon task

Semáforo Contador

Prof° Fernando Simplicio

Semáforo Contador

- **configUSE_COUNTING_SEMAPHORES = 1**

```c
void vATask( void * pvParameters )
{
SemaphoreHandle_t xSemaphore;

    /* Create a counting semaphore that has a maximum count of 10 and an
    initial count of 0. */
    xSemaphore = xSemaphoreCreateCounting( 10, 0 );

    if( xSemaphore != NULL )
    {
        /* The semaphore was created successfully. */
    }
}
```

API FreeRTOS

Prof° Fernando Simplicio

API FreeRTOS (Task)

[Task Creation](#)

- TaskHandle_t (type)
- xTaskCreate()
- xTaskCreateStatic()
- vTaskDelete()

Task Control

- vTaskDelay()
- vTaskDelayUntil()
- uxTaskPriorityGet()
- vTaskPrioritySet()
- vTaskSuspend()
- vTaskResume()
- xTaskResumeFromISR()
- xTaskAbortDelay()

Task Utilities

- uxTaskGetSystemState()
- vTaskGetInfo()
- xTaskGetApplicationTaskTag()
- xTaskGetCurrentTaskHandle()
- xTaskGetHandle()
- xTaskGetIdleTaskHandle()
- uxTaskGetStackHighWaterMark()
- eTaskGetState()
- pcTaskGetName()
- xTaskGetTickCount()
- xTaskGetTickCountFromISR()
- xTaskGetSchedulerState()
- uxTaskGetNumberOfTasks()
- vTaskList()
- vTaskStartTrace()
- ulTaskEndTrace()
- vTaskGetRunTimeStats()
- vTaskSetApplicationTaskTag()
- xTaskCallApplicationTaskHook()
- 'SetThreadLocalStoragePointer()
- 'GetThreadLocalStoragePointer()
- vTaskSetTimeOutState()
- xTaskGetCheckForTimeOut()

API FreeRTOS (Task)

Direct To Task Notifications

- xTaskNotifyGive()
- vTaskNotifyGiveFromISR()
- ulTaskNotifyTake()
- xTaskNotify()
- xTaskNotifyAndQuery()
- xTaskNotifyAndQueryFromISR()
- xTaskNotifyFromISR()
- xTaskNotifyWait()
- xTaskNotifyStateClear()

API FreeRTOS (Filas)

Queues

- xQueueCreate()
- xQueueCreateStatic()
- vQueueDelete()
- xQueueSend()
- xQueueSendFromISR()
- xQueueSendToBack()
- xQueueSendToBackFromISR()
- xQueueSendToFront()
- xQueueSendToFrontFromISR()
- xQueueReceive()
- xQueueReceiveFromISR()
- uxQueueMessagesWaiting()
- uxQueueMessagesWaitingFromISR()
- uxQueueSpacesAvailable()
- xQueueReset()
- xQueueOverwrite()
- xQueueOverwriteFromISR()
- xQueuePeek()
- xQueuePeekFromISR()
- vQueueAddToRegistry()
- vQueueUnregisterQueue()
- pcQueueGetName()
- xQueueIsQueueFullFromISR()
- xQueueIsQueueEmptyFromISR()

Queue Sets

- xQueueCreateSet()
- xQueueAddToSet()
- xQueueRemoveFromSet()
- xQueueSelectFromSet()
- xQueueSelectFromSetFromISR()

API FreeRTOS (Semáforos)

Semaphore / Mutexes

- xSemaphoreCreateBinary()
- xSemaphoreCreateBinaryStatic()
- vSemaphoreCreateBinary()
- xSemaphoreCreateCounting()
- xSemaphoreCreateCountingStatic()
- xSemaphoreCreateMutex()
- xSemaphoreCreateMutexStatic()
- xSem'CreateRecursiveMutex()
- xSem'CreateRecursiveMutexStatic()
- vSemaphoreDelete()
- xSemaphoreGetMutexHolder()
- uxSemaphoreGetCount()
- xSemaphoreTake()
- xSemaphoreTakeFromISR()
- xSemaphoreTakeRecursive()
- xSemaphoreGive()
- xSemaphoreGiveRecursive()
- xSemaphoreGiveFromISR()

API FreeRTOS (*Software Timers*)

Software Timers

- xTimerCreate()
- xTimerCreateStatic()
- xTimerIsTimerActive()
- xTimerStart()
- xTimerStop()
- xTimerChangePeriod()
- xTimerDelete()
- xTimerReset()
- xTimerStartFromISR()
- xTimerStopFromISR()
- xTimerChangePeriodFromISR()
- xTimerResetFromISR()
- pvTimerGetTimerID()
- vTimerSetTimerID()
- xTimerGetTimerDaemonTaskHandle()
- xTimerPendFunctionCall()
- xTimerPendFunctionCallFromISR()
- pcTimerGetName()
- xTimerGetPeriod()
- xTimerGetExpiryTime()

API FreeRTOS (*Flags*)

Event Groups (or 'flags')

- xEventGroupCreate()
- xEventGroupCreateStatic()
- vEventGroupDelete()
- xEventGroupWaitBits()
- xEventGroupSetBits()
- xEventGroupSetBitsFromISR()
- xEventGroupClearBits()
- xEventGroupClearBitsFromISR()
- xEventGroupGetBits()
- xEventGroupGetBitsFromISR()
- xEventGroupSync()

API FreeRTOS (MPU)

FreeRTOS-MPU Specific

- xTaskCreateRestricted()
- vTaskAllocateMPURegions()
- 'SWITCH_TO_USER_MODE()